MODEL-BASED VISION NAVIGATION FOR A FREE-FLYING ROBOT

by

ALI J. AZARBAYEJANI

Scientiae Baccalaureus in Aeronautics and Astronautics,
Massachusetts Institute of Technology
(1988)

Submitted in Partial Fulfillment of the
Requirements for the Degrees of

SCIENTIAE MAGISTER IN AERONAUTICS AND ASTRONAUTICS

and

SCIENTIAE MAGISTER IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1991

Author [illegible]
Department of Aeronautics and Astronautics
Department of Electrical Engineering and Computer Science
August 1991

Certified by [illegible]
Professor Harold L. Alexander
Thesis Supervisor, Department of Aeronautics and Astronautics

Certified by [illegible]
Professor Tomás Lozano-Pérez
Thesis Supervisor, Department of Electrical Engineering and Computer Science

Accepted by
Professor Harold Y. Wachman
Chairman, Department Graduate Committee
Department of Aeronautics and Astronautics

Accepted by [illegible]
Professor Campbell L. Searle
Chair, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

"Ve-ri-tas"

- Motto, Harvard University

"One can have three principal objects in the study of truth: the first, to discover it when one searches for it; the second, to prove it when one possesses it ; the last one, to distinguish it from falsity when one examines it."

- Blaise Pascal

"The pursuit of truth and beauty is a sphere of activity in which you are permitted to remain a child all of your life"

- Albert Einstein

"There are several 'macroscopic truths' which we state without proof prior to engaging in geometrical/mathematical development of the subject matter. Four 'truths' are as follows:..."

- J.J. and J.T.

"It is clear that..."

- A.A.

This page not used.

Model-Based Vision Navigation for a Free-Flying Robot

by

Ali J. Azarbayejani

Submitted to the Department of Aeronautics and Astronautics
and to the Department of Electrical Engineering and Computer Science
on 31 July 1991 in partial fulfillment of the requirements
for the Degree of Scientiae Magister in Aeronautics and Astronautics
and the Degree of Scientiae Magister in Electrical Engineering and Computer Science

Abstract

This study describes a vision technique applicable to the navigation of free-flying space robots. The technique is based on the recursive estimation framework of the extended Kalman filter. It requires simple measurements to be taken from images of a known object in the environment. Models of the object, the vision sensor, and the robot dynamics are used, along with the measurements, to recursively compute optimal estimates of the robot position, orientation, velocity, and angular velocity with respect to the object.

Unlike many artificial vision algorithms, this one is fundamentally dynamic, operating on a perpetual sequence of images rather than a single image. Information from previous frames is used to increase the efficiency of analyzing the current frame and the accuracy of the state estimate produced from measurements taken on that frame. The technique is computationally efficient so that it can be implemented in real-time on cheap existing hardware. The technique is also flexible because the routines for parsing images are independent of the underlying recursive estimator. The technique is extensible because new models and other sensor measurements can be incorporated.

Experimental results are presented to demonstrate the accuracy of the vision navigator under mild and extreme conditions of uncertainty using computer simulated robot trajectories and image measurements. Additional experimental results illustrate the navigator performance on a real trajectory with measurements taken by digital image processing of video imagery of a navigation target.

Thesis supervisor:	Harold L. Alexander
Title:	Bradley Career Development
	Assistant Professor of Aeronautics and Astronautics

Thesis supervisor:	Tomás Lozano-Pérez
Title:	Associate Professor of Computer Science and Engineering

This page not used.

Acknowledgements.

It's done. Even though this page is near the beginning of the document, it is actually the last page written after seven long and arduous months of writing, revising, and reorganizing the thesis in whole or in part, quite often in whole. Fortunately, normal grammar rules and writing style do not apply on acknowledgement pages, so I can say anything I want here in any way I want. Let's have some fun...

First, I would like to thank Mom & Dad, to whom this thesis is dedicated, for providing me with all that I've ever needed, particularly a good education, never-ending love and support, and occasional money and chocolate chip cookies.⇒

More directly related to this thesis, I would like to thank Professor Dave Akin for hiring me originally and letting me return after immediately taking a year off to teach. The SSL, and its first child LSTAR, exist largely because of Dave and will remain exciting research environments for lots of time to come. Good luck Dave, Russ, Beth and others at UM. ⇒

Thanks, of course, are most deserved by my primary advisor Professor Sandy Alexander who allowed me from the start to pursue this crazy project, which looks like it will end in success. Together we struggled with contracts, hardware, software, and about ten or twenty revisions of what has finally become a THESIS and hopefully soon will become an operational navigation system. LSTAR is a most excellent facility largely because of Sandy's personal touch and commitment to quality work. It will be a long time before I see so much of a robot built with so much lack of money. Onward to total autonomy (of the vehicle, that is).
⇐

Another reason for the most excellence of the LSTAR organization is of course because of the boyz (and girlz) in the lab, gradual students and UROPers alike. Thanks be to all those who have made LSTAR such an exciting and enjoyable place to work the last couple years...to Kurt, who designed and built a robot that they said could not possibly be built in two years—the most excellent STARFISH behaves so well, I actually think navigation is going to be eeasy...to Harald, who I now hand the vision baton to (remember, it's going to be easy)...to Matt, who, when he reads this, will correct the previous clause ("to whom I now hand...")...to Michael, who has been named the MVP for the last NBA season...to Michael, who still owes us all a round because his pal's team won the NBA championship...and to Moonbeam and Starlight and all the others who have worked so hard for so little pay and who cannot join us for the round because they're too young. ⇓

Thanks to Professor TLP for taking the time to be my second thesis advisor—his excellent advice, particularly to try the Kalman Filter, obviously had a large positive effect on the final results of this project. Thanks to Professors Horn, Pentland, and Adelson, for whose classes I generated the initial work that ended up being the foundation of this thesis. ⇓

Finally, special thanks to Jon Russell, Keith Briggs, and the Advanced Programs Group at Ford Aerospace Corporation for their generous support which partially funded the LSTAR vision navigation research. ♦

This page not used.

Table of Contents.

Nomenclature.

Conventions.

y	plainface symbols are scalars;
$\mathbf{x}$	boldface lowercase symbols are vectors (of any dimension) or 3D points;
$\mathbf{H}$	boldface uppercase symbols are matrices;
$\mathbf{x}_{nav}$, $\mathbf{p}_i$	normal subscripts serve to describe or index quantities;
$\mathbf{p}_{:f}$, $\mathbf{p}_{i:f}$	a colon subscript is a frame reference (point $\mathbf{p}$ in the "f" frame, $\mathbf{p}_i$ in the "f" frame);
$\mathbf{H}^{(k)}$, $\mathbf{H}_k$	"k" indexes a quantity to a particular time step, $t_o + k\,\Delta t$;
$\mathring{\mathbf{q}}$	boldface letters with hollow dots above are quaternions;
$\underline{\mathring{\mathbf{x}}}$	underlined quaternions are vectors, i.e. have no scalar part;
$\mathbf{t}_{bc}$, $\mathbf{t}_{fb}$	translation vectors between origins of frames (frame "b" to "c", frame "f" to "b");
${}^{f}_{b}R$	rotation operator, from b-frame to f-frame; for vector $\mathbf{x}$, $\mathbf{x}_{:f} = {}^{f}_{b}R[\mathbf{x}_{:b}]$;
$\mathring{\mathbf{q}}_{bc}$, $\mathring{\mathbf{q}}_{fb}$	rotation unit quaternions ($\mathring{\mathbf{q}}_{bc} \Leftrightarrow {}^{b}_{c}R$, $\mathring{\mathbf{q}}_{fb} \Leftrightarrow {}^{f}_{b}R$);
$\mathcal{Q}_{bc}$	script matrix - quaternion premultiplier matrix for $\mathring{\mathbf{q}}_{bc}$, defined $\mathring{\mathbf{q}}_{bc}\mathring{\mathbf{r}} = \mathcal{Q}_{bc}\mathring{\mathbf{r}}$;
$\bar{\mathcal{Q}}_{fb}$	script with bar - postmultiplier matrix for $\mathring{\mathbf{q}}_{fb}$, defined $\mathring{\mathbf{r}}\mathring{\mathbf{q}}_{fb} = \bar{\mathcal{Q}}_{fb}\mathring{\mathbf{r}}$;
$\mathbf{R}_{bc}$, $\mathbf{R}_{fb}$	orthonormal rotation matrices ($\mathbf{R}_{bc} \Leftrightarrow {}^{b}_{c}R$, ...)
$\dot{\mathbf{x}}$, $\dot{y}$	solid dot above a vector or scalar means time derivative;
$\tilde{\mathbf{x}}$, $\tilde{\mathbf{P}}_{k+1}$	a tilde above designates a predicted quantity (vector or scalar);
$\hat{\mathbf{x}}$, $\hat{\mathbf{P}}_k$	a caret above designates an estimated quantity (vector or scalar);
[$\hat{\mathbf{n}}$	in isolated places, carets denote unit vectors; context should be clear;]

List of symbols.

$\mathbf{t}_{fb}$	translation vector (3D) - position of robot w.r.t. fixed frame;
$\mathring{\mathbf{q}}_{fb}$	rotation unit quaternion - attitude of robot w.r.t. fixed frame;
$\mathbf{v}_{fb}$	translational velocity vector (3D) - $\partial \mathbf{t}_{fb}/\partial t$ or $\dot{\mathbf{t}}_{fb}$;
ω_{fb}	angular velocity vector (3D) - angular velocity of robot about body axes;
$\mathbf{x}_{nav}$	navigation state vector (13D) - $\begin{pmatrix} \mathbf{t}_{fb:f} \\ \mathring{\mathbf{q}}_{fb} \\ \mathbf{v}_{fb:f} \\ \omega_{fb:b} \end{pmatrix}$;
$\boldsymbol{\Theta}$	vector of small Euler angles (3D);
$\mathbf{x}_{aux}$	auxilliary state vector (12D) - $\begin{pmatrix} \mathbf{t}_{fb:f} \\ \boldsymbol{\Theta} \\ \mathbf{v}_{fb:f} \\ \omega_{fb:b} \end{pmatrix}$;
$\mathcal{F}$	force vector (3D) - applied forces on robot in fixed coordinates;
$\mathcal{T}$	torque vector (3D) - applied torques on robot in body coordinates;
$\mathbf{u}$	command vector (6D) - $\begin{pmatrix} \mathcal{F} \\ \tau \end{pmatrix}$;
N	number of point correspondences used in the vision processing;
$\mathbf{p}_i$	feature point (3D), indexed by i = 1…N;
$\mathbf{y}_i$	feature point image coordinates (2D), indexed by i = 1…N;
$\mathbf{y}$	measurement vector (2N-D) - image coordinates of N feature points $\begin{pmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_N \end{pmatrix}$;
$\mathbf{t}_{bc}$	translation vector (3D) - position of camera w.r.t. body frame;
$\mathring{\mathbf{q}}_{bc}$	rotation unit quaternion - attitude of camera w.r.t. body frame;

$\mathbf{x}_{cam}$	camera state vector (13D) - $\begin{pmatrix}\mathbf{t}_{bc}\\ \mathring{\mathbf{q}}_{bc}\end{pmatrix}$;
$\mathring{\mathbf{q}}_{fc}$	rotation unit quaternion - attitude of camera w.r.t. fixed frame. $\mathring{\mathbf{q}}_{fc} = \mathring{\mathbf{q}}_{fb}\mathring{\mathbf{q}}_{bc}$;
$\mathbf{K}$	Kalman gain matrix;
$\mathbf{P}$	state error covariance matrix;
$\mathbf{H}$	measurement matrix;
$\mathbf{Q}$	dynamics noise covariance matrix;
$\mathbf{R}$	measurement noise covariance matrix;
f	Effective focal length (principal distance) of the imager.

$\mathbf{T} : \mathbf{x}_{nav}(t), \mathbf{x}_{cam}(t), \mathbf{p}_{:f} \rightarrow \mathbf{p}_{:c}$

Frame Transformation Relation, as a function of a feature point.

$\mathbf{T}_i : \mathbf{x}_{nav}(t) \rightarrow \mathbf{p}_{i:c}$

Frame Transformation for ith feature point, as function of the state.

$\mathbf{C} : \mathbf{p}_{i:c} \rightarrow \mathbf{y}_i$

Image projection relation.

$\mathbf{h}_i = \mathbf{C}\mathbf{T}_i$

Measurement relation for the ith point, $\mathbf{h}_i = \mathbf{c}\mathbf{T}_i$.

$\mathbf{h} = \begin{pmatrix}\mathbf{h}_1\\ \vdots\\ \mathbf{h}_N\end{pmatrix}$ Measurement relation.

$\mathbf{f} : \mathbf{x}_{nav}(t), \mathbf{u}(t) \rightarrow \dot{\mathbf{x}}_{nav}(t)$

Dynamics relation.

$\mathbf{f}_{aux} : \mathbf{x}_{aux}(t), \mathbf{u}(t) \rightarrow \dot{\mathbf{x}}_{aux}(t)$

Auxilliary dynamics equation, valid for small rotational states.

This page not used.

1. Introduction.

As the field of robotics matures, free-flying robots are increasingly studied for space and underwater operations. Both autonomous free-flying robots and human-controlled teleoperated robots have been proposed for a variety of tasks, from ocean exploration to space construction. For most of the applications, it is necessary for the robot to interact with other objects in the environment. Thus, it is crucial to have accurate information about the position and orientation of the robot with respect to objects in the vicinity, whether that information is passed on to a human operator or to an automatic control system.

This type of object-relative navigation for free-flying robots is the subject of this study. It is proposed that an artificial vision system, consisting of a video camera mounted on the robot and on-board digital computational hardware, can be used to perform such navigation efficiently and accurately. The technique applies to a general navigation environment for a free-flying robot, with the only stipulation being that there exists, visible to the camera on the robot, at least one "navigation target" whose physical relationship to the external reference frame is known and is available to the vision navigation system.

The robot vision navigator presented herein can perform this task in real time using current technology. Consequently, this vision-based navigator can be presently built, tested, and perfected in a relatively inexpensive submersible environment, developing it into a proven technology available for use in space on a time scale consistent with the planned launching of the first free-flying space robots.

1.1 Background on Free-flying Robot Navigation.

For over a decade, researchers at NASA, at the MIT Space Systems Laboratory (SSL), and elsewhere have actively experimented with free-flying robots, particularly with regards to their application to assembly, servicing, and repair tasks in space. Although no such robots have yet been in earth orbit, free-flying submersible robots have been used under the sea for exploration and in neutral buoyancy tanks for simulating future space robots. These include the JASON robot of the Woods Hole Oceanographic Institute (WHOI), the BAT and MPOD robots of the MIT SSL, and the recently developed STARFISH submersible of the MIT Laboratory for Space Teleoperation and Robotics (LSTAR).[1]

Attempts to navigate these submersible free-flying robots, i.e. to determine their position and orientation, have utilized various combinations of sensors. These include water pressure sensors (for depth), inertial sensors (for local vertical and angular rates), side-scanning sonar (for nearby obstacles), and acoustic triangulation networks (for absolute position and orientation with respect to an external reference). One immediate drawback of all of these sensors, excepting inertial angular rate sensing, is that they are specific to the submersible environment and not applicable to space robots.

Proposed instruments for space robot navigation include radar, laser range finders, inertial sensing, and fixed radio beacons. Radar or laser range finders can perform functions analogous to sonar. Inertial angular rate sensing is still applicable and inertial accelerations can still be used, but due to the zero-gravity environment in orbit, they would be used for measuring translational accelerations rather than for measuring the gravity vector. Finally, radio beacons might perform a function analogous to acoustic positioning, but they would provide a number of directions to known locations rather than distances from known locations.

The underwater instruments are not directly transferable to space, and, conversely, the instruments most commonly suggested for space navigation are not applicable for submersibles. This is a legitimate concern because the most effective testbeds for free-flying space robots to date have been submersible robots in neutral buoyancy tanks. It would be prudent to implement and verify the functionality of space navigation systems on relatively inexpensive submersible robots before launching them into space, especially when they function as part of automatic control systems.

In contrast to most other navigation sensors, optical sensing applies both underwater and in space. Digitally-processed machine vision navigation is particularly attractive because it is potentially highly flexible with respect to the kinds of environments it can handle. However, vision has rarely been seriously considered for navigation because existing vision techniques are either too computationally expensive for real-time operation or too inflexible. A successful approach to vision navigation for free-flying robots must apply to a wide range of navigation environments at a computational cost commensurate with the computing hardware likely to be available on relatively small vehicles.

[1]JASON was used to explore the sunken *Titanic* deep under the Atlantic Ocean. BAT, MPOD, and STARFISH are intended to simulate space telerobots in neutral buoyancy tanks. BAT is the Beam Assembly Teleoperator which has a manipulator and is controlled remotely by a human operator. MPOD is the Multimode Proximity Operations Device and can be operated by an (underwater) astronaut onboard or by a remote human operator. (BAT, MPOD, and SSL are now at University of Maryland, College Park.) STARFISH is the neutral buoyancy teleoperator research platform of LSTAR at MIT.

1.2 Overview of the Robot Vision Navigator.

Both computational efficiency and flexibility are addressed by the vision navigator developed herein. The goals of this study include not only developing a machine vision technique applicable to the general problem of close-range navigation of space and submersible free-flying robots, but also ensuring that the technique can be implemented well within current technological limits because the technique is intended for immediate operation on free-flying submersibles.

The role of the navigator in the motion control loop for a free-flying robot is illustrated in Figure 1.2-1. The output being controlled is the robot state, the static and dynamic spatial relationship between the robot and its environment. The term "environment" refers to the external object-fixed frame of reference. The navigator is the feedback element which provides measurements of the robot state to the controller to facilitate closed-loop control of the robot.

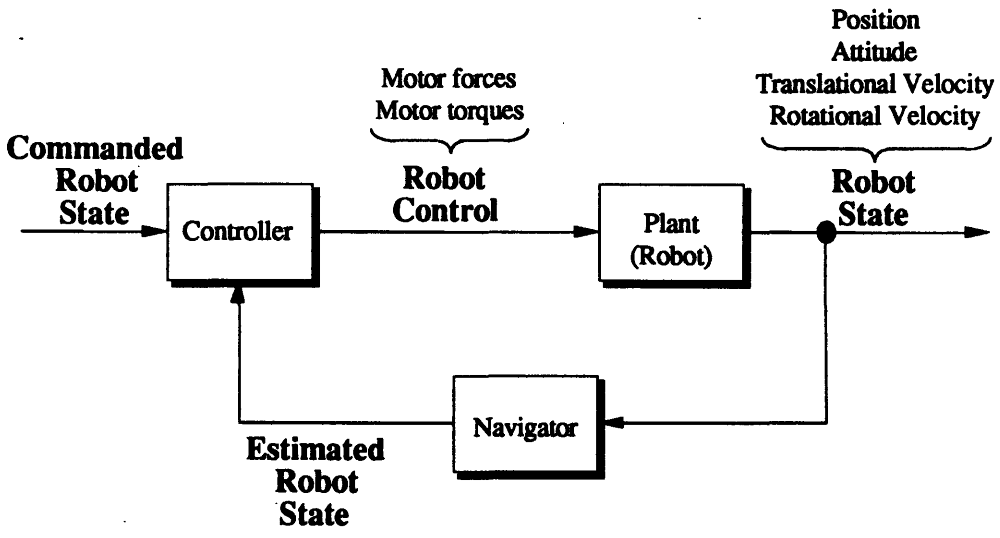


Figure 1.2-1: The role of the navigator in the control loop.

Traditional navigators for many vehicles consist of instruments which directly measure the output quantities of interest. This is very difficult for free-flying robots because they have no physical contact with references in the environment. Rotational velocities (with respect to an inertial frame) can be directly measured using rate gyros, but measurement of the other robot state quantities requires some sort of communication with the surroundings.

A vision navigator accomplishes this by receiving images of the environment through the navigation camera on the robot. This permits the robot to "see" its surroundings and deduce its spatial relationship to the environment according to the way the features of the environment appear in the view. The process mimics the way humans deduce by visual observation their spatial relationship to objects.

However, a vision navigator need not, and with current technology cannot, interpret general scenes in real time as humans do. Fortunately, simplifications are natural for the vision navigation problem because most free-flying space robots and many submersible robots operate in known, structured environments. This observation implies that physical models of objects in the environment can be made available to the navigation system. Relevant physical models contain spatial information regarding the three-dimensional (3D) structure of the environment, e.g. locations of points on the objects relative to the environment-fixed reference frame. An extensive physical model of an entire structure may be available, but it will be shown that only a simple model describing the 3D locations of a few points is required.

Such models allow important information to be gained from simply identifying known points as they appear in images. For each individual point, the corresponding image location depends upon the spatial relationship of the the robot to the environment, the relationship of the camera to the robot, and the projection geometry of the camera. The latter two pieces of information can be assumed known for a calibrated vision system. Thus, for a set of known 3D points, the corresponding image locations are determined solely by the vehicle state and can therefore be used to estimate the vehicle state. In fact, deducing robot state from the set of point correspondences from a single image is a simple extension of the machine vision problem called *exterior orientation*, the problem of finding camera orientation relative to the external world [HORN86].

However, since the vision navigation problem is dynamic and operates on a sequence of highly correlated images rather than a series of independent single images, it is beneficial to use a dynamic estimation technique capable of exploiting the evolving nature of the incoming data rather than using traditional single-image machine vision techniques. Such a dynamic estimation technique, namely the Kalman filter, is commonplace in the field of navigation for processing sensor measurements which are taken periodically at discrete time intervals. The Kalman filter uses a model of the measurement process to update the state estimate each time a measurement arrives and uses a model of the vehicle dynamics to propagate the vehicle state between arrivals of measurements.

It has been argued above that, for a set of 3D points in the environment, the set of corresponding image locations depends solely on the state of the robot, and can therefore be treated as a set of measurements of the robot state. With this interpretation, a sequence of images can produce a sequence of image measurements which can be used in the Kalman filter framework for dynamically estimating state. Hence, vision measurements can be used in the same way that traditional navigation sensor measurements, such as gyro readings or range measurements, are normally used for estimating vehicle state.

Model-based vision navigation founded on this concept requires three fundamental models and two primary functional components. The models include a 3D spatial model of a navigation target in the environment, a geometric model of the imaging system, and a dynamic model of the robot vehicle dynamics. The functional components include an image sequence processing subsystem which locates and tracks known points in images, and a recursive estimation subsystem which maintains an estimate of the vehicle state, incorporating image measurements when they arrive.

Figure 1.2-2 illustrates the functional architecture for such a vision navigator. An additional feature of this approach, illustrated in the figure, is that the recursive estimator can provide state predictions to the image sequence processor which can be used to enhance the efficiency of searches for points in the images.

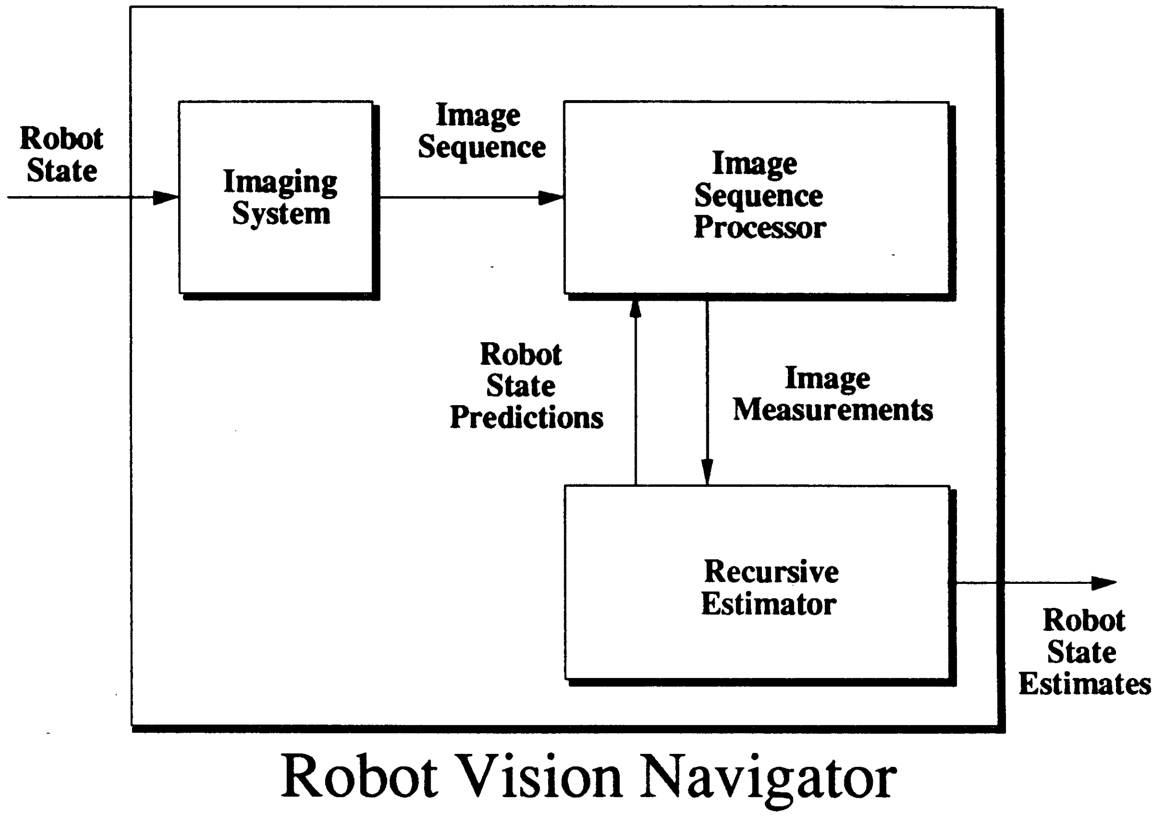


Figure 1.2-2: Basic architecture of the robot vision navigator based on a Kalman filter recursive estimator and model-based image processing.

This thesis presents the design and experimental evaluation of this type of vision navigator, demonstrating its potential for great computational efficiency and flexibility. The recursive estimator is developed first; its performance is evaluated by passing it simulated measurements that are generated by applying a noise-corrupted measurement model to a simulated actual trajectory of the robot. Various aspects of the estimator performance are evaluated by corrupting the measurements, the vehicle dynamic model, and the initial state prediction with various levels of errors. The image sequence processor is discussed afterwards; further experimental data results from applying the estimation algorithm to actual measurements taken from real imagery by the image sequence processor. A analysis of the applicability to real free-flying robots and related problems concludes the study.

1.3 Reader's Guide.

Robot vision navigation requires concepts from control theory, rigid body kinematics and dynamics, optical sensing, and optimal estimation. **Chapter 2** is devoted to summarizing the required background concepts and developing a consistent notation which can be consistently applied throughout the document. Specifically, the Extended Kalman filter (EKF) algorithm is summarized and rigid body mechanics and dynamics are reviewed to the extent that they apply to robot vision navigation. In the process of discussing these concepts, a consistent notation is developed for describing spatial relationships and the quantities used in the EKF algorithm. The notation can be referenced in the **Nomenclature** section at the beginning of the document.

Chapters 3 and **4** describe the estimation portion of the vision navigation system. Chapter 3 develops the measurement and dynamic models required for implementing an EKF estimator for navigating a robot with vision measurements. The algorithm as it used for robot vision navigation is detailed. Chapter 4 describes the computer implementation of the estimator and presents the results of the simulation experiments.

Chapters 5 and **6** describe an image processing subsystem for obtaining the image measurements required by the EKF. Chapter 5 describes an experimental environment and image processing system. Chapter 6 presents the results of simulated trajectory estimations based on measurements from real imagery processed by the experimental system.

Chapter 7 concludes the study by discussing the implications of, and new directions for, the technology.

Appendix A describes the algebra of Hamilton's quaternion. The appendix is a short tutorial on the basic concepts leading to the use of unit quaternions in rigid body kinematics. The appendix contains information necessary for a reader to be comfortable with the mathematics appearing in Chapter 3.

Appendix B contains the source code for the machine-independent portion of the computer simulations. The code serves as documentation of the experimental process and also as a resource for developing operational systems.

2. Background.

Before commencing with the specifics of robot vision navigation, it is necessary to review the Kalman Filter recursive estimation algorithm as it applies to the general case and to review general concepts of 3D rigid body mechanics and dynamics.

2.1 The Extended Kalman Filter.

The Extended Kalman Filter (EKF) is based upon the Discrete Kalman Filter (DKF), an optimal recursive estimation algorithm which applies to discrete-time systems with linear measurement relations and linear state transition relations. The EKF extends the DKF to apply to systems with nonlinear measurements and dynamics by linearizing the system around the estimated state at each time step. This section motivates the EKF algorithm by first discussing the basic DKF algorithm and then describing the nonlinear extensions that lead to the EKF.

2.1.1 The Discrete Kalman Filter

The DKF algorithm is used for recursively estimating the state of a discrete-time dynamic system using state-dependent measurements which arrive at discrete time intervals. Each time step marking the arrival of a measurement, the DKF updates the state estimate by combining the measurement information with an internal prediction of the state. A dynamic model of the system is used to update the state between measurements, providing the prediction of state at each time step.

For performing the state estimation and prediction, the DKF requires a model describing the dynamic behavior of the system and a measurement model describing the relationship between state and measurements. The linear models are called the *state equation* and the *measurement equation* and are of the form,

$$
\begin{aligned}
x_{k+1} &= \Phi_k x_k + \xi_k && \text{(State Equation),} \\
y_k &= H_k x_k + \eta_k && \text{(Measurement equation),}
\end{aligned}
$$

where

$\mathbf{x}_k$	is a sample of the state vector at the k^{th} time step,
Φ_k	is the state transition matrix from the k^{th} to $(k+1)^{th}$ time steps,
ξ_k	is a sample of zero-mean white noise at the k^{th} time step, $\mathbf{Q}_k = E[\xi_k \xi_k^T]$,
$\mathbf{y}_k$	is a sample of the measurement vector at the k^{th} time step,
$\mathbf{H}_k$	is the linear measurement matrix at the k^{th} time step, and
η_k	is a sample of zero-mean white noise at the k^{th} time step, and $\mathbf{R}_k = E[\eta_k \eta_k^T]$.

Often included in the state equation is an additional term describing the effect of control inputs on the vehicle state. The input term is omitted in this discussion of the DKF because it is deterministic and does not affect the estimation. The input does affect the propagation of state and can be incorporated straightforwardly if it exists.

Figure 2.1-1 illustrates the basic functionality of the Kalman Filter algorithm. When measurements arrive, the estimator uses the measurements, the measurement model, and state and error covariance predictions from the last filter step to find a state estimate which minimizes expected square error of the state. The state estimator also computes the corresponding error covariance matrix. The predictor propagates the estimated state and covariance using the dynamic model, resulting in the required prediction of the state vector and covariance for the next time step.

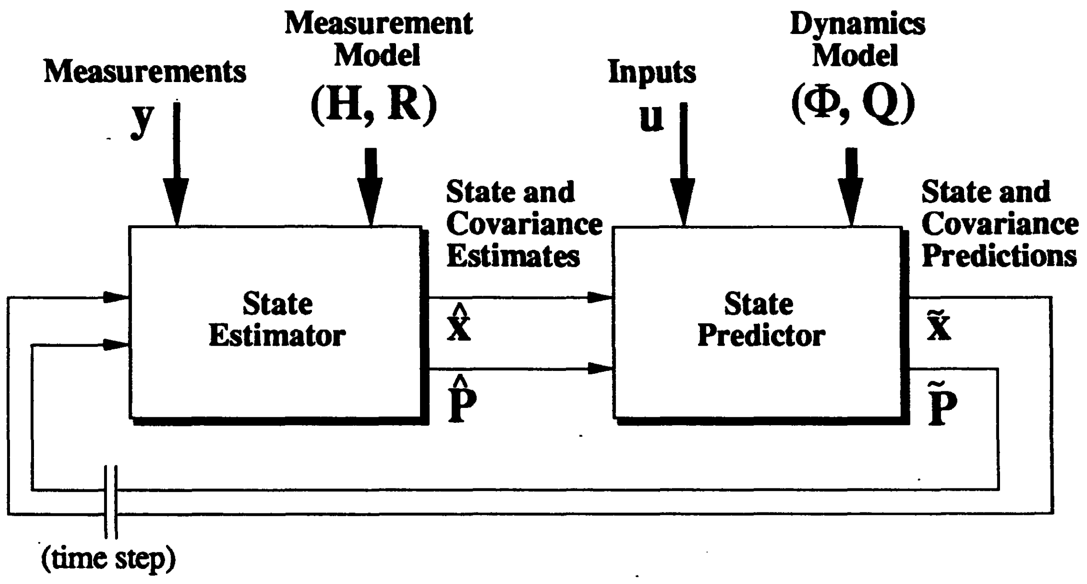


Figure 2.1-1: Kalman Filter loop.

The symbol $\hat{\mathbf{x}}$ represents an estimate of the state vector, $\tilde{\mathbf{x}}$ a state prediction. Notationally, this convention of using carets (^) for estimated values (output of the estimator) and tildes (~) for predictions (output of the predictor) shall hold for Kalman filter states and covariances.[1] Thus, the matrix $\hat{\mathbf{P}}$ represents the error covariance associated with the estimated state and $\tilde{\mathbf{P}}$ represents the error covariance associated with the predicted state, i.e.

$$\hat{\mathbf{P}} = E[(\mathbf{x} - \hat{\mathbf{x}})(\mathbf{x} - \hat{\mathbf{x}})^{\mathrm{T}}]$$

and

$$\tilde{\mathbf{P}} = E[(\mathbf{x} - \tilde{\mathbf{x}})(\mathbf{x} - \tilde{\mathbf{x}})^{\mathrm{T}}].$$

The error covariance of the state is computed after each estimation step and each prediction step. It effectively accumulates information about the estimation accuracy over all previous time steps.

In the estimator, the optimal estimate of state is computed via the state estimation equation,

$$\hat{\mathbf{x}}_{\mathrm{k}} = \tilde{\mathbf{x}}_{\mathrm{k}} + \mathbf{K}_{\mathrm{k}}(\mathbf{y}_{\mathrm{k}} - \mathbf{H}_{\mathrm{k}}\tilde{\mathbf{x}}_{\mathrm{k}}),$$

where the *Kalman gain matrix*, **K**, for the current time step is computed

$$\mathbf{K}_{\mathrm{k}} = \tilde{\mathbf{P}}_{\mathrm{k}}\mathbf{H}_{\mathrm{k}}^{\mathrm{T}}\left(\mathbf{H}_{\mathrm{k}}\tilde{\mathbf{P}}_{\mathrm{k}}\mathbf{H}_{\mathrm{k}}^{\mathrm{T}} + \mathbf{R}_{\mathrm{k}}\right)^{-1}.$$

The Kalman gain matrix blends state prediction with the measurement information to produce an optimal state estimate. The reader is referred to [BROWN83] for a derivation of the Kalman gain equation and the DKF equations which follow.

The error covariance for the estimated state is computed using the gain matrix **K** in conjunction with the measurement model (**H**,**R**):

$$\hat{\mathbf{P}}_{\mathrm{k}} = (\mathbf{I} - \mathbf{K}_{\mathrm{k}}\mathbf{H}_{\mathrm{k}})\,\tilde{\mathbf{P}}_{\mathrm{k}}.$$

It is worthy to note that the value of (**I** – **KH**) at each time step has a norm between zero and unity.[2] Hence, the norm of the state error covariance always decreases when a measurement is taken. Thus the DKF always perceives an improvement in its current estimate each time a measurement is taken, even when the quality of the measurement is extremely poor.

[1]Carets are also used in a few places to denote unit vectors. The distinction should be clear from context.
[2]Although a significant amount of knowledge of linear algebra and normed spaces is required to make this argument for the general matrix case, the argument can be simply followed by considering the 1D case in which all matrices are scalars and the norm is absolute value.

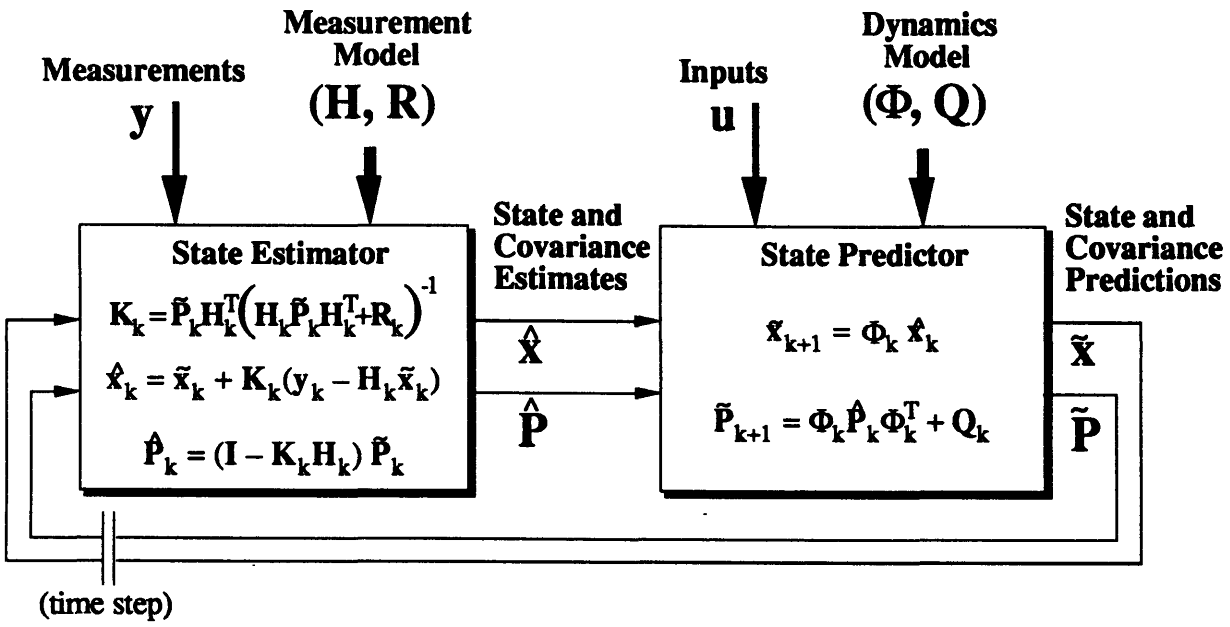
The prediction portion of the DKF consists of propagating the state and covariance to the next time step using the dynamics model, (**Φ**,**Q**). The equations are

$$\tilde{x}_{k+1} = \Phi_k \hat{x}_k$$

$$\tilde{P}_{k+1} = \Phi_k \hat{P}_k \Phi_k^T + Q_k.$$

The state prediction equation above is the appropriate place for incorporating known inputs, if any.

Figure 2.1-2 summarizes the computational steps required for the DKF algorithm. The estimator uses the measurements, the measurement model, and predictions from the previous time step to generate estimates for the current time step. The predictor uses these estimates and the dynamics model, with the known inputs **u**, if any, to generate predictions for the next time step.



Measurements
y
Measurement Model
(H, R)
Inputs
u
Dynamics Model
(Φ, Q)
State Estimator
$K_k = \tilde{P}_k H_k^T (H_k \tilde{P}_k H_k^T + R_k)^{-1}$
$\hat{x}_k = \tilde{x}_k + K_k(y_k - H_k \tilde{x}_k)$
$\hat{P}_k = (I - K_k H_k)\tilde{P}_k$
State and Covariance Estimates
$\hat{x}$
$\hat{P}$
State Predictor
$\tilde{x}_{k+1} = \Phi_k \hat{x}_k$
$\tilde{P}_{k+1} = \Phi_k \hat{P}_k \Phi_k^T + Q_k$
State and Covariance Predictions
$\tilde{x}$
$\tilde{P}$
(time step)

Figure 2.1-2: The Discrete Kalman Filter loop with constituent equations.

The reader is once again referred to the abundant literature on Kalman filtering for a mathematical proof of the optimality of the DKF. In lieu of reviewing well-documented mathematical derivations, some qualitative observations are offered here which may provide insight into the operation of the filter. First, note that for noiseless measurements, i.e. $\mathbf{R} = \mathbf{0}$, and a simple description of uncorrelated state error variances, $\tilde{P} = \alpha I$, the Kalman gain is

$$K = H^T(HH^T)^{-1},$$

a pseudo-inverse for the measurement matrix **H**. Hence, in its simplest form, without the weighting of **R** and $\tilde{P}$, **K** represents an inversion of the measurement operator.

Thus, the Kalman gain, computed with realistic values of $\mathbf{R}$ and $\tilde{\mathbf{P}}$, behaves roughly like a weighted inverse of $\mathbf{H}$. The measurement noise covariance matrix $\mathbf{R}$ describes the level of random noise present in the measurement equation, hence the quality or accuracy of the measurement model. Likewise, the error covariance matrix $\tilde{\mathbf{P}}$ describes the predicted accuracy of the current state prediction. The weighting characteristic of the gain matrix, then, reflects the a priori confidence level in the measurement model and the current level of error in the state prediction. The gain computation equation ensures that this weighting is optimal, producing a state estimate with the lowest aggregate error variance.

The functionality of the gain matrix as inverse to the measurement matrix and as optimal weighting appears in the state estimation equation:

$$\begin{aligned}(\text{state estimate}) &= (\text{state prediction}) + (\text{state correction})\\ &= \underset{\substack{\text{obtained from}\\ \text{internal dynamics}\\ \text{model}}}{(\text{state prediction})} + (\text{gain})\ \underset{\substack{\text{obtained from}\\ \text{sensor input}}}{(\text{measurement error})}\end{aligned}$$

The gain maps the measurement error to a state correction, thus performing a function which is qualitatively inverse to the function performed by $\mathbf{H}$. The weighting property of the gain matrix serves to optimally blend the state information maintained internally via the dynamic model (the state prediction) with the information gained from sensory input and the measurement model (the state correction). Given the form of the state estimation equation, a simple qualitative analysis of how the gain matrix weights the estimation is that "high" gains favor sensory input and "low" gains favor the internal propagation of state based on the dynamics model. For the gain matrix, "high" and "low" can be defined in terms of a matrix norm.

Using this insight, one should expect that the a priori confidence levels assigned to the dynamics and measurement models, manifested in the covariance matrices $\mathbf{Q}$ and $\mathbf{R}$, should affect the norm of the gain appropriately. Indeed, the Kalman gain equation demonstates that an increasingly high norm on $\mathbf{R}$, representing an increasingly uncertain measurement model, decreases the norm of $\mathbf{K}$, thus reducing the influence of measurements on the state estimate. Likewise, an increasingly poor dynamics model is characterized by an increasingly large norm on $\mathbf{Q}$, which increases the norm of $\tilde{\mathbf{P}}$ and $\mathbf{K}$, thus reducing the influence of the internal state prediction in favor of the measurements.

2.1.2 The Extended Kalman Filter

The EKF shares the form and function of the DKF, but utilizes models that are nonlinear and possibly continuous-time:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x},\mathbf{u},t) + \xi(t)$$

$$\mathbf{y}(t) = \mathbf{h}(\mathbf{x},t) + \eta(t),$$

where

$\mathbf{x}$	is a state vector (unknown - to be estimated),
$\mathbf{u}$	is a control input vector driving the dynamics (known),
$\mathbf{y}$	is the measurement vector (obtained from sensors),
$\mathbf{f}(\mathbf{x},\mathbf{u},t)$	is a nonlinear function on vehicle state $\mathbf{x}$ and control vector $\mathbf{u}$,
$\xi(t)$	is zero-mean Gaussian white noise driving the dynamics,
$\mathbf{h}(\mathbf{x},t)$	is a nonlinear function describing measurements $\mathbf{y}$ as a function of state,
$\eta(t)$	is zero-mean Gaussian white noise driving the measurements.

The idea behind the EKF is that if the current state prediction is reasonably accurate, a linearization of the dynamics and measurement models around the current estimated state is an accurate characterization of the system for sufficiently small perturbations of state. Under this condition, the DKF state estimation process, operating on the linearized system, produces a near-optimal state estimate. If, additionally, the time interval between measurements is sufficiently small, the prediction for the next time step will be close to the actual state, thus satisfying the condition for a successful linearization and state estimation at the next time step. The trajectory can be tracked in this manner.

Conversely, of course, a poor initial state prediction can lead to subsequently worse predictions and divergence of the estimated trajectory from the actual trajectory. Hence, conditions that are generally required for successful implementation of an EKF include that a reasonably accurate state prediction can be provided to initialize the filter and that the sampling interval is sufficiently small so that perturbations of the actual state from the predicted state remain sufficiently small at each time step.

The estimation process begins by expressing the true vehicle state as the current predicted state plus a perturbation:

$$\mathbf{x}_k = \tilde{\mathbf{x}}_k + \delta\mathbf{x}_k.$$

The measurement equation can then be expanded in a Taylor series about $\tilde{x}_k$. Assuming that the perturbation remains small, the measurement can be written as

$$y_k = \tilde{y}_k + \delta y_k \overset{\Delta}{=} h(\tilde{x}_k,t) + \delta y_k \approx h(\tilde{x}_k,t) + \left[\frac{\partial h}{\partial x}\right]_{\tilde{x}_k} \delta x(t) + \eta(t)$$

by truncating the Taylor series expansion.

Cancellation leads to

$$\delta y_k = H_k\, \delta x_k + \eta_k$$

where

$$H_k = \left[\frac{\partial h}{\partial x}\right]_{\tilde{x}_k}.$$

This linearized measurement equation can be used in the DKF framework for estimating the optimal state perturbation δx via the usual Kalman gain computation. Since the predicted values of δx and δy are zero, the state estimation equation is simply

$$\widehat{\delta x} = K\ \delta y.$$

The global state estimate results from combining the perturbation estimate with the nominal state as

$$\hat{x} = \tilde{x} + \delta\hat{x},$$

or, after some substitutions and re-inserting the time index,

$$\hat{x}_k = \tilde{x}_k + K_k\big(y_k - h(\tilde{x}_k,t)\big).$$

A simple prediction of the next state results from assuming constant velocity over the time interval:

$$\tilde{x}_{k+1} = \hat{x}_k + f(\hat{x}_k,u,t)\,\Delta t,$$

This formula avoids the state-space linearization of the operator f(). (Of course, this propagation equation still represents a linearization in time.) More sophisticated state propagation can be performed using a multi-step numerical integration procedure and/or using stored values of previous state estimates.

Propagation of the error covariance matrix is most conveniently performed using the state-space linearization of the state equation. As above, let δx represent a small deviation from the current estimated state, so that

$$\mathbf{x} = \hat{\mathbf{x}} + \delta\mathbf{x}.$$

Applying this to the nonlinear state equation above yields

$$\begin{aligned}\dot{\hat{\mathbf{x}}}(t) + \dot{\delta\mathbf{x}}(t) &= \mathbf{f}(\hat{\mathbf{x}}+\delta\mathbf{x},\mathbf{u},t) + \xi(t) \\ &\approx \mathbf{f}(\hat{\mathbf{x}},\mathbf{u},t) + \left[\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right]_{\hat{\mathbf{x}},\mathbf{u}} \delta\mathbf{x}(t) + \xi(t)\end{aligned}$$

which gives at the kth time step

$$\dot{\delta\mathbf{x}}(t) = \mathbf{F}_k\, \delta\mathbf{x}(t) + \xi(t)$$

with

$$\mathbf{F}_k = \left[\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right]_{\hat{\mathbf{x}}_k,\mathbf{u}_k}.$$

Forward Euler approximation of the derivative yields

$$\begin{aligned}\delta\mathbf{x}_{k+1} &\approx (\mathbf{I} + \mathbf{F}_k\, \Delta t)\, \delta\mathbf{x}_k + \xi(t)\, \Delta t \\ &= \Phi_k\, \delta\mathbf{x}_k + \xi_k,\end{aligned}$$

where the state transition matrix is computed

$$\Phi_k = (\mathbf{I} + \mathbf{F}_k\, \Delta t).$$

The covariance is propagated as usual using

$$\tilde{\mathbf{P}}_{k+1} = \Phi_k\, \hat{\mathbf{P}}_k\, \Phi_k^T + \mathbf{Q}_k,$$

where $\mathbf{Q}_k = E[\xi_k \xi_k^T]$.

Figure 2.1-3 illustrates the EKF and the computations required for the estimation and prediction steps. This recursive estimation framework is the basis for the vision navigation state estimator.

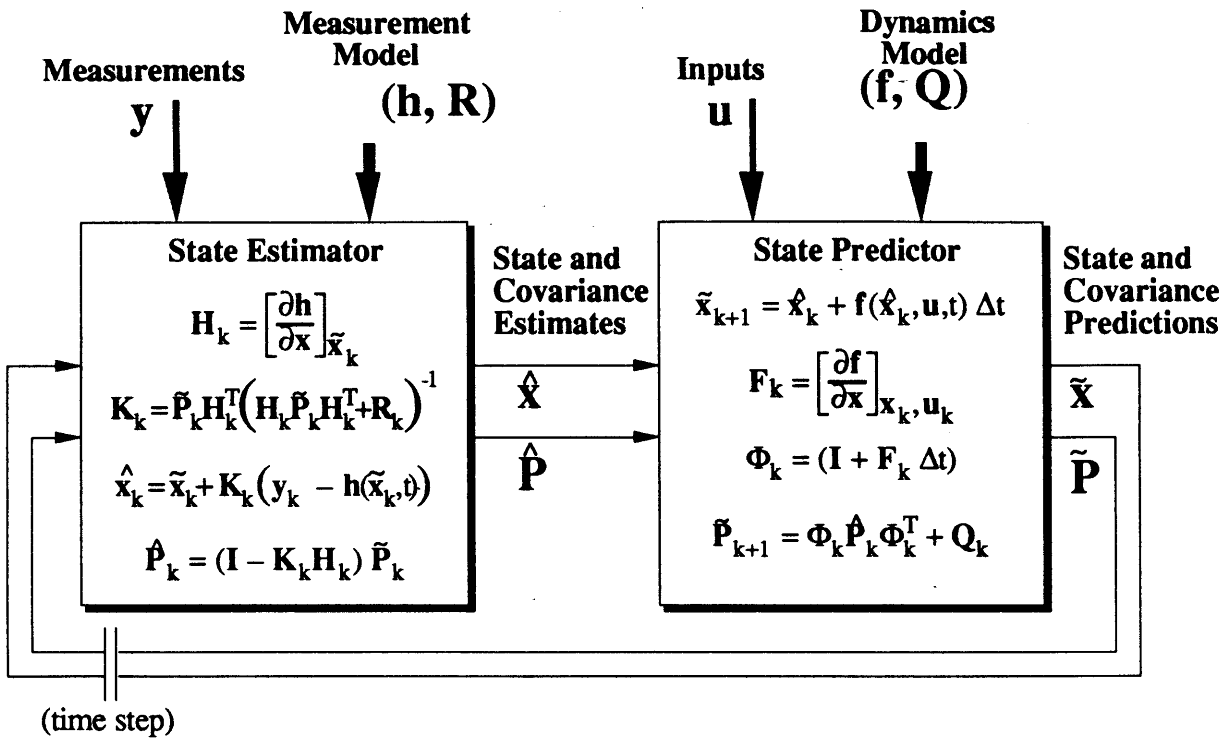

Measurements
y
Measurement Model
(h, R)
Inputs
u
Dynamics Model
(f, Q)
State Estimator
$H_k = \left[\frac{\partial h}{\partial x}\right]_{\tilde{x}_k}$
$K_k = \tilde{P}_k H_k^T (H_k \tilde{P}_k H_k^T + R_k)^{-1}$
$\hat{x}_k = \tilde{x}_k + K_k(y_k - h(\tilde{x}_k, t))$
$\hat{P}_k = (I - K_k H_k)\tilde{P}_k$
State and Covariance Estimates
$\hat{x}$
$\hat{P}$
State Predictor
$\tilde{x}_{k+1} = \hat{x}_k + f(\hat{x}_k, u, t)\Delta t$
$F_k = \left[\frac{\partial f}{\partial x}\right]_{x_k, u_k}$
$\Phi_k = (I + F_k \Delta t)$
$\tilde{P}_{k+1} = \Phi_k \hat{P}_k \Phi_k^T + Q_k$
State and Covariance Predictions
$\tilde{x}$
$\tilde{P}$
(time step)

Figure 2.1-3: The Extended Kalman Filter algorithm.

2.2 Spatial Relationships and Notation.

To discuss vision navigation, it is necessary to describe the spatial relationship of the robot to the environment. It is also necessary to consider the spatial relationship of the camera to both the environment which it is viewing and the robot to which it is attached. It is necessary, therefore, to develop an analytical language for discussing spatial relationships between rigid bodies.

2.2.1 Reference Frames

The three primary elements of vision navigation—the environment, the robot, and the vision sensor—define three distinct rigid reference frames. They are henceforth denoted the "fixed frame" (fixed to the navigation environment), the "body frame" (fixed to the robot), and the "camera frame" (fixed to the vision sensor). Figure 2.2-1 illustrates these frames for a submersible robot in a neutral buoyancy tank. Because the robot moves in the environment and the camera may be steerable, the spatial relationships between the reference frames are generally dynamic.

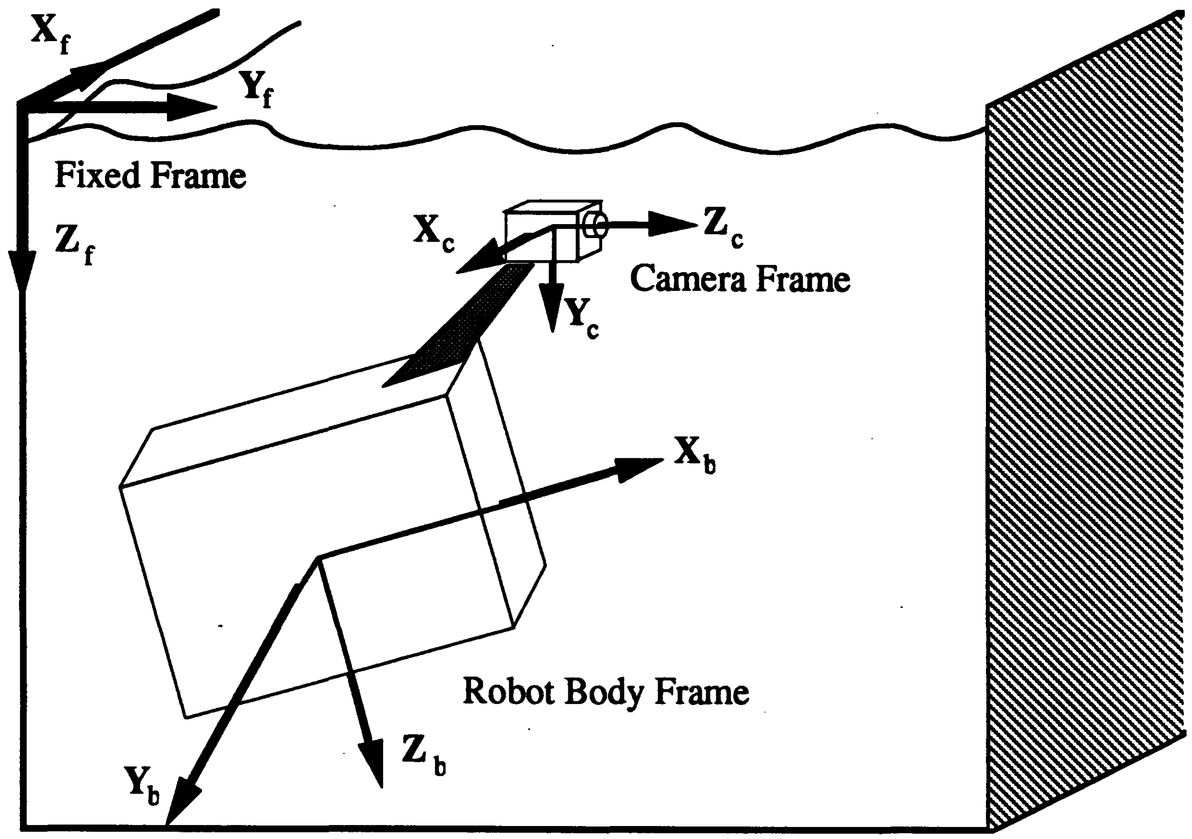


Figure 2.2-1: The three frames of reference required for robot vision navigation.

Each reference frame is defined by a Cartesian coordinate system with a right-handed set of axes (X,Y,Z).

The camera frame axes (X_c,Y_c,Z_c) conform to conventions used in machine vision and image processing. The origin is at the center of projection[3] (COP) of the imager, usually physically inside the camera lens. The Z_c-axis is the optical axis and points from the COP out into the world, as illustrated in Figure 2.2-1. Looking from the origin in the direction of the +Z_c-axis, the X_c-axis points to the right and the Y_c-axis points down. The image plane is perpendicular to the optical axis, parallel to the X_c-Y_c plane. Imaging geometry is discussed further in Chapter 3.

The body axes (X_b,Y_b,Z_b) conform to conventions for aircraft body axes. The origin is the center of mass of the vehicle. The X_b-axis points forward, the Y_b-axis points out along the right wing, and the Z_b-axis points down. Although free-flying robots are not required to have preferred orientations, most submersible robots and many space robots will have nominal attitudes in which the directions "down", "right", and "forward" are meaningful. If a robot does not have a preferred orientation, any arbitrary assignment of axes can be used.

[3]The imaging geometry is defined in section 3.2.

The coordinate axes for the fixed frame (X_f,Y_f,Z_f) can be oriented arbitrarily as well. However, like the robots, the environment will often have a natural orientation. In these cases, it is sensible to define the origin at some nominal hovering location of the robot, the Z_f-axis pointing "down" (usually along the gravity vector) from the origin, and the X_f-axis pointing toward the navigation target. With this definition of axes, the body axes align with the fixed axes when the robot hovers in an upright position at the reference location facing the target. This choice of axes allows the navigation variables to describe physically meaningful quantities.

The three frames of reference provide three different spatial bases from which 3D points and vectors can be specified. "Point" as used here is a physical location in space, a "vector" defines a magnitude and direction in space. Once referenced to a frame, both points and vectors can be specified using 3-vectors in the frame of reference. The notation used subsequently distinguishes between physical 3D points or vectors and their mathematical representations as 3-vectors in the various frames.[4]

For a physical point denoted **p**, the expression $\mathbf{p}_{:f}$ denotes the 3-vector in the fixed frame that extends from the fixed origin to **p**. Likewise, subscripts ":b" and ":c" reference the point **p** to the body and camera frames, respectively. A 3D vector **r** is frame-referenced using the same type of notation. The expressions $\mathbf{r}_{:f}$, $\mathbf{r}_{:b}$, and $\mathbf{r}_{:c}$ denote the 3-vectors in each frame that have the magnitude and direction of **r**.

Notationally, the colon before frame-reference subscripts is used to delineate between "descriptive" or "index" subscripts and "frame-reference" subscripts. The former type of subscript is used for identification as in the expression $\mathbf{p}_i$, which designates the ith point in some set of points, for example. This point referenced to the fixed frame can be specified as the 3-vector $\mathbf{p}_{i:f}$.

To specify components of the 3-vector, the subscripts of the 3-vector plus a "1", "2", or "3" are used. For example,

$$\mathbf{p}_{i:f} = \begin{pmatrix} p_{i:f1} \\ p_{i:f2} \\ p_{i:f3} \end{pmatrix} = \begin{pmatrix} X_f\text{-component} \\ Y_f\text{-component} \\ Z_f\text{-component} \end{pmatrix}.$$

Although the notation for 3-vectors and their components is lengthy, it is required for systematically describing the large number of variables used in the mathematical models for vision navigation.

[4]A 3-vector, as used in this document, is a mathematical element, an ordered set of 3 scalars called the components of the 3-vector. Clearly, an unreferenced point or vector cannot have components and thus is not referred to as a 3-vector. To refer to the physical element without committing to a particular mathematical reference, the terminology "3D point" or "3D vector" is used.

2.2.2 Reference Frame Relationships

The spatial relationship of one reference frame to another can be described by a 3D translation and a 3D rotation, each of which has three degrees of freedom (DOF). The translation between the origin of any frame "a" to the origin of a frame "b" is designated by the 3D vector $\mathbf{t}_{ab}$, the rotation by a *rotation operator* ${}^{a}_{b}R$.

The notation $\mathbf{t}_{ab}$ is reserved exclusively for specifying translation vectors between the origins of two reference frames "a" and "b". The order of subscripts is important, with

$$\mathbf{t}_{ab} = -\mathbf{t}_{ba}$$

being the vector from the origin of "a" to the origin of "b".

The operator ${}^{a}_{b}R : \Re^3 \rightarrow \Re^3$ maps $\mathbf{r}_{:b} \rightarrow \mathbf{r}_{:a}$, i.e.

$$\mathbf{r}_{:a} = {}^{a}_{b}R(\mathbf{r}_{:b}),$$

where $\mathbf{r}$ is any 3D vector and "a" and "b" designate arbitrary reference frames.

The arrangement of the sub- and superscript on the rotation operator symbol determines the direction of transformation. The mnemonic "cancelling" of the operand frame reference with the operator subscript yields the superscript as the frame reference for the result. This prevents confusion between the operator ${}^{a}_{b}R$ and its inverse ${}^{a}_{b}R^{-1} = {}^{b}_{a}R$.

Together, the pair $(\mathbf{t}_{ab}, {}^{a}_{b}R)$ is sufficient for describing the relationship between two frames through the transformation $\mathbf{p}_{:b} \rightarrow \mathbf{p}_{:a}$ as follows:

$$\mathbf{p}_{:a} = \mathbf{t}_{ab:a} + {}^{a}_{b}R(\mathbf{p}_{:b}),$$

where $\mathbf{p}$ is some 3D point. The above equality is the basic *frame transformation equation.*

Figure 2.2-2 illustrates some key relationships of vision navigation. The origins of the reference frames are designated $\mathbf{0}_f$, $\mathbf{0}_b$, and $\mathbf{0}_c$. Under this notation, $\mathbf{t}_{fb} = \mathbf{0}_b - \mathbf{0}_f$ and so on. Also, for any two reference frames "a" and "b", $\mathbf{0}_{a:a} = \mathbf{0}$ (the zero vector) and $\mathbf{0}_{a:b} = \mathbf{t}_{ba:b}$.

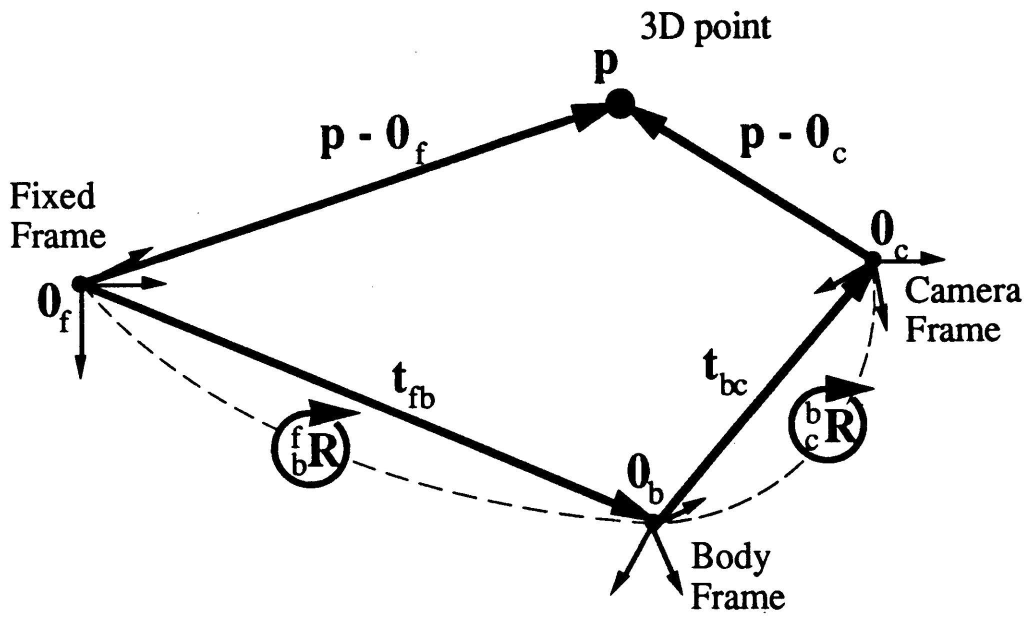


Figure 2.2-2: Key relationships between reference frames and 3D points in the environment.

In the vision navigation problem, each vector ($\mathbf{p}-\mathbf{0}_f$) is constant for a set of points and the other three vectors are generally time-varying. The camera-body relationship ($\mathbf{t}_{bc}, {}^{b}_{c}R$) is assumed known. The fixed-body relationship ($\mathbf{t}_{fb}, {}^{f}_{b}R$) represents the navigation parameters of interest. Thus, navigation can be thought of as the process of estimating the vector $\mathbf{t}_{fb}$ and the operator ${}^{f}_{b}R$.

The notion of estimating the position is straightforward because it is parameterized by the three scalar elements of the translation vector $\mathbf{t}_{fb:f}$. In order to estimate the rotational orientation and use it for attitude control, however, the rotation operator must likewise be parameterized.

The most common, and often useful, parameterization of the rotation operator consists of the nine elements of a direction cosine matrix. The transformation

$$\mathbf{r}_{:a} = {}^{a}_{b}R(\mathbf{r}_{:b})$$

for vector **r** can be written

$$\mathbf{r}_{:a} = \mathbf{R}_{ab}\ \mathbf{r}_{:b},$$

where $\mathbf{R}_{ab}$ is a (3,3) orthonormal matrix, the *direction cosine matrix*. The rows of $\mathbf{R}_{ab}$ are the unit vectors along the axes of the "a" frame, expressed in the "b" frame. The nine parameters of $\mathbf{R}_{ab}$ are necessarily subject to six constraints, however, because 3D rotation has only three degrees of freedom. Mathematically, the six constraints come from orthonormality of the matrix: orthogonality provides three constraints, normality another three.

Although this parameterization provides a convenient tool for rotating vectors, it proves cumbersome in both analytical and numerical manipulations, particularly of vehicle dynamics. There are, in fact, many alternate parameterizations of rotation, all of which have their particular strengths and weaknesses. The unit quaternion parameterization of the rotation operator, described in the next section, has four parameters and one constraint and is used because of its analytic and computational convenience.

2.2.3 Unit Quaternions for Representing Rotation

The unit quaternion representation of rotations is best characterized by its relationship to the axis/angle representation of rotation. Any 3D frame rotation can be described as a single rotation through an angle θ about some axis $\hat{\mathbf{n}}$ [HUGHES86, et al.]. The angle and axis comprise a four-parameter description of the rotation with one constraint (the axis is a unit vector).

Each axis/angle pair $(\theta,\hat{\mathbf{n}})$ is associated with a single unit quaternion

$$\mathring{\mathbf{q}} \triangleq (q_0, \mathbf{q}) = (\cos \tfrac{\theta}{2}, \hat{\mathbf{n}} \sin \tfrac{\theta}{2}).$$

The unit quaternion and axis/angle representation are interchangeable because $(\theta,\hat{\mathbf{n}}) \rightarrow \mathring{\mathbf{q}}$ is a single-valued function and the reverse mapping $\mathring{\mathbf{q}} \rightarrow (\theta,\hat{\mathbf{n}}), (-\theta,-\hat{\mathbf{n}})$ is double valued with both values representing identical rotations. The quaternion representation is preferable for describing rotations because it has a strict one-to-one correspondence with rotation operators.

As shown above, a quaternion is denoted by a boldface character with a hollow dot over it and consists of a *scalar part*, q_0, and a *vector part*, $\mathbf{q}$. A more complete representation of the quaternion is as an ordered set of four scalar parameters (q_0,q_1,q_2,q_3) where

$$q_0 = \cos(\theta/2)$$

$$\mathbf{q} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} = \hat{\mathbf{n}} \sin(\theta/2).$$

Quaternions that represent rotation are *unit* quaternions satisfying the relationship

$$\|\mathring{\mathbf{q}}\|_2^2 \triangleq q_0^2 + \|\mathbf{q}\|_2^2 = q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1.$$

Note that, like axis/angle representation, unit quaternions have four scalar parameters and one constraint.

The rotation matrix, axis/angle, and unit quaternion are, of course, equivalent descriptors of the rotation operator. The rotation matrix is a convenient computational tool for performing a single transformation and is analytically interesting because of the various interpretations of its rows and columns in the context of linear algebra. The axis/angle representation is physically meaningful but computationally clumsy and also suffers from the lack of a one-to-one correspondence with rotation operators. The unit quaternion shares the physical insight of the axis/angle representation and the computational convenience of the orthonormal rotation matrix because it is readily transformed into either of the other two.

The unit quaternion has additional advantages over the orthonormal matrix which strongly favor its use for processing rotational dynamics of free-flying robots and for similar applications. The quaternion is more compact and has less constraints, it is more efficient for composing successive rotations, and the numerical errors resulting from finite-precision arithmetic are more easily compensated for. Salamin summarizes the comparison as follows:

> For computation with rotations, quaternions offer the advantage of requiring only 4 numbers of storage, compared to 9 numbers for orthogonal matrices. Composition of rotations requires 16 multiplications and 12 additions in quaternion representation, but 27 multiplications and 18 additions in matrix representation. Rotating a vector, with the rotation matrix in hand, requires 9 multiplications and 6 additions. However, if the matrix must be calculated from a quaternion, then this calculation needs 10 multiplications and 21 additions. The quaternion representation is more immune to accumulated computational error. A quaternion which deviates from unicity can be fixed by $q \leftarrow q/|q|$, however a matrix which deviates from orthogonality must be fixed by the more involved calculation $R \leftarrow R(R^T R)^{-1/2}$, or some approximation thereto. [SALAMIN79]

For these reasons, among others, the proposed robot vision navigator utilizes unit quaternions for parameterizing all rotation operators. The notation $\dot{q}_{ab}$ shall be reserved for the unit quaternion associated with the rotation operator ${}^{a}_{b}R$. Once again, the order of subscripts is important.

The components of a rotation quaternion shall inherit the double subscripts of the quaternion plus an appropriate integer as follows:

$$\dot{q}_{ab} = (q_{ab0}, \mathbf{q}_{ab}) = (q_{ab0}, q_{ab1}, q_{ab2}, q_{ab3}).$$

Appendix A defines the conjugate for unit quaternions as

$$\dot{q}^* \triangleq (q_0, -\mathbf{q}) = (q_0, -q_1, -q_2, -q_3),$$

where the asterisk denotes conjugation.

For unit quaternions the conjugate and inverse are the same, and they both represent the physically opposite rotation of the original quaternion, i.e.

$$\dot{q}_{ab}^{-1} = \dot{q}^*_{ab} = \dot{q}_{ba}.$$

That the conjugate $\mathring{\mathbf{q}}^*_{ab}$ represents the inverse rotation of $\mathring{\mathbf{q}}_{ab}$ is obvious from the definition of the unit quaternion with respect to axis and angle of rotation. For any axis/angle pair, negating either the angle or axis (but not both) represents the physically opposite rotation. Either negation reverses the sign of the vector part, $\mathbf{q}_{ab} = \hat{\mathbf{n}} \sin(\theta/2)$, and has no effect on the scalar part, q_{ab0}.

In quaternion algebra, the rotation operation $\mathbf{r}_{:a} = {}^{a}_{b}R(\mathbf{r}_{:b})$ is implemented through a pre-multiplication by the rotation quaternion and a post-multiplication by its conjugate,

$$\mathring{\underline{\mathbf{r}}}_{:a} = \mathring{\mathbf{q}}_{ab}\mathring{\underline{\mathbf{r}}}_{:b}\mathring{\mathbf{q}}^*_{ab} = \mathring{\mathbf{q}}_{ab}\mathring{\underline{\mathbf{r}}}_{:b}\mathring{\mathbf{q}}_{ba},$$

where an underscored quaternion denotes a four-element form of the corresponding vector,

$$\mathring{\underline{\mathbf{r}}}_{:a} = [0,\mathbf{r}_{:a}] \text{ and } \mathring{\underline{\mathbf{r}}}_{:b} = [0,\mathbf{r}_{:b}],$$

which are (non-unit) quaternions with zero scalar part. The operation of *quaternion multiplication* is described in Appendix A.

By forming the (4,4) pre- and post-multiplication matrices (Q_{ab} and $\overline{Q}_{ab}$) for rotation $\mathring{\mathbf{q}}_{ab}$, as described in Appendix A, the rotation formula can be written

$$\mathring{\underline{\mathbf{r}}}_{:a} = \left(Q_{ab}\,\mathring{\underline{\mathbf{r}}}_{:b}\right)\mathring{\mathbf{q}}^*_{ab} = \left(\overline{Q}^{T}_{ab}Q_{ab}\right)\mathring{\underline{\mathbf{r}}}_{:b} = \begin{pmatrix} \mathring{\mathbf{q}}\cdot\mathring{\mathbf{q}} & \mathbf{0}^{T} \\ \mathbf{0} & \mathbf{R}_{ab} \end{pmatrix}\begin{pmatrix} 0 \\ \mathbf{r}_{:b} \end{pmatrix}$$

where

$$\mathbf{R}_{ab} = \begin{pmatrix} (q_0^2 + q_1^2 - q_2^2 - q_3^2) & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) \\ 2(q_1q_2 + q_0q_3) & (q_0^2 - q_1^2 + q_2^2 - q_3^2) & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & (q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{pmatrix}$$

and

$$\mathring{\mathbf{q}}\cdot\mathring{\mathbf{q}} \triangleq q_0^2 + q_1^2 + q_2^2 + q_3^2.$$

The (3,3) matrix $\mathbf{R}_{ab}$ is the orthonormal direction cosine matrix associated with the rotation. Here Salamin's claim that 10 multiplications and 21 additions are required to compute the elements of $\mathbf{R}$ from the elements of $\mathring{\mathbf{q}}$ is readily apparent. The reverse transformation $\mathbf{R}_{ab} \Rightarrow \mathring{\mathbf{q}}$ is possible as well but more complicated and is described in [SALAMIN79] and [HUGHES86].

In effect, using quaternion algebra to process a vector rotation amounts to generating the direction cosine matrix and performing the usual matrix multiplcation. For a single static rotation, then, there is no point in involving quaternions at all. The advantages of quaternions appear when multiple rotations are processed together, as in rotational dynamics of the robot vehicle.

An example of the use of quaternions in rotational computations is the frame transformation equation. Using the underscore notation $\mathring{\underline{\mathbf{p}}}_{:a} = [0, \mathbf{p}_{:a}]$, the frame transformation equation for point $\mathbf{p}$ can be re-written as the quaternion formula

$$\mathring{\underline{\mathbf{p}}}_{:a} = \mathring{\underline{\mathbf{t}}}_{ab:a} + \mathring{\mathbf{q}}_{ab}\, \mathring{\underline{\mathbf{p}}}_{:b}\, \mathring{\mathbf{q}}_{ba}.$$

The actual computation is performed by executing the transformation

$$\mathring{\mathbf{q}}_{ab} \Rightarrow \mathbf{R}_{ab}$$

and performing the matrix operation

$$\mathbf{p}_{:a} = \mathbf{t}_{ab:a} + \mathbf{R}_{ab}\mathbf{p}_{:b}.$$

A final issue of rotational representation is why an unconstrained three-parameter representation should not be used for describing the 3DOFs of rotational orientation of a robot instead of the unit quaternion (4 parameters, 1 constraint) or orthonormal matrix (9 parameters, 6 constraints). Three-parameter representations for rotation do exist (Euler angles, Gibbs vector), but any set of three parameters chosen to describe the three DOFs of rotation contains one or more geometrically singular orientations somewhere in the rotation space [JUNKINS86].

Attitude control of free-flying robots requires a robust description of rotation across the entire rotation space and therefore cannot rely on three-parameter rotational descriptions. The quaternion representation lacks singularities (as do the rotation matrix and axis/angle representations).

However, a three-parameter rotational representation is completely appropriate for describing small rotations from a nominal attitude because only a small portion of the rotation space is used precluding the possibility of encountering a singularity. The discrete-time estimator developed in Chapter 3 uses such a three-parameter representation for the small changes in rotation over a sampling interval. The total rotational state of the vehicle, however, is always represented by a unit quaternion.

This page not used.

3. Recursive Estimator.

Recursive estimation for vision navigation is based on the Extended Kalman Filter (EKF) described in Chapter 2 where the dynamic model is for the motion of the robot vehicle with respect to a fixed external frame and the measurements are image feature point locations provided by an image sequence processor. The role of the estimator is illustrated in Figure 3.0-1, where the state of the robot is designated $\mathbf{x}_{nav}$ and the measurements $\mathbf{y}$.

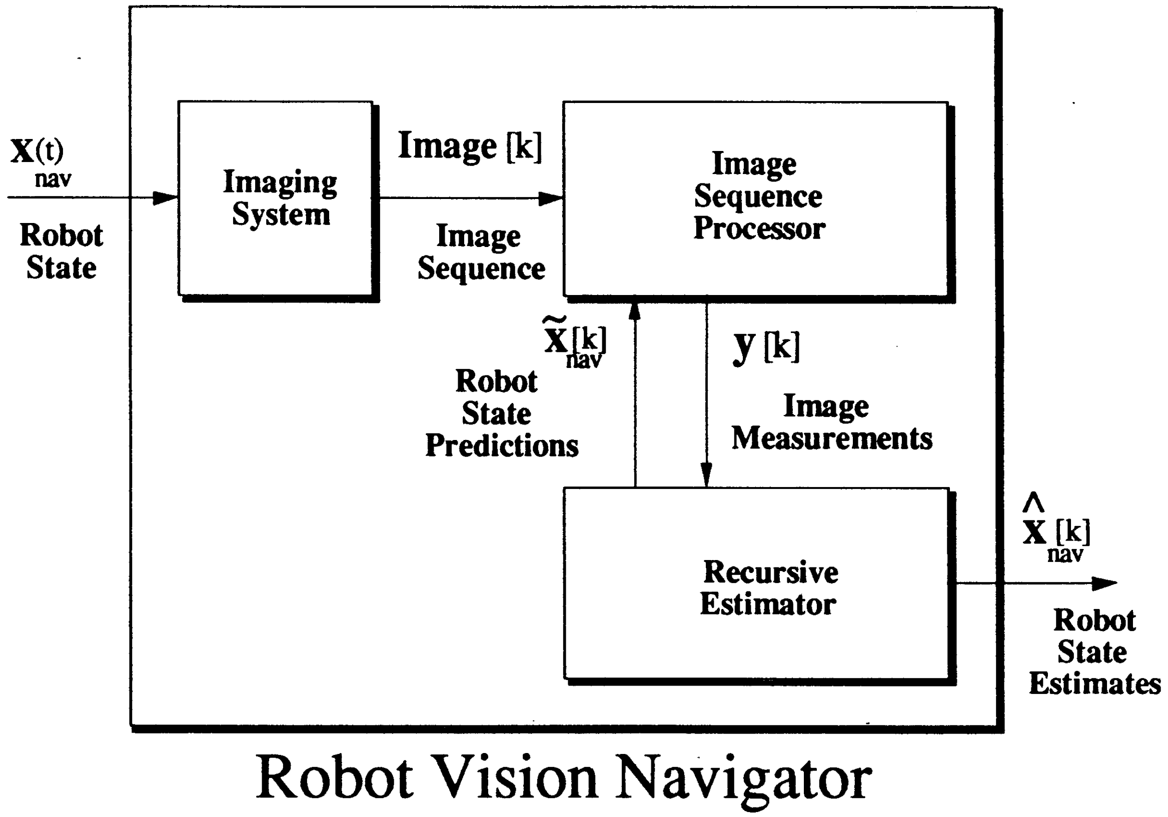


Figure 3.0-1: The role of the recursive estimator in the vision navigator.

In order to use the EKF framework, it is first necessary to define the state vector **x**, the measurement vector **y**, and the two functions **f**() and **h**() which define the dynamic and measurement models. To this end, the first two sections of this chapter develop the models, leading to description of the estimation algorithm for vision navigation in the final section.

3.1 Robot Vehicle Dynamics Model.

The motion control loop for free-flying robots is revisited in Figure 3.1-1, illustrating the input-output structure of the dynamic process in the notation of the EKF. The robot dynamics are mathematically modelled as a function mapping a control input trajectory **u**(t) and an initial state to a state trajectory $\mathbf{x}_{nav}(t)$.

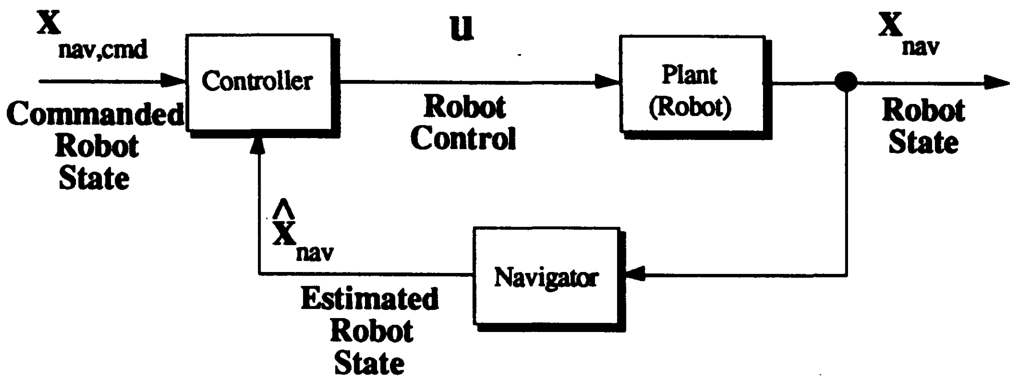


Figure 3.1-1: Robot motion control loop.

For free-flying robots, the control inputs are forces and torques generated by the vehicle's thrusters. In general, the thrusts can be combined into a net force and net torque on the vehicle, each of which has three degrees of freedom. The control vector is defined as

$$\mathbf{u} \triangleq \begin{pmatrix} \mathcal{F}_{:f} \\ \mathcal{T}_{:b} \end{pmatrix} \triangleq \begin{pmatrix} \mathcal{F}_{:f1} \\ \mathcal{F}_{:f2} \\ \mathcal{F}_{:f3} \\ \mathcal{T}_{:b1} \\ \mathcal{T}_{:b2} \\ \mathcal{T}_{:b3} \end{pmatrix}$$

where $\mathcal{F}_{:f}$ is the net force in the fixed frame and $\mathcal{T}_{:b}$ is the net torque in the body frame. The force and torque are referenced to fixed and body frames, respectively, for convenience in expressing the vehicle equations of motion. In reality, forces and torques will probably both be known in the body frame and so the net force must be transformed into the fixed frame using the current estimated orientation of the robot.[1]

The robot state vector is denoted $\mathbf{x}_{nav}$, and contains variables describing the position, attitude, translational velocity, and rotational velocity of the vehicle with respect to the environment. [The subscript "nav" is used to delineate the navigation state vector from the slightly modified state vector that shall be required for the estimator ($\mathbf{x}_{aux}$) and the vector used to describe the state of the camera relative to the robot ($\mathbf{x}_{cam}$).]

As discussed in Chapter 2, the mathematical parameterizations of position and attitude are, respectively, $\mathbf{t}_{fb:f}$ and $\mathring{\mathbf{q}}_{fb}$. The translation vector $\mathbf{t}_{fb:f}$ is the vector from fixed origin to body origin (center of mass of the robot) and the unit quaternion $\mathring{\mathbf{q}}_{fb}$ represents the rotation operator mapping vectors in the body frame to corresponding vectors in the fixed frame.

Translational velocity is defined by the vector $\mathbf{v}_{fb}$ as follows:

$$\mathbf{v}_{fb} = \dot{\mathbf{t}}_{fb} \triangleq \frac{d\mathbf{t}_{fb}}{dt}.$$

The velocity is referenced to the fixed frame for navigation purposes, i.e. $\mathbf{v}_{fb:f}$ is used.

Rotational velocity is referenced to the body frame and is notated $\mathbf{w}_{fb:b}$. Unfortunately, $\mathbf{w}_{fb:b}$ is not simply related to the quaternion as velocity is to the translation vector. However, it can be shown [HUGHES86] that

$$\mathbf{w}_{fb:b} = 2\frac{1}{q_{fb0}}\left(q_{fb0}^2\,\mathbf{I} + \mathbf{q}_{fb}\mathbf{q}_{fb}^T\right)\dot{\mathbf{q}}_{fb} - \mathbf{q}_{fb}\times\dot{\mathbf{q}}_{fb},$$

where $\mathring{\mathbf{q}}_{fb} = [q_{fb0}, \mathbf{q}_{fb}]$ represents the fixed-body rotation.

Thus rotational velocity depends not only on the time derivative of the quaternion (defined in Appendix A), but also on the value of the quaternion itself.

The *navigation state vector* is defined as follows:

[1]This is a source of error and possible divergence of the estimator because the state is not known exactly. However, if the estimator is tracking properly, the state should always be close enough to eliminate drastic effects of this uncertain transformation. The point in the estimation in which significant errors are most likely is at initialization. This can be compensated for by allowing the estimator to converge at the intialization of the estimator before powering the vehicle. More generally, the problem can be compensated for by adjusting the a priori dynamics noise covariance matrix, Q, perhaps dynamically, to reflect the uncertainty.

$$\mathbf{x}_{nav} \triangleq \begin{pmatrix} \mathbf{t}_{fb:f} \\ q_{fb0} \\ \mathbf{q}_{fb} \\ \mathbf{v}_{fb:f} \\ \mathbf{w}_{fb:b} \end{pmatrix} = \begin{pmatrix} t_{fb:f1} \\ t_{fb:f2} \\ t_{fb:f3} \\ q_{fb0} \\ q_{fb1} \\ q_{fb2} \\ q_{fb3} \\ v_{fb:f1} \\ v_{fb:f2} \\ v_{fb:f3} \\ w_{fb:b1} \\ w_{fb:b2} \\ w_{fb:b3} \end{pmatrix}$$

Note that there is a single constraint on the thirteen-element navigation state vector, namely that the sum of the squares of the four quaternion elements is equal to one. Hence, the state vector has twelve degrees of freedom.

Under these definitions of **u** and $\mathbf{x}_{nav}$, the EKF requires a state space model of the form

$$\dot{\mathbf{x}}_{nav} = \mathbf{f}(\mathbf{x}_{nav},\mathbf{u},t).$$

Such a model can be generated by applying Newtonian dynamics, treating the vehicle as a rigid body. The result is

$$\dot{\mathbf{x}}_{nav} = \begin{pmatrix} \dot{\mathbf{t}}_{fb:f} \\ \dot{q}_{fb0} \\ \dot{\mathbf{q}}_{fb} \\ \dot{\mathbf{v}}_{fb:f} \\ \dot{\mathbf{w}}_{fb:b} \end{pmatrix} = \begin{pmatrix} \mathbf{v}_{fb:f} \\ -\frac{1}{2}\mathbf{q}_{fb}\cdot\mathbf{w}_{fb:b} \\ \frac{1}{2}(q_{fb0}\mathbf{w}_{fb:b} + \mathbf{q}_{fb}\times\mathbf{w}_{fb:b}) \\ \frac{1}{m}\sum_i \mathbf{f}_i \\ \mathbf{J}^{-1}(\sum_i \boldsymbol{\tau}_i - \mathbf{w}_{fb:b}\times \mathbf{J}\,\mathbf{w}_{fb:b}) \end{pmatrix} \triangleq \begin{pmatrix} \mathbf{f}_t \\ \mathbf{f}_q \\ \mathbf{f}_v \\ \mathbf{f}_w \end{pmatrix} \triangleq \mathbf{f}(\mathbf{x}_{nav},\mathbf{u},t),$$

where m is the mass of the vehicle, $\{\mathbf{f}_i\}$ are forces acting on the vehicle (including $\mathcal{F}$), **J** is the inertia tensor measured with respect to the body axes, and $\{\boldsymbol{\tau}_i\}$ are torques acting on the vehicle (including $\mathcal{T}$). The relations

$$\dot{\mathbf{t}}_{fb:f} = \mathbf{f}_t, \quad \begin{pmatrix} \dot{q}_{fb0} \\ \dot{\mathbf{q}}_{fb} \end{pmatrix} = \mathbf{f}_q, \quad \dot{\mathbf{v}}_{fb:f} = \mathbf{f}_v, \quad \dot{\mathbf{w}}_{fb:b} = \mathbf{f}_w$$

apply.

The expression for $\mathbf{f_t}(\mathbf{x}_{nav},\mathbf{u},t)$ comes from formulas for velocities. The expression for $\mathbf{f_q}(\mathbf{x}_{nav},\mathbf{u},t)$ is derived in Appendix A. The expressions for $\mathbf{f_v}$ and $\mathbf{f_w}$ come from Newtonian rigid body dynamics and can be found in [MERIAM78].

3.2 Measurement Model.

As discussed in Chapter 1, the measurements of interest for vision navigation are a set of image locations corresponding to a set of known 3D points in the environment. The points are known in the sense that their coordinates in the fixed frame are known. Thus, the measurement model assumes knowledge of a set

$$\left\{\mathbf{p}_{i:f}\right\}, \quad i = 1,\ldots,N.$$

The measurements consist of the corresponding locations of these points in the image plane. For each point $\mathbf{p}_i$, the corresponding measurement is defined as $\mathbf{y}_i$, a 2-vector describing the image plane location at which $\mathbf{p}_i$ appears.

The measurement vector is thus defined as

$$\mathbf{y} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_N \end{pmatrix}$$

and has dimension 2N.

The measurement model required for the EKF is of the form

$$\mathbf{y} = \mathbf{h}(\mathbf{x}_{nav},t),$$

where $\mathbf{x}_{nav}$ is the navigation state vector defined in the previous section.

However, to derive the measurement relation, it is convenient to break the operator $\mathbf{h}$ into components, each corresponding to a single feature point. Thus

$$\mathbf{y} \triangleq \begin{Bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_N \end{Bmatrix} \triangleq \begin{Bmatrix} \mathbf{h}_1(\mathbf{x}_{nav},t) \\ \mathbf{h}_2(\mathbf{x}_{nav},t) \\ \vdots \\ \mathbf{h}_N(\mathbf{x}_{nav},t) \end{Bmatrix} \triangleq \mathbf{h}(\mathbf{x}_{nav},t).$$

It is sufficient to derive the general form of $\mathbf{h}_i$ in terms of $\mathbf{p}_i$ since the structure of each component is exactly the same, with $\mathbf{p}_i$ as a parameter. With the general form of $\mathbf{h}_i$ in hand, the measurement model can be built from the set of components $\{\mathbf{h}_i\}_{i=1}^{N}$, where each component is generated by applying the corresponding point from the set $\{\mathbf{p}_{i:f}\}_{i=1}^{N}$ as a parameter.

The operator $\mathbf{h}_i$ can be described as the composition of two basic operations: a frame transformation and a 3D⇒2D projection. The general frame transformation $\mathbf{T}$ maps the navigation state and any point in the fixed frame to the corresponding point in the camera frame. The specific frame transformation $\mathbf{T}_i$ has $\mathbf{p}_{i:f}$ built in as a parameter and maps the navigation state to $\mathbf{p}_{i:c}$, i.e.

$$\mathbf{p}_{i:c} \triangleq \mathbf{T}(\mathbf{x}_{nav},\mathbf{p}_{i:f}) \triangleq \mathbf{T}_i(\mathbf{x}_{nav}).$$

The camera operator $\mathbf{C}$ projects the point onto the image plane, i.e.

$$\mathbf{y}_i = \mathbf{C}(\mathbf{p}_{i:c}).$$

The combination of the two operators defines $\mathbf{h}_i$, i.e. $\mathbf{h}_i = \mathbf{C}\cdot\mathbf{T}_i$ or, equivalently,

$$\mathbf{y}_i = \mathbf{h}_i(\mathbf{x}_{nav}) = \mathbf{C}(\mathbf{T}_i(\mathbf{x}_{nav})).$$

Dependencies on time have been dropped for brevity. It is understood that the state and the frame transformation are dynamic.

The following sections define $\mathbf{C}$ and $\mathbf{T}_i$, thus implicitly defining $\mathbf{h}_i$ and $\mathbf{h}$. An explicit expression for $\mathbf{h}$ including all of its scalar arguments and parameters is complex and not very helpful and therefore omitted. However, the minute structure of $\mathbf{h}_i$ is examined in great detail when linearization of the measurement model is discussed in Section 3.3.

3.2.1 Camera Projection Model.

The camera projection operator $\mathbf{C}$ describes how a 3D point external to the camera comes to appear at a particular location in the image. For vision navigation, the camera imaging geometry is modelled as a *point-projection* operator, which is a good approximation to the actual imaging geometry of most lensed cameras [HORN86]. It is based upon the pinhole model of a camera, in which it is assumed that all light rays captured by the camera pass through a single point, the *center of projection* (COP).

In a pinhole camera, the light rays enter a box through a small aperture in the front and strike a photographic plate at the rear of the box, forming a 2D image, as depicted in Figure 3.2-1. If the hole is small enough, it approximates a point, the COP. The plane containing the rear surface of the box where the image forms is the *image plane*. The line perpendicular to the image plane which passes through the center of projection is the *optical axis*. The point where the optical axis pierces the image plane is called the *principal point*, and the distance along the optical axis between the center of projection and the principal point is called the *principal distance* or *effective focal length* of the imaging system.

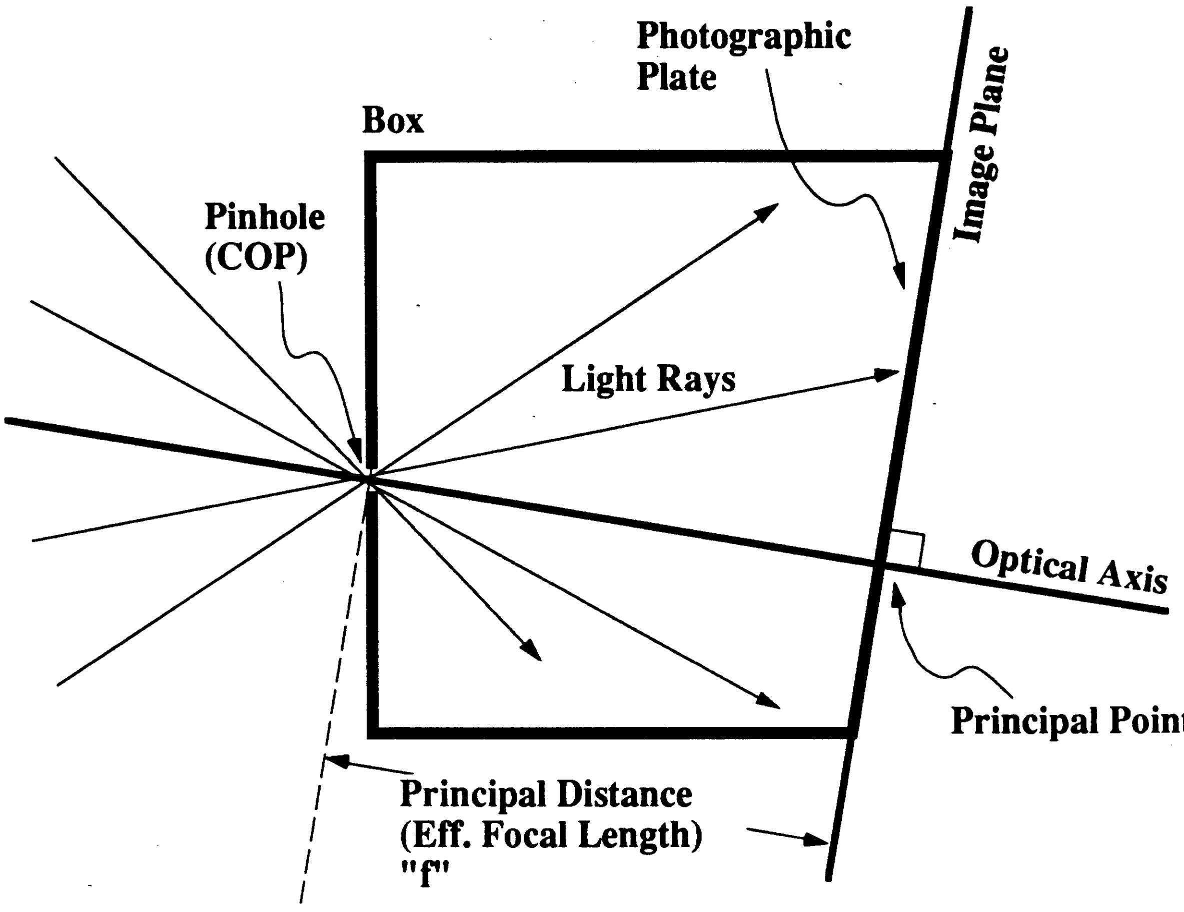


Figure 3.2-1: Schematic of a pinhole camera, demonstrating the point-projection imaging model. This picture represents a 2D slice of a pinhole camera.

Since the physical image formed by a pinhole camera is inverted, it is more convenient to consider the geometrically equivalent point-projection geometry in which a virtual image plane is in front of the center of projection, as shown in Figure 3.2-2. The virtual image which appears in a plane a distance f in front of the COP is identical in dimension and content to the image formed on a photographic plate a distance f behind the COP, except it is oriented in the same direction as the scene, i.e. it is not inverted.

According to this model, the image from a vision sensor arises from the geometrical arrangement depicted in Figure 3.2-2, based in the camera reference frame. As defined in §2.2, the COP is the origin of the camera frame, the Z_c-axis is the optical axis, and the image plane is parallel to the X_c-Y_c plane. The image plane is defined by the equation $Z_c = +f$, where f is the principal distance of the imaging system.

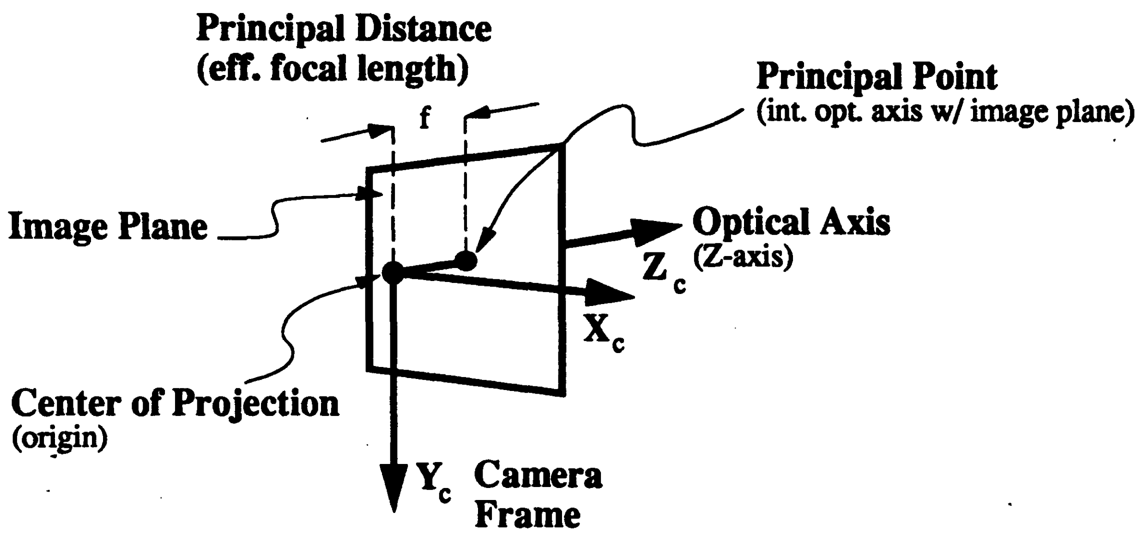


Figure 3.2-2: The Camera Frame coordinate system.

The image coordinates which are the components of $\mathbf{y}_i$ for a particular point $\mathbf{p}_i$ are the X_c- and Y_c-coordinates of the intersection of the image plane and a ray from the COP to $\mathbf{p}_i$. Thus,

$$\mathbf{y}_i \triangleq \begin{pmatrix} y_{i:1} \\ y_{i:2} \end{pmatrix} = \begin{pmatrix} \text{horizontal image coord } (X_c\text{-coord}) \\ \text{vertical image coord } (Y_c\text{-coord}) \end{pmatrix}.$$

This geometry leads to the defining equation for the projection operator:

$$\mathbf{y}_i \triangleq C(\mathbf{p}_{i:c}) = \frac{f}{p_{i:c3}} \begin{pmatrix} p_{i:c1} \\ p_{i:c2} \end{pmatrix}.$$

The equation is easily derived by considering similar triangles. The geometry governing the $y_{i:2}$ variable is shown in Figure 3.2-3 below.

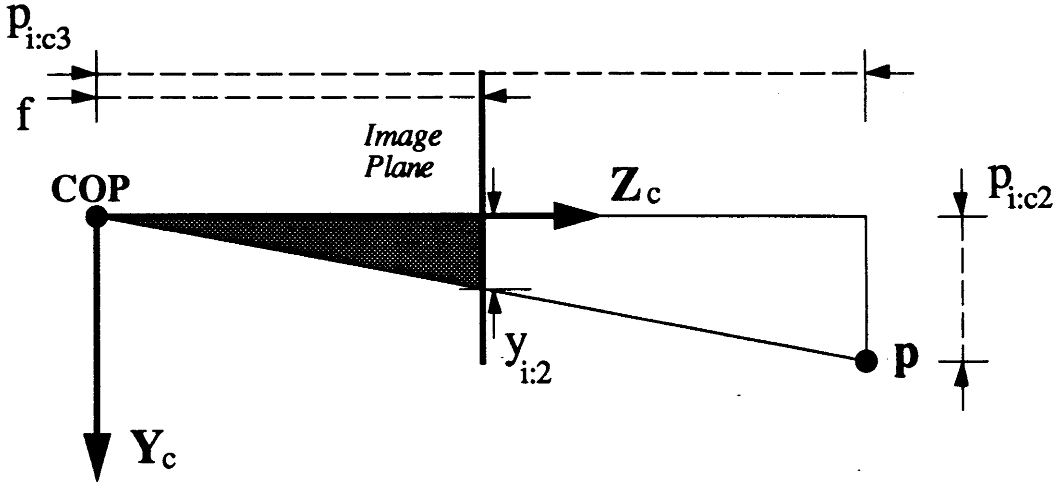


Figure 3.2-3: Similar triangles for deriving the projection equation.

3.2.2 Frame Transformation Operator.

For notational convenience the *camera state vector*, relating the camera and body frames, is defined as follows:

$$x_{cam} \triangleq \begin{pmatrix} t_{bc:b} \\ q_{bc0} \\ q_{bc} \end{pmatrix} = \begin{pmatrix} t_{bc:b1} \\ t_{bc:b2} \\ t_{bc:b3} \\ q_{bc0} \\ q_{bc1} \\ q_{bc2} \\ q_{bc3} \end{pmatrix}$$

Although the camera is allowed to move between video frames, it is assumed that the camera is stationary with respect to the vehicle when each frame is captured. Hence, no camera velocities are considered and none appear in the camera state vector.[2] Since the camera is mounted on the robot, it is assumed that $\mathbf{x}_{cam}$ can be accurately measured and provided to the navigator at any time.

The operator $\mathbf{T}_i$ depends upon $\mathbf{x}_{nav}$, $\mathbf{x}_{cam}$, and $\mathbf{p}_{i:f}$, but for estimation is considered a function only of $\mathbf{x}_{nav}$ since $\mathbf{x}_{cam}$ and $\mathbf{p}_{i:f}$ are precisely known at each time step. Later, the transformation will be written as a function of both $\mathbf{x}_{nav}$ and time, t, to indicate the possibly time-varying nature of $\mathbf{x}_{cam}$.

[2]This assumption may not be possible to meet for very robots which move very fast relative to frame rate. However, motion in navigation images, caused by camera motion relative to the robot or robot motion relative to the scene, introduces many complicated issues. Most applications of free-flying robots involve remote, often human-supervised, operation of relatively large vehicles with limited power supplies. As a result, they are likely to move slowly compared to frame capture rate, reducing the problems associated with motion. Additionally, use of a strobe and shuttering of the video sensor can virtually eliminate the remaining motion effects.

The derivation of the frame transformation operator $\mathbf{T}_i$ proceeds using the basic frame transformation introduced in §2.2. The following expressions use the orthonormal matrix form of the rotation operator:

$$\begin{aligned}
\mathbf{p}_{i:c} &= \mathbf{t}_{cf:c} + \mathbf{R}_{cf}\mathbf{p}_{i:f} \\
&= -\mathbf{t}_{fc:c} + \mathbf{R}_{cf}\mathbf{p}_{i:f} \\
&= -\mathbf{t}_{fb:c} - \mathbf{t}_{bc:c} + \mathbf{R}_{cf}\mathbf{p}_{i:f} \\
&= -\mathbf{R}_{cf}\mathbf{t}_{fb:f} - \mathbf{R}_{cb}\mathbf{t}_{bc:b} + \mathbf{R}_{cf}\mathbf{p}_{i:f} \\
&= \mathbf{R}_{cf}(\mathbf{p}_{i:f} - \mathbf{t}_{fb:f}) - \mathbf{R}_{cb}\mathbf{t}_{bc:b} \\
&= \mathbf{R}_{cb}\mathbf{R}_{bf}(\mathbf{p}_{i:f} - \mathbf{t}_{fb:f}) - \mathbf{R}_{cb}\mathbf{t}_{bc:b} \\
&= \mathbf{R}_{bc}^{-1}\mathbf{R}_{fb}^{-1}(\mathbf{p}_{i:f} - \mathbf{t}_{fb:f}) - \mathbf{R}_{bc}^{-1}\mathbf{t}_{bc:b} \\
&\triangleq \mathbf{T}_i(\mathbf{x}_{nav})
\end{aligned}$$

where $\mathbf{x}_{cam}$ and $\mathbf{p}_{i:f}$ are parameters of $\mathbf{T}_i$.

The last defining equality depends on the fact that

$$\mathbf{x}_{nav} \Rightarrow \dot{\mathbf{q}}_{fb} \Rightarrow \mathbf{R}_{fb}^{-1}$$

and

$$\mathbf{x}_{cam} \Rightarrow \dot{\mathbf{q}}_{bc} \Rightarrow \mathbf{R}_{bc}^{-1}.$$

Hence, $\mathbf{T}_i$ depends exclusively on independent variable $\mathbf{x}_{nav}$ and parameters $\mathbf{x}_{cam}$ and $\mathbf{p}_{i:f}$ as proposed.

To show explicit dependence on the navigation and camera state vectors, the rotation matrices can be converted to quaternion form, yielding the quaternion algebraic formula

$$\begin{aligned}
\underline{\dot{\mathbf{p}}}_{i:c} &= \dot{\mathbf{q}}_{cf}\,[\underline{\dot{\mathbf{p}}}_{i:f} - \underline{\dot{\mathbf{t}}}_{fb:f}]\,\dot{\mathbf{q}}_{fc} - \dot{\mathbf{q}}_{cb}\,\underline{\dot{\mathbf{t}}}_{bc:b}\,\dot{\mathbf{q}}_{bc} \\
&= \dot{\mathbf{q}}_{bc}^{*}\,\dot{\mathbf{q}}_{fb}^{*}\,[\underline{\dot{\mathbf{p}}}_{i:f} - \underline{\dot{\mathbf{t}}}_{fb:f}]\,\dot{\mathbf{q}}_{fb}\,\dot{\mathbf{q}}_{bc} - \dot{\mathbf{q}}_{bc}^{*}\,\underline{\dot{\mathbf{t}}}_{bc:b}\,\dot{\mathbf{q}}_{bc}.
\end{aligned}$$

The identities

$$\dot{\mathbf{q}}_{ab}^{*} = \dot{\mathbf{q}}_{ba} \quad \text{and} \quad \dot{\mathbf{q}}_{ab}\dot{\mathbf{q}}_{bc} = \dot{\mathbf{q}}_{ac}$$

have been used in the last step.

An explicit expansion of this formula in terms of the scalar components of $\mathbf{x}_{nav}$, $\mathbf{x}_{cam}$, and $\mathbf{p}_{i:f}$ is quite extensive and not particularly illuminating. It is also not required for numerical computations because the equation can be linearized using the chain rule, as will be shown in the next section.

3.3 The Estimation Algorithm for Vision Navigation.

The robot vision navigation problem does not exactly conform to the structure required for performing recursive estimation using the EKF because the 3DOF rotation is represented by a 4-parameter quaternion which contains a constraint. If the EKF is used to estimate the 13D $\mathbf{x}_{nav}$, it will find an optimal state in 13-dimensional space. However, valid values of $\mathbf{x}_{nav}$ exist only in a 12D subspace, the state space for the system. Estimation, therefore, should not take place on the 13D state vector.

Several approaches to solving the estimation problem were considered. The first of these is to formulate a new algorithm analagous to the Kalman Filter which solves a constrained least squares optimization rather than the unconstrained optimization of the original Kalman Filter. This route was discarded in light of the success and relative simplicity of the alternatives.

The second approach uses the existing Kalman Filter algorithm and assumes that the 13D estimate is not very far from the true optimal value in the 12D subspace. The state estimate could be obtained by projecting the 13D estimate into state space by normalizing the quaternion. This approach seems logical and was tested successfully in simulations.

The approach chosen for the final version of the vision navigator takes into account that the entire system is already linearized at each step of the EKF. Since only small rotations are expected over the course of a time step, there is no reason not to use a simple 3-parameter small-angle representation of rotation for describing the correction rotation in the estimation equation and for the propagated rotation in the state prediction equation.

The chosen angular representation is a set of Euler angles

$$\boldsymbol{\Theta} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix},$$

where θ_i represents a small rotation about the ith axis in the body frame. That is, θ_1 represents rotation about the X_b-axis, θ_2 about the Y_b-axis, and θ_3 about the Z_b-axis.

The result is a 12D state vector that represents the state space for small rotational states. This new state vector, $\mathbf{x}_{aux}$, is used internally in the estimator. The modified dynamics model, valid only for small rotations is

$$\dot{\mathbf{x}}_{aux} = \begin{pmatrix} \dot{\mathbf{t}}_{fb:f} \\ \dot{\Theta} \\ \dot{\mathbf{v}}_{fb:f} \\ \dot{\mathbf{w}}_{fb:b} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_t \\ \mathbf{f}_\Theta \\ \mathbf{f}_v \\ \mathbf{f}_w \end{pmatrix} = \mathbf{f}_{aux}(\mathbf{x}_{aux}, \mathbf{u}, t).$$

The *auxiliary state vector* $\mathbf{x}_{aux}$ is identical to the navigation state vector $\mathbf{x}_{nav}$ except for the description of rotation. The components $\mathbf{f}_t$, $\mathbf{f}_v$, and $\mathbf{f}_w$ are also identical to those from the original dynamics model. The function $\mathbf{f}_\Theta$ is defined by

$$\mathbf{f}_\Theta \triangleq \mathbf{w}_{fb:b}.$$

Since the model is valid only for small rotations, the value for the Euler angles obtained by integrating the function $\mathbf{f}_{aux}$ is valid only for small time intervals.

These Euler angles are used for representing the correction rotation in the estimator and for representing the propagated rotation in the predictor. In both cases, the Euler angles can be transformed into quaternions and composed with the nominal rotation quaternion to produce the total rotation. To demonstrate this, the Euler angles can be interpreted as an axis/angle pair, i.e.

$$\Theta = \phi \, \hat{n} = (\text{angle}) \; (\text{axis})$$

where ϕ is a small angle of rotation and $\hat{n}$ is a unit vector representing the axis of rotation.

The estimator uses a composite representation of rotation

$$\dot{q}_{fb} = \tilde{\dot{q}}_{fb} \, \dot{q}_{corr}$$

where

$$\dot{q}_{corr} = \left(\cos(\phi/2), \sin(\phi/2) \, \hat{n}\right)$$

for Euler angles $\phi\hat{n}$ defined relative to the <u>predicted</u> state.

Likewise, the predictor uses the composite representation of rotation

$$\dot{q}_{fb} = \hat{\dot{q}}_{fb} \, \dot{q}_{step}$$

where

$$\mathring{q}_{step} = \left(\cos(\phi/2), \sin(\phi/2)\,\hat{n}\right)$$

for Euler angles $\phi\hat{n}$ defined relative to the estimated state. Figure 3.3-1 illustrates the composite rotations used in the steps of the EKF for robot vision navigation.

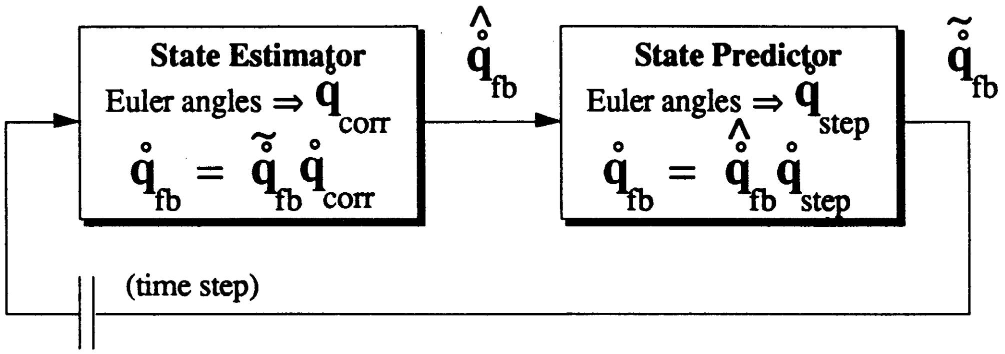

State Estimator
Euler angles ⇒ q_{corr}
$\mathring{q}_{fb} = \tilde{\mathring{q}}_{fb}\mathring{q}_{corr}$
$\hat{\mathring{q}}_{fb}$
State Predictor
Euler angles ⇒ $\hat{q}_{step}$
$\mathring{q}_{fb} = \hat{\mathring{q}}_{fb}\mathring{q}_{step}$
$\tilde{\mathring{q}}_{fb}$
(time step)

Figure 3.3-1: The use of Euler angles and composite rotations in the estimation and prediction steps of the robot vision navigator recursive estimator.

The remainder of this chapter describes the EKF-based recursive estimator for vision navigation.

3.3.1 Linearizing the Measurement Equation.

The measurement model for vision navigation was derived in Section 3.2. The measurement equation,

$$\mathbf{y} = \mathbf{h}(\mathbf{t}_{fb:f}, q_{fb0}, q_{fb}, t) = \begin{pmatrix} \mathbf{h}_1(\mathbf{t}_{fb:f}, q_{fb0}, q_{fb}, t) \\ \vdots \\ \mathbf{h}_N(\mathbf{t}_{fb:f}, q_{fb0}, q_{fb}, t) \end{pmatrix},$$

consists of N single-point measurement equations,

$$\mathbf{y}_i = \mathbf{h}_i(\mathbf{t}_{fb:f}, q_{fb0}, q_{fb}, t),$$

all of which are structurally identical.

The linearization with respect to the 12D auxiliary state vector results in a measurement matrix of the form

$$\left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}_{aux}}\right]_{\tilde{\mathbf{x}}^{(k)}_{aux}} = \mathbf{H}^{(k)} = \begin{pmatrix} \mathbf{H}^{(k)}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{H}^{(k)}_N \end{pmatrix}$$

where the single-point linearizations $\mathbf{H}^{(k)}_i$ have identical structures. Dropping the time-step index (k) for brevity, the matrix **H** has dimension (2N,12) and each $\mathbf{H}_i$ has dimension (2,12). It is sufficient to describe how to build $\mathbf{H}_i$, because the procedure can be performed N times to generate the entire **H** matrix.

To that end, consider the single-point measurement operator, defined in Section 3.2,

$$\mathbf{h}_i = \mathbf{C} \cdot \mathbf{T}_i,$$

the projection operator,

$$\mathbf{C}(\mathbf{p}_{i:c}) = \frac{f}{p_{i:c3}} \begin{pmatrix} p_{i:c1} \\ p_{i:c2} \end{pmatrix}$$

and the transformation operator, in matrix form,

$$\mathbf{T}_i(\mathbf{x}_{nav}) = \mathbf{R}^{-1}_{fc}(\mathbf{p}_{i:f} - \mathbf{t}_{fb:f}) - \mathbf{R}^{-1}_{bc}\mathbf{t}_{bc:b}.$$

Note that the transformation operator, and hence the measurement operator, depend only upon the first six state variables of $\mathbf{x}_{aux}$, the elements of $\mathbf{t}_{fb:f}$ and $\boldsymbol{\Theta}$. This is because the image from which the measurements are taken is assumed to represent a single instant of time and thus the measurements do not depend on translational or rotational velocity. The consequence for $\mathbf{H}_i$ is that the last six columns are all zero, i.e.

$$\underset{(2,12)}{\mathbf{H}_i = \left[\frac{\partial \mathbf{h}_i}{\partial \mathbf{x}_{est}}\right]_{\tilde{\mathbf{x}}_{est}}} = \left[\underset{(2,3)}{\frac{\partial \mathbf{h}_i}{\partial \mathbf{t}_{fb:f}}} \quad \underset{(2,3)}{\frac{\partial \mathbf{h}_i}{\partial \boldsymbol{\Theta}}} \quad \underset{(2,3)}{\frac{\partial \mathbf{h}_i}{\partial \mathbf{v}_{fb:f}}} \quad \underset{(2,3)}{\frac{\partial \mathbf{h}_i}{\partial \mathbf{w}_{fb:b}}}\right] = \left[\underset{(2,3)}{\frac{\partial \mathbf{h}_i}{\partial \mathbf{t}_{fb:f}}} \quad \underset{(2,3)}{\frac{\partial \mathbf{h}_i}{\partial \boldsymbol{\Theta}}} \quad \underset{(2,3)}{\mathbf{0}} \quad \underset{(2,3)}{\mathbf{0}}\right].$$

Thus each matrix $\mathbf{H}_i$ contains only two (2,3) matrices, $[\partial \mathbf{h}_i/\partial \mathbf{t}]$ in the first three columns and $[\partial \mathbf{h}/\partial \boldsymbol{\Theta}]$ in the fourth through sixth columns.

First three columns.

The first matrix is relatively simple to compute. From the chain rule,

$$\frac{\partial \mathbf{h}_i}{\partial t_{fb:f}} = \frac{\partial(\mathbf{C}\cdot\mathbf{T}_i)}{\partial t_{fb:f}} = \frac{\partial(\mathbf{C}\cdot\mathbf{T}_i)}{\partial \mathbf{T}_i}\frac{\partial \mathbf{T}_i}{\partial t_{fb:f}}.$$

The definition of the projection operator reveals

$$\frac{\partial(\mathbf{C}\cdot\mathbf{T}_i)}{\partial \mathbf{T}_i} = \frac{\partial \begin{pmatrix} T_{i:1} \\ T_{i:2} \end{pmatrix}}{\partial \mathbf{T}_i}\frac{f}{T_{i:3}} + \begin{pmatrix} T_{i:1} \\ T_{i:2} \end{pmatrix}\frac{\partial \left(\frac{f}{T_{i:3}}\right)}{\partial \mathbf{T}_i} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}\frac{f}{T_{i:3}} + \begin{pmatrix} T_{i:1} \\ T_{i:2} \end{pmatrix}\begin{pmatrix} 0 & 0 & -f/T^2_{i:3} \end{pmatrix}$$

which leads to

$$\frac{\partial \mathbf{h}_i}{\partial t_{fb:f}} = \frac{f}{T^2_{i:3}}\left(\begin{pmatrix} \partial T_{i:1}/\partial t_{fb:f} \\ \partial T_{i:2}/\partial t_{fb:f} \end{pmatrix} T_{i:3} - \begin{pmatrix} T_{i:1} \\ T_{i:2} \end{pmatrix} \partial T_{i:3}/\partial t_{fb:f}\right\}$$

Since

$$\frac{\partial \mathbf{T}_i}{\partial t_{fb:f}} \triangleq \begin{pmatrix} \partial T_{i:1}/\partial t_{fb:f} \\ \partial T_{i:2}/\partial t_{fb:f} \\ \partial T_{i:3}/\partial t_{fb:f} \end{pmatrix} = -\mathbf{R}^{-1}_{fc},$$

which derives directly from the transformation equation, the first three columns of $\mathbf{H}_i$ can be obtained by computing $\mathbf{T}_i(\mathbf{x}_{nav})$ and $\mathbf{R}_{fc}$. Figure 3.3-2 summarizes the computation of $[\partial \mathbf{h}_i/\partial t_{fb:f}]$ from the predicted navigation state.

$$\boxed{\begin{array}{rcll} \tilde{\mathbf{x}}_{nav} & \Rightarrow & \mathbf{R}_{fb} & \\ \mathbf{x}_{cam} & \Rightarrow & \mathbf{R}_{bc} & \\ \mathbf{R}^{-1}_{fc} = (\mathbf{R}_{fb}\mathbf{R}_{bc})^{-1} & \Rightarrow & \dfrac{\partial \mathbf{T}_i}{\partial t_{fb:f}} & \\ \tilde{\mathbf{x}}_{nav} & \Rightarrow & \mathbf{T}_i(\mathbf{x}_{nav}) & \\ \mathbf{T}_i(\mathbf{x}_{nav}), \dfrac{\partial \mathbf{T}_i}{\partial t_{fb:f}} & \Rightarrow & \dfrac{\partial(\mathbf{C}\cdot\mathbf{T}_i)}{\partial t_{fb:f}} = & \dfrac{\partial \mathbf{h}_i}{\partial t_{fb:f}} \end{array}}$$

Figure 3.3-2: Steps involved in computing the first three columns $[\partial \mathbf{h}_i/\partial t]$ of the measurement matrix $\mathbf{h}_i$.

<u>**Second three columns.**</u>

Computation of the second three columns requires a similar formula, derived as above:

$$\frac{\partial \mathbf{h}_i}{\partial \boldsymbol{\Theta}} = \frac{\partial (\mathbf{C}\cdot\mathbf{T}_i)}{\partial \boldsymbol{\Theta}} = \frac{f}{T_{i:3}^2}\left(\begin{pmatrix}\partial T_{i:1}/\partial\boldsymbol{\Theta}\\ \partial T_{i:2}/\partial\boldsymbol{\Theta}\end{pmatrix} T_{i:3} - \begin{pmatrix}T_{i:1}\\ T_{i:2}\end{pmatrix}\partial T_{i:3}/\partial\boldsymbol{\Theta}\right).$$

The required computation of $[\partial \mathbf{T}_i/\partial\boldsymbol{\Theta}]$ is more complex than the computation of $[\partial \mathbf{T}_i/\partial \mathbf{t}]$, requiring either lengthy expansions of the rotation operations in terms of the angles, or repeated invocation of the chain rule. Since the approaches are equivalent, the latter is used so that the problem can be tackled in steps. It requires evaluation of

$$\underset{(3,4)}{\frac{\partial \mathbf{T}_i}{\partial \boldsymbol{\Theta}} = \frac{\partial \mathbf{T}_i}{\partial \mathring{\mathbf{q}}_{fc}}}\ \underset{(4,4)}{\frac{\partial \mathring{\mathbf{q}}_{fc}}{\partial \mathring{\mathbf{q}}_{fb}}}\ \underset{(4,4)}{\frac{\partial \mathring{\mathbf{q}}_{fb}}{\partial \mathring{\mathbf{q}}_{corr}}}\ \underset{(4,3)}{\frac{\partial \mathring{\mathbf{q}}_{corr}}{\partial \boldsymbol{\Theta}}}.$$

The first matrix can be found from the transformation equation because $\mathbf{R}_{fc}$ depends upon $\mathring{q}_{fc}$. Specifically,

$$\mathbf{R}_{fc}^{-1} = \mathbf{R}_{fc}^{T} = \begin{pmatrix} (q_{fc0}^2 + q_{fc1}^2 - q_{fc2}^2 - q_{fc3}^2) & 2(q_{fc1}q_{fc2} - q_{fc0}q_{fc3}) & 2(q_{fc1}q_{fc3} + q_{fc0}q_{fc2}) \\ 2(q_{fc1}q_{fc2} + q_{fc0}q_{fc3}) & (q_{fc0}^2 - q_{fc1}^2 + q_{fc2}^2 - q_{fc3}^2) & 2(q_{fc2}q_{fc3} - q_{fc0}q_{fc1}) \\ 2(q_{fc1}q_{fc3} - q_{fc0}q_{fc2}) & 2(q_{fc2}q_{fc3} + q_{fc0}q_{fc1}) & (q_{fc0}^2 - q_{fc1}^2 - q_{fc2}^2 + q_{fc3}^2) \end{pmatrix}^T.$$

Expanding the rotation matrix in terms of its columns,

$$\mathbf{R}_{fc}^{-1} = \mathbf{R}_{cf} \triangleq \left(\mathbf{r}_{cf1}\ \mathbf{r}_{cf2}\ \mathbf{r}_{cf3}\right),$$

the transformation equation yields

$$\begin{aligned} \mathbf{T}_i(\mathbf{x}_{nav}) &= \left(\mathbf{r}_{cf1}\ \mathbf{r}_{cf2}\ \mathbf{r}_{cf3}\right)\left(\mathbf{p}_{i:f} - \mathbf{t}_{fb:f}\right) - \mathbf{R}_{bc}^{-1}\mathbf{t}_{bc:b} \\ &= (p_{i:f1}-t_{fb:f1})\mathbf{r}_{cf1} + (p_{i:f2}-t_{fb:f2})\mathbf{r}_{cf2} + (p_{i:f3}-t_{fb:f3})\mathbf{r}_{cf3} - \mathbf{R}_{bc}^{-1}\mathbf{t}_{bc:b}. \end{aligned}$$

Note that the last term does not depend upon the auxiliary state vector and will dissappear under differentiation.

It then follows that

$$\frac{\partial \mathbf{T}_i}{\partial \mathring{\mathbf{q}}_{fc}} = (p_{i:f1}-t_{fb:f1})\frac{\partial \mathbf{r}_{cf1}}{\partial \mathring{\mathbf{q}}_{fc}} + (p_{i:f2}-t_{fb:f2})\frac{\partial \mathbf{r}_{cf2}}{\partial \mathring{\mathbf{q}}_{fc}} + (p_{i:f3}-t_{fb:f3})\frac{\partial \mathbf{r}_{cf3}}{\partial \mathring{\mathbf{q}}_{fc}}$$

$$= \quad 2(p_{i:f1}-t_{fb:f1})\begin{pmatrix} q_{fc0} & q_{fc1} & -q_{fc2} & -q_{fc3} \\ -q_{fc3} & q_{fc2} & q_{fc1} & -q_{fc0} \\ q_{fc2} & q_{fc3} & q_{fc0} & q_{fc1} \end{pmatrix}$$

$$+ \quad 2(p_{i:f2}-t_{fb:f2})\begin{pmatrix} q_{fc3} & q_{fc2} & q_{fc1} & q_{fc0} \\ q_{fc0} & -q_{fc1} & q_{fc2} & -q_{fc3} \\ -q_{fc1} & -q_{fc0} & q_{fc3} & q_{fc2} \end{pmatrix}$$

$$+ \quad 2(p_{i:f3}-t_{fb:f3})\begin{pmatrix} -q_{fc2} & q_{fc3} & -q_{fc0} & q_{fc1} \\ q_{fc1} & q_{fc0} & q_{fc3} & q_{fc2} \\ q_{fc0} & -q_{fc1} & -q_{fc2} & q_{fc3} \end{pmatrix},$$

which can be computed from the predicted state ($\tilde{t}_{fb:f}$, $\tilde{\dot{q}}_{fb}$) and the camera state $\dot{q}_{bc}$ along with the known 3D coordinates of the feature point in the fixed frame, $p_{i:f}$.

The second and third matrices in the partial derivative chain are simply quaternion multiplication matrices. Since (see Appendix A)

$$\dot{q}_{fc} = \dot{q}_{fb}\dot{q}_{bc} = \bar{Q}_{bc}\begin{pmatrix} q_{fb0} \\ q_{fb} \end{pmatrix} \triangleq \begin{pmatrix} q_{bc0} & -q_{bc1} & -q_{bc2} & -q_{bc3} \\ q_{bc1} & q_{bc0} & q_{bc3} & -q_{bc2} \\ q_{bc2} & -q_{bc3} & q_{bc0} & q_{bc1} \\ q_{bc3} & q_{bc2} & -q_{bc1} & q_{bc0} \end{pmatrix}\begin{pmatrix} q_{fb0} \\ q_{fb1} \\ q_{fb2} \\ q_{fb3} \end{pmatrix}$$

then

$$\frac{\partial \dot{q}_{fc}}{\partial \dot{q}_{fb}} = \bar{Q}_{bc} \quad \text{where} \quad \dot{q}_{bc} \Rightarrow \bar{Q}_{bc}$$

Likewise,

$$\frac{\partial \dot{q}_{fb}}{\partial \dot{q}_{corr}} = Q_{\tilde{fb}} \quad \text{where} \quad \tilde{\dot{q}}_{fb} \Rightarrow Q_{\tilde{fb}}$$

because

$$\dot{q}_{fb} = \tilde{\dot{q}}_{fb}\dot{q}_{corr}.$$

The matrix $\bar{\mathbf{Q}}_{bc}$ is the postmultiplier matrix for $\partial\mathring{\mathbf{q}}_{bc}$ and $\mathbf{Q}_{\tilde{fb}}$ is the premultiplier matrix for the prediction of $\partial\mathring{\mathbf{q}}_{fb}$ as defined in Appendix A.

Finally,

$$\frac{\partial\mathring{\mathbf{q}}_{corr}}{\partial\boldsymbol{\Theta}} = \begin{pmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} \end{pmatrix}$$

This comes directly from the definition of $\mathring{\mathbf{q}}_{corr}$. Since $\boldsymbol{\Theta} \triangleq \phi\hat{\mathbf{n}}$, where $\hat{\mathbf{n}} \triangleq (n_1,n_2,n_3)$ is a unit vector, gives

$$\frac{\partial\mathring{\mathbf{q}}_{corr}}{\partial\boldsymbol{\Theta}} = \frac{\partial}{\partial\boldsymbol{\Theta}}\begin{pmatrix} \cos(\phi/2) \\ \sin(\phi/2)\hat{\mathbf{n}} \end{pmatrix} \approx \frac{\partial}{\partial\boldsymbol{\Theta}}\begin{pmatrix} 1 - \frac{\phi^2}{8} + \cdots \\ \frac{\phi}{2}\hat{\mathbf{n}} \end{pmatrix} = \begin{pmatrix} \mathbf{0}^T \\ \frac{1}{2}\frac{\partial}{\partial\boldsymbol{\Theta}}\boldsymbol{\Theta} \end{pmatrix} = \begin{pmatrix} \mathbf{0}^T \\ \frac{1}{2}\mathbf{I} \end{pmatrix},$$

near $\boldsymbol{\Theta} = \mathbf{0}$.

The steps involved in linearization of the fourth through sixth rows of $\mathbf{H}_i$ are summarized in Figure 3.3-3 below.

- $\left.\begin{matrix} \tilde{x}_{nav} & \Rightarrow & \dot{\tilde{q}}_{fb} \\ x_{cam} & \Rightarrow & \dot{q}_{bc} \end{matrix}\right\} \Rightarrow \dot{q}_{fc}$
- $\dot{q}_{fc} \Rightarrow \frac{\partial r_{cf1}}{\partial \dot{q}_{fc}}, \frac{\partial r_{cf1}}{\partial \dot{q}_{fc}}, \frac{\partial r_{cf1}}{\partial \dot{q}_{fc}} \Rightarrow \frac{\partial T_i}{\partial \dot{q}_{fc}}$
- $\partial \dot{q}_{bc} \Rightarrow \bar{Q}_{bc} = \frac{\partial \dot{q}_{fc}}{\partial \dot{q}_{fb}}$
- $\partial \tilde{q}_{fb} \Rightarrow Q_{\tilde{fb}} = \frac{\partial \dot{q}_{fb}}{\partial \dot{q}_{cor}}$
- $\frac{\partial \dot{q}_{cor}}{\partial \Theta}, \frac{\partial \dot{q}_{fb}}{\partial \dot{q}_{cor}}, \frac{\partial \dot{q}_{fc}}{\partial \dot{q}_{fb}}, \frac{\partial T_i}{\partial \dot{q}_{fc}} \Rightarrow \frac{\partial T_i}{\partial \Theta}$
- $\tilde{x}_{nav} \Rightarrow T_i(\tilde{x}_{nav})$
- $T_i(\tilde{x}_{nav}), \frac{\partial T_i}{\partial \Theta} \Rightarrow \frac{\partial (C \cdot T_i)}{\partial \Theta} = \frac{\partial h_i}{\partial \Theta}$

Figure 3.3-3: Steps toward computing the fourth through sixth columns of H_i.

The H_i matrix can be constructed from the calculations outlined here. Repeating the steps which depend upon the particular feature point p_i for i = 1...N yields the matrices $H_1,\ldots,H_N$, which collectively define the measurement matrix H for the current time step.

3.3.2 State Estimation.

The remainder of the state estimation follows the DKF framework, assuming the linearization is valid for small deviations of state $\delta x_{aux} = x_{aux} - \tilde{x}_{aux}$. The gain is computed as usual:

$$K = \tilde{P}H^T(H\tilde{P}H^T - R)^{-1}.$$

The nonlinear measurement equation is used to compute the measurement estimate, which is in turn used to compute the estimated auxiliary state vector:

$$\hat{x}_{aux} \triangleq \begin{pmatrix} \hat{t}_{fb:f} \\ \hat{\Theta} \\ \hat{v}_{fb:f} \\ \hat{w}_{fb:b} \end{pmatrix} = \tilde{x}_{aux} + K(y - h(\tilde{x}_{nav},t)) = \begin{pmatrix} \tilde{t}_{fb:f} \\ 0 \\ \tilde{v}_{fb:f} \\ \tilde{w}_{fb:b} \end{pmatrix} + K(y - \tilde{y}).$$

The auxiliary state vector prediction $\tilde{\mathbf{x}}_{\mathrm{aux}}$ consists of the state predictions for position, velocity, and rotational velocity and $\mathbf{0}$ for the Euler angle prediction.

As discussed in the introduction to this section, the correction quaternion $\mathring{\mathbf{q}}_{\mathrm{corr}}$ is computed from the estimated angles $\hat{\boldsymbol{\theta}}$ and composed with the predicted quaternion to update the estimated quaternion.

The error covariance matrix representing the error covariances of the auxiliary state vector is computed as usual:

$$\hat{\mathbf{P}} = (\mathbf{I} - \mathbf{K}\,\mathbf{H})\tilde{\mathbf{P}}.$$

3.3.3 State Prediction.

The position, velocity, and rotational velocity are propagated as usual, using the nonlinear dynamics model:

$$\tilde{\mathbf{t}}^{(k+1)}_{\mathrm{fb:f}} = \hat{\mathbf{t}}^{(k)}_{\mathrm{fb:f}} + \mathbf{f}_{\mathbf{t}}(\hat{\mathbf{x}}^{(k)}_{\mathrm{nav}})\ \Delta t = \hat{\mathbf{t}}^{(k)}_{\mathrm{fb:f}} + \hat{\mathbf{v}}^{(k)}_{\mathrm{fb:f}}\,\Delta t$$

$$\tilde{\mathbf{v}}^{(k+1)}_{\mathrm{fb:f}} = \hat{\mathbf{v}}^{(k)}_{\mathrm{fb:f}} + \mathbf{f}_{\mathbf{v}}(\hat{\mathbf{x}}^{(k)}_{\mathrm{nav}})\ \Delta t$$

$$\tilde{\mathbf{w}}^{(k+1)}_{\mathrm{fb:b}} = \hat{\mathbf{w}}^{(k)}_{\mathrm{fb:b}} + \mathbf{f}_{\mathbf{w}}(\hat{\mathbf{x}}^{(k)}_{\mathrm{nav}})\ \Delta t.$$

The rotation is processed differently, since it makes no sense to add rotation quaternions. One way of propagating the state is to consider the rotation

$$\hat{\mathbf{w}}^{(k)}_{\mathrm{fb:b}}\ \Delta t = \left(\|\hat{\mathbf{w}}^{(k)}_{\mathrm{fb:b}}\|\ \Delta t\right) \frac{\hat{\mathbf{w}}^{(k)}_{\mathrm{fb:b}}}{\|\hat{\mathbf{w}}^{(k)}_{\mathrm{fb:b}}\|} = (\text{angle})\ \ (\text{axis}),$$

which represents the angle and axis of a rotation over the time interval, assuming a constant rotational velocity equal to the estimated rotational velocity. Since it is in axis/angle form, this rotation is easily expressed as a quaternion,

$$\mathring{\mathbf{q}}_{\mathrm{step}} = \left(\cos(\tfrac{1}{2}(\text{angle})),\ \sin(\tfrac{1}{2}(\text{angle}))\cdot\text{axis} \right),$$

and then composed with the estimated rotation to yield the predicted rotation for the next time step[3]:

[3]Although this is an analytically sound way of describing the correction quaternion $\mathring{\mathbf{q}}_{\mathrm{step}}$, it is numerically poor because the angles are small and vulnerable to quantization error. Note, in particular, that if the magnitude of the angular velocity is zero, or very close, computation of the axis becomes undefined. Hence, in numerical

$$\tilde{\mathbf{q}}_{fb}^{(k+1)} = \hat{\mathbf{q}}_{fb}^{(k)} \bullet \mathbf{q}_{step}.$$

The covariance is computed as usual, with the state transition matrix resulting from $\mathbf{f}_{aux}()$, i.e.

$$\tilde{\mathbf{P}}_{(k+1)} = \mathbf{\Phi}_{(k)} \hat{\mathbf{P}}_{(k)} \mathbf{\Phi}_{(k)}^{T} + \mathbf{Q},$$

where

$$\mathbf{\Phi}_{(k)} = \left(\mathbf{I} + \left[\frac{\partial \mathbf{f}_{aux}}{\partial \mathbf{x}_{aux}}\right]_{\hat{\mathbf{x}}_{aux}} \Delta t\right)$$

3.3.4 The Vision Navigation Recursive Estimator.

The steps required for recursive estimation for the robot vision navigator are summarized in Figure 3.3-4 below.

The peculiarities of the vision navigation estimator include that the right half of the measurement matrix **H** contains zero, **H** is computed two rows at a time, and both a global state vector $\mathbf{x}_{nav}$ and a state vector local to the current computation $\mathbf{x}_{aux}$ are maintained. Furthermore, in the updates and prediction steps for the global state vector, rotations are composed rather than added. Otherwise, the vision navigator estimator is a straightforward application of the EKF.

computations, the small angle approximation should be made for the sine function yielding the unit quaternion $(\sqrt{1-\mathbf{\Theta}\bullet\mathbf{\Theta}/4}, \theta_1/2, \theta_2/2, \theta_3/2)$.

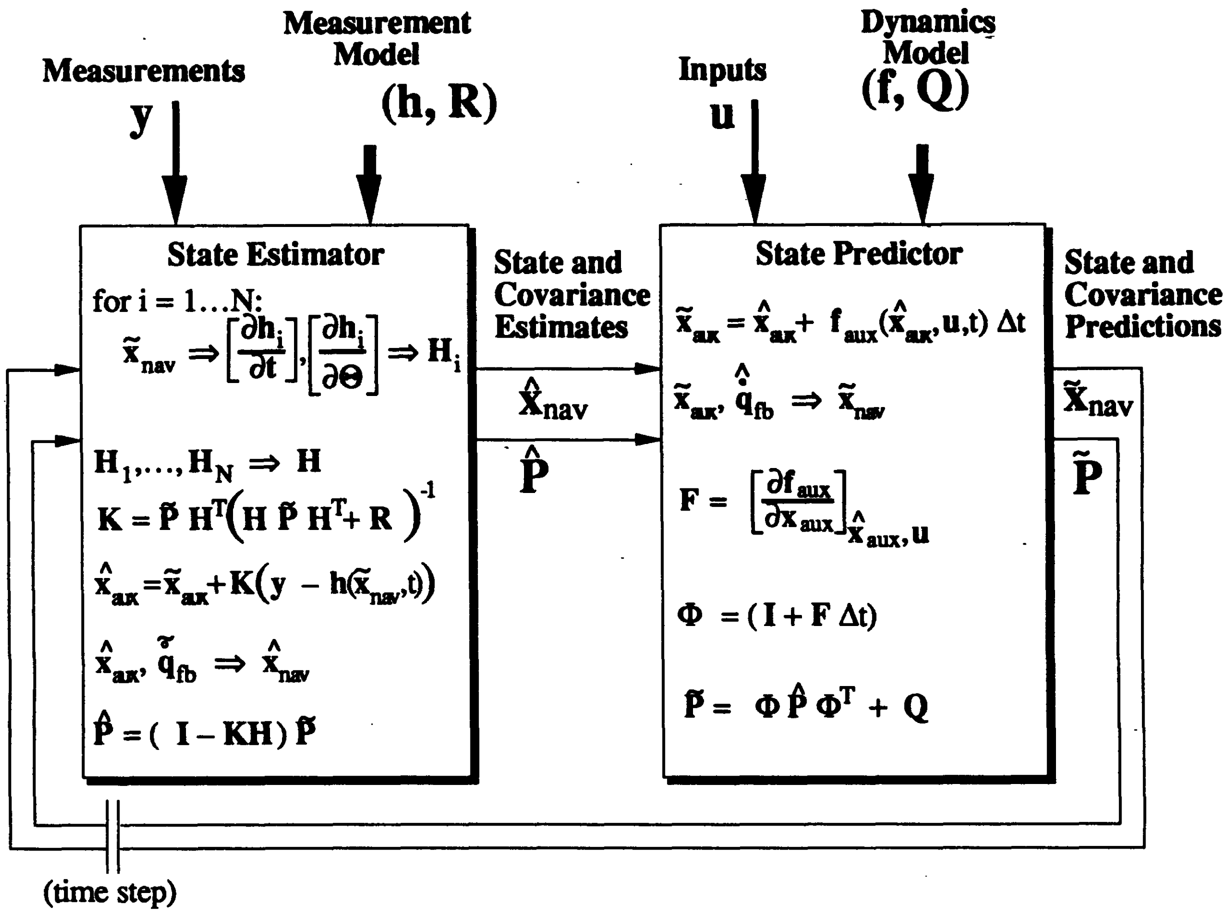

Measurements
y
Measurement Model
(h, R)
Inputs
u
Dynamics Model
(f, Q)
State Estimator
for i = 1...N:
$\tilde{x}_{nav} \Rightarrow \left[\frac{\partial h_i}{\partial t}\right], \left[\frac{\partial h_i}{\partial \Theta}\right] \Rightarrow H_i$
$H_1, \ldots, H_N \Rightarrow H$
$K = \tilde{P} H^T (H \tilde{P} H^T + R)^{-1}$
$\hat{x}_{aux} = \tilde{x}_{aux} + K(y - h(\tilde{x}_{nav}, t))$
$\hat{x}_{aux}, \tilde{q}_{fb} \Rightarrow \hat{x}_{nav}$
$\hat{P} = (I - KH)\tilde{P}$
State and Covariance Estimates
$\hat{x}_{nav}$
$\hat{P}$
State Predictor
$\tilde{x}_{aux} = \hat{x}_{aux} + f_{aux}(\hat{x}_{aux}, u, t)\,\Delta t$
$\tilde{x}_{aux}, \hat{q}_{fb} \Rightarrow \tilde{x}_{nav}$
$F = \left[\frac{\partial f_{aux}}{\partial x_{aux}}\right]_{\hat{x}_{aux}, u}$
$\Phi = (I + F\,\Delta t)$
$\tilde{P} = \Phi \hat{P} \Phi^T + Q$
State and Covariance Predictions
$\tilde{x}_{nav}$
$\tilde{P}$
(time step)

Figure 3.3-4: The recursive estimator for robot vision navigation. The steps involved in state estimation and state prediction at each time step are summarized.

4. Estimator Simulations.

Simulations of the recursive estimator are used to experimentally verify its performance under various conditions of uncertainty in measurements, dynamics, and initial prediction.

A computer program generates an "actual trajectory" for a submersible free-flying robot by applying a pre-selected set of forces and torques to a numerical dynamic model of the robot vehicle. Various parameters of the dynamic model are perturbed from those used in generating the actual trajectory to evaluate the effects of dynamics modelling errors. The program obtains measurements by computing the geometrically correct measurements through the measurement equation and perturbing them randomly, simulating noisy measurement data.

Two types of simulations are run: tracking and convergence. The tracking simulations begin with the state initialized to the actual state, demonstrating the stability of tracking under the various degrees of dynamics and measurement uncertainties. The convergence simulations begin with the state initialized in error, illustrating the convergence of the estimated trajectory to the actual trajectory.

This chapter describes the experimental procedure and the computer routines. Selected results are presented which typify the performance observed over many simulations.

4.1 Hardware and software

The simulations were run on an Apple Macintosh IIx computer equipped with a Motorola 68030 microprocessor and a Motorola 68882 floating-point coprocessor operating at 16 MHz. Computer code was compiled from source code written in the C programming language. The THINK C™ development environment accomodated inclusion of Macintosh ROM trap routines.

The computer code is not optimized for speed. It includes an extensive graphics output which includes a "Camera View" window showing a simulated perspective view of the target plus an "X-Y View" window and "Y-Z View" window which offer orthographic spatial views of the robot actual state and the estimated state.

The orthographic views display the location and attitude of the body frame within the fixed frame from a "bird's eye view" (looking at the XY Plane from above, i.e. along the +Z axis) and from a view facing the target (looking at the YZ Plane from behind, i.e. along the +X axis). The robot actual position is displayed as a triad of orthogonal axes representing the body frame axes. The estimated position is displayed similarly, using a different color for the axes. The orthographic views display orthographic projections of these triads in the appropriate planes as illustrated in Figure 4.1-1.

The graphics display allows the user to monitor the performance of the filter as the estimation trajectory unfolds. For convergence simulations, the initial estimated position begins in error (the user chooses either a specified or random initial error) and the two sets of axis triads gradually converge, illustrating the navigation estimate approaching the actual trajectory. For tracking simulations, the two sets of axes remain more or less on top of one another.

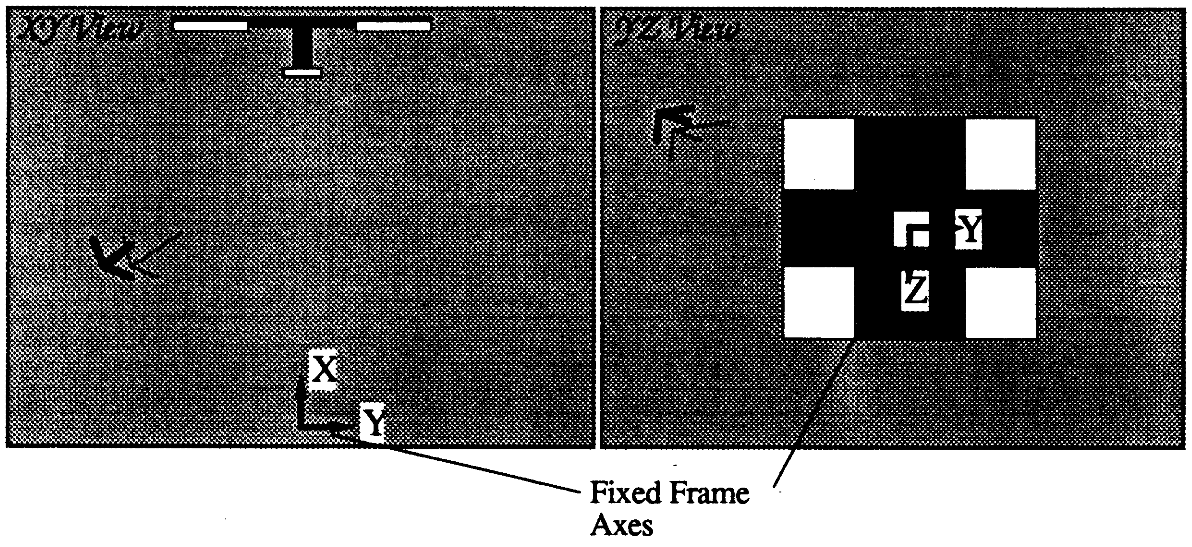


Figure 4.1-1: Illustration of graphics display used to monitor progress of the EKF simulations. The Fixed Frame Axes are labelled. The other orthogonal triads of line segments represent the location and attitude of the Body Frame along the actual trajectory (bold triad) and the estimated trajectory (thin line triad) as orthographically projected onto the XY and YZ Planes. When the state estimate is close enough to the actual state, the two triads coincide. The black-and-white-squares geometrical figure is the experimental navigation target described in Chapter 5 and the simulated navigation target for this simulation. In this picture, the state estimate (thin triad) is in error—it is too low (+Z) and too far to the right (+Y).

Additional numerical information is displayed in the remaining areas of the computer VDT screen to allow monitoring of particular dynamic values of the algorithm as it proceeds. For documentation, the program features a data file option which saves the numerical values of all the estimated and actual state values along the trajectory. The plots of "actual vs. estimated" trajectories found in this chapter were made from these files.

When the navigator is not running, the main program allows configuration of the parameters of the algorithm and configuration of the program. The user can configure the program to add random noise of several levels of variance to the measurements, to the dynamics, and to the initial prediction. The user can also choose the number of time steps (each representing 1/30th sec.) in the simulation, can choose among several pre-programmed trajectories, and can determine whether the data is to be saved or not.

The following code segment constitutes the primary loop of the program when the robot vision navigator simulation, "KalmanVision", is running. Ellipses "⋮" indicates ommissions in the code listing; this is a skeleton of the program. Appendix B contains an extensive source code listing.

```
void kvKalman(int trajmode)
{
        ⋮
    /*      Initialize Kalman Filter Values [==>xp(0),Pp(0),...]
    */
    kvInitVar(trajmode);

    for(kt=0,doneLoop=FALSE;kt<NSteps && doneLoop==FALSE;kt++) { /* EKF LOOP */
        SystemTask();

        /*      Project image to get measurement vector y [xa(t0)==>y(t0)]
        */
        kvPjctIm();

        /*      Run Kalman Filter loop [xp(t0),Pp(t0),y(t0)==>xe(t0),xp(t1),Pp(t1)]
        */
        kvFilter();

        /*      Place Camera View, XY-Plane, and YZ-Plane graphics on screen
                    [   y(t0)  ==> Camera View
                        xa(t0) ==>  Actual locations (green axes) in XY- and YZ-Planes
                        xe(t0) ==>  Estimated locations (bl/pink/yel axes) in XY and YZ]
        */
        kvPlaceGrph();
            ⋮
        /*      If SAVEing, write data to file [xa(t0),xe(t0) ==> FILE]
        */
        if(storeFlag)
            kvStore(fRN);
                ⋮
        /*      Compute next simulation state vector xa(t1)
        */
        kvNextState(kt);
    }
        ⋮
}
```

At the beginning of each navigation sequence, the function `kvInitVar()` initializes the actual state trajectory, the covariance matrices, and the predicted state vector. For tracking, the initial predicted state is the same as the actual; for convergence, a random or prescribed error is added to each state variable. The noise covariances for the dynamics model and the measurement model are prescribed constants.

The function `kvPjctIm()` geometrically projects known 3D feature points of the navigation target onto the image plane according to the camera model, filling the measurement vector `ya[]` with the image coordinates. It adds random noise to these measurements if the option is selected by the user. The feature points are the corners of the white squares for the experimental target, which is the same one used for obtaining measurements from real imagery as described in Chapter 5. (In the later experiments on real imagery, the program is altered so that `kvPjctIm()` is replaced by the function `kvGetMeas()`, which retrieves stored image measurements, filling the measurement vector `ya[]` with actual measurements taken from real imagery.)

The function `kvFilter()` implements one step of the EKF procedure. It utilizes the measurement vector, the predicted state vector, and the predicted error covariance matrix to produce the estimate for the current time step and the predicted state vector and error covariance matrix for the next time step. (Details of `kvFilter()` are discussed below.) The measurement vector `ya[]` comes from `kvPjctIm()`. The predicted state vector `xp[]` and predicted error covariance matrix `Pp[][]` for time step `kt` are produced by `kvFilter()` from the previous time step `kt-1`. For `kt=0` the initialized versions of `xp[]` and `Pp[][]` are used.

The function `kvPlaceGrph()` places the graphics on the screen representing the current actual and estimated states; `kvStore()` writes the current estimated and actual state vectors to a file if the option is selected; and `kvNextState()` generates the "actual" state vector `xa[]` for the next time step by integrating the dynamics over a time step.

The main computational portion of the algorithm takes place in `kvFilter()`. The entire function is listed below. It consists of function calls that implement the major steps of the EKF along with symbolic comments which describe the input and output global variables used by the function.

The functionality is quite self-explanatory, although some symbols need clarification. The global constant `NKF` refers to the number of feature points (N) used in the EKF. The array `ptg` contains the 3D locations of the feature points $\{\mathbf{p}_{i:f}\}$. The array `stm[][]` represents the state transition matrix Φ. All other variables follow the notation used in the development, where the suffix "`p`" indicates predicted values of **x**, **y**, or **P** (normally indicated by tildes "~"), the suffix "`a`" represents actual values generated by the simulation, and the suffix "`e`" indicates estimated values (normally indicated by carets "^").

For more details on the implementations of the EKF steps, see the C source code listed in Appendix B.

```
void kvFilter()
{
    kvLinMeas();                    /* xp,NKF,ptg ==> H */

    kvGain();                       /* Pp,H,R ==> K */

    kvPredMeas();                   /* xp ==> yp */

    kvEstState();                   /* y,yp,K,xp ==> xe */

    kvEstCov();                     /* Pp,K,H,R ==> Pe */

    kvSTM();                        /* xe ==> stm */

    kvPredState();                  /* xe ==> xp */

    kvPredCov();                    /* Pe,stm,Q ==> Pp */
}
```

Using N=4 feature points for measurement, the program can compute slightly more than one time step per second. Discarding the graphics and data saving routines, the program can compute over 2.5 steps per second. Increase in performance to 3 or 4 steps per second appears to be achievable by reprogramming with attention to optimization. The remaining factor of ten increase in speed required to achieve 30 frames per second appears to be possible by transporting the code to a faster dedicated processor system and further code optimization. Certainly special purpose architectures can accomodate more than this quantity of computation at sufficient rates with the power of one or two modern arithmetic processors.

4.2 Experimental Procedure and Results

The general goals of the experiments with this computer simulation include

(1) verification of the functionality of the EKF with the vision measurement equation,

(2) examination of the tracking performance of the estimator in the presence of measurement noise and errors in the dynamic model, and

(3) examination of the convergence of the estimator from initial prediction errors.

The procedure for (1) involved implementing and debugging the computer code and using the graphics display to verify tracking and convergence from initial errors. The procedure for (2) and (3) included collecting trajectory data for a variety of system parameters and analyzing the performance.

The tracking and convergence simulations were meant to expose the limits of noise, model uncertainty, and initial condition errors that the system could withstand and still manage to track the trajectory of the robot accurately. A positive result of the experiments was that no meaningful limits exist for these quantities because the estimator was stable and convergent within an operating envelope far exceeding what could be tolerated by other elements of the system.

For example, the system converged to the actual trajectory from an initial condition which had the camera pointed nearly 90° away from the direction of the target. The simulation succeeded because its measurements are generated by geometric projection, but a real system could never tolerate such an initial condition because the target would not appear in the image and no measurements could be taken! Hence, the tolerance on initial conditions errors is not limited by the estimation, but by other considerations.

Likewise, measurement noise was tested only to a certain extent. The system refused to fail catastrophically for repeated simulations with measurement noises having amplitudes of ten pixels or more. Estimation trajectories degraded progressively with higher levels of noise but remained convergent and stable. Since measurements can typically be taken to subpixel accuracy, as will be shown in Chapter 5, it is meaningless to simulate the navigator with ever increasing magnitudes of noise beyond what is already unreasonable to expect.

Thus, the experimental results met and exceeded all requirements. Potential failures are not expected from the estimation routine itself but rather from catastrophic errors in other parts of the system. Failures occured only for unrealistic circumstances, such as the camera pointing away from the target.

Hence, performance limits turn out not to be the issue. The important results include some characterization of how the trajectory estimation degrades with increased uncertainty in the measurements and models. The results that follow demonstrate the modest degradation of accuracy with increased measurement and dynamics uncertainty. The results are presented in two parts. First, pure tracking is demonstrated, with the state vector initialized to the actual state. Next, the state vector is initialized with substantial errors to demonstrate the convergence of the estimation to the actual trajectory.

4.2.1 Tracking.

Tracking experiments begin with the estimated state initialized to the actual state. The progress of the estimated trajectory is monitored over several seconds (30 time steps per second) to get qualitative and quantitative data on the ability of the estimator to track the trajectory.

The following sections illustrate the degradation of tracking as measurement and dynamics noise are increased from zero to very large amounts. Section 4.2.1.1 shows the effect of raising measurement noise in the absence of dynamics noise. Section 4.2.1.2 does the opposite, increasing the dynamics noise at a constant measurement noise. The last section, §4.2.1.3, illustrates the typical performance of the estimator with a substantial level of both measurement and dynamics noise.

The trajectory used in all the following simulations consists primarily of a forward velocity and a rolling velocity. The robot starts to the left of center of the target with a positive yaw angle in a position similar to the one depicted in Figure 4.1-1. A constant forward thrust and constant rolling moment around the X_b-axis cause the robot to move toward the target and roll. The initial state and the constant inputs used to generate the true trajectory are shown in the table of Figure 4.2.1-1. The actual trajectory varies when dynamic "noise" is added.

State Variables	Initial Condition	Control Inputs	Value
$t_{fb:f1}$	1.0 m	$\mathcal{F}_{:f1}$	50 N
$t_{fb:f2}$	−1.7 m	$\mathcal{F}_{:f2}$	20 N
$t_{fb:f3}$	0.35 m	$\mathcal{F}_{:f3}$	−15 N
q_{fb0}	0.9659	$\mathcal{T}_{:b1}$	50 N•m
q_{fb1}	0.0	$\mathcal{T}_{:b2}$	0 N•m
q_{fb2}	0.0	$\mathcal{T}_{:b3}$	0 N•m
q_{fb3}	0.2588		
$v_{fb:f1}$	0.2 m/s		
$v_{fb:f2}$	0.1 m/s		
$v_{fb:f3}$	0.0 m/s		
$w_{fb:b1}$	0.8 rad/s		
$w_{fb:b2}$	0.0 rad/s		
$w_{fb:b3}$	0.0 rad/s		

Figure 4.2.1-1: Initial conditions for trajectories used in tracking simulations.

For the simulation results presented in this section and the next, the various levels of random noise added to the measurements are designated by the five measurement conditions M0...M4. Condition M0 designates no added noise; M1...M4 represent increasingly high levels of random noise. The noise variances and corresponding pixel spread for uniform distributed random noise are shown in Figure 4.2.1-2.

Measurement Condition	Variance	Spread
M0	0	0 pixels
M1	0.0833×10^{-8}	0.5
M2	1.3333×10^{-8}	2
M3	8.3333×10^{-8}	5
M4	33.3333×10^{-8}	10

Figure 4.2.1-2: Definition of Measurement Conditions. Apply the conversion factor 10,000 pixels/meter. The variance and spread refer to the uniformly distributed random noise added to each measurement.

Similarly, the dynamic modelling errors used in the simulations are designated using the measurement conditions D0...D5. For each case, the internal dynamics model remains the same. The actual dynamics are altered by adding some level of uniformly distributed random noise to the accelerations and by utilizing parameters for the mass of the vehicle and drag coefficients that differ from those in the internal model. Figure 4.2.1-3 shows the noise variance and spread in m/s^2 of the acceleration noise, the actual mass, and the actual drag coefficients. The latter two parameters are expressed relative to M and C, the mass and coefficient of the internal model.

The condition D0 represents a perfect model. Conditions D1...D4 represent dynamics models with increasing acceleration noise, increased inertia, and increased drag. The last condition, D5, uses a high level of noise but the inertia and drag assumed by the model are lower than the actual vehicle.

Dynamics Condition	Variance	Spread (m/s^2)	Mass	Drag Coefficients
D0	0	0	M	C
D1	0.1×10^{-4}	0.0055	M	2 C
D2	1.0×10^{-4}	0.0173	2 M	8 C
D3	2.0×10^{-4}	0.0245	3 M	18 C
D4	4.0×10^{-4}	0.0346	4 M	32 C
D5	4.0×10^{-4}	0.0346	0.25 M	0.125 C

Figure 4.2.1-3: Definition of Dynamics Conditions. The variance and spread refer to uniformly distributed random noise added to modelled accelerations. The mass and drag coefficients are those used as parameters in the dynamics model.

The following table indicates which conditions are active for the simulation results shown in this section. The top row of the table are conditions used in the first section to demonstrate effects of measurement noise alone. The remainder of the second column are conditions used to demonstrate dynamics uncertainties alone. The remaining condition (M2,D5) is used in subsection 4.2.1.3 to show tracking for all thirteen state variables. The measurement and dynamics conditions used for each results are also listed at the top of each graph.

	M0	M1	M2	M3	M4
D0	---	Figure 4.2.1.1-1	Figure 4.2.1.1-2	Figure 4.2.1.1-3	Figure 4.2.1.1-4
D1	---	Figure 4.2.1.2-1	---	---	---
D2	---	Figure 4.2.1.2-2	---	---	---
D3	---	Figure 4.2.1.2-3	---	---	---
D4	---	Figure 4.2.1.2-4	---	---	---
D5	---	---	Figure 4.2.1.3-1	---	---

4.2.1.1 Measurement Errors

Presentation of the results for measurement errors is intended to give the reader a feeling for the effect of raising the level of measurement errors with everything else remaining constant. Condition D0 is used for the dynamics, meaning that the dynamic model is identical to the actual dynamics of the robot. Hence, there should be no errors added by the predictions.

The following four figures depict the actual and estimated trajectories for two state variables, $t_{fb:f1}$ and its derivative $v_{fb:f1}$. The other eleven trajectories are omitted for brevity. These two trajectories are typical of the degradation of performance in the other variables. All thirteen state variables trajectories are presented for the last simulation of Section 4.2.1.

In these simulations, four feature points are used. The measurement noise covariance matrix **R** is set to reflect the amount of noise actually being added to the measurements. Specifically, the covariance matrix is initialized to

$$\mathbf{R} = \sigma^2 \mathbf{I}$$

where the variance σ^2 is set to the same value of variance used in generating the noise. The difference is that the noise added to the measurements are uniformly distributed and the estimator assumes that they are Gaussian distributed.

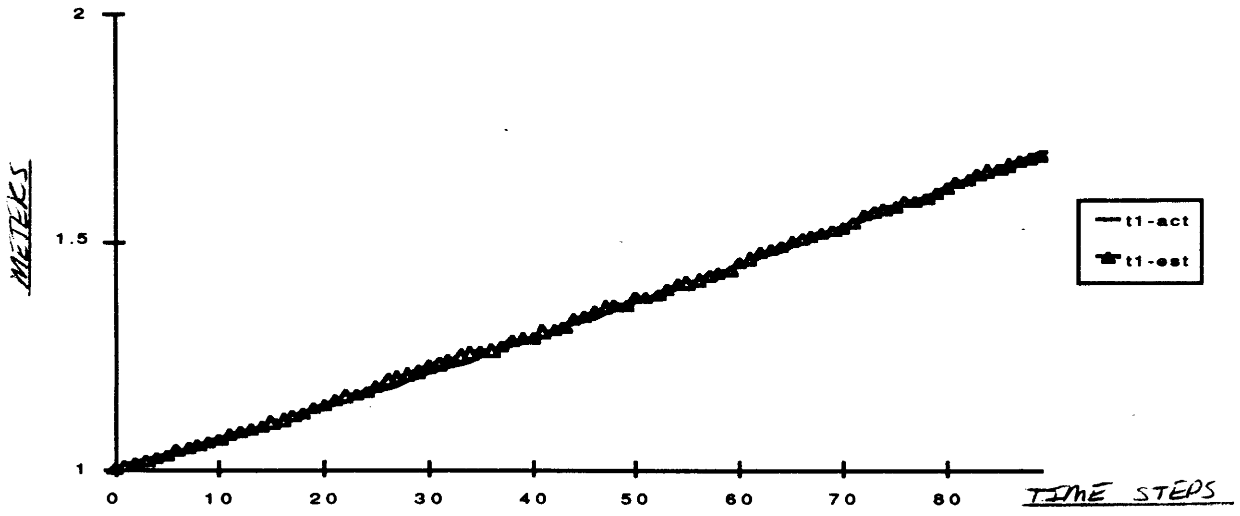
The results show little degradation in tracking performance as the measurement noise is amplified. Also, the position tracking is noticeably smoother than the velocity tracking.

(In the graphs of this chapter, an abbreviated notation is used for the state variables in which t_i replaces $t_{fb:fi}$, q_i replaces q_{fbi}, v_i replaces $v_{fb:fi}$, and w_i or ω_i replaces $w_{fb:bi}$.)

Tracking Simulation
Conditions:

M1 ($\sigma^2 = 0.0833 \times 10^{-8}$, ± 0.5 pixels uniformly distributed error)

D0 (exact dynamic model)



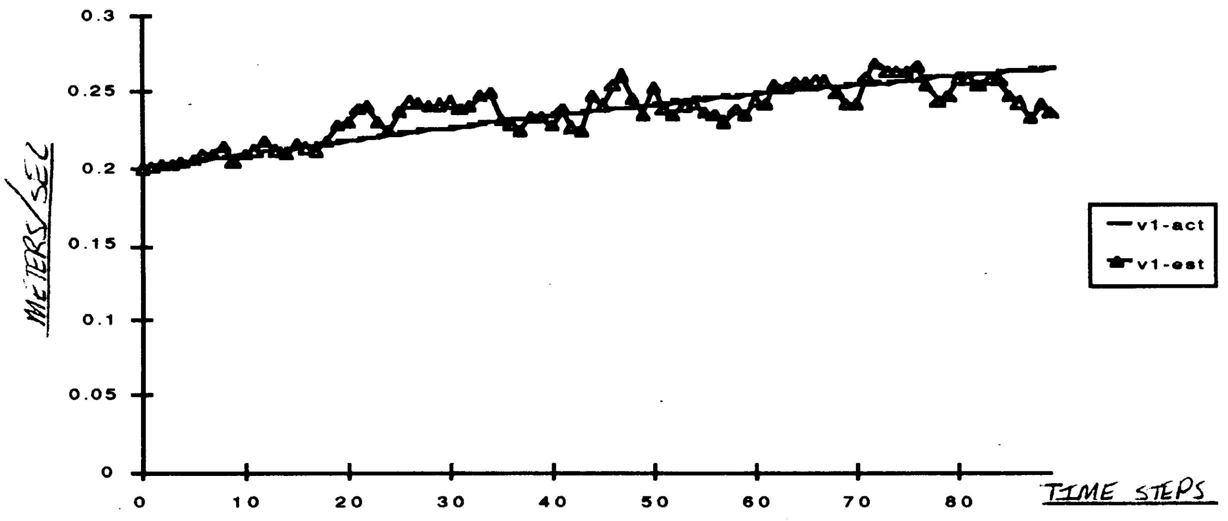
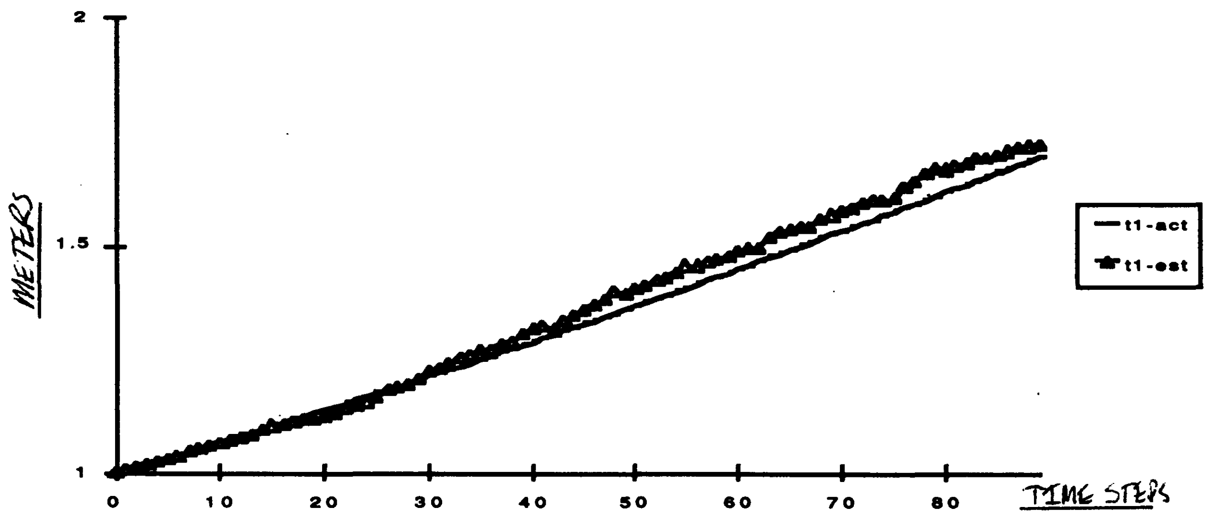


Figure 4.2.1.1-1: Simulated actual and estimated trajectories for state variables $t_{fb:f1}$ (meters) and $v_{fb:f1}$ (meters/second) versus time step index.

Tracking Simulation
Conditions:

M2 ($\sigma^2 = 1.3333 \times 10^{-8}$, ±2 pixels uniformly distributed error)

D0 (exact dynamic model)



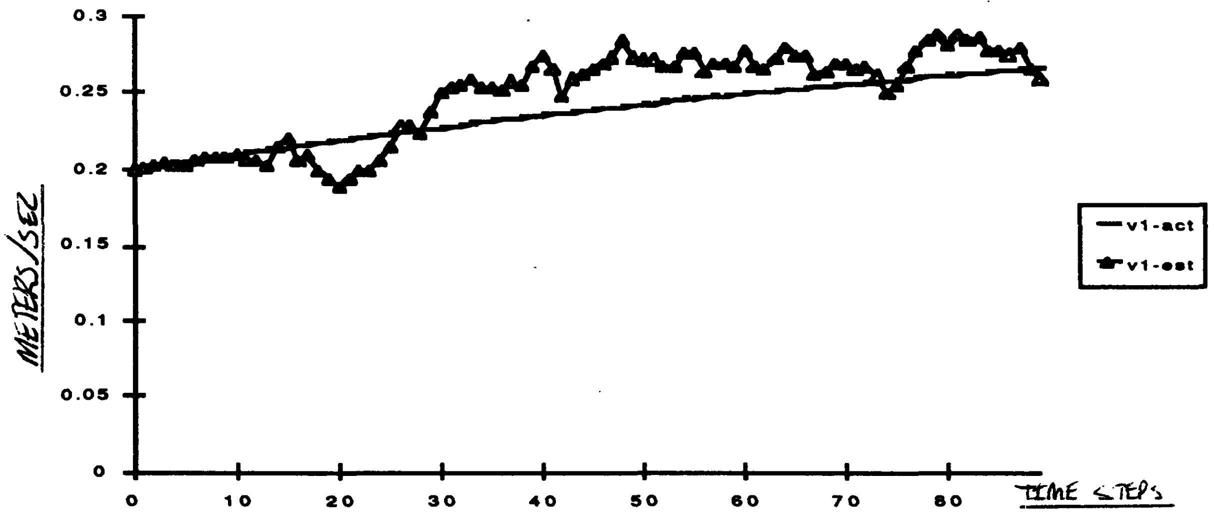
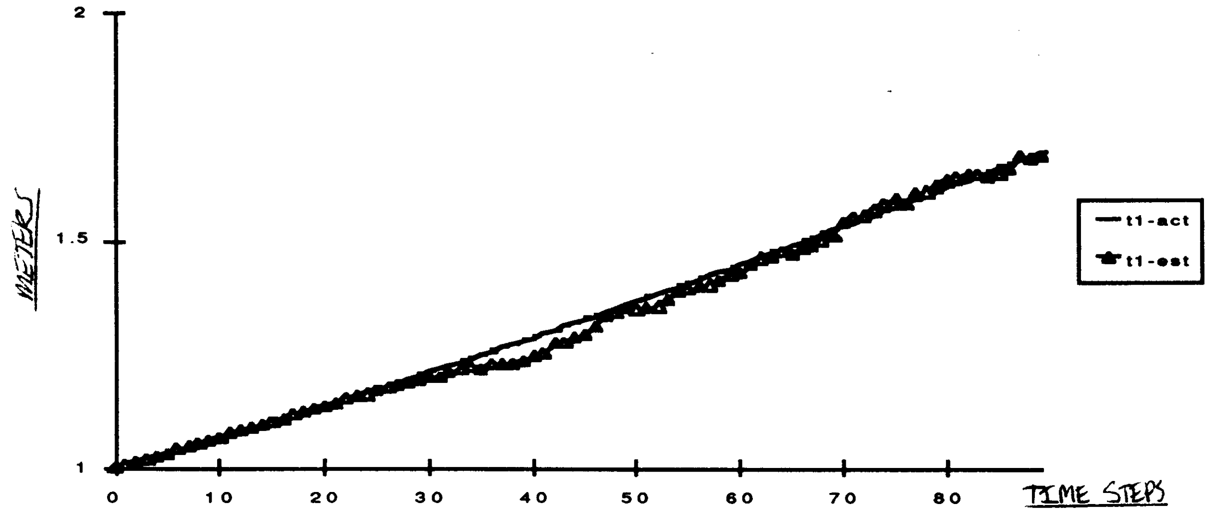


Figure 4.2.1.1-2: Simulated actual and estimated trajectories for state variables $t_{fb:f1}$ (meters) and $v_{fb:f1}$ (meters/second).

Tracking Simulation Conditions:

M3 ($\sigma^2 = 8.3333 \times 10^{-8}$, ±5 pixels uniformly distributed error)

D0 (exact dynamic model)



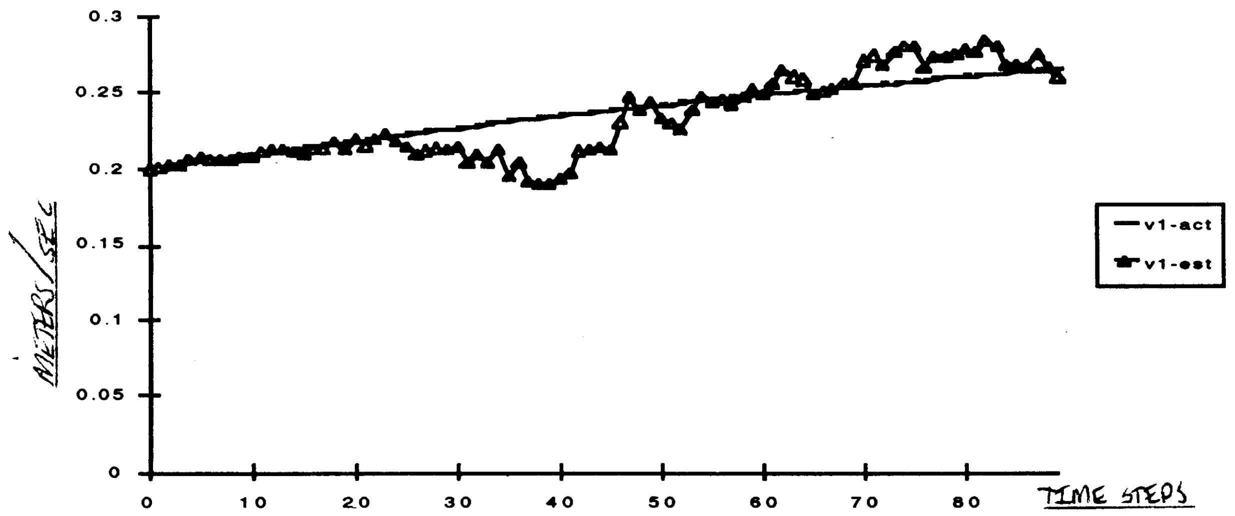
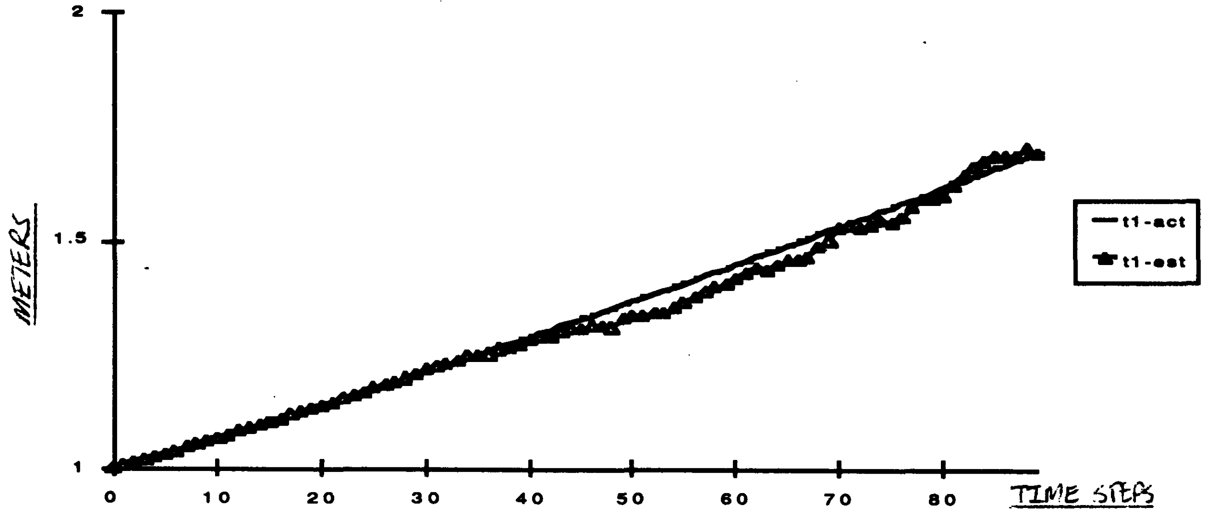


Figure 4.2.1.1-3: Simulated actual and estimated trajectories for state variables $t_{fb:f1}$ (meters) and $v_{fb:f1}$ (meters/second).

Tracking Simulation
Conditions:

M4 ($\sigma^2 = 33.3333 \times 10^{-8}$, ±10 pixels uniformly distributed error)

D0 (exact dynamic model)



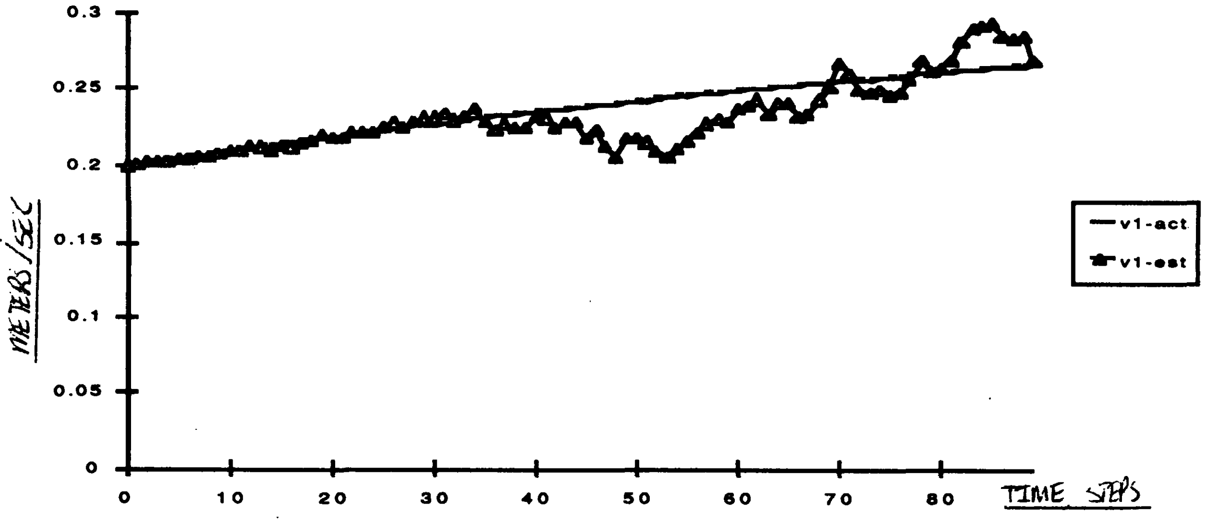


Figure 4.2.1.1-4: Simulated actual and estimated trajectories for state variables $t_{fb:f1}$ (meters) and $v_{fb:f1}$ (meters/second).

4.2.1.2 Dynamics Errors

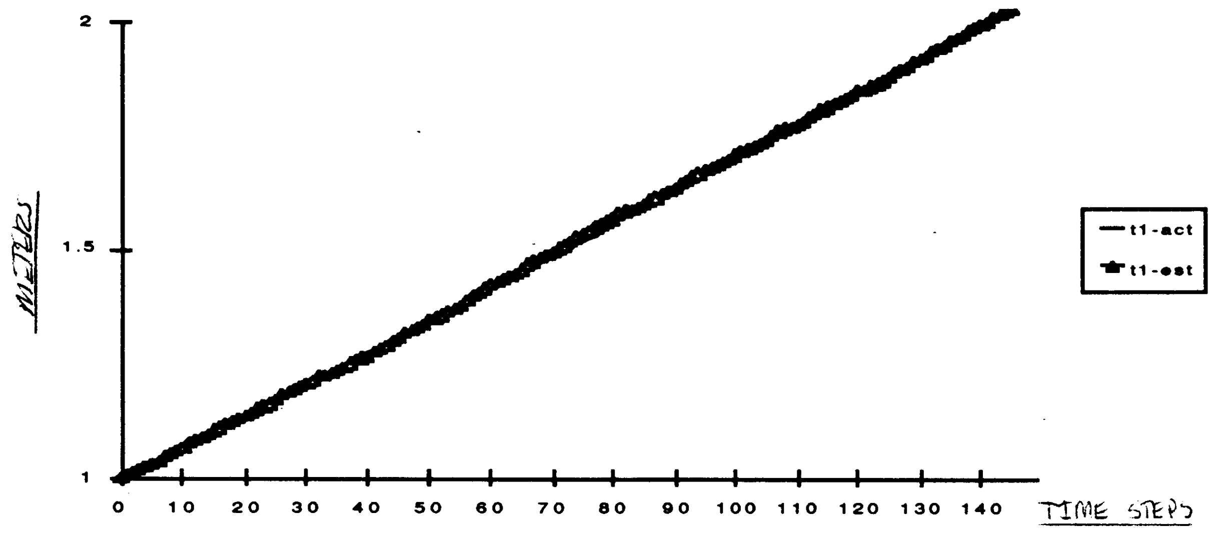
Similar data to those presented in the previous section appears in in the following four figures. The dynamic condition is degraded from D1 to D4 and the measurement condition is M1. Notice that the actual trajectories get shallower as the condition is changed from D1 to D4 because of the increased inertia and higher viscocity introduced.

As the actual dynamics move further away from the modelled dynamics, the velocity estimate degrades so that it has error peaks of nearly 10 cm/s, but the position estimate remains within a couple centimeters of the actual trajectory. This is partly because the measurements taken for vision navigation give no direct information about velocities.

Tracking Simulation
Conditions:

M1 ($\sigma^2 = 0.0833 \times 10^{-8}$, ±0.5pixels uniformly distributed error)

D1 ($\sigma^2 = 0.1 \times 10^{-4}$, $M_{act} = M_{model}$, $C_{act} = 2C_{model}$)



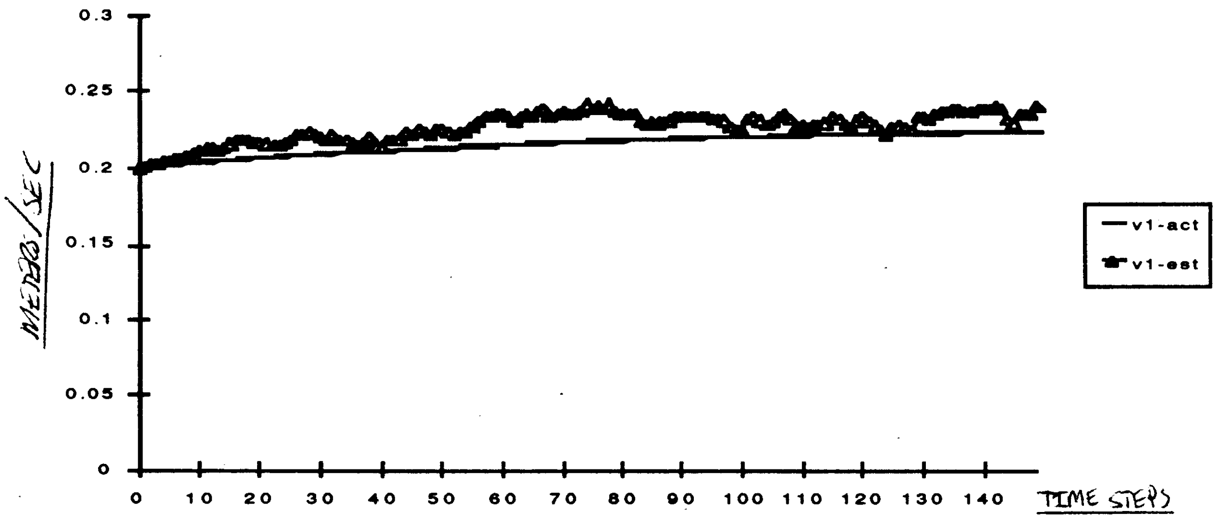
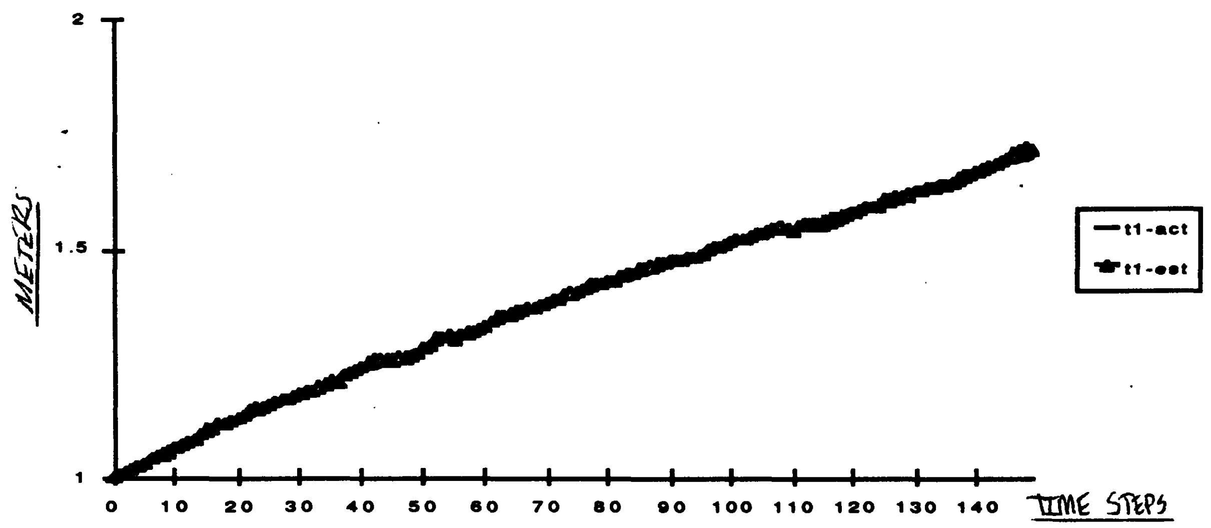


Figure 4.2.1.2-1: Simulated actual and estimated trajectories for state variables $t_{fb:f1}$ (meters) and $v_{fb:f1}$ (meters/second).

Tracking Simulation
Conditions:

M1 ($\sigma^2 = 0.0833 \times 10^{-8}$, ± 0.5 pixels uniformly distributed error)

D2 ($\sigma^2 = 1.0 \times 10^{-4}$, $M_{act} = 2\ M_{model}$, $C_{act} = 8\ C_{model}$)



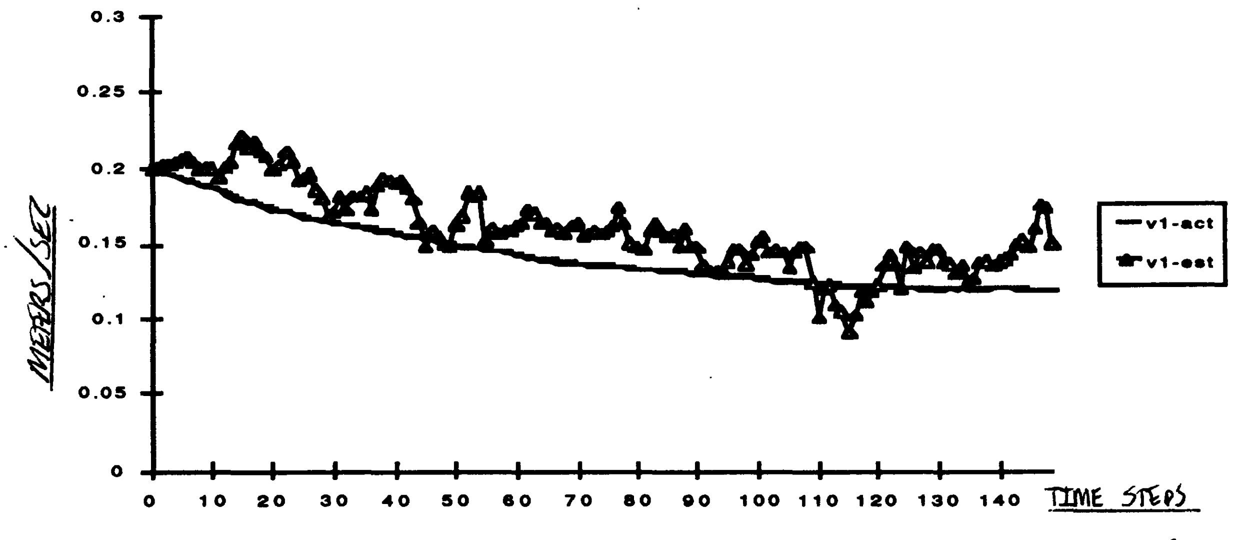
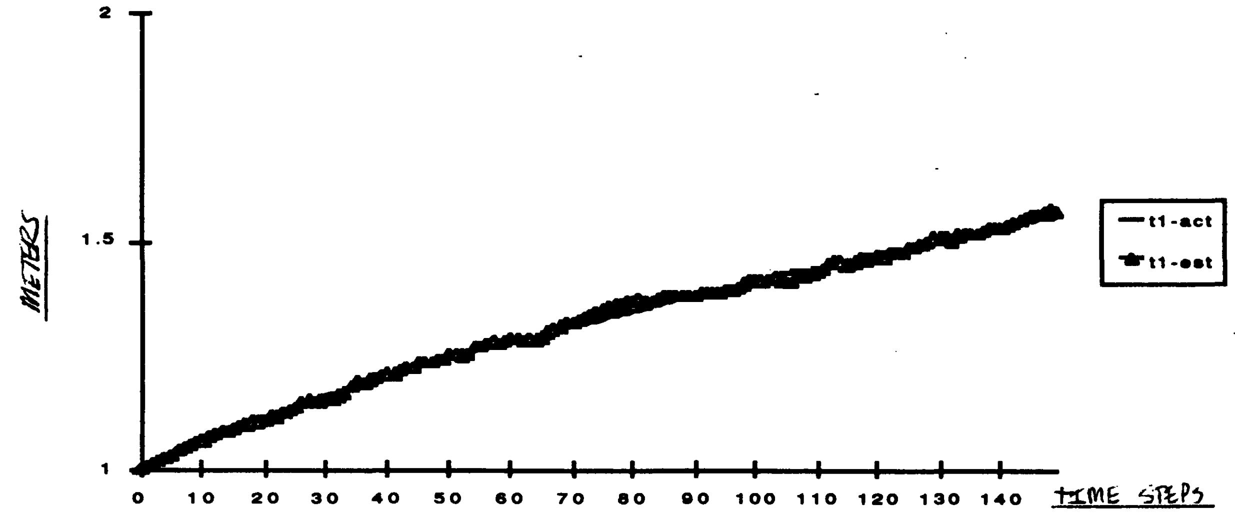


Figure 4.2.1.2-2: Simulated actual and estimated trajectories for state variables $t_{fb:f1}$ (meters) and $v_{fb:f1}$ (meters/second).

Tracking Simulation
Conditions:

M1 ($\sigma^2 = 0.0833 \times 10^{-8}$, ± 0.5 pixels uniformly distributed error)

D3 ($\sigma^2 = 2.0 \times 10^{-4}$, $M_{act} = 3\ M_{model}$, $C_{act} = 18\ C_{model}$)



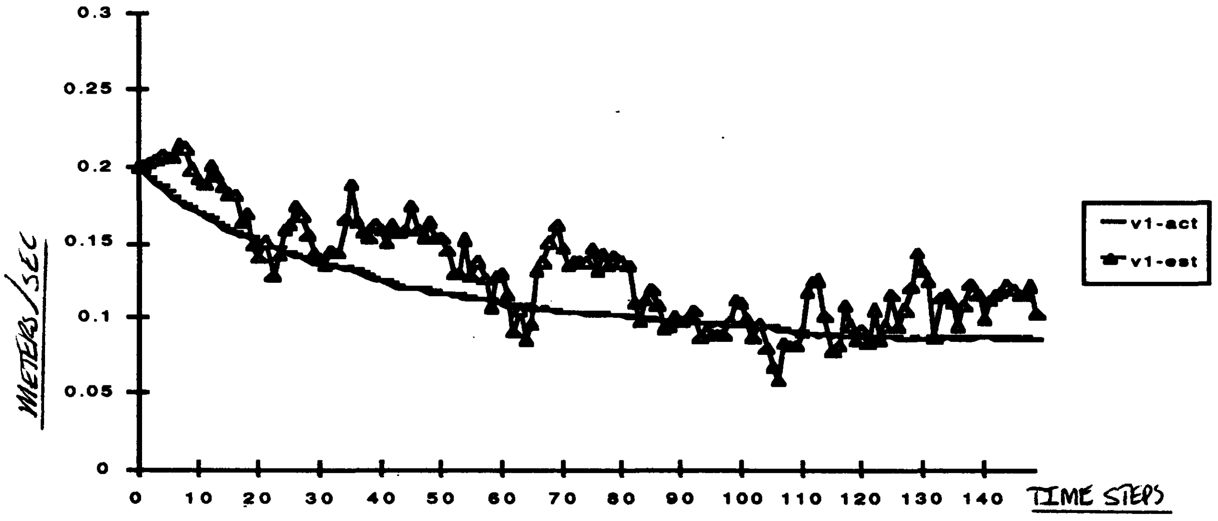
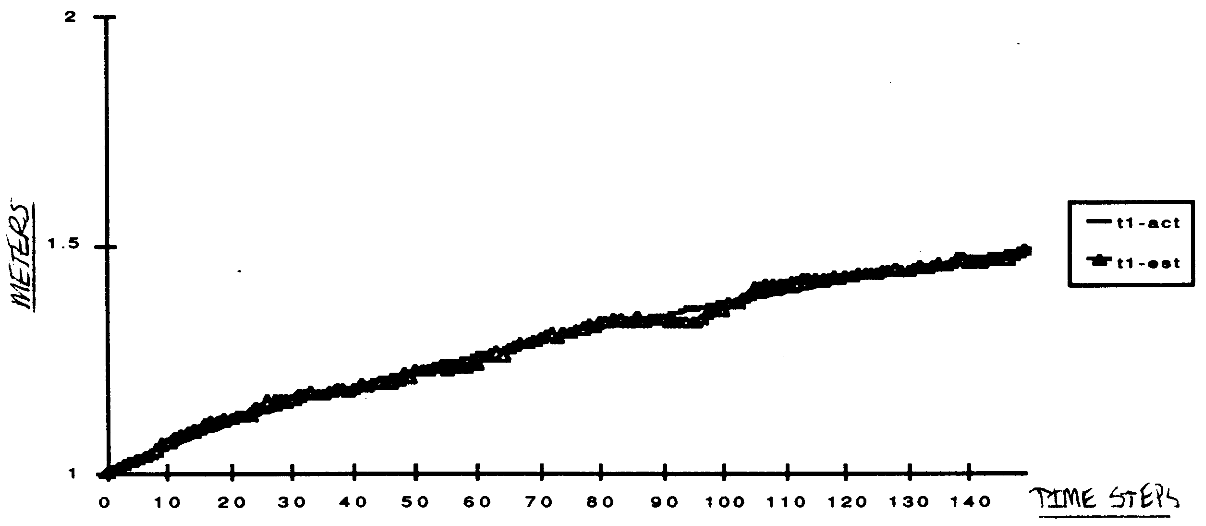


Figure 4.2.1.2-3: Simulated actual and estimated trajectories for state variables $t_{fb:f1}$ (meters) and $v_{fb:f1}$ (meters/second).

Tracking Simulation
Conditions:

M1 ($\sigma^2 = 0.0833 \times 10^{-8}$, ± 0.5 pixels uniformly distributed error)

D4 ($\sigma^2 = 4.0 \times 10^{-4}$, $M_{act} = 4\ M_{model}$, $C_{act} = 32\ C_{model}$)



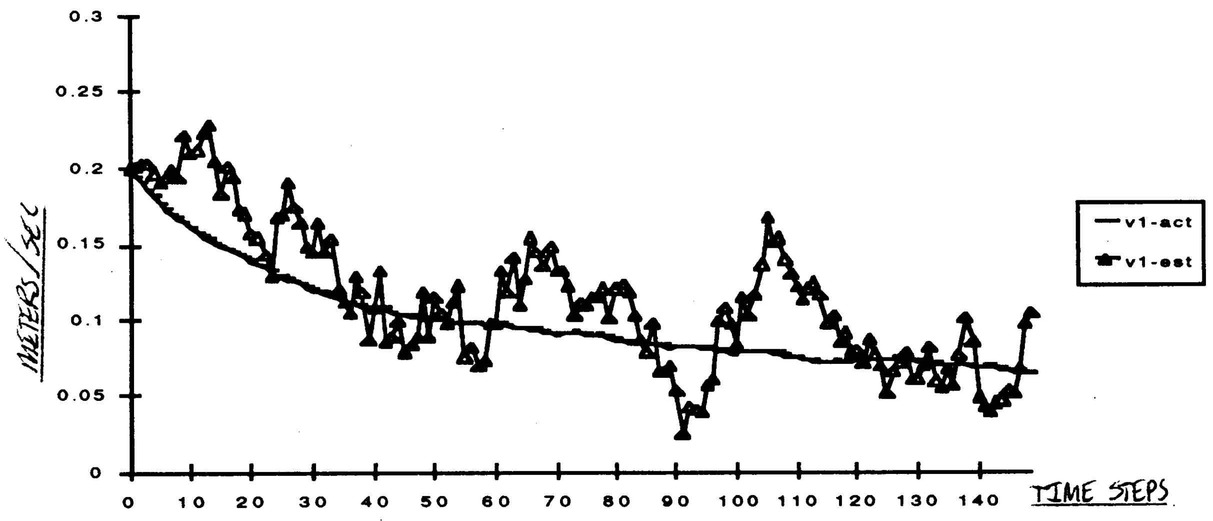


Figure 4.2.1.2-4: Simulated actual and estimated trajectories for state variables $t_{fb:f1}$ (meters) and $v_{fb:f1}$ (meters/second).

4.2.1.3 Measurement and Dynamics Errors

The last illustration of tracking performance demonstrates the performance of the estimator under substantial uncertainty in both the measurements and the dynamics for all thirteen state variables.

The following figure shows the results of a simulation using measurment condition M1 and dynamics condition D5. The measurement condition corresponds to errors as large as 2 pixels in each coordinate of the measurements. The dynamics condition represents noisy accelerations and an actual robot which is lighter and experiences less drag than the internal model suggests.

Tracking Simulation

M2 ($\sigma^2 = 1.3333 \times 10^{-8}$, ±2 pixels uniformly distributed error)

D5 ($\sigma^2 = 4.0 \times 10^{-4}$, $M_{act} = 0.25\, M_{model}$, $C_{act} = 0.125\, C_{model}$)

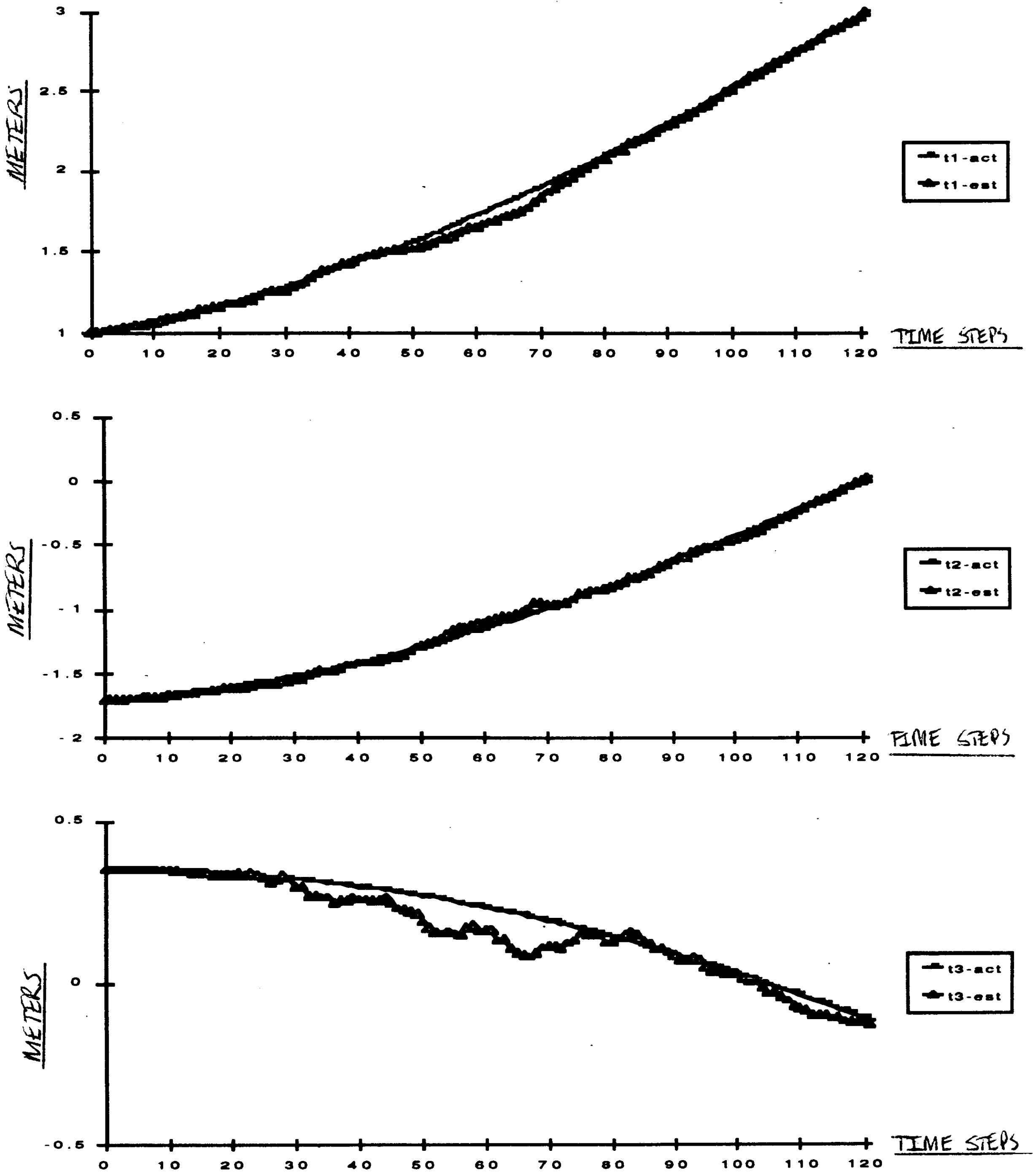
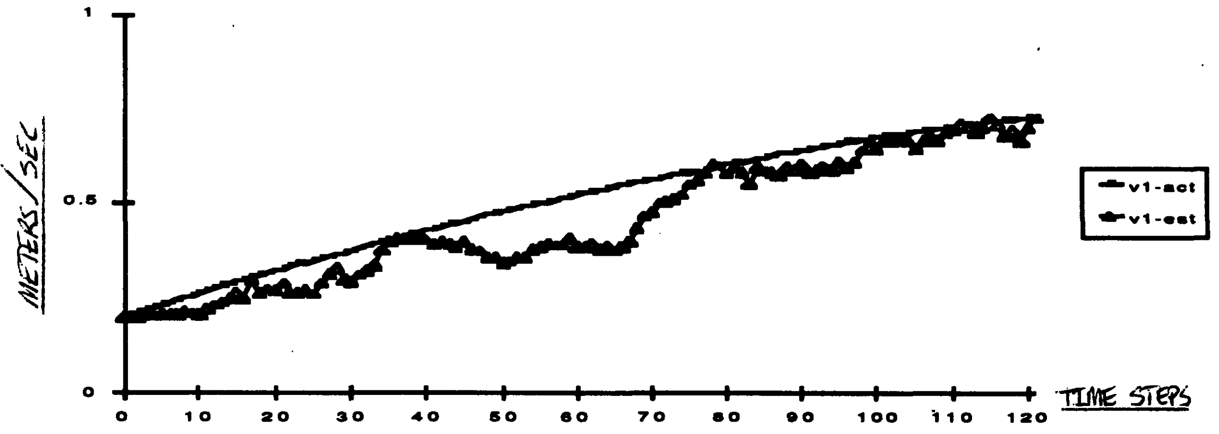
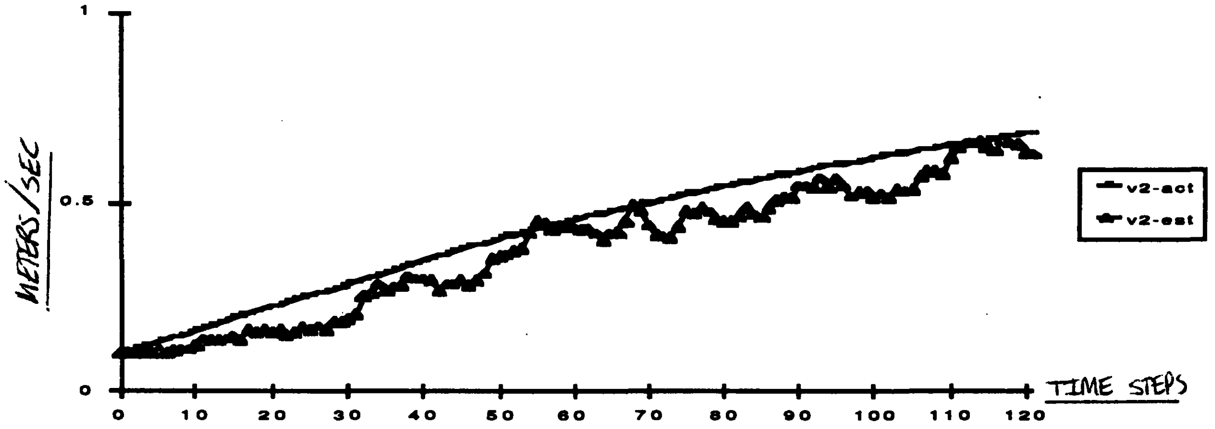


Figure 4.2.1.3-1(a,b,c): Simulated actual and estimated trajectories for state variables $t_{fb:f1}$, $t_{fb:f2}$, and $t_{fb:f3}$ (meters).

...Tracking Simulation (M2,D5), continued - VELOCITY...





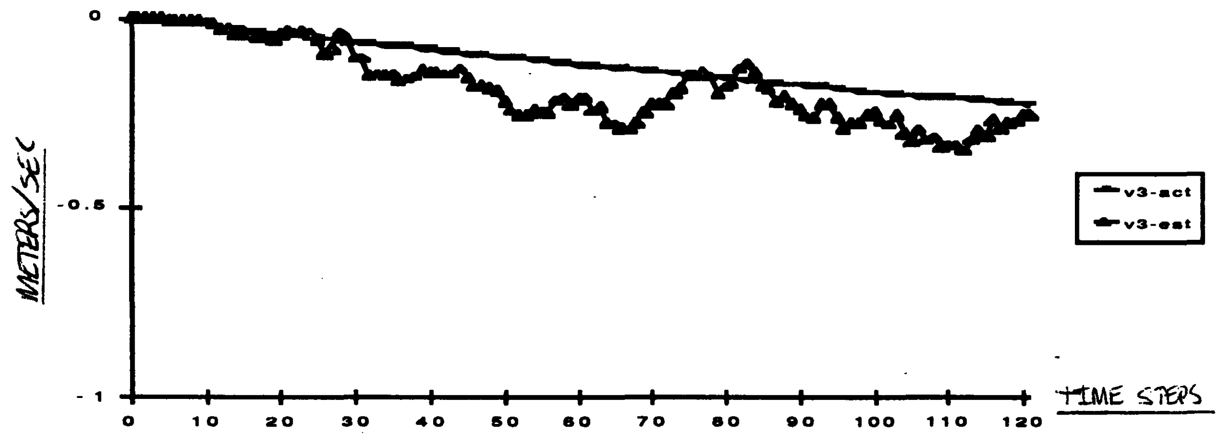
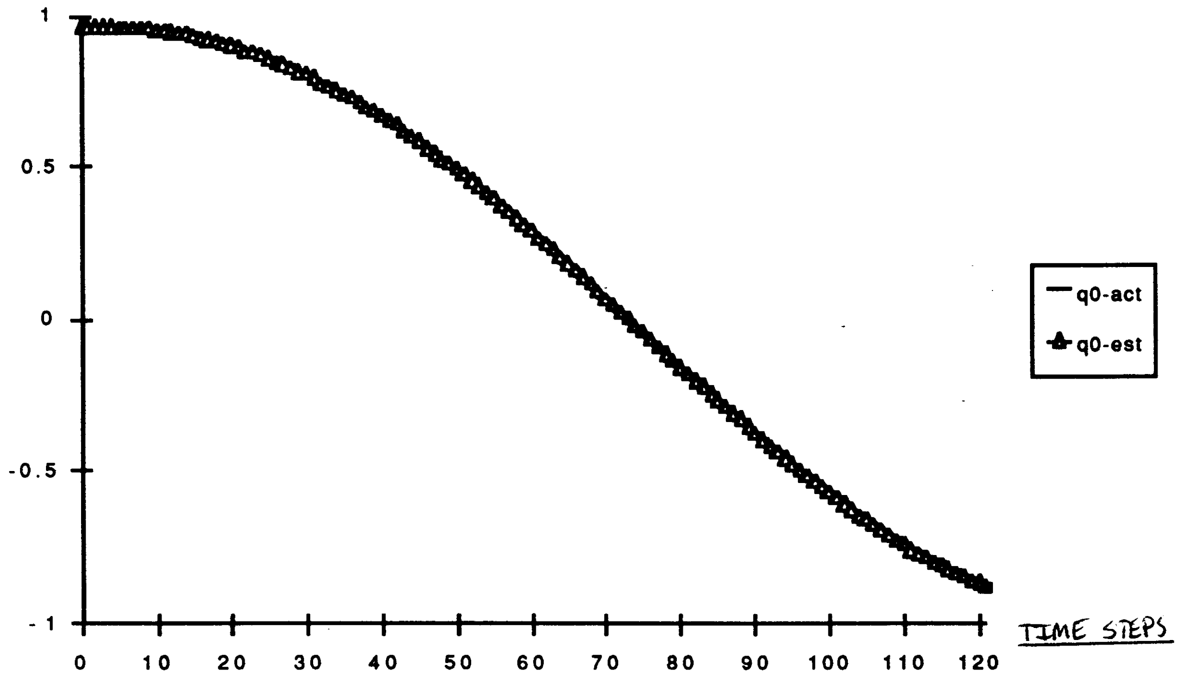


Figure 4.2.1.3-1(d,e,f): Simulated actual and estimated trajectories for state variables $v_{fb:f1}$, $v_{fb:f4}$, and $v_{fb:f3}$ (meters/second).

...Tracking Simulation (M2,D5), continued - ATTITUDE...



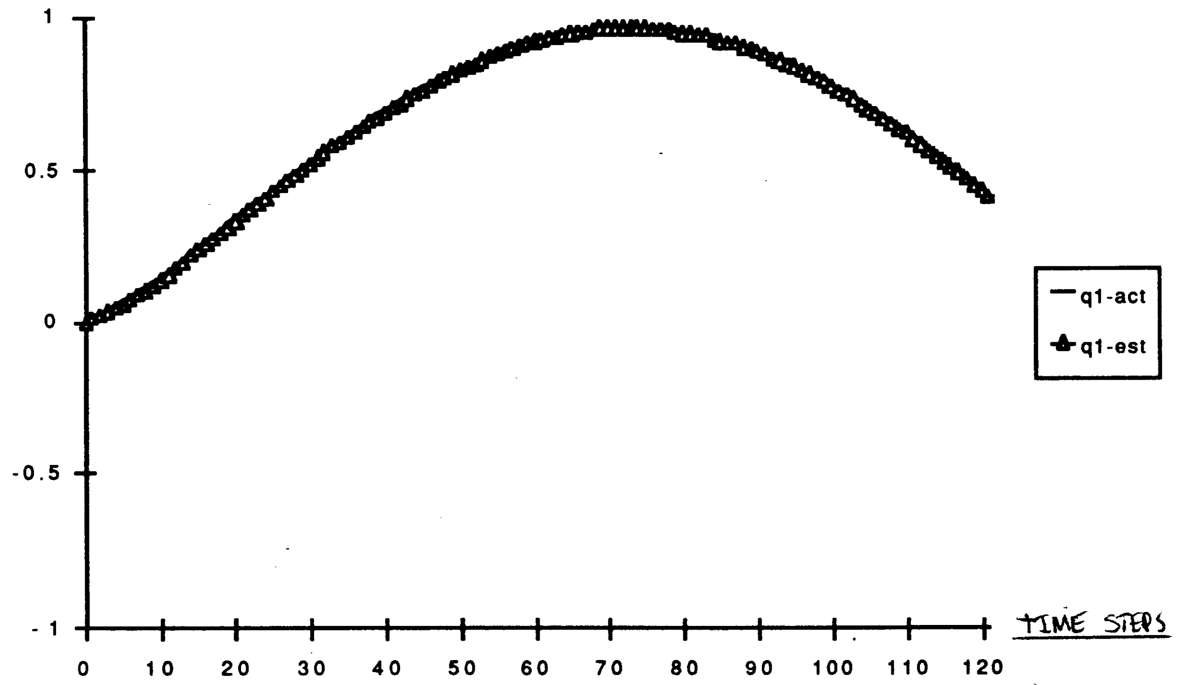
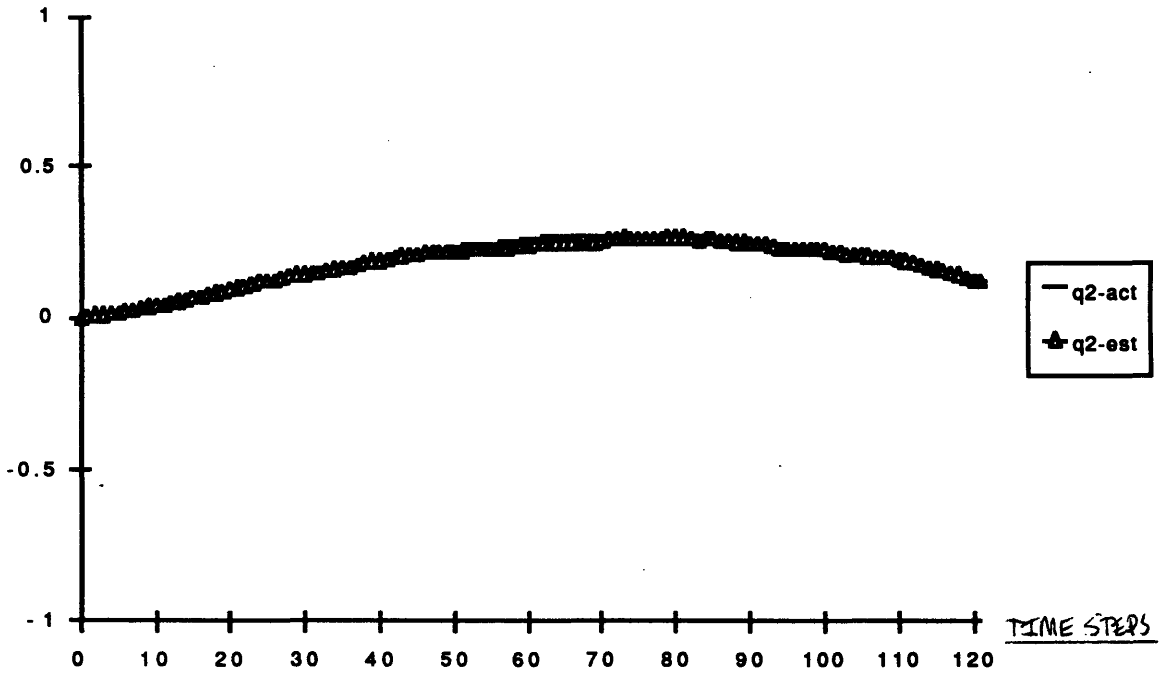


Figure 4.2.1.3-1(g,h): Simulated actual and estimated trajectories for state variables q_{fb0} and q_{fb1}.

...Tracking Simulation (M2,D5), continued - ATTITUDE...



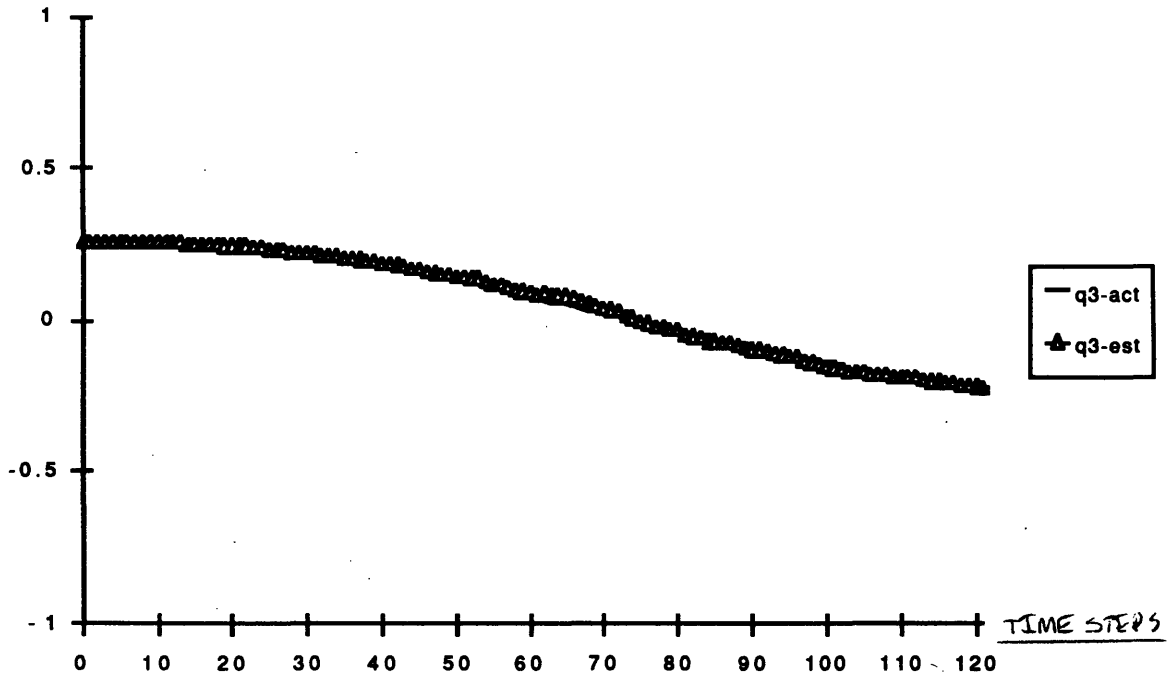
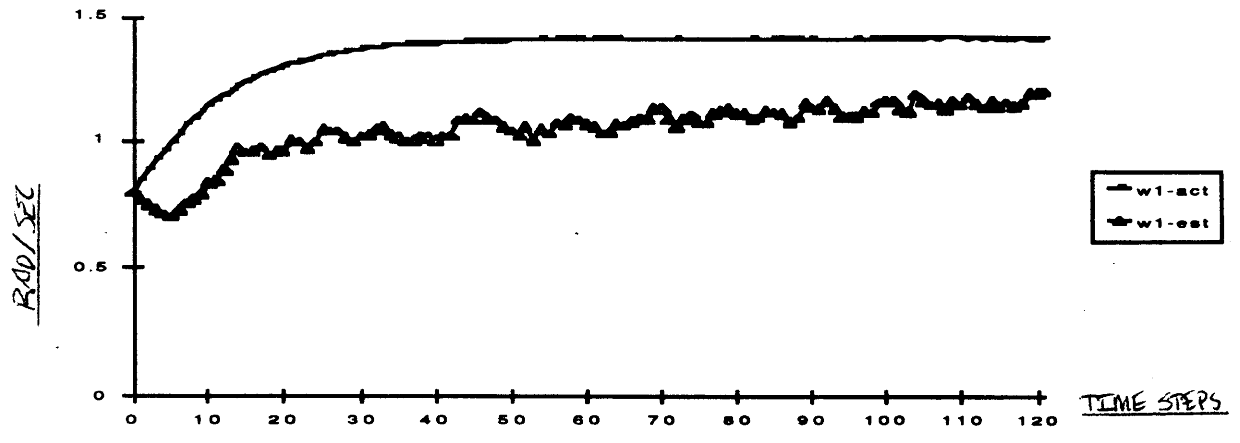
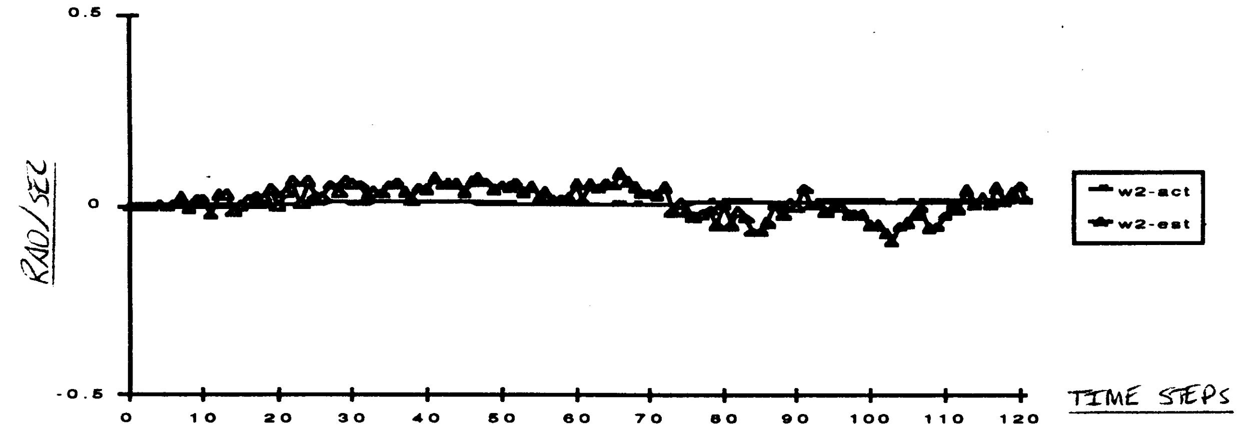


Figure 4.2.1.3-1(i,j): Simulated actual and estimated trajectories for state variables q_{fb2}, and q_{fb3}.

...Tracking Simulation (M2,D5),continued - ANGULAR VELOCITY.





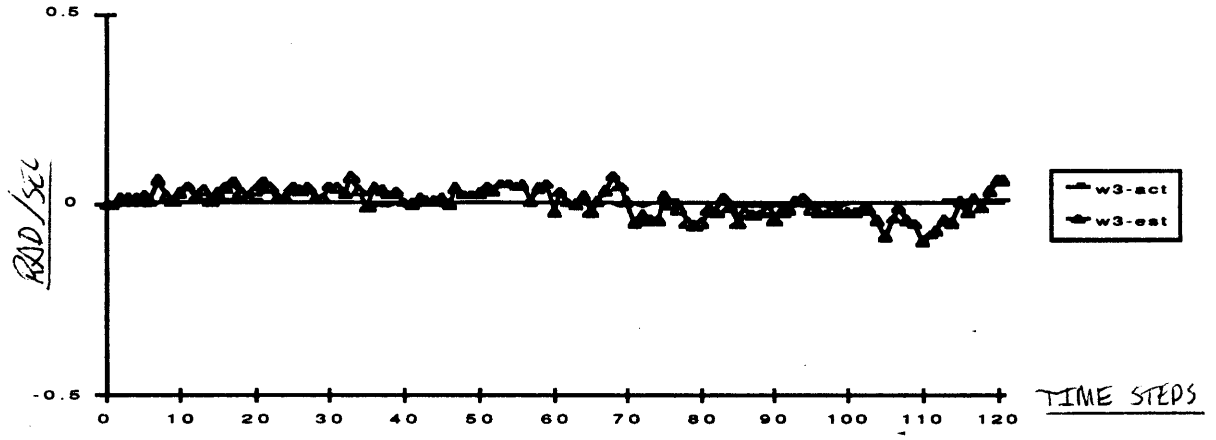


Figure 4.2.1.3-1(k,l,m): Simulated actual and estimated trajectories for state variables $w_{fb:b1}$, $w_{fb:b2}$ and $w_{fb:b3}$ (rad/sec).

4.2.2 Convergence from Initial Prediction Errors.

In the previous section, the estimation trajectories began at the same state as the actual trajectory. Those experiments were useful for visualizing the stability of the estimator under adverse measurement and modelling conditions. This section presents the results of a typical simulation in which the estimator begins in error.

All thirteen state variable trajectories are presented in the following figure. The conditions M2 and D5 are used. The trajectories show good convergence in the position and attitude state variables and convergent but noisy behavior in the velocity variables.

Convergence Simulation

M2 ($\sigma^2 = 1.3333 \times 10^{-8}$, ±2 pixels uniformly distributed error)

D5 ($\sigma^2 = 4.0 \times 10^{-4}$, $M_{model} = 0.25\ M_{act}$, $C_{model} = 0.125\ C_{act}$)

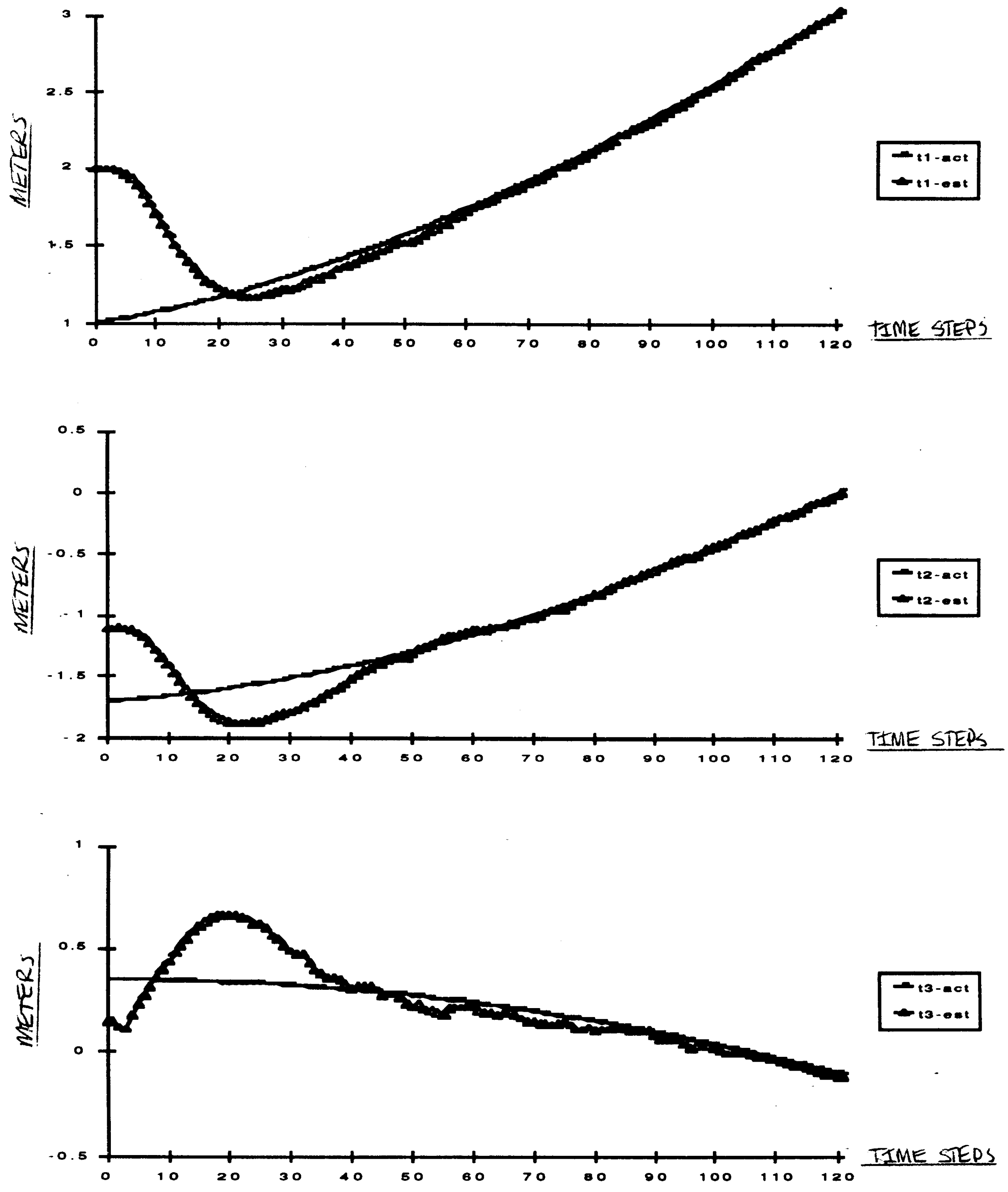
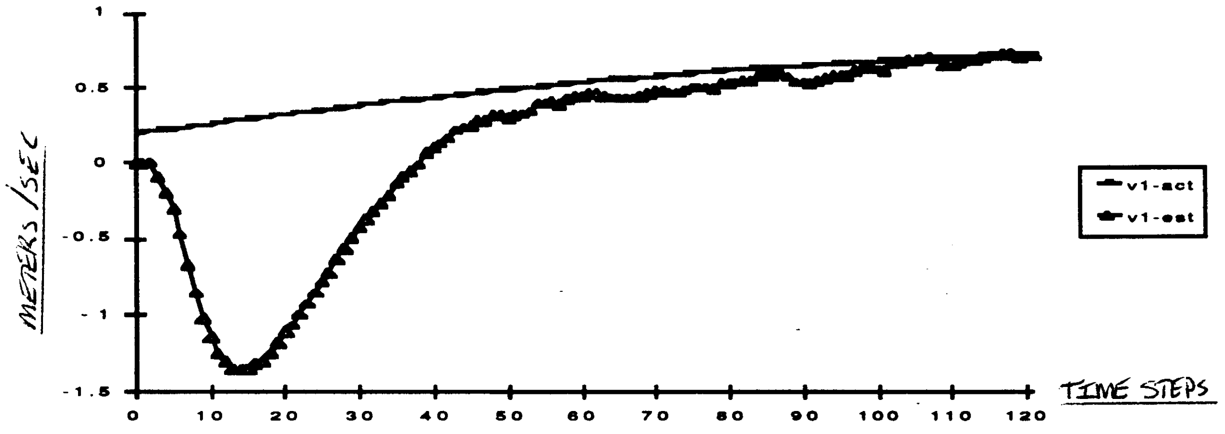
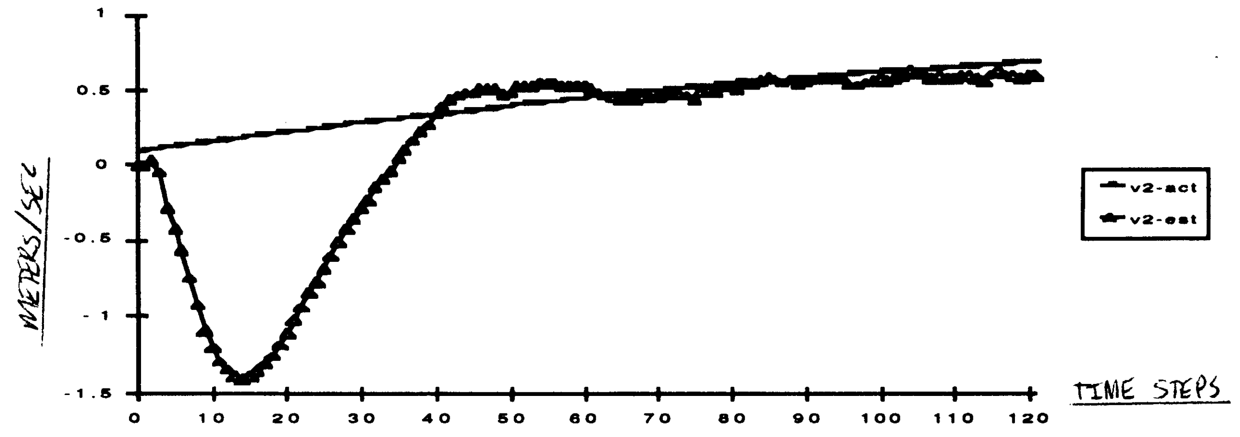


Figure 4.2.2-1(a,b,c): Simulated actual and estimated trajectories for state variables $t_{fb:f1}$, $t_{fb:f2}$ and $t_{fb:f3}$ (meters).

...Convergence Simulation (M2,D5), continued - VELOCITY.





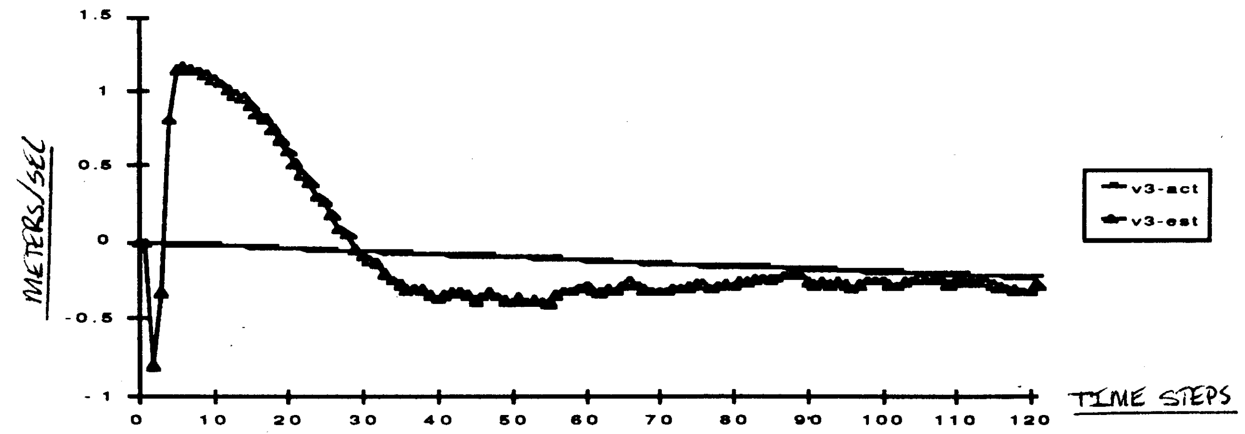
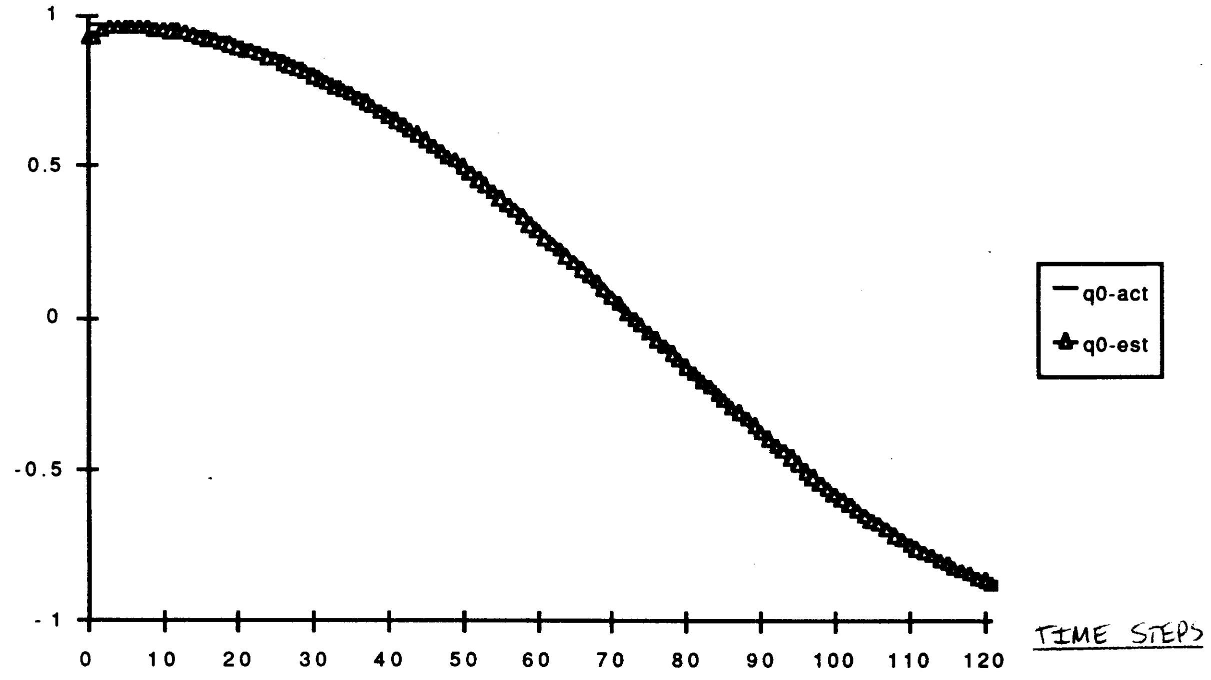


Figure 4.2.2-1(d,e,f): Simulated actual and estimated trajectories for state variables $v_{fb:f1}$, $v_{fb:f2}$ and $v_{fb:f3}$ (meters/second).

...Convergence Simulation (M2,D5), continued - ATTITUDE.



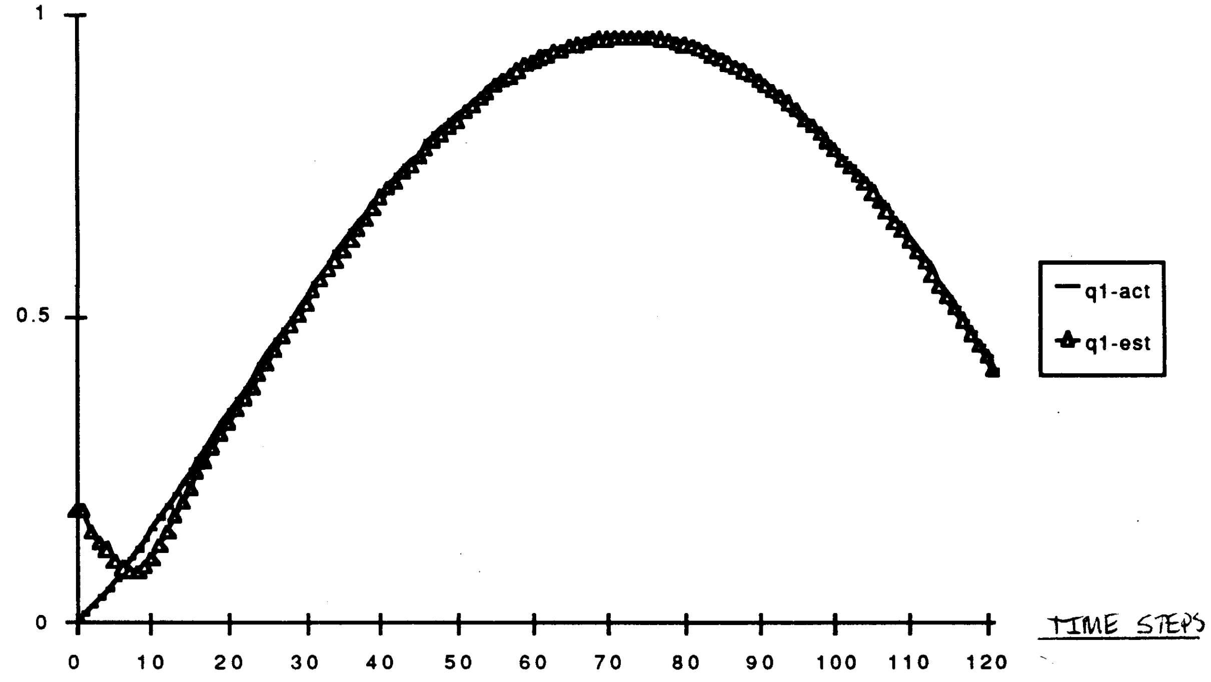
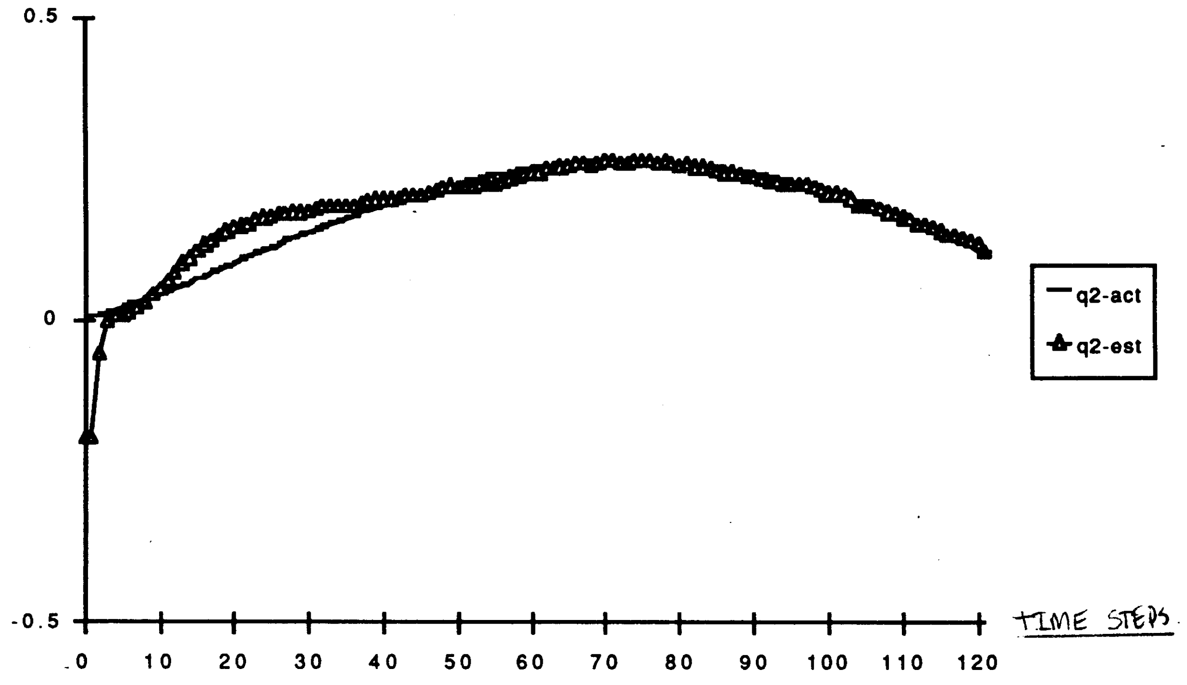


Figure 4.2.2-1(g,h): Simulated actual and estimated trajectories for state variables q_{fb0} and q_{fb1}.

...Convergence Simulation (M2,D5), continued - ATTITUDE...



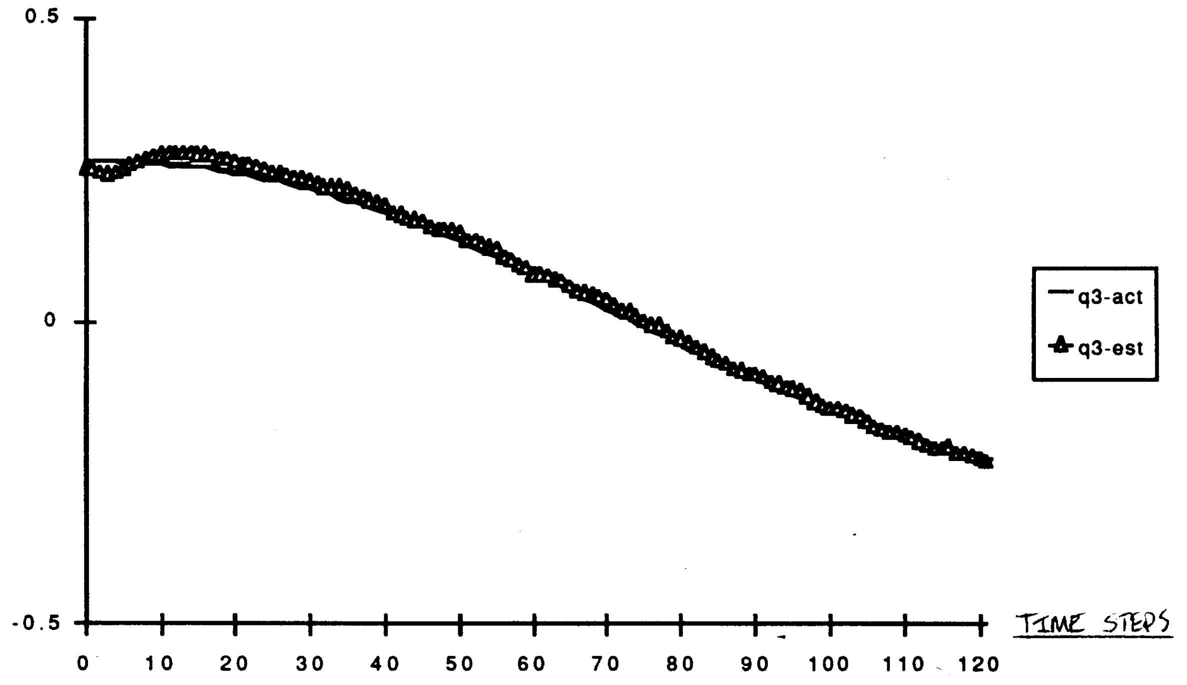
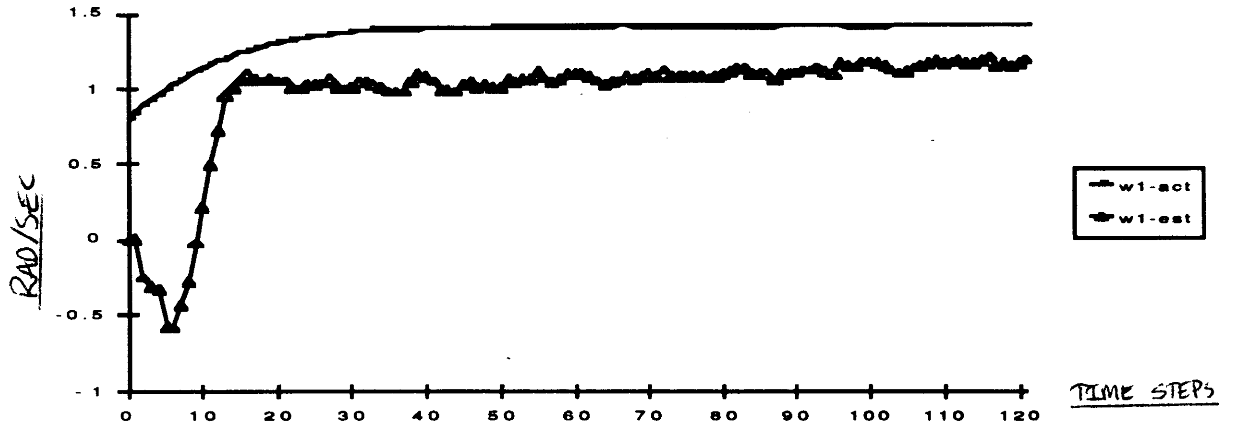
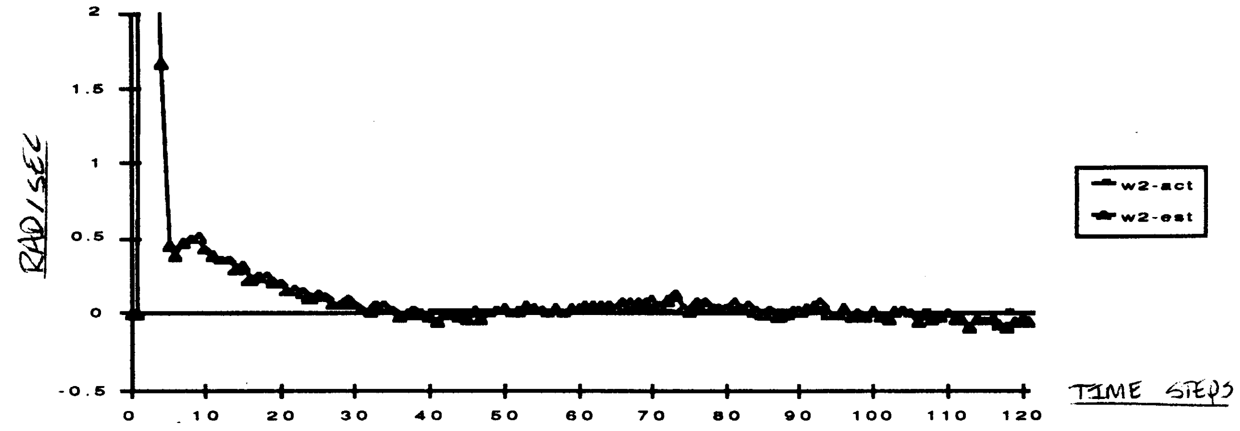


Figure 4.2.2-1(i,j): Simulated actual and estimated trajectories for state variables q_{fb2} and q_{fb3}.

...Convergence Simulation (M2,D5), continued - ANGULAR VELOCITY.





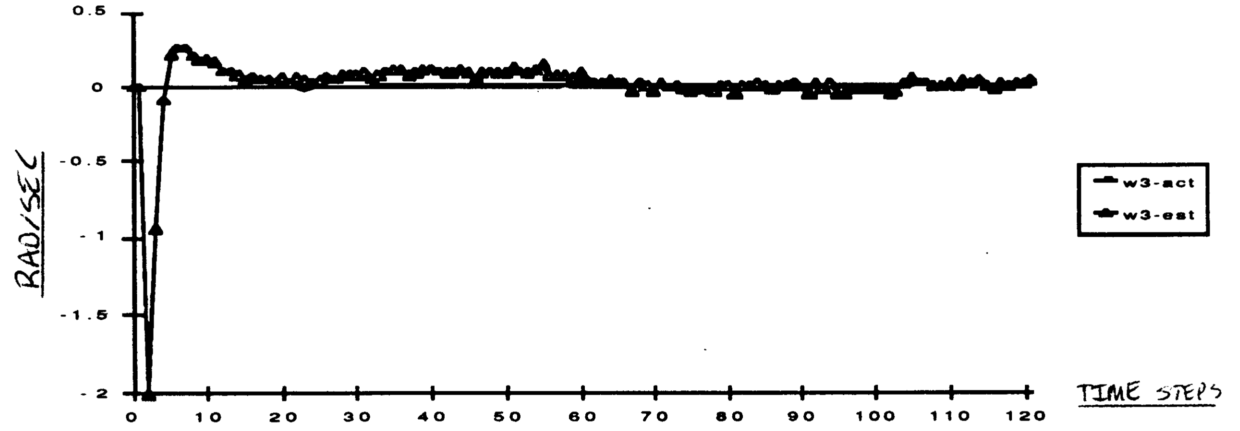


Figure 4.2.2-1(k,l,m): Simulated actual and estimated trajectories for state variables $w_{fb:b1}$, $w_{fb:b2}$ and $w_{fb:b3}$ (rad/sec).

5. Image Sequence Processing.

Obtaining image measurements for the EKF from a video sequence requires an *image sequence processor*. Model-based image processing was chosen as the paradigm for vision navigation because it is computationally efficient and because most projected applications take place in known environments, allowing the required assumptions for model-based processing to be made.

Models of navigation targets allow image processing procedures to be built specifically for efficiently parsing images of certain targets. This is because the image locations of important features are predictable for known targets so the procedures can search for only the distinctive features known to contain crucial information. The disadvantage is that certain procedures are applicable only to specific targets and can not be used for parsing general images.

Thus, measurements are obtained efficiently at the expense of requiring separate image processing functions for each navigation target. This is not an extremely high price to pay for efficiency, however, as the image processing routines required for obtaining the EKF measurements can often be very simple and compact, as shall be demonstrated in this chapter.

Note, however, that the given structure of the recursive estimator does not dictate any particular approach to image processing. The EKF will accept the appropriate measurements whether they are obtained from model-based target-specific image processing as suggested or if they are obtained by applying shape-from-shading algorithms or even manual extraction of features by a human operator, among many other possibilities. Only model-based target-specific image processing is discussed because it is one of few techniques potentially efficient enough to be implemented in real time with cheap, existing computational hardware.

The general approach of model-based processing is to use at each time step the 3D robot state prediction from the estimator along with the camera projection model and the target model to predict locations of relevant features in the current image. The portions of the image containing the features are analyzed to obtain information leading to the required measurements.

Such routines are understandably efficient because they waste no effort analyzing useless or overly redundant data. They are also relatively robust because selectimg portions of the image known to contain certain features reduces the possibility of false recognition and matching.

Since the routines used for one target are not generally applicable to other targets, this chapter can only demonstrate the model-based image sequence processing approach by example. To that end, the following sections describe an experimental target and the associated processes that are used to parse images of the target. This system is used in Chapter 6 to obtain measurements from a video sequence of the target.

5.1 Digital Image Processing.

Images are discussed in this chapter as if the image plane were a vertical wall and the image a framed picture upon the wall or as if the image were appearing on a video monitor. Each image is bounded by a rectangluar *window frame* with horizontal edges on the "top" and "bottom" and vertical edges on the "left" and "right". The orientation of the camera reference frame is such that the Y_c-axis points vertically "down" from the center of the image window frame and the X_c-axis points to the right. Thus the horizontal position of a location in the image is specified by the X_c coordinate, the vertical position by the Y_c coordinate. The relationship between the image window frame and the camera reference frame is illustrated in Figure 5.1-1.

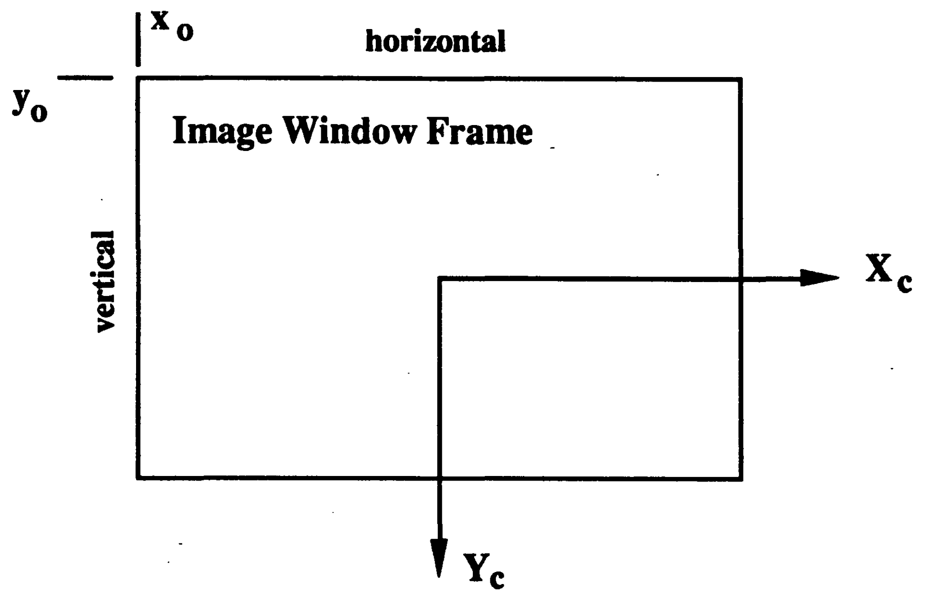


Figure 5.1-1: An image is described as a framed picture on a vertical wall.

The physical image has two spatial dimensions and a temporal one. At any time t and any location (x,y) in the image window frame, the brightness can be specified as I(x,y,t). But for processing images with a computer, the brightness signal is discretized spatially and temporally and is quantized. Temporal sampling produces a sequence of 2D images $I_k(x,y) = I(x,y,t_o+k\Delta t)$ where t_o is an initial time, Δt is the sampling interval, and k is an integer in $[0,1,\ldots,\infty)$. Spatial discretization results in images $I_k[m,n] = I_k(x_o+n\Delta x,\ y_o+m\Delta y)$ where x_o and y_o are respectively the left and top boundary of the image window frame, Δx and Δy are the horizontal and vertical separation between samples, and m and n are integers in $[0,\ldots,M]$ and $[0,\ldots,N]$ respectively. Finally, quantization results in digital images

$$b_k[m,n] = floor\left(\frac{I_k(m,n) - I_o}{\Delta I}\right)$$

where I_o is the "black level" of brightness, ΔI is a quantum of irradiance, and *floor*() returns the largest integer not greater than the argument. Additionally, $b_k[m,n]$ is usually represented by an 8-bit number, i.e. an integer in $[0,\ldots,255]$, so *floor*() must return 0 (zero) for negative results and 255 for results exceeding this maximum.

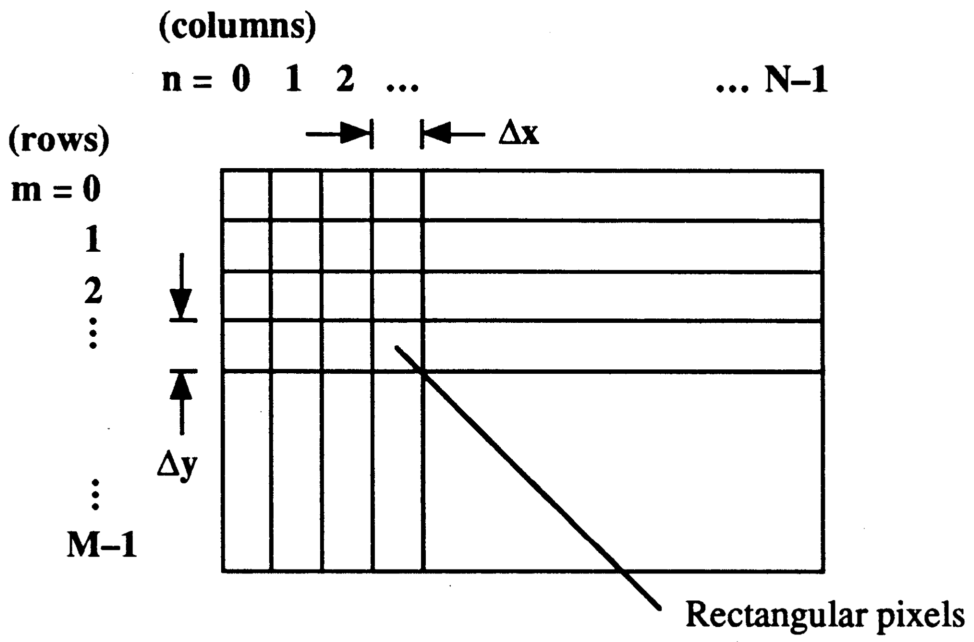


Figure 5.1-2: A digital image is a two-dimensional array of rectangular pixels arranged in rows and columns.

Even though the picture is actually captured over a small time interval rather than at a single instant of time and each pixel represents a small area rather a single point in the image, it is assumed that the digital images processed for vision navigation arise from such idealized sampling and quantization. The result is that each image in the sequence is treated as a two-dimensional array of picture elements, or *pixels*, which each belong to a *row* and *column* in the array and can be addressed this way. The arrangement of rectangular pixels in a digital image is illustrated in Figure 5.1-2.

5.2 The Experimental Navigation Target

Designed navigation targets are useful for many applications of vision navigation, including the intended application, submersible free-flying robots used in neutral-buoyancy tanks for simulating space robots. When it is possible to place a navigation target in the environment ahead of time, the target can be designed to make visibility and image parsing very simple, thus increase the speed and reliability of the navigation.

The experimental navigation target discussed here is designed for the navigation experiments described in the next chapter. Figure 5.2-1 shows a schematic view of the experimental navigation target. It consists of a (130cm, 120cm) black base, four large 40 cm white squares in the same plane as the base, and one central 20 cm white square which is raised from the base a distance of 25 cm. The target is mounted on a wall so that the large base is vertical and the central white square protrudes horizontally into the room. Diagrams in Chapter 6 illustrate the positioning of the target in the Simulation Room of the MIT LSTAR for actual imaging experiments.

The feature points on this target are the corners of the white squares. The number of feature points is N = 20, although not all of them need to be used every time step. In fact, the target was designed so that feature points are evenly distributed over the target. Since points at wide angles to each other allow the best triangulation, it is desirable to use points near the outside of pictures. The even distribution allows points to appear away from the center of the image for both close and distant viewing of the target.

Other desirable properties of this target include that it contains high contrast and it consists of basic geometric elements which allow high accuracy localization of the points. These properties enhance the efficiency and accuracy of the image measurement process.

A disadvantage of this target is that it contains a 180° symmetry, that is it looks the same upside down as it does right side up, for example. This is not a serious concern for vision navigation, however, because the estimator should never be anywhere close to 90° in error from the actual orientation, thus precluding any possible confusion.

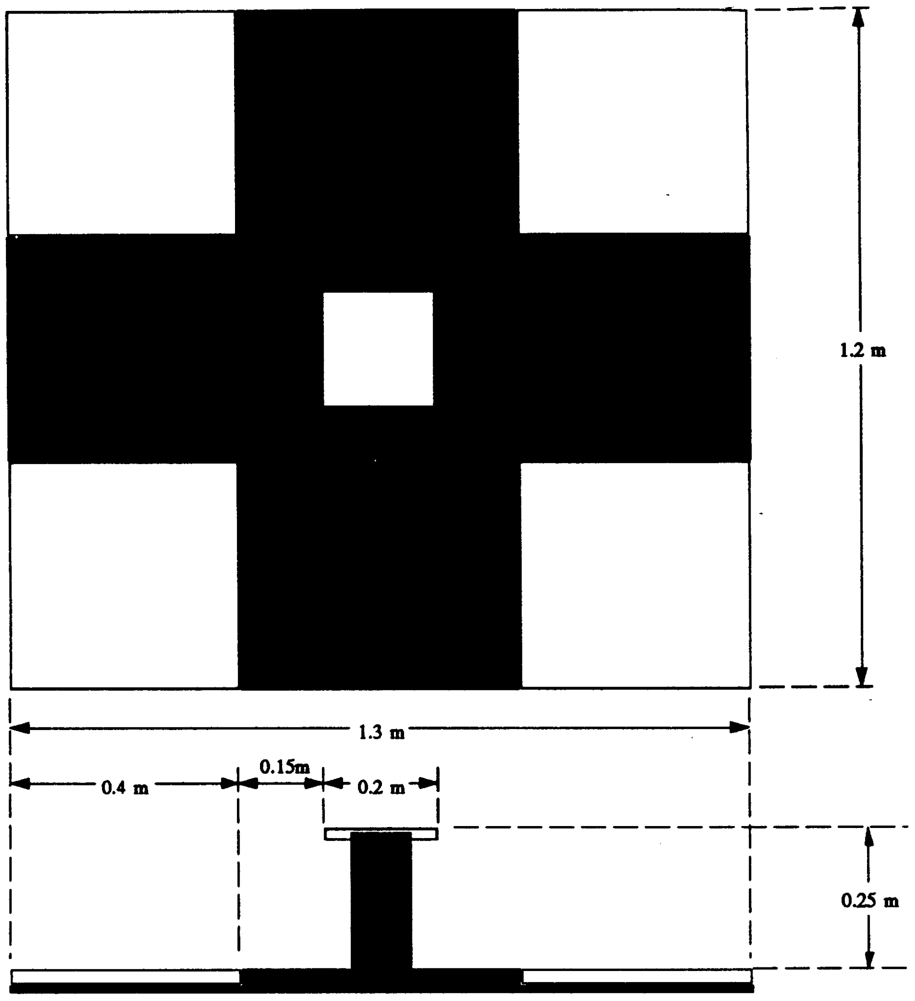


Figure 5.2-1: Experimental target configuration for testing of image processing routines. High contrast and well-defined feature points allow for simple and efficient image processing. An unfortunate oversight of this particular target design is that its symmetry does not allow unique initial alignment. For tracking, this is not an issue.

5.3 Image Processing Procedures.

The image processing task consists of finding the feature points in the image given a relatively accurate initial prediction of the robot and camera states. For this target, this can be accomplished by the following five steps:

(1) predict feature point locations in the image,

(2) use these predicted point locations to predict locations and orientations of the quadrilateral edges and to localize search windows for each quadrilateral edge,

(3) within each search window, find several points on the quadrilateral edge,

(4) compute equations for the lines containing the quadrilateral edges by using a least squares fit on the set of points found on that edge,

(5) intersect pairs of lines to locate the quadrilateral corners; these locations are the desired measurements.

These steps use detailed information about the target so that images of it can be efficiently parsed to obtain precise measurements of the image locations of the feature points. The following discussion outlines in more detail how the low-level image processing is performed.

Since straight lines on the target appear as straight lines in the image and angles depend upon perspective, the navigation target will appear in images as a set of white quadrilaterals on a black background. The locations of the corners, which are the desired measurements, are efficiently found because of the geometric simplicity of the target. Each corner is the intersection of two lines defined by adjacent sides of one of the quadrilaterals. Equations for the lines can be found by regression if the locations of several points on the sides of the quadrilaterals can be found. Points on the sides of the quadrilaterals are easily detected in the raw image data as sudden changes in image brightness because the quadrilaterals are bright and the background dark. All that is needed is a procedure for locating several points along each quadrilateral side.

A good state prediction for the robot is available, allowing the 3D feature points on the target to be projected onto the image plane using the measurement equation:

$$\tilde{\mathbf{y}} = \mathbf{h}(\tilde{\mathbf{x}}_{nav}, t).$$

The vector $\tilde{\mathbf{y}}$ contains the predicted image locations for the twenty feature points. These predicted corners can be used to localize the searches for the quadrilateral sides. Figure 5.3-1 illustrates how the location of the quadrilateral sides are predicted from predicted point measurements.

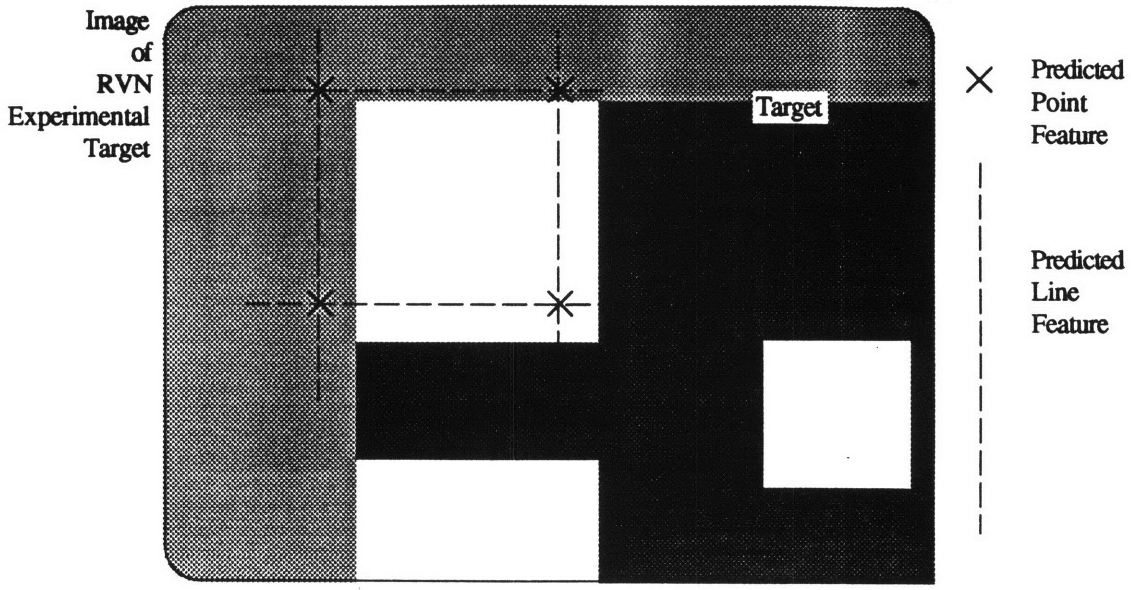


Figure 5.3-1: Using predicted feature points to localize line searches.

Figure 5.3-2 illustrates a localized search window used for seeking brightness discontinuities. The method utilized in the experimental system is to search along paths in the image roughly orthogonal to the predicted quadrilateral boundary. If the edge is roughly vertical, searches are carried out horizontally and gradients are computed along the horizontal direction. To accommodate random orientation of the quadrilateral edges, four possible search directions are available: north (vertical), east (horizontal), north-east (diagonal up and right), and south-east (diagonal down and right).

At each point in the search path the 1D brightness gradient is computed along the search direction, i.e. roughly orthogonal to the quadrilateral edge. A discrete approximation to the gradient is used. The point along each search path which contains the gradient of maximum magnitude is postulated to be on the quadrilateral boundary, as shown in Figure 5.3-2(b). The search window must be large enough to allow search paths that cross the quadrilateral boundary in several locations, spaced as far apart as possible. The window must also be small enough to guarantee that the gradients occurring at quadrilatral boundaries have much higher magnitude than other local features or noise.

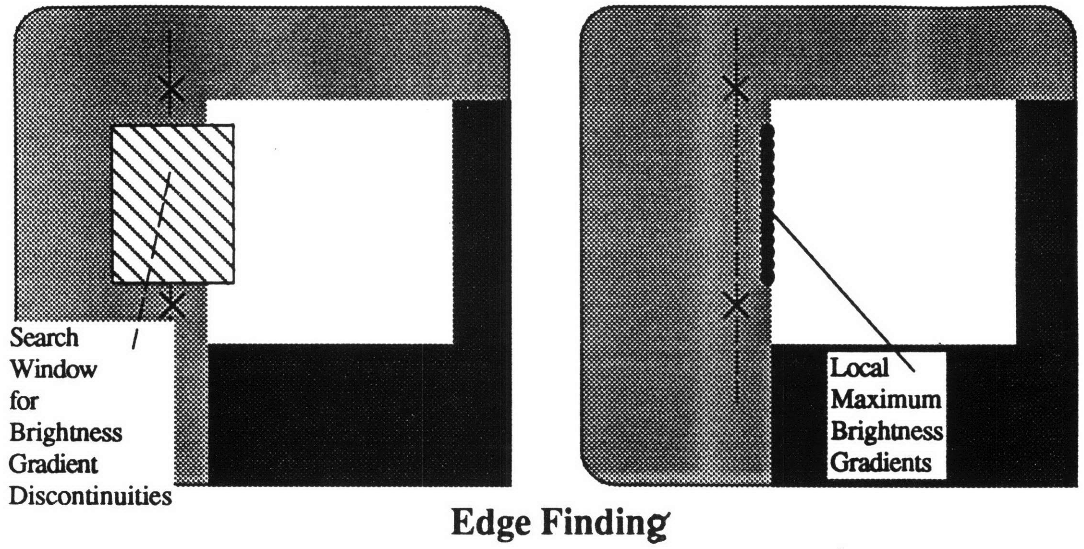

Search Window for Brightness Gradient Discontinuities
Local Maximum Brightness Gradients
Edge Finding

Figure 5.3-2: Using a localized search from a predicted line to find the sides of the squares. Searches are conducted perpendicular to the predicted line.

There are several tradeoffs associated with this procedure. There is a tradeoff of accuracy versus speed in the number of points found on each quadrilateral side. More points generally result in a more accurate equation for the line, but each additional search for a point requires additional computation to find gradients at points along the search path.

There is also a speed versus accuracy tradeoff in the choice of gradient operator. A large support operator gives a good estimate of gradient but requires much computation. Since only the location of the *maximum* gradient is sought, high accuracy is not very important. It is only important to be accurate enough to ensure that magnitudes of gradients computed at image locations maintain their relative order.

The five-cell support operator for gradient (scaled) shown in Figure 5.3-3 below accurately computes the derivative of up to a fourth-order polynomial curve. Conveniently, this five-element linear operator also requires zero real multiplications on a digital computer. One element is zero, two more elements have magnitude one, and the remaining elements are powers of 2. On a digital computer, the multiplication by ± 8 can be done with shifts and the others are trivial, resulting in efficient computation.

Forward Euler			-1	1	
Backward Euler		-1	1		
Centered Difference		-1	0	1	
Five cell support	1	-8	0	8	-1

Figure 5.3-3: Schematic representation of discrete gradient operators. For example, Forward Euler approximates $\nabla_t x = \partial x(t)/\partial t \approx [(-1)\, x(t) + (1)\, x(t+\Delta t)]$ (const.).

The speed versus accuracy tradeoffs were not investigated in this study because the total processing time was dominated by transporting image data rather than actually processing it.

5.4 Computational Efficiency Analysis.

The set of processes outlined in the previous section does not represent a great deal of computation. If k_1 points per line are used in the linear regression and search paths are k_2 pixels long, then the image processor must compute k_1k_2 gradients, each of which require 4 adds and 2 shifts. Each line fit requires approximately $2k_1+10$ multiplications and each point intersection takes about 5 multiplications. If there are N lines and N points to compute, the total computation cost is $(2k_1 + 15)N$ multiply operations, and $6k_1k_2N$ adds and shifts.

Typical parameters are $k_1 = 5$, $k_2 = 15$, and $N = 20$. This results in 500 multiply operations per frame and less than 10,000 adds and shifts. Counting one FLOPS (FLoating point Operations Per Second) as a multiply and add, or as five adds[1], a computational rate of 72,000 FLOPS is required to fully analyze 30 frames per second.

As a comparison, a typical way of finding the point features for a static image with no predicted information might begin with computing the 2D gradient at each point. Using a comparable (5,5) gradient operator on a (640,480) image, this requires over 230 million FLOPS of computation just to find gradients. Further analysis of this information is required to find the features.

Another more efficient approach for this high contrast target might be to first threshold the image. Just accessing each pixel in the image, however, requires on the order of one quarter million memory reads at each frame.

[1]There are many way of defining FLOPS. On fixed point processors, floating point multiplication requires five to fifty times the computation as one add or shift. On special purpose floating point processors multiplys can be done in one machine cycle. This measure is typical of the way multiply and add operations are converted to FLOPS.

These comparisons highlight the potential efficiency resulting from the recursive nature of the processing and the existence of the target model. Recursive processing of images in a sequence results in state predictions for each image which, coupled with the target model, allow predictions of image content, particularly the location of important features. In this example, the only part of the image that needs to be accessed are the small search windows localized about quadrilateral edges. The speed of the technique does not depend upon how large the image is, only upon how many points per quadrilateral edge are desired, how large the search windows are, and how accurately the gradients are computed.

Appendix B contains the source code required for parsing images of the experimental target using this technique.

6. Experiments on Real Imagery.

Experiments on real imagery combine the measurement procedure developed in Chapter 5 with the estimator developed and tested in Chapters 3 and 4. An image sequence is parsed by an image sequence processor, producing measurements of the locations of twenty points of a navigation target. The target is the one described in Section 5.1 and the image sequence processor is based on the procedures outlined in Section 5.2.

This chapter describes the experimental apparatus and procedure for obtaining measurements and presents results of the trajectory estimation. The results are similar to those presented in Chapter 4 except that the measurements are obtained from real imagery rather than simulated.

6.1 Experimental Procedure

The experiments took place in the "Simulation Room", housed in the Laboratory for Space Teleoperation and Robotics (LSTAR) at MIT. Figure 6.1-1 illustrates the layout of the Simulation Room. A video camera mounts into a set of three gimbals which allow it to point in any direction (although the range of motion is, as a practical matter, limited because of cable routing). The gimbal set rides atop a pedastal which stands on the mobile platform. The platform translates in two dimensions. The rail system can position the camera from the back wall (X-direction) in a range of 70cm to nearly 5m and from each of the side walls (Y-direction) up to about 50cm from each side of the room.

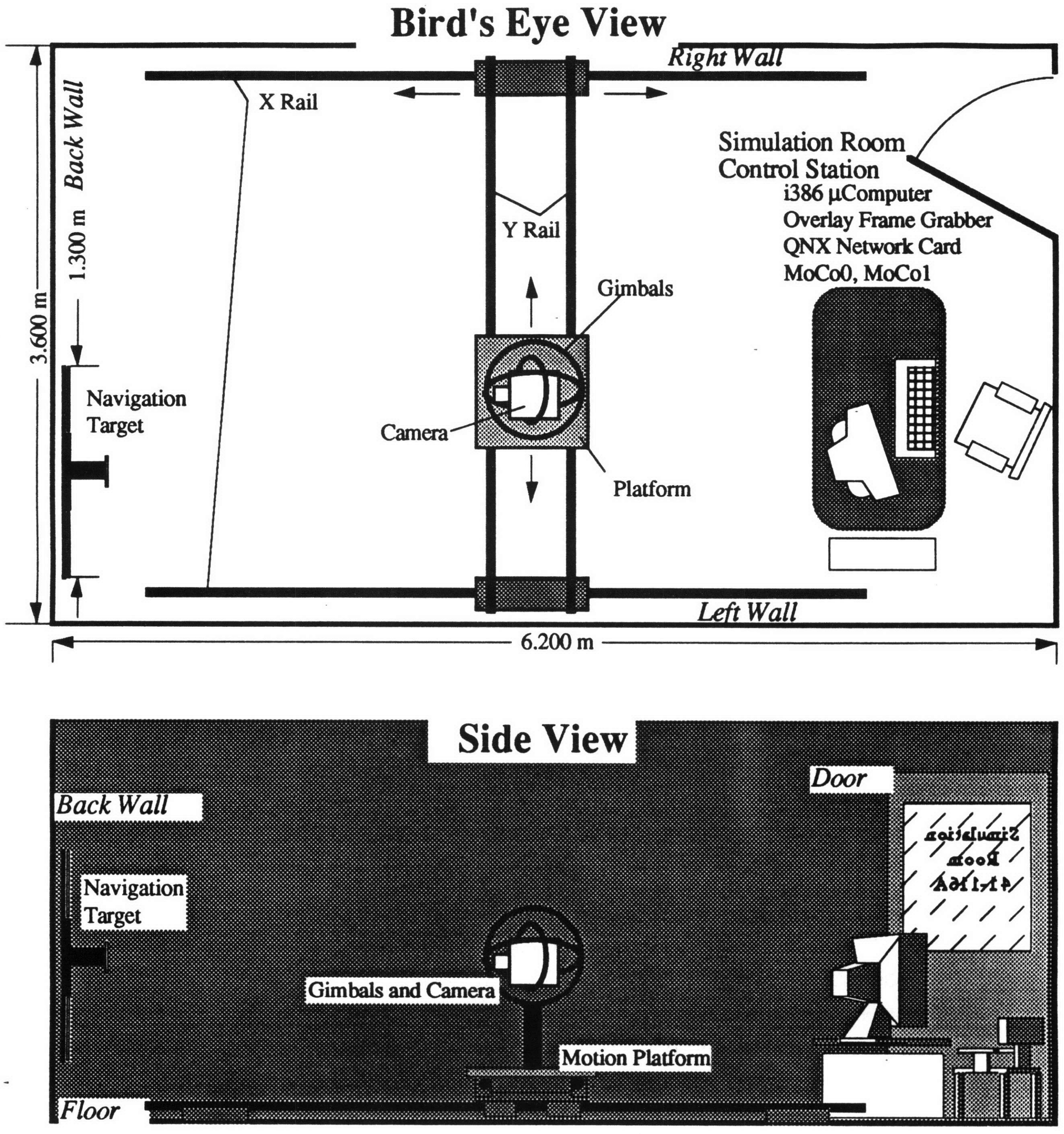


Figure 6.1-1: Simulation Room schematic. The 1.3m by 1.2m target hangs on the back wall. The X range of the platform is about 4m and the Y range is 2.6m. The gimbals allow full 3DOF rotation of a camera. The control station at the front of the roomcontains hardware for driving electric motor actuators for the platform closed-loop, hardware for collecting video imagery, and hardware for communicating with an external Control Station.

In this experiment, the platform alone is the "vehicle". The gimbals allows panning of the camera. Figure 6.1-2 illustrates the three coordinate frames as they pertain to the experiment. The fixed frame originates at the back corner of the room where the "left" and "back" walls intersect. The height of the fixed origin from the floor coincides with the height of the center of the navigation target. The positive X direction is toward the back wall, the Y-axis points to the "right" side of the room, and the Z-axis points down.

The body frame has its origin at the front-left corner of the motion platform (Figure 6.1-2), the X_b-axis points "forward", i.e. toward the back wall, the Y_b-axis points to the right, and the Z_b-axis points down. The camera frame has its origin in the center of the lens of the camera, the Z_c-axis is the optical axis, the X_c-axis points right, and the Y_c-axis points down as usual.

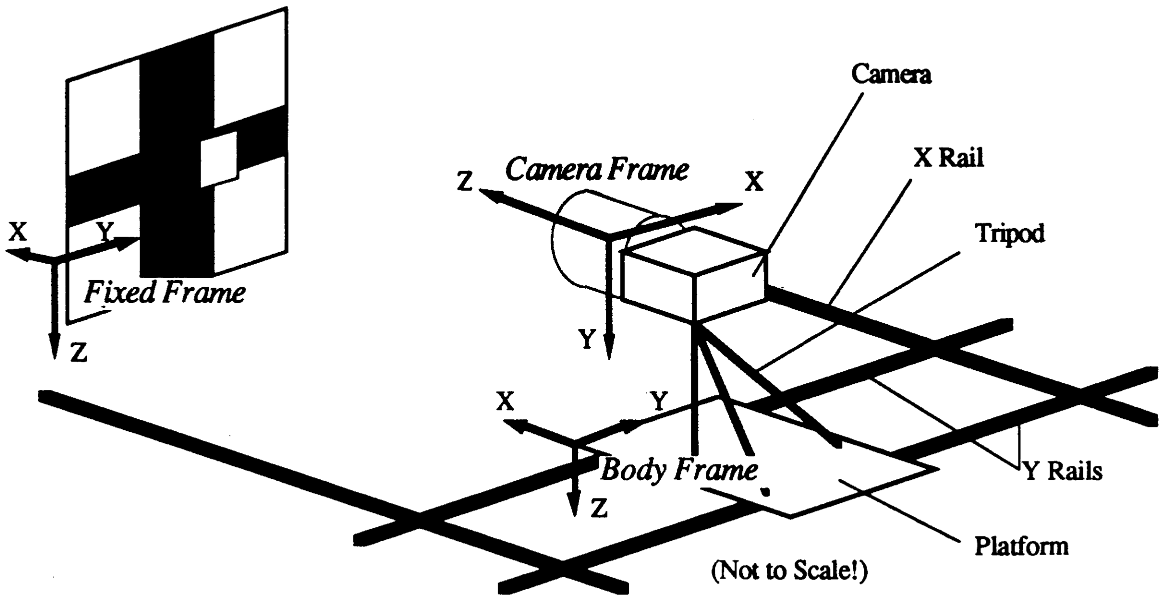


Figure 6.1-2: The experimental setup consists of a camera on a tripod on a translating platform. The platform moves in the Y (Fixed Frame) direction at constant velocity. The camera yaws as it translates (about the Camera Frame Y-axis) to keep the target in view.

The experimental vehicle has only two possible degrees of freedom (X- and Y- translation). As configured, the attitude of the robot has the body frame in perfect alignment with the fixed frame at all times and at a constant height (Z). For the experiment, the trajectory is a simple constant Y-velocity that takes the vehicle from the left side to the right side of the room. Thus all actual state variables are constant except for the Y position, $t_{fb:f2}$, which increases at a constant rate.

To ensure accurate trajectory data, the camera was moved one inch at a time across the floor with one frame being captured and processed at each step. This simulates a 30 inch per second velocity across the room. The camera was manually repointed at several locations along the trajectory. The position and orientation of the camera with respect to the platform was measured at each repointing for the purpose of updating $\mathbf{x}_{cam}$.

For the image sequence, the Y-rail assembly of the Simulation Room was positioned about 2.8m from the back wall. Over the course of the 100-inch-long trajectory, which consists of 100 equally spaced displacements, the camera has to be yawed left four times to keep the target completely in view. The simulated velocity $v_{fb:f2}$ is +30.0in/s (+0.762m/s), assuming that the time interval between frames is the video frame interval, 1/30th sec.

The largest source of error expected in measuring the actual trajectory of the platform is measurement of the camera state $\mathbf{x}_{cam}$ since the angles were measured by hand. Another potential source of error is the camera model since only a rough model exists. These errors were communicated to the estimator via the **R** matrix. Estimation trajectories will be shown for various values of **R**.

The following four tables, Figures 6.1-3 through 6.1-6, document, in order, the 3D feature point locations in the Simulation Room fixed frame, the state variables of the actual trajectory as measured directly in the Simulation Room, the command variables *implied* by the trajectory and the model dynamics of a submersible robot, and the pointing characteristics of the camera along the trajectory. The index k designates the time step, from 0 to 100.

Feature Point	X_f	Y_f	Z_f
0 (Center Square) [ul cor]	-0.3125 m	0.871 m	-0.100m
1 [ur corner]	-0.3125	1.071	-0.100
2 [lr corner]	-0.3125	1.064	0.100
3 [ll corner]	-0.3125	0.864	0.100
4 (Upper Left Square)	-0.075	0.316	-0.627
5	-0.075	0.716	-0.624
6	-0.067	0.711	-0.224
7	-0.067	0.311	-0.228
8 (Upper Right Square)	-0.075	1.212	-0.629
9	-0.075	1.611	-0.628
10	-0.067	1.614	-0.228
11	-0.067	1.214	-0.229
12 (Lower Right Square)	-0.058	1.213	0.172
13	-0.058	1.612	0.173
14	-0.050	1.613	0.572
15	-0.050	1.214	0.571
16 (Lower Left Square)	-0.058	0.312	0.175
17	-0.058	0.710	0.175
18	-0.050	0.709	0.574
19	-0.050	0.311	0.575

Figure 6.1-3: Table of target feature point locations in the fixed frame.

State variables	Initial condition	Trajectory
$t_{fb:f1}$	-2.78 m	---
$t_{fb:f2}$	0.381 m	+ 0.762 k / 30
$t_{fb:f3}$	0.948 m	---
q_{fb0}	1	---
q_{fb1}	0	---
q_{fb2}	0	---
q_{fb3}	0	---
$v_{fb:f1}$	0	---
$v_{fb:f2}$	0.762 m/s	---
$v_{fb:f3}$	0	---
$w_{fb:b1}$	0	---
$w_{fb:b2}$	0	---
$w_{fb:b3}$	0	---

Figure 6.1-4: Table of state variables for actual trajectory. Index k designates the index of the time step (k=0...100).

Commands	Initial Condition	Trajectory
$\mathcal{F}$:f1	0	---
$\mathcal{F}$:f2	284.51556 N	---
$\mathcal{F}$:f3	0	---
$\mathcal{T}$:b1	0	---
$\mathcal{T}$:b2	0	---
$\mathcal{T}$:b3	0	---

Figure 6.1-5: Table of commands for actual trajectory. This force applied to the model of the submersible robot balances the drag at a velocity of 0.762 m/s, the actual velocity of the real camera relative to the target.

k	ψ_k	r_0	r_1	r_2	r_3	t_{bc1}	t_{bc2}	t_{bc3}
0-15	+6°	$\cos(\psi_k/2)$	0	0	$\sin(\psi_k/2)$	7.6 cm	13.75 cm	90.55 cm
16-33	-3°	"	"	"	"	7.9	12.9	90.55
34-54	-12°	"	"	"	"	7.6	12.2	90.55
55-84	-23°	"	"	"	"	7.8	11.1	90.55
85-100	-36°	"	"	"	"	7.8	10.6	90.45

Figure 6.1-6: Table of camera position and orientation with respect to the body frame for five segments of the trajectory. The camera state vector x_{cam} is updated four times throughout the trajectory using these values. Index k designates the index of the time step (k=0...100).

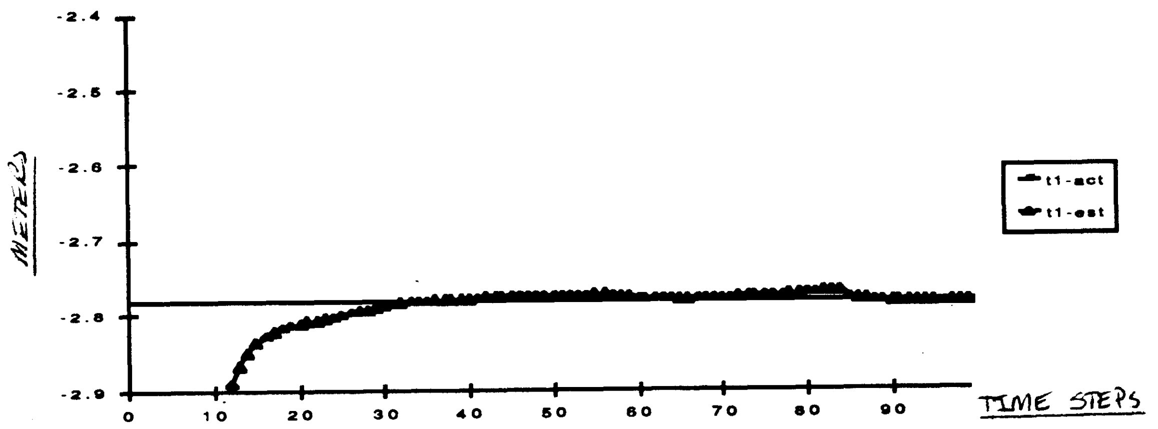
6.2 Results

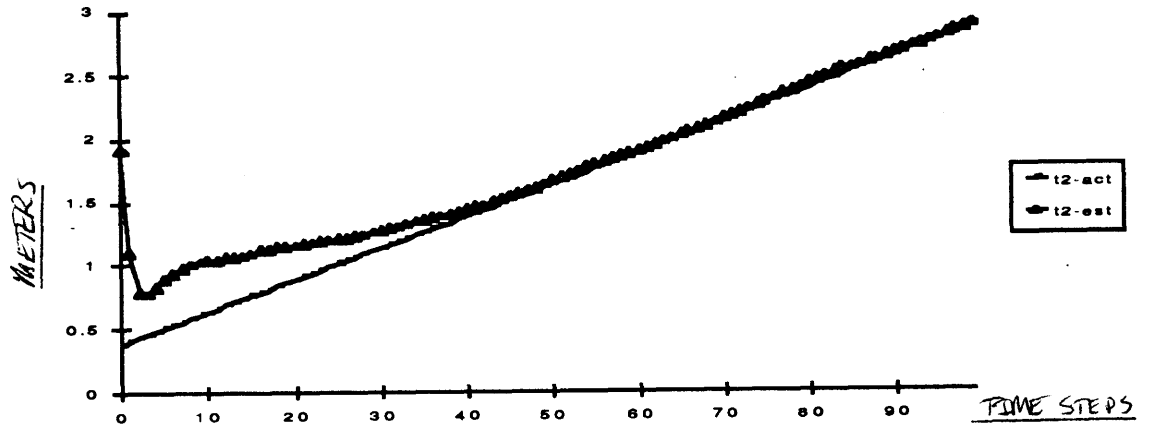
One full set of state variable trajectories is presented in Figure 6.2-1. Each state variable starts with a random initial error to demonstrate the convergence of the estimation. Notice that this error is quite significant in some state variables, far worse than should be reasonably expected from initial manual alignment of the robot. In particular, the initial translation errors in the components of $\mathbf{t}_{fb:f}$ are 70 cm, 1.5 m, and 1.4 m respectively (initial $\mathbf{t}_{fb:f}$ estimate: [-3.5m,1.9m,2.3m], initial $\mathbf{t}_{fb:f}$ actual: [-2.8m,0.4m,0.9m]), initial velocity errors reached 60 cm/s, and rotational velocity errors began at nearly 30 deg/sec.

Note also that the "actual" trajectory is itself based on manual measurements and therefore some of the "actual" state variables contain inaccuracies. The actual trajectories for $v_{fb:f1}$, $v_{fb:f3}$, $w_{fb:b1}$, $w_{fb:b2}$, $w_{fb:b3}$ are, in fact, exactly correct[1] because the platform neither rotates nor moves at all in the Z_f or X_f directions. The inaccuracies that do exist should be relatively insignificant.

Figure 6.2-2 illustrates the pure tracking performance of the estimator on the same sequence of measurements with no intial errors in the state predictions. That is, the state variables are initialized to the "actual" state. Again, the actual state is measured also and may not represent the true state of the robot any better than the estimated state. Only three state trajectories—$t_{fb:f1}$, $v_{vb:f2}$, $w_{fb:b3}$—are shown; these are typical of the other state trajectories.

[1]Except perhaps minute errors due to the Y and X tracks not being exactly orthogonal, the floor not being orthogonal to the walls, the rotation of the earth not being exactly one rotation per day,...





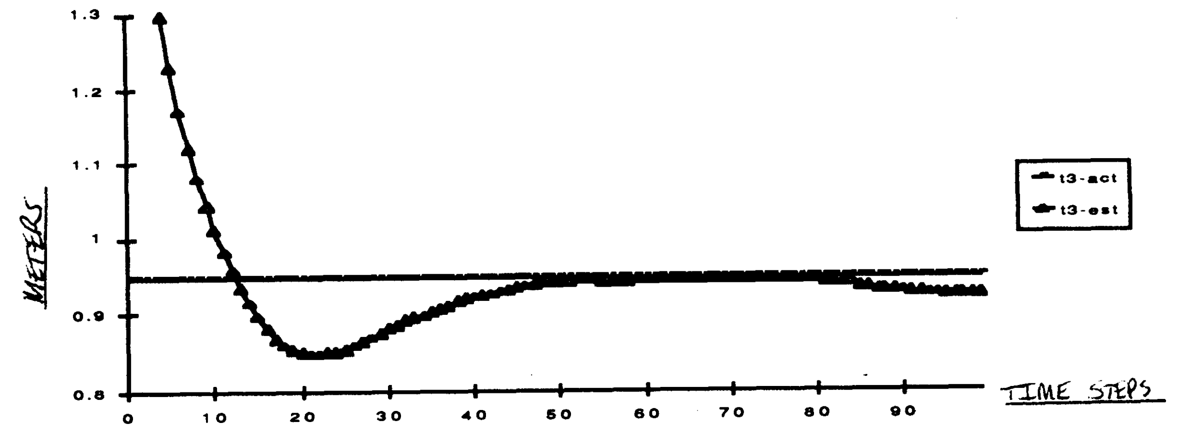
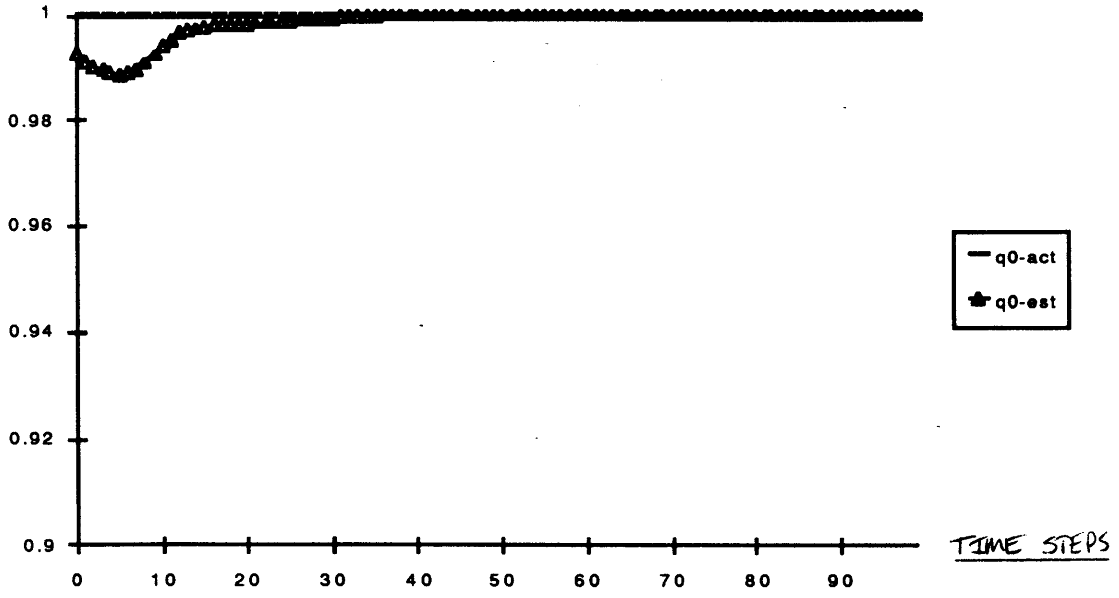


Figure 6.2-1(a-c): Evolution of estimates of state elements (a) $t_{fb:f1}$, (b) $t_{fb:f2}$, and (c) $t_{fb:f3}$ over the course of 100 time steps (3.3 seconds). The ordinate unit is meters.



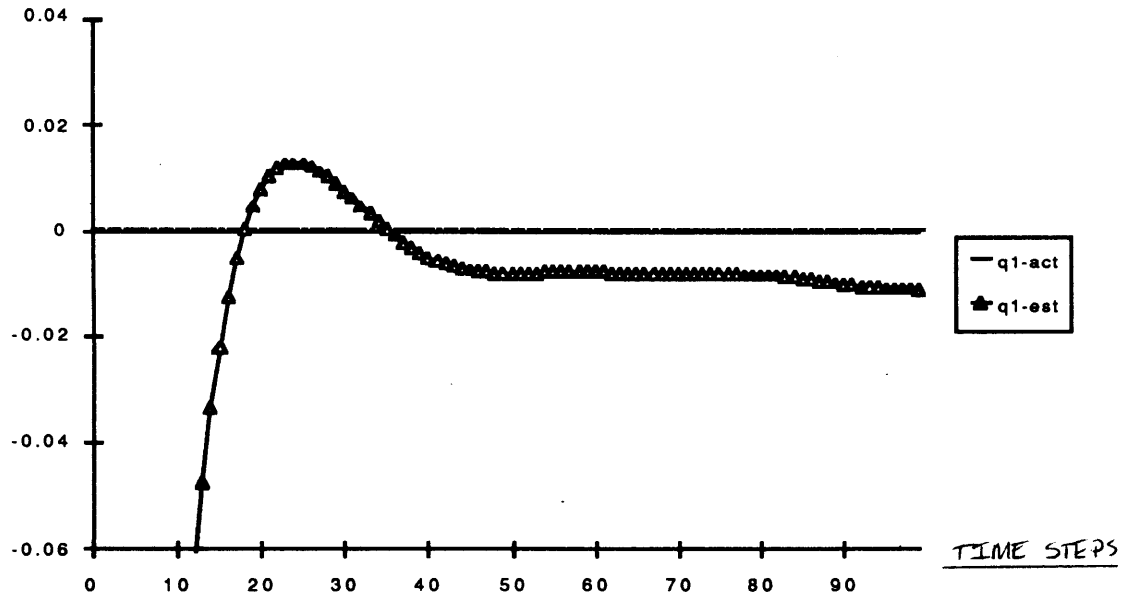
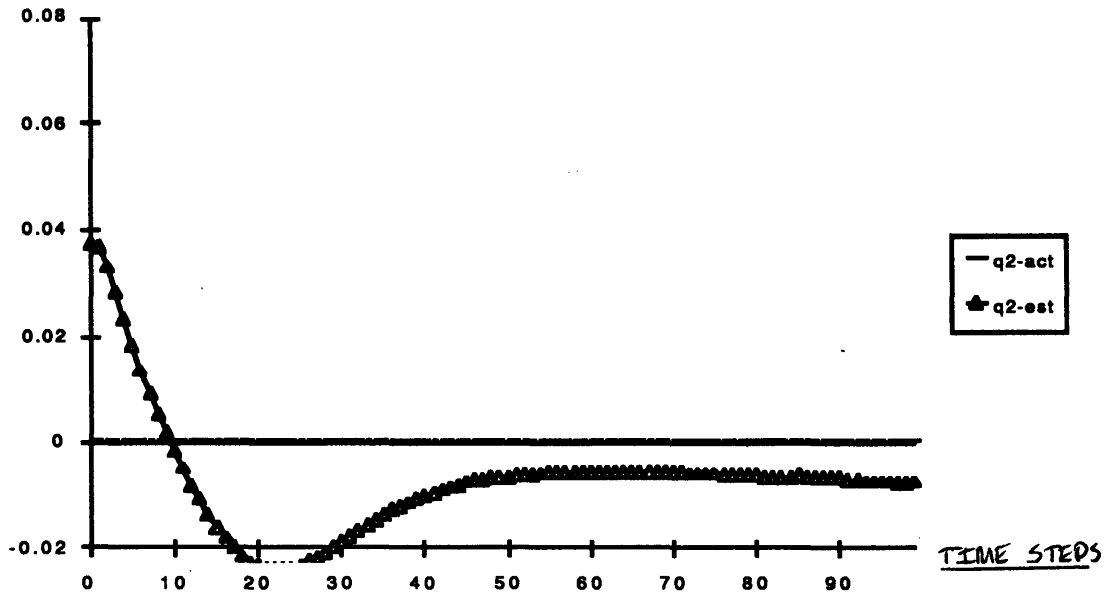


Figure 6.2-1 (d-e): Evolution of estimates of state elements (d) q_{fb0} and (e) q_{fb1} over the course of 100 time steps (3.3 seconds).



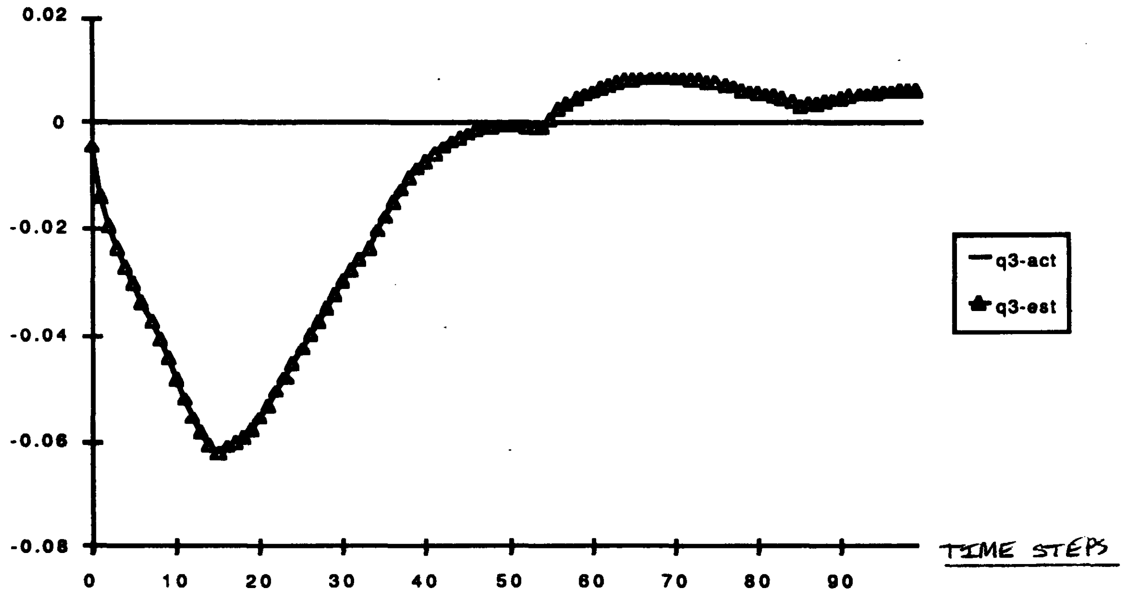
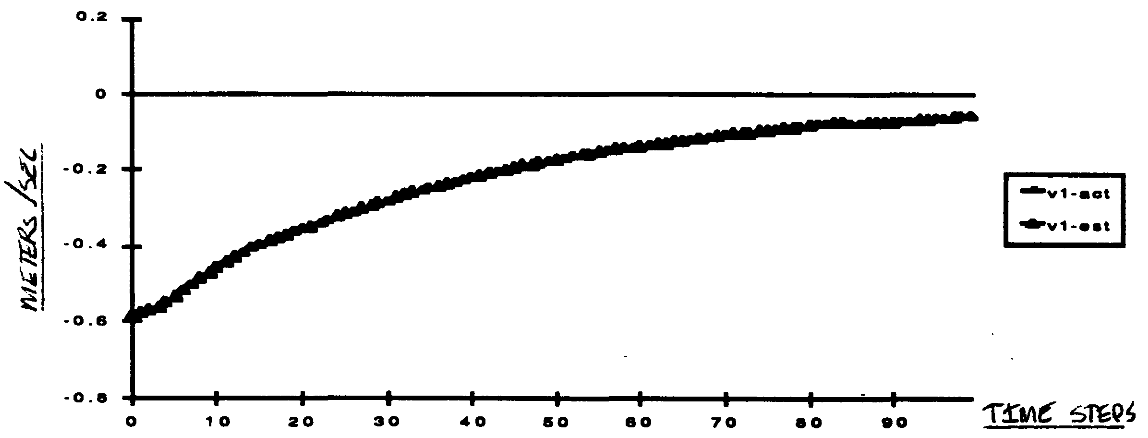
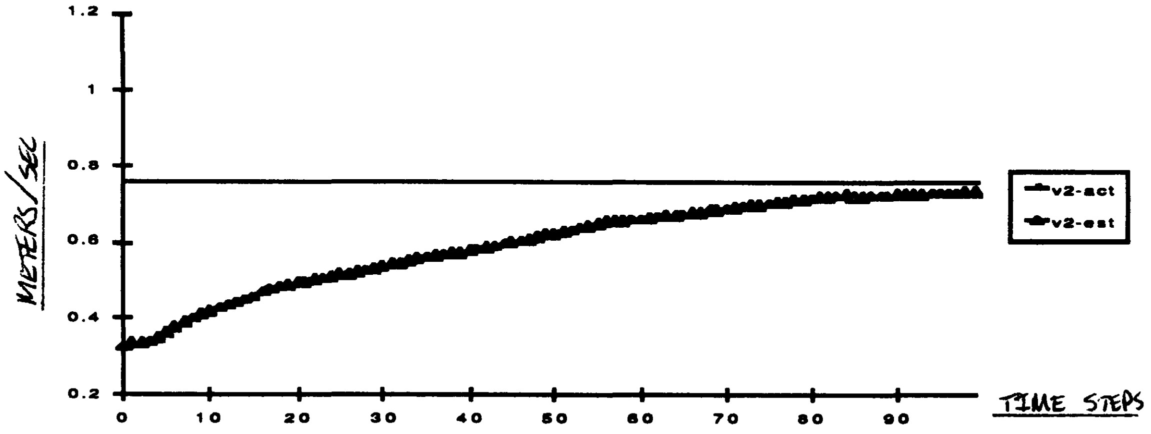


Figure 6.2-1 (f-g): Evolution of estimates of state elements (f) q_{fb2} and (g) q_{fb3} over the course of 100 time steps (3.3 seconds).





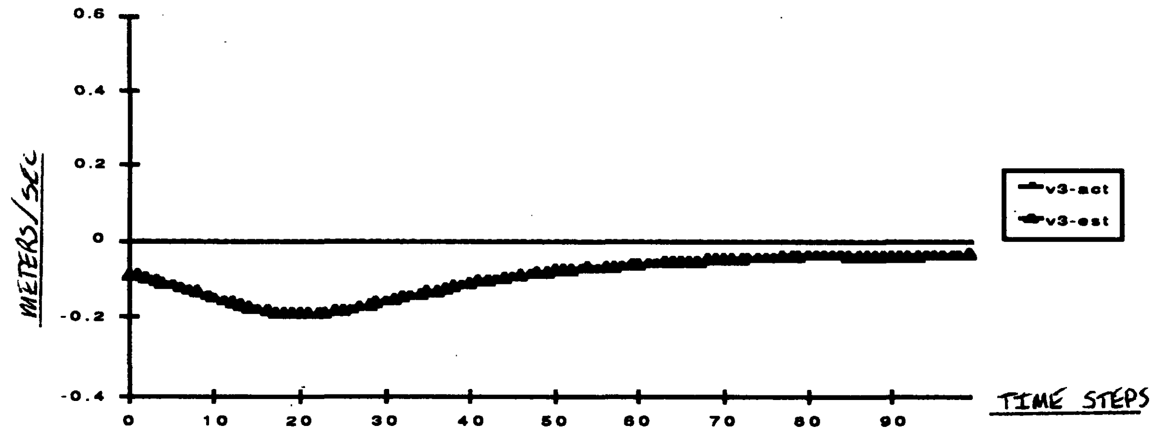
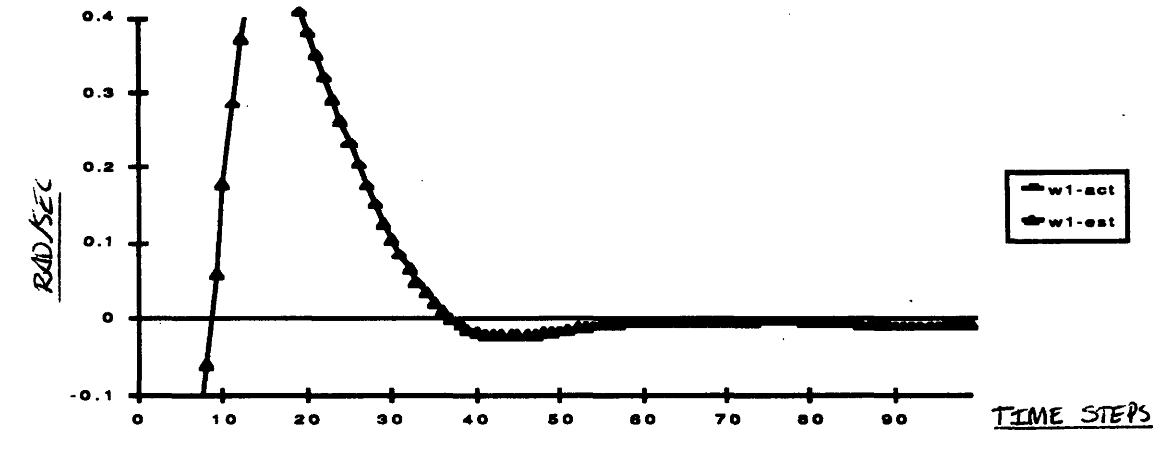
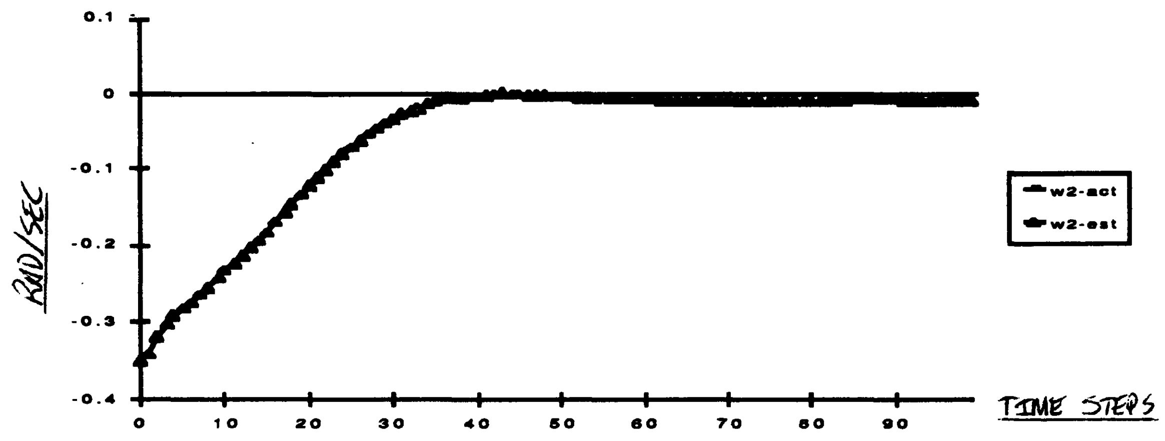


Figure 6.2-1 (h-j): Evolution of estimates of state elements (h) $v_{fb:f1}$, (i) $v_{fb:f2}$, and (j) $v_{fb:f3}$ over the course of 100 time steps (3.3 seconds). The ordinate unit is meters/second.





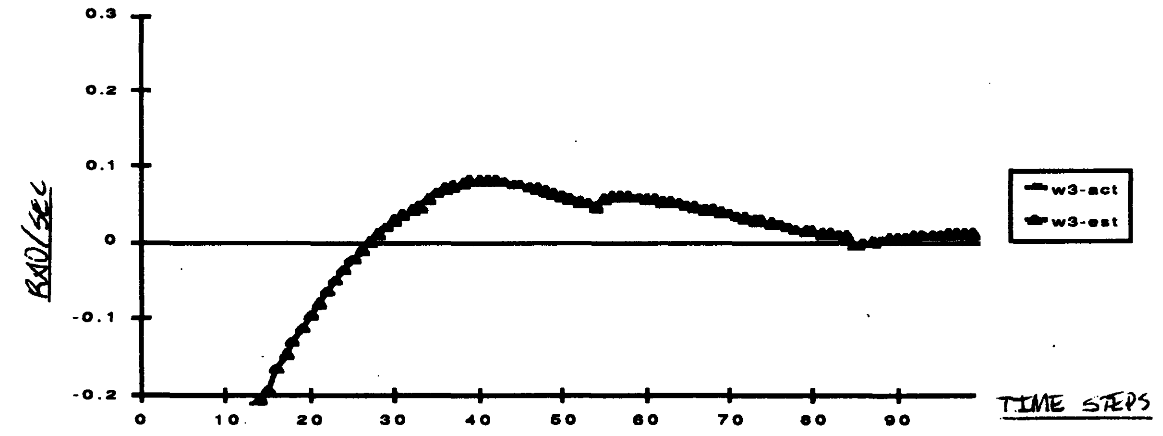


Figure 6.2-1 (k-m): Evolution of estimates of state elements (k) $w_{fb:b1}$, (l) $w_{fb:b2}$, and (m) $w_{fb:b3}$ over the course of 100 time steps (3.3 seconds). The ordinate unit is radians/second.

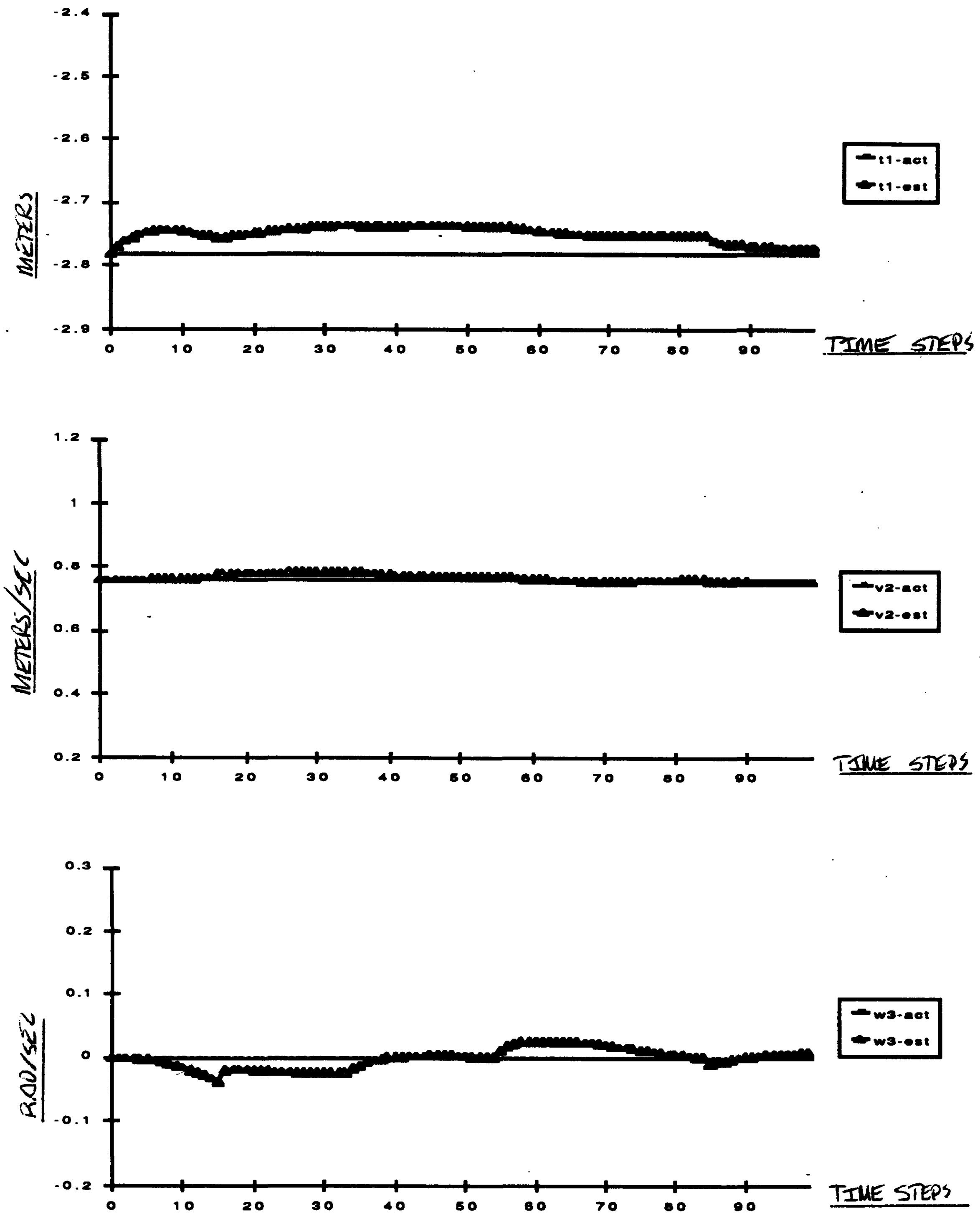


Figure 6.2-2(a-c): Estimates of (a) $t_1=t_{fb:f1}$, (b) $v_2=v_{fb:f2}$, and (c) $w_3=w_{fb:b3}$ with $\sigma^2=10^{-9}$ and no added initial error.

6.3 Tuning the Filter.

For the data shown in Figure 6.2-1, the noise covariance used in the EKF is

$$\mathbf{R} = E[\eta_k \eta_k^T] = (10^{-5})\,\mathbf{I}$$

where **I** is the (40,40) identity matrix. This represents an particular level of uncertainty in the measurement model.

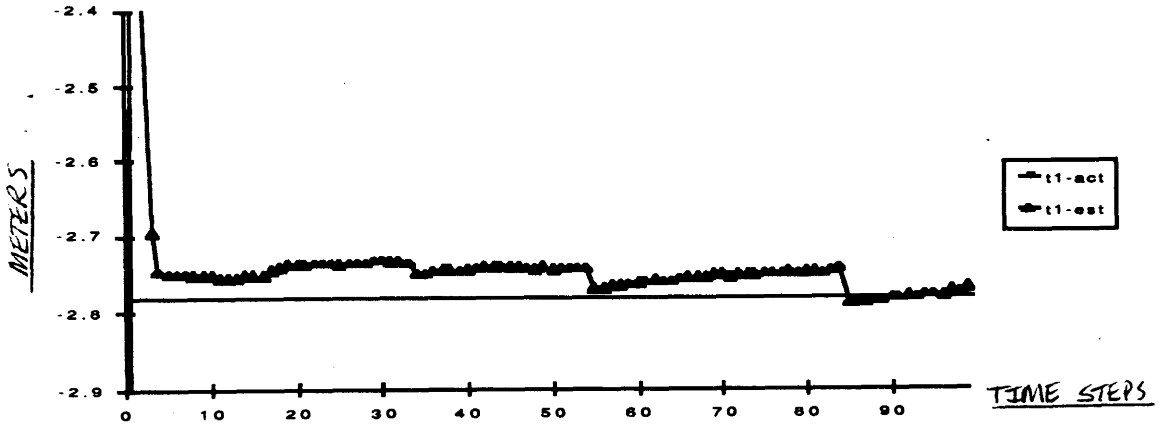
The intended procedure for incorporating such uncertainty in the EKF is to establish ahead of time what the expected uncertainty is. Uncertainties must be modelled as Gaussian distributed random noise added to each of the measurements.

However, in practice, the level of uncertainty may not be known exactly, as is the case in this experiment. The measurements themselves are taken to subpixel precision, but the camera model is not known to great accuracy and there is doubt as to how accurately the camera state vector $\mathbf{x}_{cam}$ was measured. Since the expected errors are not readily quantifiable, the value of **R** is chosen quite arbitrarily.

If the covariance is chosen too low, the estimator assumes it has a very good model and may overreact to noisy measurements, resulting in a jumpy trajectory estimate. If the covariances are chosen too high, the estimator will be assume the measurements are more noisy than they really are ard will be sluggish in following quick changes in the trajectory.

The first attempt at estimating the trajectory was performed with $\mathbf{R} = (10^{-9})\,\mathbf{I}$, which was the value used in the simulation experiments of Chapter 4. The data demonstrated a striking over-reaction whenever $\mathbf{x}_{cam}$ was changed, as shown in Figure 6.3-1. This indicates that the measurements of the camera state were indeed very poor. Hence, the confidence level in the measurement model needed to be altered to reflect the real levels of error present in the measurements.

Toward this end, the estimation was run with several levels of measurement noise variance. Defining $\mathbf{R} = \sigma^2\mathbf{I}$, the values $\sigma^2 = 10^{-n}$ were used for n = {1,3,5,7,9}. The variance $\sigma^2 = 10^{-5}$ was used to generate the trajectories in Figure 6.2-1. Figure 6.3-2 shows the estimated trajectory for state variable $w_{fb:b3}$ for the remaining four variances. Only this state variable is shown because it is the one most affected by the change.



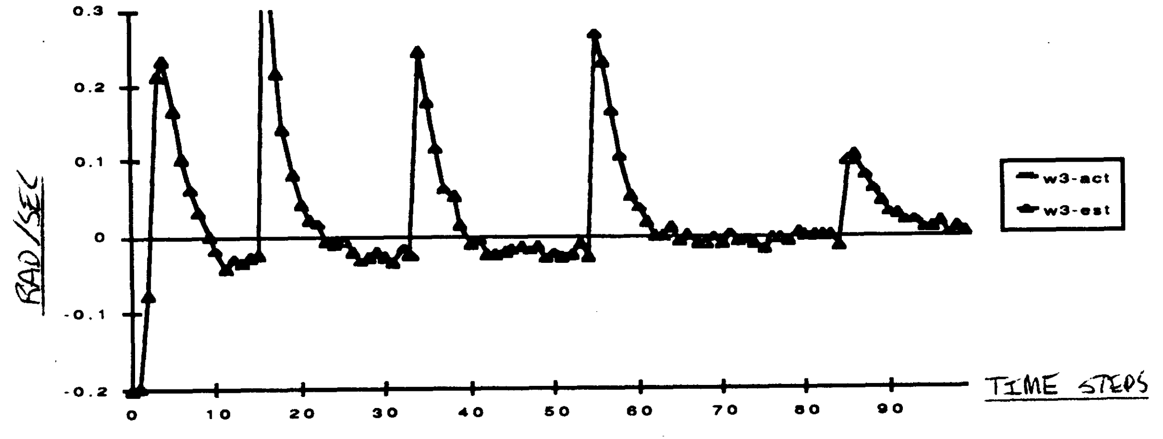


Figure 6.3-1(a-b): Estimates of state variables (a) $t_{fb:f1}$ and (b) $\omega_{fb:b3}$ with $\sigma^2 = 10^{-9}$.

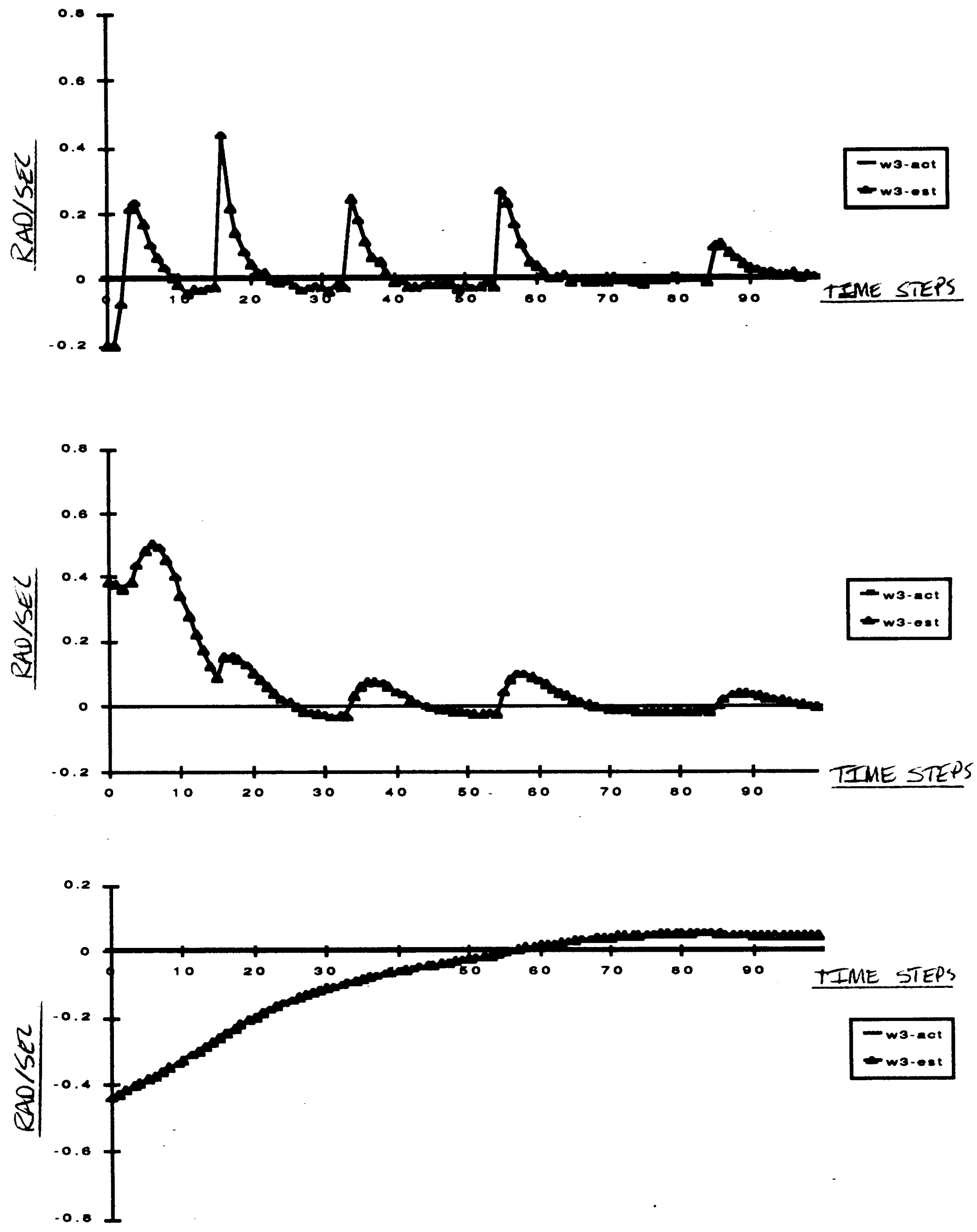


Figure 6.3-2 (a-c): State variable $w_{fb:b3}$ for σ^2= (a) 10^{-9},(b) 10^{-7}, and (c) 10^{-3}.

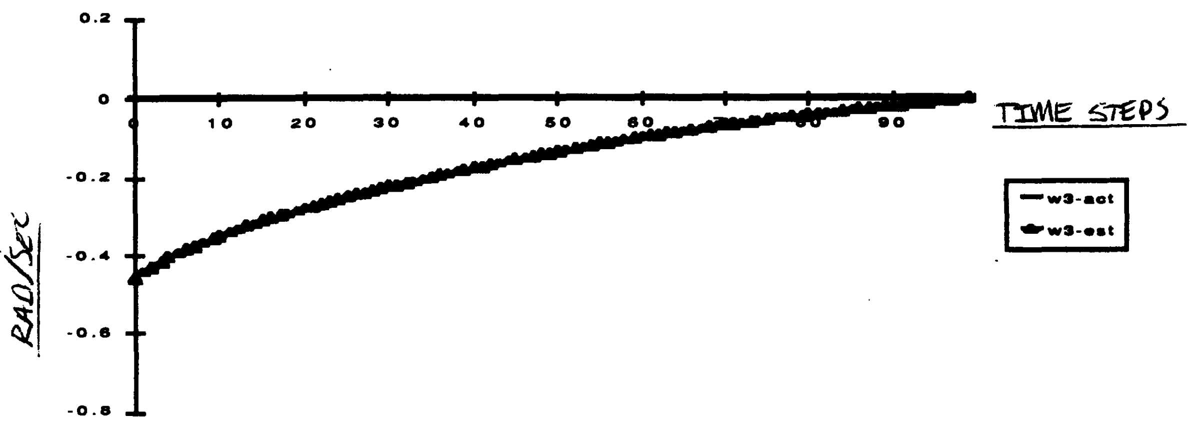


Figure 6.3-2(d): State variable $w_{fb:b3}$ for $\sigma^2 = 10^{-1}$.

7. Conclusions.

The experimental results of Chapters 4 and 6 have demonstrated the ability of the navigator to track robot trajectories based on simulated and real measurements. The experiments prove the functionality of the vision navigation technique. The remaining sections discuss implementation issues associated with application of the technique to real free-flying robots and recommended extensions of the navigator ideas to related problems.

7.1 Implementation Issues.

The vision navigator as developed and tested in the previous chapters can be implemented immediately in its present form as a functional navigation system for free-flying robots. There are several existing and potential limitations, however, which must be overcome for the system to perform to its fullest.

The first of these is a potentially poor dynamic model. Only a few of the many possible inaccuracies in the dynamic model were tested in the simulations. The reason for this is that it is simply impossible to predict and systematically simulate every possible parameter of the dynamic model which could be mis-modelled.

Dynamic modelling errors are particularly prone to occur for submersible free-flyers, which will be the first free-flying vehicles to use this vision navigation system. The dynamics of these complex robot vehicles in water are extremely intricate, involving nonlinearities and cross-coupling of dynamics between various translational and rotational degrees of freedom. These effects are not modelled in the simulations and probably cannot easily be modelled for real robots. If left unmodelled, these unknown dynamics may cause failure of the estimator when used on a real robot, particularly at high velocities. Hence, the navigator may be limited to low velocities until accurate models of robot dynamics are available. The extent of the limitation posed by this potential mis-modelling is unknown and can only be discovered through experimental trials with a vision system on an operational vehicle. The dynamic modelling issues are not a great concern for space robots because they do not share the complex drag environment of submersibles.

Another potential dynamics-related problem occurs when the frame rate of the navigator is low compared to the motion of the robot. In these cases, the simple propagation equation used in the simulations may be unstable and lead to catastrophic prediction errors.

The simulations utilize a simple one-step state propagation which computes the state change between the current time step and the next time step by assuming the current estimated velocity is constant over the time interval. That is,

$$\Delta \mathbf{x}_k = \dot{\mathbf{x}}_k \Delta t,$$

and

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k.$$

This works in the simulation because the time intervals are small compared to vehicle motion and also because the actual dynamics propagation is generated the same way.

In real situations where the sampling rate is low, as will probably be the case in the first implementations, a stable multistep numerical integration procedure, such as a high-order Runge-Kutta method, is suggested for state propagation. However, even a stable numerical integration technique cannot help if the dynamics model is sufficiently poor. The implied limitation, then, is that the motion of the robot, both speeds and accelerations, must remain low relative to the sampling rate of the navigator.

The second category of potential difficulties is associated with the mobile camera. Potential errors in the measurement of $\mathbf{x}_{cam}$ were not considered in the development of the technique because it was postulated that the camera state can be measured mechanically and thus should be quite accurate. The results of the experiment of Chapter 6 demonstrate the adverse effects of a poorly calibrated camera pointing mechnism and suggest that calibration is important for good results. The extent of accuracy required in the calibration is an unexplored topic which must be addressed experimentally for implementations utilizing mobile cameras.

A further concern, however, involves the control system and actuators which are responsible for keeping the camera trained on the navigation target. This issue was not a topic of this study. It is expected that the dynamics of the camera system in addition to the dynamics of the robot itself may make it difficult to maintain a view of the target in a way such that reliable image processing can take place. If there are significant errors in the camera pointing, the predictions of image content may be sufficiently in error so that the image processor fails to obtain valid measurements. Once again, the extent of this potential problem is unknown and must be determined through experimental trials with an operational system. The implication for initial applications is that there may be a limit to the complexity and velocity of the trajectories that the robot is allowed to follow. A good first step for implementation is to use a fixed camera and simple trajectories.

The third and final category of performance limitations discussed here regards computational hardware. The existing simulation software provides the speed for a sampling rate of approximately two frames per second. This will allow only low bandwidth motion of the robot and the use of simple navigation targets.

However, this performance level is achieved with non-optimized code and a general-purpose computing architecture. The first step to increasing performance is optimizing the code for speed, which may result in a ten to twenty percent improvement in efficiency. The second step is to move to faster hardware, which may increase the performance by another ten to twenty percent. For a serious performance increase, special purpose digitial video processing architectures are necessary. The suggested architecture is one in which the arithmetic processor has immediate access to image data so that large amounts of data do not need to be transported across general-purpose busses. Appropriate hardware is currently commercially available which could conceivably allow thirty frames per second processing for simple targets.

Better software and hardware is required for increasing frame rate, but it is also necessary for accommodating increasingly high processing overhead that is required for fault-tolerant performance. The current code for image processing associated with the experimental target is not prepared to deal with the effects of extremely poor state predictions, a partially occluded target, or other anomolies in the image sequence. Fault tolerant systems may have a processing overhead that exceeds the processing actually dedicated to performing image processing. The implication is that until fault-tolerant code can be accommodated by the computational system, initial applications will only be capable of tracking simple movements and should probably be supervised by a human operator.

A final comment on computational issues is that the processing required for the EKF will not vary a great deal. The amount of processing performed by the EKF depends significantly only on the number of measurements, which should not change much. The additional computational burden associated with complex targets, complex movements, and fault-tolerance appears in the image processing routines.

7.2 Recommended Research.

The existence of the EKF as a fundamental part of the vision navigator allows many extensions to the basic technique. One of these is to use the EKF to simultaneously estimate state and particular parameters of the system in the style of adaptive estimation. For example, a parameter associated with the camera model or dynamics model may be known only approximately. By augmenting the state vector with the parameter of interest, it can be estimated in an optimal way along with the navigation state. The measurement and dynamics operators **h**() and **f**() must be augmented accordingly, but the added components of these operators are zero when the parameter of interest is a constant.

This approach can potentially be used for refining the dynamics, measurement, or target model simultaneously with estimating state. If the estimation is successful, the augmented state variables representing constant parameters should more or less converge to the proper constant value, at which point they can be removed from the state vector and treated as constant parameters of the system.

Intuitively, however, there is no guarantee that the estimator will converge to constant values for these parameters. Also intuitively, convergence of the estimation is probably a function of how accurate the initial predictions of these parameters are. It is expected that there are limits to what one can do with this type of augmentation, because each parameter augmented to the state vector represents an overall loss of certainty in the models and an increase in fragility of the estimation process.

Investigation of these concerns and the behavior of dynamic parameter estimation with augmented systems is an exciting prospect for further research. The ultimate extension of this idea is to track a constant set of feature points on a target object whose relationship to one another is unknown but assumed rigid. The object-referenced coordinates of the points could be augmented to the state vector and in this way the structure of the object and the motion of the robot could be simultaneously obtained. In this way, the requirement of operating in a completely known environment could be eliminated.

A second extension due to the EKF framework is the fusion of various additional sensors with the vision measurements. Since the EKF is impartial to the source of its measurements, the measurement vector could be augmented with as many additional navigation sensor measurements as desired, including inertial measurements, range measurements, and depth measurements. The measurement operator **h**() would require augmentation with the appropriate relations for each of the additional measurements. The EKF would perform an optimal estimation based on all the available inputs.

Of particular interest and importance is the combination of vision and inertial measurements. For robot navigation they are extremely desirable measurements to use because they can be used both in water and in space. Also, they are complementary because each excels where the other fails to be effective. Vision measurements are generally poor for high bandwidth motion and high velocity rotations because image processing cannot keep up with the scene changes, but inertial instruments excel at measuring high accelerations and high angular velocities. Inertial navigation, on the other hand, is poor at maintaining position and attitude measurements for long periods of time because these are obtained by integration of the accelerations detected by the inertial instruments. Position and attitudes obtained this way will accumulate error increasingly over time, but vision has been shown to be excellent at providing accurate position and attitude estimates periodically, which can be used to update the inertial estimates.

Hence, vision coupled with inertial instruments can be a powerful combination for navigation of robots. The success with which humans and other animals utilize their own optical and vestibular systems for orienting themselves is living testimony to the effectiveness of the pairing. This is a research topic that can be undertaken even with the most primitive vision navigator in place because poor and unreliable vision measurements have the most to gain from adding inertial information.

The remaining research recommendations extend beyond the realm of free-flying robots. Research has already been suggested and is currently being pursued for using EKF-based vision navigation for cars and for landing planes [DICKMANNS88] and for navigating helicopters[1].

Additionally, research is currently taking place to apply a modified version of the vision navigation technique to observing the motion of human heads from a stationary camera[2]. The implications of successful implementation of this type of system is that the position and orientation of the head of a human operator relative to a computer monitor can be instantly and passively obtained by the computer. Such information is a useful user interface for such applications as interactive graphics and virtual realities. Head tracking is typically provided to computers only by burdening the user with bulky head gear.

The use of an augmented state vector, as discussed earlier in this section, for obtaining both structure and motion of the head is a plausible problem because the general shape of human heads is highly predictable. Information obtained by a computer in this way can be used for automatic identification and recognition of people, among other things.

Other applications and extensions await discovery. The principal candidates for this technology are those vision problems which are inherently dynamic and those involving observation of objects which have inertia.

[1]Part of ongoing research, Prof. Chris Atkeson, AI Laboratory, MIT.

[2]Ongoing research, Ali J. Azarbayejani, Prof. Alex Pentland, Media Laboratory, MIT.

This page not used.

References.

Altman, Simon L., *Rotations, Quaternions, and Double Groups*, Clarendon Press, Oxford, 1986.

Atkins, Ella M., *Design and Implementation of a Multiprocessor System for Positions and Attitude Control of an Underwater Robotic Vehicle*, S.M. Thesis, Department of Aeronautics and Astronautics, MIT, May 1990.

Battin, Richard H. and Gerald M. Levine, *Application of Kalman Filtering Techniques to the Apollo Program*, APOLLO Guidance, Navigation, and Control report E-2401, MIT Instrumentation Laboratory, Cambridge, MA, April 1969.

Battin, Richard H., *An Introduction to The Mathematics and Methods of Astrodynamics*, American Institute of Aeronautics and Astronautics, New York, NY, 1987.

Brown, Robert Grover, *Random Signal Analysis and Kalman Filtering*, John Wiley & Sons, New York, 1983.

Craig, John J., *Introduction to Robotics: Mechanics and Control*, Second Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

Dickmanns, Ernst Dieter and Volker Graefe, "Dynamic Monocular Machine Vision", *Machine Vision and Applications* (1988),1:223-240.

Du Val, Patrick, *Homographies, quaternions, and rotations*, Oxford, Clarendon Press, 1964.

Fischer, Otto F., *Universal Mechanics and Hamilton's Quaternions*, Axion Institute, Stockholm, 1951.

Goldstein, S., ed., *Modern Developments in Fluid Dynamics*, Dover Publications, New York, NY,1965.

Hamilton, Sir William Rowan, *Elements of Quaternions*, Second Edition, Volumes 1,2; Longmans, Green, and Company, London, 1899 (1866).

Horn, Berthold K. P., "Closed-form solution of absolute orientation using unit quaternions", *Journal of the Optical Society of America*, Vol. 4, April 1987, page 629.

Horn, Berthold Klaus Paul, *Robot Vision*, The MIT Press, Cambridge, MA, 1986.

Hughes, Peter C., *Spacecraft Attitude Dynamics*, John Wiley & Sons, New York, 1986.

Hurwitz, Adolf, *Vorlesungen über die Zahlentheorie der Quaternionen*, Julius Springer, Berlin, 1919.

Junkins, John L. and Turner, James D., *Optimal Spaceflight Rotational Maneuvers*, Elsevier, Amsterdam, 1986.

Kane, Thomas R., Likins, Peter W., Levinson, David A., *Spacecraft Dynamics*, McGraw Hill Book Company, New York, 1983.

Kelland, P. and Tait, P. G., *Introduction to Quaternions: with numerous examples*, Second Edition, Macmillan and Company, London, 1882 (1873).

Kowalski, Karl G., *Applications of a Three-Dimensional Position and Attitude Sensing System for Neutral Buoyancy Space Simulation*, S.M. Thesis, Department of Aeronautics and Astronautics, MIT, October 1989.

McRuer, Duane, Irving Ashkenas, and Graham Dunstan, *Aircraft Dynamics and Automatic Control*, Princeton University Press, Princeton, New Jersey, 1973.

Meriam, James L., *Engineering Mechanics, v. 2. Dynamics*, John Wiley & Sons, New York, 1978.

Molenbroek, P., *Anwendung der Quaternionen auf die Geometrie*, E. J. Brill, Leiden, 1893.

Nevins, J. L.; I. S. Johnson; and T. B. Sheridan, *Man/Machine Allocation in the Apollo Navigation, Guidance, and Control System*, APOLLO Guidance, Navigation, and Control report E-2305, MIT Instrumentation Laboratory, Cambridge, MA, July 1968.

Rowley, V. M., *Effects of Stereovision and Graphics Overlay on a Teleoperator Docking Task*, S.M. Thesis, Department of Aeronautics and Astronautics, MIT, August, 1989.

Schmidt, George T. and Larry D. Brock, *General Questions on Kalman Filtering in Navigation Systems*, APOLLO Guidance, Navigation, and Control report E-2406, MIT Instrumentation Laboratory, Cambridge, MA, 1969.

Spofford, J. R., *3-D Position and Attitude Measurement for Underwater Vehicles*, Space Systems Laboratory Report #21-86, MIT, December 1986.

St. John-Olcayto, Ender, *Machine Vision for Space Robotic Applications*, S.M. Thesis, Department of Aeronautics and Astronautics, MIT, May 1990.

Stanley, W. S., "Quaternion from Rotation Matrix", AIAA Journal of Guidance and Control, Vol. 1, No. 3, May 1978, pp. 223-224. [**transferred reference from JUNKINS86]

Tait, P. G., *An Elementary Treatise on Quaternions*, Third Edition [much enlarged], The University Press, Cambridge University, Cambridge, 1890 (1873) (1867).

Tarrant, J. M., *Attitude Control and Human Factors Issues in the Maneuvering of an Underwater Space Simulation Vehicle*, S.M. Thesis, Department of Aeronautics and Astronautics, MIT, August 1987.

Thompson, William Tyrrell, *Introduction to Space Dynamics*, John Wiley & Sons Inc., New York, 1963 (1961).

Viggh, Herbert, *Artificial Intelligence Applications in Teleoperated Robotic Assembly of the EASE Space Structure*, S.M. Thesis, Department of Aeronautics and Astronautics and Department of Electrical Engineering and Computer Science, MIT, February 1988.

Vigneras, *Arithmétique des Algèbres de Quaternions*, Springer-Verlag, Berlin, 1980.

Vyhnalek, G. G., *A Digital Control System for an Underwater Space Simulation Vehicle using Rate Gyro Feedback*, S.M. thesis, Department of Aeronautics and Astronautics, MIT, June 1985.

Wertz, James R. ed.and Members of the Technical Staff, Attitude Systems Operation, Computer Sciences Corporation, *Spacecraft Attitude Determination and Control*, D. Reidel Publishing Company, Dordrecht, HOLLAND, 1985 (1978).

Wiesel, William E., *Spaceflight Dynamics*, McGraw Hill Book Company, New York, 1989.

This page not used.

Appendix A. Quaternion Mathematics.

This Appendix contains the relations required for the manipulations of quaternions in this document. It is not a complete or rigorous treatment of quaternions or rotations. See the references at the end of this appendix for various treatments.

A.1 Quaternion Basics

A quaternion is an ordered set of 4 scalar quantities. Alternately it can be thought of as a set containing one scalar and one 3-vector. The quaternion can be written as a 4-vector with its elements in order

$$\dot{\mathbf{q}} = \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{pmatrix} = \begin{pmatrix} q_0 \\ \mathbf{q} \end{pmatrix},$$

or as an ordered set

$$\dot{\mathbf{q}} = (q_0, q_1, q_2, q_3) = (q_0, \mathbf{q}).$$

The first element, q_0, is the **scalar part** and the last three elements, q_1, q_2, and q_3, form the **vector part**. The quaternion itself is represented by a boldface character with a hollow circle above.

Among alternative representations of quaternions is the "complex number with three imaginary parts",

$$\dot{\mathbf{q}} = q_0 + i\, q_1 + j\, q_2 + k\, q_3$$

where the relations $i^2 = -1$, $j^2 = -1$, $k^2 = -1$, $ij = k$, $jk = i$, $ki = j$, $ji = -k$, $kj = -i$, and $ik = -j$ apply[1], or the sum of a scalar and a vector,

[1]The imaginary numbers i, j, and k are not to be confused with $\hat{i}$, $\hat{j}$, and $\hat{k}$, often used to represent unit vectors in Cartesian coordinates, although there is some relationship because the imaginary parts are normally considered the components of a 3-vector.

$$\dot{\mathbf{q}} = q_0 + \mathbf{q},$$

where vector multiplication is non-commutative and defined by $\mathbf{qr} \triangleq \mathbf{q}\times\mathbf{r} - \mathbf{q}\cdot\mathbf{r}$ (note that the product of two vectors is a quaternion, not a vector).

For both of these alternative representations, the element q_0 (the scalar part) can be referred to as the "real" part and the remaining three elements or the vector (the vector part) comprise the "imaginary" part. These "complex number" representations allow quaternion arithmetic to directly follow from regular arithmetic operations. The multiplication formula discussed later, for example, can be found directly using the defining relations for (i, j, k) above or the defining relation for vector multiplication.

The **dot product** maps two quaternions to a scalar, similar to the usual vector dot product,

$$\begin{aligned} \dot{\mathbf{p}}\cdot\dot{\mathbf{q}} &= p_0q_0 + \mathbf{p}\cdot\mathbf{q} \\ &= p_0q_0 + p_1q_1 + p_2q_2 + p_3q_3. \end{aligned}$$

The **magnitude** of a quaternion is analagous to vector magnitude (Euclidean norm),

$$\|\dot{\mathbf{q}}\|_2 = \sqrt{(\dot{\mathbf{q}}\cdot\dot{\mathbf{q}})}.$$

A **unit quaternion** has unity magnitude and can be used to represent rotation in $\Re^3$ as shown in §A.3.

Quaternions have **conjugates**, represented by a superscript asterisk:

$$\dot{\mathbf{q}}^* = \begin{pmatrix} q_0 \\ -\mathbf{q} \end{pmatrix} = \begin{pmatrix} q_0 \\ -q_1 \\ -q_2 \\ -q_3 \end{pmatrix}.$$

Note that, in analogy to conventional complex numbers, conjugation of a quaternion leaves the scalar or "real" part intact and negates the vector or "imaginary" part.

A.2 Quaternion Algebra Properties

The operations of **multiplication** and **scalar multiplication** can be meaningfully defined for the set of quaternions. Quaternion scalar multiplication maps a scalar and quaternion to a new quaternion, $\Re \times \Re^4 \rightarrow \Re^4$. This is similar to the operation of scalar multiplication in a regular linear vector space:

$$\alpha\,\dot{\mathbf{q}} = \alpha\begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{pmatrix} = \begin{pmatrix} \alpha q_0 \\ \alpha q_1 \\ \alpha q_2 \\ \alpha q_3 \end{pmatrix},$$

where $\alpha \in \Re$ and $\dot{\mathbf{q}} \in \Re^4$.

Quaternion multiplication maps two quaternions to a new quaternion: $\Re^4 \times \Re^4 \rightarrow \Re^4$. Multiplication is not commutative, and therefore the **premultiplier** and **postmultiplier** must be distinguished. The quaternion multiplication is defined as follows:

$$\dot{\mathbf{p}}\,\dot{\mathbf{q}} = \begin{pmatrix} p_0q_0-p_1q_1-p_2q_2-p_3q_3 \\ p_1q_0+p_0q_1-p_3q_2+p_2q_3 \\ p_2q_0+p_3q_1+p_0q_2-p_1q_3 \\ p_3q_0-p_2q_1+p_1q_2+p_0q_3 \end{pmatrix} = \begin{pmatrix} p_0 & -p_1 & -p_2 & -p_3 \\ p_1 & p_0 & -p_3 & p_2 \\ p_2 & p_3 & p_0 & -p_1 \\ p_3 & -p_2 & p_1 & p_0 \end{pmatrix}\begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{pmatrix} \triangleq \mathcal{P}\,\dot{\mathbf{q}}.$$

Equivalently,

$$\dot{\mathbf{p}}\,\dot{\mathbf{q}} = \begin{pmatrix} q_0 & -q_1 & -q_2 & -q_3 \\ q_1 & q_0 & q_3 & -q_2 \\ q_2 & -q_3 & q_0 & q_1 \\ q_3 & q_2 & -q_1 & q_0 \end{pmatrix}\begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix} \triangleq \bar{\mathcal{Q}}\,\dot{\mathbf{p}}.$$

As demonstrated by the above expressions, quaternion multiplication can be converted into a matrix-vector multiplication by forming a matrix from one of the multiplicands and a vector from the other. If the matrix derives from the premultiplier, as does the matrix $\mathcal{P}$ above, it is called the **Quaternion Premultiplier Matrix**[2] for that quaternion. If the matrix derives from the postmultiplier, as does the matrix $\bar{\mathcal{Q}}$ above, it is called the **Quaternion Postmultiplier Matrix** for that quaternion.

The premultiplier matrix $\mathcal{P}$ and postmultiplier matrix $\bar{\mathcal{P}}$ for a quaternion $\dot{\mathbf{p}}$ are not the same. (Otherwise multiplication would be commutative.) Specifically, $\bar{\mathcal{P}}$ varies from $\mathcal{P}$ in that the lower right (3,3) submatrix of each is transposed from the other.

A useful property is that conjugating quaternions results in transposing the multiplier matrices. That is, if $\dot{\mathbf{q}} \Rightarrow \mathcal{Q}, \bar{\mathcal{Q}}$, then $\dot{\mathbf{q}}^* \Rightarrow \mathcal{Q}^T, \bar{\mathcal{Q}}^T$.

[2]This terminology is specific to this document. The idea derives from Horn (J. Opt. Soc. Am. A, Vol. 4, No. 4, 1987, pg 629-642.) but he never gives the matrices names.

The multiplier matrix for a multiplicand is meaningful because it is the partial derivative matrix of the product with respect to the other multiplicand. That is, if $\mathring{\mathbf{r}} = \mathring{\mathbf{p}}\,\mathring{\mathbf{q}}$, then the (4,4) partial derivative matrix is

$$\frac{\partial \mathring{\mathbf{r}}}{\partial \mathring{\mathbf{q}}} = \mathcal{P}.$$

That is, the (i,j)th element of $\mathcal{P}$ is the partial derivative of the ith element of $\mathring{\mathbf{r}}$ with respect to the jth element of $\mathring{\mathbf{q}}$, $\partial r_i/\partial q_j$. Similarly,

$$\frac{\partial \mathring{\mathbf{r}}}{\partial \mathring{\mathbf{p}}} = \bar{\mathcal{Q}}.$$

A compact analytical formula for the multiplication operation is

$$\mathring{\mathbf{p}}\,\mathring{\mathbf{q}} = \begin{pmatrix} p_o q_o - \mathbf{p} \cdot \mathbf{q} \\ p_o \mathbf{q} + q_o \mathbf{p} + \mathbf{p} \times \mathbf{q} \end{pmatrix}$$

which is equivalent to the matrix equation above. This can be more useful when performing analytical manipulations, whereas the other concept is more useful for performing numerical calculations.

The quaternion algebra also has a multiplicative identity element, the **identity quaternion,**

$$\mathring{\mathbf{e}} \triangleq \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

It is easily seen from either the analytical multiplication formula or the matrix formula that $\mathring{\mathbf{e}}$ satisfies the required property of an identity element, i.e. that $\mathring{\mathbf{e}}\mathring{\mathbf{q}} = \mathring{\mathbf{q}}$ and $\mathring{\mathbf{q}}\mathring{\mathbf{e}} = \mathring{\mathbf{q}}$ for any quaternion $\mathring{\mathbf{q}}$.

Quaternions have **inverses** that satisfy $\mathring{\mathbf{q}}\mathring{\mathbf{q}}^{-1} = \mathring{\mathbf{q}}^{-1}\mathring{\mathbf{q}} = \mathring{\mathbf{e}}$. To find the inverse, notice that the product of a quaternion with its conjugate is scalar, i.e. has zero vector part,

$$\mathring{\mathbf{q}}\,\mathring{\mathbf{q}}^* = \begin{pmatrix} \mathring{\mathbf{q}} \cdot \mathring{\mathbf{q}} \\ \mathbf{0} \end{pmatrix}$$

Dividing the above equation by $\mathring{\mathbf{q}} \cdot \mathring{\mathbf{q}}$ yields (1,**0**) on the right. Hence,

$$\mathring{\mathbf{q}}^{-1} = \frac{\mathring{\mathbf{q}}^*}{\mathring{\mathbf{q}} \cdot \mathring{\mathbf{q}}}.$$

Note that, for the special case of unit quaternions, $\dot{\mathbf{q}}^{-1} = \dot{\mathbf{q}}^{*}$.

[If addition of quaternions is defined as termwise addition of the elements, as for regular vectors, then the above properties make the set of all quaternions a ***noncommutative algebra with unit element***. The set of all (n,n) matrices is also such an algebra. The set of n-vectors is not an algebra because there is no multiplication operation. Hence, many powerful properties of algebras apply to the set of quaternions; this and the relation of quaternions to physical phenomena, like 3D rotations, motivated Hamilton's intense interest in quaternions.]

A.3 Representation of Rotation

Sir Hamilton found many uses for quaternions in the analysis of physical problems. Unit quaternions were found to be useful for representing the rotation of a 3-vector in 3D space.

Quaternions can be used to compute the vector $\mathbf{x}'$ which results from rotating $\mathbf{x}$ about some axis $\hat{\mathbf{n}}$ through an angle θ in some 3D reference frame. Commonly, this rotation is computed by multiplying the original vector by a (3,3) orthonormal direction cosine matrix:

$$\mathbf{x}' = \mathbf{R}\ \mathbf{x}.$$

Alternately, Rodrigues's Formula provides a different representation of the relationship between the new vector $\mathbf{x}'$ and the original vector $\mathbf{x}$. Rodrigues's Formula describes the vector that results from rotating the original vector through an angle θ about the unit vector $\hat{\mathbf{n}}$:

$$\mathbf{x}' = \cos\theta\ \mathbf{x} + \sin\theta\ \hat{\mathbf{n}} \times \mathbf{x} + (1-\cos\theta)\ (\hat{\mathbf{n}} \cdot \mathbf{x})\ \hat{\mathbf{n}}.$$

To derive this, consider a vector $\mathbf{x}$ rotating about the axis $\hat{\mathbf{n}}$. The vector $\mathbf{x}$ consists of components along $\hat{\mathbf{n}}$ and perpendicular to it, as illustrated in Figure A.3-1. The components can be separated as

$$\mathbf{x} = (\mathbf{x}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} + (\mathbf{x} - (\mathbf{x}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}).$$

Only the second term is affected by the rotation about the axis $\hat{\mathbf{n}}$.

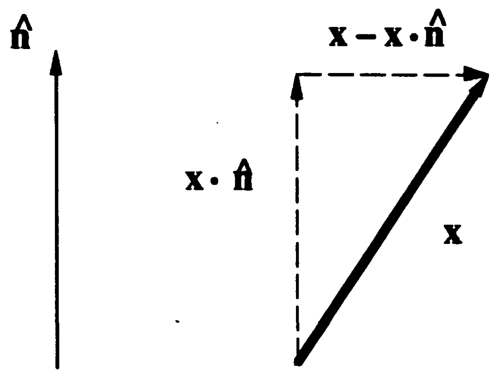


Figure A.3-1: The components of x. The component along the axis of rotation is not affected by the rotation.

After a rotation through an angle θ, the second term becomes

$$(\mathbf{x} - (\mathbf{x}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}})\ \cos\theta + \hat{\mathbf{n}} \times (\mathbf{x} - (\mathbf{x}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}})\ \sin\theta.$$

The formula of Rodrigues results from re-combining this with the unchanged $(\mathbf{x}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}$ to get $\mathbf{x}'$.[SALAMIN79]

Consider now the quaternion formula,

$$\dot{q}\ \dot{\underline{x}}\ \dot{q}^*$$

where $\dot{q}$ is a unit quaternion and $\dot{\underline{x}} = (0,\mathbf{x})$ is underlined to stress that it is a pure vector, i.e. has no scalar part. The product of the two quaternion multiplications is a quaternion with zero scalar part and a vector part equal to

$$(q_0^2 - \mathbf{q}\cdot\mathbf{q})\ \mathbf{x} + 2\ q_0\ \mathbf{q}\times\mathbf{x} + 2\ (\mathbf{q}\cdot\mathbf{x})\mathbf{q}.$$

The substitutions

$$q_0 = \cos\frac{\theta}{2}$$

and

$$\mathbf{q} = \sin\frac{\theta}{2}\,\hat{\mathbf{n}}$$

yield Rodrigues's Formula for the vector part.

Hence,

$$\underline{\mathring{x}}' = \mathring{q}\,\underline{\mathring{x}}\,\mathring{q}^*.$$

In this way, unit quaternions can be used to implement rotation of a vector. Specifically, the unit quaternion

$$\mathring{q} = [\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\hat{n}]$$

imparts a rotation to the vector **x** according to the angle θ and axis $\hat{n}$.

The reverse transformation is

$$\underline{\mathring{x}} = \mathring{q}^*\,\underline{\mathring{x}}'\,\mathring{q}$$

which results trivially from the forward transformation through premultiplication by $\mathring{q}^*$ and postmultiplication by $\mathring{q}$.

Further perspective on the operation of rotation using unit quaternions results from expanding the expression in terms of pre- and post-multiplier matrices. Using the properties of section A.2,

$$\mathring{q}\,\underline{\mathring{x}}\,\mathring{q}^* = (\mathcal{Q}\underline{\mathring{x}})\,\mathring{q}^* = \bar{\mathcal{Q}}^T\mathcal{Q}\underline{\mathring{x}}.$$

The matrix $\bar{\mathcal{Q}}^T\mathcal{Q}$ has the form,

$$\bar{\mathcal{Q}}^T\mathcal{Q} = \begin{pmatrix} \mathring{q}\cdot\mathring{q} & 0 & 0 & 0 \\ 0 & (q_0^2+q_1^2-q_2^2-q_3^2) & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 0 & 2(q_1q_2+q_0q_3) & (q_0^2-q_1^2+q_2^2-q_3^2) & 2(q_2q_3-q_0q_1) \\ 0 & 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & (q_0^2-q_1^2-q_2^2+q_3^2) \end{pmatrix}$$

$$= \begin{pmatrix} \mathring{q}\cdot\mathring{q} & \mathbf{0}^T \\ \mathbf{0} & \mathbf{R} \end{pmatrix}.$$

The lower right (3,3) matrix **R** is orthonormal and is in fact the familiar direction cosine matrix. Regardless of the representation of rotation, then, the numerical operation of rotation is the same.

However, quaternions are preferable for some analytic manipulations and for numerical manipulations. [HORN87] and [SALAMIN79] both note that composition of rotations requires less computation when quaternions are used in place of rotation matrices. And, perhaps more importantly, re-normalization of quaternions due to finite-precision calculations is trivial compared to re-normalization of a rotation matrix.

Composition of rotations is important in many applications where rotations are used. To illustrate how rotations are composed using unit quaternions, the property

$$\mathring{p}^*\mathring{q}^* = (\mathring{q}\mathring{p})^*,$$

is useful. This relation can be verified from direct evaluation of the two sides.

Now let $\mathring{p}$ describe $\mathbf{x} \rightarrow \mathbf{x}'$ and $\mathring{q}$ describe $\mathbf{x}' \rightarrow \mathbf{x}''$. Then,

$$\mathring{\underline{x}}' = \mathring{p}\,\mathring{\underline{x}}\,\mathring{p}^*$$

and

$$\begin{aligned}\mathring{\underline{x}}'' &= \mathring{q}\,\mathring{\underline{x}}'\,\mathring{q}^* \\ &= \mathring{q}\,\mathring{p}\,\mathring{\underline{x}}\,\mathring{p}^*\,\mathring{q}^* \\ &= \mathring{r}\,\mathring{\underline{x}}\,\mathring{r}^*\end{aligned}$$

where $\mathring{r} = \mathring{q}\,\mathring{p}$. Thus rotations are composed by multiplying the quaternions associated with each rotation.

A.4 Rotation Quaternions for Vehicle Dynamics

For describing vehicle dynamics it is necessary to relate the rotational state, described by a quaternion, to the rotational velocity of the vehicle.

Consider the use of a unit quaternion to represent the rotational state of a free-flying body with respect to a some fixed reference. Define a body coordinate frame fixed to the body and a fixed coordinate frame fixed to the reference. Designate the vector $\mathbf{x}$ expressed in the body frame as $\mathbf{x}_{:b}$ and in the fixed frame as $\mathbf{x}_{:f}$.

The quaternion $\mathring{q}$ describing attitude of the body obeys

$$\mathring{\underline{x}}_{:f} = \mathring{q}\,\mathring{\underline{x}}_{:b}\,\mathring{q}^*.$$

The **time derivative of quaternions** must be related to rotational velocity for dynamic modelling using quaternions. Consider any fixed vector $\mathbf{x}$ <u>stationary relative to the fixed frame</u>.

If the body moves relative to the fixed frame, the vector will appear (from the body) to move an equal magnitude in the opposite direction. Let

$\dot{\underline{\mathbf{x}}}_{:f}$ = x in fixed coordinates (constant),

$\dot{\underline{\mathbf{x}}}_{:b}^{(t)}$ = x measured in body coordinates at time t, and

$\dot{\mathbf{q}}(t)$ = the rotational relationship of body to fixed frame at time t.

The rotation formula at time t is

$$\dot{\underline{\mathbf{x}}}_{:f} = \dot{\mathbf{q}}(t)\, \dot{\underline{\mathbf{x}}}_{:b}^{(t)}\, \dot{\mathbf{q}}^*(t).$$

This leads to

$$\dot{\underline{\mathbf{x}}}_{:b}^{(t)} = \dot{\mathbf{q}}^*(t)\, \dot{\underline{\mathbf{x}}}_{:f}\, \dot{\mathbf{q}}(t).$$

at time t.

Now consider an instantaneous 3D rotational velocity $\mathbf{w}_{:b} = \omega_0\, \hat{\mathbf{n}}$ of the vehicle. The velocity is expressed <u>in the body frame</u>. Consider a small time interval [t,t+Δt], during which the incremental rotation of the vehicle can be approximated as rotation through an angle $\omega_0\Delta t$ about the axis $\hat{\mathbf{n}}$. Equivalently, the rotation can be expressed by the unit quaternion

$$\delta\dot{\mathbf{q}} = \left(\cos(\omega_0\Delta t/2),\ \sin(\omega_0\Delta t/2)\, \hat{\mathbf{n}}\right).$$

During this time interval the fixed vector $\mathbf{x}_{:b}$ undergoes incremental rotation $\delta\dot{\mathbf{q}}^*$, i.e. rotation by angle $-\omega_0\Delta t$ about $\hat{\mathbf{n}}$, from the vehicle's standpoint. Hence,

$$\dot{\underline{\mathbf{x}}}_{:b}^{(t+\Delta t)} = \delta\dot{\mathbf{q}}^*\, \dot{\underline{\mathbf{x}}}_{:b}^{(t)}\, \delta\dot{\mathbf{q}} = \delta\dot{\mathbf{q}}^*\, \dot{\mathbf{q}}^*(t)\, \dot{\underline{\mathbf{x}}}_{:f}\, \dot{\mathbf{q}}(t)\, \delta\dot{\mathbf{q}}$$

But by definition,

$$\dot{\underline{\mathbf{x}}}_{:b}^{(t+\Delta t)} = \dot{\mathbf{q}}^*(t+\Delta t)\, \dot{\underline{\mathbf{x}}}_{:f}\, \dot{\mathbf{q}}(t+\Delta t).$$

Therefore,

$$\dot{\mathbf{q}}(t+\Delta t) = \dot{\mathbf{q}}(t)\, \delta\dot{\mathbf{q}}.$$

The derivative of the quaternion follows straightforwardly:

$$\frac{d\dot{\mathbf{q}}}{dt} = \lim_{\Delta t \to 0} \frac{\dot{\mathbf{q}}(t+\Delta t) - \dot{\mathbf{q}}(t)}{\Delta t}$$

$$= \lim_{\Delta t \to 0} \frac{1}{\Delta t} \mathring{\mathbf{q}}(t) \left(\delta\mathring{\mathbf{q}} - \mathring{\mathbf{e}} \right)$$

Now as $\Delta t \to 0$, the limits $\cos(\omega_0 \Delta t/2) \to 1$ and $\sin(\omega_0 \Delta t/2) \to \omega_0 \Delta t/2$, yield

$$\frac{d\mathring{\mathbf{q}}}{dt} = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \left(q_0(t), \mathbf{q}(t) \right) \left(0, (\omega_0 \Delta t/2)\hat{\mathbf{n}} \right)$$

$$= \left(q_0(t), \mathbf{q}(t) \right) \left(0, \frac{1}{2} \mathbf{w}_{:b} \right)$$

$$= \left(\frac{1}{2} \mathbf{q}(t) \cdot \mathbf{w}_{:b} \, , \; \frac{1}{2} \left(q_0(t) \mathbf{w}_{:b} + \mathbf{q}(t) \times \mathbf{w}_{:b} \right) \right)$$

This derivative is useful for dynamics of flying bodies when the rotational velocity in the body frame is known.

References

Battin, Richard H., *An Introduction to The Mathematics and Methods of Astrodynamics*, American Institute of Aeronautics and Astronautics, New York, NY, 1987.

Horn, Berthold Klaus Paul, *Robot Vision*, The MIT Press, Cambridge, MA, 1986.

Horn, Berthold Klaus Paul, "Closed-form solution of absolute orientation using unit quaternions", Journal of the Optical Society of America A, Vol. 4, No. 4, April 1987, page 629.

Hughes, Peter C., *Spacecraft Attitude Dynamics*, John Wiley & Sons, New York, 1986.

Salamin, E., "Application of quaternions to computation with rotations", Internal Report (Stanford University, Stanford, California, 1979).

Appendix B. Source Code.

The following pages contain source code listings for the routines used on the Macintosh for the dynamic simulations and on the Gateway2000 PC (IBM Clone) for image processing. The intent of this listing is to document the details of the implementation of the vision navigator technique and to provide a reference for future users of the programs in the laboratory. The machine specific user interface routines are mostly omitted because they are not relevant to the subject matter of the thesis.

```
/*                  k2.h
                    KalmanVision Header File (kalmanVision)
                    Ali J. Azarbayejani
                    December 1990
                    © Copyright 1990. All rights reserved.

        Contains constants related to the Extended Kalman Filter Loop.
        The companion header file "kvPlus.h" includes all of the information
        related to the Graphics Simulation.
*/

#include     <stdio.h>
#include     <math.h>

/*                              Parameters
*/
#define      NFP      20        /* NUMBER OF FEATURE POINTS     */
#define      NKF      4         /* NUMBER OF FPs USED IN EKF    */
#define      NM       2*NKF     /* NUMBER OF MEASUREMENTS       */
#define      NS       12        /* NUMBER OF KF STATE ELEMENTS  */
#define      RS       16        /* NUMBER OF RVN STATES         */
#define      DT       0.033     /* DISCRETE TIME STEP (SECONDS) */

/*                              Camera Model
*/
#define      FL       0.1       /* PRINCIPAL DISTANCE (mm)      */
#define      PPM      10000     /* PIXELS PER METER             */

/*                              Noise Function Constants
*/
#define      VMODE    1         /* NOISE FUNCTIONS USE VARIANCE */
#define      AMODE    2         /*                      SPREAD  */

/*                              Physical Constants
*/
#define      CF1      490       /* kg/m          */
#define      CF2      490       /* kg/m          */
#define      CF3      490       /* kg/m          */
#define      CFR      200       /* kg m^2        */
#define      CFP      200       /* kg m^2        */
#define      CFY      200       /* kg m^2        */
#define      I11      100       /* kg m^2        */
#define      I22      100       /* kg m^2        */
#define      I33      100       /* kg m^2        */
#define      M        1000      /* kg            */

#define      CFM1     (-0.98)   /* m^-1                 -2 * CF1 / M        */
#define      CFM2     (-0.98)   /* m^-1                 -2 * CF2 / M        */
#define      CFM3     (-0.98)   /* m^-1                 -2 * CF3 / M        */
#define      CFRI1    (-4)      /*                      -2 * CFR / I11      */
#define      CFPI2    (-4)      /*                      -2 * CFP / I11      */
#define      CFYI3    (-4)      /*                      -2 * CFY / I11      */
#define      I23I1    0.0       /*                      (I22 - I33) / I11   */
#define      I31I2    0.0       /*                      (I33 - I11) / I22   */
#define      I12I3    0.0       /*                      (I11 - I22) / I33   */
#define      M1I      0.001     /* kg^-1                1 / M               */
#define      M2I      0.001     /* kg^-1                1 / M               */
#define      M3I      0.001     /* kg^-1                1 / M               */
```

```
#define     I11I    0.01    /* kg^-1 m^-2       1 / I11             */
#define     I22I    0.01    /* kg^-1 m^-2       1 / I22             */
#define     I33I    0.01    /* kg^-1 m^-2       1 / I33             */

/*                          Function Prototypes
*/
void            kvLinMeas();
void            kvGain();
void            kvPredMeas();
void            kvEstState();
void            kvEstCov();
void            kvSTM();
void            kvPredState();
void            kvPredCov();
void            ludcmp(double **,int,int *,double *);
void            lubksb(double **,int,int *,double *);
double          *vector(int);
void            nrerror(char *);
void            free_vector(double *);
double          **matrix(int,int);
void            free_matrix(double **);
void            kvInitVar(int);
void            kvNextState(int);
void            kvFilter();
void            kvPjctIm();
void            qLeft(double *,double **);
void            qRight(double *,double **);
void            qMult(double *,double *,double *);
void            qRotMat(double *,double **);
void            qCRotMat(double *,double **);
void            qConjugate(double *,double *);
void            qRotate(double *,double *,double *);
void            qNormalize(double *);
double          kvUnifNoise(char,double,double,double);

/*                      External Variable Declarations - Kalman Filter variables
*/
extern  double          *xp;            /* State Vector prediction              */
extern  double          **Pp;           /* Error Covariance Matrix prediction   */
extern  double          *xe;            /* State Vector estimate                */
extern  double          **Pe;           /* Error Covariance Matrix estimate     */
extern  double          *ya;            /* Measurement Vector                   */
extern  double          *yp;            /* Measurement Vector prediction        */
extern  double          **HH;           /* Measurement Matrix (linearized h)    */
extern  double          **KK;           /* Kalman Gain Matrix                   */
extern  double          **STM;          /* State Transition Matrix              */
extern  double          **RR,**QQ;      /* Noise Covariance Matrices            */
extern  double          *uu;            /* Command Vector                       */
extern  double          *xa;            /* Actual State Vector                  */
extern  double          **ptg;          /* Scene pts, global coords             */
extern  double          *rq;            /* Body/Camera Rotation Quaternion      */
extern  double          *tb;            /* Body/Camera Translation (Body Frm)   */
extern  double          *tc;            /* Body/Camera Translation (Cam Frm)    */
```

```
/*                      kvEstCov.c
                        Estimate Error Covariance Matrix (kalmanVision)
                        Ali J. Azarbayejani
                        December 1990
                        © Copyright 1990. All rights reserved.
*/
#include "k2.h"

void kvEstCov()
/*      Pp  "predicted error covariance matrix"         [input]
        K   "Kalman Gain matrix"                        [input]
        H   "Linearized measurement matrix"             [input]
        Pe  "estimated error covariance matrix at t0"   [output]

        This function computes Pe = (I-KH)Pp
*/
{
    double                  **m1;
    register double         temp;
    int                     i,j,k;

    m1 = matrix(NS,NS);

    for(i=0;i<NS;i++)
        for(j=0;j<NS;j++) {
            for(k=0,temp=0;k<NM;k++)
                temp += KK[i][k]*HH[k][j];
            m1[i][j] = -temp;
        }
    for(i=0;i<NS;i++)
        m1[i][i] += 1;

    for(i=0;i<NS;i++)
        for(j=0;j<NS;j++) {
            for(k=0,temp=0;k<NS;k++)
                temp += m1[i][k]*Pp[k][j];
            Pe[i][j] = temp;
        }
    free_matrix(m1);
}
```

```
/*                   kvEstState.c
                     Estimate State Vector (kalmanVision)
                     Ali J. Azarbayejani
                     December 1990
                     © Copyright 1990. All rights reserved.
*/
#include "k2.h"
#include "k2Plus.h"

void kvEstState()
/*      xp  "predicted state vector for time t0"        [input]
        K   "Kalman Gain matrix"                        [input]
        y   "actual measurement vector for time t0"     [input]
        yp  "predicted measurement vector for time t0"  [input]
        xe  "estimated state vector for time t0"        [output]

        This function computes xe = xp + K(y-yp)
*/
{
    int                 i,j;
    double              *v1,*v2,b[4];
    register double     temp,magq;

    v1 = vector(NM);    v2 = vector(NS);

    for(i=0;i<NM;i++)
        v1[i] = ya[i] - yp[i];
    for(i=0;i<NS;i++) {
        for(j=0,temp=0;j<NM;j++)
            temp += KK[i][j]*v1[j];
        v2[i] = temp;
    }
    for(i=0;i<NS;i++)
        xe[i]=xp[i]+v2[i];

    b[1] = xe[3]/2.0;   b[2] = xe[4]/2.0;   b[3] = xe[5]/2.0;
    b[0] = sqrt(1 - b[1]*b[1] + b[2]*b[2] + b[3]*b[3]);
    qMult(xp+12,b,xe+12);

    /*      Normalize the quaternion
    */
    qNormalize(xe+12);

    /*      ZERO THE EULER ANGLES ESTIMATE
            The quaternion now holds the estimated rotational state
    */
    xe[3]=0;    xe[4]=0;    xe[5]=0;

    free_vector(v1);    free_vector(v2);
}
```

```
/*                  kvFilter.c
                    Kalman Filtering for Vision Exterior Orientation
                    Ali J. Azarbayejani
                    December 1990
                    © Copyright 1990.  All rights reserved.
*/
#include "k2.h"

void kvFilter()
/*  On Entry:    xp   points to "predicted state for time t0"
                 Pp   points to "predicted error covariance for time t0"
                 y    points to "measurement at time t0"
    During Execution:
                 yp   points to "predicted measurement at time t0"
                 H    points to "matrix of dh/dx for linearization of h(x) around xp"
                 K    points to "matrix of Kalman Gain for time t0"
                 stm  points to "state transition matrix for t0 --> t1"
                 R    points to "error covariance of measurement noise"
                 Q    points to "error covariance of dynamics noise"
    On Exit:     xe   points to "estimated state for time t0"
                 Pe   points to "estimated error covariance for time t0"
                 xp   points to "predicted state for time t1"
                 Pp   points to "predicted error covariance for time t1"
    These are all external variables and are defined in "k2.h"
*/
{
    /*  First, linearize measurement equation around predicted state at t0...
    */
    kvLinMeas();                              /* xp,NKF,ptg ==> H */

    /*  ...then compute Kalman gain using linearized H.
    */
    kvGain();                                 /* Pp,H,R ==> K */

    /*  Predict the measurement at t0 using predicted state xp0...
    */
    kvPredMeas();                             /* xp ==> yp */

    /*  ...then estimate the state at t0 using K, yp0, and actual measurement...
    */
    kvEstState();                             /* y,yp,K,xp ==> xe */

    /*  ...and estimate the covariance matrix for t0.
    */
    kvEstCov();                               /* Pp,K,H,R ==> Pe */

    /*  Compute the State Transition Matrix for projecting covariance...
    */
    kvSTM();                                  /* xe ==> stm */

    /*  ...and predict the state at t1...
    */
    kvPredState();                            /* xe ==> xp */

    /*  ...and use STM to compute the error covariance matrix at t1.
    */
    kvPredCov();                              /* Pe,stm,Q ==> Pp */
}
```

```
/*                    kvGain.c
                      Compute Kalman Gain Matrix (kalmanVision)
                      Ali J. Azarbayejani
                      December 1990
                      © Copyright 1990. All rights reserved.
*/
#include "k2.h"

void kvGain()
/*      Pp  "predicted error covariance for time t0"     [input]
        H   "matrix of dh/dx at xp"                      [input]
        R   "error covariance of measurement noise"      [input]
        K   "matrix of Kalman Gain for time t0"          [output]
    This function computes K = PH'(HPH' + R)^-1
*/
{
    double              **m1,**m2,**m3;
    int                 i,j,k;
    register double     sum;

    m1 = matrix(NS,NM); m2 = matrix(NM,NM); m3 = matrix(NM,NM);

    /*      First, form product Pp H'...
    */
    for(i=0;i<NS;i++)
        for(j=0;j<NM;j++) {
            for(k=0,sum=0;k<NS;k++)
                sum += Pp[i][k]*HH[j][k];
            m1[i][j] = sum;
        }

    /*      ...then premultiply by H: H Pp H'...
    */
    for(i=0;i<NM;i++)
        for(j=0;j<NM;j++) {
            for(k=0,sum=0;k<NS;k++)
                sum += HH[i][k]*m1[k][j];
            m2[i][j] = sum;
        }

    /*      ...then add R: H Pp H' + R and invert...
    */
    for(i=0;i<NM;i++)
        for(j=0;j<NM;j++)
            m2[i][j] += RR[i][j];
    invmat(m2,NM,m3);

    /*      ...and, finally, premultiply by Pp H'.
    */
    for(i=0;i<NS;i++)
        for(j=0;j<NM;j++) {
            for(k=0,sum=0;k<NM;k++)
                sum += m1[i][k]*m3[k][j];
            KK[i][j] = sum;
        }
    free_matrix(m1);        free_matrix(m2);        free_matrix(m3);
}
```

```
/*                  kvInitProj.c
                    Initialize Variables (kalmanVision)
                    Ali J. Azarbayejani
                    December 1990
                    © Copyright 1990. All rights reserved.
*/
#include "k2.h"
#include "k2Plus.h"

#define     PI      3.141592654

#define     VART    0       /* Variance of translational velocities    */
#define     VARB    0       /* Variance of angular velocities          */
#define     VARV    1e-4    /* Variance of translational accelerations */
#define     VARW    1e-4    /* Variance of rotational accelerations    */
#define     VARY    1e-8    /* Variance of pixel measurements          */

void kvInitVar(trajmode)
/*      xa  "actual state vector"                       [output]
        xp  "predicted state vector"                    [output]
        Pp  "predicted error covariance matrix"         [output]
        QQ  "dynamics noise covariance matrix"          [output]
        RR  "measurement noise covariance matrix"       [output]
        HH  "measurement matrix"                        [output]
        rq  "Camera/Body rotation quaternion"           [output]
        tb  "Camera/Body translation (Body Frame)"      [output]
        tc  "Camera/Body translation (Camera Frame)"    [output]
*/
{
    int         i,j;
    double      ct2,st2,mag;
    double      th,w1,w2,w3;
    double      *dptr;
    double      r0,r1,r2,r3;
    double      c1,c2,c3;

    /*      SPECIFY ROTATIONAL IC FOR EACH TRAJECTORY (angle/axis)
    */
    switch(trajmode) {
        case xOnlyCmd:  th=0.0;     w1=1.0;     w2=0.0;     w3=0.0;     break;
        case yOnlyCmd:  th=PI/6;    w1=0.0;     w2=0.0;     w3=1.0;     break;
        case rOnlyCmd:  th=7*PI/6;  w1=5.0;     w2=1.0;     w3=(-1.0);  break;
        case wOnlyCmd:  th=0.2*PI;  w1=0.0;     w2=0.0;     w3=(-1.0);  break;
    }
    /*      COMPUTE THE QUATERNION
    */
    mag = sqrt(w1*w1 + w2*w2 + w3*w3);
    if(fabs(mag) < 1e-20) {
        mag = 1.0;
        th = 0.0;
    }
    ct2 = cos(th/2.0);
    st2 = sin(th/2.0);
    xa[12] = ct2;
    xa[13] = st2*w1/mag;
    xa[14] = st2*w2/mag;
    xa[15] = st2*w3/mag;
    /*      SPECIFY TRANS POS AND VEL, ROT VEL, AND CMD ICs FOR EACH TRAJ
    */
    switch(trajmode) {
        case xOnlyCmd:
            xa[0] = 0.0;     xa[1] = 0.0;        xa[2] = (-0.5);
            xa[6] = 1.0;     xa[7] = 0.0;        xa[8] = 0.0;
            xa[9] = 0.0;     xa[10] = 0.0;       xa[11] = 0.0;
            uu[0] = 60.0;    uu[1] = 0.0;        uu[2] = 0.0;
            uu[3] = 0.0;     uu[4] = 0.0;        uu[5] = 0.0;
            break;
        case yOnlyCmd:
            xa[0] = 1.0;     xa[1] = (-1.7);     xa[2] = 0.35;
            xa[6] = 0.2;     xa[7] = 0.1;        xa[8] = 0.0;
            xa[9] = 0.8;     xa[10] = 0.0;       xa[11] = 0.0;
            uu[0] = 50.0;    uu[1] = 20.0;       uu[2] = -15.0;
            uu[3] = 50.0;    uu[4] = 0.0;        uu[5] = 0.0;
            break;
        case rOnlyCmd:
            xa[0] = 2.0;     xa[1] = (-1.0);     xa[2] = 0.25;
            xa[6] = 0.0;     xa[7] = 0.0;        xa[8] = 0.0;
            xa[9] = 0.0;     xa[10] = 0.0;       xa[11] = 0.0;
            uu[0] = 0.0;     uu[1] = 0.0;        uu[2] = 0.0;
            uu[3] = 0.0;     uu[4] = 0.0;        uu[5] = 0.0;
            break;
        case wOnlyCmd:
            xa[0] = 0.0;     xa[1] = 0.0;        xa[2] = 0.0;
            xa[6] = 0.0;     xa[7] = 0.0;        xa[8] = 0.0;
            xa[9] = 0.0;     xa[10] = 0.0;       xa[11] = 0.5;
            uu[0] = 0.0;     uu[1] = 0.0;        uu[2] = 0.0;
            uu[3] = 0.0;     uu[4] = 0.0;        uu[5] = 40.0;
            break;
    }
    xa[3]=0;     xa[4]=0;     xa[5]=0;

    /*          INITIALIZE xp: PREDICTED STATE VECTOR
    */
    for(i=0;i<RS;i++)
        xp[i] = xa[i];

    /*          ADD NOISE ONLY IF OPTION IS SELECTED
    */
/*  if(iNoiseFlag) {
        for(i=0;i<RS;i++) {
            register double         AA;

            AA = sqrt(3*varI);
            if(i>5) {
                if(i<9) AA *= 0.4;
                else if(i<12) AA *= 0.3;
                else AA *= 0.1;
            }
            xp[i] += kvUnifNoise(AMODE,0,0,AA);
        }
        qNormalize(xp+12);
    }*/
    /*      Here is the code for implementing a manual controlled init cond error.
            This is not the most elegant way to do multiple experiments.
    */
    if(iNoiseFlag) {
        long        level;

        level = iNoiseFlag - iNoCmd;
        xp[0] += level*0.25;    xp[1] += level*0.15;        xp[2] -= level*0.05;
        xp[6] = 0.0;     xp[7] = 0.0;          xp[8] = 0.0;
        xp[9] = 0.0;     xp[10] = 0.0;         xp[11] = 0.0;
        for(i=12;i<16;i++)
            xp[i] += kvUnifNoise(AMODE,0,0,level*0.06);
        qNormalize(xp+12);
    }

    /*      INITIALIZE Pp: predicted error covariance matrix
    */
    for(i=0;i<NS;i++)
        for(j=0;j<NS;j++)
            Pp[i][j] = 0.0;

    /*      INITIALIZE QQ: dynamics covariance matrix
    */
    for(i=0;i<NS;i++)
        for(j=0;j<NS;j++)
            QQ[i][j] = 0.0;

    for(i=0;i<3;i++)
        QQ[i][i] = VART;
    for(i=3;i<6;i++)
        QQ[i][i] = VARB;
    for(i=6;i<9;i++)
        QQ[i][i] = varD;
    for(i=9;i<12;i++)
        QQ[i][i] = 5*varD;

    /*      INITIALIZE RR: measurement covariance matrix
    */
    for(i=0;i<NM;i++)
        for(j=0;j<NM;j++)
            RR[i][j] = 0.0;
    for(i=0;i<NM;i++)
        RR[i][i] = varY;

    /*      INITIALIZE HH: linearized measurement matrix
    */
    for(i=0;i<NM;i++)
        for(j=0;j<NS;j++)
            HH[i][j] = 0.0;

    /*      INITIALIZE Camera/Body offset r quaternion
    */
    rq[0] = 1.0;     rq[1] = 0.0;     rq[2] = 0.0;     rq[3] = 0.0;
    tb[0] = 0.0;     tb[1] = 0.0;     tb[2] = (-0.3);

    dptr = vector(4);
    dptr[0]=rq[0];
    for(i=1;i<4;i++) dptr[i]=(-rq[i]);
    qRotate(tb,dptr,tc);
    free_vector(dptr);
    /*      PRINT LOOP MESSAGE (Macintosh II-specific code. ROM Traps required)
    */
    {
        int          myFont;

        SetPort(wPtr[backW]);
        GetFNum("\pTimes",&myFont);
        TextFont(myFont);        TextSize(10);
        ForeColor(whiteColor);
        MoveTo(20,240);          DrawString("\pCmd-MouseClick aborts");
        MoveTo(20,255);          DrawString("\pMouseClick pauses/restarts");
    }
}
```

```
/*                    kvKalman.c
                      Run Discrete Simulation (kalmanVision)
                      Ali J. Azarbayejani
                      December 1990
                      © Copyright 1990. All rights reserved.
*/
#include "k2.h"
#include "k2Plus.h"

void kvKalman(int trajmode)
/*  IMPLEMENTS EXTENDED KALMAN FILTER FOR ROBOT VISION NAVIGATION
*/
{
    int                 kt;
    EventRecord         theEvent,dummy;
    WindowPtr           whichWindow;
    Boolean             doneLoop;
    OSErr               theErr;
    int                 fRN;

    /*      Allocate buffers for double arrays:
    */
    xp = vector(RS);            Pp = matrix(NS,NS);
    xe = vector(RS);            Pe = matrix(NS,NS);
    ya = vector(2*NFP);         yp = vector(NM);
    HH = matrix(NM,NS);
    KK = matrix(NS,NM);
    STM = matrix(NS,NS);
    RR = matrix(NM,NM);         QQ = matrix(NS,NS);
    uu = vector(6);             xa = vector(RS);
    rq = vector(4);
    tb = vector(3);             tc = vector(3);

    /*      If SAVE option is selected, open file for data...
    */
    if(storeFlag) {
        char                buf[256];   /* format buffer    */
        Point               dlgLoc;     /* ul corner of dlg */
        SFReply             dlgRply;    /* reply record     */
        long                cnt;        /* char counter     */
        char                *theCName;  /* file name        */

        dlgLoc.h = 265;     dlgLoc.v = 25;
        SFPutFile(dlgLoc,"\pSave DATA in:","\pkvDATA",0,&dlgRply);
        if(!dlgRply.good) return;
        if(theErr=Create(dlgRply.fName,dlgRply.vRefNum,'AliA','TEXT'))
            nrerror("Create failed, dude...");
        if(theErr=FSOpen(dlgRply.fName,dlgRply.vRefNum,&fRN))
            nrerror("FSOpen error, dude...");
        cnt=sprintf(buf,"t1-act,t1-est,t2-act,t2-est,t3-act,t3-est,");
        cnt+=sprintf(buf+cnt,"v1-act,v1-est,v2-act,v2-est,v3-act,v3-est,");
        cnt+=sprintf(buf+cnt,"w1-act,w1-est,w2-act,w2-est,w3-act,w3-est,");
        cnt+=sprintf(buf+cnt,"q0-act,q0-est,q1-act,q1-est,q2-act,q2-est,q3-act,q3-
est\n");
        if(theErr=FSWrite(fRN,&cnt,buf))
            nrerror("FSWrite error, dude...");
    }
```

```
/*      Initialize Kalman Filter Values [==>xp(0),Pp(0),...]
*/
kvInitVar(trajmode);

for(kt=0,doneLoop=FALSE;kt<NSteps && doneLoop==FALSE;kt++) {    /* EKF LOOP */
    SystemTask();

    /*      Project image to get measurement vector y [xa(t0)==>y(t0)]
    */
    kvPjctIm();

    /*      Run Kalman Filter loop [xp(t0),Pp(t0),y(t0)==>xe(t0),xp(t1),Pp(t1)]
    */
    kvFilter();

    /*      Place Camera View, XY-Plane, and YZ-Plane graphics on screen
            [   y(t0) ==> Camera View
                xa(t0) ==>  Actual locations (green axes) in XY- and YZ-Planes
                xe(t0) ==>  Estimated locations (bl/pink/yel axes) in XY and YZ]
    */
    kvPlaceGrph();
{
char        outString[50];
int         myFont;

GetFNum("\pTimes",&myFont);
SetPort(wPtr[backW]);
sprintf(outString,"%3d",kt);
CtoPstr(outString);
TextSize(18);   TextFont(myFont);
MoveTo(570,40); ForeColor(whiteColor);  DrawString(outString);
}

    /*      If SAVEing, write data to file [xa(t0),xe(t0) ==> FILE]
    */
    if(storeFlag)
        kvStore(fRN);

    /*      Cmd-Click breaks the loop; Click alone stops/restarts
    */
    if(GetNextEvent((mDownMask|updateMask),&theEvent))
        switch(theEvent.what) {
            case mouseDown:
                if(!BitAnd(theEvent.modifiers,cmdKey)) {
                    while((!GetNextEvent(mDownMask,&theEvent)
                    || (FindWindow(theEvent.where,&whichWindow)!=inContent)))
                        SystemTask();
                }
                else
                    doneLoop = TRUE;
                break;
            case updateEvt:
                CheckUpdate(&dummy);
                break;
        }

    /*      Compute next simulation state vector xa(t1)
    */
```

```
        kvNextState(kt);
    }

    /*      Free allocated buffers
    */
    free_vector(xp);            free_matrix(Pp);
    free_vector(xe);            free_matrix(Pe);
    free_vector(ya);            free_vector(yp);
    free_matrix(HH);
    free_matrix(KK);
    free_matrix(STM);
    free_matrix(RR);            free_matrix(QQ);
    free_vector(uu);
    free_vector(xa);
    free_vector(tb);            free_vector(tc);
    free_vector(rq);

    /*      If SAVEing, close file
    */
    if(storeFlag)
        FSClose(fRN);

    /*      Explicitly erases loop message and updates background
    */
    {
        Rect                        qRect;
        SetPort(wPtr[backW]);
        qRect = (*wPtr[backW]).portRect;
        DrawPicture(wPicHndl[backW],&qRect);
    }
}
```

```
/*                      kvLinMeas.c
                        Linearize Measurment Matrix (kalmanVision)
                        Ali J. Azarbayejani
                        December 1990
                        © Copyright 1990. All rights reserved.
*/
#include "k2.h"

void kvLinMeas()
/*      xp  points to "predicted state for time t0"     [input]
        NKF is number of feature points                 [input]
        ptg array of global feature pt coords           [input]
        H   points to "measurement matrix"              [output]
*/
{
    double              *q,*s,*t;
    double              pc[3],pc00,dt[3];
    double              s00,s01,s02,s03,s11,s12,s13,s22,s23,s33;
    double              c1,c2,c3,c4,c5,c6,c7;
    double              **M1,**M2,**M3,**M4,**rRMM;
    register double     temp;
    double              *dp;
    int                 i,j,k,l;

    M1 = matrix(3,3);   M2 = matrix(3,4);   M3 = matrix(3,4);   M4 = matrix(4,3);
    rRMM = matrix(4,4); s = vector(4);

/*      INITIALIZE VARIABLES...
*/
    t = xp;     q = xp+12;      qMult(q,rq,s);

    s00 = s[0]*s[0];    s01 = s[0]*s[1];    s02 = s[0]*s[2];    s03 = s[0]*s[3];
                        s11 = s[1]*s[1];    s12 = s[1]*s[2];    s13 = s[1]*s[3];
                                            s22 = s[2]*s[2];    s23 = s[2]*s[3];
                                                                s33 = s[3]*s[3];
    c1 = s00-s11-s22-s33;
    c2 = 2*s[0];

    for(i=0;i<NKF;i++) {        /* FOR EACH FEATURE POINT */

    /*      INITIALIZE SOME COMBINATIONS
    */
        for(j=0;j<3;j++) dt[j] = ptg[i][j] - t[j];
        c3 =  s[1]*dt[0] + s[2]*dt[1] + s[3]*dt[2];
        c4 =  s[0]*dt[0] + s[3]*dt[1] - s[2]*dt[2];
        c5 = -s[3]*dt[0] + s[0]*dt[1] + s[1]*dt[2];
        c6 =  s[2]*dt[0] - s[1]*dt[1] + s[0]*dt[2];

    /*      COMPUTE COORDINATES of pt i in cam coords at predicted state...
    */
        dp = vector(4);
        qConjugate(s,dp);
        qRotate(dt,dp,pc);
        free_vector(dp);
        for(j=0;j<3;j++) pc[j] -= tc[j];

        pc00 = pc[0]*pc[0];
/*      FIND ∂p/∂t (leftmost 2,3 submatrix of Hj)
*/
    dp = M1[0];
    *dp++=(-s00-s11+s22+s33);   *dp++=(-2*(s03+s12));       *dp++=(2*(s02-s13));
    *dp++=(2*(s03-s12));      *dp++=(-s00+s11-s22+s33);   *dp++=(-2*(s01+s23));
    *dp++=(-2*(s02+s13));     *dp++=(2*(s01-s23));        *dp=(-s00+s11+s22-s33);

    for(j=0;j<3;j++) {
        HH[2*i][j] = FL*(M1[1][j]*pc[0] - M1[0][j]*pc[1])/pc00;
        HH[2*i+1][j] = FL*(M1[2][j]*pc[0] - M1[0][j]*pc[2])/pc00;
    }

/*      FIND ∂p/∂Ω = [∂p/∂s][∂s/∂q][∂q/∂Ω]   (next 2,3 submatrix of HHj)
        DO:
            M2 = (1/2)[∂p/∂s]
            M3 = M2 [∂s/∂q]
            M4 = (2)[∂q/∂Ω]
            M1 = M3 M4
*/
dp = M2[0];
*dp++=c4;   *dp++=c3;   *dp++=(-s[2]*dt[0]+s[1]*dt[1]-s[0]*dt[2]);  *dp++=c5;
*dp++=c5;   *dp++=c6;   *dp++=c3;   *dp++=(-s[0]*dt[0]-s[3]*dt[1]+s[2]*dt[2]);
*dp++=c6;   *dp++=(s[3]*dt[0]-s[0]*dt[1]-s[1]*dt[2]);   *dp++=c4;   *dp++=c3;

    qRight(rq,rRMM);
    for(j=0;j<3;j++)
        for(k=0;k<4;k++) {
            for(l=0,temp=0;l<4;l++)
                temp += M2[j][l] * rRMM[l][k];
            M3[j][k] = temp;
        }

    dp = M4[0];
    *dp++=(-q[1]);  *dp++=(-q[2]);  *dp++=(-q[3]);
    *dp++=(q[0]);   *dp++=(-q[3]);  *dp++=(q[2]);
    *dp++=(q[3]);   *dp++=(q[0]);   *dp++=(-q[1]);
    *dp++=(-q[2]);  *dp++=(q[1]);   *dp++=(q[0]);

    for(j=0;j<3;j++)
        for(k=0;k<3;k++) {
            for(l=0,temp=0;l<4;l++)
                temp += M3[j][l] * M4[l][k];
            M1[j][k] = temp;
        }

    for(j=3;j<6;j++) {
        HH[2*i][j] = FL*(M1[1][j-3]*pc[0] - M1[0][j-3]*pc[1])/pc00;
        HH[2*i+1][j] = FL*(M1[2][j-3]*pc[0] - M1[0][j-3]*pc[2])/pc00;
    }
}

free_matrix(M1);    free_matrix(M2);    free_matrix(M3);    free_matrix(M4);
free_matrix(rRMM);  free_vector(s);
}
```

```
/*                  kvMatTools.c
                    Matrix Manipulation Tools (kalmanVision)
                    Ali J. Azarbayejani
                    December 1990
                    © Copyright 1990. All rights reserved.

    This source code is derived from "Numerical Recipes in C" by W. Press et al
    and adapted for the Macintosh by Ali Azarbayejani.  In particular, C indexing is
used
    rather than the screwy "kind-of-Pascal-maybe-Fortran" indexing used in the book.
    Also Macintosh trap routines are used rather than unix-style memory allocation.

    ! These functions will only run on a Macintosh !
*/
#include <math.h>
#include "k2.h"

#define TINY 1.0e-20

void ludcmp(double **a,int n,int *indx,double *d)
/*      Performs LU Decomposition of matrix
*/
{
    int             i,imax,j,k;
    double          big,dum,sum,temp;
    double          *vv,*vector();
    void            nrerror(),free_vector();

    vv=vector(n);
    *d=1.0;
    for(i=0;i<n;i++) { /* over rows of the matrix */
        big=0.0;
        for(j=0;j<n;j++) /* over columns or elements of row i */
            if((temp=fabs(a[i][j])) > big) big=temp;
        if(big == 0.0) nrerror("Singular matrix in routine LUDCMP");
        /* No nonzero largest element */
        vv[i]=1.0/big;
    }

    for(j=0;j<n;j++) { /* over columns of the matrix */
        for(i=0;i<j;i++) {
            sum=a[i][j];
            for(k=0;k<i;k++) sum -= a[i][k]*a[k][j];
            a[i][j]=sum;
        }
        big=0.0;
        for(i=j;i<n;i++) {
            sum=a[i][j];
            for(k=0;k<j;k++)
                sum -= a[i][k]*a[k][j];
            a[i][j]=sum;
            if((dum=vv[i]*fabs(sum)) >= big) {
                big=dum;
                imax=i;
            }
        }
        if(j != imax) {
            for(k=0;k<n;k++) {
                    dum=a[imax][k];
                    a[imax][k]=a[j][k];
                    a[j][k]=dum;
                }
                *d= -(*d);
                vv[imax]=vv[j];
            }
            indx[j]=imax;
            if(a[j][j] == 0.0) a[j][j] = TINY;
            if(j != n-1) {
                dum=1.0/(a[j][j]);
                for(i=j+1;i<n;i++) a[i][j] *= dum;
            }
        }
        free_vector(vv);
}

void lubksb(double **a,int n,int *indx,double b[])
/*      Performs Back-Substitution on LU-Decomposed matrix
*/
{
    int         i,ii=(-1),ip,j;
    double      sum;

    for(i=0;i<n;i++) {
        ip=indx[i];
        sum=b[ip];
        b[ip]=b[i];
        if(ii>=0)
            for(j=ii;j<i;j++) sum -= a[i][j]*b[j];
        else if(sum) ii=i;
        b[i]=sum;
    }
    for(i=n-1;i>=0;i--) {
        sum=b[i];
        for(j=i+1;j<n;j++) sum -= a[i][j]*b[j];
        b[i]=sum/a[i][i];
    }
}

void invmat(double **a,int n,double **b)
/*      Inverts a matrix using LU-Decomposition and Back-substitution
*/
{
    double      d,*col;
    int         i,j,*indx;

    col = vector(n);
    if(!(indx = (int *) NewPtr(n*sizeof(int))))
        nrerror("allocation of indx failure in invmat()");

    ludcmp(a,n,indx,&d);
    for(j=0;j<n;j++) {
        for(i=0;i<n;i++)
            col[i]=0.0;
        col[j]=1.0;
        lubksb(a,n,indx,col);
        for(i=0;i<n;i++) b[i][j]=col[i];
    }
    free_vector(col);
    DisposPtr((Ptr) indx);
}

double *vector(int n)
/*      Allocates a double vector of length n
*/
{
    double *v;

    v = (double *) NewPtr(n*sizeof(double));
    if(!v) nrerror("allocation failure in vector()");
    return v;
}

void free_vector(double *v)
{
    DisposPtr((Ptr) v);
}

double **matrix(int r,int c)
/*      Allocates a double matrix of size (r,c); r is the first index
*/
{
    double      *p1,**p2;
    int         i;

    if(!(p1 = (double *) NewPtr(r*c*sizeof(double))))
        nrerror("allocation of data block failure in matrix()");
    if(!(p2 = (double **) NewPtr(r*sizeof(double *))))
        nrerror("allocation of ptr block failure in matrix()");
    for(i=0;i<r;i++)
        p2[i]=p1+i*c;
    return p2;
}

void free_matrix(double **v)
{
    DisposPtr((Ptr) *v);
    DisposPtr((Ptr) v);
}

void nrerror(char error_text[])
/*      Error handler which exits gracefully
*/
{
    EventRecord             theEvent;

    printf("%s\n",error_text);
    while(!GetNextEvent(mDownMask,&theEvent));
/*  exit(0);*/
}
```

```
/*                      kvNextState.c
                        Compute Next State (kalmanVision)
                        Ali J. Azarbayejani
                        December 1990
                        © Copyright 1990. All rights reserved.
*/
#include "k2.h"
#include "k2Plus.h"

void kvNextState(int k)
/*      xa  "actual state vector"                     [input,output]

        This function computes next xa(next) = xa + f(xa,uu) Δt
*/
{
    double          *fptr,b[4];
    double          v1,v2,v3,w1,w2,w3;
    double          *ff;
    int             i;
    double          magq;

    ff = vector(NS);

    fptr = xa+6;
    v1 = *fptr++;         v2 = *fptr++;         v3 = *fptr++;
    w1 = *fptr++;         w2 = *fptr++;         w3 = *fptr;

    /*      NONLINEAR DYNAMIC EQUATIONS OF MOTION
    */
    ff[0] = v1;
    ff[1] = v2;
    ff[2] = v3;
    ff[3] = w1;
    ff[4] = w2;
    ff[5] = w3;
    ff[6] = 0.5*CFM1*fabs(v1)*v1 + M1I*uu[0];
    ff[7] = 0.5*CFM2*fabs(v2)*v2 + M2I*uu[1];
    ff[8] = 0.5*CFM3*fabs(v3)*v3 + M3I*uu[2];
    ff[9] = I23I1*w2*w3 + 0.5*CFRI1*fabs(w1)*w1 + I11I*uu[3];
    ff[10] = I31I2*w3*w1 + 0.5*CFPI2*fabs(w2)*w2 + I22I*uu[4];
    ff[11] = I12I3*w1*w2 + 0.5*CFYI3*fabs(w3)*w3 + I33I*uu[5];

    /*      ADD Low-Pass Filtered NOISE IF OPTION IS SELECTED
                       _______
        noise --->|  LPF  |--->(+)--> ff
                  |_______|     ^
                                |
                                ff
    */
    if(dNoiseFlag) {
        long        level;

        level = dNoiseFlag - dNoCmd;
        ff[6] = CFM1/level*fabs(v1)*v1 + M1I*uu[0]*level +
kvUnifNoise(VMODE,0,varD,0);
        ff[7] = CFM2/level*fabs(v2)*v2 + M2I*uu[0]*level +
kvUnifNoise(VMODE,0,varD,0);
```

```
        ff[8] = CFM3/level*fabs(v3)*v3 + M3I*uu[2]*level +
kvUnifNoise(VMODE,0,varD,0);
        ff[9] = CFRI1/level*fabs(w1)*w1 + I11I*uu[3]*level +
kvUnifNoise(AMODE,0,0,0.05);
        ff[10] = CFPI2/level*fabs(w2)*w2 + I22I*uu[4]*level +
kvUnifNoise(AMODE,0,0,0.05);
        ff[11] = CFYI3/level*fabs(w3)*w3 + I33I*uu[5]*level +
kvUnifNoise(AMODE,0,0,0.05);
    }

    for(i=0;i<NS;i++)
        xa[i] += ff[i]*DT;

    /*      COMPUTE ROTATION QUATERNION
    */
    b[0]=1;     b[1]=xa[3]/2.0; b[2]=xa[4]/2.0; b[3]=xa[5]/2.0;
    magq = sqrt(b[0]*b[0]+b[1]*b[1]+b[2]*b[2]+b[3]*b[3]);
    for(i=0;i<4;i++) b[i] /= magq;

    qMult(xa+12,b,xa+12);
    magq = sqrt(xa[12]*xa[12]+xa[13]*xa[13]+xa[14]*xa[14]+xa[15]*xa[15]);
    for(i=12;i<16;i++) xa[i] /= magq;

    /*      ZERO THE EULER ANGLES ESTIMATE
            The quaternion now holds the estimated rotational state
    */
    xa[3]=0;     xa[4]=0;     xa[5]=0;

    free_vector(ff);
}
```

```
/*                      kvPjctIm.c
                        Project image to obtain measurement (kalmanVision)
                        Ali J. Azarbayejani
                        December 1990
                        © Copyright 1990. All rights reserved.
*/
#include "k2.h"
#include "k2Plus.h"

void kvPjctIm()
/*      xa  "actual state vector"                         [input]
        y   "measurement vector"                          [output]
*/
{
    double      s[4];
    double      dt[3],pc[3];
    double      *dptr;
    int         i,j;

    qMult(xa+12,rq,s);

    for(i=0;i<NFP;i++) {        /* ALL FEATURE POINTS i=0..19 */
        /*      COMPUTE COORDINATES OF PT i IN CAM COORDS AT ACTUAL STATE...
        */
        for(j=0;j<3;j++) dt[j] = ptg[i][j] - xa[j];
        dptr = vector(4);
        qConjugate(s,dptr);
        qRotate(dt,dptr,pc);
        free_vector(dptr);
        for(j=0;j<3;j++) pc[j] -= tc[j];

        /*      APPLY "PROJECTION EQUATION" TO GET PROJECTIONS IN IMAGE PLANE */
        ya[2*i] = FL*pc[1]/pc[0];
        ya[2*i+1] = FL*pc[2]/pc[0];

        /*      ADD MEASUREMENT NOISE IF OPTION IS SELECTED.
        */
        if(mNoiseFlag) {
            ya[2*i] += kvUnifNoise(VMODE,0,varY,0);
            ya[2*i+1] += kvUnifNoise(VMODE,0,varY,0);
        }
    }
}
```

```
/*                  kvPredCov.c
                    Predict Error Covariance Matrix (kalmanVision)
                    Ali J. Azarbayejani
                    December 1990
                    © Copyright 1990. All rights reserved.
*/
#include "k2.h"

void kvPredCov()
/*      Pe  "estimated error covariance matrix at t0"   [input]
        stm "state transition matrix from t0 to t1"     [input]
        Q   "dynamic noise covariance matrix"           [input]
        Pp  "predicted error covariance at time t1"     [output]

        This function computes Pp = S Pe S' + Q, where S=stm
*/
{
    double          **m1,**m2,temp;
    int             i,j,k;

    m1 = matrix(NS,NS);     m2 = matrix(NS,NS);

    /*          First find Pe S'...
    */
    for(i=0;i<NS;i++)
        for(j=0;j<NS;j++) {
            for(k=0,temp=0;k<NS;k++)
                temp += Pe[i][k]*STM[j][k];
            m1[i][j] = temp;
        }
    /*          ...then premultiply by S: S Pe S'...
    */
    for(i=0;i<NS;i++)
        for(j=0;j<NS;j++) {
            for(k=0,temp=0;k<NS;k++)
                temp += STM[i][k]*m1[k][j];
            m2[i][j] = temp;
        }
    /*          ...and add Q.
    */
    for(i=0;i<NS;i++)
        for(j=0;j<NS;j++)
            Pp[i][j] = m2[i][j] + QQ[i][j];

    free_matrix(m1);        free_matrix(m2);
}
```

```
/*                  kvPredMeas.c
                    Predict Measurement Vector (kalmanVision)
                    Ali J. Azarbayejani
                    December 1990
                    © Copyright 1990. All rights reserved.
*/
#include "k2.h"

void kvPredMeas()
/*      xp  "predicted state vector for time t0"        [input]
        rq  "camera/body rotation quaternion"           [input]
        tc  "camera/body translation (cam coords)"      [input]
        yp  "predicted measurement vector for time t0"  [output]
*/
{
    double              pc[3],dt[3];
    double              s[4],sc[4];
    int                 i,j;

    qMult(xp+12,rq,s);

    for(i=0;i<NKF;i++) {
        for(j=0;j<3;j++) dt[j] = ptg[i][j] - xp[j];
        qConjugate(s,sc);
        qRotate(dt,sc,pc);
        for(j=0;j<3;j++) pc[j] -= tc[j];

        yp[2*i] = FL*pc[1]/pc[0];
        yp[2*i+1] = FL*pc[2]/pc[0];
    }
}
```

```
/*                      kvPredState.c
                        Predict State Vector (kalmanVision)
                        Ali J. Azarbayejani
                        December 1990
                        © Copyright 1990. All rights reserved.
*/
#include "k2.h"

void kvPredState()
/*      xe  "estimated state vector at time t0"         [input]
        uu  "command vector at time t0"                 [input]
        xp  "predicted state vector at time t1"         [output]

        This function computes xp = xe + f(xe,uu) Δt.
*/
{
    double          *fptr;
    double          v1,v2,v3,w1,w2,w3;
    double          *ff,b[4];
    int             i,j;
    double          magq;

    ff = vector(NS);

    fptr = xe + 6;
    v1 = *fptr++;           v2 = *fptr++;           v3 = *fptr++;
    w1 = *fptr++;           w2 = *fptr++;           w3 = *fptr;

    /*      USE NONLINEAR PROCESS FOR UPDATE
    */
    ff[0] = v1;
    ff[1] = v2;
    ff[2] = v3;
    ff[3] = w1;
    ff[4] = w2;
    ff[5] = w3;
    ff[6] = 0.5*CFM1*fabs(v1)*v1 + M1I*uu[0];
    ff[7] = 0.5*CFM2*fabs(v2)*v2 + M2I*uu[1];
    ff[8] = 0.5*CFM3*fabs(v3)*v3 + M3I*uu[2];
    ff[9] = I23I1*w2*w3 + 0.5*CFRI1*fabs(w1)*w1 + I11I*uu[3];
    ff[10] = I31I2*w3*w1 + 0.5*CFPI2*fabs(w2)*w2 + I22I*uu[4];
    ff[11] = I12I3*w1*w2 + 0.5*CFYI3*fabs(w3)*w3 + I33I*uu[5];

    for(i=0;i<NS;i++)
        xp[i] = ff[i]*DT + xe[i];

    /*      COMPUTE ROTATION QUATERNION
    */
    b[0]=1;     b[1]=xp[3]/2.0; b[2]=xp[4]/2.0; b[3]=xp[5]/2.0;
    magq = sqrt(b[0]*b[0]+b[1]*b[1]+b[2]*b[2]+b[3]*b[3]);
    for(i=0;i<4;i++) b[i] /= magq;
    qMult(xe+12,b,xp+12);
    magq = sqrt(xp[12]*xp[12]+xp[13]*xp[13]+xp[14]*xp[14]+xp[15]*xp[15]);
    for(i=12;i<16;i++) xp[i] /= magq;
    xp[3]=0;    xp[4]=0;    xp[5]=0;

    free_vector(ff);
}
```

```
/*                  k2RandomTools.c
                    Random Number Tools (kalmanVision)
                    Ali J. Azarbayejani
                    January 1991
                    © Copyright 1991. All rights reserved.

    ! These functions will only run on a Macintosh !
*/
#include    <math.h>

double kvUnifNoise(char mode,double mean,double var,double spread)
/*
                                            ^
                                            |
                                  __________|__________  1/2A
                                 |          |          |
                                 |          |          |
                            _____|__________|__________|_____
                                  -A                   A

                                    VARIANCE = 3 A^2

*/
{
    register double     A,errFlt;
    long                errInt;

    if(mode == 1)           /* "mode" determines whether "var" or "spread" valid */
        A = sqrt(3*var);
    else if (mode == 2)
        A = spread;
    else A = 1;                 /* Default unit variance
    */
    errInt = Random();          /* Macintosh Toolbox pseudorandomnumbergenerator
    */
    errFlt = errInt;            /* Automatic type conversion a la C
    */
    errFlt *= A/32767.0;        /* Normalization
    */
    return (mean + errFlt);
}
```

```
/*                      kvSTM.c
                        Compute State Transition Matrix (kalmanVision)
                        Ali J. Azarbayejani
                        December 1990
                        © Copyright 1990. All rights reserved.
*/
#include "k2.h"

void kvSTM()
/*      xe  "estimated state vector at time t0"         [input]
        stm "state transition matrix from t0 to t1"     [output]

        This function computes S = (I + F Δt)
*/
{
    int         i,j;
    double      q0,q1,q2,q3;
    double      v1,v2,v3;
    double      w1,w2,w3;
    double      *fptr;

    fptr = xe+6;
    v1 = *fptr++;       v2 = *fptr++;       v3 = *fptr++;
    w1 = *fptr++;       w2 = *fptr++;       w3 = *fptr++;

    for(i=0;i<NS;i++)
        for(j=0;j<NS;j++)
            STM[i][j] = 0.0;

    STM[0][6] = 1;      STM[1][7] = 1;      STM[2][8] = 1;
    STM[3][9] = 1;      STM[4][10] = 1;     STM[5][11] = 1;

    STM[6][6] = CFM1*fabs(v1);
    STM[7][7] = CFM2*fabs(v2);
    STM[8][9] = CFM3*fabs(v3);

    STM[9][9]=CFRI1*fabs(w1);   STM[9][10]=I23I1*w3;        STM[9][11]=I23I1*w2;
    STM[10][9]=I31I2*w3;        STM[10][10]=CFPI2*fabs(w2); STM[10][11]=I31I2*w1;
    STM[11][9]=I12I3*w2;        STM[11][10]=I12I3*w1;
    STM[11][11]=CFYI3*fabs(w3);

    for(i=0;i<NS;i++)
        for(j=0;j<NS;j++)
            STM[i][j] *= DT;

    for(i=0;i<NS;i++)
        STM[i][i] += 1.0;
}
```

```
/*                    k3QuatTools.c
                      Quaternion Tools (kalmanVision)
                      Ali J. Azarbayejani
                      January 1991
                      © Copyright 1991. All rights reserved.
*/
#include "k2.h"

void qLeft(double *q,double **qLMM)
/*      COMPUTES LEFT QUATERNION MULTIPLICATION MATRIX
        i.e. qr = Qr, Q is the qLMM, matrix operator if q premultiplies
*/
{
    double      *dptr;

    dptr = qLMM[0];
    *dptr++ = q[0]; *dptr++ = (-q[1]);  *dptr++ = (-q[2]);  *dptr++ = (-q[3]);
    *dptr++ = q[1]; *dptr++ = q[0];     *dptr++ = (-q[3]);  *dptr++ = q[2];
    *dptr++ = q[2]; *dptr++ = q[3];     *dptr++ = q[0];     *dptr++ = (-q[1]);
    *dptr++ = q[3]; *dptr++ = (-q[2]);  *dptr++ = q[1];     *dptr++ = q[0];
}

void qRight(double *q,double **qRMM)
/*      COMPUTES RIGHT QUATERNION MULTIPLICATION MATRIX
        i.e. rq = Qr, Q is the qRMM, matrix operator if q postmultiplies
*/
{
    double      *dptr;

    dptr = qRMM[0];
    *dptr++ = q[0]; *dptr++ = (-q[1]);  *dptr++ = (-q[2]);  *dptr++ = (-q[3]);
    *dptr++ = q[1]; *dptr++ = q[0];     *dptr++ = q[3];     *dptr++ = (-q[2]);
    *dptr++ = q[2]; *dptr++ = (-q[3]);  *dptr++ = q[0];     *dptr++ = q[1];
    *dptr++ = q[3]; *dptr++ = q[2];     *dptr++ = (-q[1]);  *dptr++ = q[0];
}

void qMult(double *q,double *r,double *s)
/*  s=qr    (quaternion multiplication)
    q,r,s are 4-element arrays (quaternions)
    CAN do in-place mult q = qr; CANNOT do in-place mult r=qr
*/
{
    double                  **qLMM;
    register double         temp;
    int                     i,j;

    qLMM = matrix(4,4);

    qLeft(q,qLMM);
    for(i=0;i<4;i++) {
        for(j=0,temp=0;j<4;j++)
            temp += qLMM[i][j]*r[j];
        s[i] = temp;
    }
    free_matrix(qLMM);
}
```

```
void qRotMat(double *q,double **qRM)
/*      COMPUTES ROTATION MATRIX FROM QUATERNION
        qxq* = Mx, where q,x=quaternions, M=(4,4) matrix
        M is qRM. Lower Right (3,3) of M is ortho rot matrix
*/
{
    double              **RMM,**LMM;
    int                 i,j,k;
    register double     temp;

    RMM = matrix(4,4);      LMM = matrix(4,4);
    qRight(q,RMM);          qLeft(q,LMM);

    for(i=0;i<4;i++)
        for(j=0;j<4;j++) {
            for(k=0,temp=0;k<4;k++)
                temp += RMM[k][i] * LMM[k][j];
            qRM[i][j]=temp;
        }
    free_matrix(RMM);       free_matrix(LMM);
}

void qCRotMat(double *q,double **qCRM)
/*      COMPUTES CONJUGATE ROTATION MATRIX FROM QUATERNION
        q*xq = Mx, where q,x=quaternions, M=(4,4) matrix
        M is qCRM
*/
{
    double              **RMM,**LMM;
    int                 i,j,k;
    register double     temp;

    RMM = matrix(4,4);      LMM = matrix(4,4);
    qRight(q,RMM);          qLeft(q,LMM);

    for(i=0;i<4;i++)
        for(j=0;j<4;j++) {
            for(k=0,temp=0;k<4;k++)
                temp += RMM[i][k] * LMM[j][k];
            qCRM[i][j]=temp;
        }
    free_matrix(RMM);       free_matrix(LMM);
}
```

```
void qRotate(double *xv,double *q,double *yv)
/*      ROTATES xv INTO yv USING QUATERNION q
*/
{
    double              **RM;
    int                 i,j;
    register double     temp;

    RM = matrix(4,4);
    qRotMat(q,RM);
    for(i=0;i<3;i++) {
        for(j=0,temp=0;j<3;j++)
            temp += RM[i+1][j+1] * xv[j];
        yv[i] = temp;
    }
    free_matrix(RM);
}

void qConjugate(double *q,double *qc)
/*      CONJUGATES q AND RETURNS IN qc
*/
{
    qc[0] = q[0];
    qc[1] = (-q[1]);
    qc[2] = (-q[2]);
    qc[3] = (-q[3]);
}

void qNormalize(double *q)
/*      NORMALIZES q IN PLACE
*/
{
    double      magq;
    int         i;

    magq = sqrt(q[0]*q[0] + q[1]*q[1] + q[2]*q[2] + q[3]*q[3]);
    for(i=0;i<4;i++) q[i] /= magq;
}
```

```
/*  CAMMOD.C
    Ali J. Azarbayejani
    © Copyright 1991, All Rights Reserved.

    This is the user interface running under the QNX operating system
    on the GATEWAY2000 i386 PC.
*/

#include <stdio.h>
#include <math.h>
#include <dev.h>

#define     topmode     0
#define     datamode    1
#define     filemode    2
#define     lsqmode     3
#define     ilsqmode    4

char        mtext0[80] = {"Data  ","Files ","LSQ   ","ILSQ  ","Quit",0};
char        mtext1[80] = {"n     ","h     ","q     ","r     ","Main",0};
char        mtext2[80] = {"Write ","Read  ","List  ","Main  ",0};
char        mtext3[80] = {"New   ","Main  ",0};
char        mtext4[80] = {"New   ","Main  ",0};
char        mtext10[80] = {"1 ","2 ","3 ","4 ","5 ","6 ","7 ","8 ","9 ","10",0};

char        *menu[8];

void setup()
{
    term_clear(0);
    menu[topmode] = mtext0;
    menu[datamode] = mtext1;
    menu[filemode] = mtext2;
    menu[lsqmode] = mtext3;
    menu[ilsqmode] = mtext4;
    set_option(stdin,get_option(stdin) & ~3);
}

char getchoice(mode)
int     mode;
{
    char    *cp;

    term_box_fill(1,0,80,1,0x8100,0,0xdb);
    cp = term_menu(1,0,menu[mode],menu[mode],0x9700,0,0x0009);
    if(cp)
        return(*cp);
    else
        return(0);
}

void showdata(n,h,q,r)
unsigned        n;
double          h,*q,*r;
{
    int     i;

    term_printf(4,0,0x8400,"n: %10d mmts",n);
    term_printf(5,0,0x8400,"h: %10f mm",h);
    for(i=0;i<10 && i<n;i++)
        term_printf(4+i,25,0x8400,"q[%2d]= %10f mm,      r[%2d]= %10f
mm",i+1,q[i],i+1,r[i]);
}

void modeData(np,hp,q,r)
unsigned              *np;
double                *hp,*q,*r;
{
    char         c,*cp,done=0,in[20];
    int          i;

    while(!done) {
        term_clear(0);
        showdata(*np,*hp,q,r);
        c = getchoice(datamode);
        switch(c) {
            case 'n':
                term_input(4,0,in,15,"n: ",0x8300,0x20,0x8700);
                *np = atoi(in);
                break;
            case 'h':
                term_input(5,0,in,15,"h: ",0x8300,0x20,0x8700);
                *hp = atof(in);
                break;
            case 'q':
                cp = term_lmenu(4,23,mtext10,mtext10,0x9700,0,0x0009);

                i = atoi(cp);
                term_input(3+i,25,in,17,"     q=   ",0x8300,0x20,0x8700);
                q[i-1] = atof(in);
                break;
            case 'r':
                cp = term_lmenu(4,49,mtext10,mtext10,0x9700,0,0x0009);

                i = atoi(cp);
                term_input(3+i,51,in,17,"     r=   ",0x8300,0x20,0x8700);
                r[i-1] = atof(in);
                break;
            case 'M': case 'm':
                done=1;
                break;
        }
    }
}

void modeFiler(np,hp,q,r)
unsigned        *np;
double          *hp,*q,*r;
{
    char        c,*cp,done=0,in[20],d;
    int         i,j,test;
    unsigned    len;
    FILE        *fp;

    len = *np;
    if(len > 10) len=10;
    while(!done) {
        term_clear(0);
        showdata(*np,*hp,q,r);
        c = getchoice(filemode);
        switch(c) {
            case 'W': case 'w':
                term_input(15,0,in,15,"Filename: ",0x8300,0x20,0x8700);
                for(i=0,d= *in;i<8 && d!='\0' && d!='.';i++)
                    d= *(in+i+1);
                strcpy(in+i,".adat");
                if((fp=fopen(in,"w")) == 0)
                    exit(0);
                fwrite(np,sizeof(unsigned),1,fp);
                fwrite(hp,sizeof(double),1,fp);
                fwrite(q,sizeof(double),len,fp);
                fwrite(r,sizeof(double),len,fp);
                fclose(fp);
                break;
            case 'R': case 'r':
                term_input(15,0,in,15,"Filename: ",0x8300,0x20,0x8700);
                for(i=0,d= *in;i<8 && d!='\0' && d!='.';i++)
                    d= *(in+i+1);
                strcpy(in+i,".adat");
                if((fp=fopen(in,"r")) == 0) {
                    term_printf(2,0,0x8500,"Cannot open file.");
                    break;
                }
                fread(np,sizeof(unsigned),1,fp);
                fread(hp,sizeof(double),1,fp);
                fread(q,sizeof(double),*np,fp);
                fread(r,sizeof(double),*np,fp);
                fclose(fp);
                break;
            case 'L': case 'l':
                break;
            case 'M': case 'm':
                done=1;
                break;
        }
    }
}

void modeLSQ(np,hp,q,r,x)
unsigned        *np;
double          *hp,*q,*r,*x;
{
    char        c,*cp,done=0,in[20];
    int         i;
    double      dv,e,do2lsq();

    term_clear(0);
    while(!done) {
        showdata(*np,*hp,q,r);
        c = getchoice(lsqmode);
        switch(c) {
            case 'n': case 'N':
                term_box_fill(7,0,19,5,0x8000,0,0xdb);
                term_input(7,0,in,15,"dv: ",0x8300,0x20,0x8700);
                dv = atof(in);
                term_printf(7,0,0x8400,"dv: %10f mm",dv);
                e = do2lsq(*np,*hp,q,r,dv,x);
                term_printf(9,0,0x8700,"f: %10f mm",x[0]);
                term_printf(10,0,0x8700,"dw: %10f mm",x[1]);
                term_printf(11,0,0x8700,"e: %,10f mm^4",e);
                break;
            case 'M': case 'm':
                done=1;
                term_clear(0);
                break;
        }
    }
}

double do2lsq(n,h,q,r,dv,x)
unsigned        n;
double          h,*q,*r,dv,*x;
{
    double      temp,c=h-dv,d,a1,a2,a3,a4,y1,y2,det,e;
    int         i;

    a1 = n*c*c;
    for(i=0,a2=0;i<n;i++)
        a2 += q[i];
    a2 *= c;
    a3 = a2;
    for(i=0,a4=0;i<n;i++)
        a4 += q[i]*q[i];
    for(i=0,y1=0;i<n;i++)
        y1 += q[i]*r[i];
    y1 *= c;
    for(i=0,y2=0;i<n;i++)
        y2 += q[i]*q[i]*r[i];
    x[0] = a4*y1 - a2*y2;
    x[1] = -a3*y1 + a1*y2;
    det = a1*a4 - a2*a3;
    if(det == 0) {
        term_printf(0,0,0x8500,"Zero Determinant");
        return(0);
    }
    x[0] /= det; x[1] /= det;

    d = x[0]*c;
    for(i=0,e=0;i<n;i++) {
        temp = d + q[i]*x[1] - q[i]*r[i];
        e += temp*temp;
    }
    return(e);
}

void modeILSQ()
{
    while(getchoice(ilsqmode) != 'M');
}
```

```
main()
{
    int         i;
    char        theChar,doneFlag;
    double      r[10],q[10],h,du,dv,dw,f;
    unsigned    n;
    double      x[2];

    setup();
    for(doneFlag=0;!doneFlag;) {
        theChar=getchoice(topmode);
        switch(theChar) {
            case 'd': case 'D':
                modeData(&n,&h,q,r);
                break;
            case 'f': case 'F':
                modeFiler(&n,&h,q,r);
                break;
            case 'l': case 'L':
                modeLSQ(&n,&h,q,r,x);
                break;
            case 'i': case 'I':
                modeILSQ();
                break;
            case 'q': case 'Q':
                doneFlag=1;
                break;
        }
    }
    term_clear(0);
}
```

```
/*  FUNC.C
    Ali J. Azarbayejani
    © Copyright 1991, All Rights Reserved.

    Prototype of the image processor.
*/

#include <stdio.h>
#include <math.h>
#include <dev.h>
#include "ofglib.c"

#define     topmode 0
#define     datamode    1
#define     filemode    2
#define     extmode 3
#define     data2mode   4
#define     HOZO    0
#define     VERT        1
#define     QTY     20

char        mtext0[80] = {"Data  ","Files ","Ext   ","Quit",0};
char        mtext4[80] = {"X           ","Y          ",0};
char        mtext3[80] = {"<Return>    ",0};
char        mtext2[80] = {"Write ","Read  ","Main  ",0};
char        mtext10[80] = {"1 ","2 ","3 ","4 ","5 ","6 ","7 ","8 ","9 ","10",
                 "11","12","13","14","15","16","17","18","19","20","Exit",0};
 char       *menu[8];

void  setup()
{
    term_clear(0);
    menu[topmode] = mtext0;
    menu[datamode] = mtext10;
    menu[data2mode] = mtext4;
    menu[filemode] = mtext2;
    menu[extmode] = mtext3;
    set_option(stdin,get_option(stdin) & ~3);

    initSETUP();
    initFRAME();
    initILUT();
    initOLUT();
}

char getchoice(mode)
int     mode;
{
    char    *cp;

    term_box_fill(1,0,80,1,0x8100,0,0xdb);
    cp = term_menu(1,0,menu[mode],menu[mode],0x9700,0,0x0009);
    if(cp)
        return(*cp);
    else
        return(0);
}
```

```
void showdata(qp,q)
double          qp[2][20],q[2][20];
{
    int     i;

    term_cur(3,0); term_clear(2);
    for(i=0;i<20;i++)
        term_printf(3+i,5,0x8400,"qp[%2d]= (%10f,%10f)",i+1,qp[0][i],qp[1][i]);
    for(i=0;i<20;i++)
        term_printf(3+i,45,0x8500,"q[%2d]= (%10f,%10f)",i+1,q[0][i],q[1][i]);
}

void modeData(qp,q)
double              qp[2][20],q[2][20];
{
    char        c,*cp,done=0,in[20];
    int         i,j;

    i= 0;
    while(!done) {
        term_cur(3,0); term_clear(2);
        showdata(qp,q);
        cp = term_lmenu(3,3,mtext10,mtext10+3*i,0x9700,0,0x0009);

        term_box_fill(3,0,5,20,0x8000,0,0xdb);
        term_printf(0,0,0x8700,"*cp = %x",*cp);
        if(*cp == 'E')
            done = 1;
        else {
            i = atoi(cp);
            cp = term_menu(2,14,mtext4,mtext4+12*j,0x9700,0,0x0009);
            term_box_fill(2,0,80,1,0x8000,0,0xdb);
            switch(*cp) {
                case 'X': case 'x':
                    term_input(2+i,14,in,9,">",0x8300,0x20,0x8700);
                    qp[0][i-1] = atof(in);
                    j=1;
                    i-=1;
                    break;
                case 'Y': case 'y':
                    term_input(2+i,25,in,9,">",0x8300,0x20,0x8700);
                    qp[1][i-1] = atof(in);
                    j=0;
                    break;
            }
        }
    }
}
```

```
void modeFiler(qp,q)
double          qp[2][20],q[2][20];
{
    char        c,*cp,done=0,in[20],d;
    int         i,j,test;
    unsigned    len;
    FILE        *fp;

    while(!done) {
        term_clear(0);
        showdata(qp,q);
        c = getchoice(filemode);
        switch(c) {
            case 'W': case 'w':
                term_input(15,0,in,15,"Filename: ",0x8300,0x20,0x8700);
                for(i=0,d= *in;i<8 && d!='\0' && d!='.';i++)
                    d= *(in+i+1);
                strcpy(in+i,".qdat");
                if((fp=fopen(in,"w")) == 0)
                    exit(0);
                fwrite(*qp,sizeof(double),40,fp);
                fwrite(*q,sizeof(double),40,fp);
                fclose(fp);
                break;
            case 'R': case 'r':
                term_input(15,0,in,15,"Filename: ",0x8300,0x20,0x8700);
                for(i=0,d= *in;i<8 && d!='\0' && d!='.';i++)
                    d= *(in+i+1);
                strcpy(in+i,".qdat");
                if((fp=fopen(in,"r")) == 0) {
                    term_printf(2,0,0x8500,"Cannot open file.");
                    break;
                }
                fread(*qp,sizeof(double),40,fp);
                fread(*q,sizeof(double),40,fp);
                fclose(fp);
                break;
            case 'L': case 'l':
                break;
            case 'M': case 'm':
                done=1;
                break;
        }
    }
}
```

```
void modeExt(qp,q)
double      qp[2][20],q[2][20];
{
    char            dir[20],va[20];
    int             du[20],dv[20];
    double          l[20][3],x[100],y[100];

    int             i,j,k,n,bx,by,area;
    double          x,y,pe[2],ratio,dx,dy,drx,dry;
    unsigned        u,v,u1,v1,po[2];
    long            r0x,r0y,r1x,r1y;
    /*                  ===== LINE DIRECTIONS and LENGTHS ===== */
    for(i=0;i<QTY;i++) {
        du[i] = (dx = qp[0][(i+1)&3 + i&0xfffc] - qp[0][i]);
        dv[i] = (dy = qp[1][(i+1)&3 + i&0xfffc] - qp[1][i]);
        if(dx < 0) dx=(-dx);
        if(dy < 0) dy=(-dy);
        if(dx > dy) dir[i] = HOZO;
        else dir[i] = VERT;
    }
    /*                  ===== CORNER VALIDITY =====             */
    for(i=0;i<QTY;i++)
        va[i] = 0;
    for(i=0;i<QTY;i++) {
        int         u,v;
        u=q[0][i]; v=q[1][i];
        if((u>50 && u<590) && (v>50 && v<430)) {
            va[i] = 1;
            va[(i-1)&3 + i&0xfffc] = 1;
        }
    }
    for(i=0;i<QTY;i++) {

    }
    /*              ===== DRAW LINES on OFG =====       */
    for(i=0;i<4;i++) {
        u=qp[0][i]; v=qp[1][i];
        u1=qp[0][(i+1)&3]; v1=qp[1][(i+1)&3];
        olLine(u,v,u1,v1,0x0100);
    }
    getchoice(extmode);
    for(i=0;i<4;i++) {
        r0x = qp[0][i]; r0y = qp[1][i];
        r1x = qp[0][(i+1)&3]; r1y = qp[1][(i+1)&3];
        dx = r1x - r0x;
        dy = r1y - r0y;
        drx = dx; if(dy != 0) drx /= dy;
        dry = dy; if(dx != 0) dry /= dx;
        if(dx<0) dx=(-dx);
        if(dy<0) dy=(-dy);
        if(dy > dx) {n=dy; bx=1; by=0; dry=1; if(r1y<r0y) r0y=r1y;}
        else {n=dx; bx=0; by=1; drx=1; if(r1x<r0x) r0x=r1x;}
        for(j=15;j<n-15;j++) {
            po[0] = r0x+j*drx-15*bx; po[1] = r0y+j*dry-15*by;
            edgeFind(po,pe,bx,by,30);
            po[0] = pe[0]; po[1] = pe[1];
            olBox(po[1],po[0],1,0x0800);
        }
    }
    getchoice(extmode);
    initFRAME();
}

void edgeFind(po,pe,du,dv,n)
double      pe[2];
int         po[2],du,dv,n;
{
    unsigned        *data,x,y;
    int             *t;
    int             i,j,dlen,mindex;
    long            max,grad;

    /** Check boundaries etc... **/
    /** Allocate buffer for image data... **/
    dlen=n+4;
    if((data = (unsigned *) malloc(dlen*sizeof(unsigned))) == 0)
        exit(0);
    /** Download image data from ofg... **/
    for(i=0,x=po[0],y=po[1];i<dlen;i++,x+=du,y+=dv) {
        outWord(0x1a,y);        /* YPTR      */
        outWord(0x18,x);        /* XPTR      */
        data[i] = inWord(0x1c); /* Data Register */
    }
    for(i=0;i<dlen;i++)
        data[i] = data[i]& 0xff;
    term_cur(3,0); term_clear(2);
    /*for(i=0;i<dlen;i++)
        term_printf(i+3,0,0x8600,"%d: %6x",i+po[0],data[i]);*/
    /** Scan with gradient finder, update max... **/
    for(i=0,max=0,mindex=0;i<n;i++) {
        t= ((int *)data+i);
        grad = t[0] - 8*t[1] + 8*t[3] - t[4];
        term_printf(i+3,20,0x8300,"%ld",grad);
        if(grad<0) grad = -grad;
        if(grad>max) {
            max=grad;
            mindex=i;
        }
    }
    /** Return coordinates of optimum gradient... **/
    cfree(data);
    pe[0] = po[0] + mindex*du;
    pe[1] = po[1] + mindex*dv;
}
```

```
void lineFind(x,y,n,dir,line)
double      *x,*y,*line;
int         n,dir;
{
    double      *u,*v,k11,k12,k21,k22,temp,det,a1,a2,a3,atb1,atb2;
    int         i,j;

    u=x; v=y;
    if(dir == HOZO) {u=y; v=x;}
    for(i=0,k11=0;i<n;i++) {
        temp = *(u+i);
        temp *= temp;
        k11 += temp;
    }
    for(i=0,k12;i<n;i++)
        k12 += *(u+i);
    k21 = k12;
    k22 = n;
    det = k11*k22 - k12*k21;
    if(det == 0) term_printf(0,0,0x8400,"Zero determinant");
    for(i=0,atb1=0;i<n;i++)
        atb1 += u[i] * v[i];
    for(i=0,atb2=0;i<n;i++)
        atb2 += v[i];
    a1 = (k11*atb1 + k12*atb2) / det;
    a3 = (k21*atb1 + k22*atb2) / det;
    a2 = I / sqrt(1 + a1*a1);
    line[0]=a1*a2; line[1]=a2; line[2]=a3*a2;
    if(dir == VERT) {
        line[1]=line[0]; line[0]=a2;
    }
}

main()
{
    int         i,j;
    char        theChar,doneFlag;
    double      qp[2][20],q[2][20];
    double      x[2];

    setup();
    for(i=0;i<2;i++)
        for(j=0;j<20;j++)
            qp[i][j] = 100*i+j;
    for(doneFlag=0;!doneFlag;) {
        term_clear(0);
        showdata(qp,q);
        theChar=getchoice(topmode);
        switch(theChar) {
            case 'd': case 'D':
                modeData(qp,q);
                break;
            case 'f': case 'F':
                modeFiler(qp,q);
                break;
            case 'e': case 'E':
                modeExt(qp,q);
```

```
                    break;
                case 'q': case 'Q':
                    doneFlag=1;
                    break;
            }
        }
        term_clear(0);
    }
```

```
/*  OFGLIB.C
    Ali J. Azarbayejani
    © Copyright 1991, All Rights Reserved.
*/
#include "ofgev.h"

#define BASE 0x0300

/*----------------------------------------------------------------*/
/*                   I/O Functions      */
/*----------------------------------------------------------------*/

void outByte(a,d)
unsigned    a,d;
{
    oport = BASE + a;
    odata = d & 0x00ff;
    asm("mov dx,<oport>");
    asm("mov ax,<odata>");
    asm("b out [dx]");
}

void outWord(a,d)
unsigned    a,d;
{
    oport = BASE + a;
    odata = d & 0xffff;
    asm("mov dx,<oport>");
    asm("mov ax,<odata>");
    asm("w out [dx]");
}

unsigned int inByte(a)
unsigned        a;
{
    oport = BASE + a;
    asm("mov dx,<oport>");
    asm("b in [dx]");
    asm("mov <odata>,ax");
    return(odata & 0xff);
}

unsigned int inWord(a)
unsigned        a;
{
    oport = BASE + a;
    asm("mov dx,<oport>");
    asm("w in [dx]");
    asm("mov <odata>,ax");
    return(odata & 0xffff);
}
```

```
/*----------------------------------------------------------------------*/
/*          Initialization Functions          */
/*----------------------------------------------------------------------*/

void initILUT()
{
    int         i;

    /*  y=x     */

    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Initializing ILUT...");
    outByte(0xc,0);
    for(i=0;i<256;i++)
        outByte(0xd,i);
}

void initOLUT()
{
    int         b,c,i;

    /*  0000        y=x
        0001        Full G, zero RB
        001X        Full B, zero RG
        01XX        Full RGB
        1XXX        Full R, zero GB
    */

    /*                    BANK 0: y=x             */
    for(c=0;c<3;c++) {
        outByte(1,c);
        for(i=0;i<256;i++) {
            outByte(2,i);
            outByte(8,i);
        }
    }

    for(c=0;c<3;c++)
        for(b=1;b<16;b++) {
            outByte(1,(b<<4)|c);
            for(i=0;i<256;i++) {
                outByte(2,i);
                outByte(8,0);
            }
        }

    /*                    BANK 1: FULL SCALE GREEN    */
    outByte(1,0x11);
    for(i=0;i<256;i++) {
        outByte(2,i);
        outByte(8,0xff);
    }
    /*                      BANKS 2..3: FULL SCALE BLUE */
    for(b=2;b<4;b++) {
        outByte(1,(b<<4)|2);
        for(i=0;i<256;i++) {
            outByte(2,i);
            outByte(8,0xff);
        }
    }

    /*                      BANKS 4..7: 0xd0 SCALE WHITE    */
    for(c=0;c<3;c++)
        for(b=4;b<8;b++) {
            outByte(1,(b<<4)|c);
            for(i=0;i<256;i++) {
                outByte(2,i);
                outByte(8,0xd0);
            }
        }

    /*                      BANKS 8..15: FULL SCALE RED */
    for(b=8;b<16;b++) {
        outByte(1,b<<4);
        for(i=0;i<256;i++) {
            outByte(2,i);
            outByte(8,0xff);
        }
    }
}

void initFRAME()
{
    int         i;
    unsigned    inByte();

    /*                  Clear Overlay Memory                    */

    outWord(0x10,0x4040);             /* PBCON: PB enable, Z-Mode */
    for(i=0;i<8;i++)
        outWord(0x16,0x0000);         /* PBUF: fill with 0000     */
    outWord(0x10,0x0000);             /* PBCON: PB disable, Z-mode    */
    outByte(6,0x0);               /* Pan Register         */
    outByte(7,0x0);               /* Scroll Register          */
    outByte(0x04,0x40);               /* ACQ: clear mode          */
    while((inByte(4)>>6) != 0x00);        /* wait until clear done        */
    outByte(0x04,0x40);               /* ACQ: clear mode          */
    while((inByte(4)>>6) != 0x00);        /* wait until clear done        */
    outWord(0x10,0x4040);             /* PBCON: PB enable, Z-mode */
    outWord(0x05,0x03);           /* PTRCON: auto step = 8        */
    outByte(4,0xc0);                  /* ACQ mode - GRAB      */
}
```

```
void initSETUP()
{
    outByte(0,0x04);            /* Control Register     */
    outByte(0xa,0x02);          /* Video Bus Control        */
    outByte(0xc,0x0);           /* ILUTA => ADC Ctrl Reg    */
    outByte(0xe,0x0);           /* ADC Ctrl Reg     */
    outByte(0xc,0x1);           /* ILUTA => NREF Reg    */
    outByte(0xe,0x20);          /* Negative Reference Reg   */
    outByte(0xc,0x2);           /* ILUTA => PREF Reg    */
    outByte(0xe,0x90);          /* Positive Reference Reg   */
    outWord(0x12,0x0);          /* Host Mask Register   */
    outWord(0x14,0xff00);       /* Video Mask Register  */
}

/*------------------------------------------------------------------------*/
/*          Overlay Routines              */
/*------------------------------------------------------------------------*/
void olBox(row,col,size,val)
unsigned    row,col,size,val;
{
    int             i,x,y;

    outWord(0x10,0x4040);           /** PBCON: enable pb, zmode **/
    outByte(0x05,0x07);             /** PTRCTL: step 1 **/
    outWord(0x12,0xffff);           /** HMASK: protect all  **/
    outWord(0x1c,0);                /** Data Port: clear PB **/
    outWord(0x12,0x00ff);           /** HMASK: protect image **/
    for(y=0;y<size;y++) {
        outWord(0x1a,y+row);        /** YPTR **/
        outWord(0x18,col);          /** XPTR **/
        for(x=0;x<size;x++)
            outWord(0x1c,val);      /** Data Port **/
    }
    outWord(0x12,0x0000);           /** HMASK: unprotect image **/
    outWord(0x10,0x4040);           /** PBCON: enable pb, zmode **/
    outByte(0x05,3);                /** PTRCTL: step 8 **/

}
void olHseg(u,v,size,val)
unsigned        u,v,size,val;
{
    int     x;

    outWord(0x10,0x4040);           /** PBCON: enable pb, zmode **/
    outByte(0x05,0x07);             /** PTRCTL: step 1 **/
    outWord(0x12,0x00ff);           /** HMASK: protect image **/
    outWord(0x1a,v);                /** YPTR **/
    outWord(0x18,u);                /** XPTR **/
    for(x=0;x<size;x++)
        outWord(0x1c,val);          /** Data Port **/
    outWord(0x12,0x0000);           /** HMASK: unprotect image **/
    outWord(0x10,0x4040);           /** PBCON: enable pb, zmode **/
    outByte(0x05,3);                /** PTRCTL: step 8 **/
}
```

```
void olVseg(u,v,size,val)
unsigned        u,v,size,val;
{
    int     y;

    outWord(0x10,0x4040);           /* PBCON: enable pb, zmode */
    outByte(0x05,0x07);             /* PTRCTL: step 1       */
    outWord(0x12,0x00ff);           /* HMASK: protect image     */
    for(y=0;y<size;y++) {
        outWord(0x1a,y+v);          /* YPTR             */
        outWord(0x18,u);            /* XPTR             */
        outWord(0x1c,val);          /* Data Port            */
    }
    outWord(0x12,0x0000);           /* HMASK: unprotect image   */
    outWord(0x10,0x4040);           /* PBCON: enable pb, zmode */
    outByte(0x05,3);                /* PTRCTL: step 8       */
}

void olLine(u0,v0,un,vn,val)
unsigned    u0,v0,un,vn,val;
{
    int             n,k,uo,uf,dy;
    unsigned            temp;
    double          x0,y0,xn,yn,m,m2,b;

    /* p0 always left of pn */
    if(u0 > un)
        {temp=un;un=u0;u0=temp;temp=vn;vn=v0;v0=temp;}
    dy=1;
    if(vn != v0) {
        xn=un; x0=u0; yn=vn; y0=v0;
        if(yn < y0) dy = -1;
        n = vn-v0;
        if(n<0) n= -n;
        m = (xn-x0)/(yn-y0);
        if(m<0) m= -m;
        m2=m/2;
        uo=u0; uf=(x0+m2);
        olHseg(uo,v0,uf-uo+1,val);
        for(k=1;k<n;k++) {
            uo=uf; uf=(x0+m2+k*m);
            olHseg(uo,v0+k*dy,uf-uo+1,val);
        }
        uo=uf;
        olHseg(uo,vn,un-uo+1,val);
    }
    else {
        olHseg(u0,v0,un-u0+1,val);
    }
}
```

```
/***************************************************************************/
/* snapper.c
    for snapping images with the Imaging Technology OFG
    Ali J. Azarbayejani
    © Copyright 1991, All Rights Reserved */
/***************************************************************************/
#include <stdio.h>
#define BASE 0x0300
#define GRAB 0
#define SNAP 1
#define QUIT 2

unsigned int            oport, odata;

/***** Screen format Functions *****/

void scrSETUP()
{
    term_clear(0);
    term_box_fill(2,0,80,4,0x8140,0,0xdb);
    term_printf(3,34,0x9540,"OFG Snapper");
    term_printf(4,29,0x9540,"Ali Azar, August 1990");
}

void scrGRAB()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"(S)nap  (E)dit ILUT  (M)arker  (Q)uit");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Grabbing...Enter S to snap:");
}

void scrSNAP()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"(S)ave  (G)rab  (E)dit OLUTs  (O)verlay  (L)oad  (H)ist
(Q)uit ");
}

void scrILUT()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"ILUT editor  (E)scape");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Choose an ILUT to load:");
}

void scrMARK()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"Marker: (H)ozo  (V)ert  (E)scape");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Choose horizontal or vertical marker:");
}

void scrHOZO()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"Hozo Marker: (N)ew (E)scape");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Choose row:");
}

void scrVERT()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"Vertical Marker: (N)ew (E)scape");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Choose column:");
}

void scrOLUT()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"OLUT editor  (E)scape");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Choose a color:");
}

void scrSAVE()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"Save Editor");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Name the image; to abort, enter <CANCEL> (minus)");
}

void scrOVERLAY()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"(T)ext  (G)raphics  (E)scape");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Add overlays:");
}

void scrLOAD()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"Load Editor");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Choose an image:");
}

void scrHIST()
{
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"Histogram. (E)scape");
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Histogram.");
}

void scrQUIT()
{
    term_clear(0);
}
```

```
/***** I/O Functions *****/

void outByte(a,d)
{
    oport = BASE + a;
    odata = d & 0x00ff;
    asm("mov dx,<oport>");
    asm("mov ax,<odata>");
    asm("b out [dx]");
}

void outWord(a,d)
{
    oport = BASE + a;
    odata = d & 0xffff;
    asm("mov dx,<oport>");
    asm("mov ax,<odata>");
    asm("w out [dx]");
}

unsigned int inByte(a)
{
    oport = BASE + a;
    asm("mov dx,<oport>");
    asm("b in [dx]");
    asm("mov <odata>,ax");
    return(odata & 0xff);
}

unsigned int inWord(a)
{
    oport = BASE + a;
    asm("mov dx,<oport>");
    asm("w in [dx]");
    asm("mov <odata>,ax");
    return(odata & 0xffff);
}

/***** Initialization Functions *****/

void initILUT()
{
    int         i;

    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Initializing ILUT...");
    outByte(0xc,0);
    for(i=0;i<256;i++)
        outByte(0xd,i);
}

void initOLUT()
{
    int         b,c,i;

    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Initializing OLUT bank 0...");
    for(c=0;c<3;c++) {
            outByte(1,c);                   /* color c,bank 0 */
            for(i=0;i<256;i++) {
                outByte(2,i);
                outByte(8,i);
            }
        }
        term_box_fill(23,0,80,1,0x8740,0,0xdb);
        term_printf(23,0,0xf540,"Initializing OLUT banks 1..15 to zero...");
        for(c=0;c<3;c++)
            for(b=1;b<16;b++) {
                outByte(1,(b<<4)|c);    /* color c,bank b */
                for(i=0;i<256;i++) {
                    outByte(2,i);
                    outByte(8,0);
                }
            }
        term_box_fill(23,0,80,1,0x8740,0,0xdb);
        term_printf(23,0,0xf540,"Initializing bitplane 8 to GREEN...");
        outByte(1,0x11);                    /* color green=1,bank 1 */
        for(i=0;i<256;i++) {
            outByte(2,i);
            outByte(8,0xff);
        }
        term_box_fill(23,0,80,1,0x8740,0,0xdb);
        term_printf(23,0,0xf540,"Initializing bitplane 9 to BLUE...");
        for(b=2;b<4;b++) {
            outByte(1,(b<<4)|2);            /* color blue=2,bank b */
            for(i=0;i<256;i++) {
                outByte(2,i);
                outByte(8,0xff);
            }
        }
        term_box_fill(23,0,80,1,0x8740,0,0xdb);
        term_printf(23,0,0xf540,"Initializing bitplane 10 to WHITE...");
        for(c=0;c<3;c++)
            for(b=4;b<8;b++) {
                outByte(1,(b<<4)|c);    /* color c,bank b */
                for(i=0;i<256;i++) {
                    outByte(2,i);
                    outByte(8,0xd0);
                }
            }
        term_box_fill(23,0,80,1,0x8740,0,0xdb);
        term_printf(23,0,0xf540,"Initializing bitplane 11 to RED...");
        for(b=8;b<16;b++) {
            outByte(1,b<<4);                /* color blue=2,bank b */
            for(i=0;i<256;i++) {
                outByte(2,i);
                outByte(8,0xff);
            }
        }
}

void initFRAME()
{
        int         i;
        unsigned    inByte();
```

```
    /** ----------------------------------- Clear Overlay Memory ------ **/
    outWord(0x10,0x4040);        /** PBCON: PB enable, Z-Mode **/
    for(i=0;i<8;i++)
        outWord(0x16,0x0000);    /** PBUF: fill with 0000 **/
    outWord(0x10,0x0000);        /** PBCON: PB disable, Z-mode    **/
    outWord(0x12,0x00ff);        /** HMASK: protect b0-b7 **/
    outByte(0x04,0x40);      /** ACQ: clear mode **/
    while((inByte(0x4)>>6) != 0x00);    /** wait until clear is done    **/
    outWord(0x12,0x0000);        /** HMASK: unprotect **/
    outWord(0x10,0x4040);        /** PBCON: PB enable, Z-mode **/
    outWord(0x05,0x03);      /** PTRCON: auto step = 8 **/
}

void initSETUP()
{
    scrSETUP();
    outByte(0,0x04);             /** Control Register **/
    outByte(6,0x0);          /** Pan Register **/
    outByte(7,0x0);          /** Scroll Register **/
    outByte(0xa,0x02);       /** Video Bus Control **/
    outByte(0xc,0x0);            /** ILUTA => ADC Ctrl Reg **/
    outByte(0xe,0x0);            /** ADC Ctrl Reg **/
    outByte(0xc,0x1);            /** ILUTA => NREF Reg **/
    outByte(0xe,0x0);            /** Negative Reference Reg **/
    outByte(0xc,0x2);            /** ILUTA => PREF Reg **/
    outByte(0xe,0xd8);       /** Positive Reference Reg **/
    outWord(0x10,0x4040);        /** Pixel Buffer Control **/
    outWord(0x12,0x0);       /** Host Mask Register **/
    outWord(0x14,0xff00);        /** Video Mask Register **/

    initILUT();
    initOLUT();
    initFRAME();
}

/***** Mode Handling Functions *****/
char modeGRAB()
{
    int             theChar;
    char            doneFlag=0,retval;
    void            modeILUT();
    int             testILUT(),testOLUT(),testSETUP(); /** DEBUG **/

    outByte(4,0xc0);
    scrGRAB();
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 's':
                doneFlag = 1;
                retval = SNAP;
                break;
            case 'e':
                modeILUT();
                scrGRAB();
                break;
            case 'm':
                modeMARK();
                scrGRAB();
                break;
            case 'q':
                doneFlag = 1;
                retval = QUIT;
                break;
        }
    }
    return retval;
}

char modeSNAP()
{
    int             theChar,lfs;
    char            doneFlag=0,retval;
    void            modeSAVE(),modeOLUT(),modeOVERLAY();

    for(lfs=0;!lfs;)                /** Wait until last frame.. **/
        lfs = (inByte(4)>>3)&1;
    outByte(4,0x80);                /** ...then issue snap cmd  **/
    scrSNAP();
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Picture snapped...on display");
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 's':
                modeSAVE();
                scrSNAP();
                break;
            case 'g':
                doneFlag = 1;
                retval = GRAB;
                break;
            case 'e':
                modeOLUT();
                scrSNAP();
                break;
            case 'o':
                modeOVERLAY();
                scrSNAP();
                break;
            case 'l':
                modeLOAD();
                scrSNAP();
                break;
            case 'h':
                modeHIST();
                scrSNAP();
                break;
            case 'q':
                doneFlag = 1;
                retval = QUIT;
                break;
        }
    }
    return retval;
}
```

```
void modeILUT()
{
    int             theChar;
    char            doneFlag=0;

    scrILUT();
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 'e':
                doneFlag = 1;
                break;
        }
    }
}

void modeMARK()
{
    int             theChar;
    char            doneFlag=0;

    scrMARK();
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 'h':
                modeHOZO();
                scrMARK();
                break;
            case 'v':
                modeVERT();
                scrMARK();
                break;
            case 'e':
                doneFlag = 1;
                break;
        }
    }
}

void modeHOZO()
{
    int             theChar;
    char            doneFlag=0;
    char            rownum[20];
    int             oldrow=0,newrow;

    scrHOZO();
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 'n':
                if(term_input(15,10,rownum,20,"Row number:",0x8740,0xb0,0x8340)!=0)
{
                    term_box_fill(15,0,80,1,0x8040,0,0xdb);
                    newrow = atoi(rownum);
                    term_printf(17,10,0x8740,"newrow = %10d",newrow);
                    if(newrow>=0 && newrow<511) {
                        markRow(oldrow,0);
                        markRow(newrow,0x0200);
                        oldrow=newrow;
                    }
                }
                break;
            case 'e':
                markRow(oldrow,0);
                doneFlag = 1;
                break;
        }
    }
}

void modeVERT()
{
    int             theChar;
    char            doneFlag=0;
    char            colnum[20];
    int             oldcol=0,newcol;

    scrVERT();
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 'n':
                if(term_input(15,10,colnum,20,"Column
number:",0x8740,0xb0,0x8340)!=0) {
                    term_box_fill(15,0,80,1,0x8040,0,0xdb);
                    newcol = atoi(colnum);
                    term_printf(17,10,0x8740,"newcol = %10d",newcol);
                    if(newcol>=0 && newcol<1024) {
                        markCol(oldcol,0);
                        markCol(newcol,0x0200);
                        oldcol=newcol;
                    }
                }
                break;
            case 'e':
                markCol(oldcol,0);
                doneFlag = 1;
                break;
        }
    }
}

void markRow(row,value)
int row,value;
{
    int             i,x,y;

    outWord(0x10,0x4040);           /** PBCON: enable pb, zmode **/
    for(i=0;i<8;i++)
        outWord(0x16,value);        /** Setup pixbuf **/
    outWord(0x10,0x0000);           /** PBCON: protect pb, zmode **/
    outWord(0x12,0x00ff);           /** HMASK: protect image **/
    for(y=row;y<row+64;y++) {
        outWord(0x1a,y);    /** YPTR **/
        outWord(0x18,268);  /** XPTR **/
        for(x=268;x<332;x+=8)
            outWord(0x1c,0);    /** Data Port **/
    }
    outWord(0x12,0x0000);           /** HMASK: unprotect image **/
    outWord(0x10,0x4040);           /** PBCON: enable pb, zmode **/
}

void markCol(col,value)
int col,value;
{
    int             i,x,y;

    outWord(0x10,0x4040);           /** PBCON: enable pb, zmode **/
    outByte(0x05,0x07);             /** PTRCTL: step 1 **/
    outWord(0x12,0x00ff);           /** HMASK: protect image **/
    for(y=246;y<310;y++) {
        outWord(0x1a,y);            /** YPTR **/
        outWord(0x18,col);          /** XPTR **/
        for(x=0;x<64;x++)
            outWord(0x1c,value);        /** Data Port **/
    }
    outWord(0x12,0x0000);           /** HMASK: unprotect image **/
    outWord(0x10,0x4040);           /** PBCON: enable pb, zmode **/
    outByte(0x05,3);                /** PTRCTL: step 8 **/
}

void modeOLUT()
{
    int             theChar;
    char            doneFlag=0;

    scrOLUT();
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 'e':
                doneFlag = 1;
                break;
        }
    }
}

void modeSAVE()
{
    int             theChar;
    char            doneFlag=0,fnPtr[16];
    unsigned int    x,y,s,i,d,size;
    unsigned        seg[4];
    unsigned char   *p,temp;
    long            j;
    FILE            *fp;

    scrSAVE();
    term_printf(15,10,0x8740,"8 legal characters only.");
    if(term_input(14,10,fnPtr,15,"Image name:",0x8740,0xb0,0x8340) != 0) {
        term_box_fill(6,0,80,1,0x8740,0,0xdb);
        term_printf(6,0,0xf540,"(S)ave (E)scape");
        term_box_fill(14,10,70,2,0x8040,0,0xdb);
        for(i=0,d=0;i<8 && !d;i++)
            if(*(fnPtr+i) == '\0') { d=1; i--;}
        strcpy(fnPtr+i,".vid");
        term_printf(14,10,0x8740,"Enter (S)ave to save image as '%s'",fnPtr);
        term_box_fill(23,0,80,1,0x8740,0,0xdb);
    }
    else {
        term_box_fill(23,0,80,1,0x8740,0,0xdb);
        term_printf(23,0,0xf540,"Image save ABORTED");
        doneFlag = 1;
    }
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 'e':
                doneFlag = 1;
                break;
            case 's':
                /** ---------------------------------- Allocate Memory ---- **/
                term_printf(16,10,0x8440,"Allocating segments...");
                term_box_fill(23,0,80,1,0x8740,0,0xdb);
                if((seg[0] = alloc_segment(0x1000)) == 0) {
                    term_printf(19,10,0xf540,"Memory seg0 FAILED");
                    break;
                }
                if((seg[1] = alloc_segment(0x1000)) == 0) {
                    term_printf(20,10,0xf540,"Memory seg1 FAILED");
                    break;
                }
                if((seg[2] = alloc_segment(0x1000)) == 0) {
                    term_printf(21,10,0xf540,"Memory seg2 FAILED");
                    break;
                }
                if((seg[3] = alloc_segment(0x1000)) == 0) {
                    term_printf(22,10,0xf540,"Memory seg3 FAILED");
                    break;
                }
                /** ---------------------------------- Download Image ----- **/
                term_printf(17,10,0x8440,"Downloading image...");
                for(s=0;s<4;s++) {
                    set_extra_segment(seg[s]);
                    p=0;
                    for(y=0;y<128;y++) {
                        outWord(0x1a,128*s+y);      /** Y pointer register  **/
                        outWord(0x18,64);           /** X pointer register  **/
                        for(x=0;x<64;x++) {
                            inWord(0x1c);           /** read 8 pixs into PB **/
                            for(i=0;i<8;i++)
                                @(p+512*y+8*x+i) = (unsigned char)
(inWord(0x16)&0xff);
                        }
                    }
                }
                /** ---------------------------------- Write file --------- **/
                term_printf(18,10,0x8440,"Writing file...");
                if((fp = fopen(fnPtr,"w")) == 0) {
                    term_box_fill(23,0,80,1,0x8740,0,0xdb);
                    term_printf(23,0,0xf540,"cannot open file '%s'",fnPtr);
                    break;
                }
                size = 512;
                fput((char *) &size,2,fp);
                for(s=0;s<4;s++) {
                    term_printf(18,40,0x8440,"s=%d",s);
                    set_extra_segment(seg[s]);
                    for(j=0;j<0x10000;j++) {
                        temp = @((char *) j);
                        putc(temp,fp);
                    }
                }
                fclose(fp);
                /** --------------------------------- Clean up ----------- **/
                for(s=0;s<4;s++)
                    free_segment(seg[s]);
                term_box_fill(23,0,80,1,0x8740,0,0xdb);
                term_printf(23,0,0xf540,"Image saved as '%s'",fnPtr);
                doneFlag = 1;
                break;
        }
    }
    term_box_fill(14,10,70,6,0x8040,0,0xdb);
}

void modeOVERLAY()
{
    int             theChar;
    char            doneFlag=0;

    scrOVERLAY();
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 'e':
                doneFlag = 1;
                break;
        }
    }
}

void modeLOAD()
{
    int                 theChar,rv,tint;
    char                doneFlag = 0,fnPtr[20];
    FILE                *fp;
    unsigned            seg[4],size,x,y,s,i,d;
    long                j;
    char                temp,*p;

    scrLOAD();
    /** -------------------------------------- List legal files --- **/
    term_printf(8,0,0x8540,"Image files:");
    fp = fopen("/user/ali/ofg","rq");
    dir_set_first(fp);
    term_cur(9,0); term_colour(0x87);
```

```
while(rv = dir_next_fname(fp,"*.vid",fnPtr))
    if(rv==1)
        tprintf("\t%s\n",fnPtr);
term_colour(0x83);
/** ------------------------------------- Obtain choice ------ **/
if(term_input(7,10,fnPtr,15,"Image name:",0x8740,0xb0,0x8340) != 0) {
    term_box_fill(7,0,80,16,0x8040,0,0xdb);
    term_box_fill(6,0,80,1,0x8740,0,0xdb);
    term_printf(6,0,0xf540,"(L)oad (E)scape");
    term_printf(14,10,0x8740,"Enter (L)oad to load image '%s'",fnPtr);
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
}
else {
    term_box_fill(23,0,80,1,0x8740,0,0xdb);
    term_printf(23,0,0xf540,"Image load ABORTED");
    doneFlag = 1;
}
while(!doneFlag) {
    theChar = getchar();
    switch(theChar) {
        case 'e':
            doneFlag = 1;
            break;
        case 'l':
            /** -------------------------- Validate selection - **/
            term_box_fill(23,0,80,1,0x8740,0,0xdb);
            if((fp = fopen(fnPtr,"r")) == 0) {
                term_printf(23,0,0xf540,"File access failed.");
                doneFlag = 1;
                break;
            }
            /** -------------------------- Allocate buffers --- **/
            term_printf(16,10,0x8440,"Allocating segments...");
            if((seg[0] = alloc_segment(0x1000)) == 0) {
                term_printf(19,10,0xf540,"Memory seg0 FAILED");
                break;
            }
            if((seg[1] = alloc_segment(0x1000)) == 0) {
                term_printf(20,10,0xf540,"Memory seg1 FAILED");
                break;
            }
            if((seg[2] = alloc_segment(0x1000)) == 0) {
                term_printf(21,10,0xf540,"Memory seg2 FAILED");
                break;
            }
            if((seg[3] = alloc_segment(0x1000)) == 0) {
                term_printf(22,10,0xf540,"Memory seg3 FAILED");
                break;
            }
            /** -------------------------- Read file ---------- **/
            term_printf(17,10,0x8440,"Reading file...");
            fget((char *) &size,2,fp);
            for(s=0;s<4;s++) {
                term_printf(17,40,0x8440,"s=%d",s);
                set_extra_segment(seg[s]);
                for(j=0;j<0x10000;j++) {
                    temp = getc(fp);
                    @((char *) j) = temp;
```

```
                        }
                    }
                    fclose(fp);
                    /** ------------------------ Load frame buffer -- **/
                    term_printf(18,10,0x8440,"Uploading image...");
                    for(s=0;s<4;s++) {
                        set_extra_segment(seg[s]);
                        p=0;
                        for(y=0;y<128;y++) {
                            outWord(0x1a,128*s+y);      /** Y pointer register  **/
                            outWord(0x18,0);            /** X pointer register  **/
                            /*=== Clear Pixel Buffer ===*/
                            for(x=0;x<8;x++) {
                                for(i=0;i<8;i++)
                                    outWord(0x16,0);    /** Pixel Buffer        **/
                                outWord(0x1c,0);        /** Data Port           **/
                            }
                            /*=== Write data to pixel buffer in grps of 8 ===*/
                            for(x=0;x<64;x++) {
                                for(i=0;i<7;i++)
                                    outWord(0x16,(int) @(p+512*y+8*x+i));
                                outWord(0x1c,(int) @(p+512*y+8*x+7));        /** write 8
pixs to Mem **/
                            }
                            for(x=0;x<8;x++) {
                                for(i=0;i<8;i++)
                                    outWord(0x16,0);
                                outWord(0x1c,0);
                            }
                        }
                    }
                    /** ------------------------ Clean up ----------- **/
                    fclose(fp);
                    for(s=0;s<4;s++)
                        free_segment(seg[s]);
                    term_box_fill(23,0,80,1,0x8740,0,0xdb);
                    term_printf(23,0,0xf540,"Image '%s' loaded.",fnPtr);
                    doneFlag = 1;
                    break;
        }
    }
    term_box_fill(7,0,80,16,0x8040,0,0xdb);
}

void modeHIST()
{
    int             theChar;
    char            doneFlag=0;
    unsigned        x,y,i,n,level;
    long            max,bin[64],total;
    unsigned        din;
    unsigned char   d8b,joe,dsh;

    scrHIST();
    term_bar(22,9,1,16,16,0x8740);
    term_bar(22,74,1,16,16,0x8740);
    for(i=0;i<64;i++)
        bin[i] = 0;
    for(y=0;y<480;y++) {
        outWord(0x1a,y);            /** Y PTR register   **/
        outWord(0x18,64);           /** X PTR register   **/
        for(x=64;x<576;x+=8) {
            inWord(0x1c);           /** DATA register: fill PBUF    **/
            for(i=0;i<8;i++)
                ++bin[((unsigned char) inWord(0x16)>>2)];
        }
    }
    for(i=0,max=0;i<64;i++)
        if(bin[i]>max) max = bin[i];
    for(n=11;max>(((long) 1) <<n);n++);
    term_printf(16,0,0x8540,"c=2^%d",n-4);
    for(i=0;i<64;i++) {
        level = bin[i]>>(n-5);
        term_bar(22,10+i,2,level,level,0x8140);
    }
    while(!doneFlag) {
        theChar = getchar();
        switch(theChar) {
            case 'e':
                doneFlag = 1;
                break;
        }
    }
    term_box_fill(7,0,80,16,0x8040,0,0xdb);
}
/**************************************************************************/
/***** Debugging Routines                                            *****/

int testILUT()
{
    unsigned int            joe,fails,i;

    outByte(0xc,0);
    for(i=0,fails=0;i<256;i++) {
        joe = inByte(0xd);
        if(joe != i)
            fails++;
    }
    term_printf(8,0,0x8740,"%d failures in ILUT",fails);
    return fails;
}

int testOLUT()
{
    int             b,c,i,stat[3][16];
    register int    joe,dude,fail;

    for(c=0,fail=0;c<3;c++) {
        outByte(1,c);               /* color c,bank 0 */
        for(i=0;i<256;i++) {
            outByte(2,i);
            if(i != inByte(8))
                fail++;
        }
        if(fail != 0)
            stat[c][0] = 2;
        else stat[c][0] = 3;
    }
    for(b=1;b<16;b++)
        for(c=0,fail=0;c<3;c++) {
            outByte(1,(b<<4)|c);    /* color c,bank b */
            outByte(2,0);
            joe = inByte(8);
            for(i=1;i<256;i++) {
                outByte(2,i);
                if(joe != inByte(8))
                    fail++;
            }
            if(fail != 0)
                stat[c][b] = 2;
            else if(joe == 0)
                stat[c][b] = 0;
            else stat[c][b] = 1;
        }
    for(b=0;b<16;b++)
        term_printf(b+7,40,0x8740,"%10d%10d%10d",stat[0][b],stat[1][b],stat[2][b]);
}

void testSETUP()
{
    unsigned int            i,reg[32];
    unsigned int            inByte(),inWord();

    outByte(0x00,0x04);
    term_printf(15,0,0x8440,"%x",inByte(0x00));
    for(i=0;i<16;i++)
        reg[i] = inByte(i);
    for(i=16;i<32;i+=2)
        reg[i] = inWord(i);
    for(i=0;i<16;i++)
        term_printf(i+7,40,0x8740,"%x H",reg[i]);
    for(i=0;i<8;i++)
        term_printf(2*i+7,60,0x8740,"%x H",reg[i*2+16]);
}
main()
{
    unsigned        option_save;
    char            theChar;

    initSETUP();
    option_save = get_option(stdin);
    set_option(stdin,option_save & ~3);
    for(theChar=GRAB;theChar!=QUIT;) {
        switch(theChar) {
            case GRAB:
                theChar = modeGRAB();
                break;
            case SNAP:
                theChar = modeSNAP();
                break;
        }
    }
    scrQUIT();
}
```

[illegible]