A VISION SYSTEM FOR LANDING AN UNMANNED AERIAL VEHICLE

by

Courtney S. Sharp

Memorandum No. UCB/ERL M01/10

18 December 2000

A VISION SYSTEM FOR LANDING AN UNMANNED AERIAL VEHICLE

by

Courtney S. Sharp

Memorandum No. UCB/ERL M01/10

18 December 2000

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Vision System for Landing an Unmanned Aerial Vehicle

by Courtney S. Sharp

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor S. Shankar Sastry, Research Advisor

Dec 18, 2000

Date

* * * * * *

J Malik

Professor Jitendra Malik, Second Reader

Dec 20, 2000

Date

A Vision System for Landing an Unmanned Aerial Vehicle

Abstract

A Vision System for Landing an Unmanned Aerial Vehicle

by

Courtney S. Sharp

Master of Science in Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Shankar Sastry, Chair

We present the design and implementation of a real-time vision system for a rotor-craft unmanned aerial vehicle to land onto a known landing target. This vision system consists of customized software and off-the-shelf hardware which perform image processing, segmentation, feature point extraction, camera pan/tilt control, and motion estimation. We introduce the design of a landing target which significantly simplifies the computer vision tasks such as corner detection and correspondence matching. Customized algorithms are developed to allow for real-time computation at a frame rate of 30Hz. Such algorithms include certain linear and nonlinear optimization schemes for model-based camera pose estimation. We present flight test results which show that our vision-based state estimates are accurate to within 5cm in each axis of translation and 5 degrees in each axis of rotation, making vision a viable sensor to be placed in the control loop of a hierarchical flight vehicle management system.

To my family and friends.

Contents

List of Figures

List of Tables

Acknowledgements

Most importantly, I would like to thank my research advisor, Professor Shankar Sastry, for whom I have the greatest amount of respect and admiration and who provides tireless inspiration. I would also like to thank Professor Jitendra Malik whose courses in computer vision superbly guided me in this research. I am deeply grateful to Omid Shakernia my excellent research partner and the first person I look for when something isn't working. I am debted to David Shim and Rene Vidal for their tremendous support, Yi Ma for his guidance and suggestions, and Tullio Celano for building two fantastic landing platforms. I would also like to thank the rest of the Berkeley Aerobot team: John Koo, Bruno Sinopoli, Jin Kim, Shahid Rashid, Will Morrison, Santosh Phillip, Cedric Ma, Peter Ray, and Erica Morrison who together create an absolutely top-notch research group and who it was a privilege to work with. I would like to thank my roommates and fellow graduate students: Shawn Schaffert, Johan Vanderhaegen, and Mark Donahue who provide stimulating and engaging conversation. Finally, I would like to thank my family and loved ones for their endless support. This work is dedicated to them.

This research was supported by ARO MURI grant DAAH04-96-1-0341, ONR grant N00014-00-1-0621, and DARPA grant F33615-98-C-3614.

Chapter 1

Introduction

Computer vision is gaining importance as a cheap, passive and information-rich source complementing the sensor suite for control of Unmanned Aerial Vehicles (UAV). A vision system on board a UAV typically augments a sensor suite that might include Global Positioning System (GPS), Inertial Navigation Sensors (INS), laser range finders, a digital compass, and sonar [2, 9, 11]. Furthermore, the challenge of replacing these conventional sensors by vision for real-time control tasks has not been well addressed in the computer vision community. Most pose and structure estimation algorithms developed based on existing multiple view geometry theory are in fact not suitable for real-time applications, simply due to their computational cost. Regardless, design of any real-time vision system is a daunting task: It involves a systematic integration of hardware, low level image processing (such as segmentation and feature extraction); multiple view geometry (such as pose and structure estimation); and synthesis of real-time controllers.

Because of its structured nature, the task of autonomous landing is well-suited for vision-based state estimation and control and has recently been an active topic of research [5, 7, 8, 11, 12]. In [6] a technique is presented for estimating the pose relative to a known object given a scaled orthographic projection model of a camera. In [12], the use of vanishing points of parallel lines on a landmark is proposed for the purpose of estimating the location and orientation of a UAV relative to a landing pad. Because their technique relies on vanishing points of parallel lines, their algorithm is most sensitive to noise and gives the worst pose estimates when it matters the most: when the UAV is directly over the pad.

In this paper, we introduce a design and implementation of a real-time vision system for a rotor-craft UAV which estimates its pose and speed relative to a known landing target at 30Hz. Our vision system uses *customized* vision algorithms and *off-the-shelf* hardware to perform in real-time: image processing, segmentation, feature point extraction, camera control, as well as both linear and nonlinear optimization for model-based pose estimation. Flight test results on our UAV testbed show our vision-based state estimates are accurate to within 5cm in each axis of translation and 5 degrees in each axis of rotation, making it a viable sensor to be placed in the control loop of a hierarchical flight vehicle management system such as the one described in [9].

Section 2 contains a detailed description of each component of our vision system. Section 3 presents experimental results from flight experiments, and Section 4 gives concluding remarks and directions for future research. Appendix A provides an overview of the code used throughout the vision system.

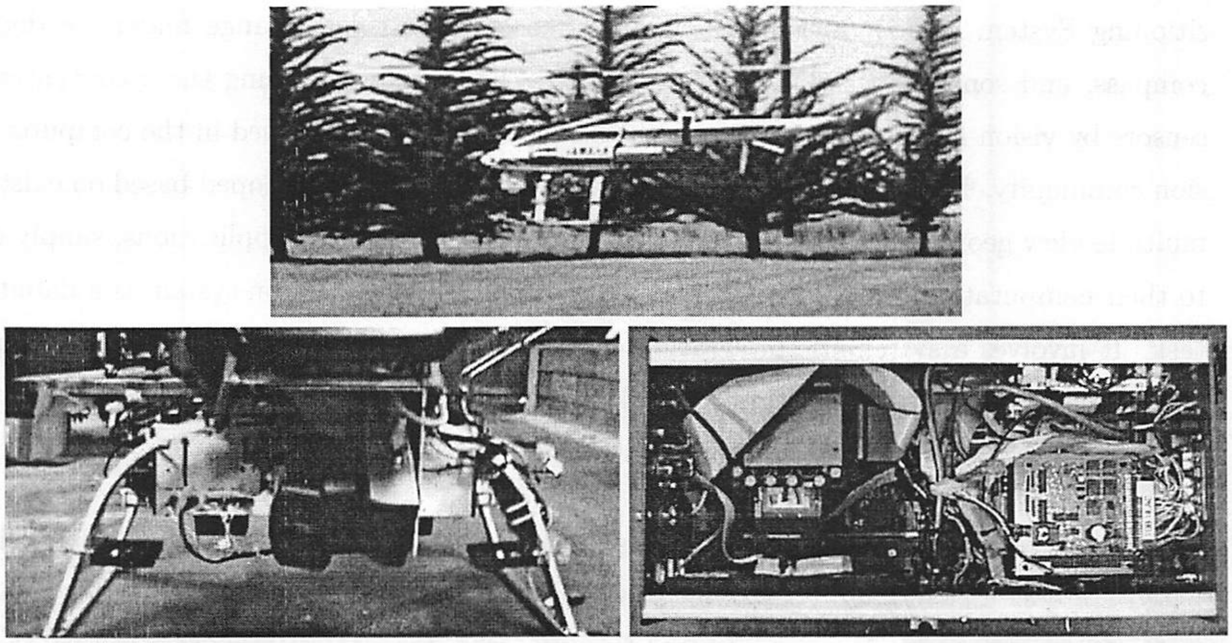

Figure 1.1: Photos of our UAV testbed: in-flight Yamaha R-50 helicopter (top), mounted Sony EVI-D30 camera and computer box (bottom-left), and on-board navigation and vision Littleboard computers (bottom-right).

Chapter 2

Vision System Algorithm Design

Our vision system software consists of two main stages of execution: image processing and state estimation, each with a sequence of subroutines. Figure 2.1 shows a flow-chart of the overall algorithm. This chapter endeavors to describe each stage of execution in detail.

2.1 Low Level Image processing

Our goal for image processing is to locate the landing target then extract and label its feature points. This process includes: 1. thresholding the grayscale image to a binary one, 2. segmenting the landing target out of the background, 3. detecting the corners in landing target, and finally 4. labeling those corners. In order to simplify the image processing, the landing target must be easy to identify and segment from the image background, provide distinctive feature points, simplify feature labeling, and allow for algorithms that can execute in realtime using off-the-shelf hardware.

Figure 2.2(a) shows our landing target design. The connected white border enclosing the target simplifies segmenting it from the background, given that the landing target lies on a dark background and that no other connected white region completely encloses it. Figure 2.2(b) shows the feature point labeling on the corners of the interior white squares of the landing target. We choose corner detection over other forms of feature point extraction

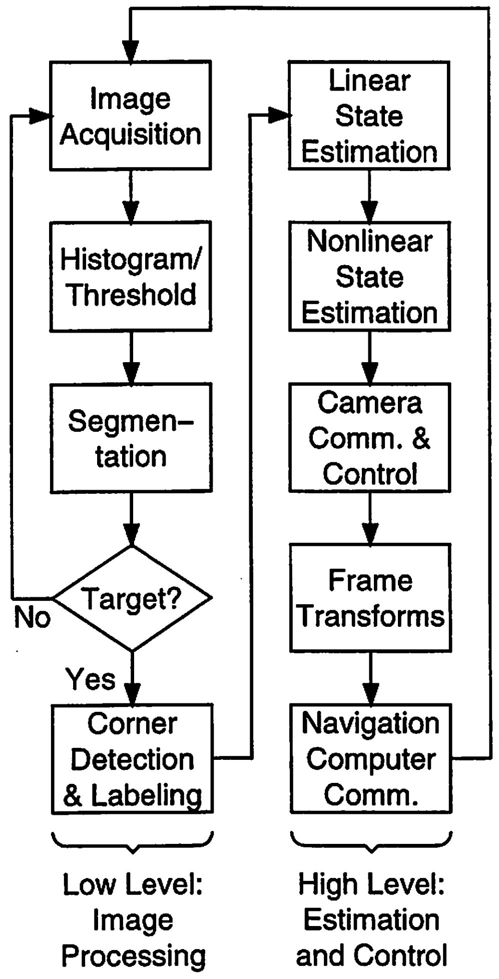
Image Acquisition
Histogram/ Threshold
Segmen-tation
Target?
No
Yes
Corner Detection & Labeling
Linear State Estimation
Nonlinear State Estimation
Camera Comm. & Control
Frame Transforms
Navigation Computer Comm.
Low Level: Image Processing
High Level: Estimation and Control

Figure 2.1: Vision system software flow-chart: image processing followed by estimation and control.

because it is simple, robust, and provides a high density of feature points per image pixel area. We choose squares over other n-sided polygons because they maximize the quality of the corners under perspective projection and pixel quantization. Moreover, our particular organization of the squares in the target allows for straight-forward feature point labeling invariant of Euclidean motion and perspective projection, as shown in the section "Feature labeling."

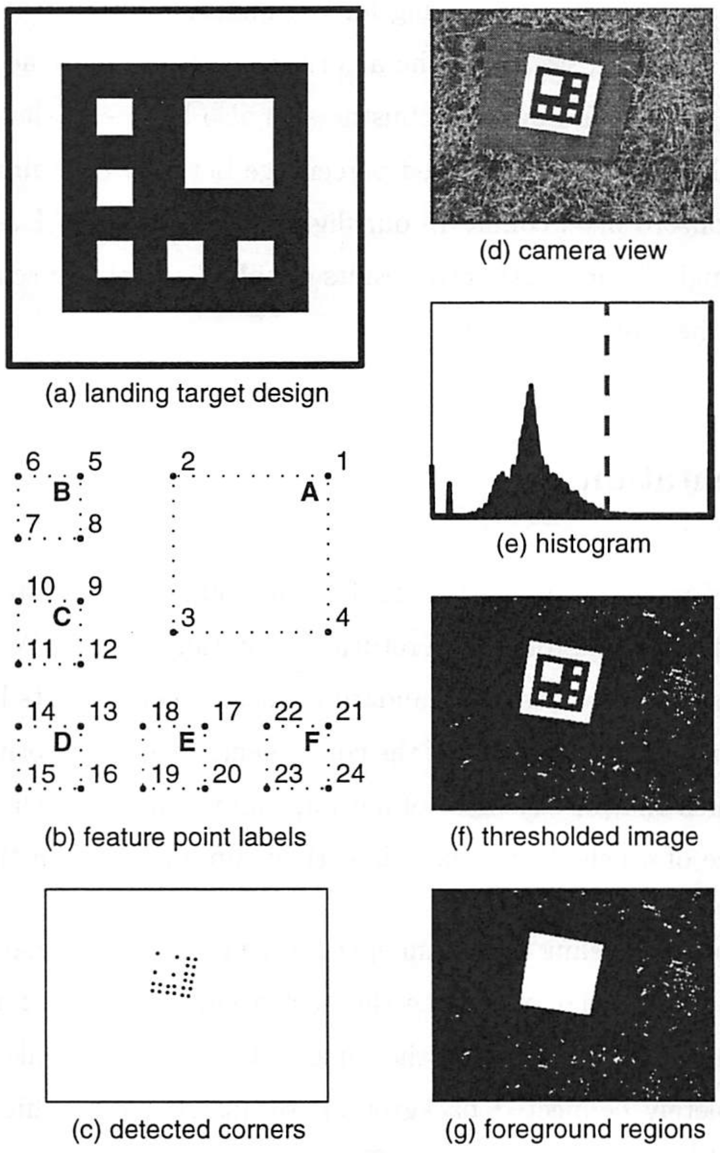

Figure 2.2: Landing target design and image processing. The target design (a) is made for simple feature labeling (b), robust feature point extraction (c), and easy image segmentation and identification (d-g).

2.1.1 Thresholding

The thresholding algorithm must produce a binary image so that the black and white regions of the landing target are preserved and that the primary image background is connected and black. During the development of the image processing algorithm, we explored a variety of techniques for selecting a threshold from the image histogram. We found the following schemes too simplistic or less robust: selecting absolute gray level, performing gradient descent from the mean gray level, or solving for the minimum of an nth order polynomial fit to the histogram. However, we found the algorithm with the most acceptable compromise between best-case quality and overall robustness to also be one of the simplest. We choose the threshold on the histogram as a fixed percentage between the minimum and maximum gray levels with nonzero pixel count. In our flight tests, we use 65% as the threshold level. Figure 2.2(d) through Figure 2.2(f) give results of this thresholding scheme tested on a real image input from the on-board camera.

2.1.2 Segmentation

Given a binary image as in Figure 2.2(f), our segmentation stage must separate the landing target from the background and return the interior squares. We segment the landing target via two consecutive passes of a standard connected components labeling algorithm [3] based on *4-connectivity.* A single pass of the components labeling algorithm numbers distinct regions of pixels such that any element of a group can reach any other element of the group through a sequence of 4-neighbors (pixels left, right, up, or down) in the group.

The first pass of the labeling algorithm operates on the original binary image to determine the initial regions. From that, we define the background to be the largest single region of black pixels touching the perimeter of the image. To increase the likelihood of a correctly labeled and completely connected background, we insert a pixel-thick perimeter of black pixels around the image prior to labeling. This connects separated background regions that may have been split by edge-to-edge white regions or by the landing target itself.

The second pass of the labeling algorithm operates on a new binary image where we define the background as black and all other regions as white, see Figure 2.2(g). We then assert the landing target to be the foreground region containing seven white components and one black component. We make no further consistency check on the landing target, leading to the implicit assumption that no foreground region other than the landing target contains a similar distribution of component regions. We improve robustness to noise by ignoring (erroneous) component regions with less than a threshold number pixels.

2.1.3 Corner Detection

Once we have segmented the landing target from the background, our next stage of processing is to detect the corner features. The corner detection algorithm we face is by design highly structured: We need to detect the corners of 4-sided polygons in a binary image. The structured nature of the problem allows us to avoid the computational cost of a general purpose corner detector and to make the detected corners more robust through the design of a custom corner detection algorithm. The fundamental invariant in our corner detector is that *convexity* is preserved under Euclidean motion and perspective projection. This implies that for a line through the interior of a convex polygon, the set of points in the polygon with maximal distance from each side of the line contain at least two distinct corners of the polygon.

To find two arbitrary corners of a 4-sided polygon, we compute the perpendicular distance from each edge point to the vertical line passing through the center of gravity of the polygon. If there is more than one point with maximal distance on a side of the line, we choose the point which is farthest from the center of gravity. We then find the third corner as the point of maximum distance from the line connecting the first two corners. Finally, we find the fourth corner as the point of the polygon with maximum distance to the triangle defined by the first three corners. Figure 2.2(c) shows the output of our corner detection algorithm on a sample image.

2.1.4 Feature Labeling

After we extract the corner features of the landing target, the next stage of processing is to label the corners according to the numbering scheme shown in Figure 2.2(b). Our feature labeling algorithm consists of two steps: first, each square is uniquely identified based on its center of gravity, then the corners within each square are identified.

The technique depends on the basic property that the clockwise ordering of a set of coplanar points is preserved under Euclidean motion and perspective projection, given that the camera always stays on one side of the target plane. We state this fact through the following proposition:

Proposition 1. *Given* $v_1, v_2, v_3 \in \mathbb{R}^3$, *for any* $\alpha_i > 0$, *and* $(R, p) \in SE(3)$, *the vectors* $\tilde{v}_i = \alpha_i(Rv_i + p)$ *satisfy:*

$$\mathrm{sign}(\tilde{v}_1^T \tilde{v}_2 \times \tilde{v}_3) = \mathrm{sign}(v_1^T v_2 \times v_3).$$

Proof. Since the volume of the parallelepiped formed by v_1, v_2, v_3 is preserved under Euclidean motion, we have $v_1^T v_2 \times v_3 = (Rv_1 + p)^T (Rv_2 + p) \times (Rv_3 + p)$. Hence, we get

$$\begin{aligned} \mathrm{sign}(\tilde{v}_1^T \tilde{v}_2 \times \tilde{v}_3) &= \mathrm{sign}(\alpha_1 \alpha_2 \alpha_3 v_1^T v_2 \times v_3) \\ &= \mathrm{sign}(v_1^T v_2 \times v_3). \end{aligned}$$

□

Proposition 1 essentially guarantees that a technique for correspondence matching based on clockwise ordering of feature points on the landing target works for any given image of the target. We begin by labeling the squares of the target starting with the identification of square D shown in Figure 2.2(b). For each square, we compute the vectors between its center to the centers of the other squares. We identify square D as the square with two pairs of collinear vectors. We identify the remaining squares by ordering them counterclockwise from square D by taking the arctangent of each vector with respect to some constant vector. Finally, we order squares associated with the collinear vectors from square D by their distance from it.

To identify the corners of a square, we calculate the vectors between its corners and the center of another particular square in the landing target. One such vector will always be first on counterclockwise ordering, and we identify the associated corner this way. We determine the labeling of the remaining corners by ordering them counterclockwise from the identified corner.

2.2 Pose Estimation

2.2.1 Geometry of Planar Features

Given the labeled feature points, estimating the UAV state is the so-called model based camera pose estimation problem from computer vision. We apply both linear and nonlinear optimization algorithms toward this problem. The linear optimization algorithm is globally robust but sensitive to noise. The nonlinear optimization algorithm requires adequate initialization but is more robust to noise. Thus, we solve the camera pose estimation problem by first solving the linear problem then using those results to initialize the nonlinear algorithm.

The equation relating a point in the landing pad coordinate frame to the image of that point in the camera-head frame is given by

$$\lambda_i \mathbf{x}_i = APgq_i, \tag{2.1}$$

where $\lambda_i \in \mathbb{R}$ is an unknown scale, $\mathbf{x}_i \in \mathbb{R}^3$ is the homogeneous coordinates of the feature point in the image plane, $A \in \mathbb{R}^{3\times3}$ is the camera calibration matrix, $P = [I\ 0] \in \mathbb{R}^{3\times4}$ is the projection matrix, $g \in SE(3)$ is the homogeneous representation of the Euclidean motion between the landing pad coordinate frame and the camera-head frame, and $q_i \in \mathbb{R}^4$ is the homogeneous representation of the point in the world. Using a *calibrated* pinhole model for the perspective projection of the camera, without loss of generality we set $A = I_{3\times3}$. Then the scale term λ_i is given by

$$\lambda_i = e_3^T Pgq_i, \tag{2.2}$$

where $e_3 = [0\ 0\ 1]^T \in \mathbb{R}^3$. Equations (2.1) and (2.2) together imply the following constraint

$$(\mathbf{x}_i e_3^T - I)[R\ p]q_i = 0. \tag{2.3}$$

We have knowledge of each q_i from the geometry of the designed landing target. The corner detection algorithm extracts each feature point $\mathbf{x}_i$ in the image frame. The feature labeling algorithm associates each $\mathbf{x}_i$ with its corresponding q_i. From this data, we need to recover the camera pose $Pg = [R\ p]$ where $R \in SO(3)$ is the rotation and $p \in \mathbb{R}^3$ is the translation from the landing target to the camera head.

2.2.2 Linear Optimization

Since all feature points lie on the plane of the landing target, without loss of generality we may choose the inertial coordinate frame such that $e_3^T q_i = 0$ for all i. Then for the calculation of $[R\ p] = [r_1\ r_2\ r_3\ p]$, we remove r_3 and the third component of q_i. Taking $q_i = [q_{i1}\ q_{i2}\ q_{i3}\ 1]^T$ and $[R\ p] = [r_1\ r_2\ r_3\ p]$, Equation 2.3 implies

$$\left(\mathbf{x}_i e_3^T - I\right)[r_1\ r_2\ p]\begin{bmatrix} q_{i1} \\ q_{i2} \\ 1 \end{bmatrix} = 0. \tag{2.4}$$

Since the above equation is linear in $[r_1\ r_2\ p]$, we can reorganize Equation (2.4) into vector form for each feature correspondence pair $(\mathbf{x}_i, q_i)$ and stack all the equations to get

$$F\begin{bmatrix} r_1 \\ r_2 \\ p \end{bmatrix} = 0, \tag{2.5}$$

where $F = [F_1^T \cdots F_n^T]^T \in \mathbb{R}^{2n\times 9}$, n is the number of feature points on the landing target, and

$$F_i = \begin{bmatrix} q_{i1} & 0 & -q_{i1}\mathbf{x}_{i1} & q_{i2} & 0 & -q_{i2}\mathbf{x}_{i1} & 1 & 0 & -\mathbf{x}_{i1} \\ 0 & q_{i1} & -q_{i1}\mathbf{x}_{i2} & 0 & q_{i2} & -q_{i2}\mathbf{x}_{i2} & 0 & 1 & -\mathbf{x}_{i2} \end{bmatrix}, \tag{2.6}$$

where $\mathbf{x}_i = [\mathbf{x}_{i1}\ \mathbf{x}_{i2}\ 1]^T \in \mathbb{R}^3$.

By a slight modification of a well known result on the planar structure from motion problem (see, for example, Weng [10]), it can be shown that if there are at least 4 features points such that no three are collinear, then $\text{rank}(F) = 8$. However, due to noise in corner detection, in practice F is always full rank. Hence we compute the least squares estimate of the null space of F by applying standard SVD techniques to compute the singular vector $[\tilde{r}_1^T\ \tilde{r}_2^T\ \tilde{p}^T]^T \in \mathbb{R}^9$ corresponding to the smallest singular value of F. Given the "positive-deth constraint" that the landing pad is in front of the camera, if necessary we negate the singular vector result of the SVD computation to ensure that $\tilde{p}_3 \geq 0$.

Next, we solve for scale of the translation $p \in \mathbb{R}^3$ by computing the scale factor on the vector $[\tilde{r}_1^T\ \tilde{r}_2^T\ \tilde{p}^T]^T$ that gives $\tilde{r}_1$ and $\tilde{r}_2$ unit norm; namely, $p = 2\tilde{p}/(\|\tilde{r}_1\| + \|\tilde{r}_2\|)$.

Finally, we solve for the rotation matrix R from the estimates $\tilde{r}_1$ and $\tilde{r}_2$ in two steps. First we project the matrix $[\tilde{r}_1\ \tilde{r}_2\ 0] \in \mathbb{R}^{3\times 3}$ onto the group of orthogonal matrices $O(3)$ by computing the SVD of $[\tilde{r}_1\ \tilde{r}_2\ 0] = U\Sigma V^T$ and setting $R = UV^T$. To ensure that R is a rotation matrix, we need $\det(R) = 1$. Thus, if our SVD computation yields $\det(UV^T) = -1$, we flip the sign of the third column vector of R, which is equivalent to setting $R = UV^TQ$, where $Q = \operatorname{diag}(1, 1, -1) \in \mathbb{R}^{3\times 3}$.

In practice, the linear algorithm described above is quite noisy because it estimates 9 parameters for a system of equations with 6 degrees of freedom. However, the linear estimate is close enough to the true solution to serve as a good initialization for the nonlinear optimization technique.

2.2.3 Nonlinear Optimization

The nonlinear algorithm optimizes over the *reprojection error* $G = [G_1^T \cdots G_n^T]^T \in \mathbb{R}^{2n}$ where $G_i \in \mathbb{R}^2$ is given by

$$G_i = (\mathrm{x}_i e_3^T - I)[R\ p]q_i, \tag{2.7}$$

where the last row is ignored from the right hand side matrix. The rotation matrix is parameterized by ZYX-Euler angles θ_1, θ_2, θ_3, where $\theta_i \in (-\pi, \pi]$ and $R = e^{\hat{e}_3\theta_3}e^{\hat{e}_2\theta_2}e^{\hat{e}_1\theta_1}$. The estimation parameters for the nonlinear optimization are $\beta = [\theta_1\ \theta_2\ \theta_3\ p_1\ p_2\ p_3]^T$.

We apply the vector form of the Newton-Raphson method to iteratively solve for β:

$$\beta_{n+1} = \beta_n - k_n(D_\beta G|_{\beta_n})^\dagger\ G(q, y, \beta_n) \tag{2.8}$$

where k_n is an adaptive step size, $D_\beta G$ is the Jacobian of G with respect to β, and $(D_\beta G|_\beta)^\dagger$ is the Moore-Penrose pseudo-inverse of $D_\beta G|_\beta$.

We symbolically calculate the Jacobian for the estimation parameters and numerically evaluate its value at runtime. β_0 is initialized by the result of the linear estimate or by a recent nonlinear estimate, if available. We adaptively select k_n to guarantee $\|G(q, y, \beta_n)\|$ monotonically decreases for successive n.

For any rotation matrix R, there exist two congruent Euler angle parameterizations such that $\delta = [\theta_1, \theta_2, \theta_3, p_1, p_2, p_3]$ and $\gamma = [\pi + \theta_1, \pi - \theta_2, \pi + \theta_3, p_1, p_2, p_3]$ are equivalent parameterizations for $[R\, p]$. When $\cos(\theta_2) \neq 0$ it is direct to check by symbolic computation that each iteration of (2.8) produces equivalent results for both parameterizations; that is

$$D_\beta G|_\delta = (D_\beta G|_\gamma)\,\mathrm{diag}(1, -1, 1, 1, 1, 1). \tag{2.9}$$

Thus, iterations of β_n in (2.8) step through equivalent rotations R_n regardless of the particular Euler parameterization.

The nonlinear algorithm outperforms the linear algorithm because it optimizes over only the necessary 6 parameters of the transformation. However, due to the nonlinearities there are many local minima. Hence, the algorithm is highly sensitive to initialization and thus only useful given a decent initialization from the linear algorithm.

Chapter 3

System Integration and Results

3.1 Hardware

As part of the BErkeley Aerial Robot (BEAR) project [1], our UAV testbed is a Yamaha R-50 helicopter (see Figure 1.1) on which we have mounted:

- *Navigation Computer*: Pentium 233MHz Ampro Littleboard running QNX realtime OS – responsible for low level flight control [9];
- *Inertial Measurement Unit*: NovAtel MillenRT2 GPS system (2cm accuracy) and Boeing DQI-NP INS/GPS integration system;
- *Vision Computer*: Pentium 233MHz Ampro Littleboard running Linux – responsible for grabbing images, vision algorithms, and camera control;
- *Camera:* Sony EVI-D30 Pan/Tilt/Zoom camera;
- *Frame Grabber*: Imagenation PXC200;
- *Wireless Ethernet*: WaveLAN.

The vision computer communicates with the navigation computer over a 115Kbps RS232 serial link. Currently, we only use that link to gather INS/GPS state information from the navigation system for "ground truth" comparison of state estimates with the vision sensor. Images at 320 × 240 pixels resoltion are captured by the frame grabber at 30 frames per second, asynchronous to program execution through memory-mapped IO and system signals. To reduce the computational cost of thresholding, we estimate the image histogram by considering one-seventh of the total image pixels. Furthermore, a multi-resolution approach is used in our segmentation algorithm to reduce processing time, effectively performing segmentation on a 160 × 120 pixel image. Then the edges of the interior squares of the landing target are calculated from the original 320 × 240 pixel image. The vision computer communicates with the PTZ camera over a 9.6Kbps RS232 serial link to send pan-tilt control commands and receive the current pan-tilt state.

3.2 Camera Control

Proper control of the pan/tilt camera can increase the range of motion of the UAV while keeping the landing target in the field of view of the camera-head. The goal of the desired camera control is simple: to pan and tilt as necessary to keep the target centered in the image. For such a scheme to work in the overall process, measurement of the pan/tilt state of the camera must be available for each image.

The PTZ camera we use has an internal controller which can be commanded to relative or absolute pan/tilt locations via a RS232 protocol. Here we describe how to compute the pan and tilt action necessary to center the target in the image. By convention, positive pan moves the target left in the image, and positive tilt moves the target down in the image. For our camera, the axes of rotation for pan and tilt coincide with its optical center. For a particular axis in the image, as in Figure 3.1, the angle between a point on the axis and the center of the axis is given by

$$\theta = h\left(p, d, f\right) = \operatorname{atan2}\left(p - \frac{d}{2}, \frac{d}{2}\cot\left(\frac{f}{2}\right)\right), \tag{3.1}$$

where f is the field of view of the axis in radians, d is the width of the axis in pixels, and p is the location in pixels of the target projected onto the axis.

Now, let the desired location of the target be at (x_0, y_0), the target be at (x_1, y_1), the

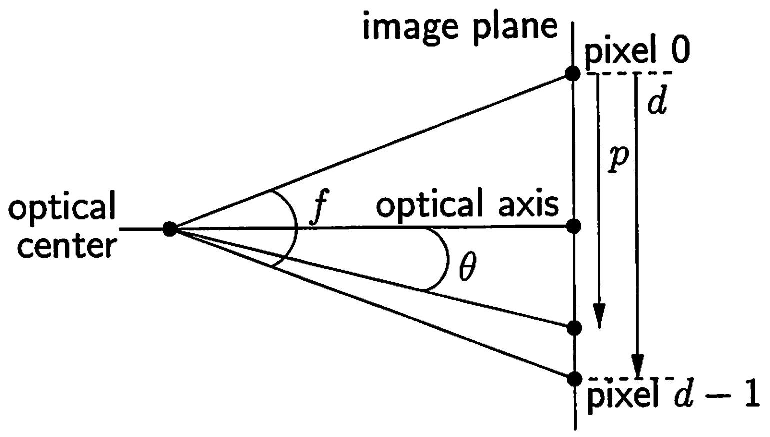

Figure 3.1: Geometry of pan/tilt with respect to the optical center and image plane. f is the field of view of the axis in radians, d is the width of the axis in pixels, p is the location in pixels of the target projected onto the axis, and θ is the rotation required to center the target.

image width and height be l_x and l_y, and the horizontal and vertical field of view be f_x and f_y. Then, using Equation (3.1), the amount of pan and tilt necessary to move the target to the desired location is given by

$$\theta_{pan} = h(x_1, l_x, f_x) - h(x_0, l_x, f_x) \tag{3.2}$$

$$\theta_{tilt} = h(y_1, l_y, f_y) - h(y_0, l_y, f_y), \tag{3.3}$$

where θ_{pan} and θ_{tilt} are relative to the current pan/tilt state. These pan/tilt commands are sent to the PTZ camera over a RS232 link in order to center the target in the image regardless of the motion of the UAV to which the camera is mounted.

3.3 Frame Transformations

The geometry of the coordinate frames and Euclidean motions involved in the vision-based pose estimation problem are shown in Figure 3.2. We label the coordinate frames as: (a) Landing target, (b) Landing pad, (c) Camera head, (d) Camera base, (e) UAV, (f) Inertial frame. The notation $g_{ba} \in SE(3)$ denotes the Euclidean motion (translation and rotation) of coordinate frame b with respect to frame a. Table 3.1 describes how each coordinate frame transformation is measured. Of particular note, the transformations g_{ef}

and g_{fb} are exclusively used in the "ground truth" comparison of the state estimates of the vision sensor and are never directly used by the vision algorithm.

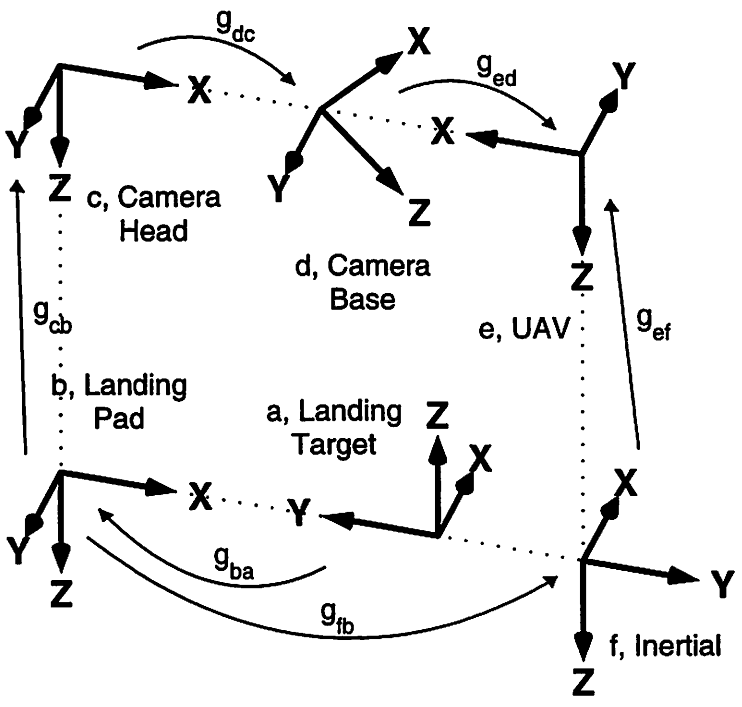

Figure 3.2: Geometry of the coordinate frames and Euclidean motions involved in the vision-based pose estimation problem.

	Description	Measurement
g_{ba}	landing pad wrt landing target frame	predefined
g_{cb}	camera head wrt landing pad	vision-based state estimate
g_{dc}	camera head wrt camera base	pan/tilt state from camera
g_{ed}	camera base wrt UAV	predefined
g_{ef}	UAV state wrt inertial frame	INS/GPS from UAV
g_{fb}	landing pad wrt inertial frame	predefined

Table 3.1: Description and measurement methods of coordinate frame transformations involved in the vision-base pose estimation problem from Figure 3.2.

3.4 Software

LAPACK (Linear Algebra PACKage) and BLAS (Basic Linear Algebra Subprograms) [4] are used to facilitate the coding of vision algorithms for standard matrix operations such as SVD, eigenvalue decomposition, and Gaussian elimination. Our remaining code uses a flexible object-oriented program structure written in C++. We have developed a "ground station" which monitors the status of flight experiments by communicating through wireless ethernet to the onboard vision computer and displaying the current state of the vision system through a Java-based GUI (shown in Figure 3.3).

In addition to the current vision-based state estimates, we bit-pack the thresholded 160×120 pixel binary image for display on the ground station. We find that this realtime view from the PTZ camera is invaluable for testing by not only giving feedback regarding the state of the helicopter, but by also showing the quality of thresholding algorithm and the pan/tilt control. In practice, we find that while the pan/tilt control performs as well as expected, there is still room for improvement in our selection of the threshold. We also display the current labeled feature point estimates to gauge the quality of the corner detection and labeling algorithms. In our experiments, we find that while our corner detection algorithm produces somewhat noisy estimates, our corner labeling algorithm is extremely robust.

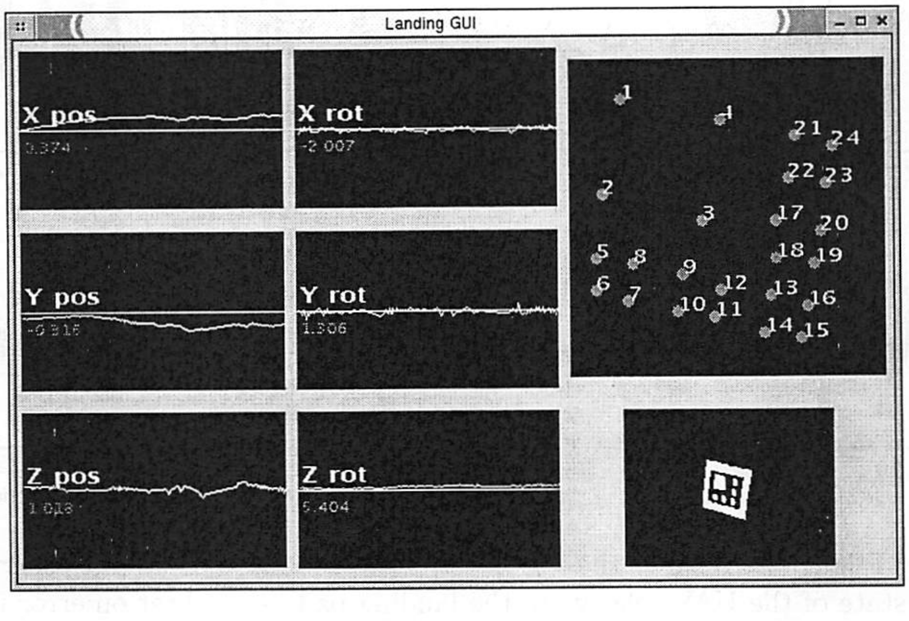

Figure 3.3: Ground-station Java-based visualization showing position and rotation estimates, extracted feature points and labels, and the binary image view from the camera.

3.5 Overall System Performance

Figure 3.4 shows the results of our flight test with the UAV, comparing the output of the vision-based state estimation algorithm with the INS/GPS measurements accurate to 2cm. All plots show the state of the UAV with respect to the landing pad. The vision estimates are more noisy, but otherwise follow the INS/GPS measurements. Also, errors in the internal and external camera calibration parameters marginally affects some of the estimates – the x-position and z-rotation, in particular. While logging a fraction of the original images in real-time, the Linux write caching algorithm caused noticeable, synchronized gaps in data in each plot.

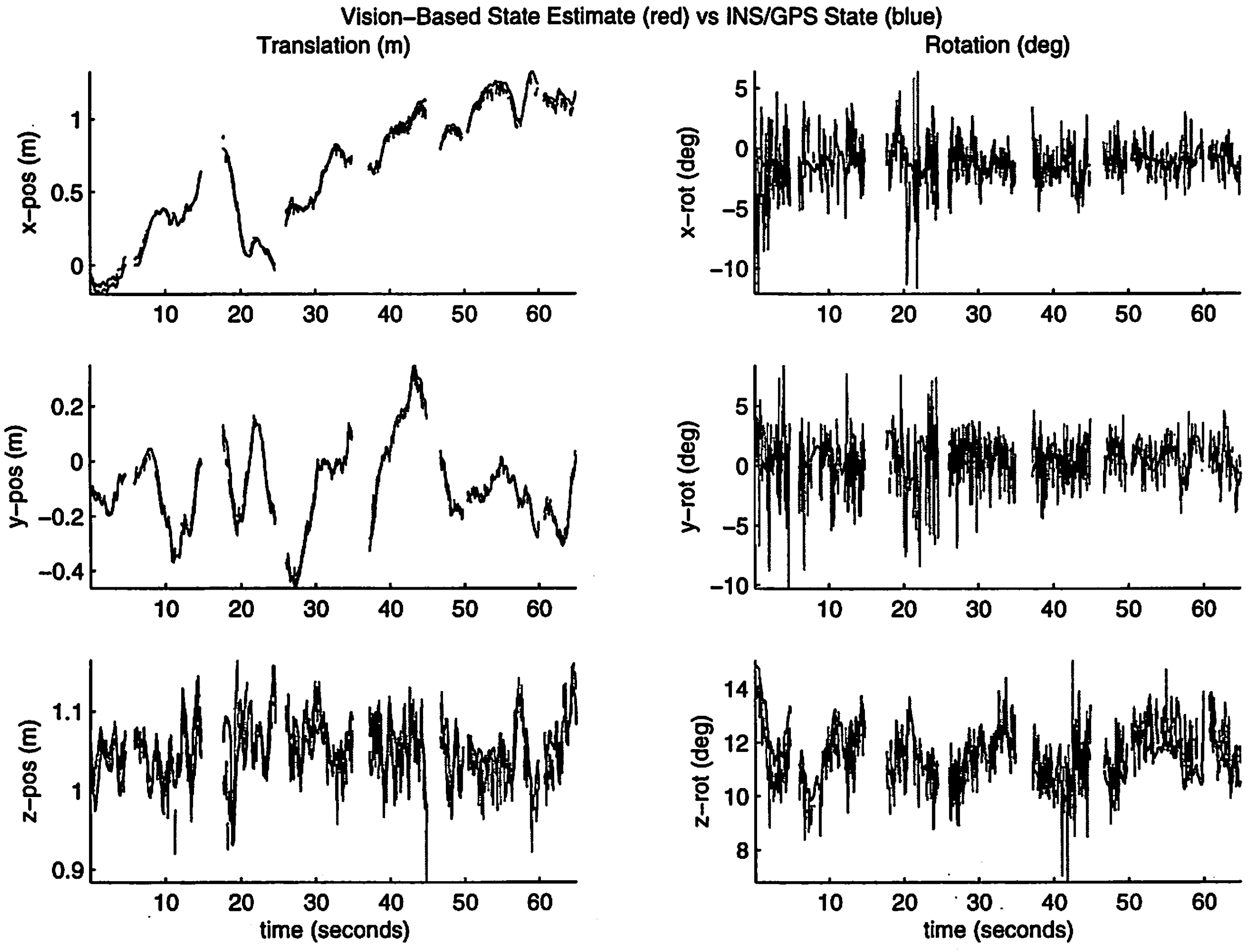

Figure 3.4: Flight test results: the vision estimates are more noisy but otherwise follow the INS/GPS state of the UAV relative to the landing pad. A calibration error is most notable in the estimates of x-position and z-rotation.

Figure 3.5 shows the root mean squared error of each estimated state parameter. The position estimates are within 5cm accuracy. The x- and y-position estimates should perform better than the z-position estimate. That is, the same amount of translation along the x- or y-axis causes more detectable change in the image than along the z-axis, the optical axis of the camera. However, this is not apparent in our plots due to the noted calibration error. The rotation estimates are all within 5 degrees accuracy. As expected, the z-rotation estimate significantly outperforms the other two. That is, the same amount of rotation about the z-axis causes more detectable change in the image than the other axes. Overall, the vision-based state estimates are accurate enough to be used in the closed-loop of a high-level controller for landing in a hierarchical flight vehicle management system, as described in [9].

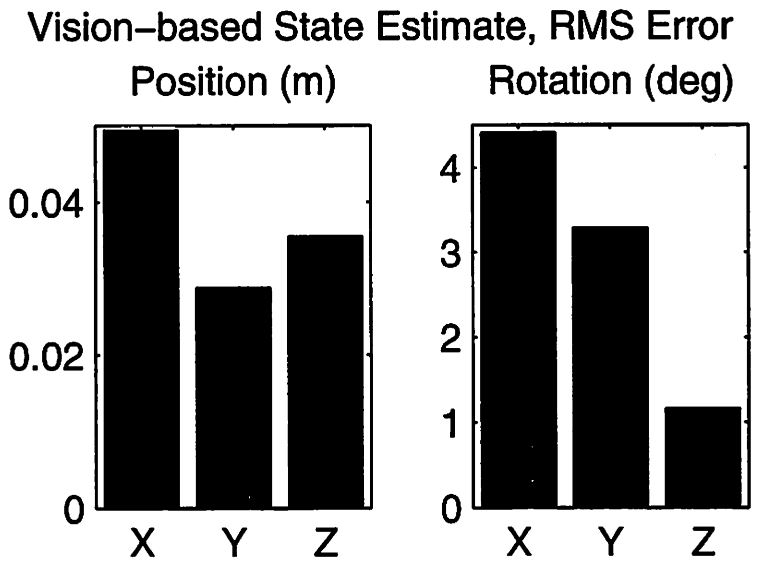

Figure 3.5: The RMS error of state estimates for each axis of translation and rotation. Position error is within 5cm. Rotation error is within 5 degrees.

Table 3.2 shows the computational cost of each stage of processing in our system, measured from actual program execution. In particular, we see that almost all of our computational time is spent on image processing. In turn, most of the image processing is spent on the segmentation algorithms for extracting the landing target. The obvious conclusion is that any further effort toward improving computational efficiency should be spent optimizing the image segmentation algorithms.

"Low Level Image Processing"	
Segmentation pass 2, edge detection (low res)	42%
Threshold, segmentation pass 1	30%
Image acquisition	8%
Corner detection (high res)	6%
Histogram estimation	4%
Feature labeling	<1%
Total	90%

Other Processing	
Nonlinear optimization	5%
Bit-encoding binary image for GUI	2%
Linear optimization	2%
Frame transformations	1%
Misc.	<1%
Total	10%

Table 3.2: Computational cost of each stage or processing, measured from actual program execution. Image processing is expensive, consumed by segmenting the landing target from the image. Refer to Figure 2.1.

Chapter 4

Conclusion and Future Work

In this paper, we have presented the design and implementation of a vision system to land a UAV. Our proposed vision system consists of off-the-shelf hardware and gives realtime estimates at 30Hz of the position and orientation of the UAV relative to a specifically designed landing pad. The estimates are accurate to within 5cm in each axis of translation and 5 degrees in each axis of rotation.

Some aspects of our system performed particularly well. Our feature labeling is computationally inexpensive and extremely robust to noise. The camera control algorithm performed well given the dynamic limits of the pan/tilt actuators of our camera. Given a good threshold for a grayscale image, our segmentation algorithm never failed to extract the landing target. With adequate initialization, our estimates from nonlinear optimization proved to be robust to noise. The vision-based state estimates are sufficiently accurate to allow our vision system to be placed in the hierarchical control loop for UAV landing.

In the future, we would like to use a more robust technique for selecting a grayscale threshold to reduce the occasionally undetected landing target. Also, our corner detector does not address the effects of pixel quantization and suffers somewhat as a result. We would also like to obtain corners to within sub-pixel accuracy by using additional constraints from the landing target design. These improvements can significantly decrease the overall error of the subsequent estimates.

For future research, we will place our vision system in the control loop for autonomous UAV landing onto a moving platform which simulates the motion of a ship deck, as shown in Figure 4.1.

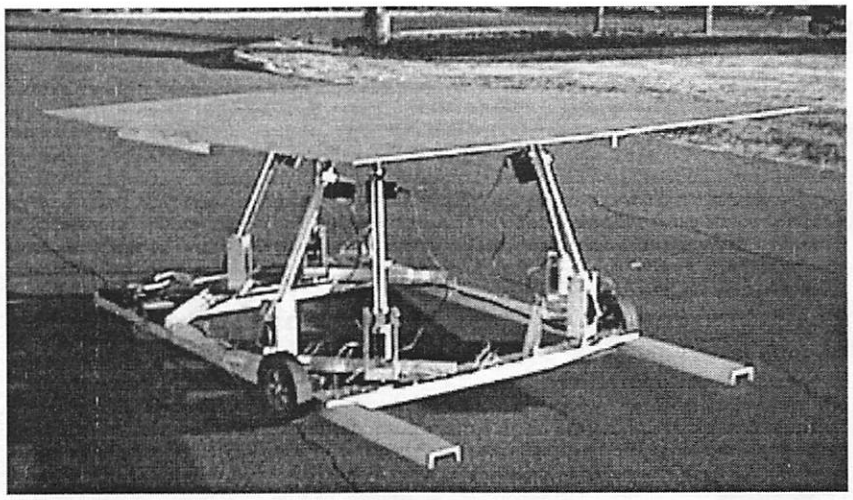

Figure 4.1: Programmable 6 DOF landing platform to simulate watercraft dynamics. To be used in future work.

Bibliography

[1] BErkeley Aerial Robot (BEAR) Project homepage. `http://robotics.eecs.berkeley.edu/bear`.

[2] M. Bosse, W.C. Karl, D. Castanon, and P. DeBitetto. A vision augmented navigation system. In *IEEE Conference on Intelligent Transportation Systems*, pages 1028–33, 1997.

[3] R. Gonzalez and R. Woods. *Digital Image Processing.* Addison-Wesley, 1992.

[4] LAPACK Homepage. `http://www.netlib.org/lapack`.

[5] I. Kaminer, A. M. Pascoal, W. Kang, and O. Yakimenko. Integrated vision/inertial navigation system design using nonlinear filtering. *To Appear: IEEE Transactions on Aerospace and Electronics.*

[6] D. Oberkampf, D.F. DeMenthon, and L.S. Davis. Iterative pose estimation using coplanar feature points. *Computer Vision and Image Understanding*, 63(3):495–511, May 1996.

[7] A. Petruszka and A. Stentz. Stereo vision automatic landing of VTOL UAVS. In *Proceedings of Assoc. Unmanned Vehicle Syst. Int.*, pages 245–63, 1996.

[8] F.R. Schell and E.D. Dickmanns. Autonomous landing of airplanes by dynamic machine vision. *Machine Vision and Applications*, 7:127–134, 1994.

[9] D.H. Shim, H.J. Kim, and S. Sastry. Hierarchical control system synthesis for rotorcraft-based unmanned aerial vehicles. In *Proceedings of AIAA Conference on Guidance, Navigation and Control*, Denver, 2000.

[10] J. Weng, T.S. Huang, and N. Ahuja. *Motion and Structure from Image Sequences.* Springer-Verlag, 1993.

[11] S. Werner, S. Furst, D. Dickmanns, and E.D. Dickmanns. A vision-based multi-sensor machine perception system for autonomous aircraft landing approach. In *Proceedings of the SPIE - The International Society for Optical Engineering*, volume 2736, pages 54–63, Orlando, FL, USA, 1996.

[12] Z.F. Yang and W.H. Tsai. Using parallel line information for vision-based landmark location estimation and an application to automatic helicopter landing. *Robotics and Computer-Integrated Manufacturing*, 14(4):297–306, 1998.

Appendix A

Code Overview

Here we describe the code for the integrated vision landing system. Because work remains for final integration of the landing system, the following details will help future research extend the current system. And since the code is fully functional, there is value in incrementally improving an existing code base instead of starting from scratch.

This Appendix describes the files corresponding to each component in Figure 2.1, then goes on to describe the files for the front-end, supporting libraries, ground visualization, and test results. And, because the intent is to enable future research, the files used for development and testing are described in addition to those used for the current implementation.

A.1 Image Acquisition

For the image segmentation and corner detection, custom image processing routines are used. So, direct access to the frame grabber must be available. The supported hardware is the Matrox Meteor MGR, a PCI card used for desktop development, and the ImageNation PXC200, a nearly equivalent PC104+ card used on-board the helicopter. Both cards are based on the Brooktree Bt848 chipset. Appropriate header files were taken from the "Bt8xx Frame Grabber Drivers for Linux" – the original archive is included as `bt848-1.1-RH-6.2.tgz`. Naturally, the appropriate Linux kernel module must be present for access to the hardware.

All relevant files are in `corner/bt848/`.

A.1.1 Development

Some files in the `corner/bt848/` directory were used purely for development. These files are stand-alone and test preliminary access to the frame grabber. For example, see `capture_pxc_mmap_single.cc`.

A.1.2 Implementation

This section describes image acquisition the files used in the current vision system.

CaptureDevice.h

`CaptureDevice.h` defines an abstract interface for capture devices. The intent is to allow code using a capture device to not depend on a particular frame grabber. For instance, a saved image sequence be presented as live data without modification of the dependent code. A capture device provides the following public members:

```
void start_capture();
void stop_capture();
bool is_capturing();

bool is_new_image_avail();
unsigned char* get_image();
int get_image_size();
int get_rows();
int get_cols();

int get_frame_count();
```

CapturePXC200.{cc,h}

`CapturePXC200.{cc,h}` implements a capture device for the PXC200 and Meteor MGR. It supports grayscale image acquisition from YUV packed or YUV planar modes. It also support RGB24, although this is currently unused because the corner detection routine only uses binary images derived from grayscale images. The class provides the following members in addition to the CaptureDevice interface:

```
CapturePXC200();
CapturePXC200(const char* dev, int rows, int cols, mode_flags flags);

void open_device(const char* dev, int rows, int cols, mode_flags flags);
void close_device();
```

where `mode_flags` is either `Y8_FROM_YUV_PACKED`, `Y8_FROM_YUV_PLANAR`, or `RGB24`.

Note that for the current system, requesting a video mode with more than 240 rows is undesirable. For video modes with more than 240 pixels, the frame grabber cannot be initialized to use only even or odd scan lines. Because NTSC is interlaced, if all scan lines are used, the 1/60 second delay between even and odd scan lines will cause noticable ghosting when the camera encounters vibration.

`CapturePXC200.cc` depends on `ioctl_bt848.h` and `ioctl_meteor.h` for interfacing with the Linux Bt848 module.

A.1.3 Testing

`test_CapturePXC200.cc` exercises CapturePXC200 by opening the frame grabber, grabbing a number of frames, reporting the frames per second grabbed, and dumping the final frame to a PGM image file.

A.2 Histogram and Threshold

The histogram and thresholding routines provide methods to convert a grayscale image from the frame grabber to a binary image for the segmentation and corner detection routines.

The base directory for histogram and threshold is `research/corner/`.

A.2.1 Development

In the `matlab/` subdirectory, the thresholding algorithm is one component in the development of the corner detector. In the `src/` directory there remain some files left over from exploring now unused histogram and threshold techniques.

graw2bw.m

`gray2bw.m` is a simple thresholding algorithm that asserts at least fifty percent of the image pixels must be black. In practice, this algorithm was found to be less robust than other techniques. However, it is sufficient for the simulated images used in development.

calc_moving_sum.h

`calc_moving_sum.h` is a templated function that calculates a simple moving-window sum for a list of numeric data. In particular, it was used to low-pass filter histogram data before performing a gradient descent.

threshold.h

`threshold.h` is a very simple thresholding algorithm. It performs a block of read operations then a block of write operations, attempting to keep the CPU pipeline full by separating read-write dependencies. Regardless, thresholding is currently implemented as a function of the segmentation routine. In this case, exploiting locality outperforms exploiting the pipeline.

threshold_mmx.cc

threshold_mmx.cc is an Intel MMX-enhanced version of the thresholding routine. It depends on mmx_load8byte.h. While faster than the non-MMX version, thresholding in the segmentation routine still marginally outperforms it.

A.2.2 Implementation

This section describes the histogram and threshold files used in the current vision system.

calc_histogram.h

calc_histogram.h provides three simple histogram calculation and estimation routines. calc_histogram is a straight-forward calculation of the exact histogram. calc_histogram_urolled also calculates the exact histogram and reduces computation time by loading 32 bits at a time instead of 8 bits. calc_histogram_nstep, used in the current vision system, estimates the histogram by evaluating every nth pixel.

get_gray2bw_threshold.h

get_gray2bw_threshold.h provides three methods for calculating the histogram of an image. get_gray2bw_threshold, the original thresholding algorithm, calculates the local minimum in a smoothed histogram above a given minimum graylevel. get_gray2bw_threshold_npoly fits a 6th-order polynomial to the histogram and picks the most appropriate local minimum. get_gray2bw_threshold_fixed, used in the current vision system, calculates the graylevel as a given percentage between the minimum and maximum graylevels present in the histogram.

A.2.3 Testing

For debugging the histogram and threshold routines, `dumphist.cc` takes the name of a PGM file on the command-line and dumps the relevant information to standard output.

A.3 Segmentation

The segmentation routines take a binary image as input and uses connected component labeling to locate the landing pad in the image and the components within the landing pad.

The base directory for segmentation is `research/corner/`.

A.3.1 Development

The segmentation routines were developed using MATLAB and can be found in the subdirectory `matlab/`.

connected_components.m

`connected_components.m` is a simple implementation of connected component labeling using 4-neighbors. It takes a 2-dimensional numeric matrix as input, and produces a 2-dimensional numeric matrix with each connected region uniquely numbered. It also optionally outputs a 2xN matrix describing the map from group number to input value. This implementation is not very fast in MATLAB because it depends heavily on for-loops iterating over scalar data.

connected_components_c.c

`connected_components_c.c` is a MEX-file that when compiled mimics the behavior of `connected_components.m` but is much faster. This is the base implementation used to develop the variations used in the current vision system.

locate_landing_pad2.m

`locate_landing_pad2.m` uses the connected component labeling routine to identify the landing pad and its internal components in a binary image, then continues to find the corners in the landing pad.

test_landing_pad.m

`test_landing_pad.m` rasterizes a landing pad given rotation and translation parameters then calls `locate_landing_pad2.m` to find the corners of the landing pad in the rasterized image.

A.3.2 Implemenatation

This section describes the segmentation files used in the current vision system. The files are found in the subdirectory `src/`.

connected_components.h

`connected_components.h` is a C++, templated incarnation of the connected components MEX-file. This file was used as a springboard to develop the two variations currently used.

connected_components_op.h

connected_components_op.h extends connected_components.h by first applying a functor object to the input data. In particular, this is used to first threshold the input data. This exploits data locality can gives a legitimate speed boost over thresholding separately. Stepsize can also be specified as a template parameter, allowing for on-the-fly down-sampling of the input image.

connected_components_op_edge.h

connected_components_op_edge.h further extends connected_components_op.h by also calculating edges in the input image. Again, this functionality is included because it exploits data locality and gives a legitimate speed boost over performing edge-detection separately.

connected_components_op_inner_loop.cc_include

This file is an extraction of a common inner loop from connected_components.h in an attempt to improve the readability of the two derived files.

encode_bw2bits.h

encode_bw2bits.h provides routines to encode and decode a byte-per-pixel binary image to a bit-per-pixel binary image. This provides 8-factor compression, redicing data when transmitting the binary image to the ground station as well as saving binary images to a log file.

find_features.cc

find_features.cc uses the connected components routines to locate the landing pad in a grayscale image, then locates the edges and corners in the landing pad, and finally uniquely labels each corner.

A.3.3 Testing

While testing the algorithm, the ability to read and write grayscale image files is invaluable. This functionality is provided by `read_image.{cc,h}` and `write_image.{cc,h}` which handle PGM files, a simple grayscale image file format.

A.4 Corner Detection and Labeling

The corner detection algorithm uses a custom algorithm to locate the four corners of a 4-sided polygon in a binary image. It first performs edge detection to find the necessary edges, then it uses convexity constraints to find the four extremities of a 4-sided, convex region. The labeling algorithm then asserts a clockwise ordering relationship between corners.

The base directory for corner detection and labeling is `research/corner/`.

A.4.1 Development

The corner detection and labeling routines were developed using MATLAB and can be found in the subdirectory `matlab/`.

calc_corners_from_perimeter.m

`calc_corners_from_perimeter.m` takes a number of perimeter pixels as input and calculates the four corner pixels as output. The input may be the full region, but the calculation over the perimeter pixels is faster and equivalent.

get_group_perimeter.m

Given an image that has probably been processed by the connected component labeling routine and a group number, `get_group_perimeter.m` calculates the edges pixels for the given group.

locate_landing_pad2.m

After using the connected component labeling algorithm to identify the landing pad, `locate_landing_pad2.m` finds and labels its corners using `get_group_perimeter.m` and `calc_corners_from_perimeter.m`.

A.4.2 Implementation

This section describes the corner detection and labeling files used in the current vision system. The files are found in the subdirectory `src/`.

LocateLandingPad.h

`LocateLandingPad.h` is the main interface to the corner detector. It allocates temporary work arrays so that major memory allocation and deallocation does not take place with each call. There are two members functions of primary interest: `set_image_size` prepares the temporary work arrays, and `find_features` takes a pointer to a grayscale image as input and calculates the features for output. Naturally, the actual size of the image must be congruent with the `cols` and `rows` parameters. The image data given to `find_features` must be stored row-first.

```
LocateLandingPad();
LocateLandingPad( int cols, int rows );

void set_image_size( int cols, int rows );
void find_features( image_type* image, double* features );
```

calc_corners_from_perimeter.h

calc_corners_from_perimeter.h is a templated function that takes a set of perimeter pixels as input and calculates the four corner pixels as output. It uses the assumption that the perimeter was taken from a convex region.

calc_corr.h

Given the 24 corner pixels of the landing pad, calc_corr.h reorganizes the corners to a consistent ordering regardless of the rotation, translation, and projection of the landing pad in the image. Corners from the same region should be adjacent in the input corner vector. On output, the corners are ordered according to Figure 2.2(b).

calc_line.h

Given two points, calc_line.h calculates the equation of the line between them. This function is used by calc_corners_from_perimeter.h.

convert_index_to_xy.h

Given a sequence of flat indices into a matrix and the number of rows in the matrix, convert_index_to_xy.h produces an the corresponding x,y indices. This function is used by find_features.cc.

convert_scaled_perim_to_unscaled.h

Segmentation and edge detection occurs on a down-sampled image. convert_scaled_perim_to_unscaled.h uses the unscaled image and down-sampled edges to calculate the edges in the unscaled image. The resulting edge pixels are used in the corner detection algorithm. The result is that corners are effectively calculated at full image resolution.

corr_ordered_angles.h

Given six centers of gravity and a reference index, `corr_ordered_angles.h` calculates the angle between the reference CoG and the remaining 5 CoG's. This function is used by `calc_corr.h`. It calculates all possible angles between the center of gravities of each square in the landing target. That information is then used to determine the identity of each square.

find_features.cc

`find_features.cc` is the work-horse of the corner detector. It brings together the histogram, threshold, segmentation, edge detection, corner detection, and correspondence routines. Of particular note, it adjusts the down-sampling and histogram estimation parameters so that the segmentation and histogram routines on 160×120, 320×240, and 640×480 images execute in approximately the same amount of time.

get_furthest_ind_from.h

`get_furthest_ind_from.h` is a small function supporting `calc_corners_from_perimeter.h` that determines the point furthest from a reference point in a given set of points. Note, its input and output are indices into the set of points, rather than the point itself.

order_4corners.h

`order_4corners.h` is a small function supporting `calc_corr.h` that takes 4 counter-clockwise corners of a square in the landing target and performs the final correspondence by considering the angle between those corners and the center of gravity of another, particular square in the landing target. This routine again asserts that counter-clockwise ordering is invariant under rotation, translation, and projection.

A.4.3 Testing

These routines test the development corner detection and labeling algorithms in the `matlab/` directory.

get_corner_grid.m

`get_corner_grid.m` is a simple support function that produces an evenly spaced $M \times N$ grid of boxes with parameterized spacing and width of the boxes.

get_landing_pad.m

`get_landing_pad.m` produces the corners of a landing target necessary for rendering. A few different configurations are supported, and the default is the one currently used in the vision system.

plot_landing_pad.m

`plot_landing_pad.m` plots the landing pad generated with `get_landing_pad.m` in a figure window. It depends on `plot_corner_grid.m`.

render_figure.m

`render_figure.m` manipulates properties of the figure window to rasterize a figure to a desired resolution. This function allows the thresholding, segmentation, edge detection, corner detection, and corner labeling routes to operate on a simulated set of image pixels at any desired resolution (up to the current screen resolution, a constraint imposed by MATLAB).

test_landing_pad.m

`test_landing_pad.m` is a script that selects particular parameters to test the corner detector.

A.5 Linear Estimation

The linear estimation directly solves for the motion parameters by relating the features in the current image to a known feature configuration.

The base directory for linear estimation is `research/landing/motion/coplanar4/`.

A.5.1 Development

Although the linear algorithm is straight forward, the MATLAB file `coplanar_getRp.m` is used to verify the structure of the algorithm is correct. The algorithm takes as input a known configuration of the feature points, where the z-coordinate is assumed to be and must be 0. Corresponding feature points in the image plane are also given as input. The routine produces a matrix in $SE(3)$ representing the rotation and translation and optionally outputs the estimated error.

A.5.2 Implementation

`coplanar_getRp.{cc,h}` directly implements the MATLAB version in C++. However, the zero configuration currently must supply the z-coordinate of the features in the known configuration, although they all must be 0. Also, the estimated error is not a return value.

A.5.3 Testing

test_coplanar_getRp.m and test_coplanar_getRp.cc exercise their MATLAB and C++ counterparts. They allow a number of rotations and translations to be stepped through, verifying the expected output with the actual output.

A.6 Nonlinear Estimation

The nonlinear estimation iteratively solves for the motion parameters by relating the features in the current image to a known feature configuration.

Development of the nonlinear estimation routines is inexplicably split between research/landing/nonlinest/ and research/landing/motion/nonlinest/. The most recent development version and current implementation is in motion/nonlinest/, and some older but somewhat useful files are in landing/nonlinest/.

A.6.1 Development

In landing/nonlinest/, the Jacobian of for the gradient function was calculated symbolically in MATLAB. A log of the derivation is in nonlinest.txt. The final result was cut and paste and modified into nonlinest.m.

A more current development version of nonlinest.m is in motion/nonlinest/. This algorithm iterates until the error becomes small, the step size becomes small, or a maximum number of iterations is reached. The step size is adaptively selected to guarantee the error monotonically decreases.

A.6.2 Implementation

nonlinest.{cc,h} is the current implementation of the nonlinear algorithm in motion/nonlinest. It is a direct translation of the MATLAB version to C++.

A.6.3 Testing

test_nonlinest.m and test_nonlinest.cc in motion/nonlinest exercise their MATLAB and C++ counterparts. In particular, they step through a number of rotation and translations, verifying the actual output matches the expected output.

A.7 Camera Communication and Control

A class was specifically written for communication with the Sony EVI-D30 PTZ camera. Routines were developed to command the camera to pan and tilt any given pixel to any other pixel location. Also, the routines support commanding the camera to zoom so that a particular set of pixels maximally fill a bounding box.

The base directory for camera communication and control is research/camera/.

A.7.1 Development

Development for the camera communication mostly translated to viable implementation. However, there were a couple of failed experiments, namely SaphiraIO.{cc,h} and ThreadedIO.{cc,h}. SaphiraIO attempted to communicate with the camera through Saphira and was abandoned. ThreadedIO attempted threading at low-level communication and was later replaced by threading at a higher level.

The camera supports multiple threads of communication, but conveys minimal context information throughout a communication sequence. Also, some particular response packets from the camera are occasionally garbled. These two artifacts of the camera caused a couple of rewrites before a bug-free communication class was produced.

A.7.2 Implemenatation

This section describes the camera communication and control files used in the current vision system.

BasicIO.h

`BasicIO.h` defines an abstract interface for a character device. This abstracts the primary camera communication class from from the particular communication device.

FileDescIO.{cc,h}

`FileDescIO.{cc,h}` implements BasicIO for a device accessible through file descriptors, such as a serial device.

PTZCamera.{cc,h}

`PTZCamera.{cc,h}` provides a means to communicate with the Sony PTZ camera. Three members of PTZCamera are useful in particular:

```
PTZCamera(BasicIO* io);

void setIO(BasicIO* io);
void putOutputPacket(const string& s);
void processCommandQueue(bool bBlock=false);
```

`setIO` assigns a `BasicIO` class providing input and output services to for camera. `putOutputPacket` places a string on the camera's command queue. `processCommandQueue` processes any pending input from the camera, the sends the next packet on the camera's command queue if it is safe to do so.

The namespace `PTZPacket` provides functions for encoding commands for the camera and decoding responses. address_set and ifclear must be sent to the camera for initialization. Other supported commands include pan, tilt, and zoom. See the header file `PTZCamera.h` for further details.

Furthermore, the class `PTZCamera` is aware of pan, tilt, and zoom commands and responses from the camera. As a result, it maintains a record of the most recently received pan, tilt, and zoom measurement responses. See the header file `PTZCamera.h` for further details.

CameraUtils.{cc,h}

`CameraUtils.{cc,h}` resides in `research/landing/motion/src/` and defines a namespace for camera control. The provided functions calculate the camera calibration matrix, automatically zoom a set of pixels to maximally fill a bounding box, and pan any given pixel to any other pixel location.

A.7.3 Testing

Two files in `research/camera/` exercise the camera communication class. `test_camera.cc` commands the camera to pan and tilt to its four extremes, in sequence, all the while printing out current pan, tilt, and zoom measurements. `test_threads2.cc` similarly exercises the camera, except that `processCommandQueue` is placed in a separate thread, allowing for unmanaged, asynchronous camera control.

A.8 Frame Transformations

The frame transformations described in Figure 3.2 cannot be slighted. Because, the overall evaluation of the algorithm depends on properly calibrated data. The current implementation of the frame transformations are found in `MotionEstimation.{cc,h}` in `research/landing/motion/src/`. It was found that a move to homogeneous coordinates was invaluable in simplifying the numerous coordinate transformations.

A.9 Navigation Computer Communication

Communication with the navigation computer occurs through an RS232 serial port. Data is transmitted using the DQI packet structure. The communication class is built on ArbFStream.

The base directory for navigation computer communication is `research/landing/motion/chopperio/`.

A.9.1 Implementation

`ChopperIO.h` provides communication with the helicopter. It depends on `S_DqiIO` and `I_DqiIOListener` provided by ArbFStream. Upon request, the navigation computer provides a structure containing the following fields: time, x-position, y-position, z-position, roll, pitch, and yaw.

A.9.2 Testing

`test_chopperio.cc` exercises the `ChopperIO` class. Upon pressing a key, a request for data is sent to the navigation computer. The response is printed to standard output.

A.10 Front-end

The front-end ties the diverse algorithms of the vision system together: initializing the frame grabber, detecting the corners in the image, running the linear and nonlinear algorithms, applying the frame transformations, controlling the camera, gathering log data from the navigation computer, and providing data to the ground station. It also provides logging the state of the vision system to a file and later replaying the log as if it were live data.

The base directory for the front-end is `research/landing/motion/src/`.

A.10.1 Development

Before developing the custom corner detector, we gathered image features using color tracking with the ActivMedia Color Tracking System (ACTS). We found that color features are not robust in a natural, outdoor environment. However, color tracking is still available in the files `ColorFeatures.{cc,h}` which implement a `FeatureInterface` (described below) for the color tracker.

A.10.2 Implementation

This section describes the front-end files used in the current vision system.

FeatureInterface.h

`FeatureInterface` defines an abstract class for providing features and controlling the camera given those features. A class implementing `FeatureInterface` provides the following members:

```
typedef double feature_type;
bool is_update_avail();
bool have_features();
vector<feature_type> get_features();
vector<feature_type> get_zero_config();

bool can_control_ptz_camera();
void control_ptz_camera(PTZCamera& cam, int image_rows, int image_cols);
```

CornerFeatures.{cc,h}

`CornerFeatures.{cc,h}` implements `FeatureInterface` for the corner detector. It uses a `CaptureDevice` for accessing images probably supplied by a frame grabber.

motion.cc

`motion.cc` provides command-line access to the front-end. It integrates the diverse functionality of the vision system into a single tool.

A.10.3 Testing

For testing, a series of grayscale images can be processed as if they were live data from a frame grabber. This functionality is provided by `TestfileFeatures.{cc,h}` and exposed by `motion.cc`.

A.11 Supporting Libraries

Through the course of development of the vision system, some libraries were necessary for basic functionality and infrastructure. Some were specifically written for the vision project, while others were publicly available on the Internet.

A.11.1 ArbFStream

ArbFStream is a library written for the project meant to implement arbitrary file streams. It was initially developed early on in the project to manage multiple input and output streams in a single thread. In all honesty, it should be rewritten to provide a cleaner, more lightweight interface. However, one of its current, primary attractions is that it allows specification of a variety of IO devices from a text string. Here is a excerpt from the `ArbFStream.h` header file describing the supported devices:

```
. Standard IO  : stdio
. File IO      : files,[input file],[output file]
. Serial IO    : serial,[device],[bps]
. TCP Listener : tcplisten,[port]
. TCP Client   : tcp,[address/name],[port]
```

In practice, string specification of devices is invaluable for simple command-line configuration.

The base directory for ArbFStream is `research/ArbFStream/`.

base/

The `base/` directory provides the basic functionality for ArbFStream.

ArbFStream.{cc,h} `ArbFStream.{cc,h}` provides the string initialization of the file streams. After opening a device, ArbFstream provides access through iostreams and file descriptors:

```
istream* is();
ostream* os();
int getInputFD() const;
int getOutputFD() const;
```

FDSelect.{cc,h} `FDSelect.{cc,h}` can wait indefinitely or for a specified period of time for read, write, or exception events for particular file descriptors.

NonBlockingStdin.{cc,h} `NonBlockingStdin.{cc,h}` provides non-blocking access to standard input. In Unix, a user must usually hit `ENTER` before the entered text becomes visible on stdin.

S_CharIO.{cc,h} `S_CharIO.{cc,h}` and `I_CharIOListener.h` is ArbFStream's solution to manage multiple IO streams in a single thread. A S_CharIO is constructed from an existing ArbFStream. Characters are then distributed to clients through the `I_CharIOListener` interface.

SerialStream.{cc,h} `SerialStream.{cc,h}` is used to create a serial stream. This class is used by ArbFStream when a serial stream is requested.

TCPStream.{cc,h} `TCPStream.{cc,h}` is used to create either a TCP listener or a TCP client. This class is used by ArbFStream when a TCP stream is requested.

services/

The `services/` directory extends the basic functionality of ArbFStream to provide interface to common communication protocols used in the project.

S_DqiIO.{cc,h} `S_DqiIO.{cc,h}` and `I_DqiIOListener.h` manage communication for the DQI protocol, used in particular for communication between the vision computer and the navigation computer. `I_DqiIOListener.h` emulates the `I_CharIOListener` interface for DQI packets.

S_GpsIO.{cc,h} `S_GpsIO.{cc,h}` and `I_GpsIOListener.h` manage communication with a GPS device. `I_GpsIOListener.h` emulates the `I_CharIOListener` interface for GPS packets.

A.11.2 AFSBridge

`AFSBridge` is a simple application built using ArbFStream that ties two streams together. This is useful in debugging, for instance, for forwarding a serial stream over a TCP connection. The base directory for AFSBridge is `research/AFSBridge`.

A.11.3 include/

The `include/` directory provides common files that are independent of the application that uses them.

c_lapack.h

LAPACK is a publicly available Fortran library used in this project for numeric calculation of SVD's, eigenvalues, matrix multiplication, matrix inversion, and so on. `c_lapack.h` provides a suitable C/C++ interface for calling the particular LAPACK functions used in the vision project. Similar LAPACK functions with different data types are overloaded to have the same calling name for C++.

tnt/

The Templated Numeric Toolkit (TNT), publicly available on the Internet, is used to manage matrices throughout the project. In retrospect, a much more suitable library exists for manipulating multidimensional numeric matrices, Blitz++.

tnt_css_fmat_ops.h

`tnt_css_fmat_ops.h` provides a number of matrix manipulation routines based on LAPACK. The primary data type used by the routines is TNT's `FortranMatrix`. Routines supplied by this header include `svd`, `eig`, `matsolve`, `matmul`, `null`, `det`, `rank`, `norm`, `inv`, `sum`, `zeros`, `ones`, and `eye`. The header also provides routines for producing rotation matrices and handling homogeneous matrices.

indexed_function.h

`indexed_function.h` allows C++ Standard Template Library (STL) functions to operate on the indices of a sequence rather than the sequence itself. This allows, for instance, a sequence of indices to be sorted according to the data they reference.

matlab/

The `matlab/` directory provides some commonly used MATLAB functions.

r2euler.m `r2euler.m` calculates the ZYX-Euler parameters from a given rotation matrix. This function was used as the basis for the equivalent C++ function used in the project.

rotxyz.m `rotxyz.m` calculates a rotation matrix given ZYX-Euler rotation parameters.

rtp.m `rtp.m` rotates, translates, and projects a set of world coordinates according to the given rotation and translation parameters.

unitize.m `unitize.m` transforms the columns of a matrix into unit-vectors.

wedge.m wedge.m implements the wedge operation, transforming a 3-vector into a skew-symmetric matrix.

r2euler.h

r2euler.h is a C++ implementation of matlab/r2euler.m, calculating ZYX-Euler parameters from a rotation matrix. It is used by the homogeneous matrix manipulation routines in tnt_css_fmat_ops.h.

A.12 Ground Visualization

Ground visualization is invaluable for identifying the performance of the system in real-time while the vision system is running. We implemented a graphical user interface in Java. It receives data from the vision computer on the helicopter through TCP commuication.

The base path for the ground visualization GUI is research/landing/gui/.

A.12.1 LandingGUI.java

LandingGUI.java implements four classes: LandingGUI, BinaryImagePlot, CoordinatedPointPlot, and TimePlot. LandingGUI is the main interface to the graphical display. It manages the TCP streams and objects displaying the data. It also provides a member function to replay a log file. BinaryImagePlot decodes bit-encoded binary image data and displays it in a small window. CoordinatedPointPlot scales and displays with labels the detected feature points in the image. TimePlot manages a running graph of time-series data, used to plot the rotation and translation estimates.

A.13 Results

We saved the results from a few flight tests. However, only one in particular produced decent, usable data. In addition to the data itself, a number of MATLAB functions were developed to aid in the analysis the flight data.

The base path for the results is `research/landing/motion/flight/`.

A.13.1 trial_{04,05,06}/

`trial_{04,05,06}/` contain data from three flight tests. `trial_06` in particular provided data suitable for analysis. Within each directory, there is a file named `flight.log`. This contains a log of data saved by the vision program while in flight. There is also a set of files that match the pattern `heligray?????.pgm`. These correspond to a series of grayscale images resulting from saving every 6th frame to an image file. And, the files matching `helihist?????.txt` are post-processed histogram results of each logged PGM image.

A.13.2 MATLAB Helpers

A number of MATLAB functions were developed to aid in the analysis of the flight data.

calc_dc_offset.m

`calc_dc_offset.m` estimates the DC offset necessary for each estimated parameter (rotation and translation) to zero-mean the vision estimates with the navigation computer measurements. This is useful in evaluating calibration errors.

plot_flight_log.m

`plot_flight_log.m` plots the contents of a flight log including the six vision estimates against the six navigation computer measurements. As input, it takes a structure produced by `read_flight_log.m`.

plot_hist.m

`plot_hist.m` plots the contents of a `helihist?????.txt` histogram analysis file. This is useful for evaluating the performance of the thresholding algorithm.

plot_hist_anim.m

`plot_hist_anim.m` animates of series of `helihist?????.txt` histogram analysis file. This is useful for analyzing the performance of the thresholding algorithm on a sequence of data.

read_flight_log.m

`read_flight_log.m` loads the contents of a `flight.log` file. The data is then readily available for plotting and analysis. This routine is very slow - `read_flight_relog.m` should be used if possible. If not, it is advised that the output is immediately saved to a MATLAB data file for quick reloads.

read_flight_relog.m

`read_flight_relog.m` loads a log file in the new format, which allows MATLAB to read the data significantly faster at the expense of not having the binary image data available. If possible, use this function over `read_flight_log.m`.