Brigham Young University

BYU ScholarsArchive

Faculty Publications

2011-06-01

BYU Indoor Flight System Circa June 2011

John C. Macdonald

Robert Leishman

Follow this and additional works at: https://scholarsarchive.byu.edu/facpub

Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation

Macdonald, John C. and Leishman, Robert, "BYU Indoor Flight System Circa June 2011" (2011). *Faculty Publications*. 1301.
https://scholarsarchive.byu.edu/facpub/1301

This Report is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Faculty Publications by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.

Indoor Navigation Baseline System Report - June 2011

John C. Macdonald Jr. and Robert Leishman

Abstract—**This report documents the theory, implementation, and performance of the baseline indoor navigation system as of June 2011. This date was chosen for a report because it marks the first time the HexaKopter flew without any direct input from the motion capture system. There are many improvements yet to be made in the performance of the system at this time, but it flies. Capturing our understanding of our system as it currently stands by means of this report should provide the basis for those future improvements.**

I. Introduction

Autonomous indoor flight in an unknown environment presents several challenges. An appropriate vehicle must be small and agile enough to maneuver in confined spaces. Due to its size, such a vehicle will have limited payload capacity for sensing and computing. The vehicle's agility also poses a problem: it will require an accurate navigation solution to enable autonomous control. However, flying indoors without prior knowledge of the environment prohibits reliance on GPS or prior maps to aid the navigation solution. This leaves inertial dead reckoning and simultaneous localization and mapping (SLAM) as the only two options for solving the navigation problem. Both of these approaches will be problematic. Inertial guidance will suffer from the low quality inertial measurement unit (IMU) the vehicle will be able to carry, and SLAM will be impacted by the limited processing power available onboard. In this report we present our work to date on solving this problem.

Our current objective is to develop a baseline system for testing the fundamental aspects of the problem. To do this we plan to use a vehicle equipped with an IMU and a computer flying inside our motion capture (MoCap) facility. MoCap temporarily takes the place of exteroceptive sensors (e.g. stereo camera, scanning laser range finder, etc.) and SLAM algorithms by modeling their measurements with corrupted and delayed MoCap data. Using this setup we can tune the vehicle's controller as well as refine the details of integrating IMU-based and SLAM-based navigation information. Once this system is working we can readily produce performance bounds (e.g. maximum measurement error, maximum processing delay, minimum update rate, etc.) that must be met by the real sensors and SLAM algorithms.

We've made considerable progress toward completing this baseline system. That progress is reported in the following sections. In Section II we briefly discuss some related work and then present the theory behind our approach to solving the navigation problem. We discuss some details of how this theory is implemented in Section III. We present our current results in Section IV followed by a discussion in Section V of some remaining steps needed to complete the baseline system and prepare for future work. In Section VI we offer some concluding thoughts.

II. Background and Theory

A. Related Work

Multirotor aircraft have recently become popular research platforms for indoor flight. These aircraft are useful for this type of research because they are simple, relatively inexpensive, easily repairable, and provide hover capability. However, because of size, weight, and power (SWaP) limitations, these aircraft are limited to a lower quality IMU and less onboard processing power.

Despite these limitations, there have been a few successful implementations of SLAM-based estimation and control on multirotor aircraft. MIT's Robust Robotics Group was the among the first [1], [3]. To produce estimates quickly enough for use within the control loop they used an extended Kalman filter (EKF) to estimate the robot states from laser scan matching and IMU updates at 30 Hz. Laser scans were also used in a particle filter-based SLAM approach to correct drift at a rate of 1 Hz. Another implementation was presented by Grzonka, et al. [7]. They created an open source quadrotor SLAM system also using a laser scanner and IMU. To simplify the estimation, the states were partitioned. The autopilot estimated and controlled roll and pitch based on IMU data. North, East, and yaw were estimated with particle filter-based SLAM, and height was estimated with an EKF based on portions of the laser scan deflected toward the ground.

Laser scanners are popular sensors for SLAM implementations as they offer fast and accurate information. Yet for a vehicle moving in six degrees of freedom (DOF), they offer a more limited awareness of the vehicle's immediate surroundings. Other researchers have tried to use vision instead of a laser scanner to navigate a multirotor aircraft indoors. Blosch, et al. [5] used a single downward-looking camera in a SLAM implementation that relies on computationally expensive bundle adjustment to construct a map. They were able to navigate with SLAM in the control loop through an unknown environment. However, their approach requires some non-trivial manual input due to scale and rotational drift in their SLAM-generated map. Their controller also requires a wired USB connection from the vehicle to a ground computer to maintain a shorter and more deterministic communication delay. Ahrens, et al. [2] used an EKF to predict states using integrated IMU information. Updates were provided by a forward-looking camera tracking visual landmarks. They were able to demonstrate short flights in fairly controlled conditions with the estimates in the control loop. They also mention that without the camera updates, the state estimates diverged before the aircraft could take off.

To control the fast dynamics of a rotorcraft, accurate estimates of the aircraft states must be available at a sufficiently high rate. In this paper, we outline a new estimation and con-

trol approach utilizing a better dynamic model for multirotor aircraft. This new model, originally presented by Martin and Salaun [9], includes terms for rotor drag. The rotor drag term is key to providing a better estimator, especially for roll, pitch, forward velocity and side velocity. We have worked to extend this method by estimating states using an EKF to fuse IMU and vision-based updates. Utilizing the more accurate rotorcraft dynamic model allows more reliable estimates of the aircraft states than can be done with the traditional approach, even with relatively slow (about 40Hz) IMU updates. These better estimates also allow for less frequent vision measurement updates, which is helpful given the processing limitations of the system.

B. Equations of Motion

The equations of motion below are taken, with only slight deviations, from [9]. Those authors have taken an innovative approach to multirotor dynamic modeling by including terms for the rotor drag, or the aerodynamic terms proportional to the propeller angular velocity multiplied by the forward or side velocities of the aircraft. First the single propeller model is stated and then the full hexacopter dynamic model is derived.

1) Single Propeller Model: The force and moment equations for a single propeller near hover are given by

$$\mathbf{f}_i = -k_F\omega_i^2\vec{k}_b - \omega_i\left(\lambda_1\mathbf{v}_i^{\perp} - \lambda_2\mathbf{\Omega}_i\times\vec{k}_b\right), \quad (1)$$

$$\mathbf{m}_i = -k_M\epsilon_i\omega_i^2\vec{k}_b - \omega_i\left(\mu_1\mathbf{v}_i^{\perp} + \mu_2\mathbf{\Omega}_i\times\vec{k}_b\right), \quad (2)$$

where the $\perp$ operator for some vector $\mathbf{x}$ is defined as

$$\mathbf{x}^{\perp} = \vec{k}_b\times(\mathbf{x}\times\vec{k}_b) = \mathbf{x} - (\mathbf{x}\cdot\vec{k}_b)\vec{k}_b. \quad (3)$$

The angular velocity of motor i is denoted ω_i, $\vec{k}_b$ is the unit vector for the body k axis, and k_F is the force per angular velocity squared of the motor/propeller combination. The constant k_M is the reaction torque per angular velocity squared of the motor/propeller. Aerodynamic constant λ_1 is the force per forward velocity times angular velocity of the motor, and λ_2 is the force per angular velocity of the aircraft times the angular velocity of the motor. The constant μ_1 is the reaction moment per forward velocity times angular velocity of the propeller and μ_2 is the reaction moment per angular velocity of the aircraft times angular velocity of the propeller. The vector forward velocity of the center of mass G_i is denoted $\mathbf{v}_i$, and $\mathbf{\Omega}_i$ is the angular velocity of the center of mass G_i. The constant ϵ_i denotes the direction the rotor is spinning, -1 for clockwise and 1 for counter-clockwise.

2) Complete Hexacopter Equations: The above equations for a single propeller/motor can be combined to model a full hexacopter platform. As shown in Fig. 1, the hexacopter is characterized by six motors attached on a rigid frame. The center of mass of the combined assembly is at G. The motors rotate according to the figure, with three motors turning one direction and three the other. The inertial frame is defined by north $\vec{i}_i$, east $\vec{j}_i$, and down $\vec{k}_i$ directions and the body frame by $\vec{i}_b$, $\vec{j}_b$, and $\vec{k}_b$, as shown. Here we have assumed that the motors are attached to the links at their centers of mass and that the center of mass for the whole vehicle is some distance h in direction $\vec{k}_b$ from the geometric center A. The motors are all positioned at a length l_1 from A, but in body coordinates, $l_2 = l_1\sin(\alpha)$ is in the $\vec{j}_b$ direction and $l_3 = l_1\cos(\alpha)$ is in the $\vec{i}_b$ direction. The states in the hexacopter model are the inertial north, east and down positions, the body frame velocities, the Euler angles defining the orientation of the body frame and the angular velocities in the body frame [4]:

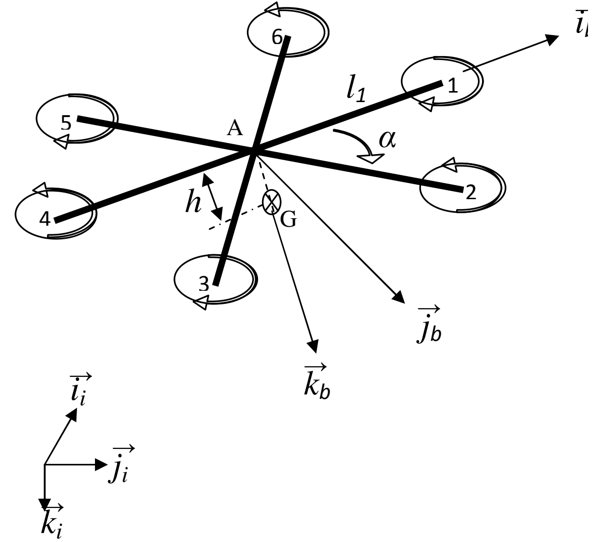

Fig. 1. Hexacopter Sketch

$$\mathbf{x} = \begin{bmatrix} n & e & d & u & v & w & \phi & \theta & \psi & p & q & r \end{bmatrix}^{\mathrm{T}}. \quad (4)$$

We assume the only forces and moments acting on the body are the forces and moments from the propellers and the weight of the body. We neglect all other forces and moments, such as the aerodynamic drag on the body. The generalized equations are then

$$m\dot{\mathbf{v}}_G + m\mathbf{\Omega}\times\mathbf{v}_G = m\left[R_I^B(\phi,\theta,\psi)\right]\mathbf{g} + \sum\mathbf{f}_i \quad (5)$$

$$[I_G]\dot{\mathbf{\Omega}} + \mathbf{\Omega}\times[I_G]\mathbf{\Omega} = \sum\mathbf{GG}_i\times\mathbf{f}_i + \mathbf{m}_i, \quad (6)$$

where $\left[R_I^B(\phi,\theta,\psi)\right]$ is the rotation matrix from the inertial to the body frame [4].

It is important to note that Eq. (1) and Eq. (2) cannot be used directly in Eq.(5) and Eq. (6) since $\mathbf{v}_i$ is not the same as $\mathbf{v}_G$. The relationship between $\mathbf{v}_i$ and $\mathbf{v}_G$ is given by

$$\mathbf{v}_i^{\perp} = \mathbf{v}_G^{\perp} - (h\mathbf{\Omega}\times\vec{k}_b)^{\perp} + (\mathbf{\Omega}\times\mathbf{AG}_i)^{\perp}, \quad (7)$$

where $\mathbf{AG}_i$ is the vector from the geometric center A to the center of mass of each of the motor/propeller components G_i. This transformation results in the following change of constants for the force and moment equations: $\acute{\lambda_2} = \lambda_2 - \lambda_1 h$, and $\acute{\mu_2} = \mu_2 - \mu_1 h$.

The equations for the summation of body forces are

$$\begin{aligned}\sum \mathbf{f}_i &= -k_F \sum_{i=1}^{6} (\omega_i^2) \overrightarrow{k}_b - \sum_{i=1}^{6} (\omega_i)(\lambda_1 \mathbf{v}_G^{\perp} - \\ &\quad \acute{\lambda_2} \mathbf{\Omega} \times \overrightarrow{k}_b) + r\lambda_1 \sum_{i=1}^{6} (\omega_i \mathbf{AG}_i) \times \overrightarrow{k}_b \quad (8) \\ &\approx -k_F \sum_{i=1}^{6} (\omega_i^2) \overrightarrow{k}_b - \sum_{i=1}^{6} (\omega_i) \lambda_1 \mathbf{v}_G^{\perp}. \quad (9)\end{aligned}$$

Martin and Salaun state that for a stiff propeller, $\acute{\lambda_2} \approx 0$. The last term in Eq. (8) is kept for the derivation of the moment equation; after that it is dropped from the final equation, Eq. (9).

The sum of all the moments is the force cross product combined with the moments induced by the motors

$$\begin{aligned}&\sum_{i=1}^{6} \mathbf{GG}_i \times \mathbf{f}_i + \mathbf{m}_i \approx -k_F (\sum_{i=1}^{6} \omega_i^2 \mathbf{GG}_i) \times \overrightarrow{k}_b - \\ &\quad k_M (\sum_{i=1}^{6} \epsilon_i \omega_i^2) \overrightarrow{k}_b - r\lambda_1 l^2 (\sum_{i=1}^{6} \omega_i) \overrightarrow{k}_b - \\ &\quad (\sum_{i=1}^{6} \omega_i) \left(\acute{\mu_1} \mathbf{v}_G^{\perp} - \lambda_1 h \mathbf{v}_G^{\perp} \times \overrightarrow{k}_b + \acute{\mu_2} \mathbf{\Omega}^{\perp} \right). \quad (10)\end{aligned}$$

Combining Eq. (9) with Eq.(5) and Eq. (10) with Eq. (6) leave us with the dynamic equations of motion

$$m \begin{bmatrix} \dot{u} + qw - rv \\ \dot{v} + ru - pw \\ \dot{w} + pv - qu \end{bmatrix} = \begin{bmatrix} -mg\sin(\theta) \\ mg\sin(\phi)\cos(\theta) \\ mg\cos(\phi)\cos(\theta) \end{bmatrix} + \begin{bmatrix} -\lambda_1(\omega_1+\omega_2+\omega_3+\omega_4+\omega_5+\omega_6)u \\ -\lambda_1(\omega_1+\omega_2+\omega_3+\omega_4+\omega_5+\omega_6)v \\ -k_F(\omega_1^2+\omega_2^2+\omega_3^2+\omega_4^2+\omega_5^2+\omega_6^2) \end{bmatrix}, \quad (11)$$

$$\begin{bmatrix} I_1 & 0 & I_{13} \\ 0 & I_2 & 0 \\ I_{13} & 0 & I_3 \end{bmatrix} \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \begin{bmatrix} I_1 & 0 & I_{13} \\ 0 & I_2 & 0 \\ I_{13} & 0 & I_3 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} -k_F l_2(\omega_2^2+\omega_3^2-\omega_5^2-\omega_6^2) \\ -k_F l_1(\omega_4^2-\omega_1^2) - a l_3(\omega_3^2+\omega_5^2-\omega_2^2-\omega_6^2) \\ -k_M(\omega_1^2-\omega_2^2+\omega_3^2-\omega_4^2+\omega_5^2-\omega_6^2) \end{bmatrix} + \begin{bmatrix} (\omega_1+\omega_2+...+\omega_6)(\acute{\mu}_2 q + \mu_1 u - \lambda_1 h v) \\ (\omega_1+\omega_2+...+\omega_6)(-\acute{\mu}_2 p + \mu_1 v + \lambda_1 h u) \\ (\omega_1+\omega_2+...+\omega_6)\lambda_1 l_1^2 r \end{bmatrix}. \quad (12)$$

The kinematic equations of motion are given by

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \frac{\sin(\phi)}{\cos(\theta)} & \frac{\cos(\phi)}{\cos(\theta)} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}, \quad (13)$$

$$\begin{bmatrix} \dot{n} \\ \dot{e} \\ \dot{d} \end{bmatrix} = \begin{bmatrix} c\theta c\psi & s\phi s\theta c\psi - c\phi s\psi & c\phi s\theta c\psi + s\phi s\psi \\ c\theta s\psi & s\phi s\theta s\psi + c\phi c\psi & c\phi s\theta s\psi - s\phi c\psi \\ -s\theta & s\phi c\theta & c\phi c\theta \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix}, \quad (14)$$

where $c\theta = \cos(\theta)$, $s\theta = \sin(\theta)$, etc.

Equations (11) through (15) are the equations of motion for the hexacopter model. It turns out that Eq. (11) is identical to that derived by [9], but there are differences in Eq. (12). The most noticeable changes are from the force cross product. There are also slight differences in the last matrix term.

Since the hexacopter is symmetric about the plane spanned by $\overrightarrow{i}_b$ and $\overrightarrow{k}_b$, two of the products of inertia are zero. We have not assumed a second plane of symmetry in our work because of the way sensors have been mounted to the platform. Beard and McLain [4] describe the method to isolate the $[\dot{p}\ \dot{q}\ \dot{r}]^T$ vector in Eq. (12)

3) Motor Dynamics and Attitude Control: As there is already an autopilot onboard the hexacopter, we only use the information in this section in our simulation model, but we have chosen to include it for completeness. We have chosen a simple speedup model for the motor model, like that described in [10]. Since the motors are always spinning about the $\overrightarrow{k}_b$ axis, we have dropped the vector notation from the motor angular velocities.

$$\dot{\omega}_i = k_m(\omega_{i(\text{des})} - \omega_i) \quad (15)$$

For the hexacopter, $\omega_{i(\text{des})}$ is determined from the matrix equation

$$\begin{bmatrix} \omega_{1(\text{des})} \\ \omega_{2(\text{des})} \\ \omega_{3(\text{des})} \\ \omega_{4(\text{des})} \\ \omega_{5(\text{des})} \\ \omega_{6(\text{des})} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & -1 \\ 1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ 1 & 0 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \delta_\omega + \omega_h \\ \delta_\phi \\ \delta_\theta \\ \delta_\psi \end{bmatrix}, \quad (16)$$

where δ_ϕ, δ_θ, $\delta\psi$, and δ_ω are changes in motor speed to obtain desired roll, pitch, yaw angles, and thrust level. The motor speed ω_h is the amount of angular velocity required to hover ($\omega_h = \sqrt{\frac{mg}{6k_F}}$).

C. State Estimation

We are estimating the first nine states in Eqn. (4) using a continuous-discrete extended Kalman filter (EKF). We utilize the standard algorithm for our implementation [4], except that we use two asynchronous measurement updates. As is shown below, care must be taken to handle these updates in the correct manner for the filter to track truth correctly. We utilize the information from our motion capture system as truth to compare the estimates of this algorithm.

1) Prediction: We use the nonlinear functions of the states and inputs $\dot{x} = f(x, u)$ and the corresponding Jacobians $A = \frac{\partial f(\mathbf{x},\mathbf{u})}{\partial \mathbf{x}}$ and $B = \frac{\partial f(\mathbf{x},\mathbf{u})}{\partial \mathbf{u}}$ to predict the state $\mathbf{x}$ and its covariance P from the previous update time to the current update time, designated δt. Rather than use the control inputs as the $\mathbf{u}$ in the EKF, we chose to use the measurements from the gyroscopes and motor speeds. These measurements give us the information we need from the inner-loop of the Mikrokopter autopilot without having to model it explicitly. With this change comes a slight change in the prediction equation for the covariance P:

$$P^+ = P + \delta t(AP + PA^{\mathrm{T}} + BGB^{\mathrm{T}} + Q), \quad (17)$$

where G is the diagonal covariance describing the noise on the input $\mathbf{u}$.

The filter states are

$$\mathbf{x} = \begin{bmatrix} n & e & d & u & v & w & \phi & \theta & \psi & \beta_x & \beta_y & \beta_z \end{bmatrix}^{\mathrm{T}}, \qquad (18)$$

where β_x, β_y and β_z are the biases of the gyroscopes aligned with the body axes. The input $\mathbf{u}$ to the EKF is

$$\mathbf{u} = \begin{bmatrix} p & q & r & m_1 & m_2 & m_3 & m_4 & m_5 & m_6 \end{bmatrix}^{\mathrm{T}}, \qquad (19)$$

as we use the gyroscope measurements as the estimates for p, q, and r directly.

Equations (11), (13), (15), and (20) are the nonlinear equations used in the prediction step. The only change is the replacement of the estimates p, q, and r in those equations with the estimate subtracted by the gyro bias states; i.e. $p = (p - \beta_x)$, $q = (q - \beta_y)$, and $r = (r - \beta_z)$.

$$\begin{bmatrix} \dot{\beta}_x \\ \dot{\beta}_y \\ \dot{\beta}_z \end{bmatrix} = \mathbf{0} \qquad (20)$$

2) IMU Update: The improved dynamic model gives us a better update for the states in the EKF. Traditionally, the $\vec{i}_b$ and $\vec{j}_b$ axis accelerometers are assumed to be zero and the $\vec{k}_b$ accelerometer is used to perform updates on ϕ and θ by comparing that accelerometer value to a rotated gravity acceleration. As Martin and Salaun [9] report, the $\vec{i}_b$ and $\vec{j}_b$ axes accelerometers actually measure the rotor drag. We have verified that this is true; see Figure 2. We are able to perform

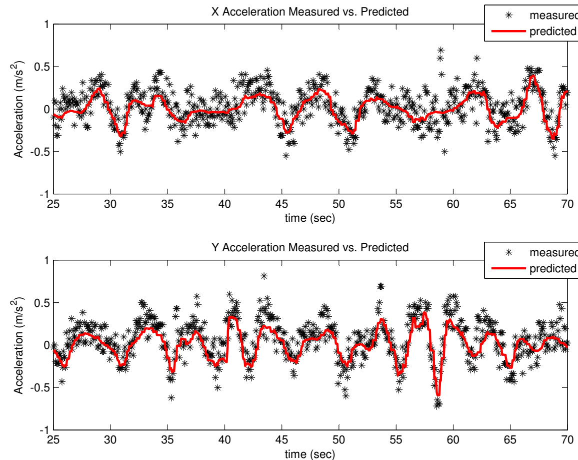

Fig. 2. Comparison of the accelerometer raw data with the rotor drag model during flight.

a direct update on the velocities u and v using the following nonlinear measurement equation

$$\begin{bmatrix} acc_x \\ acc_y \end{bmatrix} = \mathbf{h}_{acc}(\mathbf{x}, \mathbf{u}) + \begin{bmatrix} \alpha_x \\ \alpha_y \end{bmatrix} + \eta_{acc}[n], \qquad (21)$$

$$\mathbf{h}_{acc}(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} \frac{-\lambda_1}{m}(\omega_1 + \omega_2 + \omega_3 + \omega_4 + \omega_5 + \omega_6)u \\ \frac{-\lambda_1}{m}(\omega_1 + \omega_2 + \omega_3 + \omega_4 + \omega_5 + \omega_6)v \end{bmatrix} \qquad (22)$$

The values acc_x and acc_y are the measurements from the body frame x and y axis accelerometers. The vector term $\mathbf{h}_{acc}(\mathbf{x}, \mathbf{u})$ represents the model derived in [9] for body frame x and y axis accelerometer measurements. The constants α_x and α_y in (21) are predetermined biases in the respective accelerometer measurements and are treated as constants during flight. The η_{acc} term represents zero mean, white noise.

Since u and v are updated directly, we are able to obtain much better estimates of the positions n and e than is possible with the traditional approach of using the double integral of acceleration [2], [1]. This is true even at slow (40 Hz) accelerometer measurement updates. It is also noteworthy that even though ϕ and θ are not updated directly by the measurement equation, we are able to achieve better estimation of these angles than can be done using the traditional approach. This is due to the high correlation between ϕ and θ and u and v.

3) Delayed Camera Update: Initially when writing the filter, we made an assumption that camera data was instantly available to incorporate into the filter when a picture was taken. However we knew that this was not correct. There is a long delay between when an image is taken and when the data has been processed and is ready to be incorporated into the filter. Now a synthesized delayed camera update is used to correct the estimates. We have been able achieve adequate estimates while only using a very slow 2 to 3 Hz update rate.

We currently model the camera measurement as

$$\mathbf{h}_{cam}(\mathbf{x}) = \begin{bmatrix} cam_n \\ cam_e \\ cam_d \\ cam_{psi} \end{bmatrix} = \begin{bmatrix} n \\ e \\ d \\ \psi \end{bmatrix} + \eta_{cam}[n]. \qquad (23)$$

We utilize information from our motion capture system to create a measurement for the update. Based on a rough first-draft visual odometry program, we developed an estimate of the delay and noise characteristics that we might expect for the state estimates that it produces. We use this information to corrupt MoCap data. The data are delayed by up to one image time step (e.g. at 2 Hz, the data may be delayed by 0.5 seconds). Noise, corresponding to $\eta_{cam}[n]$, is also added and modeled as zero mean and white.

It is important that the delays in the image information are handled appropriately. To handle the delay, the filter must save the state and covariance estimates and IMU data from the filter time step immediately before the picture is taken and then save IMU information as it is received until the picture data is available. During that interval the filter continues to use the IMU information for prediction and updates as normal. Upon receiving the camera data, the filter recalls the saved state and covariance, predicts it forward to the time the picture was taken, applies the camera measurement update, and then re-applies the remaining IMU information. We currently require these steps to be accomplished before more IMU information is received by the filter. Figures 3 through 5 illustrate the problems with delay. Figure 3 shows the estimation without a delay. Figure 4 shows the same dataset, but with a 0.33 second delay that is not handled correctly. Figure 5 shows the results when the 0.33 second delay is handled as described above. The delayed-updated estimator does produce results that are visibly worse than when there is no delay, but the filter is still able to function, in contrast to the filter where the delay is

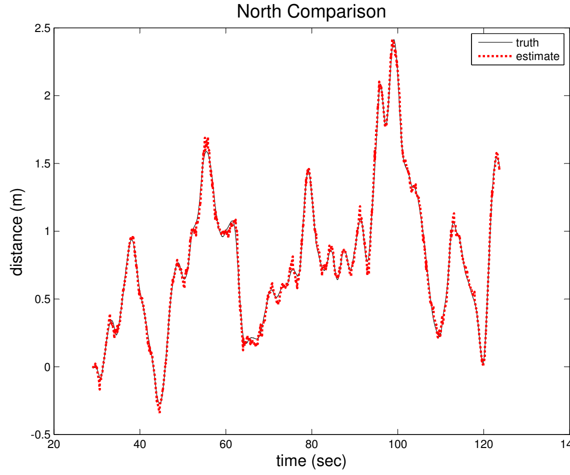

Fig. 3. Estimation applied with no delay in 0.33 second camera updates.

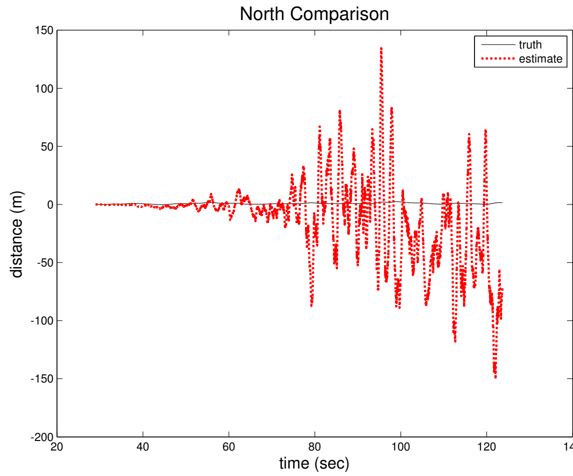

Fig. 4. Estimation applied with a delay in 0.33 second camera updates that is not handled.

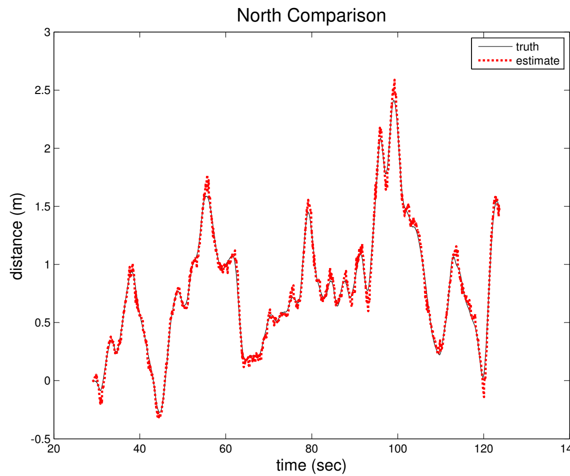

Fig. 5. Estimation applied with a delay in 0.33 second camera updates that is handled properly.

simply ignored.

D. LQR Control

Our control method revolves around the idea of a differentially flat system. A differentially flat system is one in which the state and control inputs can be expressed as functions of the output and its time derivatives [6], [12], [8]. Using differential flatness, feedforward control commands are calculated based on a desired, time-based trajectory. In particular, we are able to define as a path any twice-differentialable function of time. Since we do not have a perfect model of the system and it does not respond perfectly to control commands, we added an LQR feedback controller to keep the hexacopter on the desired path. The system diagram, with the estimation in the loop, is shown in figure 6.

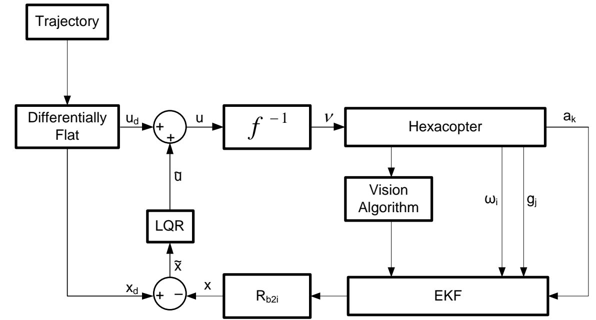

Fig. 6. System block diagram.

Using the traditional multi-rotor dynamic model to derive our differentially flat equations, we were able to define a linear state-space model for the control by wrapping the nonlinearities of the system into an input function u_c. The states for the control are

$$\mathbf{x} = \begin{bmatrix} n & e & d & \dot{n} & \dot{e} & \dot{d} & \psi \end{bmatrix}^{\mathrm{T}}. \tag{24}$$

Our equations of motion for these states are

$$\begin{bmatrix} \ddot{p}_n \\ \ddot{p}_e \\ \ddot{p}_d \end{bmatrix} = R_b^i(\psi,\theta,\phi) \begin{bmatrix} 0 \\ 0 \\ -T \end{bmatrix} \left(\frac{1}{m_h}\right) + \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix}, \tag{25}$$

where T is the thrust, m_h is the mass of the hexacopter, and $R_b^i(\psi,\theta,\phi)$ is the rotation matrix from the body frame to the inertial frame, the transpose of the rotation in Eq. (6). The control inputs that the hexacopter expects are

$$\nu = \begin{bmatrix} T \\ \phi \\ \theta \\ r \end{bmatrix},$$

which we can wrap into a non-linear function of the inputs $\mathbf{u}_c$ as shown below.

$$\mathbf{u}_c = \begin{bmatrix} \ddot{n} \\ \ddot{e} \\ \ddot{d} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_{p(3x1)} \\ u_{\psi(1x1)} \end{bmatrix} \triangleq \begin{bmatrix} f_p\left(\mathbf{x},\nu\right) \\ f_\psi\left(\nu,q\right) \end{bmatrix}, \tag{26}$$

where

$$f_p(\mathbf{x}, \nu) = R_b^i(\psi, \theta, \phi) \begin{bmatrix} 0 \\ 0 \\ -T \end{bmatrix} \left(\frac{1}{m_h} \right). \tag{27}$$

and

$$f_\psi(\mathbf{x}, \nu, q) = q \frac{\sin\phi}{\cos\theta} + r \frac{\cos\phi}{cos\theta}. \tag{28}$$

This wrapping permits us to express the dynamics as the following linear model

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u} + bg, \tag{29}$$

where

$$A = \begin{bmatrix} 0_{3x3} & I_{3x3} & 0_{3x1} \\ 0_{3x3} & 0_{3x3} & 0_{3x1} \\ 0_{1x3} & 0_{1x3} & 0_{1x1} \end{bmatrix},$$

$$B = \begin{bmatrix} 0_{3x3} & 0_{3x1} \\ I_{3x3} & 0_{3x1} \\ 0_{1x3} & 1 \end{bmatrix},$$

and

$$b = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}^{\mathrm{T}}.$$

Once the feedforward and feedback control commands are combined, the inverses of Eqs. (27) and (28) are used to map the input $\mathbf{u}_c$ into the nonlinear input ν that is expected by the platform, f^{-1} on figure 6. This trick greatly facilitates the analysis and tuning of the control, as the input $\mathbf{u}_c$ is in units of acceleration and the gains for the control can be computed using Matlab and the *lqrd* command.

III. Implementation Description

This section is intended to serve as a detailed description of the hardware and software currently being used to implement the baseline system. The goal of this section is to enable anyone with the same software and hardware (including possibly ourselves at a later date) to easily reproduce our current results. This section is especially needed for making future improvements. The development process has naturally progressed through several configurations of hardware and generated several versions of rather messy software that can be hard to decipher even when familiar. We should crystallize the current configuration in this report to ensure we're clear on where we're at and how to move forward.

A. Overview of Major Hardware Components

1) Hardware in the Air: We'll start with a discussion of the hardware setup of the air vehicle. To facilitate the description see Fig. 7 through Fig. 12.

We have chosen to implement our system based on the HexaKopter by MikroKopter. The basic HexaKopter has an impressive payload capacity of about 1.0 kg (2.2 lbs). We power our HexaKopter using a four cell lithium-polymer (LiPo) battery with a 5000 mA-h capacity. While we haven't directly tested the limits of its flight time, our experience suggests we can expect about 10 minutes of continuous flight time with the setup described in this report. Any hardware not described in the remainder of this section is part of the standard HexaKopter kit which is available from at least two US-based distributors ([11], [13]).

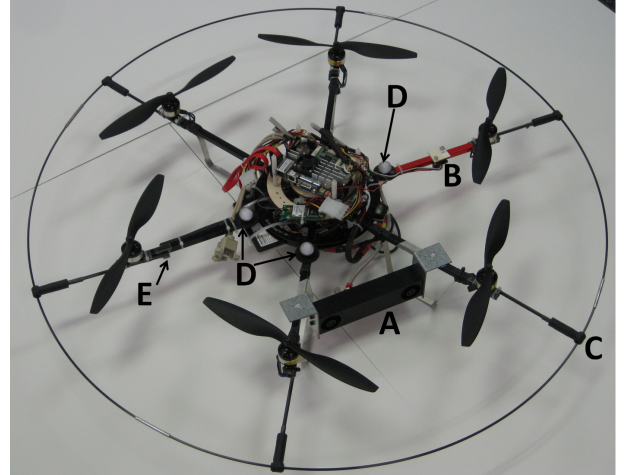

Fig. 7. This figure shows the HexaKopter configured as flown for the results in this report.

Our fully equipped HexaKopter is pictured in Fig. 7. At the point marked "A" you can see our Bumblebee2 stereo camera by Point Grey in our custom plastic case. For the results in this report the camera is not used, but it was mounted to the vehicle to provide realistic weight distribution. At the point marked "B" we've installed an ultrasonic ranger. This sensor was also not used for results in this report, but was installed just prior to this report to test its viability for integration into our system. (As an aside, we found this ranger was incapable during flight of detecting the floor when greater than 14 inches away; we believe this was due to the acoustic noise generated by the motors.) At the point marked "C" and around the whole perimeter of the HexaKopter we've installed propeller guards for added safety. The propeller guards are made of kite parts and flexible carbon rods, and the assembly is attached to the HexaKopter frame using cable ties and tape.

In Fig. 7 at the points indicated by "D" we have positioned the reflective markers tracked by the MoCap cameras. The positioning of these markers is only important if one is to reuse the template (called a 'prop' by the MoCap software) we created for tracking the HexaKopter as a single rigid body. If markers are placed in different locations, a new template can be easily created. There is one additional important note to mention here related to these markers. In the MoCap software we reduced the minimum number of horizontal lines per marker required for tracking to two. This setting makes it easier for the MoCap system to track the markers even when they are only visible by a small subset of the total MoCap cameras.

At the point indicated by "E" in Fig. 7, the standard piezo-electric speaker from MikroKopter is installed. The software in the MikroKopter controller monitors the battery voltage and activates this speaker when the voltage reaches a certain level. The result is a high-pitch beeping tone. If you are using the battery to power the little computer between flights, the HexaKopter should be left on to alert you to a low-voltage condition. It is also important to note that this speaker can

be difficult to hear during flight due to the noise of the HexaKopter motors.

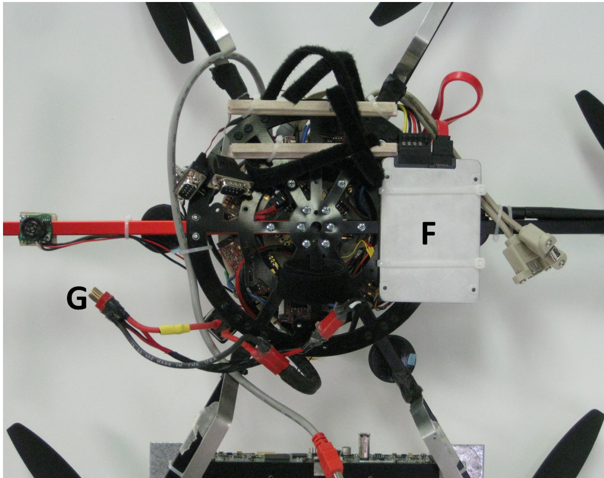

Fig. 8. This figure shows the underside of the Hex.

The underside of the HexaKopter is pictured in Fig. 8. At the point indicated by “F” we have installed a 120 GB solid state SATA hard drive. This hard drive is connected to the onboard computer through the red SATA cable, and it receives power through a connection to the left of the SATA cable. Originally we were using smaller solid state SATA drives that could directly connect to the SATA jack on the onboard computer. However, we found that even fairly gentle landings were sometimes sufficiently shocking to unseat these drives from their connection and cause the computer to crash. Also picture in Fig. 8 at the point indicated by “G” is the large deans connector to plug into the battery. Only the HexaKopter is powered directly from the battery; everything else receives its power through the DC power output of the onboard computer. The onboard computer, in turn, receives its power at 12V from a step-down voltage regulator connected between it and the battery. The battery is held to the frame of the HexaKopter by velcro straps pictured near the top of Fig. 8. This position allows it to counter balance the weight of the camera pictured opposite this position near the bottom of Fig. 8.

Fig. 9 shows a closer view of the HexaKopter payload from the right-hand side of the vehicle. At the point indicated by “H” you can see the switch installed between the battery and the HexaKopter. This switch is *not* connected between the onboard computer and the battery. At the point indicated by “I” you can see the step-down voltage regulator mentioned above. The regulator is strapped to a balsa wood ring that is mounted on standoffs connected to the HexaKopter frame. All of the electronics pictured in Fig. 9 below that balsa ring are standard MikroKopter electronics (brushless motor speed controllers and flight control board). At the point indicated by “J” in Fig. 9, a USB-to-Wifi card is strapped to the balsa ring. The antenna for the USB-to-Wifi card is strapped to the rear motor spar of the HexaKopter (see also Fig. 7 near the point indicated by “E”). This 802.11g link is the only communication channel between the HexaKopter and the ground computer.

Fig. 10 shows a view of the HexaKopter payload looking back from the front left-hand side of the vehicle. At the point indicated by “K” a MikroKopter-provided USB-to-Serial card is strapped to the balsa ring. On the USB side this card is connected to the onboard computer. On the serial side this card connects to the flight control board of the HexaKopter. This communication channel is the only connection to the underlying HexaKopter hardware. Through it the HexaKopter pushes IMU and motor speed data to the onboard computer and the onboard computer pushes control commands back to the HexaKopter. Note that the card also has a 10-pin header open to the outside of the balsa ring. These pins are for connecting to the flight control board via the serial cable *only* when reprogramming the brushless motor controllers.

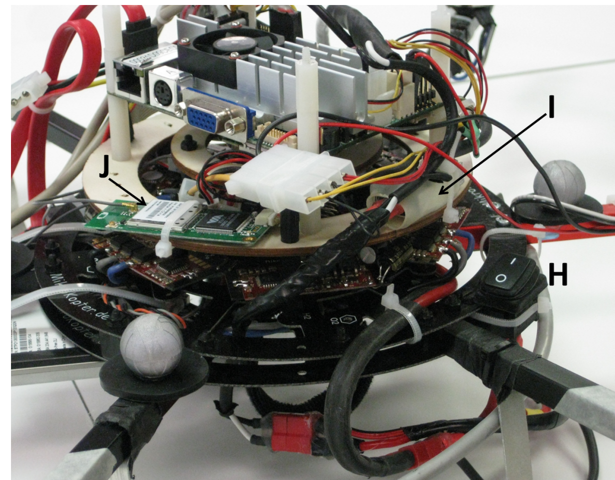

Fig. 9. This figure shows the right-hand side of the Hex (the red spar indicates the front of the Hex).

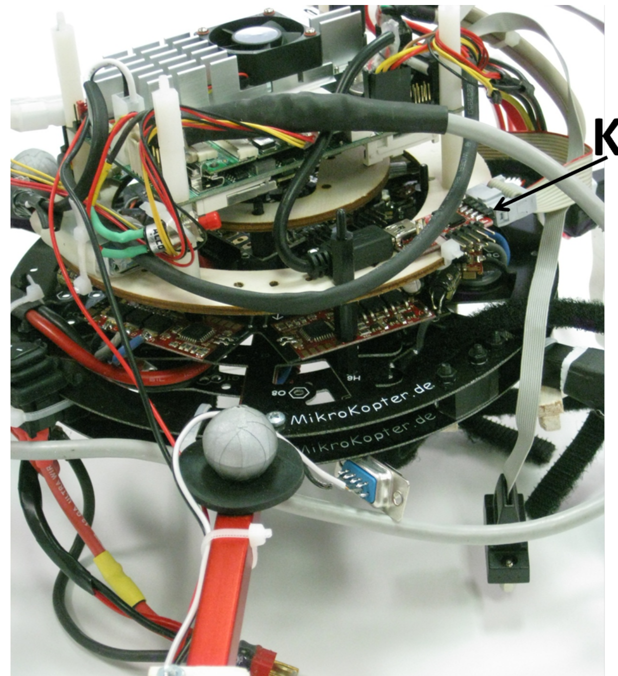

Fig. 10. This figure shows the front left-hand side of the Hex (the red spar indicates the front of the Hex).

Fig. 11 shows a top-down view of the HexaKopter payload

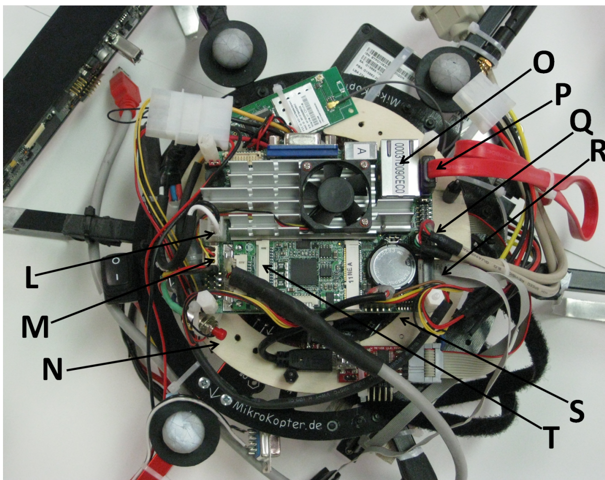

Fig. 11. This figure shows a top-down view of the Hex.

to illustrate the numerous connections made to the onboard computer. At the point indicated by “L” power is supplied to the computer from the voltage regulator. Point “M” indicates the connection for power and control of the computers cooling system. Point “N” indicates an On/Off switch for the computer that functions in the same way as the front panel buttons on a desktop (hold it down for a while to force a hard shut-off); it connects to the computer at the pins just below the point indicated by “M”. Point “O” indicates an unused LAN port on the computer, and the SATA cable to the hard drive connects at the point indicated by “P”. Extra USB connections are plugged into the computer at the point indicated by “Q”. Extra serial port connections are plugged in at the point indicated by “R”. The USB connections for the USB-to-Wifi and the USB-to-Serial cards are indicated by “S”. And finally, there is a mini-PCIe-to-Firewire card installed at the point indicated by “T”. This card provides communication and power for the stereo camera when in use. Fig. 12 provides an additional close-up view of the payload from yet another angle.

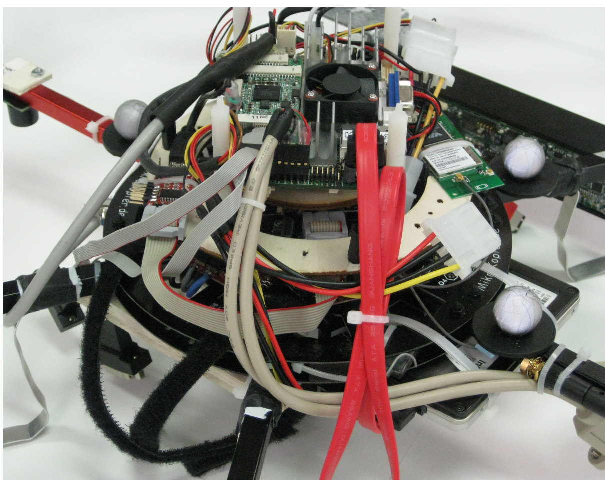

Fig. 12. This figure shows the rear left-hand side of the Hex (the red spar indicates the front of the Hex).

2) Hardware on the Ground: While the airborne hardware is more significant to document, a few notes should also be made about the necessary hardware employed on the ground. For the initial part of this discussion, see Fig. 13 through Fig. 15. These figures show images of our motion capture facility.

Fig. 13. This figure shows the “south” end of the MoCap facility. The north-east-down coordinate system used in the room is aligned such that the south direction is approximately normal into the wall completely pictured here. That would put the east direction roughly normal into the wall partially pictured in the left of this image. Care is taken with the MoCap calibration device (equipped with bubble levels) to align the down direction as close as possible with gravity.

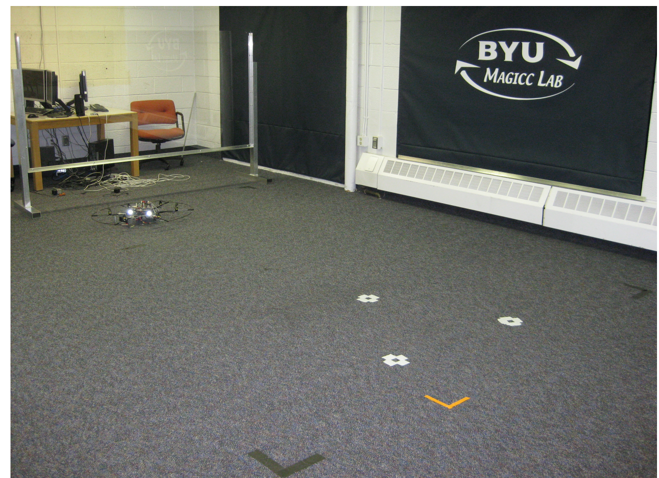

Fig. 14. This figure shows additional markings on the floor defining the approximate bounds of the flyable volume.

Fig. 13 shows the south end of the facility as per the north-east-down coordinate system defined during the calibration process. Pictured around the top of the image from left to right are Cameras 1-4. These cameras emit bursts of infrared light with a pulse repetition frequency of 200 Hz. Their lenses are filtered to only detect reflections of this infrared light, and that is how they track the reflective markers placed on the vehicle. Pictured in the middle of the left-hand side of Fig. 13 is the MoCap computer devoted to handling communication with the cameras and running the MoCap software (named

Fig. 15. This figure shows the “north” end of the MoCap facility.

Cortex). Each of the computers in the local network is assigned a static IP address; the MoCap computer is 192.168.1.100. We should mention here that the airborne computer’s IP address is 192.168.1.104.

The MoCap computer is hard wired to a router on the table in the center of Fig. 13. The table also holds two computers. The one on the left is the ground station computer, and its IP address is 192.168.1.101. The ground station computer hosts all the ground-based software specific to our baseline system. This includes the ground station software and the Cortex server (both discussed in greater detail below).

In order to acquire valid data, the MoCap system requires the vehicle fly within a volume well covered by its cameras’ fields of view. In Fig. 13, the approximate south corners of that volume are marked with black masking tape (hard to see in this image) near the feet of the left-most chair and near the right foot of the plexiglass safety shield. North corners of the volume are marked again with black masking tape near the center-right and center-bottom sides of Fig. 14. Finally, Fig. 15 shows the north end of the MoCap facility showing (from left to right) Cameras 5-8 across the top of the image.

The flyable volume is really defined by the space the MoCap cameras are able to see. To help make that aspect of the setup more clear, Fig. 16 through Fig. 21 are included in this report. They show the motion capture system’s belief, based on its calibration procedure, about where the cameras are and how they’re oriented. Only screenshots for Cameras 1 and 2 are included because Camera 1 is typical of Cameras 4, 5, and 8 in its pose relative to the volume. Such is also the case for Cameras 2, 3, 6, and 7. Saved with this report should also be a short movie giving an animated view of the camera arrangements; it is titled CortexSetup.avi

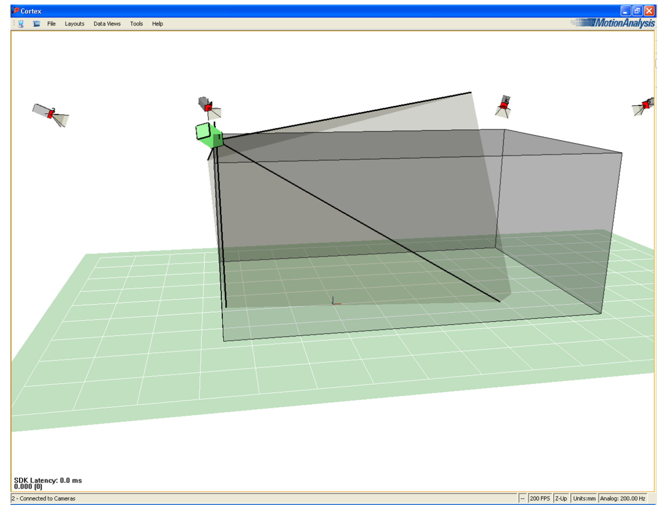

Fig. 16. This figure illustrates the MoCap system’s belief about the position and orientation of Camera 1 as seen in a perspective view from behind Camera 1.

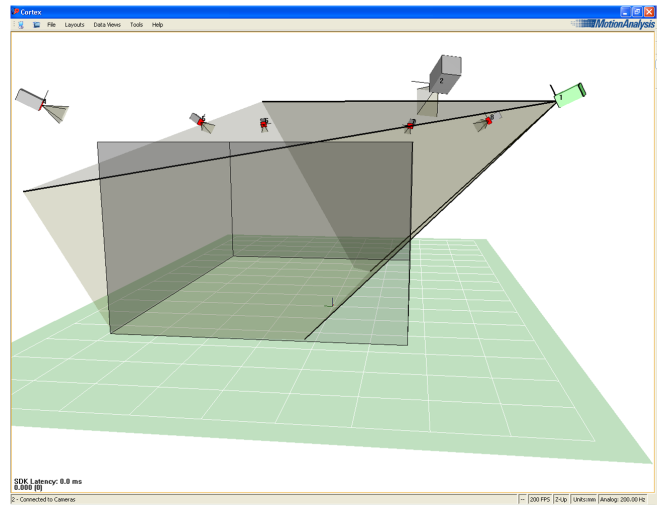

Fig. 17. This figure illustrates the MoCap system’s belief about the position and orientation of Camera 1 as seen in a perspective view from south of Camera 1.

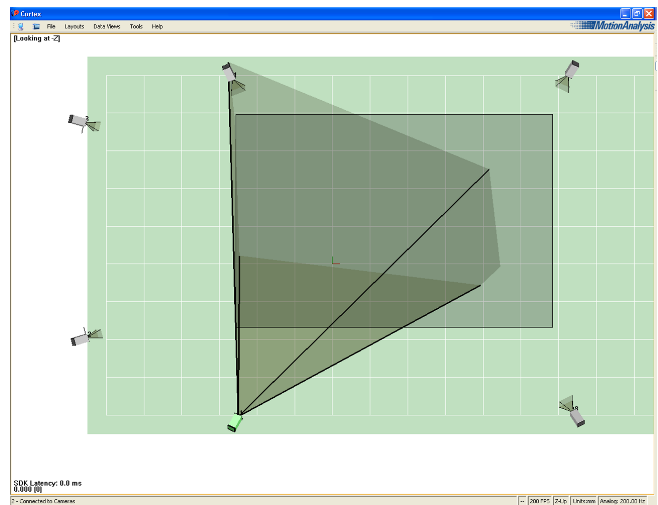

Fig. 18. This figure illustrates the MoCap system’s belief about the position and orientation of Camera 1 as seen in an orthographic view from above Camera 1.

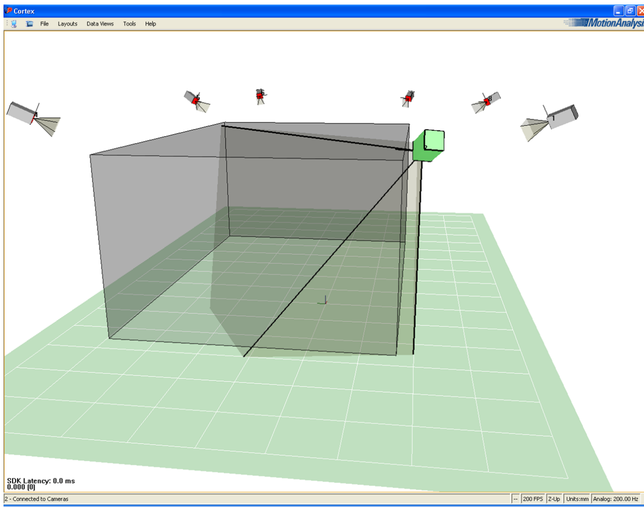

Fig. 19. This figure illustrates the MoCap system's belief about the position and orientation of Camera 2 as seen in a perspective view from behind Camera 2.

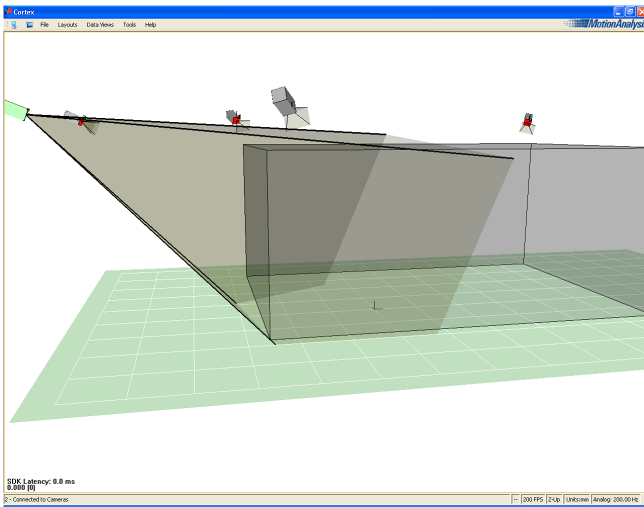

Fig. 20. This figure illustrates the MoCap system's belief about the position and orientation of Camera 2 as seen in a perspective view from east of Camera 2.

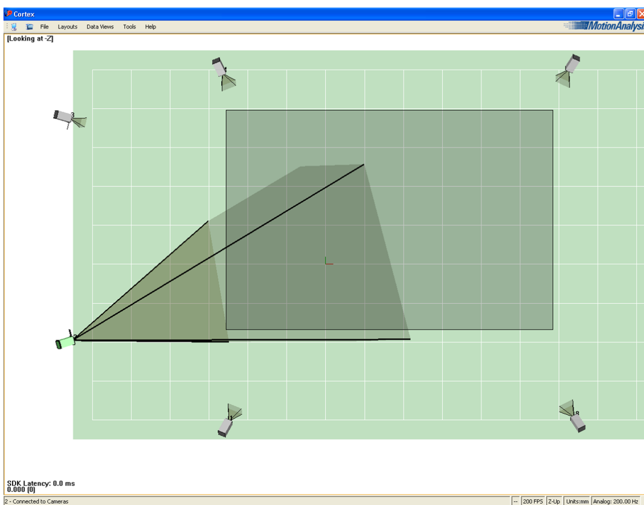

Fig. 21. This figure illustrates the MoCap system's belief about the position and orientation of Camera 2 as seen in an orthographic view from above Camera 2.

B. Overview of Major Software Components

There are three pieces of software needed to produce the results presented in this report. In order of increasing complexity, they are:

- The Cortex Server
- The Ground Station
- The HexaKopter Estimator and Controller

The former two operate on the ground station computer, while the latter resides on the computer onboard the HexaKopter. Each will be discussed in turn below.

1) The Cortex Server: The Cortex Server is relatively simple, and it has been a stable piece of code for several months now. It monitors the TCP/IP port for messages of certain, known types. One message type is pushed to the ground station computer by the MoCap software and contains the MoCap measurements of the vehicle's 3-D position and Euler angles representing orientation. When this message arrives, the Cortex Server reformats the MoCap data from its native x-y-z coordinate frame into the north-east-down coordinate frame. It also adjusts the units of the MoCap data such that positions are given in meters and angles are given in radians. This data is saved in a buffer until new MoCap data arrives or until the Cortex Server detects the other message type it listens for. This other message indicates a request from the Ground Station software. On this request the Cortex Server sends back to the Ground Station software the most recent reformatted MoCap data.

2) The Ground Station: As it exists at this writing, the Ground Station software is actually a fairly extensive bit of code. This is largely because it can also be configured to work with the Dynamics Server software (a C# implementation of the HexaKopter dynamics using a Runge-Kutta differential equation solver) to run simulations. Because of this option the Ground Station contains a version of the estimator and controller that would, in real flights, reside on the HexaKopter computer.

When used for real flights, the Ground Station software actually does very little. Its most important function is to periodically request MoCap data from the Cortex Server and then send that MoCap data along to the airborne computer where it is used to simulate stereo camera-based measurements. For the results presented in this report, the Ground Station requests and sends that data at a rate of about 10 Hz. In addition to the MoCap data sent from the Cortex Server, the Ground Station software computes and sends estimates of the vehicle's body frame linear and angular velocities. All twelve of these MoCap derived values are sent to the HexaKopter (along with a timestamp indicating when they were relevant), but only the three position measurements and the heading measurement are used to synthesize the camera measurement. The Ground Station also plots in real-time all twelve states measured or derived from MoCap using a GUI displayed on the ground station computer. As an additional option, the Ground Station software can also use its version of the controller to plot what it thinks the commanded states should be.

3) The HexaKopter Estimator and Controller: The HexaKopter Estimator and Controller software (hereafter Hex-

aKopter Software) is run by the computer onboard the HexaKopter. It implements the EKF used to fuse the IMU-based state estimates with the delayed camera measurements synthesized from MoCap data. The HexaKopter Software also implements the path planning and control algorithms. Since these are fundamental pieces of code we will discuss some of the more important parts in greater detail below.

For the results in this report, there are three main ways the HexaKopter Software interacts with things outside of itself. First, it listens for messages over the TCP/IP port and takes action when messages are received based on what type of predetermined message is given. This process is defined in the ProcessRequest method of the HexacopterServer class (see HexaKopterServer2.cs). If the message is a length four array of type double the software assumes it is a control command message being sent from the Ground Station software. This is a vestigial feature leftover from times before the controller was implemented on the HexaKopter Software, and it is not used for these results. Another unused message type the software listens for consists of an array of type char. This type of message used to be sent from the ground station to request that the HexaKopter Software send back certain types of data. The one message type that is still used is a length thirteen array of type double. In this case the software assumes the message contains the MoCap values needed to synthesize a delayed camera measurement. In response the software copies the sent values into class variables for eventual use by the EKF estimator and sets a flag high to indicate that new MoCap data has arrived.

The software's response to this last message type also contains a residual *feature* left-over from earlier development that almost became a serious bug. The software keeps a counter that it increments every time it receives MoCap data from the Ground Station. It tests the value of this counter when each new MoCap transmission comes in , and it only copies the values into the class *if this zero-based counter is greater than 3*. In other words, this little line of code has the effect of throwing away four of every five MoCap transmissions from the Ground Station. Therefore, for the results presented here, the estimator and controller are only benefiting from synthesized camera measurements at a rate of about 2 Hz.

The HexaKopter Software's remaining interactions with the outside world occur over the serial port. These interactions are governed by an instance of the HexacopterSerialLink class instantiated in the constructor of the HexacopterServer class. When the software receives a message from the HexaKopter on the serial port, it triggers the ProcessHexacopterMessage method in the HexacopterServer class. This method executes the code implementing the EKF estimator as well as the code that sends new control commands back to the HexaKopter. The messages pushed from the HexaKopter, which trigger all this activity, contain the most recent IMU and motor speed data from the HexaKopter and arrive at around 40 Hz.

C. Configuring For a Flight Test

We now present the steps necessary to configure the hardware and software for a flight test such as those presented in this report. We begin with a description of the many parameters and flags that can be set in the software and where to find them. We then provide a step-by-step checklist to follow before and during the flight test.

1) Software parameters and flags: As we've developed the software we've naturally focused more on getting changes to work at that moment than on writing it well for long-term use. As a result, there are many locations within the code where parameters are assigned values that affect the performance of the software. In this section we will list those parameters and their locations as they exist in the current version of the software. This section should be relevant to those in the future who may decide to go back and use the current version, but its near-term significance is to be used as a tool to help us rewrite some elements of the software.

A few of these parameters are set in the HexacopterServer class variable declarations. The boolean flag 'useOnlineControl' is set there (@ line 24 of HexacopterServer2.cs) to 'true' for the results presented in this report. This flag is used in a few 'if' statement conditions to differentiate between actions that should be taken when either the Ground Station or the HexaKopter Software is responsible for control. Another boolean flag, USEALTITUDE, can also be found here (@ line 31 of HexacopterServer2.cs) set to 'false'. It is used in 'if' statements to wrap actions specific to using the altimeter, and it is passed into the instance of the Estimator class for the same purpose.

The most important parameter set in the HexacopterServer class variable declarations is Time_Bias_millisec. This critical parameter captures the current best estimate of the offset between the clocks of the ground station and HexaKopter computers. Time_Bias_millisec must be adjusted before each flight because the HexaKopter Software will add this value to the timestamp sent by the Ground Station with its transmission of psuedo-camera data. The offset value is determined using the NTP Time Server Monitor software from Meinberg. For more details, see the description below in Section III-C2.

The poster child for why we're writing this section is the integer variable HexacopterServer.c_counter. This variable is initialized to zero and incremented whenever new MoCap information is received from the Ground Station. Only when c_counter is strictly greater than 3 (a threshold hardcoded at line 455 of HexacopterServer2.cs) will the HexacopterServer class copy the MoCap data into the appropriate class variables and set the flag indicating that new "camera" data is available. When this happens c_counter is reset to zero and the process repeats. The result is that the HexaKopter Software only implements 1 out of 5 of the camera updates it might otherwise. There is no current reason for doing this; it is leftover from a previous developmental expedient.

The method HexacopterServer.ProcessHexacopterMessage (@ line 227 of HexacopterServer2.cs) contains a hardcoded time threshold which is compared to the change in time (call it δt) between when the current and previous IMU data were received. If $\delta t > 0.07$ (the value used in this report's results) then we assume a hiccup occurred in the IMU data transmission and take corrective action. Also in this method (beginning at line 250) we have hardcoded several values

used to convert into SI units the IMU data transmitted by the HexaKopter. These conversions ought to be constant, but we mention them here in case this list of parameters is ever used to help apply this code to different hardware. Finally, a little further on in the method (@ line 351) a hardcoded argument of 10 is passed into the prediction step of the EKF. This argument controls the number of times by which the current δt is subdivided for intermediate prediction steps. A higher number here makes prediction more accurate but at the cost of increased computation.

The Estimator class contains even more parameters to be aware of. The first several lines of the class variable declarations (beginning at line 33 of Estimator.cs) contain important constants and their assigned values. Some of these are well known or easily found (e.g. gravity, mass of the vehicle). Others are derived from sensor measurements or are parameters of the HexaKopter model (e.g. kF, lam1x, lam1y) and may be the subject of some future tuning efforts.

Several parameters are assigned values in the Estimator class constructor. The initial values of the EKF covariance matrix, P, are assigned here (beginning at line 157 of Estimator.cs); these values were chosen without much experimentation or theoretical motivation. The process uncertainty matrix, Q, is also assigned values (beginning at line 191) in the constructor. For this report all but the last three diagonal entries of Q are left at zero since process uncertainty in the prediction step also enters through the BGB^T term as shown in Eq. (17) of Section II-C1. The values of Q have again not been experimented with much and may need tuning. Also included in this constructor are measurement uncertainty values derived from sensor tests for the altimeter (@ line 217) and for camera measurements (beginning at line 222). We should note here that the camera measurement variances assigned here are used in the Estimator.PseudoGPSlike_MeasureUpdate_DELAYED method to scale the random numbers added to MoCap data to synthesize the camera measurements.

A few remaining hardcoded parameters can be found in other methods of the Estimator class. Two instances occur in Estimator.Initialize. First (beginning at line 327 of Estimator.cs), the gyro biases in the EKF state vector are hardcoded with their initial values. Later (@ line 350), there are hardcoded thresholds to compare to current motor speed values. These thresholds and the initial values for the gyro biases could be the subject of future tuning. Finally, the Estimator.PseudoGPSlike_MeasureUpdate_DELAYED method contains three instances (@ lines 712, 749, and 765) where the EKF prediction method is called with a hardcoded input argument of five. As mentioned above, higher values for this input should make prediction more accurate but will increase the computational cost of this step.

The remaining parameters in the HexaKopter Software are currently found in the HexacopterPathControl class. Several are found in the class declarations, including a hardcoded path and text-file name (@ line 39 of HexacopterLQRControl.cs) and a few constants (beginning at line 16). The file referenced here contains values for the gain matrices used in calculating control commands. The constants assigned values here ought to be juxtaposed with some of the constants assigned elsewhere (especially the estimator) because they should be the same or related.

A few boolean flags are also set (beginning at line 23) in the HexacopterPathControl declarations. These flags, named useIntegrator and useKfromFile, control whether an integrator is used in calculating control commands and whether or not the gain values are read in from the text-file just discussed. For the results in this report, these flags were set to 'false' and 'true', respectively. If the gains were not to be read in from the file, their hardcoded values are assigned (beginning at line 132 of HexacopterLQRControl.cs) in the HexacopterPathControl class constructor.

The HexacopterPathControl class contains several paths which the vehicle can be commanded to follow. Each of these paths contain a number of hardcoded values that define their behavior in time. For the results in this report we used the path defined in the method SmoothLinePathDRAFT (beginning at line 226 of HexacopterLQRControl.cs), and we will only discuss its hardcoded parameters here. The variable 'maxVel' is set to 0.25 meters per second; this determines the maximum velocity along the line to be followed by the vehicle. The variable 'shift', set to 40, is a unitless number that determines the amount of time the vehicle hovers before smoothly transitioning into following the line. The 'initState' and 'endState' arrays also have hardcoded values assigned to some of their elements in the beginning of the method. All of these hardcoded values should eventually be replaced with arguments passed into the method and determined by a higher-level path planner.

Finally, all calls to the HexacopterPathControl.Saturate method (see lines 420, 462, and 475 of HexacopterLQR-Control.cs to identify all 12 occurrences) contain hardcoded arguments to set the saturation limits to be used for that call. These saturation limits need to be further explored and set to acheive better flight performance.

While one can find most of the parameters needed to control the system's performance in the HexaKopter Software as discussed above, some elements of the Ground Station also need to be considered. There are important values hardcoded values (beginning at line 140 of SimpleGroundStation.cs) in the constructor of the base class SimpleGroundStation. Here the IP addresses and port numbers are set that control whether the Ground Station is configured to interact with hardware or in a simulation mode. Similarly, the SimpleGroundStation.ConnectToServers method contains method calls (beginning at line 190) to connect to different servers that must be commented out based on how the hardware or simulation is configured.

We can find an interesting example of how the current code base can be confusing in the parameter SimpleGroundStation.lowLoopSpeedLimit. This private member of the base class is assigned a value of zero in the constructor (@ line 156), but it is reset to a value of 0.006 in the leaf class HexacopterGroundStationPath by means of a public getter and setter function. The parameter itself sets a bound (see line 670) on how fast the the Ground Station can execute the functions in its lower loop, including requesting MoCap data from the Cortex Server.

As stated in Section III-B2, the Ground Station software uses MoCap measurements to derive estimates of the vehicle's linear and angular velocities. This is implemented in the SimpleGroundStation.CalculateVelocities method using the "dirty derivative" algorithm [4]. The parameters controlling the cutoff frequency in this calculation are assigned hardcoded values in the base class declarations (beginning at line 70 of SimpleGroundStation.cs).

Lastly, the method SimpleGroundStation.runLowLoop contains a couple of items to consider. First, the call to the requestRobotControlChange method (see line 731 of SimpleGroundStation.cs) should be commented out for the results in this report. This ensures that the Ground Station doesn't try to send control commands to the HexaKopter. Immediately thereafter (@ line 734) we find the important hardcoded threshold used to compare against the time elapsed since the last transmission of MoCap data to the HexaKopter. If the elapsed time exceeds this threshold (set to 100 milliseconds for this report's results) then the software sends new MoCap data to the HexaKopter and performs some other related actions.

This concludes a fairly comprehensive list of the many parameters and code elements that need to be adjusted to configure the software appropriately. Some additional settings can be made, particularly in the ground station, that are not relevant for generating the results in this report. However, if one was to use the software for anything significantly different from this report (e.g. for simulation instead of hardware testing) then these parameters and settings should be sought out and understood. There is also a reasonable chance that we have overlooked in this list something important to our results. As stated in Section V, the code needs revision and refining to make it more user friendly in the future.

2) Flight Test Checklist: Several steps should be observed to prepare for a successful flight test. Those steps are documented in this section.

The first thing to do is to turn on the motion capture cameras and start the Cortex software on the motion capture computer. After connecting the Cortex software to the cameras, an important default setting in that software needs to be changed. In the 'Settings' menu under the 'Tracking' tab the 'Min horizontal lines per marker' should be set to 2. This setting allows the motion capture system to better track the vehicle. Back in the main Cortex GUI, the correct *.prop file also needs to be selected so that the Cortex software knows in what configuration to expect to see the markers. With the 'Motion Capture' tab selected, under the small 'Objects' tab on the right of the GUI, select "mikrokopter_prop_oct_20.prop" for the markers positioned according to this report. If the markers are positioned differently select the appropriate *.prop file here or create your own new one (only takes a few minutes). Finally, click 'Run' to start the motion capture system. Also, over on the ground station computer the Cortex Server software needs to be started at this point.

The next step is to set up the computer onboard the HexaKopter. Plug in and strap down the 4-cell LiPo battery. The computer should start automatically, but if it does not push the power button shown in Fig. 11 at the point indicated by "N". After waiting about 15 seconds, plug a keyboard into one of the extra USB ports on the onboard computer, type "circuit" and press enter. This logs into the user account on the onboard computer and allows it to finish booting up. Back at the ground station computer open the remote desktop connection software. Make sure the correct IP address is used for the remote desktop connection (192.168.1.104 for the onboard computer being used for the vision-based HexaKopter). The username for this computer is "MAGICCian" and the password is again "circuit". If the remote desktop connection fails, the most likely cause was not waiting long enough before entering "circuit" into the onboard computer.

Once the onboard computer is running, open the NTP monitoring software mentioned in Section III-C1 on both the onboard and ground station computers. The NTP service should be configured to begin even before this monitoring begins; ensure that it is running as desired. Then wait for 7 - 10 minutes while the NTP algorithm settles down into a fairly steady state. At a minimum this will be after the 'Reach' field has incremented to 377. It is probably also good to wait until the 'Poll' field has incremented from its initial value of 64 to its next value of 128. While waiting for the NTP algorithm to settle, open the Ground Station software on its computer and the HexaKopter Software on its computer. Once the NTP service has reached the state described above, hard code the 'Offset' value (as it appears in the HexaKopter computer's NTP monitoring software) into the HexacopterServer class variable Time_Bias_millisec.

To ensure the desired performance during the flight it is worthwhile here to also check near the end of the HexacopterServer class method ProcessHexacopterMessage. Ensure that only the desired path planner is uncommented.

At this point we should now be ready to turn on the HexaKopter using the switch pictured in Fig. 9 at the point labeled "H". Also turn on the RC transmitter. On top of the RC transmitter there is a two-position switch on the left-hand side. This switch controls whether the HexaKopter will accept control commands over its serial port and must be activated during the flight to allow the onboard computer to send control commands. To set the switch, it should be thrown toward the face of the RC transmitter. The RC transmitter should also be used at this stage to calibrate the gyroscopes on the HexaKopter. This is done by moving the throttle/yaw stick to its topmost and leftmost positions and holding it there until the HexaKopter makes three beeps from its piezoelectric speaker. Gyro calibration should only be done while the HexaKopter is left completely motionless.

The preceding steps conclude the preflight portion of the checklist. To begin flight, first run the HexaKopter Software and wait for the corresponding console to display a message indicating it has made a successful connection with the HexaKopter microprocessor. Next, run the Ground Station software and watch for the backgrounds to turn from red to green on the port numbers and IP addresses corresponding to the Cortex Server and HexaKopter. This indicates the Ground Station software is successfully communicating with the other pieces of software. Finally, start the motors at their idle speed by moving the throttle/yaw stick on the RC transmitter to its bottommost and rightmost position.

Now gradually increase the throttle on the RC transmitter to allow the HexaKopter to fly under computer control. The standby pilot holding the RC transmitter should especially be prepared to takeover control of the vehicle at this point and should remain vigilant during the duration of the autonomous flight. The HexaKopter can move very quickly. As a reminder, the RC transmitter can be used to give yaw rate and roll and pitch commands that will be *added* to whatever the computer is commanding the HexaKopter for these variables. However, the throttle command is different. The HexaKopter takes as its throttle command *whichever is lower* between the RC transmitter and the computer. For the computer to have the full control authority it expects, the throttle stick on the RC transmitter must be raised above the highest value we expect the computer to command. If the standby pilot must take control of the flight, we've found the best approach to be decreasing the throttle commanded by the RC transmitter while giving roll and pitch commands as needed to make a landing as gracefully as possible. It is generally *not* recommended to flip the switch on the RC transmitter that disables computer control because the throttle stick may be in a high position, in which case the HexaKopter might really do something unsavory.

To end the autonomous flight we currently have the conclusion of the autonomous path planned such that the vehicle is trying to indefinitely hover at a point not far off the ground. We then simply decrease the throttle on the RC transmitter until the HexaKopter has landed. Motors are turned off by moving the throttle/yaw stick on the RC transmitter to its bottommost and leftmost position

IV. Current Performance

The following results show estimates generated during flight using IMU information at 40 Hz and synthesized camera information at 2 Hz. Camera information is delayed before updating the EKF due to two reasons. First, we artificially added a delay of 0.1 seconds before sending MoCap data from the Ground Station to simulate image processing time. There is also a random communication delay over the wireless TCP/IP connection between the Ground Station and HexaKopter software. Using data from several flights we estimate this communication delay to have a mean of about 0.19 seconds and a standard deviation of 0.06 seconds. Thus, the delay of each camera data point was probably around 0.29 seconds, half of what it could have been for a 2 Hz update rate (assuming that the rate is obtained by the time needed to take a picture, transmit, process, and submit the pose data up to the estimator).

Because of the bug discussed in Section [13], we initially viewed the following results as very good. Discovering that bug clarified that we were only doing pseudo-vision updates at 2 Hz when we thought we were doing them at 10 Hz. However, in the course of writing this report, we uncovered another bug in our software that causes us now to put a little more salt on these results.

To synthesize camera data we use the information in the camera measurement uncertainty matrix to scale a standard normal random number generated. These random numbers provide the noise to add to MoCap data to produce the pseudo-camera measurements. The recently discovered bug was that we omitted taking the square root of the variances when scaling these random numbers. This meant that we were generating random numbers for position with a standard deviation of only 0.0004 meters instead of the 0.02 meters we expected. For heading angle, we were adding noise of only 0.0003 radians standard deviation instead of 0.017 radians.

TABLE I
Summary of the estimation error statistics.

State	Mean Error (m, rad, m/s)	Standard Deviation (m, rad, m/s)
n	0.0048	0.0371
e	0.0553	0.0703
d	-0.053	0.12
ϕ	-0.019	0.0154
θ	0.02	0.0111
ψ	-0.0003	0.0146
u	0.0102	0.0789
v	0.1133	0.1233
w	-0.1359	0.2765

Obviously, we will need to rerun these flight tests with the bugs fixed (see also Section V). The bug just discussed was found just 1 day before completing this report, and the HexaKopter is currently reconfigured to produce results for another effort. As soon as it is available again we will likely conduct new flight tests. For now, the following results are still worth discussing because they represent many other realistic conditions (e.g. communication delays, actual IMU and flight hardware, etc.).

The controller onboard the hexacopter was set to hover about the point 1.5, -0.5, -0.5 in north, east, and down coordinates for the following experiment. We did not make a great effort to tune the controller. We kept the gains conservative so that any spikes in the estimates would not cause drastic position changes for the hexacopter. Our goal was to fly without crashing, not necessarily to track paths with the lowest possible error. We are confident that significantly better performance will be achieved with better tuned gains. The total flight time was four and a half minutes, but only a segment of the flight is shown in figures 22 through 30 for clarity.

Table I summarizes the statistics of the estimation error that are mentioned in the captions of Figures 22 through 30.

Table II describes the error in the control in the different directions. As is illustrated by these values, there is work needed in tuning the controller. It is a delicate balance, however, since there are occasional spikes in the estimates and it is undesirable to have the controller respond quickly to these spikes.

These estimation results agree well with the offline estimator with synthesized camera updates at 2 Hz. After examining the code of the two estimators, it seems that with the bug of only using every fifth data packet eliminated, we should be

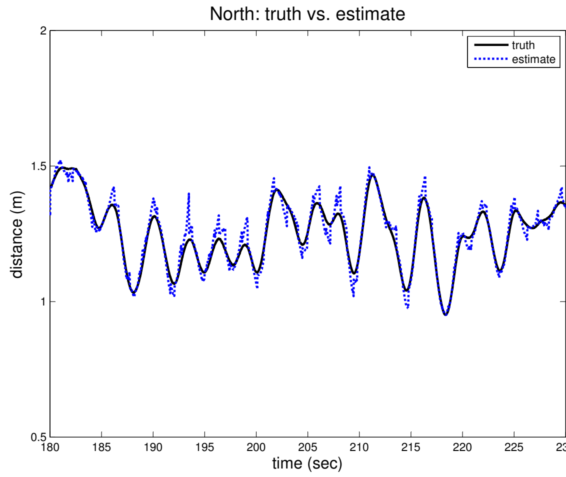

Fig. 22. The estimation error in north has a mean of 0.0048 m and a standard deviation of 0.0371 m for the whole flight.

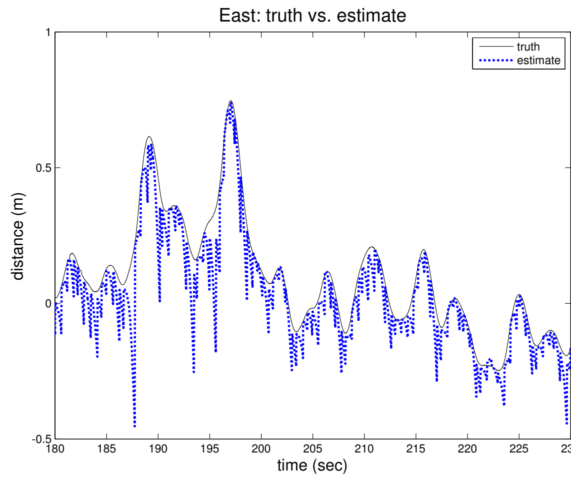

Fig. 23. East has an error characterized by a mean of 0.0553 m and a standard deviation of 0.0703 m for the flight. The estimation of east was poor due to a poor estimation of v, which is most likely due to a poor choice for the constant λ_1.

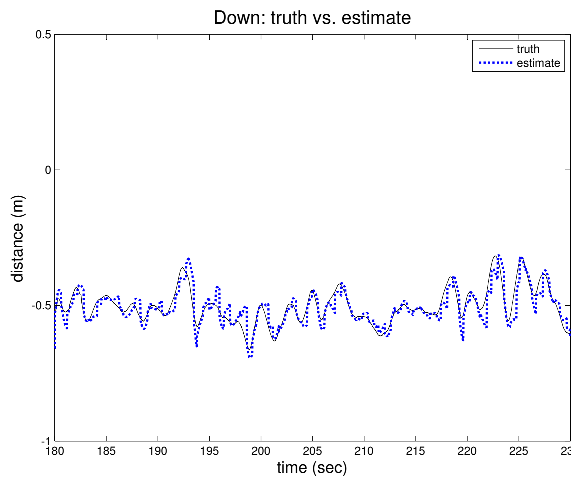

Fig. 24. The mean error is -0.053 m and standard deviation is 0.12 m. In the beginning of this flight, it took awhile for the estimates to settle onto the truth (not shown in figure).

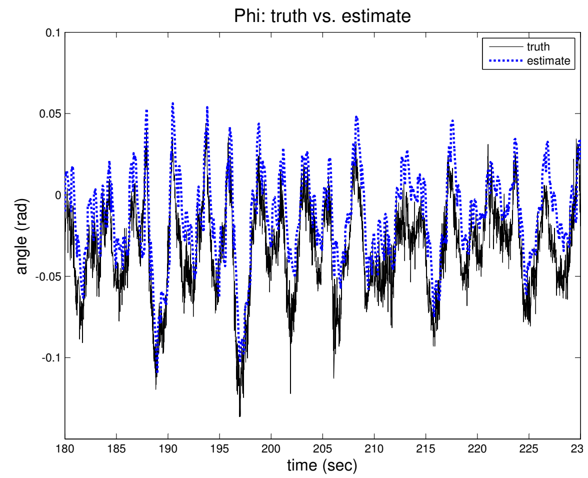

Fig. 25. As is seen in the figure, there is a bias in our estimate of ϕ: -0.019 rad and a standard deviation of 0.0154 rad.

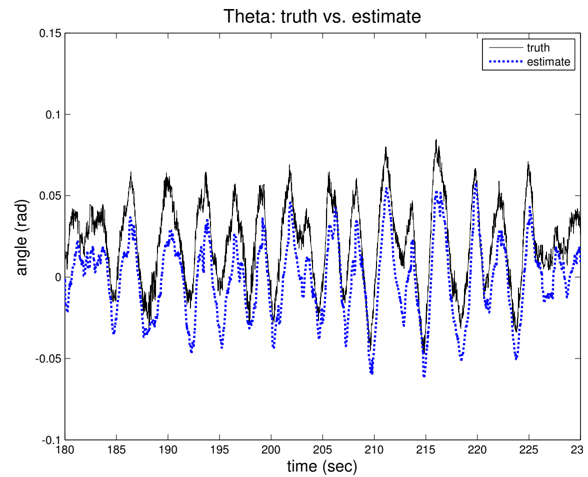

Fig. 26. The estimate of θ also has a bias, in the other direction: 0.02 rad, with a standard deviation of 0.0111 rad.

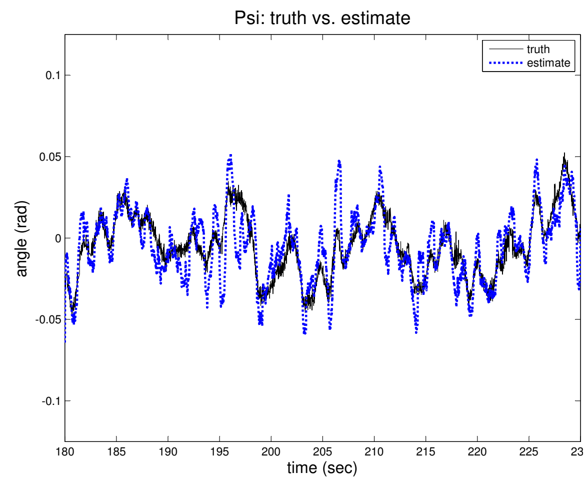

Fig. 27. Our estimate of ψ are very good, with a mean error of -0.0003 rad and a standard deviation of 0.0146 rad.

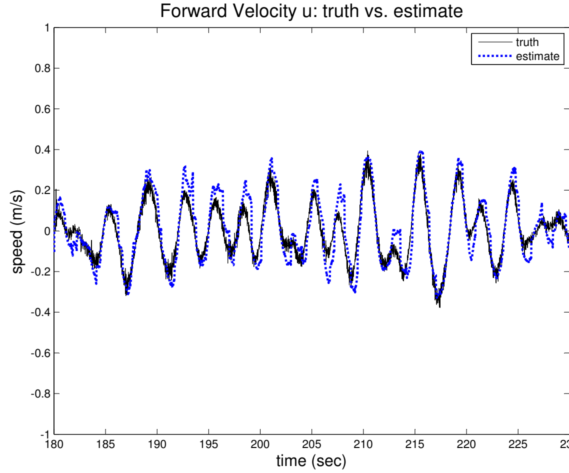

Fig. 28. Estimates of u are close. There is a mean error of 0.0102 m/s and standard deviation of 0.0789 m/s.

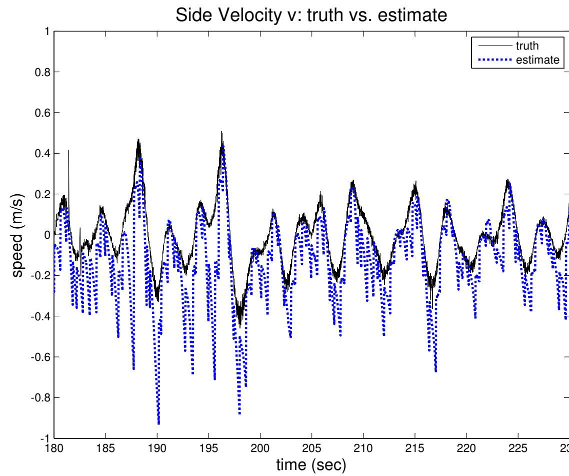

Fig. 29. Estimates of v struggle. We need to modify our constant for λ_1 in this direction. This should provide better estimates. Mean error for this flight is 0.1133 m/s and a standard deviation of 0.1233 m/s.

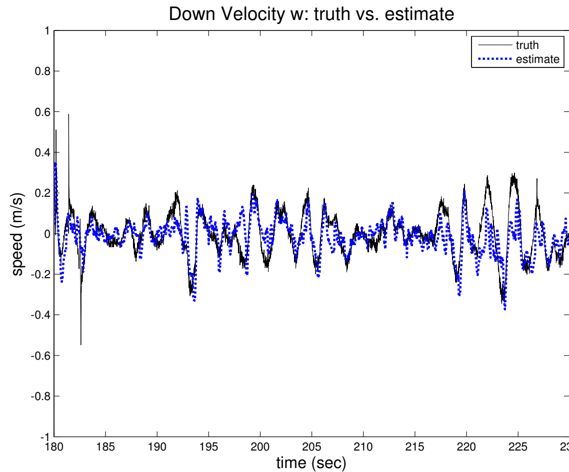

Fig. 30. Error bias is -0.1359 m/s and a standard deviation of 0.2765 m/s. In the beginning of the flight it took awhile for this estimate to settle onto the truth, at this point we are still unsure of why this was the case. That did not happen on the second hover flight we performed.

TABLE II
STATISTICS OF THE POSITION ERROR FROM CONTROL EFFORT.

Direction	**Mean (m, rad)**	**Standard Deviation (m, rad)**
North	0.2075	0.2413
East	-0.4482	0.2288
Down	0.0236	0.6935
ψ	0.0095	0.0308

able to expect the same performance in the online filter as we get with the offline filter. If we can get the real camera updates to 10 Hz or more, we should have excellent estimation performance. Table III describes some typical results that are achieved with camera updates occurring at 10 Hz. This data is from the same dataset as the online data above, with the same parameters ($\lambda_1 = 0.0001$ and $k_F = 0.00017$) used in both cases. The only difference is that the camera updates occur at 10 Hz.

TABLE III
STATISTICS OF ESTIMATION ERROR WITH 10 HZ CAMERA UPDATES.

State	**Mean (m,rad,m/s)**	**Standard Deviation (m,rad,m/s)**
n	-0.00002	0.0293
e	0.0028	0.0327
d	-0.0012	0.0171
ϕ	-0.03	0.0388
θ	0.0214	0.0355
ψ	-0.0001	0.0068
u	-0.0007	0.1819
v	0.0093	0.1952
w	0.0085	0.1159

Figures 31 through 34 show some updated results where an accelerometer bias was updated for the j_b direction (data from 6_10_1116). Not all the states are shown as they exhibit about the same results as those shown in figures 22 through 30. The IMU data was sent at a rate of 40 Hz and synthesized camera updates were completed at a rate of 2 Hz for this flight as well.

V. NEXT STEPS

While we are encouraged by our developments thus far, many improvements remain to be made. The following is a discussion of some ideas for near-term work.

As discussed in Section IV, we made a mistake in not taking the square root of the variance to scale the random numbers being added to MoCap data to synthesize camera measurements. This undoubtedly made our results better than they might have otherwise been. We need to rerun those flight tests with the more realistic camera measurements to determine the significance of this effect.

Cleaning up the code is the most immediate need to facilitate future development. The "feature" mentioned in Section III-B3 (which caused delayed camera updates to operate at a fifth of the anticipated rate) took a considerable number

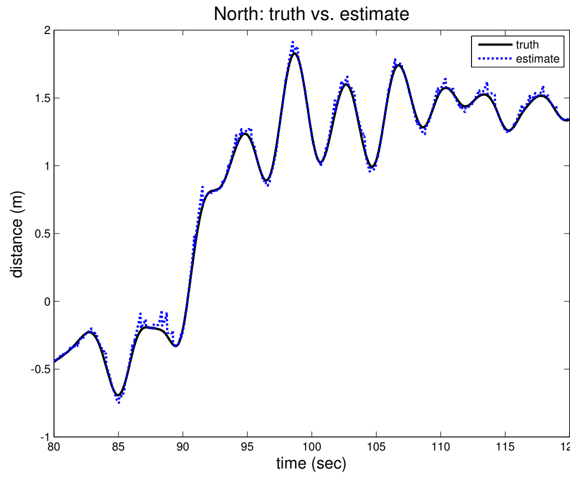

Fig. 31. Improved estimation in the North direction after adjusting an accelerometer bias for the body j_b direction. Note the scale changes from the previous North figure.

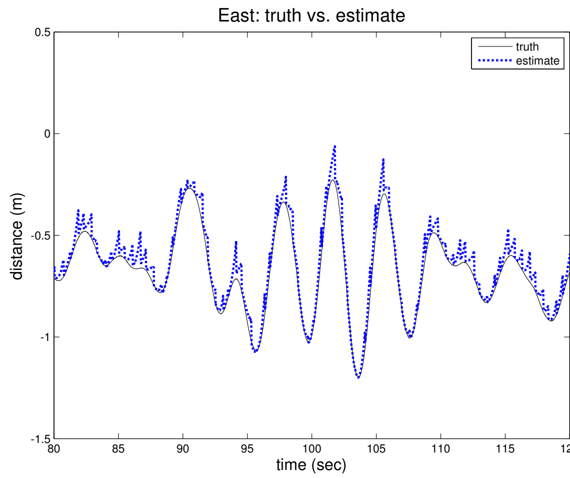

Fig. 32. Improved estimation for the East direction. The value for λ_1 was kept the same, but the lateral accelerometer bias was increased after it was observed to that the accelerometer had a bias during flight. Note the scale changes from the previous East figure.

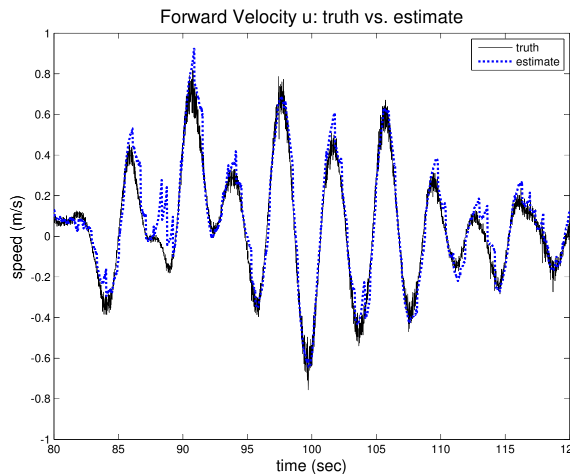

Fig. 33. Improved estimation in the body forward velocity, u after adjusting an accelerometer bias for the body j_b direction.

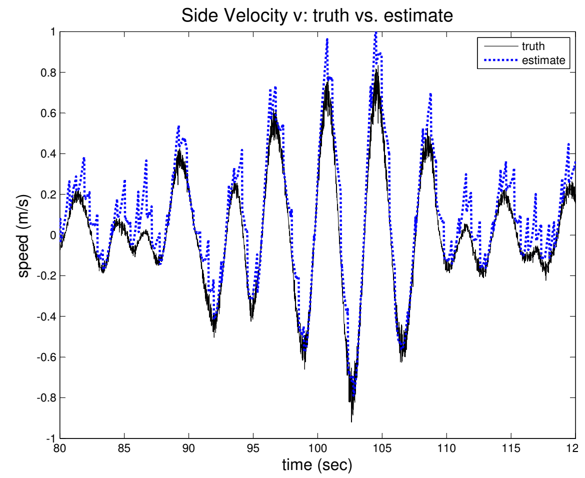

Fig. 34. Improved estimation for the body side velocity v. The value for λ_1 was kept the same, but the lateral accelerometer bias was increased after it was observed to that the accelerometer had a bias during flight.

of man hours to uncover, and a number of similar examples could be given of code elements no longer needed left over from earlier development. Section III-C1 also makes plain the fact that configuration parameters are currently spread throughout the code. Going forward, we need to remove elements of the code base that are no longer needed and move all configuration parameters into one place. We also need to change some variable names that no longer correspond to their actual function. These changes will make the code easier to understand, modify, and apply.

After cleaning up the code, but perhaps before making any other changes to the system, we would do well to pay a little more attention to the controller. We expect we can make the estimator track the true states rather accurately, yet our flight performance does not track the desired flight path as well as we probably could. This is likely just a matter of tuning the gains in the controller. Current results were obtained with soft gains that were primarily chosen to ensure the vehicle never moved too aggressively. However, we may need to take more involved steps to make sure the control is solid before moving forward much more with the estimation. We probably don't need anything especially innovative in our control, we just need it to work well.

Related to this last thought, we might need to dig a little deeper into the workings of the MikroKopter flight control code. Fortunately, there is an English rewrite of that firmware. We should verify if it is compatible with our version of the hardware. If it is, we may likely benefit during our polishing of the control by having a better bottom-up feel for what it is we're controlling. We may also find it easier in the English firmware to make other helpful changes (e.g. further speeding up the output of IMU data).

Also related to control improvements, we should probably more carefully redesign how we mount hardware on the HexaKopter. We have noticed that tuning gains to achieve good performance is easier when the hardware (onboard computer in particular) is not mounted. This is true even when using full

state feedback from the motion capture system in both cases. Of course this is not a surprising observation; the platform will be inherently more stable when the center of gravity is further down in the body-fixed reference frame. With some longer landing gear and a little forethought, we should be able to pretty painlessly improve our control performance simply by moving what we can below the plane of the motors.

Another important change to the code has to do with the way camera measurements are synthesized using MoCap data. Currently, the Ground Station software pushes MoCap data to the HexaKopter Software at a fixed rate set in the Ground Station. We would better model the real camera system if we have the HexaKopter Software notify the Ground Station when the onboard computer thinks a picture should have been taken. The Ground Station software should then go back into a queue of saved MoCap data and retrieve the value closest in time to when the picture was taken. The Ground Station can then send up the MoCap data to the HexaKopter after waiting a certain amount of time. This approach allows us to build in the same communication delays we will experience with the final system, and it also allows us to easily adjust how we model the time taken to process the image.

Synthesizing camera measurements is, of course, just a stepping stone to implementing the real camera-based algorithms which Stephen Quebe and John Macdonald began implementing earlier this calendar year. Their initial visual odometry algorithm needs to be polished up in preparation for integration with this baseline system. That polishing will likely include some of the same code cleanup as is needed in the baseline system software. We should also quantify the processing time needed for the various parts of that algorithm and take steps to minimize it where possible. We also need to better quantify the accuracy of the vision algorithm under various conditions (on a tripod vs. moving continuously, looking at feature rich vs. feature poor scenes, etc.). Quantifying better these aspects of the vision algorithm will inform our development of the baseline system that uses synthesized camera measurements. The same should also be true in reverse; that is, fine tuning the baseline system should inform our understanding of the performance needed from the vision algorithms and the conditions we can expect the camera to operate within.

Vision algorithm development would be facilitated by taking a dataset with the current baseline system that includes time-stamped images. This is something we can do in the immediate future.

Another, perhaps less essential, task could be the integration of a laser or infrared altimeter sensor into the baseline system. We specify laser or infrared here because the ultrasonic range sensor recently tested on the HexaKopter was unable to detect the floor during flight when greater than 14 inches away. We suspect this is due to the acoustic noise of the nearby motors. A height-above-ground sensor would make position down and down velocity more observable to the estimator. This may be useful since we have not as yet been able to use the z-axis accelerometer in our IMU-based updates to the EKF. The problem with the accelerometer is its measurements don't agree well with those predicted by the theory. Alternatively, we may do well to dig into the theory surrounding the z-axis accelerometer measurements and see if we can adjust the model to better explain what we get out of that sensor.

One final, longer-term idea for moving forward has to do with the rate at which camera measurements are made available to the EKF. Right now, we assume that the framerate of the synthesized camera is low enough to allow for sufficient time to process each image before the next image is taken. This makes the delayed updates to the EKF using camera data much simpler. However, if future tests suggest that we cannot meet this condition and still achieve good flight results, we may consider modifying the delayed update procedure.

VI. Conclusion

This concludes our attempts to document our work to date on solving the indoor navigation problem. The report itself is rather imperfect and rough around the edges, but then so is our current code and performance. Despite this, the process of writing the report has been very instructive. We've uncovered another important code bug in the writing process and generated a much more clear picture of the way forward for ourselves. While more might be done to polish this report for posterity, we will forgo that for the present time. We hope to prepare a superseding report, along with cleaner code and better results, sometime toward the end of this summer or early fall.

Acknowledgments

We've received tremendous amounts of help from fellow students (Jeff Ferrin most notably) and from our professors, Dr. McLain and Dr. Beard. The software and hardware underlying this system would not be possible without their support.

References

[1] M. Achtelik, A. Bachrach, R. He, S. Prentice, and N. Roy, "Stereo vision and laser odometry for autonomous helicopters in GPS-denied indoor environments," G. R. Gerhart, D. W. Gage, and C. M. Shoemaker, Eds., vol. 7332, no. 1. SPIE, 2009, p. 733219. [Online]. Available: http://link.aip.org/link/?PSI/7332/733219/1

[2] S. Ahrens, D. Levine, G. Andrews, and J. P. How, "Vision-based guidance and control of a hovering vehicle in unknown, GPS-denied environments," in *Proc. IEEE Int. Conf. Robotics and Automation ICRA '09*, 2009, pp. 2643–2648.

[3] A. Bachrach, R. He, and N. Roy, "Autonomous flight in unstructured and unknown indoor environments," in *Proceedings of the EMAV Conference*. European Micro Air Vechicle, September 2009.

[4] R. Beard and T. McLain, *Small Unmanned Aircraft*. Princeton University Press, 2011.

[5] M. Blosch, S. Weiss, D. Scaramuzza, and R. Siegwart, "Vision based MAV navigation in unknown and unstructured environments," in *Proc. IEEE Int Robotics and Automation (ICRA) Conf*, 2010, pp. 21–28.

[6] M. Fliess, J. Levine, P. Martin, and P. Rouchon, "Flatness and defect of nonlinear systems: Introductory theory and examples," CAS, Tech. Rep. A-284, January 1994.

[7] S. Grzonka, G. Grisetti, and W. Burgard, "Towards a navigation system for autonomous indoor flying," in *Proc. IEEE Int. Conf. Robotics and Automation ICRA '09*, 2009, pp. 2878–2883.

[8] P. Martin, R. Murray, and P. Rouchon, "Flat systems," 1997.

[9] P. Martin and E. Salaun, "The true role of accelerometer feedback in quadrotor control," in *Proc. IEEE Int Robotics and Automation (ICRA) Conf*, 2010, pp. 1623–1629.

[10] N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar, "The GRASP multiple micro-UAV testbed," *IEEE Robotics & Automation Magazine*, vol. 17, no. 3, pp. 56–65, 2010.

[11] Mikrokopter.us, "http://www.mikrokopter.us/index.php."

[12] R. Murray, M. Rathinam, and W. Sluis, “Differential flatness of mechanical control systems: A catalog of prototype systems,” in *Int'l Mech Eng Congress and Expo*. ASME, Novermber 1995.
[13] Quadrocopter, “http://www.quadrocopter.us/index.php.”