Final Report

Harvey Mudd College Engineering Clinic

Project Team
Austin Chun (Spring)
Robert Cyprus
Aishvarya Korde (Spring TL)
Zayra Lobo (Fall)
Paige Rinnert (Fall TL)
Jesus Villegas (Spring)

Project Advisor: Professor Christopher Clark
Project Liaisons: Ken Dreshfield ‘80
Ian Jimenez ‘11
Ron P. Smith ‘83
Stephanie Tsuei (Lead)

May 2017

ABSTRACT

The goal of this project is to implement a secure state estimation (SSE) algorithm in software and in hardware on a quadrotor. The simulation results show that the quadrotor dynamic model produces outputs that agree with expectations based on the motor inputs, and the SSE is capable of producing accurate state estimates (within 10% error) of a 10 degree of freedom (DOF) system with noisy outputs. The hardware implementation involved integrating a Jetson TK1 computer with two laser scanners, four 9-DOF inertial measurement units (IMUs), and a flight controller. The Jetson TK1 runs a ROS environment to gather sensor data, run the SSE, and communicate to the flight controller. Sensor data was also logged from test flights for testing the SSE offboard in order to confirm onboard flight results.

TABLE OF CONTENTS

1. INTRODUCTION

Northrop Grumman (NG) sponsored a 2016-2017 Clinic Project to develop a secure state estimation system that can circumvent sensor attacks on autonomous vehicles, specifically aircraft vehicles. This section contains a description of NG, presents the project statement, defines the deliverables for the project, and outlines the current project status at the end of the Fall semester.

1.1. NORTHROP GRUMMAN

NG is a leading global security company that innovates in the fields of autonomous vehicles, cyber security and communications, and air and space systems. NG is committed to preserving freedom and advancing human discovery. This Clinic Project is sponsored by NG's Aerospace Systems sector (NGAS). Based in Redondo Beach, CA, NGAS provides next-generation solutions for military aircraft and land vehicles, autonomous systems, and space systems for its customers worldwide. NG holds world records in many fields, with technology integrated in products ranging from cell phones to space satellites. NGAS's Research and Development department is sponsoring this project to implement a secure state estimation system for autonomous aircraft.

1.2. PROJECT STATEMENT

The goal of this project is to implement a secure state estimator (SSE) in software and then experimentally validate the estimator in hardware on a quadrotor. The addition of the SSE algorithm to the control loop should reduce the real-time tracking error of a quadrotor when using measurements from compromised onboard sensors.

1.2.1. Objectives

- Demonstrate quadrotor altitude tracking control with minimal tracking error [m]
- Maximize the number of compromised sensors for which the SSE can be tested
- Minimize cost

1.2.2. Constraints

- SSE must reduce tracking error when sensors are compromised, as seen in Table 1.

	w/o SSE	with SSE
No compromised sensors	e^*	$\leq 1.2e^*$
Attack	$> 5e^*$	$\leq 1.2e^*$

Table 1 | Specifications for a successful SSE

The error e^* that appears in Table 1 is defined as the tracking error when no sensors are compromised, which depends on the specifics of the flight hardware and controller. Adding the SSE to the system may increase the error slightly but this error should not exceed 1.2 times the original error. The project goal is to demonstrate that when compromised sensors result in a tracking error of more than 5 times the original error, the SSE keeps the error bounded within 1.2 times the original error, which is the same range as when all sensors are functioning normally.

- Final implementation of the SSE must be in ROS
- Quadrotor must fly for 10 minutes on a single battery charge
- Must estimate states at a frequency of at least 25 Hz
- All equipment and supplies must cost less than $7,000

1.2.3. Functions

- Quadrotor should complete pre-planned altitude profile with compromised sensors
- Quadrotor should perform real time state estimation and correction
- The SSE must be configurable to simulate compromised sensors
- The SSE should estimate the state of the quadrotor (position [m], velocity [m/sec], orientation [°], angular velocity [°/sec])

1.3. DELIVERABLES

The goals of this project include:

- Implement and simulate a state estimator in MATLAB with
    - All sensors functioning.
    - Up to 100% of sensors compromised.
- Successfully log sensor data.
- Produce a functioning quadrotor with onboard computer, controller, and sensors.
- Project documentation and presentations including:
    - Work Plan
    - Midyear Report
    - Design Review at Northrop Grumman
    - Three internal Clinic presentations
- Test the state estimator on real sensor data with
    - Up to 25% of measurements corrupted.
    - Offline and in real time.
- Demonstrate quadrotor flight with real time data logging and state estimation.
- Project documentation and presentations including:

- Final Report (including engineering drawings)
- Spring semester presentation at HMC
- Projects Day presentation
- Final Presentation at Northrop Grumman

1.4. IMPACT

This project is non-proprietary, which means that the technology produced will be owned by the team instead of becoming the property of Northrop Grumman. Such ownership provides significantly more control over the distribution of the project than if the results were owned by Northrop Grumman.

Northrop Grumman, a defense and aerospace company, hopes to use the results of the project to improve military drones for the United States government, and the project results could also be published to benefit quadrotors in non-military applications. Since the overall goal of the project is to improve the state estimation of quadrotors, the success of the project could inform observer design for quadrotors in popular applications, including wildlife monitoring and structural testing [1]. If the results from this project are published, quadrotors used for commercial delivery services, photography, mapmaking, lifeguarding, and other such activities could benefit from improved state estimation techniques, making these devices more robust and less prone to failure while in use.

1.5. REPORT OVERVIEW

Section 2 outlines the background research on secure state estimation. Section 3 discusses the design of the SSE in simulation and the transition to hardware. Section 4 contains the simulation results of the dynamic model and the SSE algorithm. Section 5

provides an overview of the hardware implementation, including the selected sensors, onboard computer, flight controller, quadrotor frame and motors, RC transmitter and receiver, and power components. The hardware results of the SSE are reported in Section 6, and the report concludes with an overview of results and completed objectives for this project in Section 0.

2. BACKGROUND

Autonomous quadrotors are beginning to be used in a wide variety of applications from product delivery to agricultural maintenance and humanitarian aid. What each of these diverse environments has in common is that the autonomous vehicle must have a robust control system to properly follow the user's instructions. Most control systems use feedback control, in which the controller uses measurements of the system to calculate necessary motor speed adjustments to track desired trajectories [8].

However, in many systems, the system outputs cannot be measured directly. This problem motivates the need for a state estimator, an algorithm that uses motor inputs and sensor measurements to provide an estimate of the states of the system [4]. These state estimates are input to the control system instead of the actual, unmeasurable, system states. Yet, a controller that relies on estimated states will only be as good as the estimates. Therefore, more accurate state estimates will ensure better path tracking, and inaccurate state estimates might create an unstable system.

Standard state estimation techniques, such as [8], are often sufficient when all sensors are functioning as expected, but these methods break down when sensors are compromised or broken. If a sensor starts to output incorrect values, a traditional state estimator may not produce an accurate estimate of the system states. If these incorrect states are fed back into the controller, it is likely that the vehicle will follow the wrong flight path and could crash into the ground or another vehicle.

To decrease the possibility that a vehicle will behave in an unexpected or dangerous way, the focus of the project is to implement a *secure* state estimator that accurately

measures the states of a quadrotor in the presence of cyber-physical attacks that compromise sensor outputs. The category of cyber-physical attacks includes both cyber-attacks, such as viruses introduced to sensors or sensor signals intercepted and altered by a hacker, and physical assaults. Both cyber and physical attacks that alter sensor outputs could cause a state estimator to fail to produce accurate estimates. These types of attacks can affect both the behavior of the control system and the actuators that control the motors. The scope of this project focuses on sensor attacks, in the form of broken, compromised, or noisy sensor outputs.

Convex optimization is a common technique employed to make secure state estimators robust against attacks (another option is satisfiability modulo theory, but only convex optimization will be discussed in detail) [4]. Convex optimization includes a special class of nonlinear optimization solvers, such as least squares programming (LP) and quadratic programming (QP) [2]. Convex optimization is used in a wide array of fields, including control, circuit design, economics, and machine learning, and thus there exist many toolkits with preprogrammed optimization solvers, such as MATLAB [2]. Recent advances have drastically reduced the time required for convex optimization solvers to generate solutions, thus making real time solver applications in hardware a possibility (like flying a quadrotor) [2]. The project goal is to implement a pre-existing SSE algorithm, only previously tested in simulation, and validate the algorithm in hardware. Improving the existing SSE algorithm or developing a new one are not objectives for this project.

2.1. LITERATURE REVIEW

An important step in some robust state estimators is detecting if any of the sensors are not working properly. Compromised sensors can be identified by comparing sensor measurements to an analytical model using the residual signal of the system [4], [5]. Although this method works in some contexts, Fawzi et al. claim it is not effective for large errors in sensor measurements and only works for bounded disturbances. Robust control also assumes that the disturbances are bounded, but this assumption may not be valid in the context of security [4]. Disturbances injected by a malicious agent may not be bounded, so a model of bounded disturbances is not a realistic representation of real world disturbances. Filters have been introduced to detect and identify attacks, but these filters are difficult to implement and computationally expensive because they are based on a graph-theoretic characterization of the system vulnerability [4].

One approach to secure state estimation found in the literature is secure state reconstruction, where the goal is to reconstruct the state of the system from functional sensor outputs. Shoukry et al. [5] focused on reconstructing the state of a nonlinear system given that some sensors are compromised. They posed the state reconstruction problem as a Boolean formula and developed a lazy SMT (Satisfiability Modulo Theory) procedure to solve this problem that uses an SAT (abbreviation of satisfiability) solver as a subroutine [5].

According to Lee et al., there are three main approaches to the problem of error correction: geometric, greedy, and combinatorial [6]. In the geometric approach, linear programming techniques are used to formulate the problem as a convex optimization problem (this is the approach used in [4]). The greedy approach uses an iterative process to approximate signal coefficients, and the combinatorial approach identifies different

combinations of parameters that could produce the measured sensor outputs to restrict the solution space. Lee et al. take the combinatorial approach to reduce computational intensity of the optimization problem. To create a secure state estimator, they consider a separate observer for each output and combine the state estimates from each observer [6]. Mishra et al. also use the combinatorial approach to produce a state estimator by looking at the residuals of all possible sets of sensors with the size of the number of sensors known to be functional [7]. This approach, like many of the models discussed, assumes an upper bound on the number of sensors that are compromised. The convex optimization algorithm implemented in this project is described in Section 3.3.

3. SSE DESIGN

This section details the process of designing the secure state estimation algorithm, including modeling quadrotor dynamics, selecting and implementing the secure state estimation algorithm, setting up a control feedback simulation of the quadrotor system without the SSE, and testing the SSE on both a simple system and a simulated quadrotor system. All simulations were implemented in MATLAB. Figure 1 contains the high-level control diagram of the system, which includes the following: the actuator, which is the Pixhawk flight controller; the sensors that measure the system state; and the Jetson TK1, which is the onboard computer that houses the SSE.

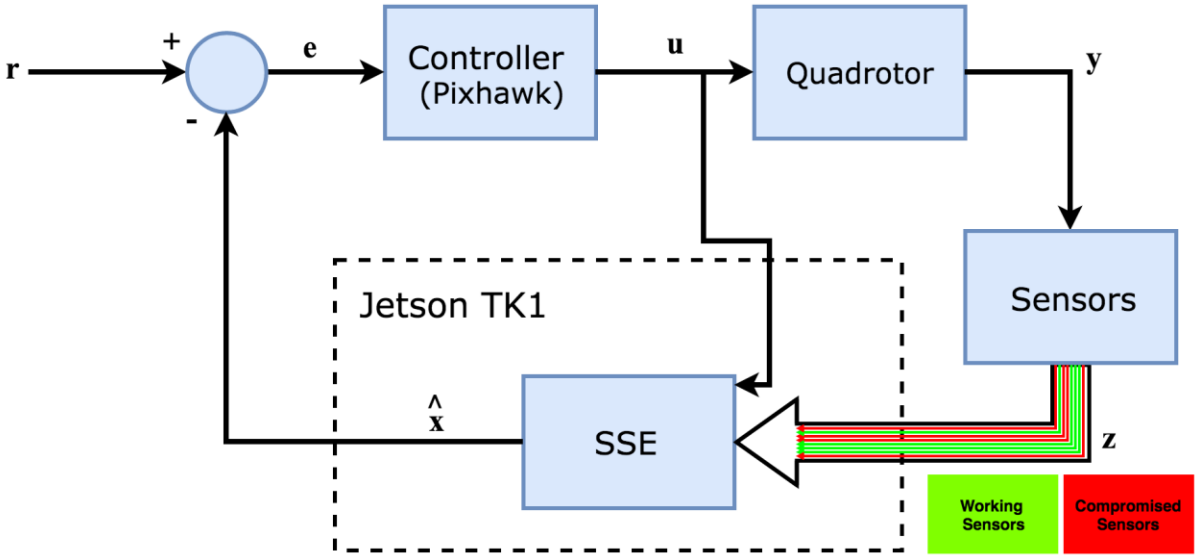


Figure 1 | High-Level Control Diagram

In Figure 1, r is the reference input (desired height) and e is the error between the estimated height and desired height, which is input to the controller.

3.1. STATES OF THE QUADROTOR

To control a quadrotor, the relevant states are those that are necessary to track a flight path and perform autonomous navigation. The initial proposed states are included in Table 2.

State	Description	Variable names
Position	Each of x, y, and z (in meters)	x, y, z
Velocity	Each of X, Y, and Z time derivatives (m/s)	v_x, v_y, v_z
Orientation	Each of Roll, Pitch, and Yaw	θ, ϕ, ψ
Angular Velocity	Each of Roll, Pitch, and Yaw time derivatives	$\omega_x, \omega_y, \omega_z$

Table 2 | Quadrotor states and variable names

All states are taken with reference to the global frame, with the global origin for x, y, and z position defined as the take-off location, as shown in Figure 2.

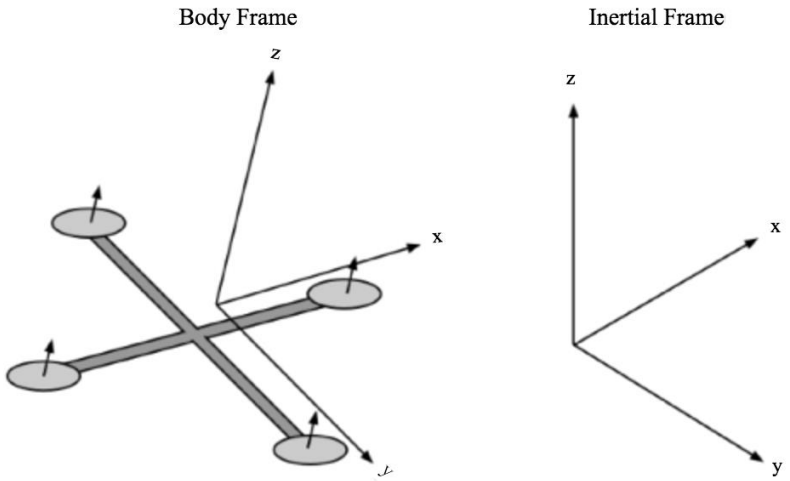


Figure 2 | Quadrotor body frame and inertial frame [8]

The quadrotor will only track an altitude flight path so the z position (height) is the primary state of interest, and the x and y position states will be set to zero. The remaining ten states (z position; x, y, and z velocity; roll, pitch, and yaw; and angular velocity in the X, Y, and Z

directions) will be used for control. If a three-dimensional flight path is implemented, the x and y position states will be added back into the control loop with the necessary sensor additions.

3.2. DYNAMIC MODEL

The equations of motion of a system are crucial to describe the system dynamics and understand the behavior of the system. Since quadrotor dynamics is a well-studied problem, a quadrotor model was adapted from previous research [8]. Quadrotor control depends on 6 degrees of freedom – 3 translational and 3 rotational, with an input from 4 motors. The dynamics can be derived using a rotation matrix to translate angles from the body frame of the quadrotor to the inertial frame (see Figure 2 in the previous section). The physical properties of the quadrotor system govern the equations of motion. The dynamic model takes into account the torque produced by the motors, conservation of energy, and the thrust acting on the quadrotor, shown in Figure 3 [8]. The forces of gravity and drag on the quadrotor are ignored in this diagram.

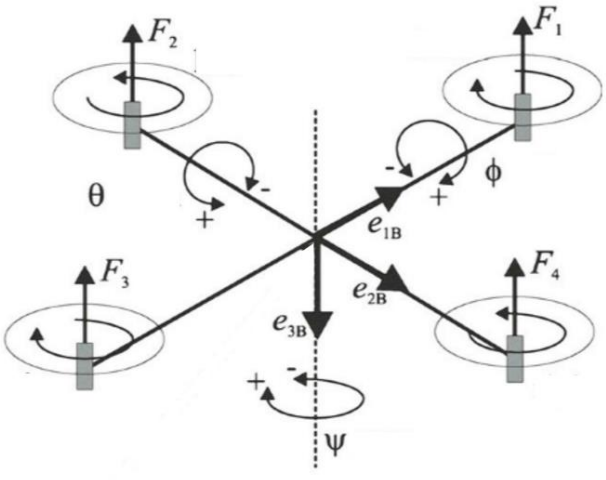


Figure 3 | Forces acting on the quadrotor [9]

The resulting governing equations are represented using state space notation in Equations (1) through (4) [8].

$$\dot{\boldsymbol{x}} = \boldsymbol{v} \tag{1}$$

$$\dot{\boldsymbol{v}} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{1}{m}\boldsymbol{R}\boldsymbol{T_B} + \frac{1}{m}\boldsymbol{F_D} \tag{2}$$

$$\dot{\boldsymbol{\theta}} = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta s_\phi \end{bmatrix} \boldsymbol{\omega} \tag{3}$$

$$\dot{\boldsymbol{\omega}} = \begin{bmatrix} \tau_\phi I_{xx}^{-1} \\ \tau_\theta I_{yy}^{-1} \\ \tau_\psi I_{zz}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy} - I_{zz}}{I_{xx}} \omega_y \omega_z \\ \frac{I_{zz} - I_{xx}}{I_{yy}} \omega_x \omega_z \\ \frac{I_{xx} - I_{yy}}{I_{zz}} \omega_x \omega_y \end{bmatrix} \tag{4}$$

where $\boldsymbol{x} = [x \quad y \quad z]'$ is the position in space; $\boldsymbol{v} = [v_x \quad v_y \quad v_z]'$ is the linear velocity in the x, y, and z directions; $\boldsymbol{\theta} = [\theta \quad \phi \quad \psi]'$ is the roll, pitch and yaw angles, $\boldsymbol{\omega} = [\omega_x \quad \omega_y \quad \omega_z]'$ is the angular velocity about the X, Y, and Z axes, $\boldsymbol{R}$ is the rotation matrix for converting body frame to global frame; $\boldsymbol{T_B}$ is the thrust force vector; $\boldsymbol{F_D}$ is the drag force vector; $I_{xx}, I_{yy}, and\ I_{zz}$ are the moments of inertia. Using the state space representation of the governing equations, the motion of the quadrotor was simulated as a function of various input conditions.

A simulation environment was created to test the results for different inputs to the quadrotor system and was adapted from the simulation code implemented by the author who wrote the quadrotor simulation previously referenced [8]. The equations of motion were written in a MATLAB script that includes functions to compute the forces, torques, and

accelerations given the system constants. The simulation input vector represents the voltage values of each of the four motors as an input. The output of the simulation is a data struct containing vectors of position, orientation, velocity, and angular velocity as a function of time. The results of this simulation can be used to visualize the state of the quadrotor based on the dynamic model of the system. The dynamic model simulation results are discussed in Section 4.1 and the code can be found in Appendix 9.1.

3.2.1. Linearized Dynamic Model

To create a linearized model of the quadrotor system, the equations must be rewritten in terms of state variables and inputs, seen in Equations (5) through (14):

$$\dot{x}_3 = \mathrm{x}_6 \tag{5}$$

$$\dot{x}_4 = -\frac{k_d}{m}\mathrm{x}_4 + \frac{1}{m}\sin x_8 \sin x_9 \,(\mathrm{u}_1 + \mathrm{u}_2 + \mathrm{u}_3 + \mathrm{u}_4) \tag{6}$$

$$\dot{x}_5 = -\frac{k_d}{m}\mathrm{x}_5 - \frac{1}{m}\cos x_9 \sin x_8 \,(\mathrm{u}_1 + \mathrm{u}_2 + \mathrm{u}_3 + \mathrm{u}_4) \tag{7}$$

$$\dot{x}_6 = -\frac{k_d}{m}\mathrm{x}_6 + \frac{1}{m}\cos x_8 \,(\mathrm{u}_1 + \mathrm{u}_2 + \mathrm{u}_3 + \mathrm{u}_4 + \mathrm{u}_5) \tag{8}$$

$$\dot{x}_7 = \mathrm{x}_{10} + \frac{\sin x_8}{\sin x_7 \,(\cos x_8 + \sin x_7)}\mathrm{x}_{11} + \frac{\tan x_8}{\cos x_8 + \sin x_7}\mathrm{x}_{12} \tag{9}$$

$$\dot{x}_8 = \frac{1}{\cos x_8 + \sin x_7}\mathrm{x}_{11} - \frac{1}{\cos x_8 + \sin x_7}\mathrm{x}_{12} \tag{10}$$

$$\dot{x}_9 = \frac{1}{\cos x_8 \,(\cos x_8 + \sin x_7)}\mathrm{x}_{11} + \frac{1}{\cos x_8 \tan x_7 \,(\cos x_8 + \sin x_7)}\mathrm{x}_{12} \tag{11}$$

$$\dot{x}_{10} = \frac{I_{yy} - I_{zz}}{I_{xx}}\mathrm{x}_{11}\mathrm{x}_{12} + \mathrm{I}_{\mathrm{xx}}^{-1}\mathrm{Lk}(\mathrm{u}_1 - \mathrm{u}_3) \tag{12}$$

$$\dot{x}_{11} = \frac{I_{zz} - I_{xx}}{I_{yy}}\mathrm{x}_{10}\mathrm{x}_{12} + \mathrm{I}_{\mathrm{yy}}^{-1}\mathrm{Lk}(\mathrm{u}_2 - \mathrm{u}_4) \tag{13}$$

$$\dot{x}_{12} = \frac{I_{xx} - I_{yy}}{I_{zz}}\mathrm{x}_{10}\mathrm{x}_{11} + \mathrm{I}_{\mathrm{zz}}^{-1}\mathrm{b}(\mathrm{u}_1 - \mathrm{u}_2 + \mathrm{u}_3 - \mathrm{u}_4) \tag{14}$$

where the vector $[x_3 \cdots x_{12}]'$ corresponds to $[z\ v_x\ y_y\ v_z\ \theta\ \phi\ \psi\ \omega_x\ \omega_y\ \omega_z]'$ (height; velocity in x, y, and z; orientation in x, y, and z; and angular velocity in x, y, and z) and $[u_1\ u_2\ u_3\ u_4\ u_5]'$ corresponds to the control inputs on the four motors and gravity.

To linearize the system, the terms containing $u_1 \cdots u_5$ must be separated from the terms containing only state variables $x_3 \cdots x_{12}$. The system matrices A and C can be found by calculating the Jacobian of the state variables and inputs respectively, as shown in Equation (15). (Note that the Jacobian is a matrix that contains partial derivatives of each equation of motion with respect to the state variables or inputs.)

$$A = \begin{bmatrix} \frac{\partial \dot{x}_3}{\partial x_3} & \cdots & \frac{\partial \dot{x}_3}{\partial x_{12}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \dot{x}_{12}}{\partial x_3} & \cdots & \frac{\partial \dot{x}_{12}}{\partial x_{12}} \end{bmatrix}, \quad B = \begin{bmatrix} \frac{\partial \dot{x}_3}{\partial u_1} & \cdots & \frac{\partial \dot{x}_3}{\partial u_5} \\ \vdots & \ddots & \vdots \\ \frac{\partial \dot{x}_{12}}{\partial u_1} & \cdots & \frac{\partial \dot{x}_{12}}{\partial u_5} \end{bmatrix} \tag{15}$$

Evaluating both Jacobian matrices at a desired fixed point yields the state matrix A (size 10 by 10) and input matrix B (size 10 by 5) matrices of the state space system, which are shown in Equation (16), the first of two state space equations.

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{16}$$

$$y(t) = Cx(t) + Du(t) \tag{17}$$

The output matrix C (size 38 by 10) in equation (17) must have a nullity of zero for the system to be observable (the observability matrix $[C \quad CA \quad \cdots \quad CA^{T-1}]'$ must have full rank). For our system, the matrix C is shown in Equation (18), with the bottom section repeated 3 more times:

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{18}$$

The feedforward matrix D is zero because the inputs do not feed directly back to the states. The system matrices A through D are then converted to discrete time and can be fed into the state estimator.

3.3. SSE ALGORITHM

Two secure state estimation approaches were considered for this project, based on literature suggested by the liaisons. The first algorithm uses concepts from compressed sensing and error correction over the reals to pose the estimation problem as a convex optimization problem [4]. The second algorithm uses Satisfiability Modulo Theory to perform state estimation on nonlinear systems [5]. The convex optimization algorithm was selected for this project because of familiarity with state space control and the liaisons' experience with CVX, a convex optimization solver that is compatible with MATLAB [3] (see Appendix 9.2). This secure state estimation algorithm can be presented in Equation (19) as a convex optimization problem:

$$D_{1,r}^{T}(y^{(0)}, \ldots, y^{(T-1)}) = \underset{x \in \mathbb{R}^n}{\operatorname{argmin}} \|Y^{(T)} - \Phi^{(T)} x\|_{\ell_1/\ell_r} \tag{19}$$

where $D_{1,r}^{T}$ is a decoder that determines the optimal initial state of the system by minimizing the error between actual sensor outputs and predicted outputs based on a model of the system. The two inputs to the decoder are the previous sensor outputs, represented by the matrix $Y^{(T)}$, and predicted outputs based on a model of the system, represented by the

quantity $\Phi^{(T)}x$. The $Y^{(T)}$ matrix contains the matrix of outputs $\left[y^{(0)} \mid y^{(1)} \mid \cdots \mid y^{(T-1)}\right]$ (number of time steps T by number of sensors) where each column represents the sensor outputs at a given timestep. $\Phi^{(T)}x$ transforms x to $\left[Cx \mid CAx \mid \cdots \mid CA^{(T-1)}x\right]$ using the matrix parameters A and C of the linearized model of the system in state space, as described in Section 3.2.1. The expression *argmin* for a vector x of real values denotes the states x for which the argument is minimized, and the notation $\|M\|_{\ell_1/\ell_r}$ is the sum of the ℓ_r norms (also known as a p-norm) of the rows of matrix M [4].

Although they are not explicit in the formulation of the optimization problem, the motor inputs must be included in the SSE algorithm. In Equation (20) shows how the system evolves based on the system parameters and motor inputs. The vectors $y[t]$ and $u[t]$ represent the actual sensor outputs and motor inputs at time t, and $x[0]$ represents the initial state of the system when $t = 0$.

$$\begin{aligned} y[0] &= Cx[0] \\ y[1] &= CAx[0] + CBu[0] \\ y[2] &= CA^2x[0] + CABu[0] + CBu[1] \\ &\vdots \\ y[t] &= CA^tx[0] + CA^{t-1}Bu[0] + CA^{t-2}Bu[1] + \cdots + CBu[t-1] \end{aligned} \tag{20}$$

This information can be rearranged into a form that fits the structure of the SSE algorithm in Equation (21):

$$\begin{aligned} y[0] - Cx[0] &= 0 \\ (y[1] - CBu[0]) - CAx[0] &= 0 \\ (y[2] - CABu[0] - CBu[1]) - CA^2x[0] &= 0 \\ &\vdots \\ (y[t] - CA^{t-1}Bu[0] - CA^{t-2}Bu[1] \cdots - CBu[t-1]) - CA^tx[0] &= 0 \end{aligned} \tag{21}$$

In Equation (21), the terms in parentheses can be grouped into the $Y^{(T)}$ matrix in the optimization algorithm. The optimization solver calculates the initial state $x[0]$, and the linear dynamics are propagated as shown in Equation (16) to obtain $x[t]$. The SSE was tested in MATLAB simulations on both a simple single degree of freedom system and on a full quadrotor model, and the results can be found in Section 4.2. The MATLAB simulation code including the SSE algorithm is included in Appendix 9.2.

3.4. SSE TRANSITION TO HARDWARE

While simulations of the SSE are feasible in MATLAB because there are no constraints on the solver speed, MATLAB cannot be used on the quadrotor because it is too slow for real time data collection and processing. The team considered two options for implementing the SSE on the quadrotor, both of which involved translating the algorithm from MATLAB into C. The first option is CVXgen, a language that allows the user to formulate a mathematical problem in an online interface and generate solver code in C [13]. The second option is to write an optimization solver in a MATLAB s-function for Simulink code generation. These two options present a trade-off between flexibility and ease of use: the CVXgen interface has limited functionality but eliminates the need to write an optimization solver in C; writing our own solver affords flexibility in the implementation but is more complicated than using CVXgen. The team decided to implement the SSE in CVXgen.

3.4.1. Solver Implementation

To generate the algorithm in CVXgen, the matrices that describe the quadrotor system in state space were assumed to already have been generated. These matrices are the

system parameters A, B, and C that appear in the linearized model of the system (see Section 3.2.1, Equations (16) – (17)**Error! Reference source not found.**).

The implementation of the algorithm in MATLAB used the full ℓ_1/ℓ_r norm as shown in Section 3.3, where the SSE takes the r-norm (a Euclidean 2-norm in our implementation) of each sensor across all timesteps and then minimizes the sum of the absolute value of these norms. However, CVXgen does not have 2-norm functionality; the only options are the 1-norm and infinity norm. The first iteration of the CVXgen SSE mimicked the algorithm in MATLAB with a 1-norm instead of a 2-norm (see Appendix 9.3.1), but was too complex for the solver. Taking the norm of measurements from each sensor and then summing these values required introducing an additional matrix that significantly increased the complexity of the problem and made it unrealistic.

The next iteration eliminated the norm entirely and simply summed absolute values (see Appendix 9.3.2). This simplification did not functionally change the problem because a 1-norm is the same operation as taking the absolute value and summing. This implementation compiled successfully for small values (all matrices were 10 by 10 or smaller), but for our system parameters the problem was too large to be solved (the quadrotor has 38 sensors that map to 10 states).

To reduce the problem size, we implemented sparsity in CVXgen, which allows the user to tell the solver which entries in a matrix are nonzero (see Appendix 9.3.3). Since the linearized system matrices are mainly zeros, specifying sparsity did decrease the problem size, but not far enough.

By leveraging the fact that the linear states (position and velocities) are decoupled from the orientation states (angles and angular velocities) in the linearized system matrices,

the SSE can be split into separate optimization solvers for the position and orientation states as shown in Figure 4 (see Appendix 9.3.4). The final CVXgen implementation is comprised of two SSEs, each with sparsity specified.

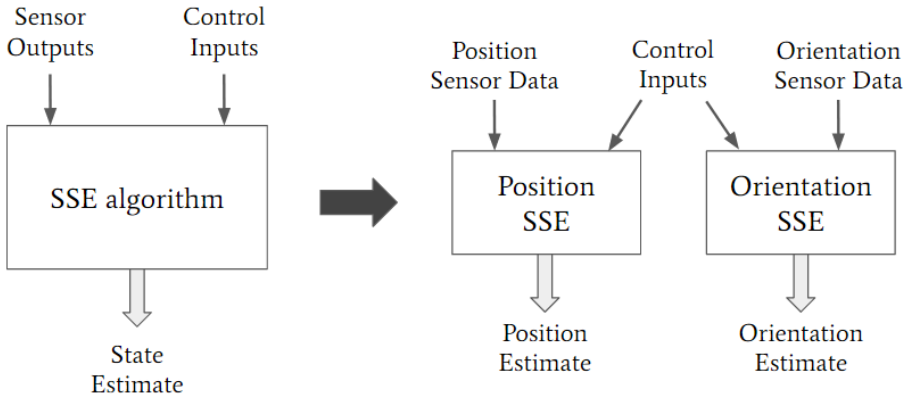


Figure 4 | Decoupled SSE algorithms

The position state vector is given by $[z\ v_x\ y_y\ v_z]'$ (height; velocity in x, y, and z) and the orientation state vector is given by $[\theta\ \phi\ \psi\ \omega_x\ \omega_y\ \omega_z]'$ (orientation in x, y, and z; and angular velocity in x, y, and z).

CVXgen produces a set of C files that can be compiled into an executable without modification. However, the team needed to implement input/output functions to read/write text files and matrix helper functions to manipulate the system matrices, sensor data, and inputs into the correct form for the SSE.

To generate the necessary inputs to the SSE, a variety of matrix helper functions were implemented (see Appendix 9.4). Matrices are stored in flat arrays by column, which is how CVXgen stores data. There are two versions of the final SSE in C: the "offline" version reads in sensor data and motor inputs from files that have previously been written, and the "online" version interfaces with ROS to read in sensor data and inputs in real time (see Section 5.2.1).

3.4.2. Windowed Approach

In the process of translating the algorithm into C, it became clear that to calculate the original initial state at each timestep, the optimization parameters that store previous sensor outputs and motor inputs must change size on each loop iteration. This approach does not work in C because the necessary parameters must have fixed size. This limitation led to the development of a windowed approach to the SSE. Instead of calculating all the way back to the initial state $x[0]$, the solver calculates the state $x[t-T]$, which is T timesteps before the current state $x[t]$. This windowed approach is shown in Equation (22):

$$\begin{aligned}
&y[t-T] = Cx[t-T] \\
&y[t-T+1] = CAx[t-T] + CBu[t-T] \\
&y[t-T+2] = CA^2x[t-T] + CABu[t-T] + CBu[t-T+1] \\
&\qquad\qquad\vdots \\
&y[t] = CA^tx[t-T] + CA^{t-1}Bu[t-T] + CA^{t-2}Bu[t-T+1] + \cdots + CBu[t-1]
\end{aligned} \tag{22}$$

These equations can be rewritten in the same way as the original SSE algorithm in Equation (21). When $t < T$, the non-windowed approach is sufficient, but when $t \geq T$ the windowed approach is necessary. The C implementation code including the SSE algorithm is included in Appendix 9.49.2.

4. SIMULATION RESULTS

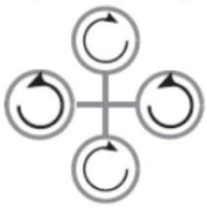
This section contains the results from simulations of the dynamic model and SSE algorithm. The dynamic model has been tested given arbitrary inputs and follows the expected behavior in a closed-loop simulation with a controller, and the SSE has been tested on a single degree of freedom system with noisy outputs.

4.1. DYNAMIC MODEL

As a preliminary test of the dynamic model, the equations of motion were tested with various combinations of inputs to confirm that the model followed the expected behavior. Motor voltage inputs were simulated by arbitrary voltage values for each of the motors. By changing the relative power of motor voltages, the dynamic model was tested for different flying scenarios, including flying straight up, flying left, and flying right.

The MATLAB code used to generate these plots is attached in Appendix 9.1. For the first simulation, the voltage input vector had the same value for all four motors. As expected, the simulation resulted in the quadrotor flying straight upwards, which is shown in Figure 5. To move the quadrotor left, the necessary motor configuration is shown in Figure 6(a). Given an input vector of voltages that follow this motor configuration resulted in the quadrotor moving left, as shown in Figure 6(b).

Fly Straight Up



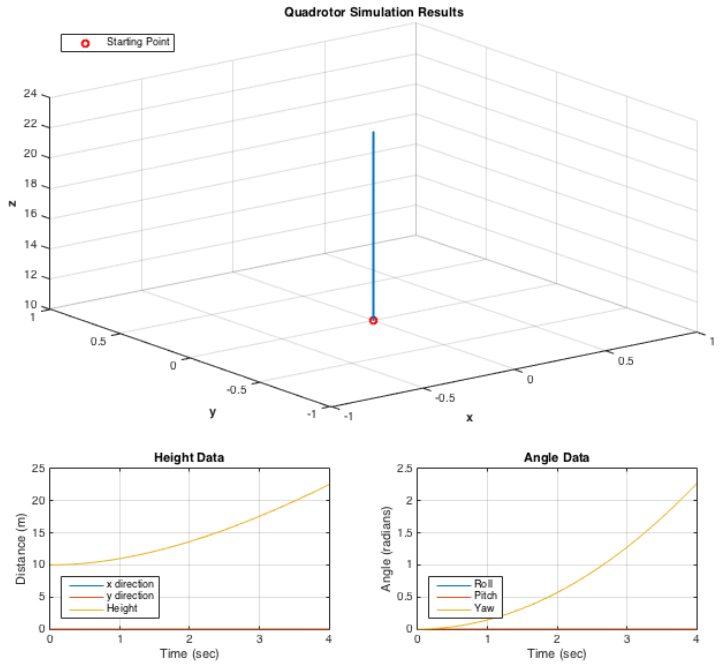
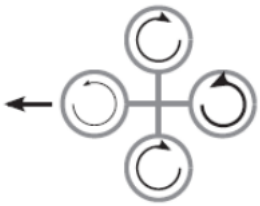


Figure 5 | Simulation of quadrotor flying straight up.

Move Left



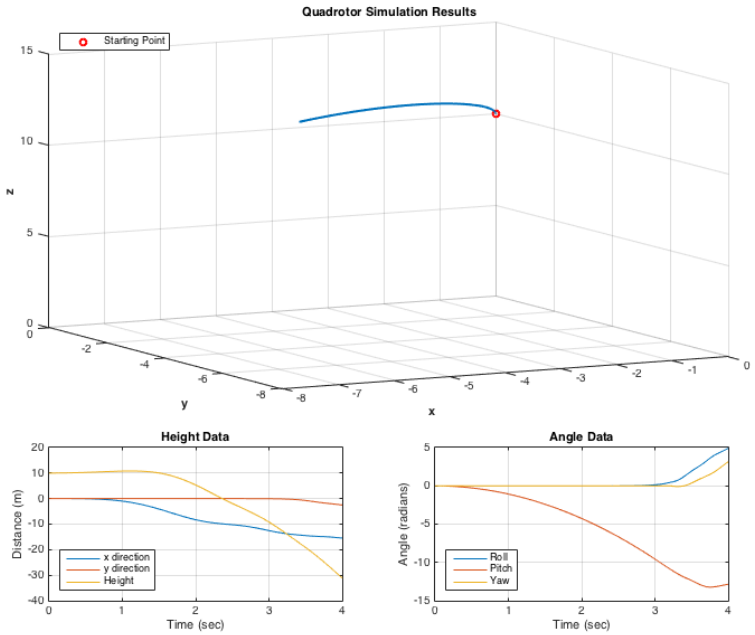


Figure 6 | (a) Motor configuration for flying left (b) Simulation results

To move the quadrotor right, the necessary motor configuration is shown in Figure 7(a). Given an input vector of voltages that follow this motor configuration resulted in the quadrotor moving right, as shown in Figure 7 (b).

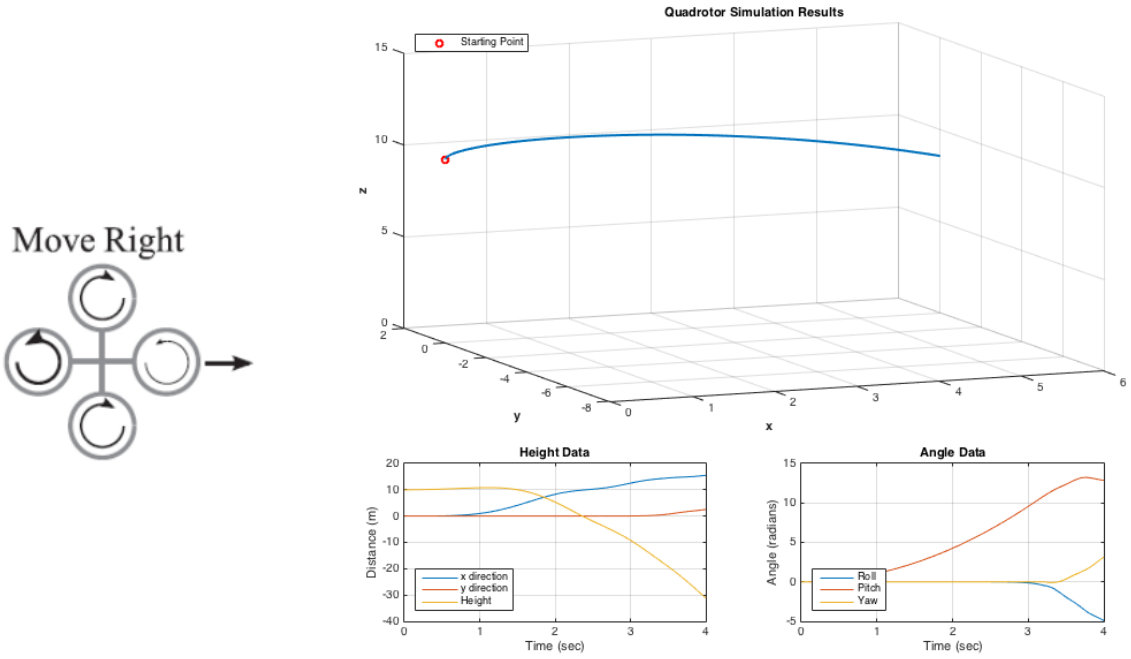


Figure 7 | (a) Motor configuration for flying right (b) Simulation results

These simulation results show that the dynamic model results agree with expectations given the input motor voltages, and therefore this dynamic model is an appropriate representation of the system.

4.2. SSE SIMULATION (MATLAB)

The SSE was first tested on a single degree of freedom system, then on the full quadrotor system with 10 states. The results of simulations on both of those systems will be discussed here, as well as the different approaches used in the SSE.

4.2.1. SDOF System

The SSE was first tested on a single degree of freedom mass damper system. The governing equation of the system is shown in Equation (23):

$$m\ddot{x} + b\dot{x} = f \tag{23}$$

Substituting $v = \dot{x}$ yields a first-order system, shown in Equation (24).

$$\dot{v} = -\frac{b}{m}v + \frac{1}{m}f \tag{24}$$

In state space, the system matrices are $A = -b/m$, $B = 1/m$, and $C = 1$ because the state and the output are the same. Since the SSE depends on a sampled system with discrete timesteps, the discrete-time equivalent matrices are calculated and used in the simulation. The simulation was run in MATLAB and the code is included in Appendix 9.2.

Figure 8 contains a plot of the performance of the SSE on a system with non-compromised sensor outputs. The system was given a reference velocity of 1 m/s and started at an initial state of 0.2 m/s.

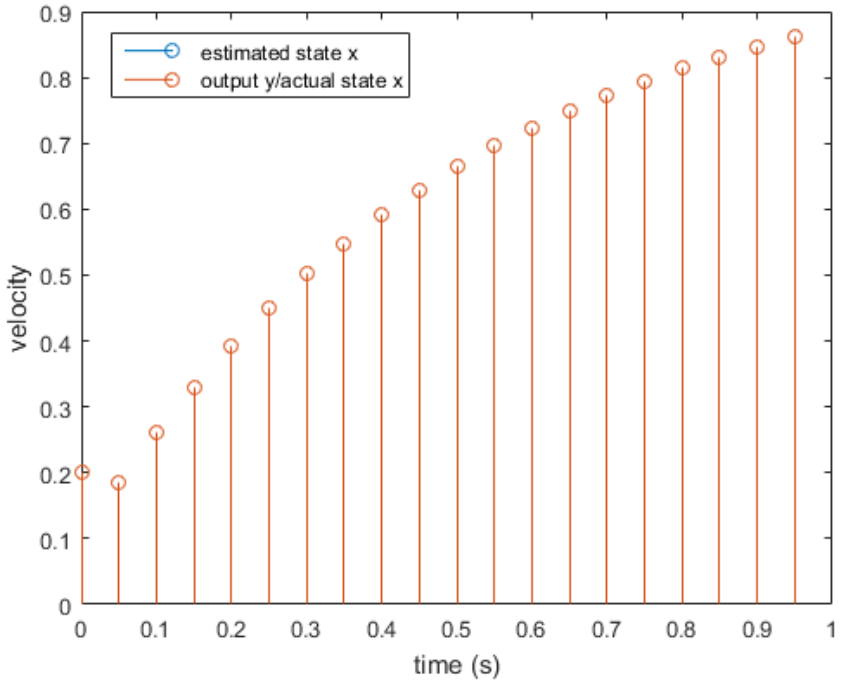


Figure 8 | SSE output with non-compromised sensors

In this plot, the SSE exactly predicts the system state, which confirms that the SSE is behaving as expected for functioning sensors in a non-attack situation. However, where the SSE really shines is in the situation where sensors are compromised. Figure 9 contains a plot of the system response to the same input and initial condition but with noisy sensor outputs.

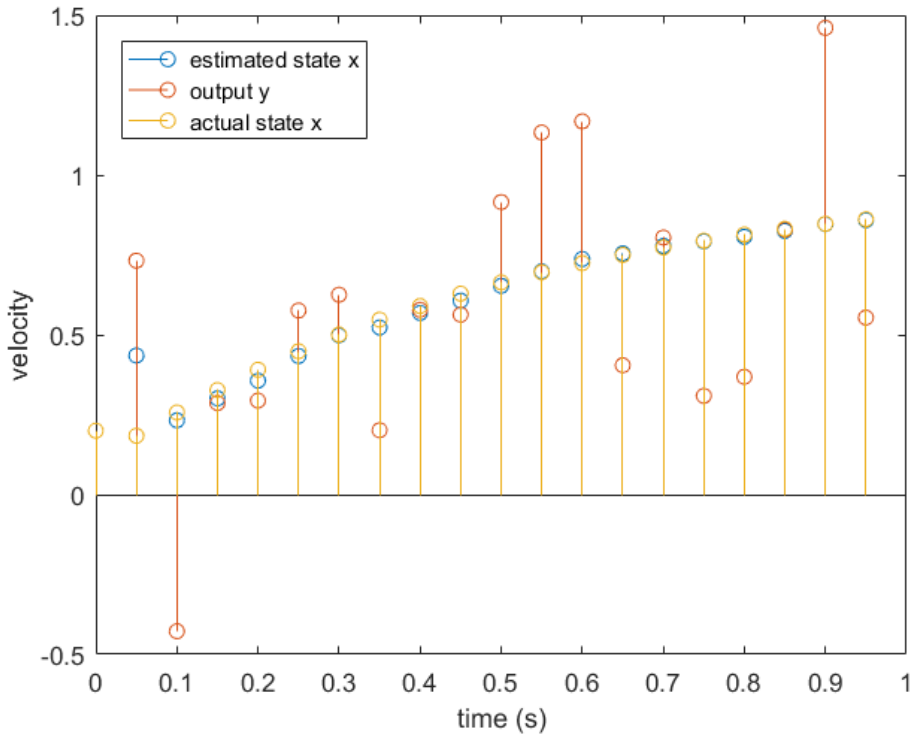


Figure 9 | SSE output with noisy sensors

Figure 8 shows that the SSE can accurately recover the system state even when the sensors are outputting noisy data. The average error of the sensors from the true state is 58%, but the average error of the estimated states from the true state is only 9% (excluding the spike at 0.5 seconds the error is only 3%).

4.2.2. Quadrotor System

After testing the SSE on a single degree of freedom system, the next step was to scale up to the full 10 degree of freedom quadrotor system. The linearized quadrotor dynamic model used in this SSE is shown in Equations (5) through (14). Shown in Figure 10 are the results from the SSE with 3 simulated height sensors, one of them compromised.

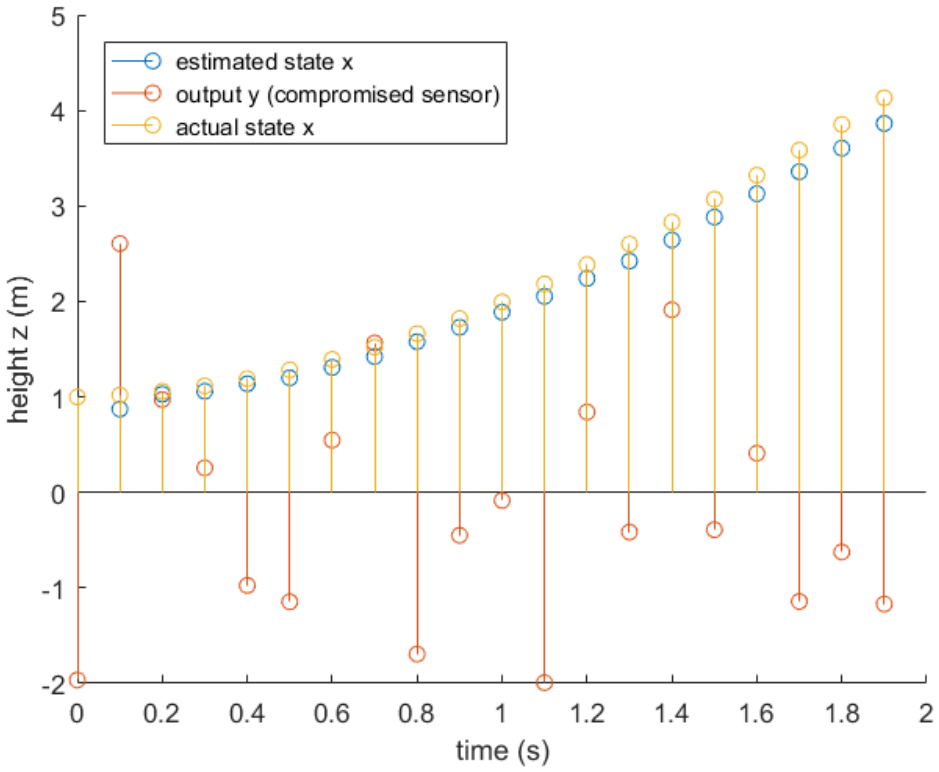


Figure 10 | Quadrotor SSE simulation with 3 height sensors (1 compromised)

In Figure 10, the estimated state produced by the SSE agrees well with the actual quadrotor state, which confirms that the SSE is working as expected on the full quadrotor system. The simulated sensor readings shown in red have an average error of 120% from the true state while the two other sensors (not shown) have an average error of 4%. Given this compromised data, the SSE produced an estimated state with an average error of only 6% from the true state. The error between the estimated and actual state is comparable to the error between the functioning sensors and is barely affected by compromising 1/3 of the sensors.

The previous plot shown in Figure 10 displays the performance of the SSE when one of three sensors is compromised, but we are also interested in the SSE performance as a function of the fraction of compromised sensors. By simulating many height sensors and

compromising different numbers of these sensors, we can obtain the behavior of the SSE as the number of compromised sensors changes. The average error in the state produced by the SSE when various numbers of sensors are compromised out of 50 total sensors is shown in Figure 11. Note that for this plot, the compromised sensors are outputting a state of zero.

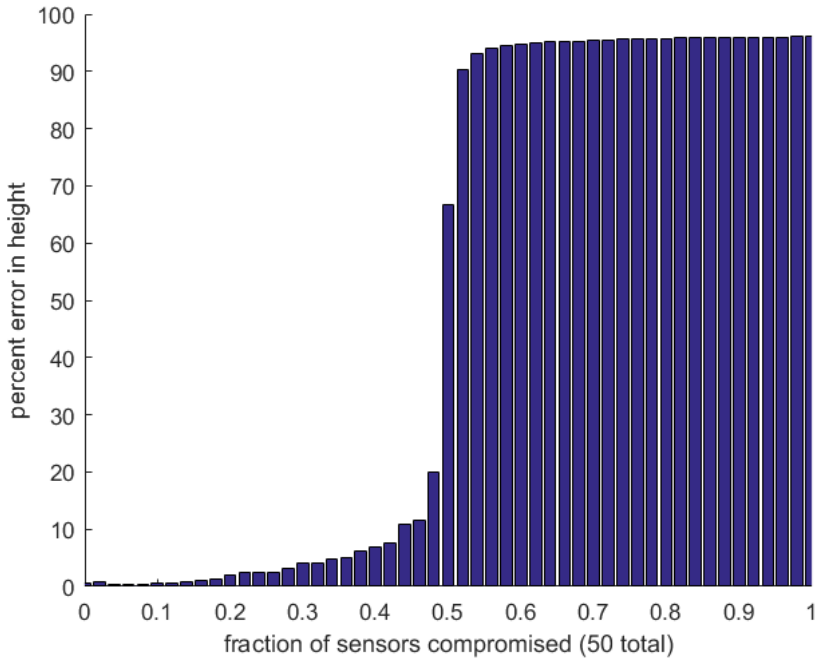


Figure 11 | Percent error as a function of compromised sensors

In Figure 11, the error in height follows a clear S-curve where the error is very low until half the sensors are compromised, and then increases drastically when more than half the sensors are outputting zero. This pattern agrees with intuition because once at least half of the sensors output zero, the algorithm optimizes for a state of zero instead of the actual initial state.

Another interesting metric is the fraction of successful recovery of the initial state, used by Fawzi et al. to verify their SSE performance [4]. Successful recovery is defined as producing an initial state of the system within a given tolerance of the true initial state. Our

results and how they compare to Fawzi et al. are shown in Figure 12. For both sets of data, 10 trials were run per number of attacked sensors. The tolerance for the quadrotor simulated data was 15%, and for the random system the tolerance was 10^{-5}.

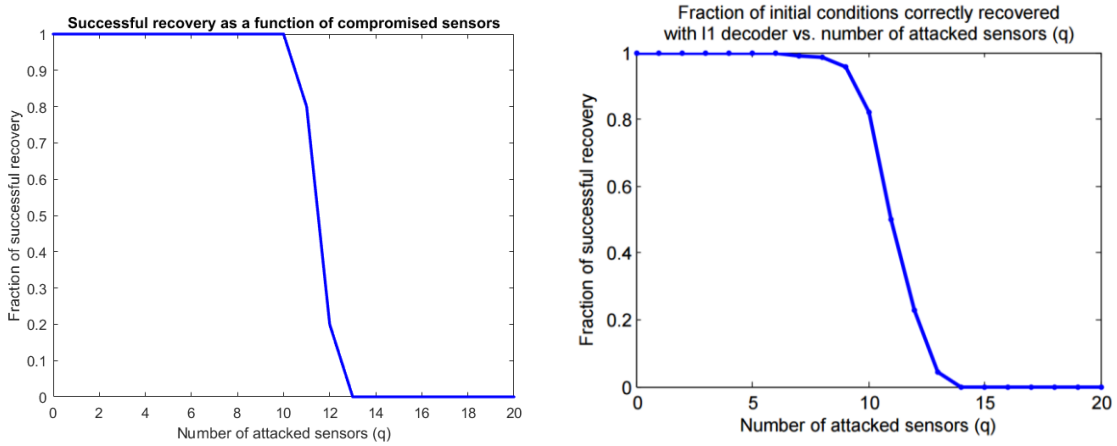


Figure 12 | (a) Quadrotor system and (b) Literature results (random system)

The results from the quadrotor system follow the same shape as the results from the random system in Fawzi et al., which confirms that the SSE is functioning as expected. The shape of the plots in Figure 12 also agrees with the trend shown in Figure 11: when more than 50% of sensors are compromised, the error increases drastically and it becomes much less likely that the original state will be successfully recovered.

Another simulation that was run was a comparison of the windowed and non-windowed approach. The results are shown in Figure 13.

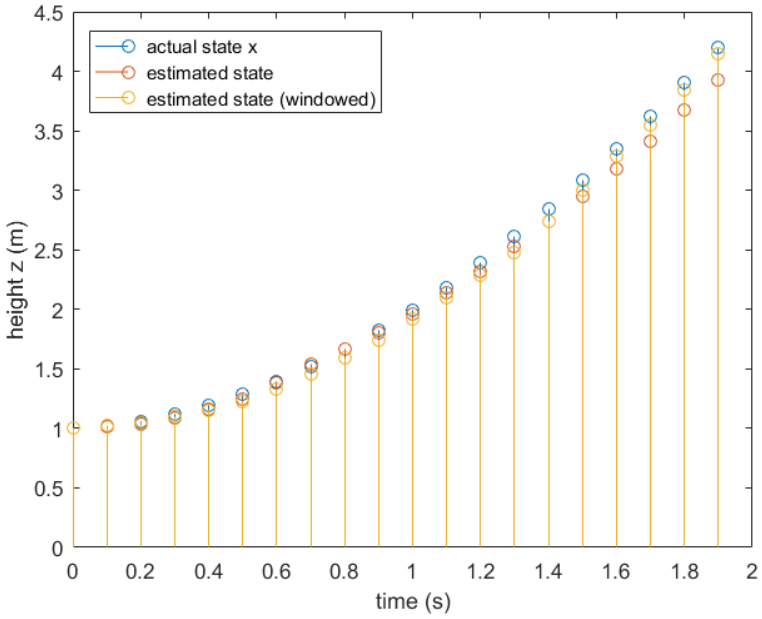


Figure 13 | Comparison of windowed and non-windowed approach in MATLAB

Figure 13 shows that as the number of timesteps increases, the windowed approach becomes a better approximation of the actual state than the non-windowed approach. One reason for this might be that in the windowed algorithm, the linear dynamic model is propagated for a shorter time and therefore the SSE can better track the real quadrotor system.

5. HARDWARE IMPLEMENTATION

This section discusses the hardware components of this project and how they are integrated, including the onboard computer, on and off-board sensors, quadrotor frame, the onboard controller, RC receiver, and power to all necessary components.

Since real time processing is a project requirement, all involved hardware must be mounted on the quadrotor. Figure 14 shows the connectivity of the onboard computer, controller, sensors, motors, and power. Each of these components are outlined in this section.

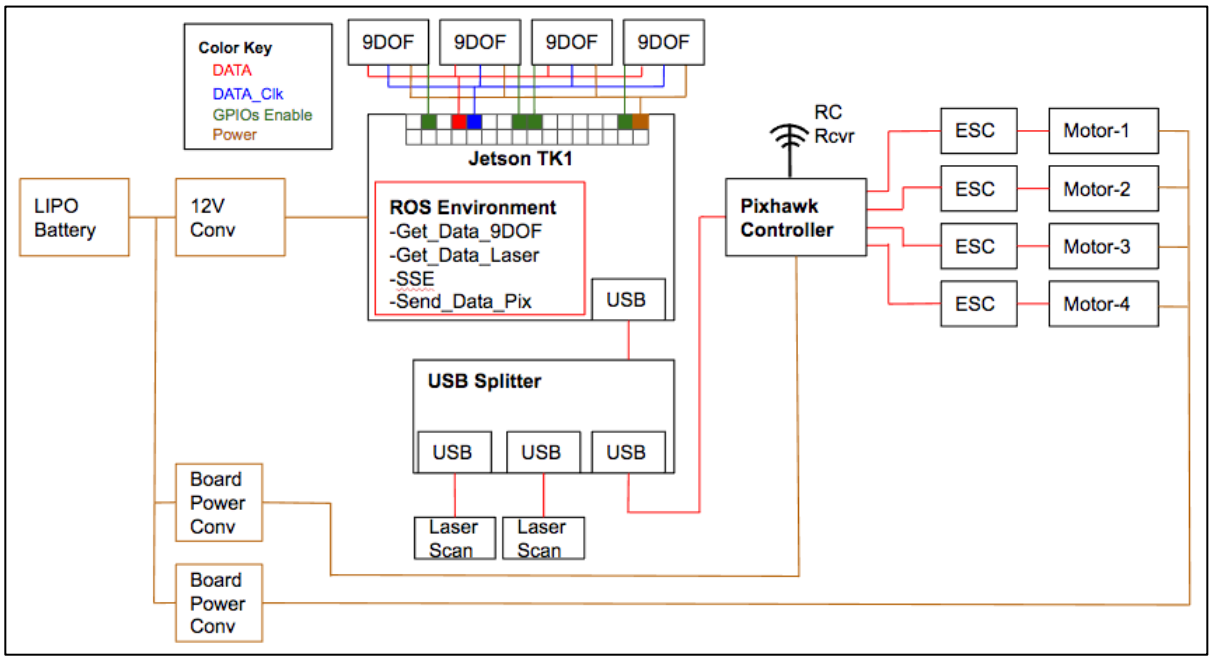


Figure 14 | Onboard Hardware Connectivity Diagram

5.1. QUADROTOR

This project requires a quadrotor frame that is customizable and as light as possible to incorporate the sensors necessary for state estimation. The NG liaisons suggested the QAV400 Quadrotor frame shown in Figure 15.

Figure 15 | Quadrotor QAV400 Frame [11]

The QAV400 quadrotor was selected because the NG R&D department has used this quadrotor frame successfully. Research into other quadrotor options revealed that the integration of motors with the frame by the manufacturer would speed up the process of quadrotor assembly, and the openings in the frame would allow sensors, computers, and power sources to be mounted on the quadrotor.

The QAV400 fits the project requirements, as shown in Table 3.

Project Constraints	Quadrotor Specs
Weight	375g
Electronic Controllers	Compatible with Pixhawk controller and battery
Payload capacity	Accommodate all sensors

Table 3 | Comparison of quadrotor specifications

The QAV400 can be purchased individually from Lumenier, allowing flexibility in choosing the flight control unit, sensors, actuators, and on-board computer [11]. In addition to the quadrotor frame, Lumenier provides electronic speed controllers (ESCs) and motors that are compatible with the quadrotor. When mounting the sensors, computers, and battery on this frame, the center of gravity will be kept as close to the center of the frame as possible to ensure the quadrotor flight is stable.

5.2. FLIGHT COMPUTER

Since the algorithm will be implemented using ROS, the NG liaisons suggested using the Jetson TK1 computer board. The Jetson TK1 was chosen over other similar options such as the Raspberry Pi 3 because the Jetson TK1 has more GPIO pins, memory, and processing power than a Raspberry Pi. The Nvidia Jetson TX1 was not selected because it is three times as expensive as the TK1, and this project's limited use of the TX1's extra graphics capabilities would not have made up for the cost increase. The Jetson TK1 satisfies all the project constraints as shown in Table 4. (For a visual representation of the Jetson TK1 ports, refer to Figure 14.)

Project Constraints	**Flight Computer Specifications**
Memory	2 GB RAM
Processing Power	Quad-core processor
Power	External power supply
Available Ports	
1 USB	USB Splitter to 2 Laser scanners and Pixhawk
I/O Pins	4 9DOF sensors (75 total pins)
SD Card Reader	SD card for data logging
Ethernet	Off-line communication

Table 4 | Comparison of flight computer specifications

The Jetson TK1 board is shown in Figure 16 with labels. The Ubuntu 14.04 Nvidia operating system has been installed to match the version of Ubuntu that the NG R&D department uses. The wide range of input/output capability (see Table 4) will facilitate the implementation and testing of different sensors [12].

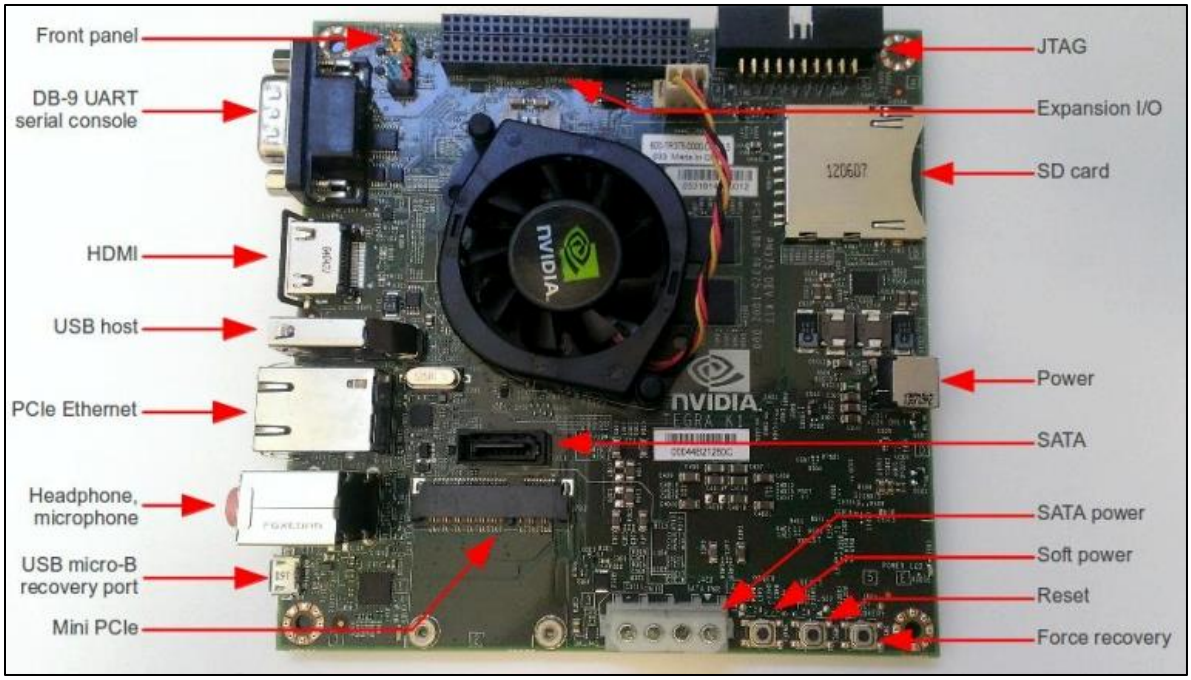


Figure 16 | Jetson TK1 [12]

All onboard processing will be hosted on the Jetson TK1. The Jetson has a separate wired connection to each onboard sensor (see Section 5.4 for a description of the sensors), and continually receives data in real time. A ROS environment manages all the asynchronous processing, including simultaneously receiving data from multiple sensors, processing the data with an SSE algorithm, and sending signals to the Pixhawk controller.

5.2.1. ROS Environment

The ROS environment on the Jetson facilitates sending, receiving, and processing asynchronous data, as well as bypassing the state estimator on the Pixhawk. All this is accomplished through the use of ROS nodes. A node can publish data, subscribe to a data stream, or do both. Publishing ROS-Indigo, two versions behind the most recent release of ROS (ROS-Kinetic) was selected because it is the version of ROS currently used by the NG R&D department. Each sensor corresponds to a ROS node that publishes data as separate rostopics. The SSE will run on another ROS node, collecting published sensor data and

control inputs and outputting state estimations. Finally, there is a node that sends state estimates to the Pixhawk controller. Figure 17 contains a diagram of the ROS nodes and other components of the ROS environment.

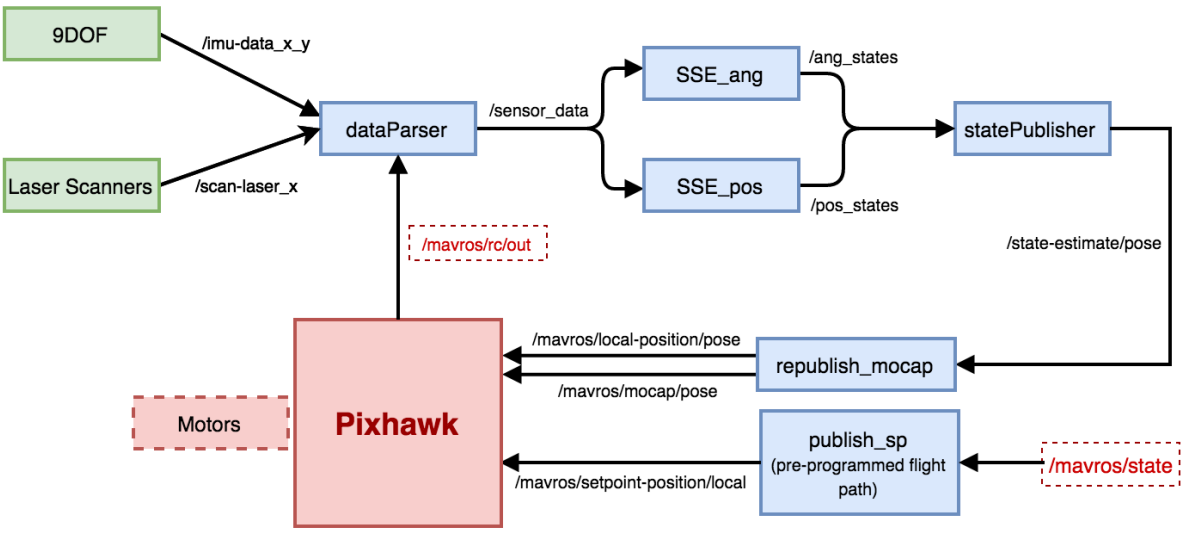


Figure 17 | MAVROS connectivity diagram

5.2.2. Data Logging

Data is logged on the Jetson through ROS; data sets from the sensors are published as rostopics that are accessible by other nodes in the ROS environment. However, the published data in rostopics are only available while the programs are running, so the data will also be recorded in a rosbag. Rosbags, unlike rostopics, are not deleted after sensors and the quadrotor are powered off. During flight, the SSE algorithm will be able to access data from either the rostopics or a rosbag during actual flight.

For data analysis, the rosbag can replay recorded data through the SSE on ROS after the flight. This data can also be recorded as text (csv) files and transferred to an offboard computer. The offboard SSE, either in C or Matlab, can post-process the flight data to

confirm results form the onboard flight. The offboard processing can also be used to look for discrepencies in either the data or the sensors.

5.3. FLIGHT CONTROLLER

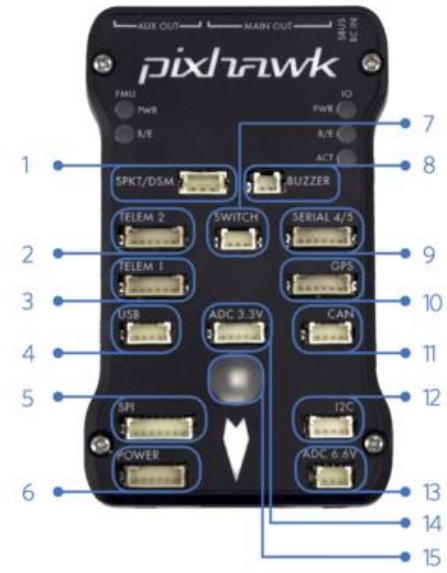
A flight controller is required to calculate the necessary control inputs for the quadrotor to track a desired flight path, and this controller must also communicate these inputs to the quadrotor motors as voltages. An off-the-shelf controller is sufficient because the focus is on the SSE algorithm and a custom controller is beyond the scope of the project. The Pixhawk 3DR flight controller was selected to control the quadrotor motors because it has been used successfully by the NG R&D department, it runs fast enough, and the software and firmware are open-source. This open-source availability is necessary because the SSE software must be integrated with the flight controller [14]. The 3DR Pixhawk controller is shown in Figure 18.

Figure 18 | 3DR Pixhawk Controller [14]

The Pixhawk PX4 software, called a flight stack, is installed on the Pixhawk, and the PX4 codebase, specifically the code which initiates the MAVLink connection to the ROS environment, must be modified to implement the SSE algorithm. The built-in state estimator in the PX4 software must be disabled so that motor inputs will be calculated using states from the SSE instead of states calculated by the built-in estimator.

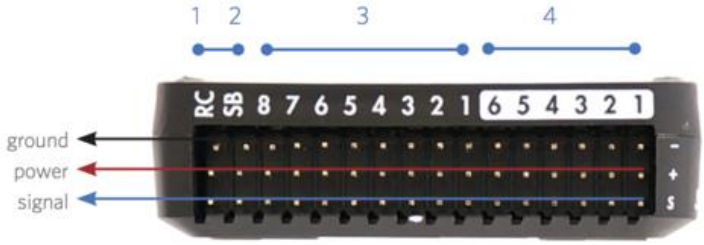
The Pixhawk itself is powered by 4.8 to 5.4 V and contains multiple sensors: an MPU6000 which is used as the main 3-axis accelerometer and 3-axis gyroscope, an ST Micro 16-bit gyroscope, an ST Micro 14-bit magnetometer, and an MEAS barometer. The Pixhawk has 5 UART ports (in the form of serial connections), one port each for I^2C and SPI connections, a USB port, and 3.3 and 6.6 V ADC ports [15]. The various ports of the Pixhawk are shown in Figure 19. The Pixhawk is compatible with the ESCs that come with the QAV quadrotor frame.

A USB to FTDI cable connects the USB port of the Jetson to a DF13 6 position cable which connects to the serial port of the Pixhawk. The Pixhawk and Jetson will be able to both send and receive data through this connection. Two-way communication between the Jetson and Pixhawk is necessary for the Jetson to send states to the Pixhawk and for the Pixhawk to send control inputs back to the Jetson.



1 Spektrum DSM receiver
2 Telemetry (radio telemetry)
3 Telemetry (on-screen display)
4 USB
5 SPI (serial peripheral interface) bus
6 Power module
7 Safety switch button
8 Buzzer
9 Serial
10 GPS module
11 CAN (controller area network) bus
12 I²C splitter or compass module
13 Analog to digital converter 6.6 V
14 Analog to digital converter 3.3 V
15 LED indicator

1 Input/output reset button
2 SD card
3 Flight management reset button
4 Micro-USB port



1 Radio control receiver input
2 S.Bus output
3 Main outputs
4 Auxiliary outputs

Figure 19 | Pixhawk Connector Descriptions [16]

5.4. SENSORS

Sensors were selected to measure the states listed in Table 2. The list of sensors that were initially considered is included in Table 5:

State	Sensor
Position	Radar Pressure sensor OptiTrack Camera tracking system
Position, Velocity	Airspeed sensor Laser scanner (2) Accelerometers (4)
Position, Acceleration	Accelerometers (4)
Orientation	3DOF compass OptiTrack Camera tracking system
Angular velocity	Gyroscopes (3)

Table 5 | Possible sensors

The project constraints narrowed down the sensor selection list. Radar sensors within the budget, such as the Doppler Radar Motion Detector [17], did not have a sufficient resolution. This sensor can detect a minimum distance of 0.3 m, but tests will be conducted where the quadrotor may fly lower than 0.3 m. Pressure sensors, such as the BMP180 Barometric Pressure/Temperature/Altitude Sensor [18], do not have the necessary precision for measuring changes in distance on a millimeter scale. The BMP180 can only sense pressure changes greater than a range of 0.25 m, which is not a high enough resolution for the low-flying tests that will be run in this study. Airspeed sensors, such as the Airspeed microSensor from RobotShop [19] can sense a minimum velocity of 9 miles per hour. This minimum velocity requirement that is higher than the intended operating speed of the quadrotor, which should not exceed 5 mph during testing. For these reasons, the sensors underlined in Table 5 were selected.

The initial sensor selections of a 3DOF compass and 6DOF IMU (with accelerometers and gyroscopes) were backordered, so a new 9DOF sensor with accelerometers, gyroscopes, and magnetometers was selected. The final list of onboard sensors is included in Table 6:

3-DOF Position & Velocity	3-DOF Orientation & Angular Velocity
Accelerometers (3) Laser scanner	Magnetometers (3) Gyroscopes (3)

Table 6 | Sensors and associated states

Two sensors onboard the quadrotor will measure all the relevant states of the system: a 9DOF IMU and a laser scanner. A third sensor, the OptiTrack vision system, can be used to determine the true states of the vehicle for calculating the tracking error; the OptiTrack system will not be used as an onboard sensor and will not input values to the SSE. In the spring semester, a stretch goal of the project may be to send position and orientation measurements from the OptiTrack system to the Jetson TK1, but the OptiTrack outputs are not currently being used in this way.

5.4.1. 9DOF Sensor

The I2C BNO055 9-Axis IMU (by Adafruit) includes three accelerometers, three gyroscopes, and three magnetometers to measure the motion of the quadrotor [20]. The 9DOF sensor can communicates using I2C, which allows the sensor to be directly connected to the I2C pins on the Jetson TK1. The BNO055 chip has only two hardware address options, and since I2C requires a unique address on the same data line, the sensors can be at most paired. Therefore, in order to use four 9DOF sensors, there will be two pairs of the BNO055 chips connected to the Jetson TK1. The power requirement is 3 to 5 V, and the maximum ranges for each of the components are ±16g for the accelerometers, ±2000dps for the

gyroscopes, and ±2500μT for the magnetometers. The accelerometer outputs can be integrated to track the quadrotor's 3DOF velocity states and the magnetometers and gyroscopes will be used to track the 3DOF orientation and angular velocity, respectively. The 9DOF sensor is shown in Figure 20.

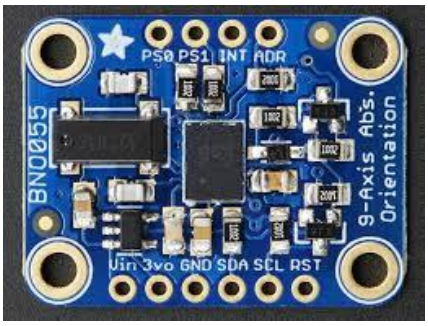


Figure 20 | 9DOF IMU [20]

5.4.2. Laser Scanner

The Hokuyo URG-04LX-UG01 Scanning Laser Rangefinder was selected as the laser scanner. The laser scanner returns distances of the nearest object in a 240-degree scan of a plane. The resolution is 0.001m with a guaranteed accuracy of 0.03m. The laser scanner updates its measurements at a frequency of 10Hz. The scanner is shown in Figure 21.

Figure 21 | Laser scanner rangefinder [21]

The device is connected to the Jetson TK1 through a USB hub, since multiple laser scanners must be connected at the same time and the Jetson only has one USB-A port. Hokuyo provides ROS drivers and wrappers for this laser scanner in C. The files were altered

slightly to allow multiple lasers to be connected at once and log data simultaneously. Simultaneous logging was accomplished by defining a new rosparam when initializing each laser scanner as a rosnode to run. The wrapper was also changed to log each connected laser to a different rostopic so each laser will not overwrite data from the other lasers (see Appendix).

5.4.3. OptiTrack vision system

The OptiTrack vision system (OptiTrack) can be used as a ground truth replacement for GPS, since testing will be conducted indoors where GPS is not available. OptiTrack is capable of tracking objects with six degrees of freedom, 3DOF position and 3DOF orientation, by using visible and IR light from multiple cameras at a frame rate of up to 240fps, with a 0.1s delay in data processing [22]. Figure 22 shows the Optitrack cameras set up on the ceiling, pointing towards the space in which the quadcopter will be flown.

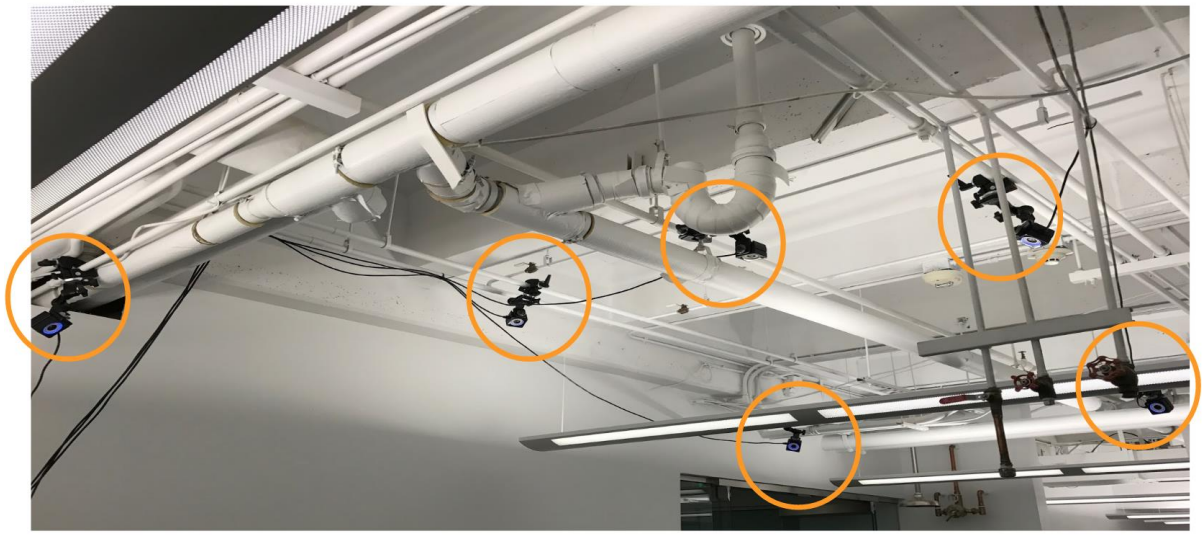


Figure 22 | OptiTrack camera setup

To track an object, IR-reflective spheres of a known radius must be attached to the object and a rigid body must be defined in the OptiTrack software. The system calculates the

position and orientation of the object by measuring the location of the IR-reflective spheres. The accuracy of the OptiTrack and orientation measurements depends on the number of spheres and how much the spheres move relative to one another during flight.

Calibrating the OptiTrack system requires a calibration stick with IR-reflective spheres, shown in Figure 23(a). The lower part of Figure 23(b) shows each camera on the calibration screen. The colorful splotches in the boxes are the views of each of seven cameras logging the movement of the calibration stick. Optitrack combines the data from every camera to obtain the global position of the spheres. To track the quadrotor, IR-reflective spheres will be attached to the quadrotor frame using two-sided tape.

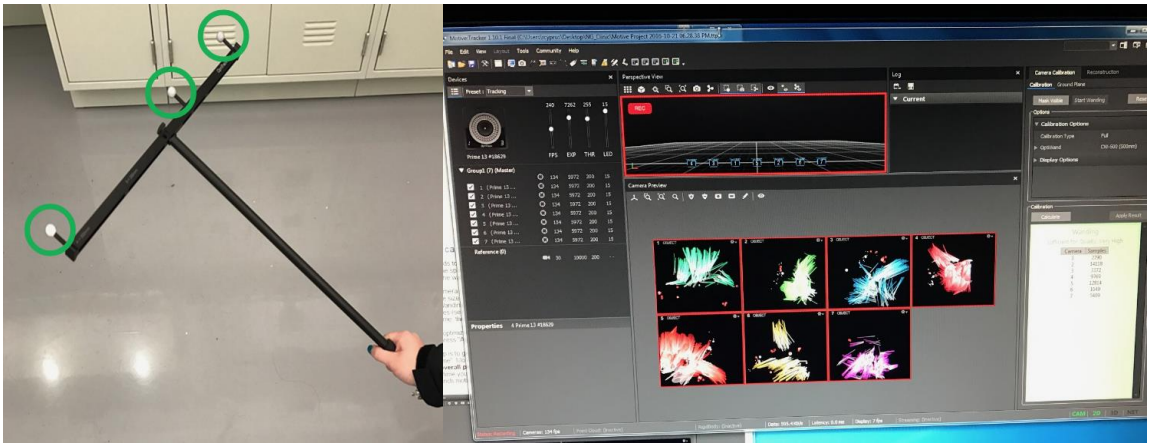


Figure 23 | (a) Optitrack calibration tool (b) Optitrack calibration screen

The positioning, calibration, and initial setup of the OptiTrack system was tasked for the first semester and was completed successfully. However, the team ran into hardware issues with the quadrotor, and the OptiTrack system was put on hold in order to overcome them. This resulted in the abandonment of the OptiTrack system for this project. In future work, the team highly recommends using a Motion Capture (MOCAP) system, such as OptiTrack or Vicon, in order to receive truth data during flights.

5.4.4. Sensor Mounts

In order to fly the quadrotor with the SSE, the two laser scanners, four IMUs, the Jetson, the Pixhawk and the LiPo battery needed to be arranged and mounted securely on the frame. The IMUs were screwed on to a laser-cut plate so that they would remain rigid relative to the frame and were placed near the center of mass of the quadrotor (Figure 24). The laser scanners needed to be oriented facing downward, so a laser-cut plate was used to mount the two laser scanners on either side of the quadrotor, also near the center of mass (Figure 25). The Jetson was mounted using a laser-cut frame that covered the bottom and top of the board, and was screwed on to the top of the quadrotor (Figure 26). The Pixhawk was screwed to the frame using the Pixhawk 3DR mount set that came with the flight controller. In order to keep the quadrotor balanced, the LiPo battery was attached using a Velcro strap opposite the Jetson, since the Jetson and the battery were the heaviest elements mounted on-board. The sensor mounts were small and light-weight so the contribution to the overall mass and inertia was insignificant.

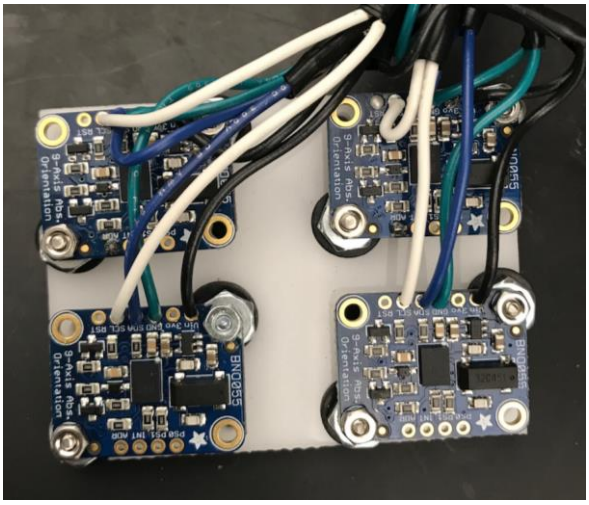


Figure 24 | IMU sensor mount for 4 sensors

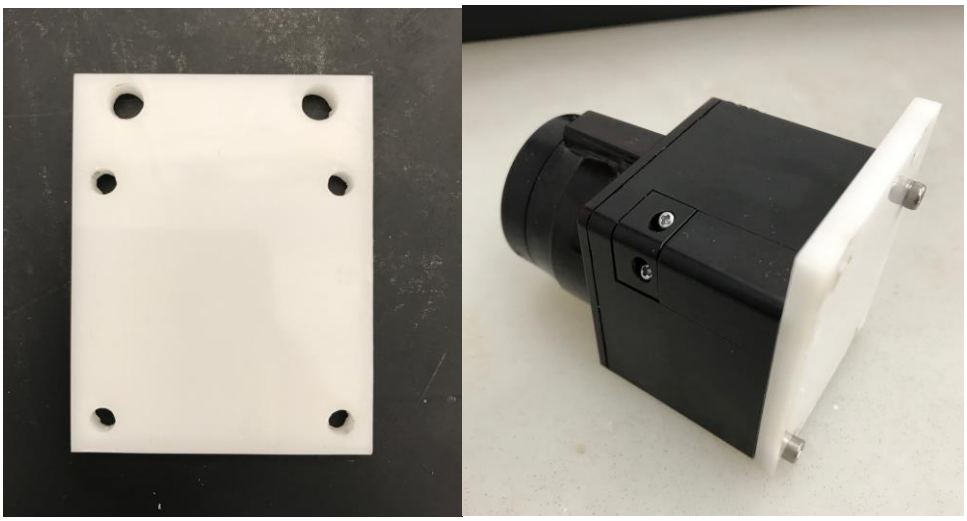


Figure 25 | Laser scanner sensor mounts

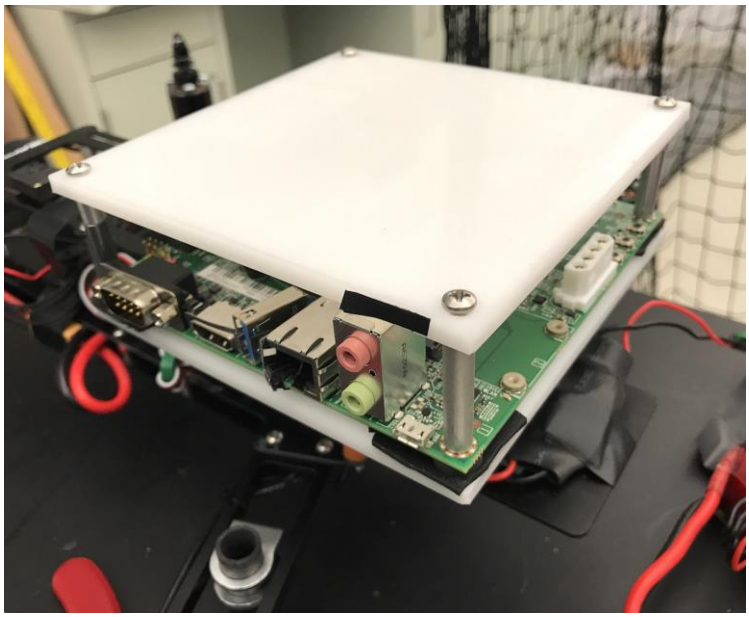


Figure 26 | Jetson mounted on quadrotor

5.5. MOTOR INPUTS

The SSE requires the control inputs to be the angular velocity squared of the motors. As the flight controller, the Pixhawk sends pulse width modulated (PWM) signals to each motor, controlling their speeds. Since there were no sensors reading the motor speeds in real time, the PWM values sent by the Pixhawk (updated at 10Hz) were used in the motor speed

calculation. Figure 27 shows an empirical derivation of the relationship between PWM output and the motor rotations per minute (RPM).

According to the Pixhawk documentation, the PWM output numbers range from 1000 to 2000. The motor rotation rate was assumed to equal the propeller rotation rate, assuming no slippage. The propeller RPM was measured using a tachometer.

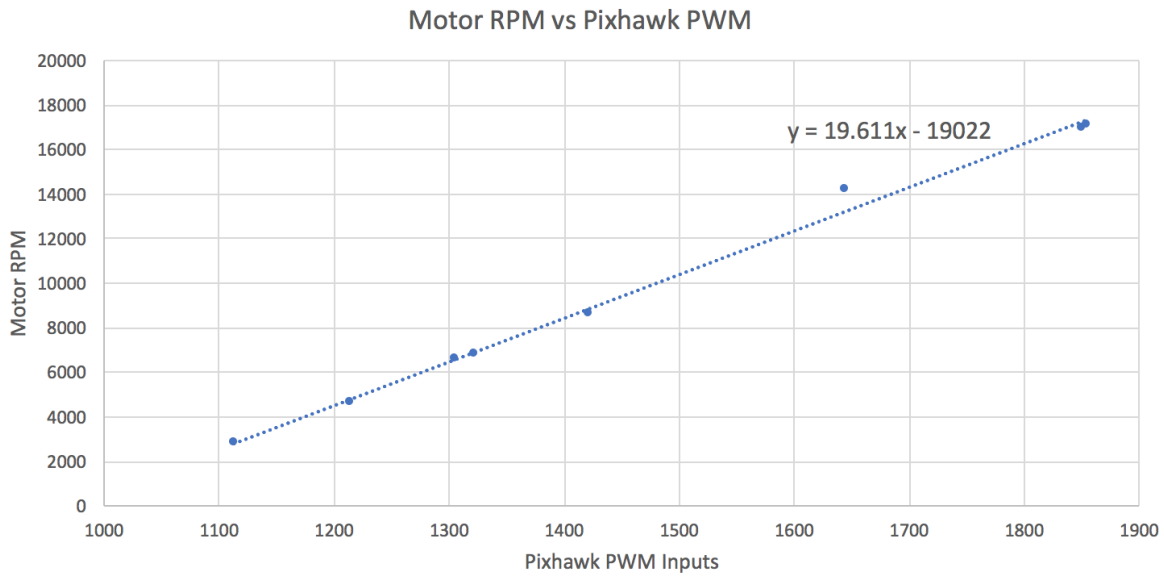


Figure 27 | Motor calibration curve

As seen in Figure 27, a linear fit was applied to the eight empirically gathered data points. This equation was implemented in the C code in the Data Parser node, shown in Section 5.2.1, which fed into the SSE node. This node also converted the RPM into the needed control input of angular velocity squared.

5.6. RC RECEIVER

A Turnigy 9X RC transmitter and its corresponding eight-channel receiver was used to control the quadrotor when not flying a pre-programmed flight path or to switch the flight mode of the Pixhawk between stabilized and off-board mode. The receiver connects to a PPM

encoder, which combines the pulse-width modulation (PWM) signals of the receiver into a single signal using pulse-position modulation (PPM). The PPM encoder then sends the signals to the RC input port on the Pixhawk, which directs the motors based on these input signals. The connection from the Pixhawk to the RC receiver through the PPM encoder is shown in Figure 28.

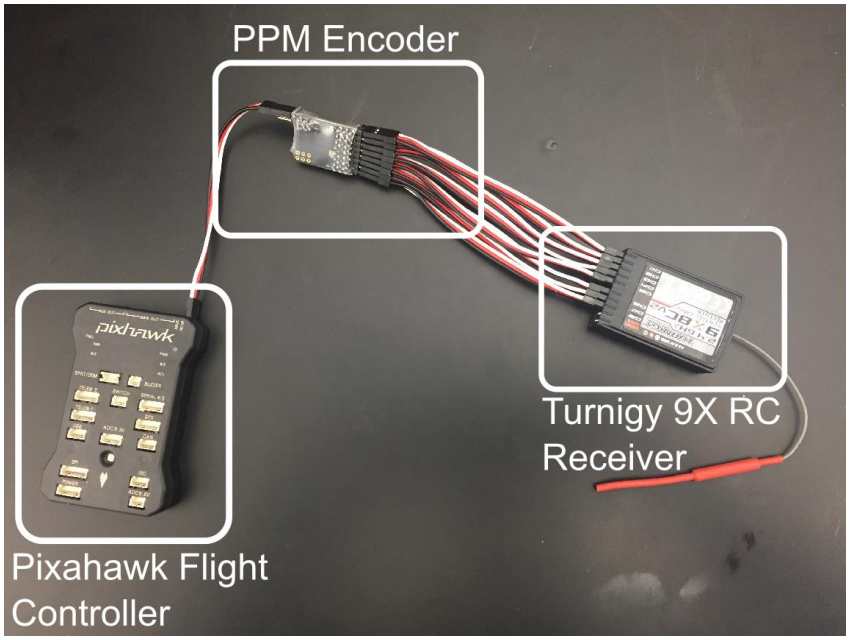


Figure 28 | RC receiver connection to the Pixhawk

5.7. POWER GRID

A 14.8 V LiPo battery will power the quadrotor, with the appropriate voltage regulators stepping down the voltage to power the Jetson, Pixhawk, and motors. The selected battery is compatible with the ESCs and motors that come with the QAV400 quadrotor frame. A 12 V regulator will connect the LiPo to the Jetson, and the board converters received from the manufacturers will connect the motors and Pixhawk to the LiPo as well. The connection from the LiPo battery to the Jetson through a voltage regulator is shown in Figure 29.

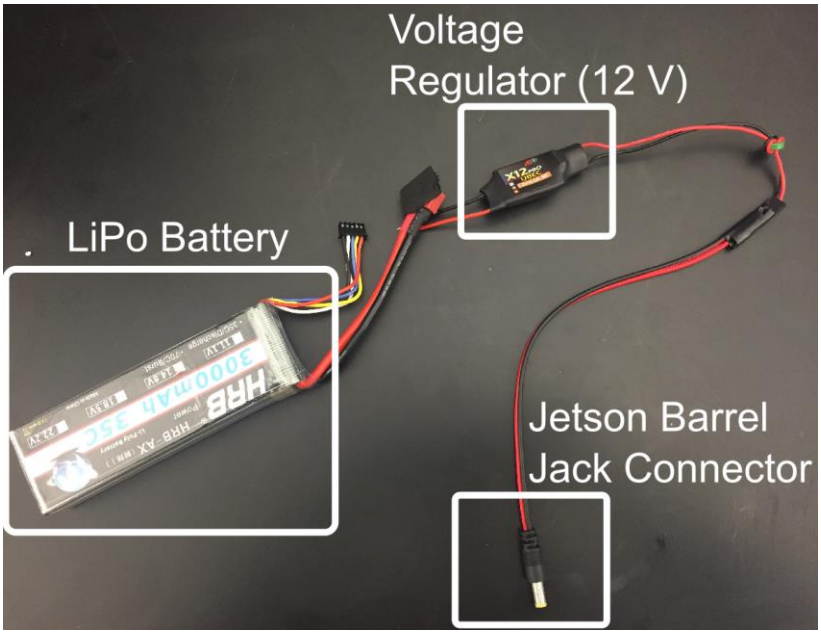


Figure 29 | Components for Jetson power supply

5.8. JETSON TO PIXHAWK COMMUNICATION

As mentioned previously, a ROS environment was used on the Jetson because it facilitates sending, receiving, and processing asynchronous data and can bypass the state estimator on the Pixhawk. This allowed for the integration of the SSE with the Pixhawk's motor controller functionality without having to delve in the Pixhawk's flight stack. The Pixhawk flight stack has MAVLink built in and we installed MAVROS onto the Jetson which acts as an extendable communication node between NuttX, the Pixhawk's operating system, and ROS, the Jetson's environment. MAVROS comes with a set of predetermined commands which allow us to abstract only the parameters we are interested in logging or modifying within the Pixhawk's flight stack. Using the abstraction capabilities of MAVROS, the Pixhawk's local state estimator was remapped to a temporary folder (essentially disabling it), and the states from the SSE on the Jetson were sent to the Pixhawk via serial as motion capture truth data.

6. HARDWARE RESULTS

This section documents the performance of the SSE during actual flight of the quadrotor in the clinic space. There are results from both real-time and post processed data, obtained through processing the data in the stored rosbag after flight.

6.1. REAL-TIME SSE (C)

The first test for the real-time SSE implemented in C was to confirm that the output matched the output of the SSE in MATLAB. These results are shown in Figure 30.

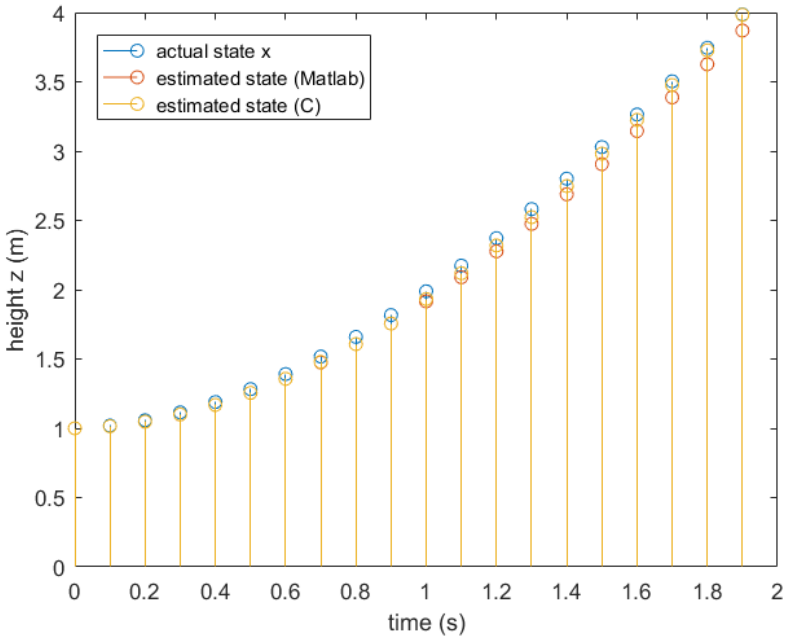


Figure 30 | Comparison of SSE in MATLAB and C

The estimated heights from MATLAB and C begin at the same height (1m) and then slightly diverge because in both algorithms because random (non-deterministic) noise was added to the simulated sensor outputs. Another cause for the disagreement can be attributed to the C algorithm using the Windowed approach, outlined in Section 3.4.1. This approach

only used the last N timesteps (in our case, N=10) in calculating optimal initial state, rather than compute the absolute initial state at time=0sec. The error between the estimated state using the MATLAB algorithm and the true state is 3%, while the error between the estimated state using the algorithm min C and the true state is only 2%. The agreement between the algorithms is good enough to verify that the real-time algorithm in C produces a reasonable estimate of the system state.

6.2. OFF-BOARD SSE TESTING

To test the SSE, the quadrotor was flown manually using the RC controller. The sensor data (laser scanners and IMU's) along with the control inputs (motor PWM values) were recorded using a rosbag that allowed for playback afterwards for offboard testing of the SSE. Initially the quadrotor was tethered to the ground for safety and to prevent damage. Figure 16 shows the SSE results for height on a tethered test flight. The averaged laser scanner data (uncompromised) is shown in orange, while the SSE state estimate is shown in blue. Since there was no real-time tracking system in place, the laser scanner data is treated as truth data.

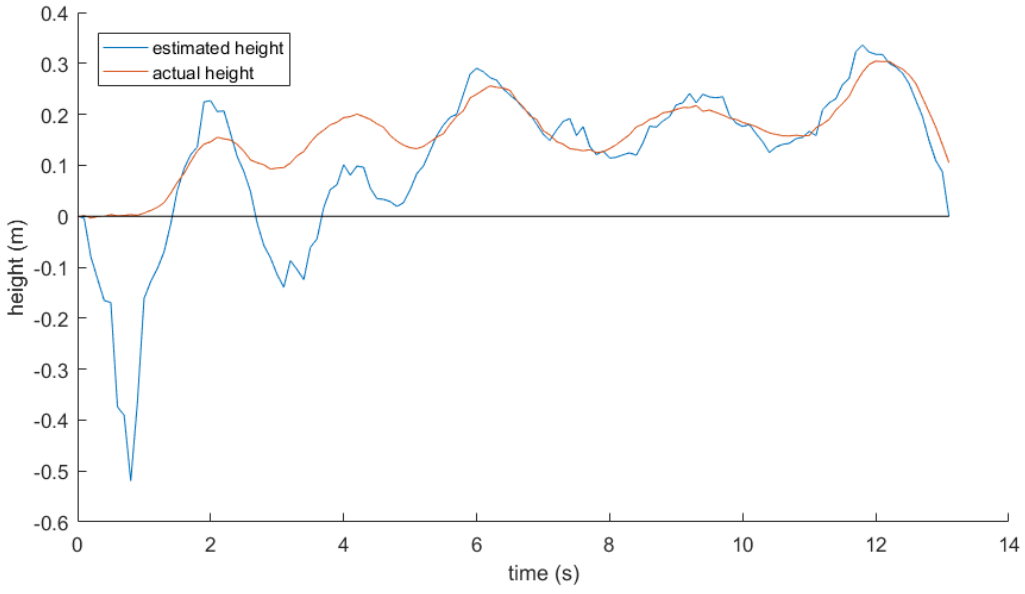


Figure 16 | SSE offboard results for tethered flight

Firstly, the state estimates seem appropriate and track the sinusoidal height profile resembled in the laser scanner measurements. From the flight time of 6 seconds to the end, there is an average of approximately 13% absolute error in height. One major issue with the tethered flight was the inability to account for external forces, such as the normal force of the ground or the tension in the tethers. This could possibly account for the unexpected behavior at about 2 to 3 seconds. However, between 3 and 6 seconds in the flight in Figure 16, the trajectory of the SSE predicted height output roughly matches the shape of the actual height, and at about 6 seconds, the SSE is recovered.

Therefore, once the quadrotor was verified to be stable in manual mode, the quadrotor was tested untethered, again using manual control with the RC controller. The SSE results for the height are shown in Figure 17.

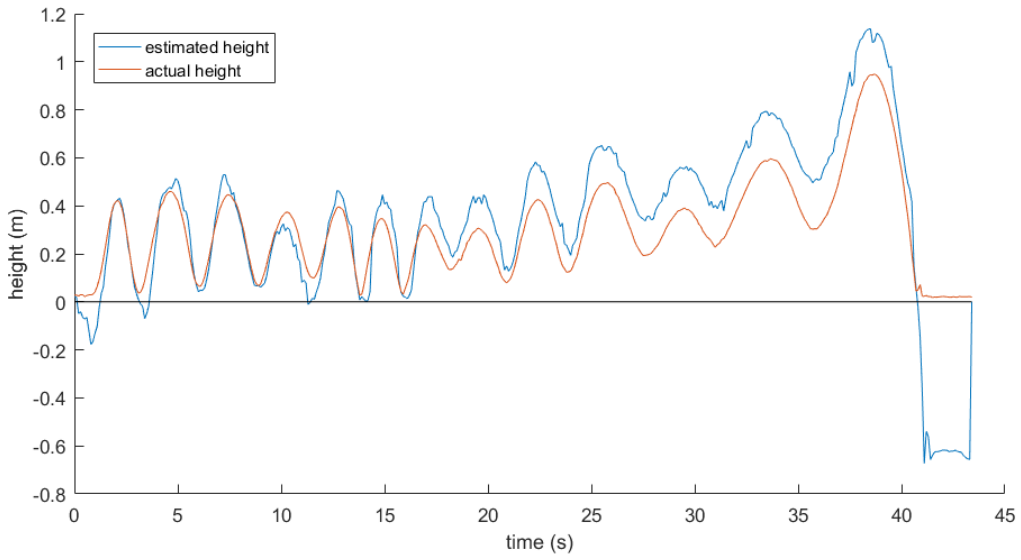


Figure 17 | SSE offboard results for untethered flight

The results of the untethered flight show less deviation between state estimate and sensor readings compared to the tethered flight results. The drop in the first two seconds is due to neglecting the external normal force from the floor, which was not accounted for in the SSE model or sensor measurements. Looking at the data after the initial drop, in the 2-10 second rage, there is about 24% absolute error in height SSE prediction from actual height measured from the laser scanners (from 2 seconds to just before the final 3 seconds when the quadrotor is stationary, the average error is 43%). After about 12 second, the SSE prediction starts to diverge more from the actual height, although the trajectory generally follows the same shape.

One likely reason for the error in state estimates comes back to the seemingly compromised velocity calculations from the IMU's accelerometers. With two z-velocity data points coming from the differentiated laser scanner height measurements, and the other two from the integrated accelerometer data, there is effectively 50% compromised sensors, as illustrated in Figure 18, which plots the z-velocity inputs to the SSE.

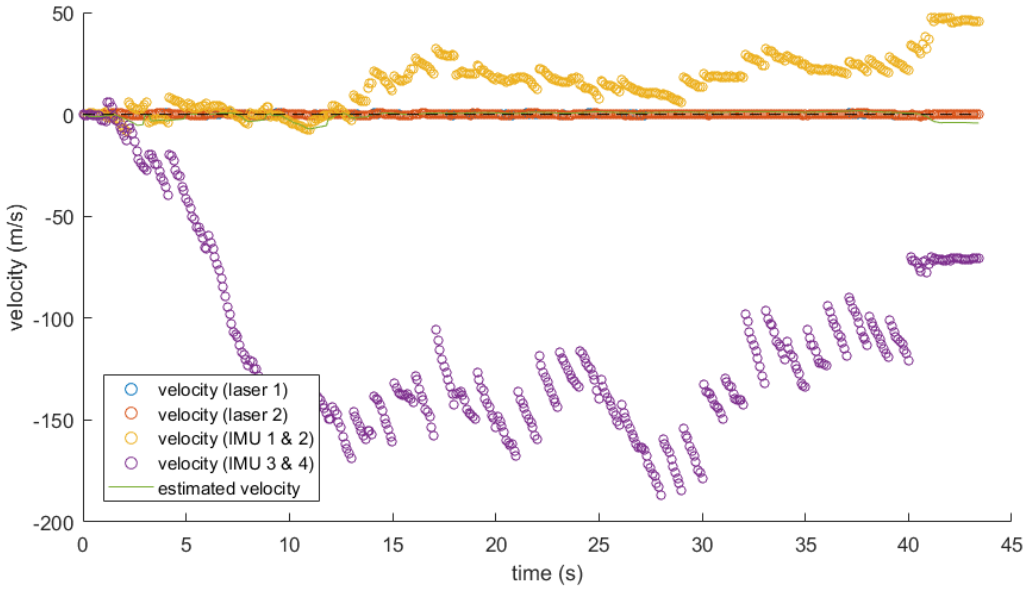


Figure 18 | Z-Velocity Inputs to the SSE

In Figure 18, there are three of the five lines which overlap around the "0" velocity level: from laser 1, laser 2, and estimated velocity. These velocities actually fall between -1 and 1m/s, but are hard to see due to the scale of the figure. The average error between the estimated velocity and the velocity from the laser scanners is 57%. The velocity measurements due to the IMU pairs (numerical integration of accelerometer data) result in extremely corrupt data, with an average error of 3000% for IMU1&2 (yellow line) and 8000% for IMU3&4 (purple line). The test space for the quadrotor flights was 8ft x 8ft x 8ft, but according to IMU3&4 in Figure 18, the quadrotor was flying at -150m/s for about 25sec, which is a descent of over 9000m. The SSE can recover a fairly reasonable state based on two functioning velocity measurements, even with two of the four velocity measurements severely compromised.

The SSE currently does not have the specific system parameters of our quadrotor. Thus, to improve the accuracy of the SSE, the exact parameters (moment of inertia, propeller velocity to thrust coefficient, etc.) must be tuned to match the system.

Due to limited time, the angle SSE was not debugged. Despite the sensor outputs seeming reasonable (magnetometer and gyroscope outputs), the state estimates were extremely inaccurate. There may be a bug in the SSE algorithm, or it may be due to inaccurate system parameters, since the position SSE seemed reasonable, and the angle SSE was largely the same.

6.3. SSE PERFORMANCE IN CONTROL LOOP

The quadrotor was tested in autonomous mode untethered, with onboard SSE updating only the z-position, letting the Pixhawk internal state estimator control the x and y position estimates, as well as the orientation. Simple flight paths were tested, defining set points linearly ascending to 0.5 meters and back to ground, keeping x and y position constant, along with the orientation constant.

Since the autonomous test flights were largely unsuccessful and with the limited time, there was no complete data processing or analysis for these flights.

However, the autonomous flights illustrated some key problems with the control system design. Namely, the lack of x-y global positioning results in drift of the quadrotor. On flights when the quadrotor seemed to be ascending properly in the z-direction, there was no way to control the x-y position, resulting in many crashes.

7. PROJECT OVERVIEW

The position SSE functionality was verified offboard and onboard (in the ROS environment), but not successfully integrated into the control loop. Although slightly inaccurate, with errors of about 10% for the majority of valid data periods of the flights, the height estimates did indeed follow the behavior of the quadrotor. The estimates were severely handicapped by the erroneous velocity data calculated from the accelerometers. Unfortunately, it seemed that these specific readings were always compromised without our intentional intervention. Further system parameter tuning is expected to also yield better results.

The position SSE was integrated into the control loop unsuccessfully, largely due to the lack of global x-y position data, resulting in lateral drift. Because the x-y was not controlled, when tests to control the z (height) were performed, the quadrotor would drift in the x-y plane into the netting setup around the test area. This team suggests either using GPS or a Motion Capture system to control x-y data for prolonged height control testing.

7.1. PROJECT CONSTRAINTS

In formulating the problem space at the beginning of the project, the following constraints were established.

Final implementation of the SSE must be in ROS: The SSE was successfully implemented in ROS and ran real-time on the quadrotor.

Quadrotor must fly for 10 minutes on a single battery charge: This constraint was not met. The quadrotor had a flight time of approximately 3 minute on a single battery.

Must estimate states at a frequency of at least 25 Hz: Although the SSE only updated states at 10 Hz (limited by the 10 Hz update rate of the laser scanner), the control loop ran at 100 Hz. Additionally, there were some tests with euler approximations to fill in the dead time after the SSE runs and before the next laser scanner update arrives.

All equipment and supplies must cost less than $7,000: In total, the quadrotor is priced at ~$4000. More money was used extra parts and replacements, or other materials not on the quadrotor (nets, foam, etc.).

7.2. FUTURE WORK

Future work for this project could be to further test the SSE in off-board mode on the quadrotor. This would involve giving the quadrotor a flight path to autonomously track and test the response of the SSE given compromised sensors. Although this was attempted during the span of this project, the results of autonomous flying were not consistent and did not track the flight path very successfully.

In order to have a better comparison of the SSE with and without compromised sensors, having a system to record truth data (OptiTrack, GPS) would allow for better quantifying the error. Once this is done it could also be possible to track a flight path that changes in the x and y direction as well and not just altitude tracking.

8. REFERENCES

[1] "10 Ways Drones Are Changing Your World," *Consumer Reports*, 24-Nov-2016. [Online]. Available: http://www.foxnews.com/tech/2016/11/24/10-ways-drones-are-changing-your-world.html. [Accessed: 02-Dec-2016].

[2] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge, UK ; New York: Cambridge University Press, 2004.

[3] "CVX: Matlab Software for Disciplined Convex Programming | CVX Research, Inc." [Online]. Available: http://cvxr.com/cvx/. [Accessed: 29-Nov-2016].

[4] H. Fawzi, P. Tabuada, and S. Diggavi, "Secure estimation and control for cyber-physical systems under adversarial attacks," *IEEE Trans. Autom. Control*, vol. 59, no. 6, pp. 1454–1467, Jun. 2014.

[5] Y. Shoukry, P. Nuzzo, N. Bezzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, and P. Tabuada, "Secure state reconstruction in differentially flat systems under sensor attacks using satisfiability modulo theory solving," in *2015 54th IEEE Conference on Decision and Control (CDC)*, 2015, pp. 3804–3809.

[6] C. Lee, H. Shim, and Y. Eun, "Secure and robust state estimation under sensor attacks, measurement noises, and process disturbances: Observer-based combinatorial approach," in *Control Conference (ECC), 2015 European*, 2015, pp. 1872–1877.

[7] S. Mishra, Y. Shoukry, N. Karamchandani, S. Diggavi, and P. Tabuada, "Secure state estimation: Optimal guarantees against sensor attacks in the presence of noise," in *2015 IEEE International Symposium on Information Theory (ISIT)*, 2015, pp. 2929–2933.

[8] “Quadcopter Dynamics and Simulation - Andrew Gibiansky.” [Online]. Available: http://andrew.gibiansky.com/blog/physics/quadcopter-dynamics/. [Accessed: 02-Dec-2016].

[9] “Fig. 1: whole structure of quadcopter robot and inbound forces on robot.” [Online]. Available: https://www.researchgate.net/figure/245031939_fig1_Fig-1-whole-structure-of-quadcopter-robot-and-inbound-forces-on-robot. [Accessed: 05-Dec-2016].

[10] Gagic, Zoran, “Introduction to Linear and Nonlinear Observers.” [Online]. Available: http://www.ece.rutgers.edu/~gajic/psfiles/observers.pdf. [Accessed: 05-Dec-2016].

[11] “QAV400.” [Online]. Available: http://www.lumenier.com/products/multirotors/qav400. [Accessed: 23-Sep-2016].

[12] “Jetson TK1 - eLinux.org.” [Online]. Available: http://elinux.org/Jetson_TK1. [Accessed: 23-Sep-2016].

[13] “CVXGEN: Code Generation for Convex Optimization.” [Online]. Available: http://cvxgen.com/docs/index.html. [Accessed: 29-Nov-2016].

[14] “3DR Pixhawk - 3DR.” [Online]. Available: https://store.3dr.com/products/3dr-pixhawk. [Accessed: 23-Sep-2016].

[15] “Pixhawk Overview — Copter documentation.” [Online]. Available: http://ardupilot.org/copter/docs/common-pixhawk-overview.html#common-pixhawk-overview. [Accessed: 16-Dec-2016].

[16] “Pixhawk Autopilot - Pixhawk Flight Controller Hardware Project.” [Online]. Available: https://pixhawk.org/modules/pixhawk. [Accessed: 05-Dec-2016].

[17] "Doppler Radar Motion Detector." [Online]. Available: http://www.robotshop.com/en/doppler-radar-motion-detector.html. [Accessed: 15-Dec-2016].

[18] "BMP180 Barometric Pressure/Temperature/Altitude Sensor." [Online]. Available: http://www.robotshop.com/en/bmp180-barometric-pressure-temperaturealtitude-sensor.html. [Accessed: 15-Dec-2016].

[19] "Airspeed MicroSensor Kit." [Online]. Available: http://www.robotshop.com/en/airspeed-microsensor-kit.html. [Accessed: 15-Dec-2016].

[20] *SPI/IIC MPU-9250 9-Axis Attitude +Gyro+Accelerator+Magnetometer Sensor Module*. .

[21] "Hokuyo URG-04LX-UG01 Scanning Laser Rangefinder." [Online]. Available: http://www.robotshop.com/en/hokuyo-urg-04lx-ug01-scanning-laser-rangefinder.html. [Accessed: 01-Dec-2016].

[22] "OptiTrack," *OptiTrack*. [Online]. Available: https://www.optitrack.com/. [Accessed: 01-Dec-2016].

[23] "URG Network / Wiki / Home." [Online]. Available: https://sourceforge.net/p/urgnetwork/wiki/Home/. [Accessed: 30-Nov-2016].

9. APPENDIX

9.1. DYNAMIC MODEL

```
function [x_new, y] = quadrotor_plant(u, state, constants)

    % Physical constants.
    m = constants.m;
    L = constants.L;
    k = constants.k;
    b = constants.b;
    I = constants.I;
    kd = constants.kd;
    g = constants.g;
    dt = constants.dt;
    C = constants.C;

    % state variables
    x = state(1);
    xdot = state(2:4);
    theta = state(5:7);
    thetadot = state(8:10);

    % Compute forces, torques, and accelerations.
    omega = thetadot2omega(thetadot, theta);
    a = acceleration(u, theta, xdot, m, g, k, kd);
    w_dot = angular_acceleration(u, omega, I, L, b, k);

    % Advance system state.
    omega = omega + dt*w_dot;
    thetadot = omega2thetadot(omega,theta);
    theta = theta + dt*thetadot;
    xdot = xdot + dt*a;
    x = x + dt*xdot(3);

    % Store simulation state for output.
    x_new = [x; xdot; theta; thetadot];
    y = C*x_new;
    % potentially change this if we need velocity as a state and
acceleration as an output

end

% Function thetadot2omega adapted from Gibiansky
% Convert derivatives of roll, pitch, yaw to omega.
function omega = thetadot2omega(thetadot, angles)
    phi = angles(1);
    theta = angles(2);
    %psi = angles(3);
    W = [
        1, 0, -sin(theta)
        0, cos(phi), cos(theta)*sin(phi)
        0, -sin(phi), cos(theta)*cos(phi)
    ];
```

```
    omega = W * thetadot;
end

% Function that calculates the acceleration a
% Inputs: inputs current vector, angles, and coefficients
function a = acceleration(inputs,angles,xdot,m,g,k,kd)
    grav = [0;0;-g];
    % TODO: figure out what the rotation function is
    R = rotation(angles);
    T = R * thrust(k,inputs);
    Fd = -kd * xdot;
    a = grav + (1/m)*T + Fd;
end

% Adapted from
http://andrew.gibiansky.com/downloads/pdf/Quadcopter%20Dynamics,%20Simulat
ion,%20a%20Control.pdf
% Compute rotation matrix for a set of angles.
function R = rotation(angles)
    phi = angles(3);
    theta = angles(2);
    psi = angles(1);

    R = zeros(3);
    R(:, 1) = [
        cos(phi) * cos(theta)
        cos(theta) * sin(phi)
        - sin(theta)
    ];
    R(:, 2) = [
        cos(phi) * sin(theta) * sin(psi) - cos(psi) * sin(phi)
        cos(phi) * cos(psi) + sin(phi) * sin(theta) * sin(psi)
        cos(theta) * sin(psi)
    ];
    R(:, 3) = [
        sin(phi) * sin(psi) + cos(phi) * cos(psi) * sin(theta)
        cos(psi) * sin(phi) * sin(theta) - cos(phi) * sin(psi)
        cos(theta) * cos(psi)
    ];
end

% Function that computes thrust
% Inputs: inputs vector of current inputs and thrust coefficient
function T = thrust(k,inputs)
    T = [0; 0; k*(inputs(1) + inputs(2) + inputs(3) + inputs(4))];
end

% Function that calculates the angular acceleration
% Inputs: inputs current vector and coefficients
function w_dot = angular_acceleration(inputs,omega,I,L,b,k)
    tau = torques(L,k,b,inputs);
    w_dot = inv(I)*(tau-cross(omega,I*omega));
end

% Torques in the body frame
% Inputs: current output from each rotor (4 rotors)
function tau = torques(L,k,b,inputs)
```

```
    tau = [ L*k*(inputs(1) - inputs(3));
            L*k*(inputs(2) - inputs(4)); ...
            b*(inputs(1) - inputs(2) + inputs(3) - inputs(4))];

end

% Convert omega to roll, pitch, and yaw
% Inputs: omega and angles
function thetadot = omega2thetadot(omega,angles)
    phi = angles(1);
    theta = angles(2);
    %psi = angles(3);
        W = [
        1, 0, -sin(theta)
        0, cos(phi), cos(theta)*sin(phi)
        0, -sin(phi), cos(theta)*cos(phi)
    ];
    thetadot = inv(W)*omega;
end
```

9.2. SIMULATION WITH SSE ALGORITHM

```
%% Inputs
Ttotal = 1; % total sample time
Ts = 0.05;  % simulation timestep

% reference signal (desired state)
ref = 1;

% initial state
x_init = 0.2;
xhat = x_init;
y_init = x_init;

% system
b=5; M=3;
A=-b/M;
B=1/M;
C=1;

%% Initialization and initial calculations
T = Ttotal/Ts; % total timesteps

% initialize matrices to store outputs and inputs
U = zeros(1,T);
Y = zeros(1,T);     Y(1) = y_init;
X = zeros(1,T);     X(1) = x_init;
Xhat = zeros(1,T);  Xhat(1) = xhat;
Xinit = zeros(1,T); Xinit(1) = x_init;

% initialize matrices for SSE calcs
Bu = zeros(1,T);
CA = zeros(T,1); CA(1) = C;

% determine digital state space equations
sys = ss(A,B,C,0);
```

```
sysd = c2d(sys,Ts);
Ad = sysd.a;
Bd = sysd.b;
Cd = sysd.c;

% control
K = dlqr(Ad,Bd,1,0.1);
Kr = -inv(Cd*inv(Ad-1-Bd*K)*Bd);

%% Simulation
for t=1:T-1 % t is the timestep number
    %% Plant
    u = U(t+1); % grab control inputs u[t]
    x = X(t);   % grab state x[t-1]
    [x_new, y] = plant_md(Ad, Bd, Cd, u, x);
    Y(t+1) = y + 0.5*rand(1);     % save noisy y[t] output in Y matrix
    X(t+1) = x_new; % save new state x[t] in X matrix

    %% State estimator
    % update observability matrix
    CA(t+1) = Cd*(Ad^t);

    % update matrix of control inputs
    for i=0:t
        Bu(t+1) = Bu(t+1) + Cd*(Ad^(t-i))*Bd*U(i+1);
    end

    % run optimization to find initial state x
    YBu = Y(1:t+1) - Bu(1:t+1);
    CA_t = CA(1:t+1);
    r = 2;
    cvx_begin quiet
        variable x(1)
        minimize( sum(norms(YBu - reshape(CA_t*x,[1,t+1]), r, 2)) )
    cvx_end
    fprintf('For t=%.2f, cvx problem is %s!\n', t*Ts, cvx_status)

    Xinit(t+1) = x;

    % propagate dynamics to find previous state from initial state
    for i=1:t
        x_new = Ad*x + Bd*U(i+1);
        x = x_new;
    end
    xhat = x_new;
    Xhat(t+1) = xhat; % save estimated state xhat[t] in Xhat matrix

    %% Controller
    u_new = Kr*ref - K*xhat; %*X(:,t+1); % calculate new control inputs
    U(t+2) = u_new; % save new inputs u[t+1] in U matrix

end

%% Plot results
figure;
stem(0:Ts:Ttotal-Ts, Xhat,'DisplayName','estimated state x')
hold on
```

```
stem(0:Ts:Ttotal-Ts, Y,'DisplayName','output y')
label('time (s)')
ylabel('velocity')
legend('show')
```

9.3. CVXGEN IMPLEMENTATION

9.3.1. Quadrotor SSE, norm-1

```
dimensions
  N = 10 # number of states
  P = 38 # number of sensors
  T = 10 # number of timesteps
end

parameters
  CA (P*T,N)
  YBu (P*T,1)
  I[p]  (T,P*T), p=1..P
end

variables
  x (N)
end

minimize
  sum[p=1..P](norm_1(I[p]*(YBu - CA*x)))
end
```

In this algorithm, `I[p]` is a matrix of mostly zeros with the following form (for $T = 2$ and $P = 3$):

$$I[1] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad I[2] = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \quad I[3] = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

9.3.2. Quadrotor SSE, summing absolute values

```
dimensions
  N = 10 # number of states
  P = 38 # number of sensors
  T = 10 # number of timesteps
end

parameters
  CA  (P*T,N)
  YBu (P*T,1)
end

variables
  x (N)
end

minimize
  sum(abs(YBu - CA*x))
end
```

9.3.3. Quadrotor SSE, with sparsity

```
dimensions
  N = 10 # number of states
  P = 38 # number of sensors
  T = 10 # number of timesteps
end

parameters
  CA  (P*T,N) {1,1 2,1 3,2 4,2 5,2 6,2 7,3 8,3 9,3 10,3  11,4  12,4  13,4  14,4
  YBu (P*T,1)
end

variables
  x (N)
end

minimize
  sum(abs(YBu - CA*x))
end
```

The sparse indices are cut off because there are 520 (row, col) pairs of sparse indices. Each pair corresponds to an entry in the `CA` matrix that is nonzero.

9.3.4. Separate position and orientation SSEs

Position SSE:

```
dimensions
  N = 4 # number of states
  P = 14 # number of sensors
  T = 10 # number of timesteps
end

parameters
  CA  (P*T,N) { 1,1   2,1   3,2   4,2
  YBu (P*T,1)
end

variables
  x (N)
end

minimize
  sum(abs(YBu - CA*x))
end
```

Orientation SSE:

```
dimensions
  N = 6 # number of states
  P = 24 # number of sensors
  T = 6 # number of timesteps
end

parameters
  CA (P*T,N) {1,1 2,2 3,3 4,4 5,5
  YBu (P*T,1)
end

variables
  x (N)
end

minimize
  sum(abs(YBu - CA*x))
end
```

Like in the previous section, the sparse indices are cut off.

9.4. SSE IN C

The matrix helper functions for the position SSE are included here as an example. The orientation SSE includes the same functions but different constants.

```
// CVX_inputs.h
// Contains helper functions to create necessary inputs to quadrotor SSE

#include "matrix-helper.h"

#ifndef INPUTS_H_INCLUDED
#define INPUTS_H_INCLUDED 1

// Define constants
#define N 4 // N = states
#define P 14 // P = sensors
#define T 10 // T = T
#define M 5 // M = inputs

// Define sparsity
#define nonZeroEntries 16

// Constant Matrices
double A[N*N]; // nxn
double B[N*M]; // nxm
double C[P*N]; // Pxn

// Update contents after each timeStep
double CA[P*T*N]; // PTxn
double YBu[P*T];  // PTx1
double U[M*T];    // MxT
double Y[P*T];
double y[P];
double x[N]; // state vector

/*
 * Creates a PT x n matrix based on matrices A and C of quadrotor model
 * Updates the specified "section" of CA matrix (global)
 */
void updateCA(int timeStep);

/*
 * Creates a PT x 1 matrix of sensor measurements and propagated inputs
 * Inputs: timeStep: only updating at a given timeStep
 *         y: sensor outputs at current timeStep (Px1)
 *         U: control inputs (all time) (mTx1)
 * Output: YBu (global matrix) (PTx1)
 */
void updateYBu(int timeStep, double* yin, double* Uin);

/*
 * Given the initial state output from CVX and U (vector of global
inputs),
 * propagate system dynamics to obtain the previous system state
```

```
 */
void propagateDynamics(int timeStep, double* U, double* x);

/*
 * Raises the square matrix A with a size len x len to the power of T
 * (Currently only applied to global matrix A)
 */
void power(int t, double* AT);

#endif

// CVX_inputs.c
// Inputs to quadrotor CVX optimization solver

#include "CVX_inputs.h"
#include <stdio.h>

////////////////////////////
// CVXgen input functions //
////////////////////////////

// CVXgen sparse indicies
static int rowInd[nonZeroEntries] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 1, 2};
static int colInd[nonZeroEntries] = {1, 1, 2, 3, 4, 2, 3, 4, 2,  3,  4,
2,  3,  4, 4, 4};

/*
 * Creates a PT x n matrix based on matrices A and C of quadrotor model
 * Updates the specified "section" of CA matrix (global)
 */
void updateCA(int timeStep) {
  // Initialize intermediate array and output array
  double tmpCA[P*N];
  double AT[N*N];

  // Compute C*(A^T)
  power(timeStep, AT);
  multiply(C, P, N, AT, N, N, tmpCA);

  //printf("tmpCA is:\n");
  //printArrayDouble(tmpCA,P,N);

  // Copy tmpCA into full CA matrix (no sparsity)
  /*
  for (int i = 0; i < N; ++i) {
    for (int p = 0; p < P; ++p) {
    //CA[(P*N)*timeStep+i] = tmpCA[i];
      CA[i*P*T + timeStep*P + p] = tmpCA[i*P + p];
    }
  }
  */
  int r = 0;
  int c = 0;
  int i = 0;
  for (i = 0; i < nonZeroEntries; ++i) {
    r = rowInd[i]-1; // Matlab/CVXgen indexes from 1
```

```
    c = colInd[i]-1;
    CA[i + nonZeroEntries*timeStep] = tmpCA[c*P + r];
  }

  // debugging
  //printf("CA is:\n");
  //printArrayDouble(CA,P*T,N);
}

/*
 * Creates a PT x 1 matrix of sensor measurements and propagated inputs
 * Inputs: timeStep: only updating at a given timeStep
 *         y: sensor outputs at current timeStep (Px1)
 *         U: control inputs (all time) (mTx1)
 * Output: YBu (global matrix) (PTx1)
 */
void updateYBu(int timeStep, double* yin, double* Uin) {
  // Initialize temporary arrays
  double AT[N*N];
  double CA[P*N];
  double u[M];
  double Bu[N];
  double CABu[P*T];
  double tmpYBu[P];
  double tmpYBu2[P];

  size_t i;
  size_t j;
  // put sensor outputs into output matrix
  for (i = 0; i < P; ++i) {
    tmpYBu[i] = yin[i];
  }

  // Sums previous inputs up through current timeStep
  for (i = 0; i < timeStep; ++i) {
    // Grab necessary section of U
    for (j = 0; j < M; ++j) {
      u[j] = Uin[ i*M + j];
    }

    //printf("Timestep is %d\n",i);
    //printArrayDouble(B,n,m);
    //printArrayDouble(u,m,1);

    // Perform calculations
    power((timeStep-1)-i, AT);
    //printArrayDouble(AT,n,n);
    multiply(C,  P, N, AT, N, N, CA);
    //printArrayDouble(CA,P,n);
    multiply(B,  N, M, u, M, 1, Bu);
    //printArrayDouble(Bu,n,1);
    multiply(CA, P, N, Bu, N, 1, CABu);
    //printArrayDouble(CABu,P,1);
    sub(tmpYBu, CABu, P, tmpYBu2);
    //printArrayDouble(tmpYBu,P,1);

    for (j = 0; j < P; ++j) {
```

```
      tmpYBu[j] = tmpYBu2[j];
    }
  }

  // Copy tmpYBu into full YBu matrix
  for (i = 0; i < P; ++i) {
    YBu[P*timeStep+i] += tmpYBu[i];
  }
}

/*
 * Given the initial state output from CVX and U (vector of global
inputs),
 * propagate system dynamics to obtain the previous system state
 */
void propagateDynamics(int timeStep, double* Uin, double* x) {
  // Initialize temporary variables
  double u[M];
  double Ax[N];
  double Bu[N];

  size_t t;
  size_t j;
  // Loop through T
  for (t = 0; t < timeStep; ++t) {
    // Grab necessary inputs
    for (j = 0; j < M; ++j) {
      u[j] = Uin[M*t + j];
    }

    // Propagate system dynamics
    multiply(A, N, N, x, N, 1, Ax);
    multiply(B, N, M, u, M, 1, Bu);
    add(Ax, Bu, N, x);
  }
}

/*
 * Raises the square matrix A with a size len x len to the power of t
 * (Currently only applied to global matrix A)
 */
void power(int t, double* AT) {
  // Create temporary matrix
  double tmpAT[N*N];

  size_t i;
  size_t j;
  // Fill the output AT matrix with zeros
  for (i = 0; i < N*N; ++i) {
    AT[i] = 0;
  }

  // Make AT the identity matrix
  for (i = 0; i < N; ++i) {
    AT[i*(N+1)] = 1;
  }
```

```
  // If t is zero we want to return the identity without doing
multiplication
  if (t == 0) {
    return;
  }
  else {
    // Loop through the number of powers desired
    for (i = 0; i < t; ++i) {
      multiply(A, N, N, AT, N, N, tmpAT);
      // Copy tmpAT into AT
      for (j = 0; j < N*N; ++j) {
        AT[j] = tmpAT[j];
      }
    }
  }
}
```

9.5. BILL OF MATERIALS

Table 7 contains the bill of materials, which includes all the flight hardware components and testing equipment purchased in the Fall semester that will be part of the final quadrotor implementation.

Description	Supplier	Supplier Part #	Unit Price ($)	Quantity	Total ($)
Quadrotor accessories					
Quadrotor	GetFPV	n/a, see [11]	743.84	1	743.84
Propellors	Rotorgeeks	TE8X5	6.20	4	24.80
Sensors and batteries					
Lipo battery	eBay	n/a	40.90	1	40.90
9DOF sensor	Amazon	41111921	38.97	1	38.97
USB hub & batteries	Amazon	LYSB01G5W2I8S-CMPTRACCS	25.60	1	25.60
RC receiver	HobbyKing	TX-9X-M2	102.44	1	102.44
Voltage regulator (12 V)	HobbyKing	12V45AUBEC	8.30	1	8.30
Onboard computer					
Jetson TK1	Amazon	n/a	210.36	1	210.36
PCI Network card	Amazon	TG-3269	10.89	1	10.89
Computer Hardware					
Pixhawk controller	eBay	n/a	127.50	1	127.50
Flight controller					

Ethernet switch	Amazon	n/a	27.14	1	27.14
FTDI cable	SparkFun	DEV-09718	30.94	1	30.94
Power connector	eBay	n/a	31.98	1	31.98
Cable (?)	Amazon	AE1032	13.98	1	13.98
Jumper wire cable connector	Tinkersphere	TS-387	2.06	9	18.54

Nets

Netting	Gourock	n/a	165.04	1	165.04

Table 7 | Bill of Materials