12-8-2015

The Development of an Autonomous Proximity Operations Demonstration System

Spencer Watza
Western Michigan University, s.g.watza@gmail.com

Follow this and additional works at: https://scholarworks.wmich.edu/honors_theses

Part of the Aerospace Engineering Commons

Recommended Citation
https://scholarworks.wmich.edu/honors_theses/2651

This Honors Thesis-Open Access is brought to you for free and open access by the Lee Honors College at ScholarWorks at WMU. It has been accepted for inclusion in Honors Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact wmu-scholarworks@wmich.edu.
The Development of an Autonomous Proximity Operations Demonstration System

By
Spencer Watza
Christopher Proctor

A thesis submitted in partial fulfillment of the requirements for graduation with
Honors
from the
Lee Honors College

Examining Committee

Jennifer Hudson, Thesis Advisor
Department of Mechanical and Aerospace Engineering

Robert Trenary, Member
Department of Computer Science

Western Michigan University
December 2015
Abstract

An Autonomous Proximity Operations Demonstration System is being developed to evaluate low cost hardware for aerial robotics and proximity operations in a GPS denied environment. This report highlights the design work and initial prototypes that have been finished along the project goals. The preliminary and detailed design of the vehicle has been completed and initial prototypes constructed. The demonstration system is a multirotor flight vehicle using a hex configuration (6 rotors). The vehicle carries onboard sensing and computational equipment to perform the mission weighing in at 28 pounds. The primary sensor chosen is the Microsoft Kinect™ to perform visual sensing of the environment for SLAM. A model was created in MATLAB® Simulink® to test the dynamics and develop an attitude control system Prototypes of the Kinect Visual Odometry program and of the guidance algorithm were also created and tested.
Disclaimer

This project report was written by students at Western Michigan University to fulfil an academic curriculum requirement. Western Michigan University makes no representation that the material contained in this report is error-free or complete in all respects. Persons or organizations who choose to use this material do so at their own risk.
# Table of Contents

Table of Figures ........................................................................................................... 7
Table of Tables ............................................................................................................... 8
Abbreviations .................................................................................................................. 9
Acknowledgements ......................................................................................................... 10
Preface ............................................................................................................................. 10
Background ...................................................................................................................... 11
  Literature Review .......................................................................................................... 12
Project Objective ........................................................................................................... 13
Project Plan ...................................................................................................................... 13
  Team ............................................................................................................................... 13
  Design Methodology ..................................................................................................... 14
Funding ............................................................................................................................ 15
  Schedule ....................................................................................................................... 15
Mission Analysis & Conceptual Design ........................................................................ 17
  Ground Rules and Assumptions (GR&A) ................................................................. 17
Evaluation Criteria ......................................................................................................... 18
  Concept of Operations ............................................................................................... 18
Mission Requirements .................................................................................................. 20
System Requirements ................................................................................................... 20
Sub-System Requirements .............................................................................................. 21
Preliminary Design .......................................................................................................... 24
  Configuration Selection .............................................................................................. 24
    Multirotor Dynamics ................................................................................................. 26
    Multirotor Sensors and Controllers ......................................................................... 28
    Simultaneous Localization and Mapping .................................................................. 29
Electronic Component Selection ................................................................................... 30
  Integrated System Design ............................................................................................ 31
Propulsion System .......................................................................................................... 32
  Vehicle Frame Design ................................................................................................. 34
Software Architecture ................................................................................................... 35
Robotic Arm .................................................................................................................... 37
Detailed Design .............................................................................................................. 37
  Vehicle Design ............................................................................................................ 37
Flight Configurations .................................................................................................... 38
Weight Breakdown

Software Design

Software Block: Main

Software Block: Communication

Software Block: Image Processing (Visual Odometry)

Software Block: Navigation and Sensor Fusion

Software Block: Mapping

Software Block: Guidance

Software Block: Controls (Position)

Software Block: Controls (Attitude)

Software Block: Motor Mixer

Software Block: Switch with RC inputs

Testing Devices

Simulink Model

Engine Thrust Test Stand

Flight Chamber

Rail System

Manufacturing

Flight Vehicle

Flight Chamber

Rail System

Testing

Testing Form & Plan

Flight Tests

Motor Checkout on Vehicle Frame

Pixhawk Flight Checkout

Testing the Kinect VO prototype

Testing the Guidance Algorithm

Results

Evaluation of the System

Evaluation of the Testing Equipment

Lessons Learned

Group Projects and volunteer workers

Project Scale and Scope

Manufacturing Lessons and Prototyping
## Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The traditional engineering design circle (from nasa.gov)</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Engineering Design Path</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Gantt Chart</td>
<td>16</td>
</tr>
<tr>
<td>Figure 4</td>
<td>NASA TRL Meter related to Space Applications (from nasa.gov)</td>
<td>17</td>
</tr>
<tr>
<td>Figure 5</td>
<td>CONOPS Simplified Diagram</td>
<td>19</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Traditional RC Helicopter (from rchelicopterfun.com).</td>
<td>25</td>
</tr>
<tr>
<td>Figure 7</td>
<td>RC Octorotor sold by 3DR</td>
<td>25</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Figure of Merit: Vehicle Configuration</td>
<td>25</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Multirotor showing toques from motors (<a href="http://blacktieaerial.com/the-physics-of-quadcopter-flight/">http://blacktieaerial.com/the-physics-of-quadcopter-flight/</a>)</td>
<td>27</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Force Body Diagram of a Multirotor in flight (<a href="http://shrediquette.blogspot.com/">http://shrediquette.blogspot.com/</a>)</td>
<td>28</td>
</tr>
<tr>
<td>Figure 11</td>
<td>LadyBird toy quadcopter; fits in the palm of your hand</td>
<td>28</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Comparison between Kinect 1 and 2 from Microsoft</td>
<td>30</td>
</tr>
<tr>
<td>Figure 13</td>
<td>System Integration Design</td>
<td>32</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Motor Equipment in testing setup</td>
<td>34</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Drawing of the frame in SolidWorks</td>
<td>35</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Software Architecture Diagram</td>
<td>36</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Flight Configuration 1 Layout Top View</td>
<td>38</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Configuration 2 without the two rear motor arms</td>
<td>39</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Configuration 3 with robotic arm underneath</td>
<td>39</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Legend for software diagrams</td>
<td>41</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Main Block Software Flow Chart Diagram</td>
<td>42</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Communications Block Flow Chart Diagram</td>
<td>43</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Kalman Filter Diagram</td>
<td>45</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Navigation Block Flow Chart Diagram</td>
<td>46</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Map Block Flow Chart Diagram</td>
<td>48</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Guidance Block Flow Chart Diagram</td>
<td>51</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Position Controller for the Vehicle</td>
<td>52</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Attitude Controller for the Vehicle</td>
<td>52</td>
</tr>
<tr>
<td>Figure 29</td>
<td>Simulink Vehicle Dynamics Model</td>
<td>55</td>
</tr>
<tr>
<td>Figure 30</td>
<td>Motor Test Stand</td>
<td>56</td>
</tr>
<tr>
<td>Figure 31</td>
<td>Picture of the rail system; the motor and pulley are not attached to the rail</td>
<td>57</td>
</tr>
<tr>
<td>Figure 32</td>
<td>The Constructed Vehicle frame from rear facing view</td>
<td>58</td>
</tr>
<tr>
<td>Figure 33</td>
<td>Flight Chamber inside one of the labs</td>
<td>59</td>
</tr>
<tr>
<td>Figure 34</td>
<td>Motor mount block for the rail system</td>
<td>60</td>
</tr>
<tr>
<td>Figure 35</td>
<td>Images from the Kinect; Left is depth and the Right is Depth mapped to Color</td>
<td>63</td>
</tr>
<tr>
<td>Figure 36</td>
<td>Feature Detection with Kinect using FAST</td>
<td>63</td>
</tr>
<tr>
<td>Figure 37</td>
<td>Feature Matching with Kinect using FAST and BRISK</td>
<td>64</td>
</tr>
<tr>
<td>Figure 38</td>
<td>Guidance Test Scenario &amp; Results 2</td>
<td>64</td>
</tr>
<tr>
<td>Figure 39</td>
<td>Guidance Test Scenario &amp; Results 1</td>
<td>64</td>
</tr>
<tr>
<td>Figure 40</td>
<td>Guidance Test Scenario &amp; Results 4</td>
<td>64</td>
</tr>
<tr>
<td>Figure 41</td>
<td>Guidance Test Scenario &amp; Results 3</td>
<td>65</td>
</tr>
<tr>
<td>Figure 42</td>
<td>Guidance Test Scenario &amp; Results 4</td>
<td>65</td>
</tr>
</tbody>
</table>
## Table of Tables

Table 1: Commonly used terms for UAV ........................................................................................................................................................................11
Table 2: A list of possible UAS missions .........................................................................................................................................................Error! Bookmark not defined.
Table 3: List of the members that contributed to the project .................................................................................................................................13
Table 4: List of Ground Rules and Assumptions .................................................................................................................................................17
Table 5: Mission Requirements .................................................................................................................................................................................20
Table 6: List of Systems .........................................................................................................................................................................................20
Table 7: List of the requirements for each of the systems .................................................................................................................................21
Table 8: Flight Vehicle Sub System Requirements ..........................................................................................................................................21
Table 9: Onboard Processing Sub System requirements .................................................................................................................................23
Table 10: Sensing Sub System Requirements .................................................................................................................................................23
Table 11: Robotic Arm Sub System Requirements ...........................................................................................................................................23
Table 12: Vehicle Degrees of Freedom ...............................................................................................................................................................26
Table 13: Selected Components to satisfy system requirements .................................................................................................................30
Table 14: Main electrical components of the flight vehicle weights ...........................................................................................................32
Table 15: Flight Configuration 1 Weight Breakdown .......................................................................................................................................39
Table 16: Flight Configuration 2 Weight Breakdown .......................................................................................................................................40
Table 17: Flight Vehicle Configuration 3 Weight Breakdown ...............................................................................................................................40
Table 18: Telemetry variables .................................................................................................................................................................................53
Table 19: A list of tests that the project needs to perform .................................................................................................................................60
Abbreviations

NASA – National Aeronautics and Space Administration
FAA – Federal Aviation Administration
AFRL – Air Force Research Lab
ESC – Electronic Speed Controller
RPM – Rotations per Minute
Ft = Feet
FMU – Flight Management Unit
PPM – Pulse Position Modulation
PWM – Pulse Width Modulation
DoD – Department of Defense
DAA – Detect and Avoid
BLOS – Behind line of sight
GPS – Global Positioning System
Hz – Hertz
UART – Universal Asynchronous receiver/transmitter
USB – Universal Serial Bus
IMU – Inertial Measurement Unit
DAQ – Data Acquisition
VI – Virtual Instrument
GB – gigabyte
RAM – Random Access Memory
I2C – Inter-Integrated Circuit
RC – Radio Controlled
FEA – Finite Element Analysis
AIAA – American Institute of Aeronautics and Astronautics
WMU – Western Michigan University
Ah – Amp Hours
2D – Two Dimensional
3D – Three Dimensional
Acknowledgements
The team would like to acknowledge all of the support that was provided by individuals at Western Michigan University and elsewhere. Without them, this project would have never gotten off the ground. We would like to especially thank our Committee Chair, Dr. Jennifer Hudson for allowing us to take over a large portion of her lab for the past year. In addition the following individuals for their guidance and support in answer questions throughout the project.

Dr. Robert Trenary
Dr. Damon Miller
Dr. Bradly Bazuin
Dr. Kapenga
Dr. Rho
Dr. Tianshu Liu and Suddesh Woodiga
Adam Polak
Amarnath Raveendran
Christopher Becker

Preface
This report details the work conducted over a two and half year project that satisfied the requirements for Western Michigan University’s undergraduate Senior Design project to graduate from the College of Engineering and Applied Sciences for courses ME 4790 Project Planning and ME 4800 Engineering Project. In addition this work also qualified as undergraduate thesis work for the Lee Honors College to satisfy the requirements for graduating with Honors. Although the project was initially conceptualized in fall of 2013, the work began in the summer of 2014. This report will summarize the work completed through the fall of 2015 but as the project is still on going the final work may differ. To follow the project, check out “watzasolutions.com”.

Background

Unmanned vehicles or systems have been around for several decades, but have often been called robotic systems such as a robotic rover (Curiosity Mars Rover). Unmanned systems are described by their name, they do not have a human onboard. The operators are either controlling the vehicle remotely (in the loop) or monitoring an autonomous operation (on the loop). The term Unmanned Aerial Vehicle (UAV) traditionally described flying unmanned systems and is still used today. However, the Federal Aviation Administration (FAA) and the Department of Defense (DoD) adopted the term Unmanned Aerial System in 2005. Even though the U.S. has selected a preferred terminology, there is little unification across the globe. Table 1 shows a few commonly used terms that are similar to UAS.

<table>
<thead>
<tr>
<th>Commonly used Acronyms/Terms for unmanned Aerial Systems</th>
<th>Aerial Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmanned Aerial Vehicle</td>
<td>UAV</td>
</tr>
<tr>
<td>Unmanned Aerial System</td>
<td>UAS</td>
</tr>
<tr>
<td>Micro Aerial Vehicle</td>
<td>MAV</td>
</tr>
<tr>
<td>Small Unmanned Aerial System</td>
<td>sUAS</td>
</tr>
<tr>
<td>Remote Piloted Aerial Vehicle</td>
<td>RPAV</td>
</tr>
<tr>
<td>Remotely Piloted Aircraft</td>
<td>RPA</td>
</tr>
<tr>
<td>Remote Piloted Vehicle</td>
<td>RPV</td>
</tr>
</tbody>
</table>

In the U.S. UAS have two common sub categories or sub classifications based on size. There are small UAS (sUAS) that are designated by weighing under 55 lbs. The other are called UAS, so we will now refer to them as the “normal” size for clarification throughout this paper.

Unmanned Aerial Systems have become extremely interesting in the past five years for several reasons. One reason is the low cost technology available to the hobbyist consumers allowing them to perform simple autonomous flights that was not previously possible. There is a large push from aerospace and tech industries to allow for integration of these vehicles into the National Airspace System (NAS). The plethora of possible missions that are opening up and becoming more feasible that were otherwise too risky for human life has become a driving factor for this integration. In addition to new missions, older operations can be replaced with UAS to lower the cost and increase efficiency. Table 2 has several generic possible missions for UAS.
Table 2: A list of possible UAS missions

<table>
<thead>
<tr>
<th>UAS Mission Possibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Search and Rescue</td>
</tr>
<tr>
<td>• Wildlife Monitoring</td>
</tr>
<tr>
<td>• Wild Fire Support</td>
</tr>
<tr>
<td>• Border Protection</td>
</tr>
<tr>
<td>• Highway Monitoring</td>
</tr>
<tr>
<td>• Surveillance</td>
</tr>
<tr>
<td>• Infrastructure inspections</td>
</tr>
<tr>
<td>• Law Enforcement</td>
</tr>
<tr>
<td>• Delivery of Packages/Supplies</td>
</tr>
<tr>
<td>• Scientific Observations</td>
</tr>
</tbody>
</table>

However, there are still limiting technological gaps (in addition to the political issues) that are preventing UAS from the full integration into the NAS and reaching their full potential. Out of the many technological issues, one of the primary areas of focus is Detect and Avoid (DAA). In manned vehicles this is performed by the pilot subjectively in order to observe hazards and remain “well clear”. When the pilot is removed a system must be able to perform the observation, however the subjective nature of “well clear” must be replaced with a physical definition.

There has been progress on some of these technological issues. NASA, FAA, MIT, and AFRL are all major partners on a UAS integration into the NAS project. Their research work and progress (so far) has provided significant insight into the systems required by “normal” UAS. A second project called Unmanned Traffic Management has been developed to address the small side of UAS in 2015. While these major projects are working to combine engineering and policy for a final solution, they rely heavily on individuals’ research from the academic and business communities.

**Literature Review**

For universities’, most research is done with sUAS for budgetary reasons and avoiding regulation issues by flying indoors. Initial research into UAS DAA systems pointed towards radar systems, however applications and testing for small scale is difficult. The small size of the vehicles requires finer radar systems which are expensive and complicated. In addition, foam can be invisible to radar, which is a common material for RC scaled vehicles. Investigating other sensors such as LIDAR and ultrasonic showed that they had their own technical limitations. The best option was optical sensors.

Further investigation into aerial robotics and vision sensors shows that by 2013 most vehicles performing complex missions were using external sensors and computer equipment. However, a few academic groups demonstrated the capability of using onboard sensing and computing for simplistic environments. Others have explored using the Microsoft Kinect™ for sensing [Q]. The Kinect™ was marked as an effective sensor by combining color images with depth images at a low cost, but there has been work on other similar systems [H]. One of the pioneering groups with MAV technology and have
developed a groundbreaking work that many others have based off of for open source software is from ETH Zurich. Their work is shown in [N], [O], [P], and [AC] with MAV and autonomous flights using a variety of onboard computing and light weight visual sensors. They are also responsible for developing a sophisticated autopilot system known as the Pixhawk FMU along with a variety of modules that is now the primary open source autopilot system. Other work detailing a variety of work for robotics and the navigation and control of these vehicles are found in [E], [M], [W], [AB], and [AD]. As seen by the dates of these papers a majority of the aerial vehicle work has been done in the past seven years, but is built upon the work from the two previous decades in robotics, image processing, and navigation and control from [A], [C], [F], [I], [R], [S], & [X].

**Project Objective**

This project aims to develop an autonomous flying vehicle in order to create a platform for future work to be performed on Unmanned Aerial Systems and Robotics. The goal is for the team to design, develop and test a flight vehicle and related system to perform a complex mission. The mission is to have the flight vehicle fly autonomously in a GPS-Denied environment with minimal human interactions. By doing so, the team can evaluate the feasibility of advanced aerial robotics with low cost sensors and open source software as well as create a stepping stone for future projects in aerial robotics at Western Michigan University. The works contributed in this report include the design and initial prototyping of systems.

**Project Plan**

This section will start by discuss the history and composition of the team before continuing into the design methodology. Afterwards the project funding and design schedule will be presented.

**Team**

The project started out in the fall of 2013 by Spencer Watza after an intern experience at NASA Marshall Space Flight Center. The initial conceptual design and funding proposals were created and a team was gathered. By the end of the following semester, spring 2014, there were seven team members on the project. However, in less than a year the team was reduced to four. This considerably delayed the project work and part of the work was “contracted” to another design team. Table 3 lists the members that contributed to the project; those who are bolded are still actively contributing. It should be noted that the project goals and schedule were originally designed with all of these team members actively contributing through a time period of a year and half.
Table 2: List of the members that contributed to the project

<table>
<thead>
<tr>
<th>Name</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spencer Watza</td>
<td>Project Lead and Guidance / Navigation Systems</td>
</tr>
<tr>
<td>Christopher Proctor</td>
<td>Control Systems and Modeling</td>
</tr>
<tr>
<td>Nicole St. Louis</td>
<td>Vehicle Design</td>
</tr>
<tr>
<td>Christopher Pleasant</td>
<td>Vehicle Design</td>
</tr>
<tr>
<td>Tyler Pease</td>
<td>Software Development &amp; Sensors</td>
</tr>
<tr>
<td>Joseph Urso</td>
<td>Robotic Arm</td>
</tr>
<tr>
<td>Oseas “Ben” Hudy-Velasco</td>
<td>Manufacturing</td>
</tr>
<tr>
<td>James Jenkins</td>
<td>Software Development</td>
</tr>
</tbody>
</table>

Design Methodology

The design process chosen by the team mimics that which is used by NASA while following the typical design circle as shown in Figure 1 and 2. The design process started out with defining the goals of the project and developing a concept of operations (CONOPS). From this a set of mission and system requirements were derived from the objective and CONOPS to dictate future design parameters. From these design parameters components were selected and a preliminary design was crafted.

Figure 1: The traditional engineering design circle (from nasa.gov)
Figure 2: Engineering Design Path

Applying the above design circle to each of the individual systems in the detailed design, the systems were developed to satisfy the requirements and complete the mission. One key point of NASA’s design process is the review sessions before the project can move forward. The goal of these reviews is to analyze the decisions made by the team to reduce as many risks as possible by utilizing both internal and external experts. The project being worked on lacks experts and the time commitment to do large design reviews although the team discusses any designs that may change the requirements that were defined earlier.

**Funding**

The project was initially supported by an undergraduate fellowship through the Michigan Space Grant Consortium allowing the project to start preliminary design. Additional funding was provided by the Lee Honors College and the Office of the Vice President of Research through scholarships. The total amount that backed the project was $4700 with additional expenditures paid by team members.

**Schedule**

The team had developed several schedules throughout the project’s history that made prediction for the timeline of work. However, many of these were created without fully understanding the depth and complexity of the problem at hand. In addition, as the team shrank in size the project became further behind schedule to about a time of 6 months behind. The Gantt chart in Figure 3 (on the following page) shows a more ideal project timeline that also includes current progress.

Although the project fell behind schedule, a key point is that the plan of the project still holds for the development of the system. However, there are some key dates that are required based on graduation and scholarship follow ups that need to be kept despite being behind. The current team would need 2 more years to finish.
Figure 3: Gantt Chart

|------------|-----------------------------|--------------------------|-----------|------------|-------|----------|------------|----------|-------------|-------------------|---------------------------------|------------|------------|---------|-------------|---------------|--------------------------|-----------------------------|-------------------------------|----------------|
Mission Analysis & Conceptual Design
This section details the conceptual work and mission analysis that led to the requirements for the preliminary design and component selection. First a set of base ground rules and evaluation criteria are established before developing a CONOPS. After the CONOPS is discussed, the mission requirements are crafted. System and Sub-System requirements are derived from the mission requirements and ground rules creating a conceptual design for the entirety of the system and its functionality.

Ground Rules and Assumptions (GR&A)
The ground rules and assumptions are fundamental concepts to be applied throughout the project that help define the actual work being done. Whether the project is theoretical or practical is defined by the GR&As. In Table 4 the following ground rules and assumptions are defined for this project.

Table 3: List of Ground Rules and Assumptions

<table>
<thead>
<tr>
<th>Ground Rules and Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>That there exists previous work on SLAM algorithms that can be</td>
</tr>
<tr>
<td>applied</td>
</tr>
<tr>
<td>That the onboard software must be open source</td>
</tr>
<tr>
<td>That the work will be released via open source</td>
</tr>
<tr>
<td>That the project will be continued after the current students</td>
</tr>
<tr>
<td>leave</td>
</tr>
<tr>
<td>The vehicle must be able to be constructed within the budget</td>
</tr>
<tr>
<td>available</td>
</tr>
<tr>
<td>Commercial off the shelf parts must be used</td>
</tr>
</tbody>
</table>

In addition to the Ground Rules and Assumptions, there is a technology readiness level (TRL) that is described for projects by many government organizations including NASA and the Department of Defense for space vehicles, although this can be applied generally. Figure 4 is a chart that describes the different levels of TLR for space applications.

Figure 4: NASA TRL Meter related to Space Applications (from nasa.gov)
In compliance with above rules and assumptions, the commercially available parts will be qualified as anything TLR 7 or above. To convert from the space application, a TLR 9 can be thought of as having been on the market for longer than a year and tested, while TLR 7 will be considered a new prototype or design released by a company or individual. The tech level was chosen as some parts may be used in non-standard ways for the purpose of the project.

**Evaluation Criteria**

In order to determine the success or failure of the project and system, two types of criteria were selected. The first being the skills and lessons gained from working the variety of systems. The second criteria is the system performance.

The first is crucial for future researchers carrying out similar project work or furthering the specifics of this system design. This becomes especially important when the original contributors are no longer available to communicate with. As there are no easy ways to compare this solution to another, a more subjective approach is required for vehicle mission performance. Some of the key aspects to measure the different projects are to compare basic flight functionality for hovering and simple waypoint navigation with and without the advanced sensor system onboard the vehicle. This can be mathematically measured. The mission testing would be more subjective although time to mission completion could be measured along with internal logs of how the system was processing. Another possibility would be to run the algorithm through a replay system to see if at any point a better flight solution could have been developed given the information, either mathematically or subjectively measured. Some items that would be measured in the flight logs could be number of errors that occurred; number of features mapped, length of missions, processor and ram usage levels, and map size. In addition by stress testing this vehicle through multiple environments, certain environments may cause the vehicle to fail more often than others. A lot of algorithms use benchmarks to compare implementations and designs which are standardized; however there is no common robotic mission to be used to evaluate performance characteristics. This is a topic that should be explored further.

**Concept of Operations**

The specific final mission that vehicle is being designed to accomplish is shown in Figure 5.
The environment for testing is a large laboratory that blocks GPS signals. There are a variety of normal day to day obstacles such as chairs, tables, and desks that the vehicle must detect and avoid while flying the mission. In the mission the vehicle will start from some “home” location that will become its point of origin in the global coordinate system it uses. As the vehicle starts up, initialization parameters will be loaded which include boundary conditions, a search pattern, and a mission. The vehicle will then be cleared for takeoff after all systems checkout.

The vehicle will take off and perform an initial spin to map the surrounding area and provide some initial data for the sensor bias and errors. Upon completing the spin, the vehicle will begin its search pattern within a prescribed boundary, moving from waypoint to waypoint. At any time the vehicle can no longer arrive at a waypoint, the waypoint is canceled and the vehicle moves to the next. During this operation, the vehicle will be updating its environment map and calculate new paths to avoid obstacles.

If the target completes the search pattern and fails to detect the target, the vehicle will return home. However, once the target has been detected by the sensors the vehicle will end the search pattern and plot a new course to the target. The vehicle will arrive at the target and perform the commanded operation which could be to move the target from one place to another or return home with it. The vehicle, after completing the desired mission, will plot a path home and land. Upon landing the vehicle will enter safe mode where the team can extract the logs from the flight systems and turn it off.

Possible targets include empty pop cans, cardboard boxes, and plastic containers. This will help clarify and refine specific systems in the next design sections. Criteria for selecting the target should be symmetric, lightweight, and inexpensive.
Mission Requirements
Analyzing the mission description and concept of operations a list of requirements were developed that are necessary for the vehicle system to fulfill in order to accomplish the mission. Table 5 is the list of mission requirements.

<table>
<thead>
<tr>
<th>Mission Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vehicle shall have a minimal flight time of 10 minutes</td>
</tr>
<tr>
<td>Vehicle shall have the capability to hover &amp; vertical takeoff and landing</td>
</tr>
<tr>
<td>Vehicle shall have a telemetry link onboard to communicate with a ground station</td>
</tr>
<tr>
<td>Vehicle shall be able to detect a prescribed target</td>
</tr>
<tr>
<td>Vehicle shall be able to detect obstacles in the environment</td>
</tr>
<tr>
<td>Vehicle shall be able to navigate around obstacles</td>
</tr>
<tr>
<td>Vehicle shall be able to interact with the target</td>
</tr>
<tr>
<td>Vehicle shall perform all calculations onboard</td>
</tr>
<tr>
<td>Vehicle shall contain all sensors onboard</td>
</tr>
</tbody>
</table>

We believe that the vehicle would need to fly for at least 10 minutes to complete a single mission on a small scale. Ideally, it would be better to have a longer flight time to perform larger and more complex missions.

In order to fly indoors and interact with a stationary target the vehicle needs to be able to hover and take off vertically. In addition the hover mode adds the ability to simulate spacecraft and landers.

Having a telemetry unit will provide additional information to the ground team for scientific objectives and mission monitoring. The safety diagnostic information will alert the team when a major failure has occurred onboard, allowing them to revert to manual control.

To fly in an unknown environment the vehicle must be able to see the environment around itself and accurate map and avoid any obstacles that it encounters. This is also necessary as a way to localize the vehicle while flying in an unknown environment without a position sensor like GPS.

The last three requirements are specifically stated by the project objective and are thus included in the mission requirements.

System Requirements
To convert the mission requirements into a solution a set of systems were created that would each be tasked with solving a specific problem of the mission. These systems will be later on refined to a lower level of sub systems in the next section. In Table 6 is a list of the primary systems that the flight system needs in order to complete the mission.

<table>
<thead>
<tr>
<th>List of Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight Vehicle</td>
</tr>
<tr>
<td>Onboard Processing</td>
</tr>
<tr>
<td>Sensing</td>
</tr>
<tr>
<td>Ground Station</td>
</tr>
<tr>
<td>Robotic Arm</td>
</tr>
</tbody>
</table>
The flight vehicle encompasses all components that make up the UAV but exclude any of non-propulsive electronics for controlling the vehicle. The onboard processing system includes all processing that will occur onboard the vehicle that is not directly performed on the low level firmware for the different components. This includes control, guidance, and navigation of the vehicle. Sensing includes all of the sensors onboard the vehicle in order to perform the operation. Ground station is the system that the team can use to monitor the vehicle which is connected by telemetry. Lastly the robotic arm is the mechanical and electrical systems that are needed to interact with the target excluding the system that processes the information. From here the mission requirements were expanded and put under the specific systems that would satisfy the requirement.

Table 6: List of the requirements for each of the systems

<table>
<thead>
<tr>
<th>List of System Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flight Vehicle</strong></td>
</tr>
<tr>
<td>Shall have a flight time of 10 minutes</td>
</tr>
<tr>
<td>Shall have the capability to hover</td>
</tr>
<tr>
<td>Shall be electrically powered for propulsion (Batteries)</td>
</tr>
<tr>
<td>Shall be able to house the sensors and onboard processing systems</td>
</tr>
<tr>
<td>Shall be able to house all power required for each of the systems</td>
</tr>
<tr>
<td>Shall be controlled by the onboard processing</td>
</tr>
<tr>
<td><strong>Onboard Processing</strong></td>
</tr>
<tr>
<td>Shall be able to generate paths around obstacles in the map</td>
</tr>
<tr>
<td>Shall be able to generate a map of the environment observed</td>
</tr>
<tr>
<td>Shall be able to process any data coming from the sensors</td>
</tr>
<tr>
<td>Shall be able to communicate with the ground station</td>
</tr>
<tr>
<td>Shall be able to estimate the state of the vehicle at a given time</td>
</tr>
<tr>
<td>Shall be able to communicate/control the flight vehicle propulsion system</td>
</tr>
<tr>
<td>Shall be able to communicate/control the robotic arm system</td>
</tr>
<tr>
<td>Shall be able to log information during the flight</td>
</tr>
<tr>
<td><strong>Sensors</strong></td>
</tr>
<tr>
<td>Shall be able to detect the pre-defined target</td>
</tr>
<tr>
<td>Shall be able to detect obstacles in the environment in 3D space</td>
</tr>
<tr>
<td>Shall communicate with the Onboard Processing</td>
</tr>
<tr>
<td><strong>Ground Station</strong></td>
</tr>
<tr>
<td>Shall communicate with the flight vehicle on one of the ISM bands via telemetry</td>
</tr>
<tr>
<td>Shall display information to the team in an easily understood format</td>
</tr>
<tr>
<td>Shall be able to log the information that is received</td>
</tr>
<tr>
<td>Shall be able to send commands to the flight vehicle in operation</td>
</tr>
<tr>
<td><strong>Robotic Arm</strong></td>
</tr>
<tr>
<td>Shall be able to interact with a pop can, jar, and cardboard box</td>
</tr>
<tr>
<td>Shall be able to carry the load of the target</td>
</tr>
<tr>
<td>Shall not hinder the flight vehicle in motion</td>
</tr>
<tr>
<td>Shall not hinder the sensors while in operation</td>
</tr>
<tr>
<td>Shall be controlled by the onboard processing</td>
</tr>
</tbody>
</table>

**Sub-System Requirements**

The system requirements help break down the problem into several fundamental systems to solve each. From here they are broken down further into individual components for that system. The purpose is to bridge the gap between “generic” system ideas to physical components that can be chosen to satisfy the
requirements. For example the flight vehicle is broken down into three distinct parts; propulsion, structure, and power systems. As this system is flying, light weight components are a necessity for successful completion of the mission; however in the following tables when a requirement of “lightweight” is being applied it is meant to say that this component primary goal is to be light weight while other systems may focus on completing separate objectives, weight being derived from other parameters.

**Table 7: Flight Vehicle Sub System Requirements**

<table>
<thead>
<tr>
<th>Flight Vehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Propulsion System</strong></td>
</tr>
<tr>
<td>o Shall be able to produce thrust in order to hover (1.5x Thrust to Weight Ratio)</td>
</tr>
<tr>
<td>o Shall be powered using electrical energy / batteries</td>
</tr>
<tr>
<td>o Shall be controlled using conventional Electronic Speed Controllers</td>
</tr>
<tr>
<td>o Shall be converting mechanical energy into thrust via a propeller</td>
</tr>
<tr>
<td>o Shall be converting electrical energy to mechanical energy via a DC Motor</td>
</tr>
<tr>
<td><strong>Vehicle Frame/Structure</strong></td>
</tr>
<tr>
<td>o Shall be lightweight</td>
</tr>
<tr>
<td>o Shall be able to hold the weight of all systems onboard</td>
</tr>
<tr>
<td>o Shall have room to house all of the onboard equipment</td>
</tr>
<tr>
<td>o Shall have a mounting point for the robotic arm</td>
</tr>
<tr>
<td>o Shall have landing legs</td>
</tr>
<tr>
<td><strong>Power System</strong></td>
</tr>
<tr>
<td>o Shall be a set of electrical batteries</td>
</tr>
<tr>
<td>o Shall be able to provide 10 minutes of propulsive power</td>
</tr>
<tr>
<td>o Shall be able to provide 10 minutes of electronics power</td>
</tr>
</tbody>
</table>

The decision to use electric systems rather than other methods such as nitro or gas is that the vehicle is flying indoors and having this exhaust is dangerous. In addition those systems would add additional complexity to the design for safety and vibration even without the issue of exhaust. The number of 1.5 thrust to weight ratio comes from discussion on hover vehicles. However as this vehicle is not designed for high mobility, the number could have been dropped lower but the 1.5x value allows for some design changes. We decided to separate the flight computer and flight controller onto two different hardware components labeled the Flight Computer and Flight Controller to add redundancy. The Flight computer is responsible for the complex visual processing and mapping that the vehicle will be doing while the flight controller is responsible for getting the vehicle to a designated waypoint. If the flight computer systems fail, the flight controller will still be able to operate independently to control the vehicle from pilot inputs. A robotic arm controller is necessary to operate the interaction task, however is a software component and could be attached to either the flight controller or flight computer. MAVlink was chosen as the software to communicate via serial, as it was designed for Micro Aerial Vehicles.
Table 8: Onboard Processing Sub System requirements

<table>
<thead>
<tr>
<th>Onboard Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Flight Controller</td>
</tr>
<tr>
<td>o Shall be controlling the vehicle propulsive system for flight</td>
</tr>
<tr>
<td>o Shall be able to communicate with the Flight Computer</td>
</tr>
<tr>
<td>o Shall be able to communicate with the Robotic Arm Controller</td>
</tr>
<tr>
<td>o Shall be able to communicate with the telemetry unit</td>
</tr>
<tr>
<td>o Shall be able to generate a state estimate of the vehicle</td>
</tr>
<tr>
<td>o Shall receive commands from the RC Transmitter</td>
</tr>
<tr>
<td>• Flight Computer</td>
</tr>
<tr>
<td>o Shall be generating a map of the environment observed</td>
</tr>
<tr>
<td>o Shall be able to processes data from the image sensors</td>
</tr>
<tr>
<td>o Shall be able to communicate with the Flight Controller</td>
</tr>
<tr>
<td>o Shall be able to processes data from the Target sensor</td>
</tr>
<tr>
<td>• Robotic Arm Controller</td>
</tr>
<tr>
<td>o Shall control the operation of the robotic arm</td>
</tr>
<tr>
<td>o Shall communicate with the Flight Controller</td>
</tr>
<tr>
<td>• Telemetry Unit</td>
</tr>
<tr>
<td>o Shall communicate the required information via MAVlink to the Ground Control Station</td>
</tr>
</tbody>
</table>

Table 9: Sensing Sub System Requirements

<table>
<thead>
<tr>
<th>Sensing</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Image Sensors</td>
</tr>
<tr>
<td>o Shall provide “images” of the environment to the Flight Computer</td>
</tr>
<tr>
<td>o Shall be able to detect absolute depth</td>
</tr>
<tr>
<td>o Shall run at a frequency equal to or greater than 10Hz</td>
</tr>
<tr>
<td>• Attitude Sensor</td>
</tr>
<tr>
<td>o Shall be able to measure angular rates (Gyrosopes)</td>
</tr>
<tr>
<td>o Shall be able to measure accelerations (Accelerometer)</td>
</tr>
<tr>
<td>• Altitude Sensor</td>
</tr>
<tr>
<td>o Shall be able to measure an altitude estimate</td>
</tr>
<tr>
<td>• Target Sensor</td>
</tr>
<tr>
<td>o Shall be able to detect the target</td>
</tr>
<tr>
<td>o Shall be able to communicate with the Flight Computer</td>
</tr>
</tbody>
</table>

The following sensors were chosen based on a basic understanding of flight vehicle state observation along with the additional sensors to observe the environment and target. The image sensor was chosen to be run at a minimum speed of 10 Hz. This rate was chosen as the team believed the vehicle would be travelling at speeds around 10 cm per second, allowing 1 image per 1 cm of movement.
Table 10: Robotic Arm Sub System Requirements

<table>
<thead>
<tr>
<th>Robotic Arm</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Frame</td>
</tr>
<tr>
<td>o Shall be lightweight</td>
</tr>
<tr>
<td>o Shall be mounted to the vehicle frame underneath</td>
</tr>
<tr>
<td>o Shall be able to rotate/translate with 2 or 3 degrees of freedom</td>
</tr>
<tr>
<td>• Gripper</td>
</tr>
<tr>
<td>o Shall be able to grip the specified targets in CONOPS</td>
</tr>
<tr>
<td>• Actuators</td>
</tr>
<tr>
<td>o Shall provide 2 or 3 degrees of freedom for motion</td>
</tr>
<tr>
<td>o Shall be controlled by the Robotic Arm Controller</td>
</tr>
<tr>
<td>o Shall be lightweight</td>
</tr>
</tbody>
</table>

The robotic arm is broken down into three parts, the gripper, frame, and actuators which are all self-explanatory. The decision to mount the robotic arm under the frame is to help with stability of the vehicle (an inverted pendulum is less stable) as well as avoid interference from the rotors and sensors.

Preliminary Design
This section of the report will detail the configuration and component selections before describing the vehicle frame and software design. Lastly it will touch base on the robotic arm design that was handed off to a different groups’ project and the budget that was used for the project.

Configuration Selection
There are two traditional types of flight vehicles, rotorcraft and fixed wing. The requirements state the vehicle must be able to take off vertical which excludes traditional fixed wing designs. Sub categories inside of rotorcraft include helicopter and multirotor designs; two examples are shown in Figures 6 and 7.
Both require rotors to produce thrust and lift but a helicopter has a rotor that can change its pitch and speed in order to provide control. This process is expensive and complex to model for a control system. A multirotor removes the complex parts of varying pitch of the main rotor to have several different rotors all contributing to the vehicle’s motion. Rather than performing complex aerodynamic actions, the vehicle is controlled by generating different amounts of force at each motor to cause rotational moments about the center. The price ends up being similar for a vehicle with similar payload weight and endurance. Helicopters are more efficient than a multirotor by spinning a single larger rotor rather than several smaller ones. The flight profiles end up being fairly similar but the helicopter requires a boom to control yaw, making the vehicle’s profile larger. Figure 8 is a figure of merit describing the general comparison between the two types, each of the criteria are weighted and then a score assigned to each design. The helicopter is assigned neutral values and the multirotor rating is than a comparison between the two.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Weighting</th>
<th>Helicopter</th>
<th>Multirotor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficiency</td>
<td>0.15</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Cost</td>
<td>0.3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Dynamics Complexity</td>
<td>0.4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Flight Profile</td>
<td>0.15</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>1</td>
<td>3</td>
<td>3.8</td>
</tr>
</tbody>
</table>
The multirotor edges out the helicopter in a few areas but only marginally. This is consistent as multirotors have been used by researchers for indoor flights and robotics applications from the literature review. The number of rotors for the multirotor will be selected during the propulsion system design. The following sections now describe the equations of motion, sensors, and controllers used for a multirotor vehicle.

**Multirotor Dynamics**
In order to describe the motion of any dynamic system, it is useful to understand the concept of degrees of freedom. The number of degrees of freedom a system has is equivalent to the number of independent variables the system requires to describe the motion. In general a flight vehicle undergoes six degrees of freedom motion. These six degrees of freedom are composed of the vehicle’s position and attitude; they are listed in the table below.

<table>
<thead>
<tr>
<th>Table 11: Vehicle Degrees of Freedom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roll</td>
</tr>
<tr>
<td>Pitch</td>
</tr>
<tr>
<td>Yaw</td>
</tr>
<tr>
<td>X position</td>
</tr>
<tr>
<td>Y position</td>
</tr>
<tr>
<td>Z position</td>
</tr>
</tbody>
</table>

These degrees of freedom can be expressed in several different frames of reference. The most useful frames to flight vehicle dynamic applications are the inertial, wind, body, and stability frames. In order to relate the variables across these frames of reference we can rotate the axes of one frame onto the frame of interest. This can be achieved by using the dot product and constructing direction cosine tables, which can then be used to construct rotation matrices. Multiplying the rotation matrices by the variables of interest allow us to shift from one frame of reference to the other. The rotation matrix necessary to our dynamic model are below.

\[
L_{BN} = \begin{bmatrix}
C_\theta & C_\psi C_\theta & C_\theta S_\psi & -S_\theta \\
S_\phi S_\theta C_\psi - C_\phi S_\psi & S_\phi S_\theta S_\psi + C_\phi C_\psi & C_\phi S_\theta S_\psi - S_\phi C_\psi & S_\phi C_\theta \\
C_\phi S_\theta C_\psi + S_\phi S_\psi & -C_\phi S_\theta S_\psi + C_\psi C_\phi & C_\phi S_\theta S_\psi + S_\phi C_\psi & S_\phi C_\theta
\end{bmatrix}
\]

The angles \( \psi, \theta, \phi \) are the angles rotated between \( x, y, z \) in the inertial frame and \( x, y, z \) in the body frame respectively.

There is one equation of motion for each degree of freedom of the system, three translation equations, and three rotational equations. Each equation is described individually below.

**Translational Equations**
These three equations represent the translational motion of the vehicle in the body frame.

\[
\dot{u} = rv - qw - g \sin \theta \quad \text{-- Forward acceleration}
\]

\[
\dot{v} = pw - ru + g \cos \theta \sin \Phi \quad \text{-- Side acceleration}
\]
\[ \dot{w} = qu - pv + g \cos \theta \cos \Phi + \frac{1}{m}(T_z) \] -- Vertical acceleration

The values \( p, q, r \) are roll, pitch, and yaw rates respectively. The values \( u, v, w \) are the \( x, y, z \) velocities in the body frame. The variable \( g \) is the acceleration due to gravity, and \( T_z \) is the force due to thrust in the body frame.

**Rotational Equations**

These equations describe the attitude of the vehicle in the body frame.

\[
\begin{align*}
\dot{p} &= \left( \frac{l_{yy} - l_{zz}}{l_{xx}} \right) q r + \frac{l_{xz}}{l_{xx}} (\dot{r} + p q) + \frac{1}{l_{xx}} L \quad \text{-- Roll rate} \\
\dot{q} &= \left( \frac{l_{zz} - l_{xx}}{l_{yy}} \right) r p + \frac{l_{xy}}{l_{yy}} (r^2 - p^2) + \frac{1}{l_{yy}} M \quad \text{-- Pitch rate} \\
\dot{r} &= \left( \frac{l_{xx} - l_{yy}}{l_{zz}} \right) p q + \frac{l_{zx}}{l_{zz}} (\dot{p} - q r) + \frac{1}{l_{zz}} N \quad \text{-- Yaw rate}
\end{align*}
\]

The values \( I \) with subscript are moment of inertia values, and \( L, M, N \) are moments in the roll pitch and yaw direction respectively.

These six equations come from assuming a flat earth approximation as we are neglecting the curvature of the earth because locally the earth is nearly flat. If one is interested in the accelerations, either rotational or translational, of the vehicle in the inertial frame, these equations can be multiplied by the inverse of the rotation matrices above.

For all multi rotor configurations it is beneficial to spin the motors in different directions such that the torque produced by the motors sums to zero. Theoretically this would eliminate the unwanted tendency to yaw. The Figure 9 below shows a simple quad configuration of a multirotor with two of its engines spinning clockwise and two counter clockwise, attaining the desired effect.

![Figure 9: Multirotor showing toques from motors](http://blacktiaerial.com/the-physics-of-quadcopter-flight/)

The forces that act upon a multirotor are much simpler to describe than fixed wing craft. Multirotor dynamics down not involve aerodynamic forces acting on the body. The only forces one must be conscious of are thrust and gravity. This project involves fixed motor angles, and so the thrust is always in the negative \( Z \) direction only. As the vehicle rotates, the thrust vector stays in line with the body frame \( Z \) axis, this rotation causes a component of the thrust vector to act in the direction of another axis in the inertial frame, allowing for translation. See Figure 10 for an illustration.
Multirotor Sensors and Controllers

There are a few levels of standard equipage for a multirotor but they all require a basic flight controller onboard. The reason is that it is not easy to directly take commands from an RC transmitter and perform vehicle motion like a fixed wing vehicle. Thus multirotors need gyroscopes in order to perform angle estimation for control. Increasing the equipage an accelerometer is added. Multirotors that can perform waypoint tracking include GPS for position data, barometer for altitude and a magnetometer for heading. Some even add in down facing cameras for performing velocity flow calculations. Figure 7 shows an example setup of a waypoint tracking multirotor while Figure 11 is of a simple toy that has only a gyroscope. The cost is $30 for the toy compared to $1350 for the waypoint tracking.

The way the controllers work for these vehicles is that a vehicle estimate will be created with some sort of sensor fusion and filtering. This works by fusing all of the sensor information together to get the most accurate representation of the state and filtering out sensor noise. Barometer provides altitude, GPS
provides global relative position through latitude and longitude, and magnetometer provides angle offset from the polar north. The most common implementation of a controller for a multirotor is a form of the PID controller (Proportional – Integral – Derivative). These are often put into cascading forms where the first will control the attitude of the vehicle and then the next controls the attitude rate. By doing this, the controllers will calculate the desired accelerations the vehicle needs to achieve the referenced values. Lastly these accelerations are converted into the forces and torques that each motor needs to produce which is then calculated to a motor velocity based on a transfer function.

The PID controller is one form of linear controllers. It is accompanied by P, PI, and PD controllers. Each controller affects the system response in a different way. Proportional control multiplies the signal by an integer gain, thus the control effort is proportional to the error of the system. PI control adds an integrator into the controller; this integrator is applied to the error. The PI controller has the same effect as the proportional controller but with the added effects of slowing the system response time, thereby reducing or eliminating the overshoot a system might have. The PD controller, of course contains the proportional benefit, but decreases the system response time and generally increases overshoot. PID control is the culmination of all three, by changing the individual values of the P I and D parts, one can tailor a system response to be fast or slow, exerting a large amount of control over what the system output looks like.

When flying indoors the GPS and magnetometer can no longer function accurately due to the interference from the environment materials. A barometer can still function, however the accuracy and precision of one of the quality may not be as useful due to the small differences in height from floor to ceiling. Thus a new set of sensors are needed for the vehicle to provide the position estimates that GPS used to provide along with heading orientations. This is where SLAM comes into play.

**Simultaneous Localization and Mapping**

SLAM is the idea of a system performing both localization and creating a map at the same time in an unknown environment. The papers by Durrant-Whyte and Bailey discuss history and provide an introduction into the algorithm [G]. The difficulty is that both of the state and map require the other to function. The localization helps predict the vehicle’s state relative to the current environment, but the localization needs a map of the current environment. To generate the map, the vehicle must know its state. There are a few different types of SLAM Algorithms and the work on SLAM is numerous, a key component in all situationally aware robotics. The two main types of SLAM are point maps/occupancy grids and Graph methods. Graph SLAM works on relative navigation between nodes in a graph while the point/grid algorithms work to perform global navigation. There are multiple ways that a SLAM algorithm will perform the navigational approach such as utilizing a Kalman filter or particle filter.

The generic idea, as there are many specific implementation on how to perform each of these functions, is to first take a measurement of the environment either through laser scanning, multiple image correlations, or some other method. From this image, features can be extracted which are also commonly referred to as corners and compared between successive images or scans. By analyzing the transition from one image to the other, an estimate of the translational and pose changes can be made. This measurement information is then fed into state estimation. While this is occurring, a point cloud is assembled from the image data to be added to the map. The map takes this new information and adds it to the map based on its current estimated state. In addition, landmarks are created and added to the map so that future features can be compared to these landmarks to see if the vehicle has seen the same object again. By referencing landmarks, the vehicle’s state can be constrained and updated and is called closing the loop. Closing the loop is the idea that if the vehicle saw a landmark that it had previously seen earlier in the trip, perhaps
when it first started, the vehicle could constrain the measurements by stating the “known” position of the landmark and where it thinks the landmark should be currently.

For example by walking through a city a man saw a statue in the city and knew that it was on the corner of two streets and later on after wandering through the city came upon the statue but didn’t know what two streets he was on. He could reference that previously known information to constrain all of his motion since the first time seeing the statue. This is also difficult for a system to do accurately, even if the environment is static. Once the target has been found now the task must be to optimize and correct all of the previous vehicle’s motion in order to find the “true” current location.

**Electronic Component Selection**

From the requirements derived the following components were selected to perform the tasks listed.

**Table 12: Selected Components to satisfy system requirements**

<table>
<thead>
<tr>
<th>Components Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Image Sensor: Microsoft Kinect Sensor (Version 2)</td>
</tr>
<tr>
<td>• Flight Controller: Pixhawk Flight Management Unit</td>
</tr>
<tr>
<td>• Flight Computer: Gigabyte Mini Desktop Computer</td>
</tr>
<tr>
<td>• Electronics Battery: Antigravity XP-10</td>
</tr>
<tr>
<td>• Target Sensor: Android Phone</td>
</tr>
</tbody>
</table>

The Kinect Sensor was chosen because it is a low cost 3D sensor that provides depth data and intensity data (color or infrared images) that can be merged to create 3D point cloud. There are two versions, the first released in 2010 and later updated for Window usage in 2012 after the robotics community began to use it for their applications. The newer Kinect which was released with the Xbox One was released in the summer of 2014. The project had originally planned to use the original Kinect but the newer Kinect performed significantly better. Two studies have been performed recently on the new Kinect [K] and [V].

![Figure 12: Comparison between Kinect 1 and 2 from Microsoft](image)
The cost of the sensor is around $200 and has an API already provided for ease of use. One problem with the sensor is the large cables and the weights associated, the entire bundle weighs about 1200 grams. The new Kinect Sensor requires USB 3.0 and 12V power supply. It also requires at least Windows 8 operating system but in recent months OpenKinect group has released their initial version of Linux/Unix interface for the new sensor, which still has lower functionality. Microsoft provides better tools and examples to work with the sensor that is already calibrated. This is why the Flight Computer was chosen to be a mini desktop as smaller computer systems can only have embedded Linux. The Flight Computer has 4 USB 3.0 Slots onboard to communicate with the variety of systems, processing speed of 3.2GHz and has 8GB of RAM but requires a 19V power supply (similar to laptops).

The Pixhawk FMU is an open source RC autopilot system designed for autonomous applications by the Zurich Lab in Switzerland. The main chip is an Arm Cortex processor and includes a variety of onboard interface ports for serial, PWM, and I2C as well as an onboard barometer and IMU for state estimation. Some accessories added were a 915 MHZ telemetry unit from 3DR, PWM to PPM converter to communicate with the RC Transmitter/Receiver which is PWM signals. A UART to USB cable allows the vehicle to communicate with the Flight Computer and PWM ports for the different electronic speed controllers in the propulsion system as well as the robotic arm servos that drive its motion.

The Antigravity Battery XP-10 is a large Lithium Polymer battery that is encased with circuitry to provide a variety of different voltages and currents. The XP-10 has 18Ah and can supply 19V, 12V and 5V for a weight of 18 ounces. The company has stated that it can provide power on two different ports at the same time thus being able to provide power for the Kinect and Flight Computer. The difficulty was finding the correct adapters to match the ports together.

The image sensor was chosen to be an android smart phone due to the relevance to previous work on the Smart Phone Visual Guidance Sensor (SVGS) that was developed at NASA Marshall Space Flight Center to perform attitude estimation based on a predefined target layout [D]. This also allows the sensor to process its own data and can be linked in a variety of different ways. Of course the Kinect Sensor could also perform the same operation, the idea would be that the android phone could be attached to the robotic arm in order to provide estimates directly in the robotic arm frame and help track’s its own movements. This system could have been connected via USB to the Flight Computer or through USB to I2C conversion for the Flight Controller.

**Integrated System Design**

In Figure 13 the system integrated design is shown with the communication/connections drawn. Some simplifications are made such as grouping the propulsive batteries, electronic speed controllers, and motors together as single blocks rather than showing all six of each.
The image highlights the different connections that are going to be used. The Flight Computer is the primary communication hub for the additional sensors and will perform much of the more complicated SLAM algorithms while the Flight Controller operates the propulsion system, telemetry and robotic arm utilizing the already developed architecture for hardware and software. It should be noted that there are still additional ports available on both the Flight Controller and Flight computer to add additional sensors and systems to the vehicle to expand its capability.

Propulsion System

Table 14 lists the main components that were chosen for the vehicle. This is necessary for designing the propulsion system.

<table>
<thead>
<tr>
<th>Main Component Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Component</strong></td>
</tr>
<tr>
<td>Kinect</td>
</tr>
<tr>
<td>Flight Computer</td>
</tr>
<tr>
<td>Pixhawk</td>
</tr>
<tr>
<td>XP-10</td>
</tr>
</tbody>
</table>

The propulsion system was based around two main design parameters; thrust and endurance. Motors are typically rated in KV with a voltage and max current rating. Thrust is generated based on maximum power output and area covered by rotor motion. Electrical batteries are rated based on the number of cells, the capacity in mAH, and current discharge rate (labeled C). The DC motors are controlled by Electronic Speed
Controllers (ESC) which controls the current direction inside the motor. The ESC is controlled via RPM signals and the Motor, ESC, Rotor can be modelled by a transfer function.

Before motor analysis was done, the team decided on the number of motors to help simplify the analysis as there could be thousands of possible motor, prop, battery combinations that would satisfy. The number chosen was six as four motors seemed unlikely to lift the large weights initially calculated without using massive rotors while eight was too many. For propellers, multirotor props were investigated and it was determined that large diameter and shallow pitch is the best for hover-like flight. Higher pitch means that the rotor will travel farther through the air on each rotation. Finding large and inexpensive propellers is difficult especially with multirotors as most are designed for smaller payloads. A list of available Multirotor props was taken from Advanced Precision Composites (an RC propeller manufacturer) as both directions of rotations are needed for the props that do not come in normal sizes. These were used in the analysis.

The calculation for flight time is the capacity divided by the current. By reiterating through combinations of motors and batteries along with their weights on E-Calc (an online motor calculator for hobbyists) the choice was narrowed down to a few motors. E-Calc was used as it has an assortment of motors, props, and batteries that have been tested and inputted in the program along with using accurate (within 10% error) propulsion models. This has been used by the AIAA student group for several years and has been able to confirm the reliability. A MATLAB® script was also used to get initial estimates of thrust to weight ratio, power required, and cost analysis to compare with E-Calc. These were further reduced by weight and cost. Two motors were then purchased for testing. The maximum current was chosen to be at 70 Amps as that is typically the max normal ESC value before having to buy custom tailored components which increase the cost. The selected battery cell size was 5S with lithium polymer which equates to 18.5 to 20.1 volts for nominal voltage. The propeller chosen was 14x5.5 plastic.

The two motors tested were the Turnigy G46 and the Turnigy AX-4008Q. The G46 hooked up for testing is shown in Figure 14. The AX-4008Q motor ended up over heating from too much current draw and produced a maximum of 4.5 pounds of thrust. The G46 peaked at over 7 pounds of thrust. However the AX-4008Q was light weight. The G46 was decided due to the higher reliability, durability of the design and larger excess power.
Vehicle Frame Design

A key aspect for multicopters is that the frame exhibits symmetry to eliminate products of inertia which complicate the vehicle dynamics. Often this comes in the form of several arms protruding from a central circularized structure that houses all of the electronics and batteries. However many of these do not include forward facing cameras and the large amount of electronics so a less conventional approach was taken.

The frame will still have arms that stick out from the main part of the vehicle; however the main part or the base of the vehicle will no longer be circular and instead be a long plate that the equipment can sit on. This serves two purposes, allowing the vehicle to be easily setup and removing complexity from mounting devices. Each of the arms will require a motor mount structure. Lightweight properties are important so the base plate should be as thin as possible as it will take up a large area. To support this, two spars will run along the length of the vehicle with the motor arms sitting on top of these. The plate is on the bottom as it will be the primary load carrying part and the CG will be lower which tends to be more stable than if it was flipped. This also reduces interference from the rotors and the equipment. A front piece of the plate will protrude outside of the spar zone so that the Kinect™ sensor will be able to have a clean line of sight. Figure 15 shows the frame concept.
The CG is designed to be under the middle bar which allows for the proper dynamics to be assumed for the vehicle dynamics. The two large sections in front and behind allow for tuning of the CG location with the variety of hardware and batteries on board to balance with different configurations.

Software Architecture
The original design for the software architecture was to use pre-existing open source software that had developed different SLAM solutions for a variety of sensors. This would simply have been a matter of reconstructing an example with Kinect V2 sensor and connecting pipes for the outputs to the Flight Controller. There are a few different sources of open source SLAM algorithms, however Mobile Robotics Programming Toolkit (MRPT) was the most extensive and most updated one with Kinect examples. Unfortunately the team was unable to get the software to compile with the new Kinect sensor, thus a new solution was proposed.

The new solution was to build a specific SLAM solution from the ground up utilizing Microsoft, OpenCV, and other generic helpful libraries. The system was broken down into different blocks each performing a specific set of similar algorithms. These blocks and their connections are shown in Figure 16.
Figure 16: Software Architecture Diagram

On the right side of the Figure are the flight controller blocks and the left side are the flight computer blocks. The architecture design expands on what was explained for the basic multirotor by replacing the advanced navigation with the SLAM algorithm pieces utilizing a Kinect Sensor instead of GPS and a magnetometer. The information flow is as follows. The Kinect provides images of the environment in a set of streams which are processed in the image processing or Visual Odometry block. This block extracts features from the images to compare to previous images in order to calculate relative pose changes in the vehicle. In addition, this image processing block could accommodate future functionality in the form of gesture tracking or other things. The image processing also creates a point cloud of objects detected for the map. The features and map are then passed to the map functionality which adds the point cloud to the existing map and creates global landmarks for future reference. This map information is then used for the guidance algorithm. The pose measurement is passed to the advanced navigation block which uses a Kalman filter to perform state estimation. This state estimation is passed to the guidance algorithm so that the vehicle understands where the vehicle is in relation to its target. It runs an iterative algorithm that refines an optimal pathing solution to the targeted waypoint from a list of waypoints provided by the mission settings. The guidance algorithm generates a set of vectors that are passed to the control system to
be performed. Inside the mapping block there is also an optimization function which will constrain the vehicle once the loop has been closed with landmarks.

**Robotic Arm**
This part of the project was delegated to a different group of students for senior design; their report was unavailable at this time for reference. They were given information of the overview of the project as well as the design requirements. Their objectives were to build the robotic arm, write the control software, and implement the targeting sensor. A discussion of their work on how it relates to the project can be found in the results section of this report.

**Detailed Design**
This section discusses the design on the vehicle frame, component layout for different configurations and software onboard the vehicle in greater depth. The software sections correspond to blocks software architecture diagram. In these sections a diagram is shown along with a brief explanation of the algorithm functionality.

**Vehicle Design**
The design of the vehicle was focused around being able to geometrically fit and support the weight of the onboard systems, be affordable while having as little structural weight as possible. The total area required to seat all of the systems is 312 square inches. The basic geometry of the craft was selected because it offers a simple easy to machine alternative to the traditional circular geometry.

To meet the need of a light and affordable material that could support the weight of the electrical components birch wood was selected. Birch wood is a material that has been used in many hobbyist flight vehicle structures, proving it capable of meeting our needs. The WMU AIAA has also used birch wood multiple times and could recommend this material for the vehicle, being able to attest to its suitability, having used it themselves in the structures of aircraft. The base plate is 1/8th inch thick and was laser cut offsite. As shown in the Figure X to reduce the weight of the base plate further rectangular sections were cut out. It was determined beforehand that these cutouts would not endanger the integrity of the craft, and therefore were an effective way of reducing structural weight.

Carbon fiber tubing was selected to support the main birch wood base plate of the structure. Carbon fiber is both lightweight and strong adding the minimum weight to the vehicle for the structural support it offers. The carbon fiber tubes have a square cross section, making it easier to mount the motors, and easier to attach to each other and the base plate of the vehicle. The carbon fiber was bought as a long single tube, and was cut to size on site to meet the requirements of the vehicle. It was cut by hand and is not nearly as accurate as the base plate dimensions. This tolerance was allowed as shipping the fiber offsite to be cut would have been outside of the budget.

Being a thin plate the birch is susceptible to bending loads normal to its primary surface along the x-axis direction. The rectangular area provides a simple shape that can be supported along its length by easily attainable linear carbon fiber elements, thereby greatly increasing the base plate’s resistance to shear and torsion loads. In addition to structural support, this geometry allowed for easily placed motors, using only three hollow carbon fiber tubes as motor arms. This particular geometry does unfortunately limit the arrangements of some of the components, making it difficult to completely erase the products of inertia, because not all of the components have direct counterparts that can to be placed on board symmetrically across the two pertinent axes.
The final vehicle related device to be considered are the motor mounts. The mounts like the other materials, need to be strong and light with the addition of being able to fit over the square ends of the tubes. Cubical mounts with platforms on the upper surface were designed to meet this need. Made of the same birch plywood as the plate, these pieces were laser cut such that the top plate had slots, and the three side plates had notches. The notch/slot design allowed the mounts to be cut offsite and assembled onsite without variation between each mount. Each mount is cut with a hole in the center of the platform to accommodate for motor, and with a single hole through each side plate that fashions it to the carbon fiber rod.

Flight Configurations
There are several flight configurations used for testing purposes. Each of these configurations differ by geometric location of components, and the amount of components onboard. Each configuration was constructed such that the center of gravity (CG) existed below the Pixhawk. This location for the CG was chosen so that the body frame originated at a point where accelerations were known. Placing the CG at a point where accelerations are known gets rid of the need to calculate relative accelerations for the dynamic model, making the task of simulation and control system building simpler. It is important to note that the motors, which are symmetrical in every configuration, do not move the CG from the center of the craft, where the Pixhawk is located. Each configuration is displayed in the pictures below.

Configuration 1 consists of the vehicle with basic flight configuration and includes the propulsion system and the Pixhawk. This configuration is used to fly the vehicle by hand and verify that the batteries, motors, esc, transmitter and Pixhawk can all communicate. In this configuration the batteries are the primary weight components and need to be properly balanced on parts of the frame. This configuration is shown in Figure 17.

![Figure 17: Flight Configuration 1 Layout Top View](image)

Configuration 2 includes all systems but the robotic arm. In this configuration all the batteries are located in the rear, in a 4-2 pattern. The 4 batteries are closer to the vehicle center and two further away. The batteries can all be moved to the rear due to the addition of the Kinect™, flight computer, and electronics battery all placed in front of the CG. This configuration is shown in Figure 18.
Configuration 3 adds the robotic arm to configuration 2. With the addition of the robotic arm ahead of the CG, a larger moment arm in the rear is required. To achieve this balance, it was as simple as switching the position of the 2-4 pattern. With the 4 battery set father from the vehicle center and the 2 battery set closer to the center, opposite of the last configuration. This configuration is shown in Figure 19.

Weight Breakdown
A lot of weight is added for the large cables that are provided for the USB to UART and Kinect Sensor. In a more optimal solution, these wires would be trimmed to be the minimal weight required. However that increases the risk especially with the Kinect Sensor that could damage the sensor or ruin the cable. These cannot be bought individually so it would require repurchasing the entire Kinect bundle again. Some of the wires are much easier to clean up however it was decided at the current phase of the project it was not worth the effort to trim the wire lengths. Tables 16, 17, and 18 show the weight breakdown for the different configurations (with varying levels of equipage) for the flight vehicle.
### Table 14: Flight Configuration 1 Weight Breakdown

<table>
<thead>
<tr>
<th>Component</th>
<th>Weight (lbs)</th>
<th>Weight (kg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spars</td>
<td>2.2968</td>
<td>1.044</td>
</tr>
<tr>
<td>Base Plate</td>
<td>0.792</td>
<td>0.36</td>
</tr>
<tr>
<td>Motor Mounts</td>
<td>0.7128</td>
<td>0.324</td>
</tr>
<tr>
<td>Motors</td>
<td>4.8664</td>
<td>2.212</td>
</tr>
<tr>
<td>ESC</td>
<td>1.1924</td>
<td>0.542</td>
</tr>
<tr>
<td>Batteries</td>
<td>9.3236</td>
<td>4.238</td>
</tr>
<tr>
<td>Pixhawk</td>
<td>0.2904</td>
<td>0.132</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>19.4744</strong></td>
<td><strong>8.852</strong></td>
</tr>
</tbody>
</table>

### Table 15: Flight Configuration 2 Weight Breakdown

<table>
<thead>
<tr>
<th>Component</th>
<th>Weight (lbs)</th>
<th>Weight (kg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config 1</td>
<td>19.4744</td>
<td>8.852</td>
</tr>
<tr>
<td>Kinect</td>
<td>2.4816</td>
<td>1.128</td>
</tr>
<tr>
<td>Flight Computer</td>
<td>2.134</td>
<td>0.97</td>
</tr>
<tr>
<td>XP-10</td>
<td>1.5796</td>
<td>0.718</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>25.6696</strong></td>
<td><strong>11.668</strong></td>
</tr>
</tbody>
</table>

### Table 16: Flight Vehicle Configuration 3 Weight Breakdown

<table>
<thead>
<tr>
<th>Component</th>
<th>Weight (lbs)</th>
<th>Weight (kg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration 1</td>
<td>19.4744</td>
<td>8.852</td>
</tr>
<tr>
<td>Kinect</td>
<td>2.4816</td>
<td>1.128</td>
</tr>
<tr>
<td>Flight Computer</td>
<td>2.134</td>
<td>0.97</td>
</tr>
<tr>
<td>XP-10</td>
<td>1.5796</td>
<td>0.718</td>
</tr>
<tr>
<td>Robotic Arm</td>
<td>2.2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>27.8696</strong></td>
<td><strong>12.668</strong></td>
</tr>
</tbody>
</table>
**Software Design**

The following sections are going to discuss the design for each of the following major components in the software architecture diagram. For the Flight Computer there are several diagrams representing the flow of the program and they share the same legend shown in Figure 19.

![Legend for software diagrams](image)

**Figure 20: Legend for software diagrams**
The Main block is where the process starts for the program on the Flight Computer. The first thing this program does is to load the information for the mission and the systems being utilized from their respective files, Mission Config and System Config. From here the system will initialize the parameters such as what sensors are being used, what configuration files each algorithm should use, etc. Then the program is to create the threads for each of the main block. The five blocks it spins off is the Communication, Guidance, Navigation, Visual Odometry, and Mapping. These threads will then start the main blocks and look for flags from an exit status if they crash. From here the program runs its main waiting loop looking for a flag or message indicating that an error has occurred or that the mission is finished. If there is an error it will try to fix it with a certain amount of attempts before declaring it a failure and setting the quit function to be true. Once the mission is finished it will begin to clean up the process by ending the different blocks.
Software Block: Communication
This block is the communication system for the Flight Computer to talk with Flight Controller and publish internal messages from Flight Controller. This can be expanded for additional communications with other hardware components. The diagram below in Figure 22 shows the process flow.

Figure 22: Communications Block Flow Chart Diagram
When this algorithm is called the first thing it does is initialize the different parameters specified to it for it such as the Ports, data rate, and log file location. From here it will spin off several threads to perform three separate tasks. A second possible way for the threading is to only do it when a message has been received or is ready to be sent. After creating the threads, the block loops while it is not told to quit and there are no errors. It checks to see if there is a message from the Pixhawk (and/or other systems) and sends that message to the decoder. The decode splits and parses the data into usable information based on message type and then publishes them to the internal message queue. The main function then continues to see if there is a message ready to be sent out, which is checked by the encoding message loop which processes the internal out message queue, encodes them into a transmittable format, and then provides the packet and flag to the main. The main will then send this message out. Finally if the log has available parts in its own queue it will write those out. When an error occurs the main loop will be exited and a message will be published internally. The system may try to fix the issue internally here or back in the main loop. If the system finishes and quits, it will finish publishing all messages in the queues and then clean up the block. This will be the last sub block to clean up.

Software Block: Image Processing (Visual Odometry)
This block’s main responsibility is interpreting the data provided by the Kinect Sensor and extracting useful information for the vehicle. The desired outputs of this block are image features that can be tracked, 3D point cloud of the environment, and a relative pose estimate.
The first part of the block’s objective decides which data streams to use out of the available ones. At certain times, a data stream may be providing erroneous or unnecessary data. For example in a large open area, the depth data may return nothing that is usable. The first goal is to analyze the different streams to determine which ones will be the most efficient, as done in [T]. Likewise, the following cases are acceptable; only depth, only intensity, and combined.

For only depth images, scan matching will be performed. This is the technique used with LIDAR systems for feature extraction. By making a graph for each row of the depth image, they can be compared between each other to find the translation and rotation.

For intensity images, the process is different but follows the same idea of finding features and then comparing them between successive images to find the translation and rotation of the two images. First the image is converted to gray scale and smoothed using a Gaussian function. The next step is to find features which are also called corners. A corner is a point where there is a maximum or minimum in the second derivative of image intensity. A list of the features is compiled that satisfy the criteria of proximal separation and intensity threshold created by the user. A descriptor is then generated for each of the features so that they can be compared between images.

To compare a current image with the key image, the descriptors are compared in a brute force algorithm with mutual exclusion. This means that features can only have 1 true match (those with multiple matches are excluded). A list of corresponding features between the two images is created and passed onto the next step of calculating the pose estimate. This is where the 2D only image and 3D combined image diverge. The 2D image utilizes a 7 point model with RANSAC and 3D image comparison is done through RANSAC [L] & [T] utilizing a single value decomposition model for calculating the translation and rotation between the two frames. This provides a relative pose change between the key frame image and the current image.

In addition, this block provides the depth data when available as a 3D point cloud to the Map block so that the system can learn more about its environment to plan more efficient paths and avoid obstacles. Finally, key frames and their associated features / descriptors are saved into a keyframe list on a file.

**Software Block: Navigation and Sensor Fusion**

This section is about the advanced navigation and sensor fusion and not the basic navigation filter used on the flight controller. The idea is to use an extended Kalman filter for estimating the state from measurements.

Kalman filter is a type of linear estimator algorithm that revolves around the principle of taking a series of measurements observed through time which contain inaccuracies to provide an estimate of the state variables of the vehicle that are more precise than using any one single measurement.
As seen in Figure 23, the algorithm works in two steps; the prediction step and the measurement or update step. The prediction step takes the current known information about the state from previous estimates along with the estimated previous error to predict the future of the vehicle’s states along with the uncertainties. When measurements become available, these estimates are updated giving higher priority to those with less bias. The extended Kalman filter is an extension to nonlinear system, the idea being that a set of matrices are calculated every step interval that effectively linearizes the system.

The difficulty is that the different sensors all have different frequency rates and then there is the time lag for computations and communication transport which greatly affects the filters ability to perform. In order to account for this, the filter will run at the lowest rate, for us is the image processing algorithm. During the time the image is not available a buffer will be created of all of the measurements recorded. When a pose is received from the image algorithm, the time stamp is compared to the buffer to find the closest measurements.

In addition, the navigation is performed relative to each key frame. When a new key frame is created, the Kalman filter records the covariance matrices and the current pose associated with the frame is passed to the map block. This will be discussed in the following section. Figure 24 shows the flow chart for this block.
Although, this figure is a very simple flow chart this process is mathematically intensive with the Extended Kalman Filter implementation. First part of this block does, like the rest, is initialize parameters by reading from a specified navigation configuration file. This also determines the bias parameters and other information needed to understand the sensor errors and help tune the filter. While the program has not been told to quit the system will first wait for a pose estimate from the VO block. During this waiting process the system will store the other sensor data into an internal buffer. When the VO state is ready, the VO pose measurement will be synced as closely as possible with the rest of the sensor data to perform an accurate state estimation. Also depending on the type of data coming from the pose (2D, 3D, and 2D Scan) the correct bias for this calculation is needed. From here a predicted state is calculated which is than
updated with the measurements. Afterwards the state is published and is compared to criteria specified for creating a new keyframe. If the criteria are met, a new key frame is generated and the covariance and keyframe is published. In addition the filter properties are reinitialized which brings the block back to the beginning of the loop. When the block's task is completed it will publish the last covariance and clean up.

Software Block: Mapping
By taking the information associated with each keyframe (state, time, & features) a node on a graph algorithm can be generated. Each length between nodes are the estimated uncertainties that were produced between the keyframes. This is useful so when the system recognizes that there is a loop closure it can perform optimization on the graph to constrain the nodes and then update the map itself.

This block is also responsible for building the data files for the landmark lists, keyframe lists, and the map itself. The map cannot be contained in memory as it would be too large thus is stored in files. In a directory on the computer there would dozens of files that are designed to partition the entire map environment into small sectors. The resolution of the map was decided to be 2cm so that small sensor errors could be self-contained in individual nodes. The file specifies a specific area on the map and a Z plane in binary format. Each byte in the file represents the value of the specific point on the map having two possible values in the current design (although additional values could be assigned) for null / empty and filled. The location inside the file is the X and Y coordinate specified by new line characters.

For example every 50 bytes in a file represent the width of a row; where rows are the X value and Y are the columns. If the coordinate (32, 49) was desired it could be calculated based on this simple formula; \( P = X \times \text{width} + Y \) with X being 32 and Y being 49. The Z value would be specified by the file. These points may be then translated form their sector coordinates to a global coordinate in the map functions. For the landmark lists and keyframe lists, a single binary file is used with fixed length fields separated by new line characters between each keyframe and landmark. These files are then indexed onboard the system based on global position and sector so that searching for nodes is faster.

Figure 25 is the diagram showing the process for the map block algorithm.
Figure 25: Map Block Flow Chart Diagram

The first part of this block initializes a map based on the boundary conditions provided by the main configuration of the program. From there the program loops while it has not been told to quit. In this loop the program waits for data from the visual Odometry program and a state to correspond with that set of data. If the keyframe flag has been set, then this information will be stored into the keyframe list. From there two threads are created, one to compare features between current image and from previous created landmarks. From there a new set of landmarks are taken. Meanwhile the point cloud for the map is being created in the global reference frame and then the map is being updated with the new values. If there was significant matching between current features and landmarks than optimization will be performed to update and constrain the pathing of the vehicle.

Software Block: Guidance
There are many different guidance algorithms available for use. The most common rely on graph theory derived from Dijkstra’s graph algorithm created in 1956 for finding the least cost path between nodes.
Some of the more common are A Star which is anytime planning which adds heuristics to help shape the direction of progression speeding up the calculation and D star which performs real-time planning in dynamic environments, updating the path whenever the environment affects the path between nodes.

There have been many adaptations of these two algorithms to optimize specific problems but one that is important to note is AD star, the combination of A star and D star. The Anytime Dynamic Star re-planning algorithm takes the heuristic approaches in A star and combines the re-planning elements of D star to create optimal solutions in a dynamic environment. There have been improvements since this algorithm was published in [Y] and [AN], but as the algorithms become more adaptive, their complexity increases even with provided pseudo code. This algorithm was created by combining D star Lite(a simpler version that performed as well as D star) and A star that would generate a less than optimal solution at first and over time refine the solution, refining the outputted trajectory. The idea was to move through the nodes that were on the top of the priority queue by key. The keys were calculated based on cost to arrive at that node as well as the minimum look ahead cost for the next node. The goal was to check consistency of the nodes, that the look ahead value matches the actual cost to arrive at that node and when it was consistent it would be moved into a Closed List. Over time the path would move heuristically toward the finish point and would end there supplying a list of points. The exact details can be found in [Y] paper along with the pseudo code. The implemented prototype used backwards path finding (starting at the target and proceeding to the start) which can be confusing at times.

Some adaptations were added to the program due to the system used for the map and nodes. This algorithm performs a comparison with all of the nodes in the map to evaluate if any of them changed. Another alternative is that whenever the map is updated in the defined sector, the system will perform an update. In addition, rather than having the entire point cloud loaded into memory (which in large scenarios is physically impossible) the algorithm will take the target and start point to create a vector between the two. From here, a buffer is established adding vector width so that only a small sector of the map is loaded around the start and target and the points in between. Whenever the path fails to arrive at the start or end, the algorithm will expand the sector size. This expansion can be tuned and made more adaptive depending on where the algorithm fails but is not explored further in this report.

After receiving a list of points from the guidance algorithm for the path, these points are then converted into the minimal number of vectors possible. During this calculation the vectors are also checked for consistency with the environment point cloud, making sure that none violate vehicle buffers. These lists of nodes are then passed to the control system as sub-waypoints. Because of the relative basis for the navigational state estimates, the waypoints must be converted into the local frame before being passed to the controllers. This can be problematic when the navigation algorithm resets for a new reference key frame and there is lag that affects the waypoints from being properly converted, IE the sub-waypoint that is desired is still be defined in the previous coordinate frame. To solve this, the algorithms need to be careful and check for coordinate synchronization which a possible implementation would be an integer descriptor that describes which coordinate frame node it is referenced to.

Figure 26 shows the block diagram for the flow of the algorithm on the following page.

The first part initializes parameters for the guidance algorithm such as the starting optimality of a path, buffer size for the vehicle during pathing calculation and cost functions. From here the mission objectives are loaded from a file and a set of waypoints are generated and added to the waypoint list. The main loop continues for this block while there are still waypoints left. The first part of the process is to grab a small section of the map called a sector. This is done to minimize the number of points brought into the algorithm. To do this, the block will spawn off many threads to each deal with a specific file that contains a
portion of the sector. From this data a list of nodes will be created. The next part is to calculate the path. If no successful path is created, the sector size will be expanded until it hit a predefined limit in which the waypoint is no longer valid. Once a valid path is discovered is passed on to a vectorization thread through threading. This vectorization converts the list of grid points into vectors for the vehicle. This brings the algorithm to the next internal loop which waits for the current node to be the goal node. The current node is calculated based on the state of the vehicle. While the vehicle has not reached the waypoint, the system will check to see if the map has changed while flying or if a more optimal solution exists. If the change to the map is large the vehicle will replan from scratch. If there is only a small change then the efficiency parameter will be increased making the path less efficient so that the path can be refined quickly, although this may really only be necessary if the map invalidates the path or is relatively near the intended path. Finally if there were no changes then the path will increase its efficiency parameter to find a more optimal solution. The path is then recalculated mimicking the loop earlier for expanding the sector if no valid solution was found. When the waypoint is reached, a waypoint is removed from the list and a waypoint reached message is published. Once the algorithm has finished all of the waypoints and completed the mission the system will publish the message completion message and begin cleanup once the main has responded to the finished message.
Figure 26: Guidance Block Flow Chart Diagram

Software Block: Controls (Position)

The simple position controller can be viewed in figure 27 below.
Figure 27: Position Controller for the Vehicle

Starting from the inputs on the left, the measured values are subtracted from the commanded values and passed into the controller. The controller output is then subtracted from the known velocity and passed into the velocity PID controller. The output of each of the velocity controllers is then fed into the upper level block diagram for processing.

PID control was implemented in the position controller because a faster response to position is needed to help accommodate the lower frame rate that the optical system and software is able to process. Employing a PID controller allows for the rise and settling time of the system to be easily manipulated through the I and D values.

Software Block: Controls (Attitude)

The attitude controller minimizes the amount of error between the requested value of an angle, and the measured value. This controller was built in Simulink software, and then embedded onto the computer after being encoded in C. A diagram of the attitude controller is shown below.

Figure 28: Attitude Controller for the Vehicle
As seen from Figure 28, the currently measured roll, and pitch values are sent into the diagram from the left. These values are fed into summing blocks that take the difference between the measured and the commanded value before feeding the resulting error value into the PI controller. The value the controller outputs is then differenced from the measured angular rates of the vehicle. The resulting error is fed into another set of PI controllers to be minimized. The values from the set of rate controllers flow into the motor mixer which outputs PWM values for each motor individually.

The attitude controller consists of cascaded PI controllers. PI control was chosen because it slows the response of the system and reduces overshoot. The vehicles attitude controller does not require a fast response, because the vehicle will be translating very slowly, using small angles when rotating. As stated previously the PI controller also reduces overshoot, essentially driving the system to meet the commanded angle without going past it, reducing the amount of oscillations in the system.

Notice that the pitch and roll controllers are of the same value, while the yaw has a much lower I value. This drastic difference in values is a result of the geometry of the vehicle. The moment of inertia about the Z axis is much larger than the other two values, meaning the vehicle needs more torque for a given angular velocity around that axis. This directly influences the values of the controllers. The controllers were tuned such that they did not respond too fast, in case the accelerations cannot be achieved safely, or too slow, as the vehicle would take too long to change attitude. Once leaving the motor mixer the data goes into the upper level block diagram for processing.

**Software Block: Motor Mixer**

A motor mixer for this vehicle was modeled after a motor mixer written for another multirotor vehicle [AD]. The function of the motor mixer is to describe the relative amounts each engine will contribute to each type of attitude change and then calculate a motor speed and PWM signal required for each motor. That PWM signal is then sent to the affected ESCs which generate a specific rpm for each engine. The motor mixer code is not entirely intuitive because it describes the amount of influence each motor has relative to each other over the control of the system. For example:

\[ M_1 = \left( \text{Pitch} + \frac{\text{Roll}}{2} - \text{Yaw} \right) \]

There are two motors that produce positive pitch and two that produce positive yaw. Thus for this example the motors relative contribution to these two rates is half, thus they are divided by two. Because all six motors have a contribution to yaw it is not necessary to divide by two. The motor also has a negative contribution to yaw. This way of thinking can be applied to all six motors. As the state of the vehicle is updated the motor mixer will continuously update the PWM signals appropriately.

**Software Block: Telemetry**

The telemetry link with the ground station utilizes MAVlink protocol which is common for most of the autopilot systems out there. The only difference is what information is being sent to the ground station. The following list is the information being transmitted to the ground station and vice versa.
## Software Block: Switch with RC inputs

The system has a major design challenge being that there are two separate computer systems that must interact and communicate with another to perform to its full ability. However, this does provide some safety redundancy in that if the complex algorithms fail it doesn't have to affect the Flight Controller. The idea is that the vehicle has several different flight modes that can be toggled through the ground station with the telemetry unit or the radio controller transmitter. The three flight modes are manual, auto, and mission. The manual flight mode operates like a normal multirotor requiring the user to provide the control inputs to dictate the attitude of the system. Auto allows the user to no longer be using the stick to fly; the vehicle is now reliant on the sensors onboard to hold position. The last mode is mission which tells the vehicle to start the mission parameters and begin its own operation without pilot input. If mission fails for some reason the vehicle will first try to switch to auto mode to hover and then perform an emergency landing.

Mission mode might fail when a major function on the Flight Computer fails and an error status message is sent to the flight controller. It could be that the Kinect Sensor has timed out or some other issue has been reported. Depending on the status the system may try different options while it corrects the problem, for instance the Flight Computer may spend some time reinitializing the algorithms and reset itself or reconnect with the Kinect. The vehicle may hover while this is performed. If the communication is lost entirely then the system will switch to manual mode. By switching modes, the internal system will no longer rely on state estimates provided by the flight computer and use its own internal less complex filter to still allow the pilot some refined control of the vehicle. This is then the work of the pilot monitoring the vehicle to step in and safely abort the mission, landing the vehicle.

## Testing Devices

During the course of the project it was necessary to design and build several testing apparatuses to verify the systems, both individually and integrated sections, before full integration. Three testing apparatus were designed, one to test sensors, one to test motors, and the last to test the two integrated.
Simulink Model
The highest level block diagram for the dynamic model is shown in the Figure below. From the right the PWM values are fed in to the system, which move into the MATLAB block “Lipo_Voltage” that describes the batteries. This MATLAB block calculates the voltage based on the PWM signals, and passes it to the

Figure 29: Simulink Vehicle Dynamics Model
“Brushless Motors” block. In the “Brushless Motors” block, the voltage, and load are passed through the motor transfer functions to finally result in a thrust value. The “multicopter” block then takes the output thrust values and calculates the forces, and moments produced. Using this information the rest of the block diagram can simulate the dynamics, and output the current state of the system which is then fed back into the position and attitude controllers.

**Engine Thrust Test Stand**

The engine thrust test stand pictured below was used to validate the thrust values of the motors with the propellers chosen. A small pressure sensor is located near the rear of the apparatus. When the propeller spins up it pulls the sliding portion forward causing a bolt to apply pressure to the sensor. That sensor supplies a voltage based on the pressure that the DAQ measures. The DAQ then sends that information to a computer running LabVIEW, where a VI was built in order to turn those voltage signals into thrust values. This test stand was constructed for previous engineering design projects, AIAA student group is currently working on adding additional sensing equipment and controllers to provide remote operation and testing all within LabVIEW. Right now the thrust data is in LabVIEW but the RPM, voltage, and current are recorded on other devices.

![Motor Test Stand](image)

**Figure 30: Motor Test Stand**

**Flight Chamber**

The Flight Chamber was built to test sections of the system and the entire vehicle in an enclosed environment. This was a measure taken to provide safety for the engineers and the vehicle from unexpected errors in the control system, system integration, or guidance. Figure 33 shows a picture of the flight chamber.

The requirements for the flight chamber were that the vehicle needed to be separated from the observers, the chamber needed to be lightweight for carrying, easily assembled and disassembled, low cost and large enough to house the vehicle. A large cube was manufactured with PVC piping, plastic outdoor mesh
netting, pillows, and plywood. The PVC provides structure for the netting which separates the vehicle from the observers, the pillows are a soft landing area employed to mitigate damage should things go terribly wrong, and the plywood gives the PVC a stable base to which the frame can be attached to keep the structure in place. The limiting factors in chamber size are material properties of the PVC and available space in the lab. With the selected materials the construction of a chamber whose area is 64 sq ft, and height is 6 ft was completed.

**Rail System**
The Rail System was built in order to validate the sensors and navigation algorithm. It is a linear rail system that uses a stepper motor, two gears, a belt, and a small cart. Figure 31 shows the constructed testing rail system.

![Image of the rail system](image)

**Figure 31: Picture of the rail system; the motor and pulley are not attached to the rail**

The inspiration for this particular design came from linear motion tables used at Aerospace Laboratory for Plasma Experiments. These tables can be used to translate objects along one axis at varying speeds and are easily controlled via LabVIEW. The simple, easy to implement, and affordable design was a natural choice for this testing.

In order to drive the stepper motor code was written and embedded onto an Arduino board. This microcontroller regulated the voltages sent by the power supply to the motor effectively controlling the motor through a shield. The system was then calibrated by measuring the angle that ten steps resulted in from a datum; using that information we determined how many steps needed to be taken to get a full revolution of the motor shaft. From there the number of steps could be converted into a number of revolutions which could then be converted into a change in translation of the cart having measured the
radius of the gear. The code was then modified to allow the user to input a distance and a rate at which the distance was traveled.

Different sensor combinations can be attached to the cart to evaluate the sensor bias parameters and the overall effectiveness of the navigation algorithm. The main goal would be to test the Visual Odometry with and without the rest of the IMU sensor to further tune the Kalman filters and to correct any other unknown sensor errors that may arise.

While being well within our budget to construct, there are several drawbacks to this particular design that a larger budget could have easily remedied. This system only allows for one dimensional translational motion, allowing us to test our sensors in one of the six degrees of freedom at a time. The inability to test any of the sensors in attitude determination is a blatant flaw, as attitude is more important to the dynamics of the vehicle.

**Manufacturing**

**Flight Vehicle**

The manufacturing of the frame with the carbon fiber forced the team to slow down the process. This part of the frame manufacturing required aligning holes through the different spars in order to make a rigid frame structure. Although the frame works and the holes are aligned well enough for the purposes of this project, much time could have been saved by using a different method for drilling the holes including using a machine. In the future it may be worth having the company that manufactures the carbon fiber rods perform this machining for an additional cost instead of using the equipment at Western. Figure 32 shows the constructed frame.

![Figure 32: The Constructed Vehicle frame from rear facing view](image)

The motor mount manufacturing proved to be a challenge due to the simple hurried nature of the design. The pieces all had flat edges and needed to be glued together after being laser cut rather than having one solid piece. In addition, there were no slots for the cap section of the mount so they had to be fitted to the spars before being put in place. A second mistake was that the holes were drilled into the spars for the connecting bolt before finishing each mount causing additional issues (as the hole would be laser cut out).

During the first major damage to the vehicle in a flight test, the motor mount snapped off. The top plate with the motor sheared off the cap section protecting both the frame and the motor (the prop had small damage). However the design team realized that it would be a problem to fix this motor mount, deciding to redesign instead. The first solution is to use the same design but design slots so that the mount can be
assembled and glued like a kit. Extra weight could also be trimmed off but is unclear if the design time is worth the savings. The second solution is to use a stiffer material that would allow us to build a single piece and mold it into the shape like plastic or a metal. The associated problems are cost and shear capabilities.

The second iteration of motor mounts (after a few of the first ones broke) proved to be simpler. The individual wood blocks now had laser cut grooves so they could be assembled into the proper shape before gluing, however the design failed to correct the center hole issue for the motors and the double bolt holes to prevent rotation on the arms.

**Flight Chamber**

Constructing the flight chamber required cutting the PVC pipes into the correct dimensions. The next step was to assemble the base plate and attach the PVC mounts. Once the two sides of the base plate were assembled, the cage structure was constructed. The locations for the PVC mounts were measured and then tested with net support structure before assembled. The PVC ends were then embedded into a wooden block and epoxied. The wood blocks were lastly screwed to the base plate.

![Flight Chamber inside one of the labs](image)

**Figure 33: Flight Chamber inside one of the labs**

Assembly of the structure took a minimum of two people. The netting was wrapped around the external of the surface and cut into two major piece sets. The first set of pieces covered the roof of the chamber and the rest was to wrap around the edges. The net was kept in place by tape and zip ties. Disassembling the structure would take several minutes to detach the netting and then only a few more seconds of dispersing the frame.
**Rail System**

The rail system manufacturing consisted of creating the stepper motor mount, a pulley mount and then blocks to raise the rail off the ground. A wooden board was cut into a 10” by 10” section to be attached to the top of the slider. Several 2x4 wood blocks were created to raise the rail off the ground. Another wood block was created to mount the motor to one end of the path. The pulley cable was closed by epoxying the two ends together and the cable was attached to the sliding cart by a wood piece bolted to it.

![Motor mount block for the rail system](image)

**Figure 34: Motor mount block for the rail system**

**Testing**

This section discusses the tests that were planned and conducted for the project along with some of the methodology for those tests.

**Testing Form & Plan**

In appendix A is a form used for testing and experiments so that the information can be recorded. This is used to record data and make a collection of all the flights that have been performed. Appendix B is a list of forms that have been used.

The testing plan for this project is complicated and overlaps with a lot of the development. The best way to develop software is create modules and prototypes that are iteratively tested while they are written. Once these modules are completed they can be assembled and tested in collaborations. Another area of testing is the simulation environment which is used to evaluate the controls and dynamics while being able to add in additional capabilities to eventually simulate the entire vehicle’s performance in a virtual world. Outside of the virtual world, testing occurs on simulated systems to evaluate real world scenarios such as the rail
system. Finally there are flight tests to evaluate the different flight systems as well as the final product. Below are tables showing the progression of testing for each of the different systems.

Table 18: A list of tests that the project needs to perform

<table>
<thead>
<tr>
<th>System Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Flight Tests</td>
</tr>
<tr>
<td>o Motor Test on vehicle frame</td>
</tr>
<tr>
<td>o PixHawk flight checkout</td>
</tr>
<tr>
<td>o Hover Test with Config 1</td>
</tr>
<tr>
<td>o Hover Test with Config 1 with maximum simulated weight</td>
</tr>
<tr>
<td>o Hover test with Config 2</td>
</tr>
<tr>
<td>o Waypoint Position Controller Test w/out Kinect</td>
</tr>
<tr>
<td>o Waypoint Position Controller Test w/ Kinect</td>
</tr>
<tr>
<td>o Multiple Waypoint Test w/out Kinect</td>
</tr>
<tr>
<td>o Multiple Waypoint Test w/ Kinect</td>
</tr>
<tr>
<td>o Simple Object Detection</td>
</tr>
<tr>
<td>o Simple Object Avoidance</td>
</tr>
<tr>
<td>o Search Mission Type 1 with No Obstacles</td>
</tr>
<tr>
<td>o Search Mission Type 2 with No Obstacles</td>
</tr>
<tr>
<td>o Complex Object Avoidance</td>
</tr>
<tr>
<td>o Search Mission Type 1 with Obstacles</td>
</tr>
<tr>
<td>o Search Mission Type 2 with Obstacles</td>
</tr>
<tr>
<td>o Mission Evaluation with varying levels of environment</td>
</tr>
<tr>
<td>• Simulations</td>
</tr>
<tr>
<td>o Attitude Control System</td>
</tr>
<tr>
<td>o Position Control System without Advanced Navigation</td>
</tr>
<tr>
<td>o Position Control System with Advanced Navigation</td>
</tr>
<tr>
<td>o Waypoint Flight Tests</td>
</tr>
<tr>
<td>o Obstacle Detection and Avoidance</td>
</tr>
<tr>
<td>o SLAM Demo</td>
</tr>
<tr>
<td>• Hardware</td>
</tr>
<tr>
<td>o Hardware in the loop playback test</td>
</tr>
<tr>
<td>o Hardware in the loop simulation</td>
</tr>
</tbody>
</table>

One problem with simulations and hardware in the loop tests are the complexity to interface the different components necessary to do this in the full scale simulation test. One method is to do video playback but that just evaluates the sensor’s ability to fly on a predetermined route. A better way is to build an entire environment and fake Kinect sensor model however that requires a lot of time and effort to do in order to properly replicate the sensor noise, etc. This would also require building a 3D modeled environment so that the sensor could perform in a similar way. The decision was to make a simplified 2D laser scan like sensor to perform the operation of a SLAM sensor to estimate the position.

Flight Tests

Motor Checkout on Vehicle Frame
Getting the Pixhawk system to work with the large multirotor proved to be much more challenging task than anticipated. One major issue encountered was calibrating the accelerometer and radio transmitter. A small stand was produced to assist in the accelerometer calibration which requires the vehicle to be placed on each face (left, right, front, back, top, and bottom). The radio transmitter and receiver had to be configured, reset, and then reconfigured in order to have the system operational. Thankfully, many helpful
tips can be found online on DIYDrones and the APM website. The biggest issue was that the calibration of the remote control did not respond well. The APM software allows for testing of each motor and can set the throttle level for that test and duration. This software showed that the motors were working and there was an issue with what our RC equipment was doing. The vehicle would be powered on but the throttle levels would be stuck at above idle speed but would not increase further.

Once the Pixhawk and radio was recalibrated the system worked fine and a motor spin up test was enabled through the APM mission planner software. Inside APM, there is a function to specifically do a motor test by setting a throttle percentage and time duration. This was tested on every motor to validate they were producing significant thrust and spinning in the correct direction. The wires on the motors are three pronged and the way the ESC sends the signal is that it flips the sinusoid rapidly to push the magnet around in the motor.

**Pixhawk Flight Checkout**

Flight Form 1 in Appendix B

Initial flight test inside the cage system was unsuccessful. The amount of thrust produced by the vehicle for takeoff was sufficient to blow the pillows around on the ground. In addition the vehicle had very little room to maneuver for the first flight (which the vehicle has not been trimmed and there is no experience for flying characteristics) which ended with the vehicle becoming entangling in the net. The good news is that the net worked as designed and caught the vehicle’s rotors.

The next sets of flights took place outside of the cage and instead were located in one of the lobbies of the engineering building when the building was relatively empty. This open area allowed the aircraft to leave the ground and immediately the aircraft was observed to have a spinning motion. This could be shown without the aircraft even leaving the ground as it slid around the ground on foam padding. Using the controller, the pilot was unable to counter the spinning motion but could decrease the rotational rate. After watching this occur in several attempts, the situation was analyzed. The theory was that because in the preconfigured software the center arms are at a longer reach than those on the edges, the vehicle’s operation when all arms were the same length resulted in a spinning motion.

Two flights were then attempted outside. The first one climbed to an altitude of 2 meters and began to spin again while drifting across the field. The vehicle thrust was lowered but the vehicle tipped and landed with more weight on one of the motor mounts separating it from the vehicle (one of the center motors). The other center motor was removed to balance the vehicle and transformed into a quad rotor. Another flight was attempted but no spinning motion occurred although there was significant drift caused by the wind (the first outside flight drift is not directly correlated to wind). This led the team to believe the first theory was correct otherwise the products of inertia would have caused additional spinning this case, although it does not rule out the possibility one motor was spinning backwards.

**Conclusion of Test**

The preconfigured flight controller will not work with the designed multirotor configuration. However a different “preconfigured” design may work better. There was an H design which is similar but it was for a quad rotor. The team decided to move forward with developing their own controller and implementing it into the APM for the next set of tests. The main goal of the test was accomplished proving that the vehicle could fly, that the Pixhawk was working, and that the propulsion system was working. It also provided necessary setup experience for future missions and streamlined the process.
**Testing the Kinect VO prototype**

A Kinect VO prototype was created to test different algorithms for feature detection and feature matching. The first methodology was using Wang-Brady corner detector, however the image rate was 2 frames per second with full resolution. Decreasing the resolution to what is suggested for optimal Kinect sensing resulted in 5 frames per second. This code can be found in Appendix D and the implementation of the algorithm from the paper [AK]. The second test was utilizing OpenCV with FAST detector that resulted in a frame rate greater than 10 per second, which was the requirement. This demo also proved the feature descriptors were capable of matching at high frame rates. Two possibilities exist on why the first algorithm ran so slowly, the first being the implementation of Kinect App vs a Kinect console file using windows to render versus OpenCV. The second is that the algorithm mathematical implementation could be simplified by using additional software techniques. In Figure 35, two example data sets are shown from the Kinect. In Figure 36 and 37 the feature detectors and the descriptor matching is applied. The lines are connecting features from the keyframe image to the current image.

**Figure 35: Images from the Kinect; Left is depth and the Right is Depth mapped to Color**

**Figure 36: Feature Detection with Kinect using FAST**
Testing the Guidance Algorithm

A prototype and implementation of the AD star algorithm was created in Java for rapid prototyping rather than dealing with C's memory allocation and the team's greater experience in Java than C. The algorithm was difficult to implement from the original paper and required additional readings from the following [AN, Y], mainly the proof paper that supported AD star. Tests were generated demonstrating parts of the algorithm including reading from binary files and converting this data into nodes for the algorithm. A simple case which is often used to demonstrate the path finding can be seen in Figure 1. A L shape wall blocks the vehicle from a direct path and thus it must find a path around it. To test this further, if a hole was placed in the wall at any point the path should take that route instead as it is shorter. The following figures show some of the test results. Figure 4 runs into an issue where the path it just went to was closed and it fails to traverse to the other direction and corners itself. The code can be found in Appendix C.

Several different solutions were attempted however it did not appear to be a fault with the AD star implementation but the data wrapper around it. Two parts that are believed to be at fault are the heuristic and cost functions associated with pathing and the node neighbors which removes “walls” as neighbors. This makes the cost and heuristic function no longer consistent violating the mathematical principles behind the algorithm. A proposed solution would be to add the neighbors of walls back in but update the functions to include a cost when the path runs into a wall.
Results

This is broken up into two sections; evaluation and lessons. Evaluation of the system and testing equipment that was prototyped and tested is followed by lessons learned which are important for future participants to not make the same mistakes that happened.

Evaluation of the System

One of the major issues was the weight of all the components. This was unknown territory for most of the members and understanding the processing requirements at the start were best guesses from the rudimentary knowledge. Major goals for a redesign would be to decrease the weight of the electronic systems assuming the sensors remain the same. The computation system was chosen because of the Windows requirements although that may not be as necessary anymore with the Linux version for the Kinect sensor becoming more developed. This would allow a lot more types of computation boards available because of running embedded Linux. That could save weight right away, although the current design does allow more flexibility in future development with a single piece of hardware. The next would be decreasing the amount of carbon fiber used for both thickness and size of the frame. The carbon fiber thickness was chosen because at that time the structures person had quit and it was decided that there would not be enough funds for buying a second round of carbon fiber if the first broke. The carbon fiber that was received was of excellent quality and it was over designed for the requirements. It would be expected that if FEA was performed that a more optimal structural design could be obtained. The size of the carbon fiber is larger than necessary; the biggest worry was that motors would strike the hardware on the platform. However there was room to shorten the booms. Another major issue is the moment of inertia for the vehicle. The flat design was thought to be effective because of the simplicity in the structure rather than building a large stack. However this caused products of inertia due even though the design attempted to be as symmetrical as possible. Further investigation into this should be done. Of course as the vehicle size and weight changes, a different set of motors should be considered for the propulsion system.
Evaluation of the Testing Equipment
The biggest issue with the flight chamber was the size and padding. The vehicle was so large that even though the area was 64 square feet the vehicle at first was not evenly balanced and would quickly catch itself in the net. The next biggest issue was that because the testing chamber was not fixed in one position and needed to be disassembled, the pillows could not be permanently mounted to the bottom. The vehicle generated so much thrust that the pillows blew away and some even got off the ground when the vehicle spun up to 40% power and had yet to achieve flight. Making the chamber even larger is possible but at that point it would be cheaper to block off an area in a lab and lay down foam mats. However on testing the net did perform properly by catching the vehicle’s blade and stopping it however the vehicle was only a few inches off the ground and at slightly less than hover. The problem then became of untangling the rotor blades from the net.

The rail was tested by itself but not with the sensors and thus no evaluation can be made.

Lessons Learned
Below are a few lessons learned during the project development that would be helpful for future members of the project or for other groups working on the vehicle.

Group Projects and volunteer workers
A problem faced by this project was keeping students interested and committed to the project. This is often faced outside of engineering when workers are volunteering their time to a project with no monetary compensation.

Project Scale and Scope
One difficulty with project is scale and scope and determining the correct balance. It seems to be much easier to get funding for a project that has the full scope from start to finish, however this is an issue because it forces a lot of commitment to a project. If things start to go downhill, it becomes more difficult to decrease the scope of the project without violating the original project goals and the promise involved. One possible solution is to provide the full scope in the proposal but only talk about a specific component of the entire scope, slowly scaling the project up as it goes. One of the project failures was that the original size team was probably the right size for the project scope that was defined for a 1.5 year time period, however as soon as the team size changed this goal was no longer feasible. The only possible solutions are to attempt to recruit more individuals (which is difficult at a late stage) and to decrease the scale of the project, focusing on a few key parameters as a result with the full system. Of course, one alternative is just to extend the project end result for another group to continue the work.

Manufacturing Lessons and Prototyping
Future changes to the design warrant making the vehicle smaller however CFD testing would be best to validate this proposal. Shortening the spars for the motors could allow the vehicle to save additional weight but also place the rotors closer to the main plate structure (if not over it) that may cause propulsion issues. The rest of the size is locked because of the main electronics and batteries being placed on the vehicle. However, due to the lack of resources to redesign and build the vehicle structure will not be attempted in the near future.

It is also recommended that the carbon fiber rods that were chosen should have a smaller diameter and would not need as large of thickness. The original rod size was decided because it would be better to have an overly strong design rather than have them break and be unable to afford new ones. However, the company that the rods were purchased from have put up models and provided material information so that preliminary structural analysis could be run in the future.
Using precision tools to perform cutting, whether is contracting it out to an outside vendor or using tools at the university, is a better way of manufacturing than what was carried out. The project was needlessly rushed in the excitement to build the prototype where better equipment and more time would have resulted in a high quality finished vehicle frame.

**Software Development**

Working with others’ software comes with both a blessing and a detriment. The advantage is that it cuts down on the amount of prototyping, testing, and writing of code to do however it still requires a large amount of time to understand what is going on, especially when trying to modify. Other times this turns the software into a block box because of the size and organization of the source code. The MRPT library set has tens of thousands of lines of code which builds off of tens of thousands of lines of code from their dependent libraries such as OpenCL, OpenCV, and OpenGL. This can make it really challenging to do something that would be considered “simple.” The flip side is writing from scratch or a lower level. It sometimes appears that writing from a lower level provides better optimality, control, and understanding of the problem. However takes considerably more time as every bit of code needs to built and tested before moving forward. Finding some level of middle ground is the best, moving down to a lower level of implementing OpenCV functions is great because most of their functions are based off scientific papers. A scientist or engineer will create a novel algorithm for a specific problem and prove that it works in so many cases. In addition the author provides the sophisticated math that is the background for that algorithm. People in the community will then take the math and turn it into code (sometimes even using pseudo code and examples from the original author). This allows the writer to do research on the different algorithms through journal articles and then use the implementations in OpenCV to reduce the amount of work. This still requires large amounts of work but it is a safer middle ground.

**Conclusion**

Although the project has not been finished, the ground works of the project have been finished and are ready for implementation and testing. The team developed a mission, a set of criteria to evaluate the systems and derived the requirements to accomplish the mission. These requirements led to the creation of the system’s preliminary and detailed design and initial prototyping of algorithms. Evaluation of the current vehicle design shows that it will be able to satisfy its requirements and that all of the hardware components are able to interact with each other.

**Future Work**

The list of the future work is highlighted across the paper and the framework for most of the details has been outlined throughout this report. The main components are the software algorithms to perform SLAM an interface with the different systems onboard the vehicle followed by testing to evaluate these algorithms and systems. The most important part is finding individuals that are dedicated and willing to push themselves beyond the normal limits to continue the project work rather than letting it slip. The project will still continue on by the original authors even though they may no longer be at Western Michigan University, the goal is still the same to develop an autonomous proximity operations demonstration system.

For those interested in continuing the project work, the best way to proceed is by following the schedule chart and testing plan. This includes continued development on the modelling and simulation software, guidance and navigation algorithm testing, and system architecture of the vehicle. Once the vehicle systems have been finished and configured, the final test flights and demos can proceed. A few of the original authors will continue to work on this project and publish it at their website (watzasolutions.com).
## Appendices

### Appendix A: Testing Form

<table>
<thead>
<tr>
<th>Pilot</th>
<th>First</th>
<th>Last</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team Members</td>
<td>First</td>
<td>Last</td>
<td>First</td>
</tr>
<tr>
<td>Test Objectives</td>
<td>First</td>
<td>Last</td>
<td>First</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equippage</th>
<th>Type</th>
<th>Location</th>
<th>Weather (if App.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kinect</td>
<td>Motion Sim</td>
<td>Chamber</td>
<td>Temp</td>
</tr>
<tr>
<td>Computer</td>
<td>Simulink</td>
<td>Inside</td>
<td>Wind Dir</td>
</tr>
<tr>
<td>Telemetry</td>
<td>Flight</td>
<td>Outside</td>
<td>Wind Spd</td>
</tr>
<tr>
<td>Pixhawk</td>
<td>Mission</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robotic Arm</td>
<td>Demo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Config (4</td>
<td>6)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batteries</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
</tr>
<tr>
<td>B2</td>
</tr>
<tr>
<td>B3</td>
</tr>
<tr>
<td>B4</td>
</tr>
<tr>
<td>B5</td>
</tr>
<tr>
<td>B6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description Of Test Event</th>
</tr>
</thead>
</table>
Appendix B: Completed Testing Forms

Flight Form 1:

<table>
<thead>
<tr>
<th>Pilot</th>
<th>Team Members</th>
<th>Date</th>
<th>May 10th 2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spencer</td>
<td>James Jenkins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Watson</td>
<td>Christopher Proctor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Test Objectives: To checkout the Pixhawk

<table>
<thead>
<tr>
<th>Equippage</th>
<th>Type</th>
<th>Location</th>
<th>Weather (if App.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kinect</td>
<td>Motion Sim</td>
<td>Chamber</td>
<td></td>
</tr>
<tr>
<td>Computer</td>
<td>Simulink</td>
<td>Inside</td>
<td>Wind Dir S</td>
</tr>
<tr>
<td>Telemetry</td>
<td>Flight</td>
<td>Outside</td>
<td>Wind Spd 5-10mph</td>
</tr>
<tr>
<td>X Pixhawk</td>
<td>Mission</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robotic Arm</td>
<td>Demo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Config (4</td>
<td>6) (4 &amp; 6)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Batteries

<table>
<thead>
<tr>
<th>B1 X B7</th>
<th>B2 X B8</th>
<th>B3 X B9</th>
<th>B4 X B10</th>
</tr>
</thead>
<tbody>
<tr>
<td>B5 X B11</td>
<td>B6 X B12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description: The aircraft was tested first indoors in the flight chamber when it was realized this was not large enough for initial flight characteristic handling. Moving into a deserted lobby the vehicle was tested again. Showed issues with yaw instability. Lastly two flights outdoors happened. Yaw instabilities still present and a motor mount broke. Tested quad configuration no yaw instabilities but vehicle susceptible to wind drift.
Appendix C: Guidance Algorithm Demo Code

Main.java

```java
package test;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import javax.xml.soap.Node;

public class Main {
    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub
        String FilePath = "";
        int c;

        //Step 1: Calculate Section

        //Step 2: Grab all nodes iNodeom the files
        ArrayList<PointNode> list = grabNodes(FilePath);
        for(int i = 0; i < list.size(); i++) {
            //System.out.println(list.get(i).print());
        }
        //System.out.println("Finished Test");

        //Step 3: Calculate Buffer zones for spots versus blocked / filled spots??

        //Step 4: Calculate all adjacencies for each node

        //Step 5: Pass the node list to the pathing algorithm along with start/goal
        PointNode start = list.get(0);
        PointNode end = list.get(list.size()-1);
        PathPlanner p1 = new PathPlanner(start, end, list.toArray(new PointNode[list.size()]), 2.5);
    }

    public static ArrayList grabNodes(String path) throws IOException {
        //Set Variables
        String filename = path + "file.bin";
        int width = 7;
        int in; int counter = 0;
        int x = 0; int y = 0;
        ArrayList<PointNode> List = new ArrayList<PointNode>();

        //Open File
        File file = new File(filename);
        FileReader fr = new FileReader(file);
        BufferedReader br = new BufferedReader(fr);

        //Loop through
        while((in = br.read()) != -1) {
            //System.out.println(in);
            List.add(new PointNode(x, y, Character.getNumericValue(in)));
            x++;
            if(x == width - 1) {
                y++;
                x = 0;
            }
        }
        return List;
    }
}
```
PathPlanner.java

package test;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;
import javax.xml.soap.Node;

public class PathPlanner {

    //Field
    private PriorityQueue<PointNode> OPEN = new PriorityQueue<PointNode>(new MyKeyComparator());
    private List<PointNode> CLOSED = new ArrayList<PointNode>();
    private List<PointNode> INCONS = new ArrayList<PointNode>();
    private List<PointNode> PATH = new ArrayList<PointNode>();
    private PointNode[] NodeArray;
    private PointNode START;
    private PointNode GOAL;
    private double INF = 999999;
    private double EPS;
    private double EPS_NOT;
    private boolean CHANGES;
    private boolean SIGNIFICANT;

    //Main Methods
    public PathPlanner(PointNode start, PointNode end, PointNode[] nodes, double f) throws IOException{
        //initialize
        NodeArray = nodes;
        EPS_NOT = f;
        //Calculate nodes
        calculateAdjacencies();
        START = NodeArray[0];
        GOAL = NodeArray[NodeArray.length - 1];
        //Run pathing
        FindPaths();
    }

    private void FindPaths() throws IOException{
        //initialize
        START.setG(INF);
        START.setRHS(INF);
        GOAL.setG(INF);
        GOAL.setRHS(0);
        EPS = EPS_NOT;
        boolean invalidPath;
        //Insert goal into Open
        }
    }

    private void FindPaths() throws IOException{
        //initialize
        START.setG(INF);
        START.setRHS(INF);
        GOAL.setG(INF);
        GOAL.setRHS(0);
        EPS = EPS_NOT;
        boolean invalidPath;
        //Insert goal into Open
        }
GOAL.setkeys(createkey(GOAL));
OPEN.add(GOAL);

// Compute the path
PrintAllNodes();
computeOrImprovePath();
Publish();

while(START != GOAL){
    // move the the vehicle one spot
    moveVehicle();
    // perform sensor scan (all spots around it?)
    if(sensorScan()){  
        invalidPath = false;
        // check to see if the path is invalid
        for(int i = 0; i < PATH.size(); i++){
            if(PATH.get(i).VALUE() != 0){
                // path is invalid
                invalidPath = true;
            }
        }
        // if so recalculate path
        if(invalidPath){
            computeOrImprovePath();
            Publish();
        }
    } else if(EPS != 1){
        EPS--;
        if(EPS < 1){
            EPS = 1;
        }
        computeOrImprovePath();
    } else{
        // Don't do anything
    }
}

private void moveVehicle(){
    // Move the vehicle to the first node in the list -> set the START to be new current position
    START = PATH.remove(0);
    System.out.println("Vehicle is now at "+START.X()+"","+START.Y());
}

private boolean sensorScan() throws IOException{
    boolean changed = false;
    // set the current position
    int x = START.X();
    int y = START.Y();
    int v, a;
    int width = 7;
    int maxX = 6; // -1 of actual width
    int maxY = 6; // -1 of actual height
    // grab those nodes & update current Node List Info
    String filename = "file.bin";
    // Open File
    RandomAccessFile br = new RandomAccessFile(filename,"r");
    if(x >= 0 & & (y+1) >= 0 & & x <= maxX & & (y+1) <= maxY){
        a = (y+1)*width + x;
        //System.out.println(NodeArray[a].X() + "," + NodeArray[a].Y() + "," + a);
        if(NodeArray[a].VALUE() != (v = grabNode(a, br))){
            changed = true;
        }
    }
    return changed;
}
changed = true;
NodeArray[a].setValue(v);
}
}
if(x >= 0 && (y-1) >= 0 && x <= maxX && (y-1) <= maxY){
a = (y-1)*width + x;
//System.out.println(NodeArray[a].X() + "," + NodeArray[a].Y() + "," + a);
if(NodeArray[a].VALUE() != (v = grabNode(a, br))){
changed = true;
NodeArray[a].setValue(v);
}
}
if(x+1 >= 0 && (y) >= 0 && x+1 <= maxX && (y) <= maxY){
a = y*width + x+1;
//System.out.println(NodeArray[a].X() + "," + NodeArray[a].Y() + "," + a);
if(NodeArray[a].VALUE() != (v = grabNode(a, br))){
changed = true;
NodeArray[a].setValue(v);
}
}
if(x-1 >= 0 && (y) >= 0 && x-1 <= maxX && (y) <= maxY){
a = y*width + x-1;
//System.out.println(NodeArray[a].X() + "," + NodeArray[a].Y() + "," + a);
if(NodeArray[a].VALUE() != (v = grabNode(a, br))){
changed = true;
NodeArray[a].setValue(v);
}
}
if(x-1 >= 0 && (y+1) >= 0 && x-1 <= maxX && (y+1) <= maxY){
a = (y+1)*width + x-1;
//System.out.println(NodeArray[a].X() + "," + NodeArray[a].Y() + "," + a);
if(NodeArray[a].VALUE() != (v = grabNode(a, br))){
changed = true;
NodeArray[a].setValue(v);
}
}
if((x-1) >= 0 && (y-1) >= 0 && (x-1) <= maxX && (y-1) <= maxY){
a = (y-1)*width + x-1;
//System.out.println(NodeArray[a].X() + "," + NodeArray[a].Y() + "," + a);
if(NodeArray[a].VALUE() != (v = grabNode(a, br))){
changed = true;
NodeArray[a].setValue(v);
}
}
//if any where changed;
return changed;
}
private int grabNode(int pos, RandomAccessFile br) throws IOException{
br.seek(pos);
int ret = Character.getNumericValue(br.readByte());
//System.out.println("Returned value:" + ret);
return ret;
}

//====================================
// Side Methods
//====================================

private void computeOrImprovePath(){
    PointNode minNode;
    while(!OPEN.isEmpty() && OPEN.peek().compareKeys(START) >= 0 || START.RHS() != START.G()){
        //remove the node with the minimum key
        minNode = OPEN.remove();
        //Compare the Cost with the Look Ahead
        if(minNode.G() > minNode.RHS()){  
            minNode.setG(minNode.RHS());
            //Set it as visited
            minNode.setVISITED(true);
            //add it to closed
            if(!CLOSED.contains(minNode)){
                CLOSED.add(minNode);
            }
        } else{
            minNode.setG(INF);
            updateSetMembership(minNode); //was missing this line
        }
        for(int i = 0; i<minNode.AdjacentNodeList().size();i++){  
        PointNode successor = minNode.AdjacentNodeList().get(i);
        if(successor.VISITED()==false){
            successor.setRHS(INF);
            successor.setG(INF);
            successor.setPreviousNode(null);
        }
        if(successor.RHS() > minNode.G() + minNode.calculateCost(successor)){
            //set previous node
            successor.setPreviousNode(MinArgs(successor));
            successor.setRHS(successor.getPrevious().G() + successor.getPrevious().calculateCost(successor));
            updateSetMembership(successor);
        }
    }
}
System.out.println(minNode.printAll());
//PrintAllNodes();
}

private PointNode MinArgs(PointNode s){
    PointNode ret = null;
    double mincost = INF+INF;
    return ret;
}
double newcost;
for(int i = 0; i < s.AdjacentNodeList().size(); i++) {
    PointNode predecessor = s.AdjacentNodeList().get(i);
    newcost = predecessor.G() + predecessor.calculateCost(s); // this needs to be investigated
    if (newcost <= mincost) {
        mincost = newcost;
        ret = predecessor;
    }
}
System.out.println("Min Args Prnt: " + s.printAll() + ". This needs to be investigated.");
System.out.println(ret.printAll());
return ret;
}

private void updateSetMembership(PointNode s) {
    if (s.RHS() != s.G()) {
        if (!CLOSED.contains(s)) {
            // insert/update s
            // create new keys
            s.setVISITED(true);
            s.setKEYS(createkey(s));
            // put into open
            OPEN.add(s);
        } else if (!INCONS.contains(s)) {
            INCONS.add(s);
        }
    } else {
        if (OPEN.contains(s)) {
            OPEN.remove(s);
        } else if (INCONS.contains(s)) {
            INCONS.remove(s);
        }
    }
}

private double[] createkey(PointNode n) {
    double[] k = new double[2];
    if (n.G() > n.RHS()) {
        k[0] = n.RHS() + EPS * START.calculateHeuristic(n);
        k[1] = n.RHS();
    } else {
        k[0] = n.G() + START.calculateHeuristic(n);
        k[1] = n.G();
    }
    // System.out.println("Keys are: " + k[0] + " & " + k[1]);
    return k;
}

// Extra Methods
// Extra Methods
private void calculateAdjacencies() throws IOException {
    FileWriter fr = new FileWriter("NodeOutput.txt");
    BufferedWriter bw = new BufferedWriter(fr);
    ArrayList<PointNode> list;
    int c;
    for (int i = 0; i < NodeArray.length; i++) {
        list = new ArrayList<PointNode>();
        c = 0;
        for (int j = 0; j < NodeArray.length; j++) {
            if (i != j && NodeArray[i].VALUE() == 0 && NodeArray[j].VALUE() == 0) {
                if (NodeArray[i].isAdjacent(NodeArray[j])) {
                    list.add(NodeArray[j]);
                    c++;
                }
            }
        }
        bw.write("Node Output: ");
        bw.close();
    }
}
```java
list.add(NodeArray[j]);
c++;
}
}
//Send it the list of the adjacent nodes
NodeArray[i].setAdjacentNodes(list);
bw.write(NodeArray[i].printAll());
bw.write(NodeArray[i].printAdjacencies());
bw.newLine();
}
bw.close();
}

private void Publish(){
    //Print the path starting from start and going backwards
    int i = 0;
    //PrintAllNodes();
    System.out.println("Started Publishing Path");
    PointNode n = START;
    System.out.println(i + "th node => " +n.print());
    while(n!=null){
        i++;
        PATH.add(n);
        System.out.println(i + "th node => " + n.print());
        n = n.getPrevious();
    }
    System.out.println("Finished Publishing");
}

private void PrintAllNodes(){
    System.out.println("====== Print all Nodes ===========");
    System.out.println("X,Y,Value,Key1,Key2,rhs,g,prevNode,Visited");
    for(int i=0;i<NodeArray.length;i++){
        System.out.println(NodeArray[i].printAll());
    }
    System.out.println("==============================");
}

PointNode.java

package test;
import java.util.ArrayList;
public class PointNode {
    //Fields
    private double[] KEYS = new double[2];
    private double RHS;
    private double G;
    private int X;
    private int Y;
    private ArrayList<PointNode> adjacentNodes;
    private PointNode previousNode;
    private int VALUE;
    private boolean VISITED;

    public PointNode(int x, int y, int i){
        X = x;
        Y = y;
        VALUE = i;
        VISITED = false;
    }
```
previousNode = null;
G = 99999;
RHS = 99999;
}

//====================================
// Get Methods
//====================================
public double Key1(){
    return KEYS[0];
}
public double Key2(){
    return KEYS[1];
}
public ArrayList<PointNode> AdjacentNodeList(){
    return adjacentNodes;
}
public PointNode getPrevious(){
    return previousNode;
}
public double G(){
    return G;
}
public double RHS(){
    return RHS;
}
public int X(){
    return X;
}
public int Y(){
    return Y;
}
public int VALUE(){
    return VALUE;
}
public boolean VISITED()
    {
        return VISITED;
    }

    // Other Methods

    public double calculateCost(PointNode n2)
    {
        double ans = Math.sqrt(Math.pow(X-n2.X(),2) + Math.pow(Y-n2.Y(),2));
        if(Double.isNaN(ans))
            //Print out stuff
        return ans;
    }

    public double calculateHeuristic(PointNode n2)
    {
        double ans = Math.sqrt(Math.pow(X-n2.X(),2) + Math.pow(Y-n2.Y(),2));
        return ans;
    }

    public boolean updateNode(int value)
    {
        if(value != VALUE)
            VALUE = value;
        return true;
    }

    public String print()
    {
        return this + "," + X + "," + Y + "," + VALUE;
    }

    public String printAll()
    {
    }

    // Comparisons Methods

    public boolean isSuccessor(PointNode n)
    {
        if(n == previousNode)
            return false;
        else
            return true;
    }

    public boolean isAdjacent(PointNode n2)
    {
        if((X-n2.X()) > 1) || (Y-n2.Y() > 1))
            return false;
        else if((X-n2.X()) < -1 || (Y-n2.Y() < -1))
            return false;
        else
            return true;
    }

    public int compareKeys(PointNode n2)
    {
        PointNode n1 = this;
        if(n1.Key1() == n2.Key2())
            //Compare Key 2
            if(n1.Key2() < n2.Key2())
                return -1; // less than
            else if (n1.Key2() == n2.Key2())
                return 0; // equal
        else
            else{}
//Greater than
return 1;

else if (n1.Key1[] < n2.Key1[])
    return -1; //less than
else{
    return 1; //greater than
}

public String printAdjacencies() {
String Returner = "";
for(int i = 0; i < adjacentNodes.size();i++){
    Returner = Returner +  " Adjacent:" + adjacentNodes.get(i).print() + "\n";
}
return Returner;
}

MyKeyComparator.java

package test;

import java.util.Comparator;

public class MyKeyComparator implements Comparator<PointNode>{
    public int compare(PointNode n1, PointNode n2){
        return n1.compareKeys(n2);
    }
}
Appendix D: Kinect Demo Code

// KinectPoseDemo.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include <Kinect.h>
#include <opencv2/opencv.hpp>
#include "MyKinectSensor.h"
#include <Windows.h>
#include <cstdlib>

int _tmain(int argc, _TCHAR* argv[])
{
    cv::setUseOptimized(true);

    // Sensor
    IKinectSensor* pSensor;
    HRESULT hResult = S_OK;
    hResult = GetDefaultKinectSensor(&pSensor);
    if (FAILED(hResult)) {
        std::cerr << "Error : GetDefaultKinectSensor" << std::endl;
        return -1;
    }

    hResult = pSensor->Open();
    if (FAILED(hResult)) {
        std::cerr << "Error : IKinectSensor::Open()" << std::endl;
        return -1;
    }

    // Source
    IColorFrameSource* pColorSource;
    hResult = pSensor->get_ColorFrameSource(&pColorSource);
    if (FAILED(hResult)) {
        std::cerr << "Error : IKinectSensor::get_ColorFrameSource()" << std::endl;
        return -1;
    }

    IDepthFrameSource* pDepthSource;
    hResult = pSensor->get_DepthFrameSource(&pDepthSource);
    if (FAILED(hResult)) {
        std::cerr << "Error : IKinectSensor::get_DepthFrameSource()" << std::endl;
        return -1;
    }

    // Reader
    IColorFrameReader* pColorReader;
    hResult = pColorSource->OpenReader(&pColorReader);
    if (FAILED(hResult)) {
        std::cerr << "Error : IColorFrameSource::OpenReader()" << std::endl;
        return -1;
    }

    IDepthFrameReader* pDepthReader;
    hResult = pDepthSource->OpenReader(&pDepthReader);
    if (FAILED(hResult)) {
        std::cerr << "Error : IDepthFrameSource::OpenReader()" << std::endl;
        return -1;
    }

    // Description
    IFrameDescription* pColorDescription;
    hResult = pColorSource->get_FrameDescription(&pColorDescription);
    if (FAILED(hResult)) {
        std::cerr << "Error : IColorFrameSource::get_FrameDescription()" << std::endl;
    }
return -1;
}

int colorWidth = 0;
int colorHeight = 0;
pColorDescription->get_Width(&colorWidth); // 1920
pColorDescription->get_Height(&colorHeight); // 1080
unsigned int colorBufferSize = colorWidth * colorHeight * 4 * sizeof(unsigned char);
cv::Mat colorBufferMat(colorHeight, colorWidth, CV_8UC4);
cv::Mat colorMat(colorHeight / 2, colorWidth / 2, CV_8UC4);
//cv::namedWindow("Color");

lFrameDescription* pDepthDescription;
hResult = pDepthSource->get_FrameDescription(&pDepthDescription);
if (FAILED(hResult)) {
    std::cerr << "Error : IDepthFrameSource::get_FrameDescription()" " << std::endl;
    return -1;
}

int depthWidth = 0;
int depthHeight = 0;
pDepthDescription->get_Width(&depthWidth); // 512
pDepthDescription->get_Height(&depthHeight); // 424
unsigned int depthBufferSize = depthWidth * depthHeight * sizeof(unsigned short);
cv::Mat depthBufferMat(depthHeight, depthWidth, CV_16UC1);
cv::Mat depthMat(depthHeight, depthWidth, CV_8UC1);
//cv::namedWindow("Depth");

// Coordinate Mapper
ICoordinateMapper* pCoordinateMapper;
hResult = pSensor->get_CoordinateMapper(&pCoordinateMapper);
if (FAILED(hResult)) {
    std::cerr << "Error : IKinectSensor::get_CoordinateMapper()" " << std::endl;
    return -1;
}

cv::Mat coordinateMapperMat(depthHeight, depthWidth, CV_8UC4);
//cv::namedWindow("CoordinateMapper");

unsigned short minDepth, maxDepth;
pDepthSource->get_DepthMinReliableDistance(&minDepth);
pDepthSource->get_DepthMaxReliableDistance(&maxDepth);

//Create the detector and descriptors
cv::Mat currentGray; //current gray
cv::Mat keyGray; //previous gray
//cv::namedWindow("Features");

//instantiate feature keypoints
std::vector<cv::KeyPoint> keypoints, keypoints2; //instantiate
int threshold = 30; //instantiate

//instantiate the feature detector
cv::Ptr<cv::FeatureDetector> detector;
detector = new cv::FastFeatureDetector(threshold, true);

//instantiate the feature detector
cv::namedWindow("Matches");
cv::Mat outimg; //instantiate

std::vector<cv::DMatch> indices; //instantiate
// instantiate the descriptor extractor
cv::Ptr<cv::DescriptorExtractor> descriptorExtractor;
descriptorExtractor = new cv::BriefDescriptorExtractor(); // probably need to set this up for Brisk

// instantiate the descriptor matcher
std::vector<std::vector<cv::DMatch>> matches; // instantiate
cv::Ptr<cv::DescriptorMatcher> descriptorMatcher; // instantiate
descriptorMatcher->create("BruteForce"); // new cv::DescriptorMatcher();

bool twoImages = false;
int counter = 0;

while (1) {
    // Color Frame
    IColorFrame* pColorFrame = nullptr;
    hResult = pColorReader->AcquireLatestFrame(&pColorFrame);
    if (SUCCEEDED(hResult)) {
        hResult = pColorFrame->CopyConvertedFrameDataToArray(colorBufferSize,
        reinterpret_cast<BYTE*>(colorBufferMat.data), ColorImageFormat::ColorImageFormat_Bgra);
        if (SUCCEEDED(hResult)) {
            cv::resize(colorBufferMat, colorMat, cv::Size(), 0.25, 0.25);
        }
    }
    //SafeRelease( pColorFrame );

    // Depth Frame
    IDepthFrame* pDepthFrame = nullptr;
    hResult = pDepthReader->AcquireLatestFrame(&pDepthFrame);
    if (SUCCEEDED(hResult)) {
        hResult = pDepthFrame->AccessUnderlyingBuffer(&depthBufferSize,
        reinterpret_cast<UINT16*>(depthBufferMat.data));
        if (SUCCEEDED(hResult)) {
            depthBufferMat.convertTo(depthMat, CV_8U, -255.0f / 8000.0f, 255.0f);
        }
    }
    //SafeRelease( pDepthFrame );

    // Mapping (Depth to Color)
    if (SUCCEEDED(hResult)) {
        std::vector<ColorSpacePoint> colorSpacePoints(depthWidth * depthHeight);
        hResult = pCoordinateMapper->MapDepthFrameToColorSpace(depthWidth * depthHeight,
        reinterpret_cast<UINT16*>(depthBufferMat.data), depthWidth * depthHeight, &colorSpacePoints[0]);
        if (SUCCEEDED(hResult)) {
            coordinateMapperMat = cv::Scalar(0, 0, 0, 0);
            for (int y = 0; y < depthHeight; y++) {
                for (int x = 0; x < depthWidth; x++) {
                    unsigned int index = y * depthWidth + x;
                    ColorSpacePoint point = colorSpacePoints[index];
                    int colorX = static_cast<int>(std::floor(point.X + 0.5));
                    int colorY = static_cast<int>(std::floor(point.Y + 0.5));
                    unsigned short depth = depthBufferMat.at<unsigned short>(y, x);
                    if ((colorX >= 0) && (colorX < colorWidth) && (colorY >= 0) && (colorY < colorHeight) && (depth >= minDepth) && (depth <= maxDepth)) {
                        coordinateMapperMat.at<cv::Vec4b>(y, x) = colorBufferMat.at<cv::Vec4b>(colorY, colorX);
                    }
                }
            }
            //Do image processing Here
            //Do image processing Here

            if (counter == 0) {
//save the first image here
//convert to gray image
cv::cvtColor(colorMat, keyGray, CV_BGR2GRAY);
//perform the detection
detector->detect(keyGray, keypoints);
//create the descriptor
descriptorExtractor->compute(keyGray, keypoints, descriptors);
//drawKeypoints(keyGray, keypoints, keyImage, cv::Scalar(0, 255, 0), cv::DrawMatchesFlags::DEFAULT);
//cv::imshow("Features", keyImage);
counter++;
}
else {

cv::cvtColor(colorMat, currentGray, CV_BGR2GRAY);
detector->detect(currentGray, keypoints2);
descriptorExtractor->compute(currentGray, keypoints2, descriptors2);
counter++;
//drawKeypoints(currentGray, keypoints2, keyImage2, cv::Scalar(0, 255, 0), cv::DrawMatchesFlags::DEFAULT);

//try matches
//Brute force matching between current and previous: is this Mutual Exclusion?
descriptorMatcher = new cv::BFMatcher(cv::NORM_HAMMING, false);
descriptorMatcher->radiusMatch(descriptors, descriptors2, matches, 10.0);

//-- Localize the object
//std::vector<cv::Point2f> obj;
//std::vector<cv::Point2f> scene;
//RANSAC Implementation with SVD
//RANSAC with SVD has yet to be completed and tested!
/*
for (int i = 0; i < matches.size; i++)
    //keyGray.push_back(keypoints[i]);
    //currentGray.push_back(keypoints2[matches[i]]);
*/

//cv::Mat H = findHomography(keyGray, currentGray, CV_RANSAC);
//cv::Mat E = findFundamentalMat(keyGray, currentGray, CV_FM_RANSAC);

    cv::Mat w;
    cv::Mat u;
    cv::Mat vt;
    //cv::SVDecomp.compute()
    cv::SVD svd;
    svd.compute(E, w, u, vt);
    double m[3][3] = { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };
    w = cv::Mat(3, 3, CV_64F, m);
    cv::Mat Eclosest = u*w*vt;
    /*
    //Matching the images
drawMatches(keyGray, keypoints, currentGray, keypoints2, matches, outimg, 
    cv::Scalar(0, 255, 0), cv::Scalar(0, 0, 255), 
    std::vector<std::vector<char>> x(), cv::DrawMatchesFlags::DEFAULT);
    cv::imshow("Matches", outimg);
    */
}
if (counter == 25) {
    counter = 0;
}
SafeRelease(pColorFrame);
SafeRelease(pDepthFrame);
//cv::imshow("Keyframe Features", keyImage);
//cv::imshow("Color", colorMat);
//cv::imshow("Features Comparison", colorMat);
cv::imshow("Depth", depthMat);
//cv::imshow("CoordinateMapper", coordinateMapperMat);

if (cv::waitKey(30) == VK_ESCAPE) {
    break;
}
}

SafeRelease(pColorSource);
SafeRelease(pDepthSource);
SafeRelease(pColorReader);
SafeRelease(pDepthReader);
SafeRelease(pColorDescription);
SafeRelease(pDepthDescription);
SafeRelease(pCoordinateMapper);
if (pSensor) {
    pSensor->Close();
}
SafeRelease(pSensor);
cv::destroyAllWindows();
return 0;

int main()
{
    cv::setUseOptimized(true);
    int colorHeight, colorWidth, depthHeight, depthWidth;
    //Create the Kinect Class
    MyKinectSensor myKinect (&colorHeight,&colorWidth,&depthHeight,&depthWidth); //??

    //Initialize the Kinect Sensor
    myKinect.InitializeSensor();

    //Setup output windows?

    //========== setup variables for data =============
    //
    cv::Mat colorBufferMat(colorHeight, colorWidth, CV_8UC4);
    cv::Mat colorMat(colorHeight / 2, colorWidth / 2, CV_8UC4);
    cv::namedWindow("Color");
    
    cv::Mat depthBufferMat(depthHeight, depthWidth, CV_16UC1);
    cv::Mat depthMat(depthHeight, depthWidth, CV_8UC1);
    cv::namedWindow("Depth");
    
    cv::Mat coordinateMapperMat(depthHeight, depthWidth, CV_8UC4);
    cv::namedWindow("CoordinateMapper");

    //Excute main loop
    while (1) { //no hardware error or quit, keep running
        //pass in obsVariables into kinect update
        HRESULT hr;
        hr = myKinect.update(&colorBufferMat, &depthBufferMat, &coordinateMapperMat);
        if (SUCCEEDED(hr)) {
            //copy from buffers to storage points
            cv::resize(colorBufferMat, colorMat, cv::Size(), 0.5, 0.5);
            depthBufferMat.convertTo(depthMat, CV_8U, -255.0f / 8000.0f, 255.0f);
            //Possibly better way to do this is; is pass in the final result mat, and
            //process Kinect data
            //cv::imshow("Color", colorMat);
            cv::imshow("Depth", depthMat);
            //cv::imshow("CoordinateMapper", coordinateMapperMat);
//I HAVE TO FREE THE FRAMES HERE OR ELSE IT EXPLODES! I'll fix this later
if (cv::waitKey(30) == VK_ESCAPE) {
    break;
}
}
cv::destroyAllWindows();
return 0;
*/
Bibliography


