

LOCALIZATION AND CONTROL OF DISTRIBUTED MOBILE ROBOTS WITH THE  
MICROSOFT KINECT AND STARL

by

NATHAN HERVEY

Presented to the Faculty of the Graduate School of  
The University of Texas at Arlington in Partial Fulfillment  
of the Requirements  
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2016

Copyright © by Nathan Hervey 2016

All Rights Reserved



## **Acknowledgements**

I would first like to thank Dr. Taylor Johnson for giving me the opportunity to work on a fun and challenging project. His advice and direction have been invaluable throughout this process. I would also like to thank both Dr. Roger Walker and Dr. Farhad Kamangar for being on my committee and for providing enjoyable courses in which I learned many new things. Finally, I would like to thank my wife Katie, who has supported me and allowed me to stay in school for a very long time.

The material presented in this thesis is based upon work supported by the National Science Foundation (NSF) under grant number CNS 1464311, the Air Force Research Laboratory (AFRL) through contract number FA8750-15-1-0105, and the Air Force Office of Scientific Research (AFOSR) under contract number FA9550-15-1-0258. The U.S. government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of AFRL, AFOSR, or NSF.

April 22, 2016

## **Abstract**

### LOCALIZATION AND CONTROL OF DISTRIBUTED MOBILE ROBOTS WITH THE MICROSOFT KINECT AND STARL

Nathan Hervey, MS

The University of Texas at Arlington, 2016

Supervising Professor: Taylor Johnson

With the increasing availability of mobile robotic platforms, interest in swarm robotics has been growing rapidly. The coordinated effort of many robots has the potential to perform a myriad of useful and possibly dangerous tasks, including search and rescue missions, mapping of hostile environments, and military operations. However, more research is needed before these types of capabilities can be fully realized. In a laboratory setting, a localization system is typically required to track robots, but most available systems are expensive and require tedious calibration. Additionally, dynamical models of the robots are needed to develop suitable control methods, and software must be written to execute the desired tasks. In this thesis, a new video localization system is presented utilizing circle detection to track circular robots. This system is low cost, provides  $\sim 0.5$  centimeter accuracy, and requires minimal calibration. A dynamical model for planar motion of a quadrotor is derived, and a controller is developed using the model. This controller is integrated into StarL, a framework enabling development of distributed robotic applications, to allow a Parrot Cargo Minidrone to visit waypoints in the x-y plane. Finally, two StarL applications are presented; one to demonstrate the capabilities of the localization system, and another that solves a

modified distributed travelling salesman problem where sets of waypoints must be visited in order by multiple robots. The methods presented aim to assist those performing research in swarm robotics by providing a low cost easy to use platform for testing distributed applications with multiple robot types.

## Table of Contents

Acknowledgements .....	iii
Abstract .....	iv
List of Illustrations .....	viii
List of Tables .....	x
Chapter 1 Introduction.....	1
Chapter 2 Background and Related Work .....	4
Robot Localization .....	4
Dynamic Model of a Quadcopter .....	5
StarL .....	5
Chapter 3 Video Localization System .....	7
State Space Representation and Estimation.....	7
Experimental Set-up .....	8
Camera System Set-up .....	8
Robot Set-up .....	9
Position Estimation Using Circle Tracking.....	9
Yaw Estimation Using Color Detection .....	11
Altitude Estimation Using Kinect Depth Sensor .....	13
Localization Program Flow .....	13
Initial Location and Identification of Robots with <code>findBots</code> Function .....	13
Localization of Minidrones and ARDrones with Find Drones and <code>isARDrone</code> function.....	14
Localization of iRobotCreate2s with Find iRobots.....	19
Tracking Robots after Initial Localization.....	20
System Performance .....	22

Position Estimation .....	22
Yaw Estimation .....	24
Altitude Estimation .....	25
Average Sampling Rates .....	26
ARDrone Tracking Results .....	27
Tracking Larger Numbers of Robots .....	31
Chapter 4 Modeling and Control of Parrot Minidrone .....	33
Parrot SDK and StarL Integration .....	33
Equations for Quadrotor Planar Acceleration .....	34
Simulink Model of the Drone .....	37
Minidrone Control in StarL .....	39
PID Controller Class .....	39
Minidrone Motion Automaton .....	39
Minidrone Control Results .....	41
Chapter 5 StarL Implementation and Applications .....	43
StarL Implementation .....	43
Follow App .....	44
Modified Distributed Travelling Salesman App .....	46
Chapter 6 Conclusion and Future Work .....	53
References .....	56
Biographical Information .....	62

## List of Illustrations

Figure 3-1 Experimental set-up of video localization system .....	8
Figure 3-2 Set-up for (a) iRobot Create2 (b) Parrot Minidrone (c) Parrot ARDrone .....	9
Figure 3-3 Coordinate systems as viewed from above, (a) camera with z-axis positive towards the ground (b) real-world with z-axis positive towards the ceiling .....	11
Figure 3-4 (a) Program flow for tracking script (b) program flow for <code>findBots</code> function .....	14
Figure 3-5 (a) Program flow for finding minidrones and ARDrones (b) steps for adding a robot to <code>botArray</code> .....	16
Figure 3-6 Localized minidrone and iRobot Create2 with bounding boxes shown in yellow .....	17
Figure 3-7 Program flow for <code>isARDrone</code> function .....	18
Figure 3-8 Distance ranges between circle centers for an ARDrone on the ground .....	19
Figure 3-9 Program flow for find iRobots step of <code>findBots</code> .....	20
Figure 3-10 Program flow for <code>trackBots</code> function .....	21
Figure 3-11 Estimated (a) x and (b) y position for an iRobot Create2 placed near the origin. Red line shows the mean, green lines show a 95% confidence interval .....	22
Figure 3-12 Estimated (a) x and (b) y position for a Parrot Minidrone placed near the origin. Red line shows the mean, green lines show a 95% confidence interval .....	23
Figure 3-13 Estimated yaw for (a) iRobot Create2 and (b) Parrot Minidrone at near 90° orientation. Red line shows the mean, green lines show a 95% confidence interval .....	24
Figure 3-14 (a) Drone hovering, green line shows measured height of 820 mm (b) drone changing altitude .....	26
Figure 3-15 Kilobots localized using circle detection .....	32



Figure 4-1 Forces acting on the drone with body frame shown in red (a) common model (b) model with body frame B rotated 45 degrees.....	34
Figure 4-2 Quadrotor propeller orientation when $\theta \neq 0$ . Inertial frame A is shown in green, body frame B is shown in red.....	35
Figure 4-3 Simulink model of the quadrotor.....	37
Figure 4-4 Quadrotor model with PID controllers .....	38
Figure 4-5 Simulated Drone Position Moving from (-700, -500) mm to (700, 500) mm ...	39
Figure 4-6 (a) x and (b) y positions versus time for a drone being controlled from StarL. Green lines indicate the goal position. ....	42
Figure 5-1 Illustration of the modified distributed travelling salesman problem.....	47
Figure 5-2 State machine diagram for the MDTTS App.....	49

## List of Tables

Table 3-1 Threshold values used to isolate yellow and magenta regions .....	12
Table 3-2 Means, 95% Confidence Intervals and Ranges for Estimated $x_i, y_i$ Positions.....	24
Table 3-3 Means, 95% Confidence Intervals and Ranges for Estimated $\psi_i$ Positions....	25
Table 3-4 Average Sampling Rates Over 500 Frames.....	27
Table 3-5 Parrot ARDrone center tracking trajectory results in the plane plotted versus time for different altitudes (measured height from ground) and motion of the quadcopter. ....	27
Table 3-6 Images of localized ARDrone for different scenarios .....	29

# **Chapter 1**

## **Introduction**

Research in swarm robotics [25, 5, 20, 17, 27, 19] has become increasingly popular due to the availability of capable and inexpensive robotic platforms and the many possible applications of such systems, which are representative of distributed cyber-physical systems (CPS) [24, 18, 23, 22, 21, 26, 11]. Swarm robotics is generally defined as the coordination of simple autonomous robots with local sensing capabilities to execute a desired task [6]. Potential applications include but are not limited to tasks that cover large areas, such as environmental monitoring and mapping, tasks where the loss of inexpensive individual robots is preferable to the loss of a single more complicated robot, such as in landmine detection, and tasks in which redundancy is critical such as military communication networks where individual nodes may have a high probability of being destroyed by an enemy [37]. Before these applications can be fully realized, more research needs to be done, much of which will be performed in a laboratory setting. The purpose of this work was to develop new tools and techniques to make swarm robotics research in a laboratory setting easier.

To perform swarm robotics research a system capable of localizing the robots is typically required. Chapter 3 of this work describes an inexpensive and easy-to-use system, where the robots are assumed to be circular in shape, or composed of circular shapes. The system uses Microsoft Kinect cameras with depth sensing mounted on ceilings looking downward, although it is applicable to other cameras such as simple webcams, especially if depth information is not needed, such as for ground robots. For many types of robots (such as helicopters, quadcopters [32], hexacopters, iRobot Create (1 and 2), Rice r-one [33], Harvard Kilobots [35], etc.), this is an effective location as it allows viewing a circular shape orthogonally from above and maximizes field-of-view for

localization of the robots to increase the effective area of their workspaces. After describing the system and localization script, results obtained with the system are presented including estimations of precision and accuracy, measured sampling rates, and plots of tracked robot trajectories.

Quadrotors are a popular choice for distributed robotics applications as they provide a capable aerial platform and reliable models are now available from several companies. Chapter 4 develops a simple model for a quadrotor when only planar motion is considered. The model considers only pitch and roll commands, as these commands are commonly available to users of commercial drones like Parrot's ARDrone and Minidrones. It also does not require the estimation of difficult to measure parameters like quadrotor moments of inertia. The model developed provides equations for acceleration in the  $x$  and  $y$  direction of an inertial coordinate frame. These equations were used to make a Simulink model which took integrals to find velocities and positions. PID controllers were added to the model to simulate the actual control of the drone, and the parameters were used with a PID controller written in Java to move Parrot Minidrones to waypoints in the  $x$ - $y$  plane. Control for the Minidrones was integrated into the StarL framework, so StarL applications can now be run with quadcopters in addition to iRobot Creates (1 and 2).

Software is needed in any swarm robotics application to control the robots. The StarL framework provides developers with the tools needed to create distributed robotics applications quickly and easily [31, 42]. The applications can be run in simulation or deployed to hardware using Android tablets. Chapter 5 describes two new StarL applications developed in this work. The first is a simple application where robots move to unique waypoints, wait for all other robots to arrive at their waypoints, then move to the next waypoint in a sequence. The second solves a modified distributed travelling

salesman problem where robots must visit sets of waypoint in a certain order, and each robot must visit at least one waypoint in each set.

## **Chapter 2**

### **Background and Related Work**

#### **Robot Localization**

Localization and mapping are fundamental problems in robotics and have been extensively investigated. Simultaneous localization and mapping (SLAM) is frequently used to create a representation of the environment (a map) and determine where a robot is in relation to that representation (its location in the map) [10, 12, 3]. Landmark and beacon-based methods are common approaches, where some knowledge of the environment (identification of certain landmarks) or the robots progression through the environment (entering or crossing certain regions) [29, 4, 39]. Many approaches rely on probabilistic methods, such as belief propagation to share robot team location information in swarm systems [13]. Multi-camera localization and motion-capture setups such as those now commercially available by OptiTrack and Vicon are extensively used in robot localization [28]. Calibration of multi-camera systems is a challenging problem (i.e., movement of one camera requires recalibration) and many methods have been developed to perform automatic or semi-automatic calibration [7]. Circle detection from images and videos is a classic computer vision problem [8, 9] and also has many solutions [30]. In this work, the primary method used is based on the Hough transform [41, 2]. Circle detection has been used to localize robots in a landmark-based setup, where landmarks are assumed to be circular shaped, and robots traverse the environment for localization purposes by identifying these landmarks in the environment [36]. Localization of quadcopters from video data has been extensively studied. For example, some systems assume the robots have certain markers on them [16]. In contrast to many of these existing methods, the only assumption made about the robots is that they are circular in shape or composed of circular shapes as many robots

commonly are [40, 35, 32, 33]. The method is extensible to any robots meeting this assumption, and any robot to which a circular shape can be attached.

### **Dynamic Model of a Quadcopter**

Dynamic models of quadrotor drones are well described in the literature [34, 15, 38]. These models provide a detailed description of quadcopter dynamics taking into account all forces and torques acting on the drone. These include the torques produced by the motors, the torques induced from propeller thrust, the propeller thrust forces themselves, the gravitational force, and drag forces. These models tend to focus on the lowest level of control i.e. how should voltages be applied to the motors to achieve a desired orientation and/or angular or vertical speed, but some do build on the low level controller to provide a high level control model capable of moving the drones along sequences of waypoints. A model this complex is well suited for building a custom quadcopter, but is not needed when using a commercially available drone. In fact, this type of model is not desirable as it requires knowledge of parameters that are difficult to measure such as the quadcopter's moments of inertia, and requires the ability to directly control motor speed. On products like Parrot's ARDrone and Minidrones this is not available, as commands are sent as pitch and roll angles, and angular and vertical velocities rather than motor speeds. In contrast to the more detailed models, this thesis presents a simple model suitable for moving a commercially available quadcopter in the x-y plane by taking as input only pitch and roll angle commands.

### **StarL**

StarL is a framework that allows developers to easily create distributed robotics applications. StarL provides several primitives useful for developing distributed applications including mutual exclusion, point-to-point motion, and leader election, and many others [31, 42]. StarL is implemented in Java, and can be run in simulation on a

desktop or in hardware using Android devices and supported robotic platforms. Several interesting applications have already been developed including distributed search, light painting, and traffic intersection coordination applications [31]. In this work the StarL framework is extended by adding support for more robots and developing two new applications. Support was added for the iRobot Create2 by developing a Bluetooth to serial communication bridge using a Raspberry Pi, allowing an Android device running StarL to send Bluetooth commands to the robot. Support was added for Parrot Minidrones by refactoring sample application code provided with the Parrot SDK and creating a motion automaton class for the drone. Two new applications, Follow App and Modified Distributed Travelling Salesman App were developed and are described in Chapter 5.



## Chapter 3

### Video Localization System

#### State Space Representation and Estimation

The pose  $p_i$  of a robot  $i$  is the tuple of its real position in Euclidean coordinates  $(x_i, y_i, z_i) \in \mathbb{R}^3$  and its orientation about these axes,  $(\phi_i, \theta_i, \psi_i)$ , known as Euler angles. The subscript  $i$  is dropped when the robot  $i$  is clear from context. These quantities are estimated in two reference frames: one from the perspective of the camera, denoted by  $\hat{p}$ , and one denoted from the perspective of the robot, denoted by  $p$ . To ensure circular shapes are visible in the camera's orthogonal field-of-view, suppose  $\phi_i = 0$  and  $\theta_i = 0$ , so only the planar orientation  $\psi_i$  is relevant. For many robots (such as quadcopters under nominal operating conditions and all ground robots on nearly flat surfaces) this assumption is valid, although acrobatic flight of helicopters and other rotorcraft may violate this assumption yielding other conic sections like ellipsoids instead of circles. Additionally, for ground robots on nearly flat surfaces, one may assume  $z = 0$ , although from the camera's ceiling mounted point-of-view, the pose  $\hat{p}$  would have  $\hat{z} = d$  where  $d$  is the distance of the camera to the robot.

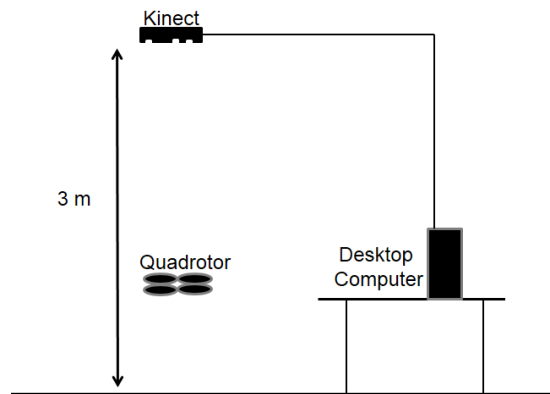
The purpose of the localization system is to find an estimate of the Euclidean coordinates in some reference frame, specifically to find  $(\hat{x}_i, \hat{y}_i, \hat{z}_i) \in \mathbb{R}^3$  that is near to  $(x_i, y_i, z_i)$ , i.e., such that  $\|(\hat{x}_i, \hat{y}_i, \hat{z}_i) - (x_i, y_i, z_i)\| \leq \epsilon$  for some small nonnegative  $\epsilon \in \mathbb{R}$ . The reference frame is assumed to be centered at the camera, so all measurements are relative to that point. For planar ground robots like the iRobot Create, only estimating the planar coordinates  $(x_i, y_i)$  and yaw angle  $\psi_i$  is necessary. For aerial robots like quadcopters, estimating the altitude (height) as the  $z_i$  coordinate is also necessary, and

depth information from the camera may be utilized. If calibration has been performed, the radii length of the circles in the image could instead be used to estimate the altitude.

## Experimental Set-up

### *Camera System Set-up*

The localization system uses a ceiling-mounted camera (in this case, the Microsoft Kinect Version 1) viewing orthogonally downward toward the ground, as shown in Figure 3-1. The camera is approximately 3 m from the ground, and is connected to a desktop computer running MATLAB version R2014a with the Image Acquisition Toolbox and Microsoft Kinect support package installed. The Kinect has an RGB camera and an infrared depth sensor, and both record data with a 640x480 resolution at maximum of 30 frames per second (fps). The depth sensor measures the distance from the camera to any objects in the camera's field of view in millimeters. In addition to MATLAB, Kinect for Windows SDK, Kinect for Windows Drivers and Kinect for Windows Runtime (all v1.8) were installed on the desktop computer.



*Figure 3-1 Experimental set-up of video localization system*

White and gray patterned photo backdrop paper was taped to the floor to allow a light background for the dark colored robots. Additionally, this backdrop improved the

ability of the quadrotors to hover in place, as the carpet in the lab proved too uniform for their built-in hover algorithm to function properly.

#### *Robot Set-up*

The robots were set-up as shown in Figure 3-2, with yellow and magenta circles/triangles attached to form a line through the center of the robots. These were used to estimate the robot yaw  $\psi_i$  angles. Colored paper was attached to the center of each robot to enable unique identification. Additionally, a circle made from poster board was attached to the quadrotor's protective hull to enable tracking by circle detection. No attachments were required for the Kilobots, as they are already circular and the yaw was not estimated.

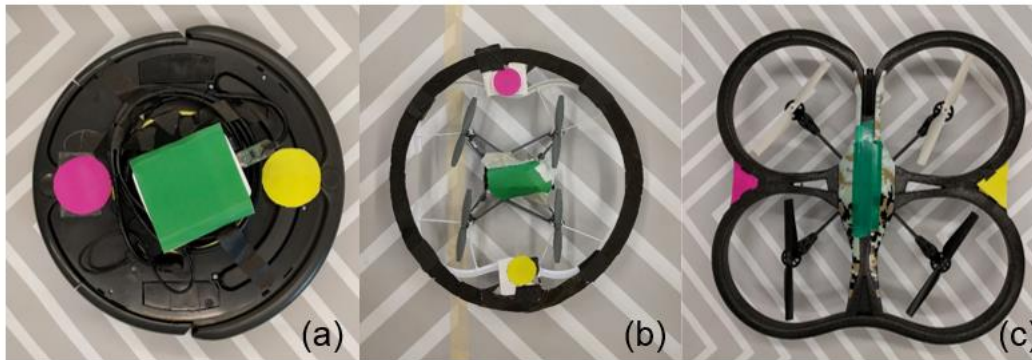


Figure 3-2 Set-up for (a) iRobot Create2 (b) Parrot Minidrone (c) Parrot ARDrone

### **Position Estimation Using Circle Tracking**

The localization script utilizes the `imfindcircles` MATLAB function, which relies on a circular Hough transform to detect circles in images. This function accepts several parameters, and for this study `ObjectPolarity` was set to dark, `Sensitivity` was 0.92, and the minimum and maximum radii values varied depending on the type of robot and estimated  $z_i$  value. The function returns the center coordinates and radii in pixel values and a metric indicating the relative strength for all circles found in the image. These pixel coordinates can be transformed from the camera frame to a real-world frame

with dimensions in millimeters (mm) using a mm per pixel value calculated by dividing the invariant size of the robot radius by the radius value in pixels provided by the `imfindcircles` function. The robot's  $(x_i, y_i)$  position in this coordinate frame can be calculated in a general case using the following equations,

$$x_{center_{mm}} + (x_{measured_{px}} - x_{center_{px}}) * mm/px_d \quad (1)$$

$$y_{center_{mm}} + (y_{measured_{px}} - y_{center_{px}}) * mm/px_d \quad (2)$$

where  $x_{center_{mm}}$  and  $y_{center_{mm}}$  are the x and y center coordinates of the real-world frame in millimeters (both 0 for this study),  $x_{measured_{px}}$  and  $y_{measured_{px}}$  are the x and y pixel coordinates of the item being tracked,  $x_{center_{px}}$  and  $y_{center_{px}}$  are the x and y center coordinates of the image in pixels (320,240), and  $mm/px_d$  is the millimeters per pixel value at a measured distance  $d$  from the camera.

For the iRobot Create 2 and other ground robots the distance from the camera remains constant, so the radius value measured in pixels is also nearly constant, with only very small fluctuations (std 0.138 px over 300 frames). This allows constant minimum and maximum radius values (25 to 35 pixels in this study) to be used in the `imfindcircles` function, and a constant value for  $mm/px_d$  to be calculated by dividing the robot's diameter in mm by an average radius value in pixels. For the camera height used in this study, this value was found to be 5.6 mm/pixel.

A coordinate frame in the real-world was defined where the origin corresponded to the image center pixel and the axes of the camera frame and real-world frame were aligned, but with the y and z axes in opposite directions as show in Figure 3-3. The viewable area of the coordinate frame ranged from -1792 to 1792 mm in the x direction, and -1344 to 1344 mm in the y direction.

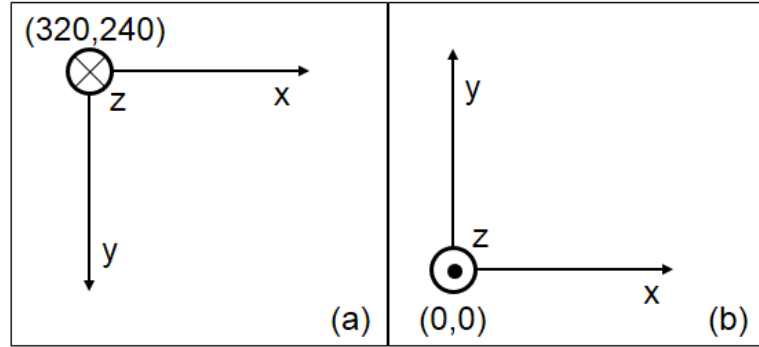


Figure 3-3 Coordinate systems as viewed from above, (a) camera with z-axis positive towards the ground (b) real-world with z-axis positive towards the ceiling

For robots like quadrotors, which are not confined to the ground, the  $mm/px_d$  value changes with the robot's altitude. For these robots, a  $mm/px_d$  value was calculated in each frame by dividing the quadrotor's radius in mm by its measured value in pixels. Additionally, constant minimum and maximum radius values could not be used with the `imfindcircles` function. Instead, a radius range was calculated using a 4<sup>th</sup> order polynomial equation with the quadrotor's estimated distance from the camera (see Altitude Estimation Using Kinect Depth Sensor) as input. To construct this equation, data was acquired at various depths and the radius size was plotted versus the estimated depth. The MATLAB curve fitting tool was then used to fit a function to the data, and the minimum and maximum radius values were calculated by subtracting and adding 5 to the value output by the function.

### Yaw Estimation Using Color Detection

The iRobot Create2 is moved by two fixed wheels that can be driven at different speeds allowing the robot to move in a forward or backward direction or change its yaw angle  $\psi_i$ . Therefore, in order to move the robot in the x-y plane, the yaw angle  $\psi_i$  must be estimated in addition to its  $(x_i, y_i)$  position. This estimation is not strictly necessary for a

quadrotor as it can move in the  $x$  and  $y$  directions independently by varying roll and pitch angles, but it does allow for yaw correction when it deviates from the desired value.

To estimate  $\psi_i$ , yellow and magenta circles/triangles were attached to form a line along either the  $x$ -axis (quadrotor) or  $y$ -axis (iRobot Create2) of the coordinate frame attached to the robot. To isolate these colored areas, the RGB values of pixels not contained within the robot's detected circle(s) were set to zero. The image was thresholded using the RGB values shown in Table 3-1 to produce two binary images indicating the location of yellow or magenta pixels.

*Table 3-1 Threshold values used to isolate yellow and magenta regions*

	Yellow	Magenta
Red	$140 < px < 240$	$120 < px < 200$
Green	$140 < px < 230$	$px < 70$
Blue	$px < 110$	$60 < px < 130$

For each image, the MATLAB function `bwlabel` is called to label all connected components. The MATLAB function `regionprops` is then used to find the area and centroid of all connected component regions. The centroid from the region with the largest area is then selected as the center of the colored area. A vector  $\mathbf{A}_i$  is created by subtracting  $\mathbf{C}_m$  from  $\mathbf{C}_y$  where  $\mathbf{C}_m$  and  $\mathbf{C}_y$  are vectors containing the  $x$  and  $y$  centroid coordinates of the magenta and yellow areas for robot  $i$ , respectively. The yaw angle is then calculated using the following equation [14],

$$\psi_i = \arctan2(\mathbf{A}_i \cdot \mathbf{B} + \det \begin{bmatrix} \mathbf{A}_i \\ \mathbf{B} \end{bmatrix}) \quad (3)$$

where  $\mathbf{B}$  is the either the unit vector along the camera frame  $x$ -axis (quadrotor) or  $y$ -axis (iRobot Create2).

### Altitude Estimation Using Kinect Depth Sensor

The altitude ( $z_i$  position) of the quadrotor is estimated using the depth image acquired by the Kinect. The quadrotor's distance from the camera is estimated by finding the median value of all pixels within a bounding box (see Initial Location and Identification of Robots for details on bounding box) from the depth image between 2900 (~100 mm above ground level) and 50 mm. If no pixels fall within this range, the distance is assumed to be 100 mm above ground level. Otherwise, the altitude is calculated by finding the difference between the distance from the camera to the ground and the estimated distance from the camera to the drone. Estimation below 100 mm is unnecessary, as the drone is unstable at altitudes this low.

### Localization Program Flow

#### *Initial Location and Identification of Robots with `findBots` Function*

Figure 3-4 (a) shows the overall flow of the tracking script. Before execution, the user must specify the number of each type of robot to be tracked. The variables  $M$ ,  $A$ , and  $R$  represent the number of minidrones, ARDrones, and iRobot Create2s to be tracked, respectively. The first step in the tracking algorithm is to create an empty array of Robot objects of length  $N$  ( $N = M + A + R$ ) called *botArray*. Each Robot object in *botArray* represents one tracked robot and contains information including the robot's center position, radius size, bounding box dimensions, yaw angle, type, and unique color identifier. The *botArray* variable is defined to be global, allowing it to be updated in each frame using sub-functions as the robots change positions. Next, a Boolean variable named *Found* is set to false, and a frame (RGB and depth) is captured using the Kinect. After the color and depth frames are acquired, the value of *Found* is tested. If *Found* is false the `findBots` function is called, otherwise the `trackBots` function is called. The purpose of `findBots`, which is shown in Figure 3-4 (b), is to perform an initial

localization of the robots by performing circle detection on the entire color image. When `findBots` has localized all robots it returns true, and `trackBots`, which is called once for each robot but performs circles detection only on a subset of the image pixels, is called in all subsequent frames. The `trackBots` function is explained in more detail later in a later section.

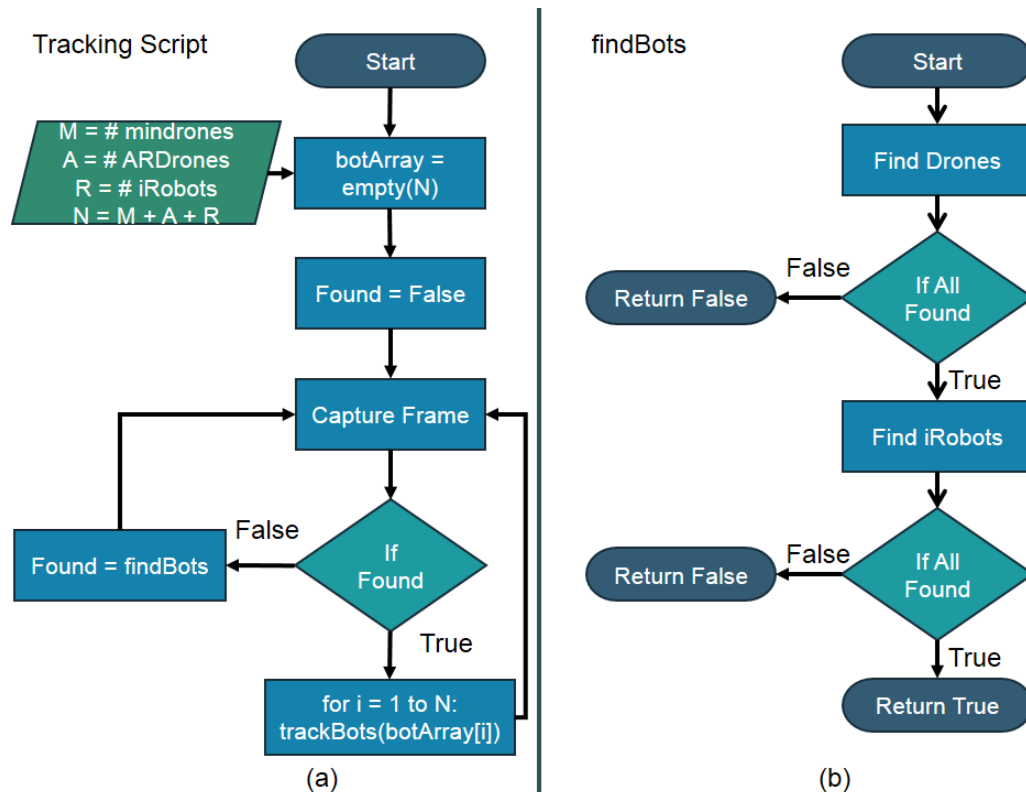


Figure 3-4 (a) Program flow for tracking script (b) program flow for `findBots` function

#### Localization of Minidrones and ARDrones with `Find Drones` and `isARDrone` function

The `findBots` function occurs in two steps, with each step performing a circle detection on the entire color image. Through experimentation, it was discovered that the `imfindcircles` function works best when called with a radius range of approximately 10 pixels, with the expected radius value near the middle of the range. When on the ground, the minidrones have a radius of ~ 20 pixels, while each ARDrone circle radius is



~ 22.5 pixels. Since the radius ranges used for `imfindcircles` would overlap significantly, one circle detection is performed to find both minidrones and ARDrones in the `find drones` step. If all drones are found, the `find iRobots` step is performed, otherwise `findBots` returns false.

Figure 3-5 (a) shows how the minidrones and ARDrones are localized in the `find drones` step. First, circle detection is performed, and if the number of circles found is less than the number of minidrones plus the number of ARDrones times 4 (since each ARDrone consists of four circles) false is returned. Otherwise, the  $M + 4 \cdot A$  strongest circles are selected using the metric values provided by `imfindcircles` and added to an array `c`. Next, the circles are sorted in ascending order by radius size, and the  $M$  smallest circles are added to `botArray` as minidrones.

Figure 3-5 (b) shows the steps taken to add a robot to `botArray`. First, the circle center and radius values of the Robot object are set. Then a bounding box is calculated such that the side lengths are a bounding box factor times longer than the detected circle diameter, and the circle center coincides with the bounding box center, as show in Figure 3-6. Next, the robot is uniquely identified using a simple color detection, where it is assigned the identity white if the circle center pixel value has all RGB values greater than 100. Otherwise it is assigned a red, green, or blue identity value by selecting the maximum of the circle center RGB pixel values. Finally, the yaw is set to 0 and the proper type (MINIDRONE, ARDRONE, or CREATE2) is assigned to the robot.

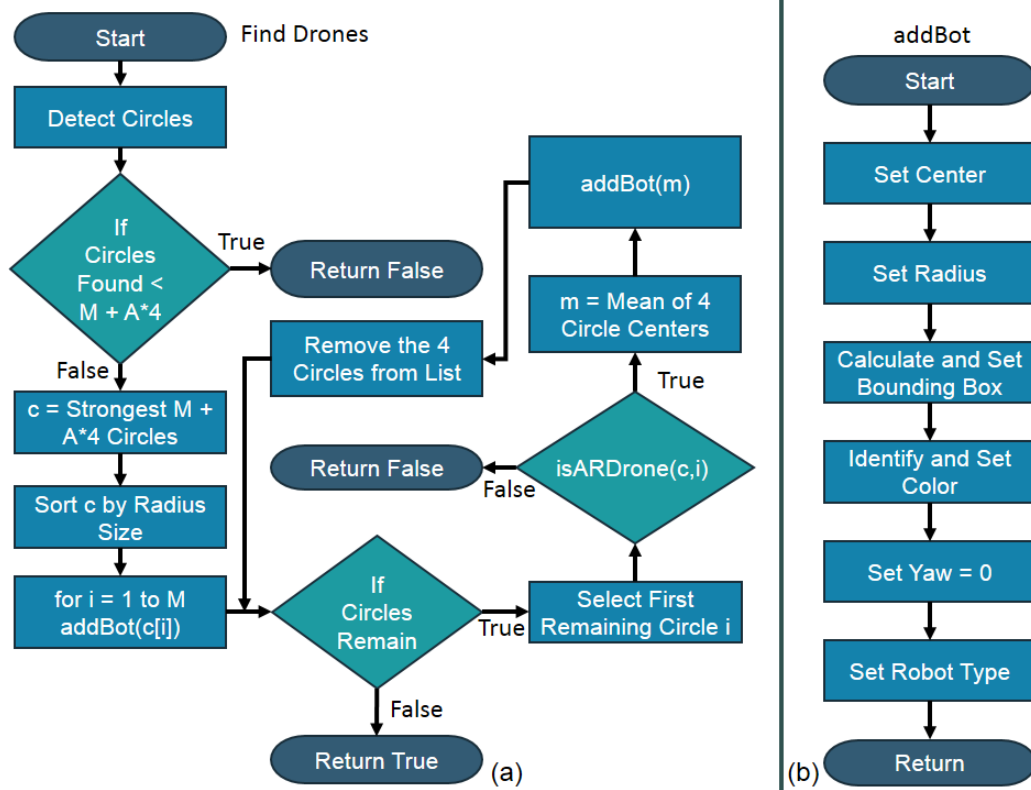


Figure 3-5 (a) Program flow for finding minidrones and ARDrones (b) steps for adding a robot to botArray

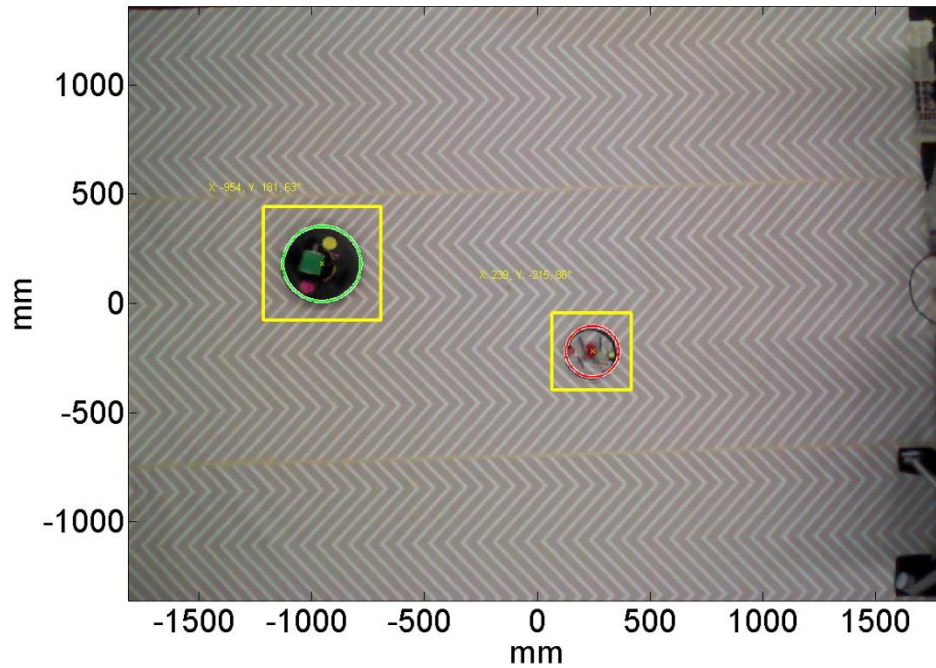


Figure 3-6 Localized minidrone and iRobot Create2 with bounding boxes shown in yellow

After the smallest  $M$  circles have been added to *botArray* as minidrones, they are removed from *c* and  $A*4$  circles remain. A check is then performed to determine if any circles remain in *c*. If  $A = 0$ , true is returned as there are no ARDrones to be tracked and *c* is empty. Otherwise, the first remaining circle is selected, and the selected circle's index  $i$  and *c* are passed to the *isARDrone* function. The *isARDrone* function, which is shown in Figure 3-7, returns false if the selected circle does not belong to an ARDrone, otherwise it returns the selected circle along with the other three circles that make up the drone. If *isARDrone* returns false, false is returned from *findBots*. Otherwise, the mean of the four circle centers returned is calculated as the drone's center. The drone is then added to *botArray*, and the four circles making up the drone are removed from *c*. Another check is then performed to determine if any circles remain in *c*, and the steps are repeated until *c* is empty.

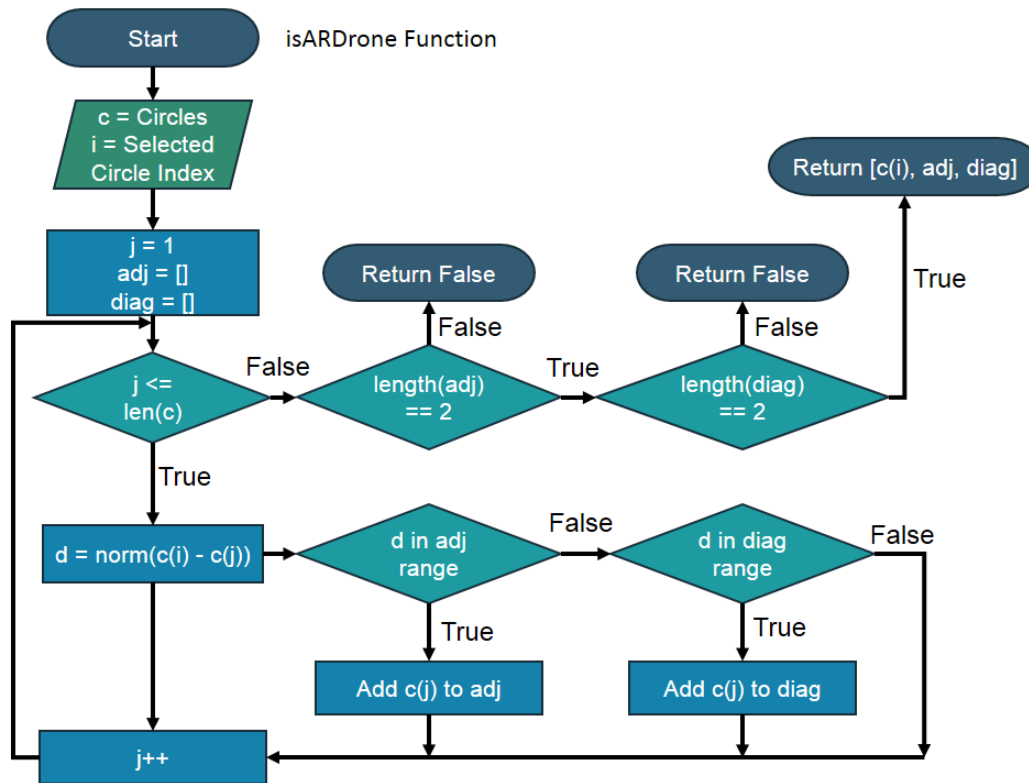


Figure 3-7 Program flow for *isARDrone* function

The *isARDrone* function determines if a circle belongs to an ARDrone by finding the distance between the selected circle  $c[i]$  center and all other centers. If the four circle centers making up an ARDrone are thought of as a square, then when on the ground the distance between  $c[i]$  and the two adjacent centers will always fall within a certain range (in this case 40 to 50 pixels), as will the distance between  $c[i]$  and the diagonal corner (61 to 71 pixels) as shown in Figure 3-8. As the distance is found between  $c[i]$  and the other circles, if the distance falls within the adjacent range of 40 to 50 pixels, it is added to an array named *adj*, and if falls within the diagonal range of 61 to 71 pixels, it added to *diag*. After all distances have been found, if the length of *adj* is two and the length of *diag* is one, then the array  $[c[i], adj, diag]$  is returned. Otherwise, the circle does not belong to an ARDrone, and false is returned.

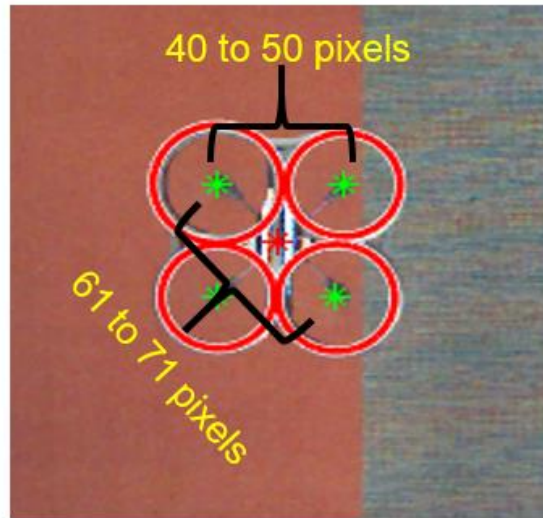


Figure 3-8 Distance ranges between circle centers for an ARDrone on the ground

#### Localization of iRobotCreate2s with Find iRobots

If all the drones are found in the find drones step of `findBots`, control moves to the find iRobots step, shown in Figure 3-9. The iRobot Create2s have radius values of ~ 30 pixels, so a radius range of 25 to 35 pixels is used with the `imfindcircles` function. Since this range does not include the 20 or 22.5 pixel size of the minidrone and ARDrone radii, only circles corresponding to iRobot Create2s will be found when circle detection is performed. If the number of circles detected is less than the number of iRobots  $R$  to be tracked, false is returned. Otherwise, the strongest  $R$  circles are selected and added to `botArray` with type CREATE2 using the process shown in Figure 3-5 (b).

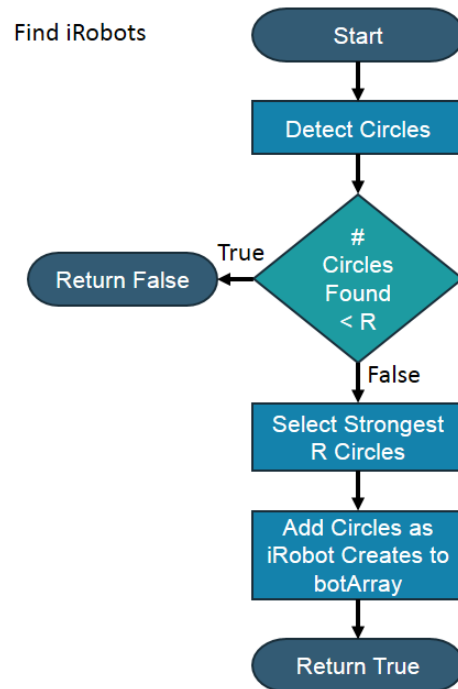


Figure 3-9 Program flow for find iRobots step of *findBots*

#### Tracking Robots after Initial Localization

As shown in Figure 3-4 (a), after all robots are localized with *findBots* the *trackBots* function is called once for each robot in all subsequent frames. Figure 3-10 shows the program flow for the *trackBots* function. The first step in *trackBots* is to make a new image out of the pixels contained within the robot's bounding box. For relatively small numbers of robots (~4-5) performing circle detection on several smaller images results in higher sampling rates than finding circles in the whole image. After the bounding box image is created, if the robot's type is MINIDRONE or ARDRONE, the depth (i.e. altitude) is found as described in Altitude Estimation Using Kinect Depth Sensor, and a radius range is found using the estimated depth. If the robot type is CREATE2, these two steps are skipped. Circle detection is then performed on the

bounding box image. If no circles are found, or less than four circles are found for an ARDrone, the function returns without updating *botArray*. In this case the values from the previous frame are reused for the robot's position, bounding box, and yaw.

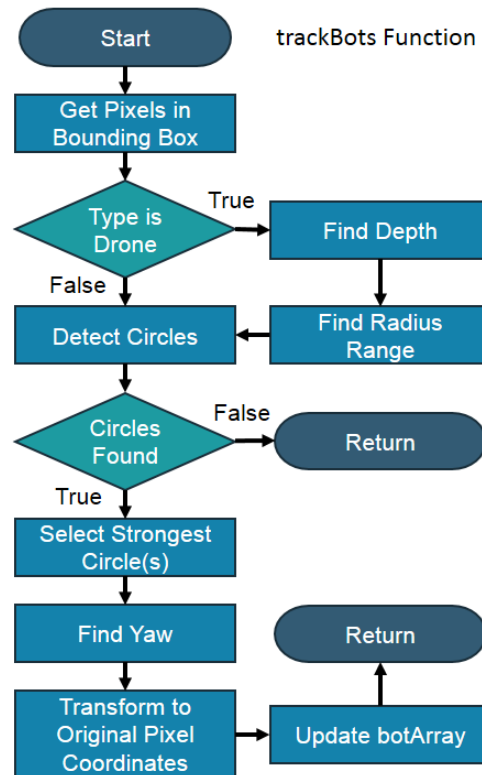


Figure 3-10 Program flow for *trackBots* function

If circles are found, the strongest (or four strongest for an ARDrone) circle(s) are kept and used as the robot's position. Then the yaw is estimated as described in Yaw Estimation Using Color Detection, and the circle center coordinates are transformed from the bounding box image back into the original image. The robot's values in the *botArray* are then updated in a way similar to that shown in Figure 3-5 (b), except that the yaw value is set to the estimated value instead of zero, and the color and type are not updated as these values do not change.

## System Performance

### Position Estimation

The precision and accuracy of the localization system was measured by placing a robot as near to the origin of the real-world coordinate system as possible and capturing 500 frames while the robot was still. Figure 3-11 (a) and (b) show the results obtained for the estimation of an iRobot Create2's  $x_i$  and  $y_i$  positions, respectively, and Figure 3-12 (a) and (b) show the same results for a Parrot Minidrone. The red lines in the figures indicate the means, and the green lines give 95% confidence intervals calculated as the mean  $\pm 1.96$  times the standard deviation. Values for the mean, confidence intervals, and data ranges are given in Table 3-2 for the iRobot Create2 and Parrot Minidrones.

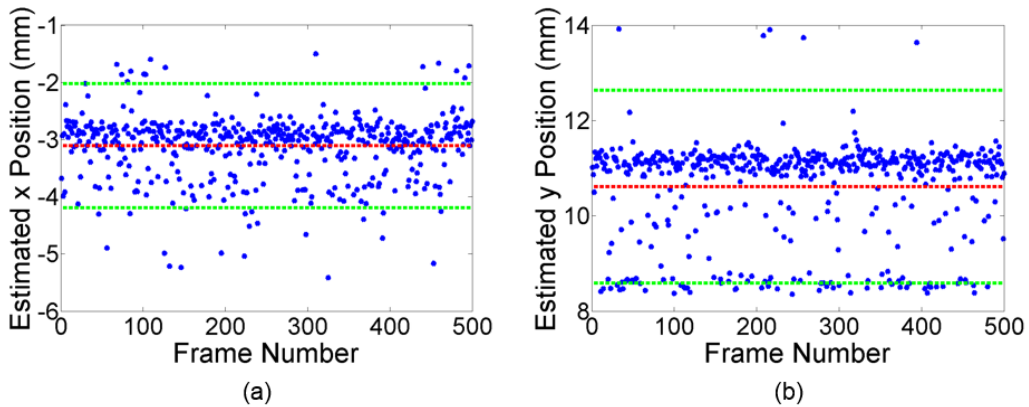


Figure 3-11 Estimated (a) x and (b) y position for an iRobot Create2 placed near the origin. Red line shows the mean, green lines show a 95% confidence interval



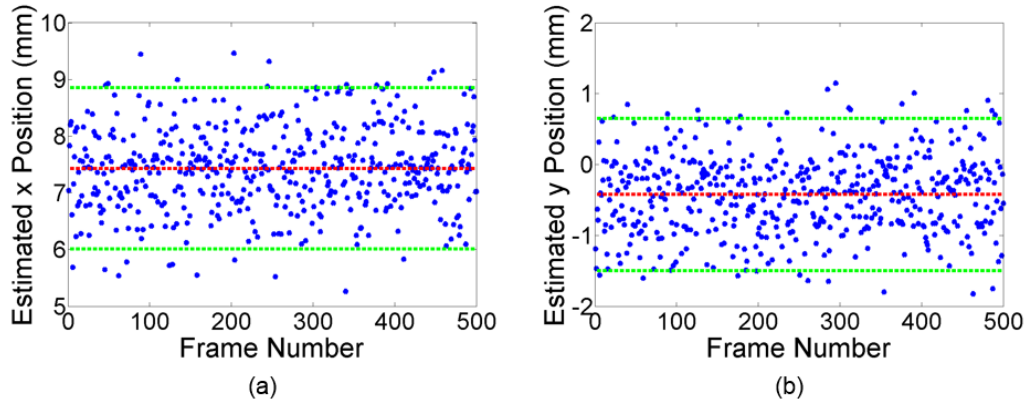


Figure 3-12 Estimated (a) x and (b) y position for a Parrot Minidrone placed near the origin. Red line shows the mean, green lines show a 95% confidence interval

As Table 3-2 shows, the maximum range for all measurements was the iRobot Create2's estimated  $y_i$  value at 5.57 mm. This gives a good estimation for the localization system's precision, which at roughly half a centimeter (cm) is more than adequate for tracking robots in a lab setting. The mean value that deviated the most from the origin was also the iRobot Create2's estimated  $y_i$  value, at 10.61 mm. This suggest an accuracy measurement of approximately 1 cm, however there are several sources of error in this measurement, including the location of the real-world coordinate system origin and the actual location of the robot's center. Since the robot center cannot be reliably placed at the origin of the coordinate system with millimeter accuracy, the confidence intervals and data ranges give a better characterization of the system performance.

Table 3-2 Means, 95% Confidence Intervals and Ranges for Estimated  $(x_i, y_i)$  Positions

	Mean (mm)		95% Confidence Interval (mm)		Data Range (mm)	
	x	y	x	y	x	y
iRobot Create2	-3.12	10.61	-4.19 to -2.02	8.59 to 12.62	3.91	5.57
Parrot Minidrone	7.43	-0.42	6.01 to 8.85	-1.50 to 0.65	4.20	2.97

### Yaw Estimation

The precision and accuracy of the yaw detection was tested by placing the robots as near to a  $90^\circ$  orientation as possible and capturing 500 frames. Figure 3-13 (a) and (b) show the results for an iRobot Create2 and a Parrot Minidrone, respectively.

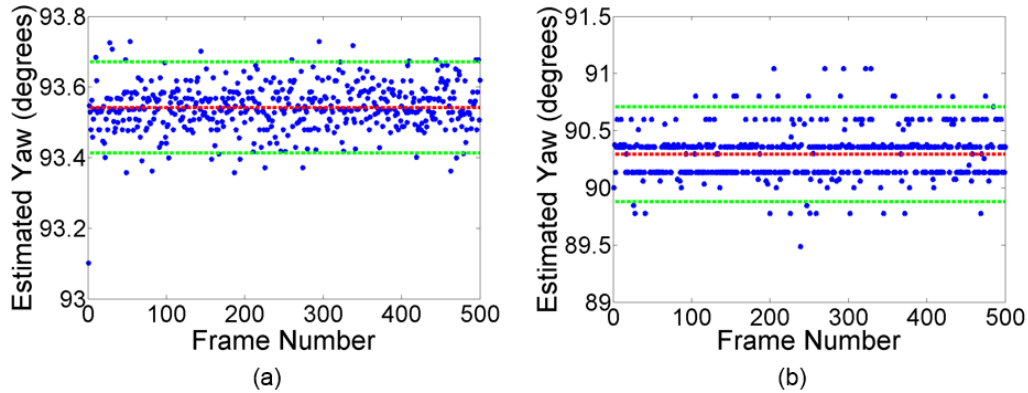


Figure 3-13 Estimated yaw for (a) iRobot Create2 and (b) Parrot Minidrone at near  $90^\circ$  orientation.

Red line shows the mean, green lines show a 95% confidence interval

Table 3-3 shows the mean, 95% confidence interval, and data range for an iRobot Create2 and Parrot Minidrone for the estimated  $\psi_i$  values. The largest range was for the Parrot Mindrone, at  $1.56^\circ$ , which is a good estimate of the yaw detection precision. The largest mean value deviation from  $90^\circ$  was the iRobot Create2 at  $93.54^\circ$ , but similar to the  $(x_i, y_i)$  estimation it is difficult to verify that the robot was actually placed at a  $90^\circ$

orientation. Again, the confidence intervals and data ranges better represent the system performance.

*Table 3-3 Means, 95% Confidence Intervals and Ranges for Estimated  $\psi_i$  Positions*

	Mean	95% Confidence Interval	Data Range
iRobot Create2	93.54°	93.14° to 93.67°	0.63°
Parrot Minidrone	90.30°	89.88° to 90.71°	1.56°

#### *Altitude Estimation*

Altitude estimation accuracy was measured by allowing a Parrot Minidrone to hover in place with no commands being sent. Using a tape measure, the drone's altitude was determined to be approximately 820 mm above the ground. The altitude estimation provided by the localization system is shown in Figure 3-14 (a), where the green line is at the measured 820 mm. The figure shows the estimated altitude as the drone takes off (see Altitude Estimation Using Kinect Depth Sensor for why drone does not start at 0 mm altitude) and then settles at an altitude of 814 mm, closely matching the measured value. Figure 3-14 (b), which shows the altitude estimation for a drone being sent throttle commands to change its height, illustrates the system's ability to estimate altitude as the drone is moving up and down.

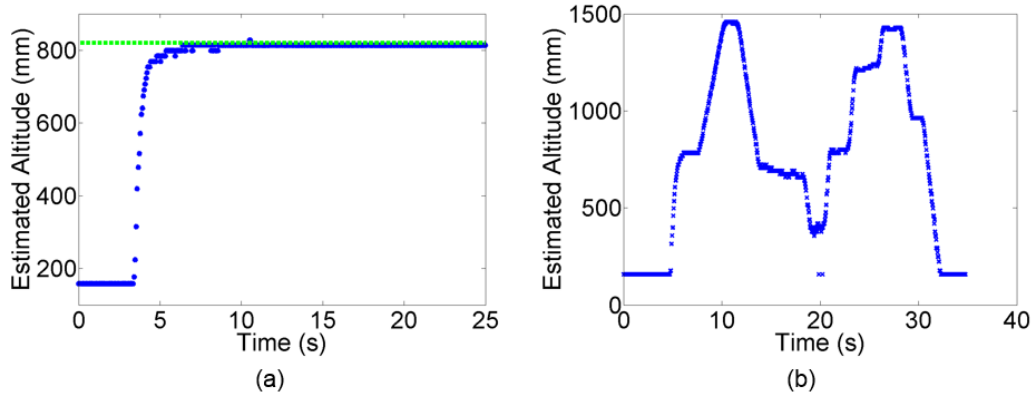


Figure 3-14 (a) Drone hovering, green line shows measured height of 820 mm (b) drone changing altitude

#### Average Sampling Rates

Average sampling rates were measured by placing robots on the ground, and using the MATLAB `tic` and `toc` functions to time the execution of the localization script. The results for the average sampling rate achieved over 500 frames with different numbers of robots are shown in Table 3-4. As expected, the average sampling rate decreases as more robots are tracked, because the `imfindcircles` function must be called once for each robot. However, for a relatively small number of robots, the advantage of calling the circle detection function on smaller images instead of the entire image outweighs the cost of multiple function calls, resulting in higher average sampling rates. During early development, a Parrot ARDrone was tracked by calling the `imfindcircles` function on the entire image in each frame with average sampling rates of approximately 6-7 Hz. Comparing this value with those in Table 3-4 show performance is increased roughly 4 to 15 Hz, depending on the number of robots being tracked, by using the current method of multiple function calls on smaller images.

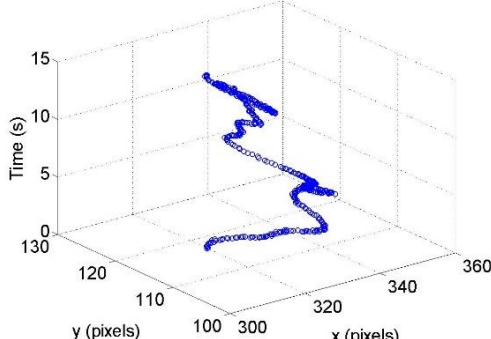
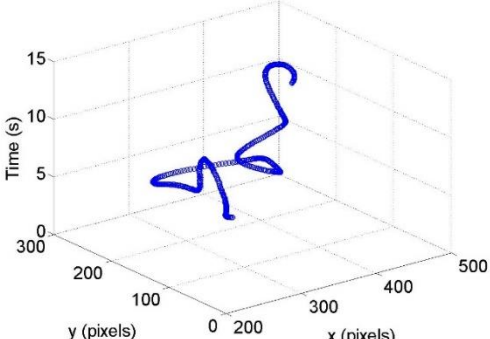
Table 3-4 Average Sampling Rates Over 500 Frames

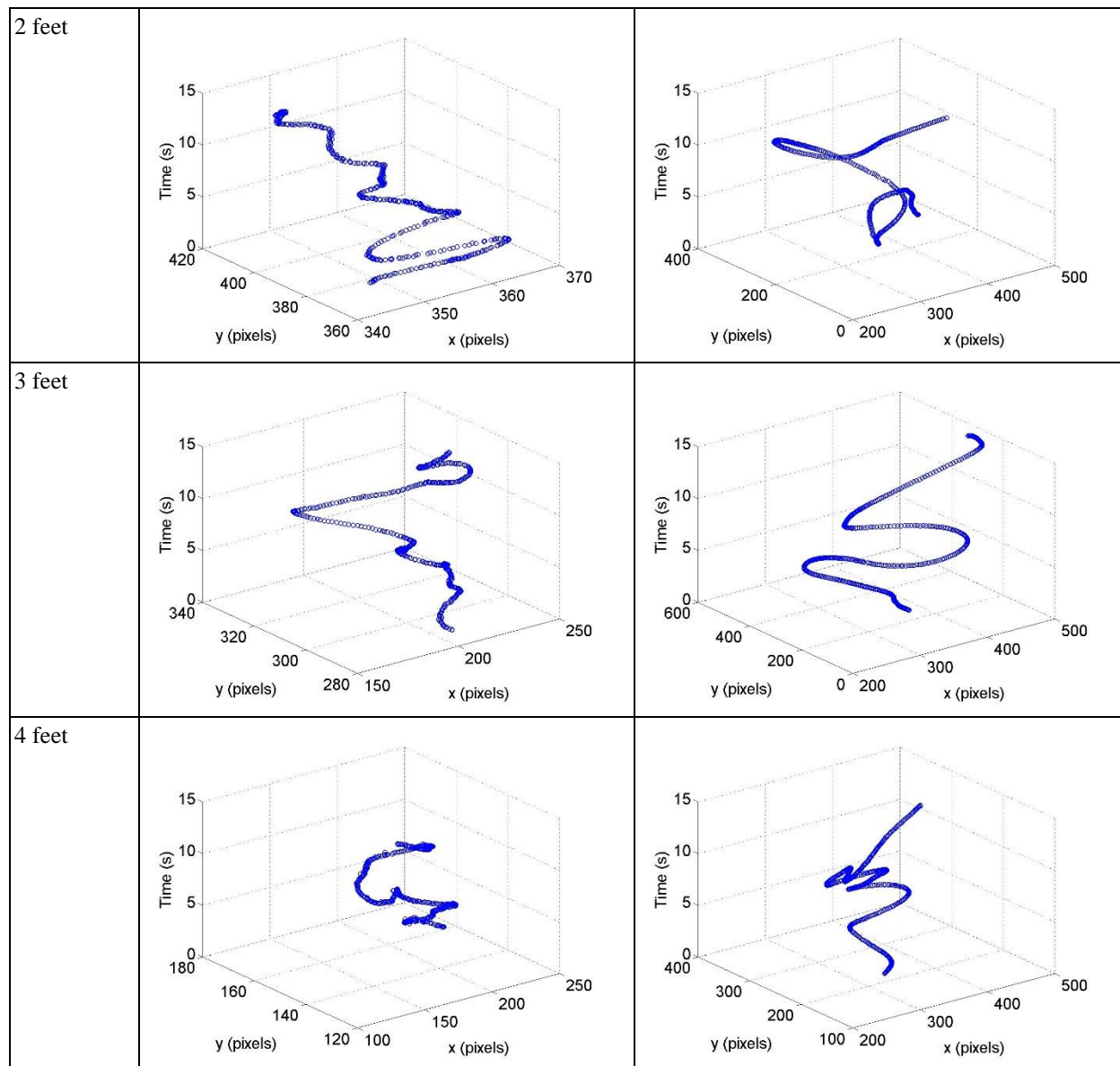
	iRobot Create2	Parrot Minidrone
1	21.00	24.99
2	14.50	15.19
3	12.05	13.96
4	10.02	N/A

### ARDrone Tracking Results

Several different scenarios were examined using the ARDrone quadcopter. Table 3-5 shows estimated  $(\hat{x}_i, \hat{y}_i)$  trajectories in the camera coordinate frame for a drone flying at different altitudes (1, 2, 3, 4, 5 feet and variable) with and without planar motion.

Table 3-5 Parrot ARDrone center tracking trajectory results in the plane plotted versus time for different altitudes (measured height from ground) and motion of the quadcopter.

Altitude	Without Planar Motion	With Planar Motion
1 foot		



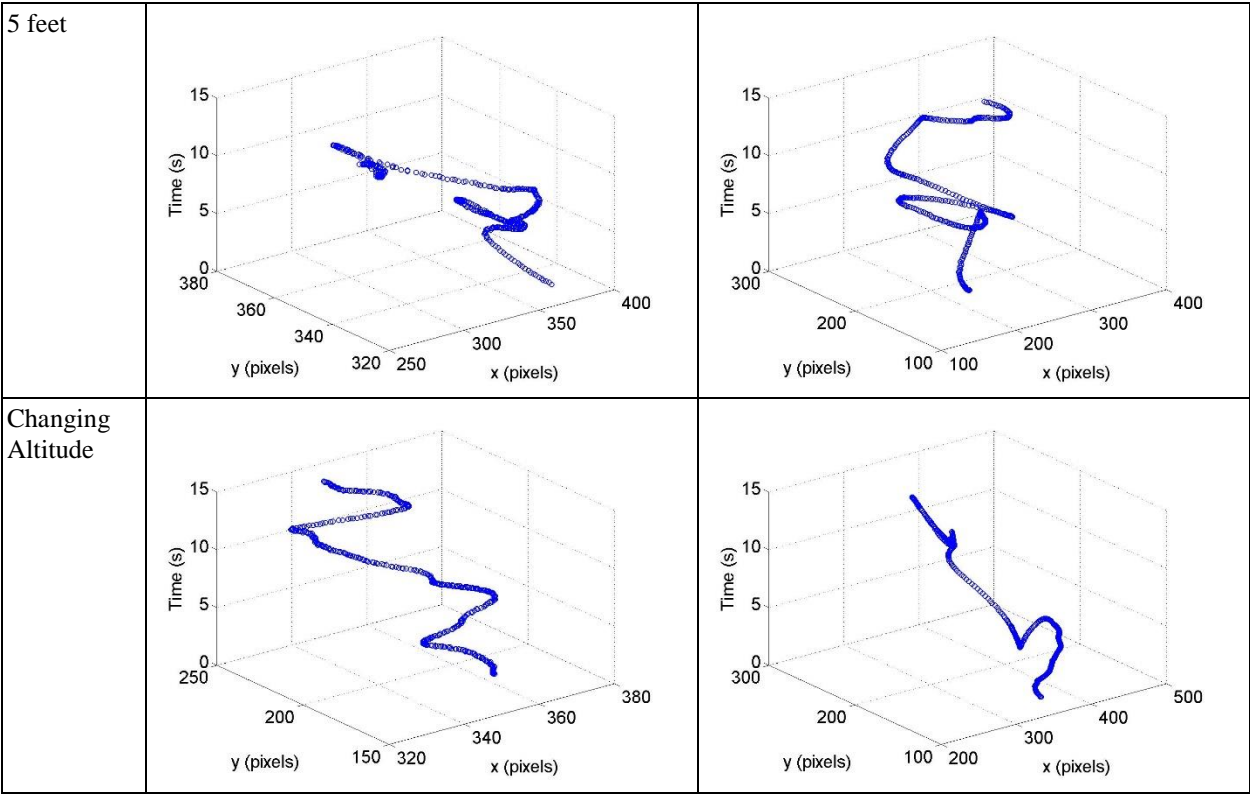
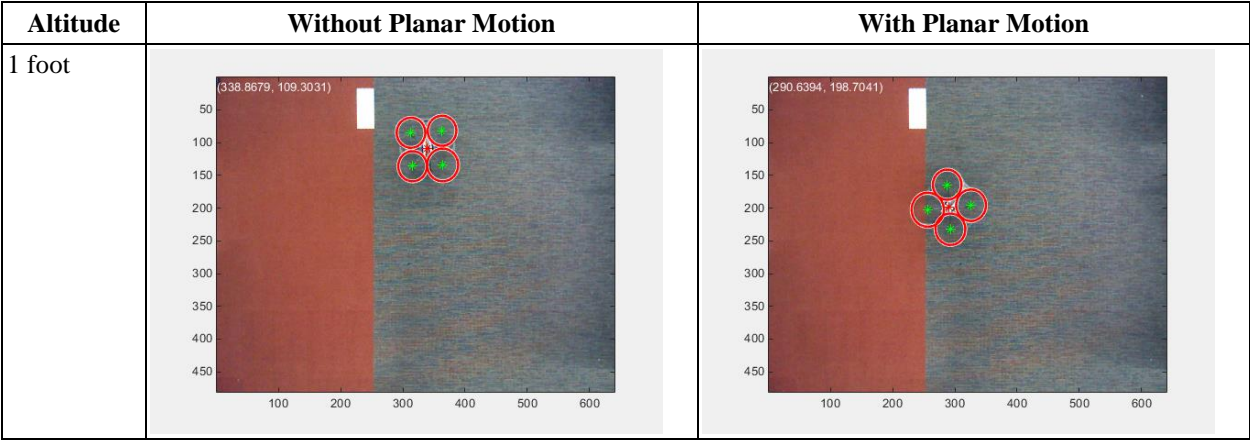
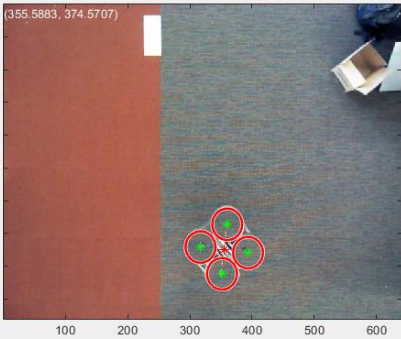
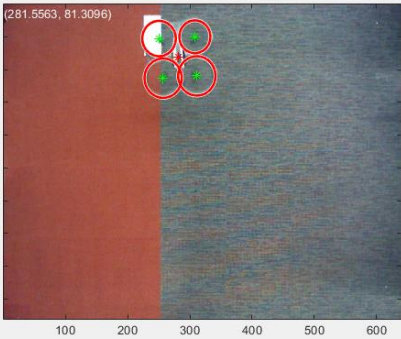
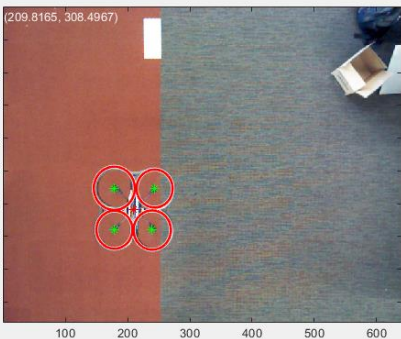
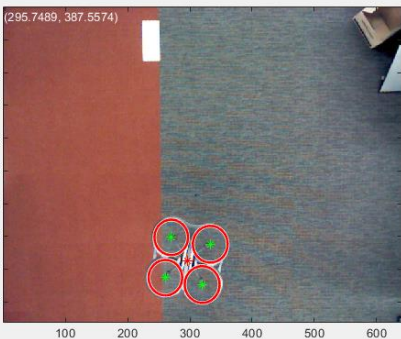
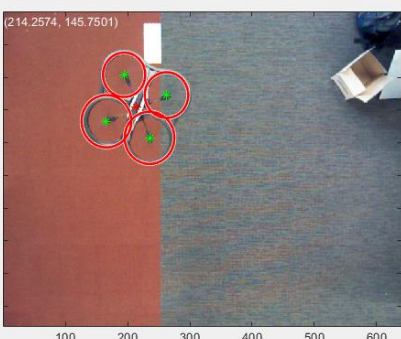



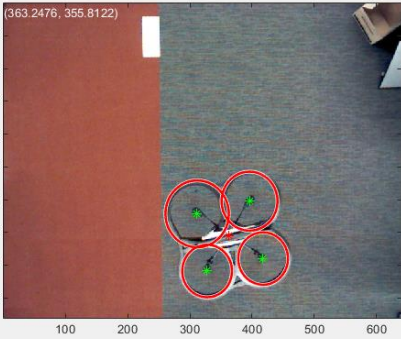
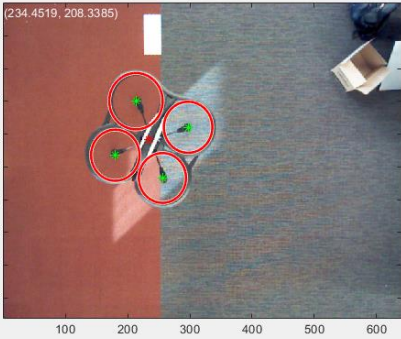

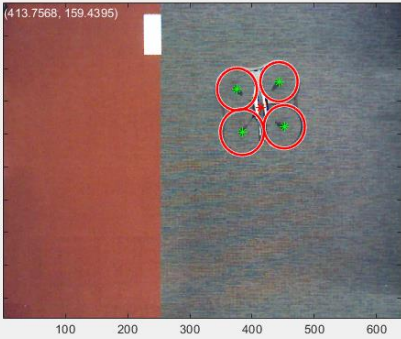
Table 3-6 shows images captured of the ARDrones in different scenarios, with localized circles, circle centers, and drone centers overlaid.

Table 3-6 Images of localized ARDrone for different scenarios



2 feet		
3 feet		
4 feet		



5 feet		
Changing Altitude		

### Tracking Larger Numbers of Robots

Circle detection can be used to track larger numbers of robots, however the method of calling `imfindcircles` multiple times becomes inefficient as the number of robots increases. For example, assuming the trend for iRobot Create2s in Table 3-4 continues, with the sampling rate decreasing by approximately 2 Hz for each added robot, a sampling rate of about 4 Hz would be expected when 8 robots are present. At this point, it would better to call `imfindcircles` once on the entire image, where a sampling rate of 6-7 Hz would be expected.

The inefficiency of multiple function calls for large numbers of robots was illustrated when the script was run on a video of 24 Harvard Kilobots recorded at a 1920 x 1080 resolution using a Kinect for Windows version 2. The sampling rate when calling

`imfindcircles` once for each robot was not measured, but the performance was very poor. An acceptable average sampling rate of 1.85 Hz (lower than 6-7 Hz due to the higher resolution video) was achieved by calling `imfindcircles` once on the entire image, and tracking robots from frame to frame by simply matching them to the robot they were closest to in the previous frame. This simple method works well for Kilobots, which move at a relatively slow speed. Figure 3-15 shows an image of the Kilobots localized using circle detection.



*Figure 3-15 Kilobots localized using circle detection*

## **Chapter 4**

### **Modeling and Control of Parrot Minidrone**

#### **Parrot SDK and StarL Integration**

Parrot provides an SDK with some open source components that allows developers to easily create Android, iOS, and Unix applications for drone control [1]. As of this writing the current version of the SDK is 3.8, which allows control of the Cargo Minidrone, along with several other Parrot products. Included with the SDK are libraries needed for communicating with the drone and example applications for each platform. The SDK allows several commands for drone control, including take-off, landing, emergency, pitch, roll, yaw, and throttle. Pitch and roll commands are sent with values ranging from -100 to 100. These values represent a percentage of the maximum allowed pitch and roll angle, which can be set with the SDK. Similarly, the yaw and throttle commands are sent as percentages of maximum angular and vertical speed, which can also be set using the SDK. A flag bit is also available, which toggles between hover/non-hover modes. When in hover mode, the drone will ignore any pitch and roll commands, and attempt to stay in the same location at the same orientation and height.

To integrate minidrone control into StarL, the DeviceControllerListener interface and DeviceController class provided with the sample Android application were added to the StarLib motion folder. The MainActivity and PilotingActivity classes were refactored into a single class named MiniDroneBTI, which provided methods for connecting to the drone and sending piloting commands. The library files from the SDK were added to the StarLib library folder, and added as dependencies to the StarLib module using Android Studio.

### Equations for Quadrotor Planar Acceleration

To develop a model for planar motion of a quadrotor, the forces acting on the drone must be considered. Figure 4-1 (a) shows the free body diagram of a quadrotor. The forces  $f_i$  (for  $i = 1$  to 4) represent the forces generated by the propellers, and  $mg$  represents the gravitational force where  $m$  is the quadrotor's mass and  $g$  is the gravitational constant  $9.81m/s^2$ . A body frame is attached to the drone with  $\tilde{x}$  and  $\tilde{y}$  axes along lines connecting the rotors and  $\tilde{z}$  in the vertical direction. The pitch, roll, and yaw angles denoted by  $\theta$ ,  $\phi$ , and  $\psi$  represent the drone's orientation with respect to the  $\tilde{x}$ ,  $\tilde{y}$ , and  $\tilde{z}$  axes, respectively. The body frame shown in Figure 4-1 (a) is typical of drone models found in the literature [15], but when given pitch and roll commands the Parrot minidrone does not rotate about these axes. Instead, it rotates about the axes shown in Figure 4-1 (b), which shows the quadrotor as seen from above, with the  $\tilde{z}$  axis (not shown) coming out of the page. This body frame  $B$  is simply rotated 45 degrees in a clockwise direction from the one shown in Figure 4-1 (a).

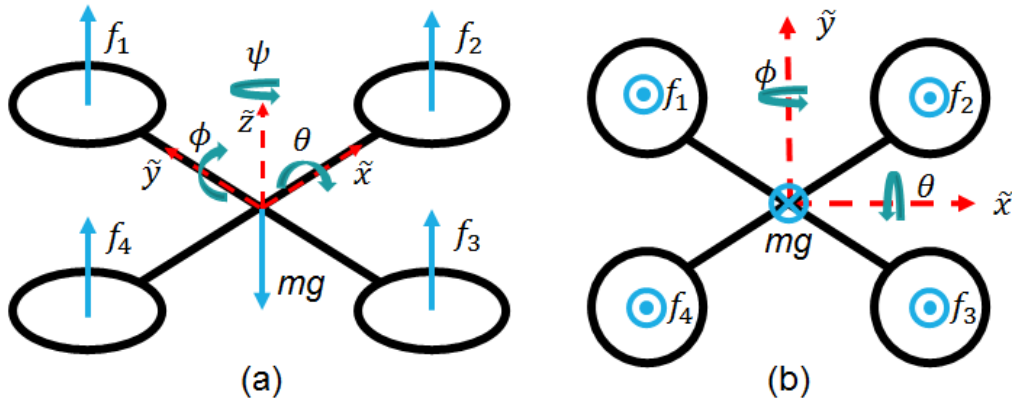


Figure 4-1 Forces acting on the drone with body frame shown in red (a) common model (b) model with body frame B rotated 45 degrees

To derive the model for planar motion, first suppose that a quadrotor in flight has  $\theta = 0$ ,  $\phi = 0$ , and  $\psi = 0$ . In this case, the axes of  $B$  and an inertial frame  $A$ , which is defined as the coordinate system shown in Figure 3-3 (b), are aligned, and the drone will not be moving in the x-y plane. Now suppose a pitch command is supplied to the quadrotor so that  $\theta \neq 0$ . Each propeller will then be oriented as shown in Figure 4-2, where the propeller force now has components in the  $-y$  and  $z$  directions. To determine the magnitude of these forces in the inertial frame  $A$ , a rotation matrix can be applied. First, let  $F$  represent the sum of all propeller forces, i.e.,  $F = \sum_{i=1}^4 f_i$ . Then the propeller forces in  $B$  can be represented by the vector  $[0 \ 0 \ F]^T$ . Multiplying this vector and the matrix for rotation about the  $\tilde{x}$ -axis gives the following,

$$\mathbf{P}_A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ F \end{bmatrix} = \begin{bmatrix} 0 \\ -F\sin(\theta) \\ F\cos(\theta) \end{bmatrix} \quad (4)$$

where  $\mathbf{P}_A$  is a vector that represents the propeller forces in  $A$ . The drone will now be accelerating in the negative  $y$  direction due to the horizontal components of the propeller forces  $-F\sin(\theta)$ .

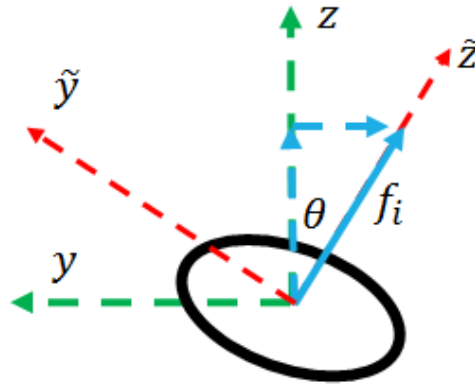


Figure 4-2 Quadrotor propeller orientation when  $\theta \neq 0$ . Inertial frame  $A$  is shown in green, body frame  $B$  is shown in red.

Now suppose a roll command is sent to the drone so that  $\phi \neq 0$ .  $\mathbf{P}_A$  can then be multiplied by the matrix for rotation about the  $\tilde{y}$ -axis to get a new  $\mathbf{P}_A$ .

$$\mathbf{P}_A = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \begin{bmatrix} 0 \\ -F\sin(\theta) \\ F\cos(\theta) \end{bmatrix} = \begin{bmatrix} F\sin(\phi)\cos(\theta) \\ -F\sin(\theta) \\ F\cos(\phi)\cos(\theta) \end{bmatrix} \quad (5)$$

$\mathbf{P}_A$  now represents the propeller forces acting on the drone when a pitch and roll angle are applied. For a more general model,  $\mathbf{P}_A$  could be multiplied by the matrix for rotation about the  $\tilde{z}$ -axis. This would give the propeller forces acting on the drone in  $A$  when  $\psi \neq 0$ . However, a non-zero yaw angle is not required for motion in the  $x$ - $y$  plane, and the drone's  $\psi$  angle can be measured and controlled to stay within  $0 \pm \epsilon$ , where  $\epsilon$  is some small value. Considering rotation about the  $\tilde{z}$ -axis is therefore unnecessary.

The forces acting on the drone can now be summed by adding the gravitational force to the  $z$  component of  $\mathbf{P}_A$ . These sums can then be set equal to quadrotor's mass times its acceleration.

$$\begin{bmatrix} F\sin(\phi)\cos(\theta) \\ -F\sin(\theta) \\ F\cos(\phi)\cos(\theta) - mg \end{bmatrix} = m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} \quad (6)$$

Equation (6) can be solved to yield the following equations for acceleration.

$$\ddot{x} = \left(\frac{F}{m}\right) \sin(\phi)\cos(\theta) \quad (7)$$

$$\ddot{y} = -\left(\frac{F}{m}\right) \sin(\theta) \quad (8)$$

$$\ddot{z} = \left(\frac{F}{m}\right) \cos(\phi)\cos(\theta) - g \quad (9)$$

A drone moving only in the  $x$ - $y$  plane will maintain a constant altitude, and therefore  $\ddot{z} = 0$  can be assumed to be zero. Equation (9) can then be solved for  $F$ .

$$F = \frac{mg}{\cos(\theta)\cos(\phi)} \quad (10)$$

Substituting equation (10) into equations (7) and (8) gives the following for acceleration in the  $x$  and  $y$  directions.

$$\ddot{x} = g * \tan(\phi) \quad (11)$$

$$\ddot{y} = -g * \frac{\tan(\theta)}{\cos(\phi)} \quad (12)$$

The above equations give a simple model of the quadrotor for planar motion ideal for using with the Parrot Minidrone because they do not require the estimation of unknown parameters and take as input pitch and roll angles, which are easily controlled by using the Parrot SDK.

### Simulink Model of the Drone

Equations (11) and (12) were used to create a Simulink model of the drone, as shown in Figure 4-3. The model took as input the pitch and roll angles, which were then converted from degrees to radians. After calculating  $\ddot{x}$  and  $\ddot{y}$ , integrals are taken to get  $\dot{x}$  and  $\dot{y}$ , and then taken again to get  $x$  and  $y$ .

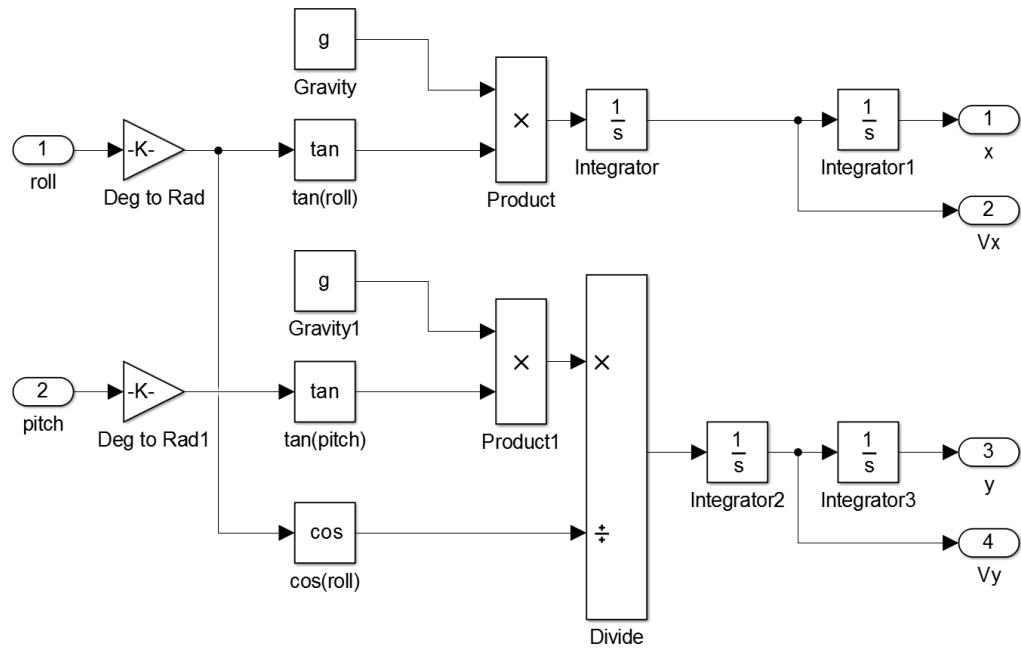


Figure 4-3 Simulink model of the quadrotor

PID controllers were added to the model, as shown in Figure 4-4, to simulate the control of the drone. The PID blocks were set up to include wind-up protection and limit output to values between -50 and 50. The PID parameters were chosen by using the graphical tool provided by Simulink that allows the user to adjust the response time and transient behavior of the controller. The proportional, integral, and derivative parameters used were 0.0714, 0.0110, and 0.1132, respectively. The output of the controller was then multiplied by the maximum angle, which was set to be 5 degrees, and divided by 100 in order to replicate the control provided by the SDK. This resulted in an effective maximum allowable angle of 2.5 degrees.

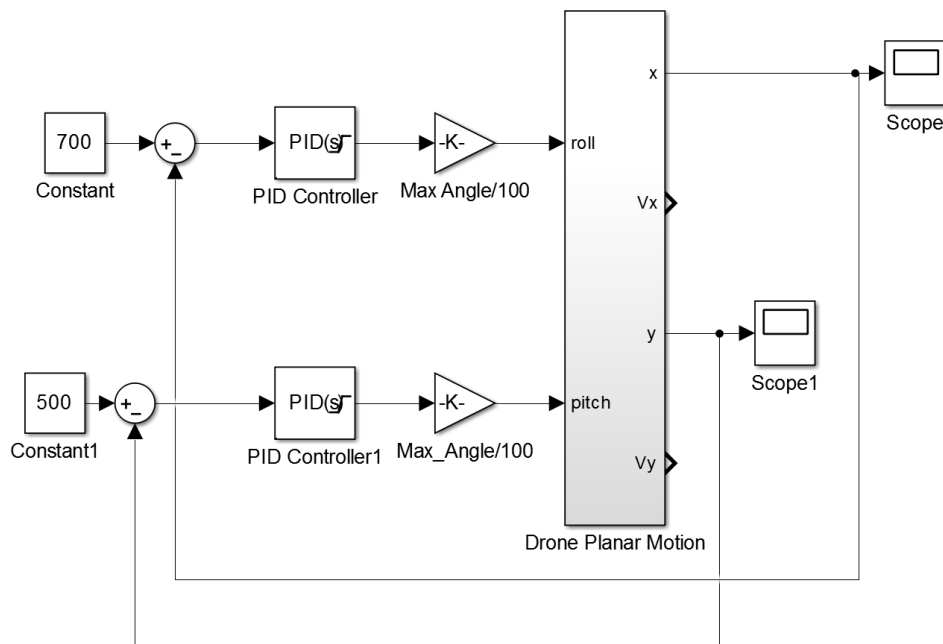


Figure 4-4 Quadrotor model with PID controllers

A simulation was run where the drone travelled from (-700, -500) mm to (700, 500) mm. Figure 4-5 shows the x and y positions versus time for this simulation. For both positions, there is an overshoot of about 300 mm, and the drone arrives and settles at its position in approximately 10 seconds.



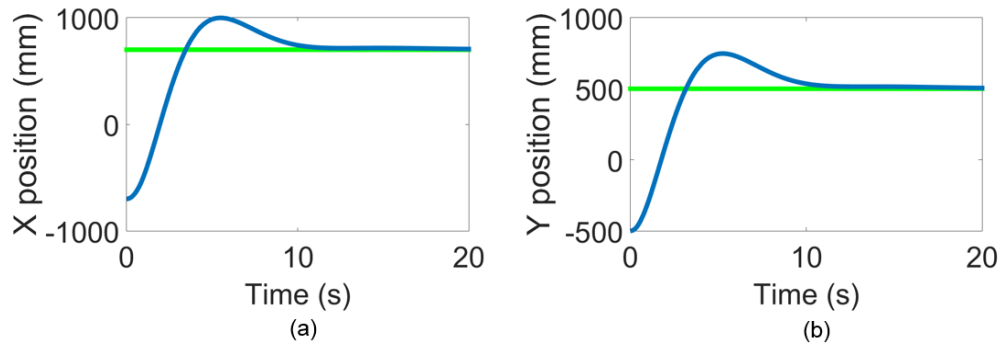


Figure 4-5 Simulated Drone Position Moving from (-700, -500) mm to (700, 500) mm

## Minidrone Control in StarL

### *PID Controller Class*

A PID controller class was written in Java to provide pitch and roll commands to the drone when given the drone's current position. The controller used a window filter of length 8 for the derivative term, and limited the output to between -50 and 50. It also limited the accumulated error value to between -185 and 185. The same PID parameters used for simulation were used to control the minidrones.

### *Minidrone Motion Automaton*

A motion automaton class was written to control the minidrone from within StarL, and pseudocode is shown on the next page. The drone starts in the INIT stage, where the PID controllers are reset by setting all accumulated values to zero. If the drone is landed, control moves to the TAKEOFF stage, otherwise it goes to the MOVE stage. In the TAKEOFF stage, the take-off command is sent to the drone and the landed variable is set to false. Control is then transferred to the MOVE stage.

```

while(true) {
    myPosition = getMyPosition();
    distanceToGoal = getDistanceToGoal();

    switch(stage) {
        case INIT:
            PID_x.reset();
            PID_y.reset();
            if(landed) {
                nextStage = TAKEOFF;
            }
            else {
                nextStage = MOVE;
            }

        case MOVE:
            if(distanceToGoal < goalRadius) {
                nextStage = GOAL;
            }
            else {
                xCmd = PID_x.getCmd(myPosition.x, goal.x);
                yCmd = PID_y.getCmd(myPosition.y, goal.y);
                droneController.send(xCmd);
                droneController.send(yCmd);
                droneController.adjustYaw();
            }

        case HOVER:
            if(distanceToGoal < goalRadius) {
                droneController.send(hover);
            }
            else {
                xCmd = PID_x.getCmd(myPosition.x, goal.x);
                yCmd = PID_y.getCmd(myPosition.y, goal.y);
                droneController.send(xCmd);
                droneController.send(yCmd);
            }
            droneController.adjustYaw();

        case TAKEOFF:
            droneController.send(takeoff);
            landed = false;
            nextStage = MOVE;

        case GOAL:
            nextStage = HOVER;
    }
}

```

In the MOVE stage, if the drone's distance to its goal waypoint is less than the goal radius value, control is transferred to the GOAL stage. Otherwise, the drone's position and goal values are given to the PID controller objects, which return pitch and roll commands. These commands are then sent to the drone, and an `adjustYaw` method is called, which sends a small angular velocity command to the drone if its estimated yaw value is not within a defined range. Control remains with the MOVE stage until the goal condition is met. In the GOAL stage, control is simply transferred to the HOVER stage. The GOAL stage is present to allow future extensions to the motion automaton, such as providing an option for sending a landing command instead of going to the HOVER stage. In the HOVER stage, if the drone is within the goal radius value of its destination waypoint, the drone's hover flag bit is set. Otherwise, control is returned to the PID controller objects to move it back within the goal radius of the waypoint. The `adjustYaw` method is then called, regardless of whether the drone is within the goal radius. When the motion automaton receives a new goal waypoint for the drone, the stage is set to INIT and control starts over again.

#### *Minidrone Control Results*

To test the controller, a simple StarL application was written that moved a drone between waypoints located at (-700, -500) and (700, 500) mm. Figure 4-6 (a) and (b) show the drone's x and y positions, respectively, versus time while moving from (-700, -500) to (700, 500) mm. Note that the drone does not reach its goal y position in Figure 4-6 (b) because it arrives within the goal radius, at which point it is sent to the next waypoint, before actually reaching the current waypoint. Comparing Figure 4-6 (a) and (b) to Figure 4-5 (a) and (b), which shows the simulation for the same movement, there is much less overshoot, but more oscillations are observed. Several factors may account for these differences, including delays between position capture and control commands,

external forces not accounted for by the model such as aerodynamic drag, and non-instantaneous response of the drone to pitch and roll commands. Despite these differences, the PID values provided by the Simulink model do provide adequate control for the minidrones.

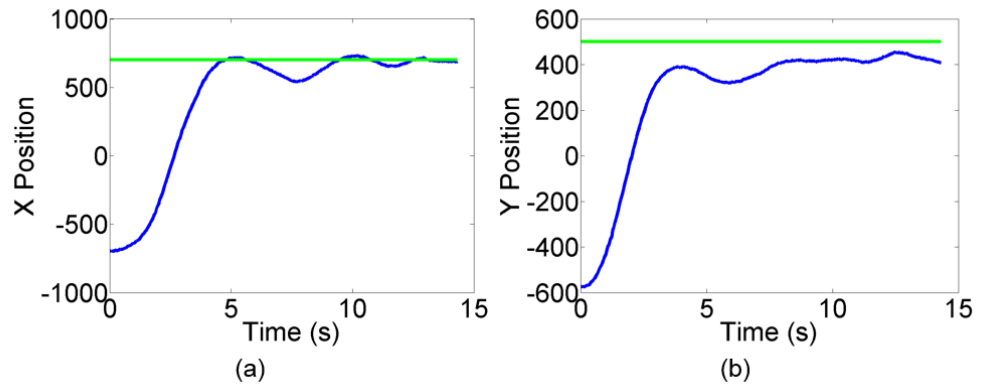


Figure 4-6 (a) x and (b) y positions versus time for a drone being controlled from StarL. Green lines indicate the goal position.

## **Chapter 5**

### **StarL Implementation and Applications**

#### **StarL Implementation**

One of StarL's most useful features is the ability to write distributed robotics applications that can be run in both simulation and hardware with minimal to no changes required to the underlying application code. To run StarL in simulation, the only requirements are a desktop computer, the freely available Android Studio IDE, and the StarL source code. Running in StarL simulation is advantageous because it allows users who may not have the resources to purchase hardware to develop distributed applications, and it allows researchers to scale up to larger number of robots than a typical laboratory setting may allow. Despite these advantages, running StarL in hardware is still desirable, as issues not observed in simulation sometimes occur in hardware and therefore would never be discovered without deployment to real robots. Additionally, running StarL applications in hardware showcases the framework's capabilities in ways a simulation simply cannot, like in the light painting application where robots coordinate to draw a pattern using long exposure photography and Android phones [42].

To be implemented in hardware, StarL requires a localization system capable of estimating robot positions in a real-world coordinate frame. In previous studies an OptiTrack system, which tracks reflective markers placed on the robots using infrared light, has been used, but in this study the system described in Chapter 3 was utilized. StarL applications were run on Google Nexus 7 tablets, which received broadcast UDP packets containing the estimated robot positions from the desktop running the localization script. Packets were sent over an ad-hoc wireless network created with a 300 Mbps TP-Link wireless N router. Each tablet was paired to either an iRobot Create2 or a Parrot

Minidrone, and sent movements commands to the robots using Bluetooth communication.

### Follow App

Two new StarL applications were developed in this study. The first, named Follow App, is a simple application that was created to demonstrate the capabilities of the video localization system described in Chapter 3 and the drone controller described in Chapter 4. In Follow App, a sequence of waypoints is provided to the application, where the number of waypoints must be greater than or equal to the number of robots. The application starts by having each robot move to a unique waypoint in the sequence. Each robot waits at its destination until all other robots arrive at their waypoints. Then all robots move to the next waypoint in the sequence, again waiting until all robots have reached their destinations before moving again. This process then repeats indefinitely.

Pseudocode for Follow App is shown below.

```
int myID, myIndex, msgCount = 0, numBots = N;
boolean arrived;
List<Position> destinations;
Position myDest;
myIndex = myID;
private enum Stage {PICK, GO, WAIT};
stage = PICK;
while(1)
    switch(stage)
        case PICK:
            arrived = false;
            myDest = destinations[myIndex];
            myIndex++;
            moveTo(myDest);
            stage = GO;
        case GO: if at myDest then
            send(msg: Arrived, to: All);
            arrived = true;
            stage = WAIT;
        case WAIT: if msgCount >= numBots - 1 && arrived then
            msgCount = 0;
            stage = PICK;

when msg received: msgCount++;
```

Notice that Follow App is structured as a state machine, with control moving between stages as certain conditions are met. All StarL applications have a similar state machine structure, and developers are free to create as many stages as necessary to implement their distributed application.

In Follow App, each robot has its own unique integer ID in the range 0 to  $N-1$ , where  $N$  is the total number of robots being used. Each instance of the application has access to its own copy of the sequence of waypoints, which is stored in the destinations list. Waypoints are selected from this list using the *myIndex* variable, which is initially set to the robot's ID number. This ensures each robot initially travels to a unique waypoint as long as the number of waypoints is greater than  $N$ . After setting the *myIndex* variable, the program enters the state machine while loop in the PICK stage. In PICK, a Boolean variable named *arrived* is set to false, a destination is selected from the waypoint sequence using the *myIndex* variable, and the *myIndex* variable is incremented. Next, the selected destination is passed to the `moveTo` function. This function is an example of a StarL primitive, in this case handling point to point planar motion. Finally, before exiting the PICK stage, the stage is set to GO.

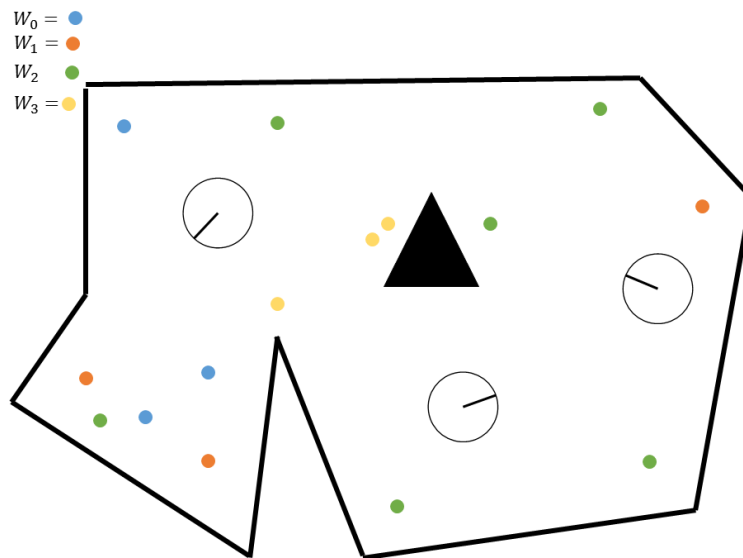
In the GO stage, an if statement checks if the robot has reached its destination. If it has not, no code is executed and control simply returns to the GO stage in the next loop iteration. If the destination has been reached, a message is sent to all other robots indicating the arrival. Message passing is an important aspect of distributed robotics applications, and StarL makes it easy for developers to send messages with customized content. However, in this application no custom content is needed since the only purpose of the messages is to indicate arrival at a destination. After the message is sent, the *arrived* variable is set to true, and the stage is set to WAIT.

In the WAIT stage, an if statement checks to see if the *msgCount* variable is greater than or equal to  $N - 1$ , and if the robot has arrived at its destination. As its name suggest, the *msgCount* variable is simply a count of the number of messages that have been received from other robots, i.e., how many other robots have arrived at their destinations. The *msgCount* variable is incremented in a receive message function, which is called every time the robot gets a new message. If these conditions are not met, nothing is executed and control returns to WAIT in the next iteration. If they are, then all robots have arrived at their destinations and are ready to move to the next waypoint in the sequence. The *msgCount* variable is reset to 0, and the stage is set to PICK, so the robots can select and move to new destinations.

### **Modified Distributed Travelling Salesman App**

The travelling salesman problem is a classic theoretical computer science problem where given a list of cities and the distances between them, the shortest path visiting all cities exactly once and returning to the original city must be found. An interesting variation of this problem for distributed robotics systems can be stated as follows.  $K$  sets of waypoints,  $W = W_0, W_1, \dots, W_{k-1}$  must be visited in order by  $N$  robots. Each robot must visit at least one waypoint in each set. Assume that for each waypoint set  $W_i$ , the number of waypoints belonging to the set is greater than or equal to  $N$ . It is allowable for a robot to pass through a waypoint in a set that is not currently being visited, but the waypoint should not be counted as visited.





*Figure 5-1 Illustration of the modified distributed travelling salesman problem*

This problem has many potential applications, including search and rescue missions in large multi-floor buildings and systematic aerial mapping of large areas. As stated above, the problem has no optimization requirements like the traveling salesman problem, but several opportunities for optimization are available including minimizing the time it takes for the robots to visit all waypoints, minimizing the total distance travelled by all robots, and minimizing or eliminating collisions with environmental obstacles and/or other robots.

An important complication that arises in this problem is the possibility for deadlocks. A simple example of a possible deadlock would be one robot arriving at a waypoint and then waiting for a signal to move to a new point once all other robots have reached their destinations. If this robot is situated in a way that makes it physically impossible for another robot to reach its destination, then a deadlock situation will occur where the stationary robot will not move until the travelling robot reaches its destination, and the travelling robot will not reach its destination until the stationary robot moves.

More complicated deadlock situations could be encountered in challenging environments. For example, it may be necessary for one robot to leave a small or narrow space before another robot enters to avoid a potential livelock situation where one robot is trying to leave an area while another is trying to enter, and neither makes any progress.

Given that the travelling salesman problem is NP-hard and adding distributed elements such as the possibility of deadlocks further complicates the problem, obtaining a solution that optimizes the time taken or distance travelled is difficult when the number of waypoints is not small. A naïve solution is presented below that does not aim to optimize the time or distance travelled, but does attempt to avoid deadlock situations even in challenging environments. The overall strategy of this approach is to have one robot move towards a waypoint contained in the first waypoint set, while all other robots move to random positions within the environment. Given enough time, this strategy should prevent deadlocks in most environments, because if one robot is in another's way, it will eventually move. After visiting a waypoint, the robot will begin to move to random positions, and the next robot in the robot sequence will go to a goal waypoint. If there are more waypoints in a set than robots, the final robot in the sequence of robots visits all remaining waypoints. When all points in a set have been visited, the entire process is repeated with the next set until no more sets remain.

Figure 5-2 shows the state machine diagram for the Modified Distributed Travelling Salesman (MDTS) app. The variables used in Figure 5-2 are defined below.

*sets* – array containing the sets of waypoints to be visited

*dest* – an array containing the set of waypoints currently being visited

*index* – the array index value of the set currently being visited

*numSets* – number of waypoint sets to be visited

*N* – total number of robots

*myID* – unique integer ID of the robot in range 0 to  $N - 1$

*goRand* – Boolean variable that is true when the robot is going to a random location

*myDest* – robot's destination waypoint (random or goal waypoint)

*obs* – list of environmental obstacles (static and other robots)

*path* – stack containing a sequence of points to be visited, with *myDest* as the final point

*midDest* – point from *path* the robot is currently moving towards

*fromID* – the *myID* of the robot a message was received from

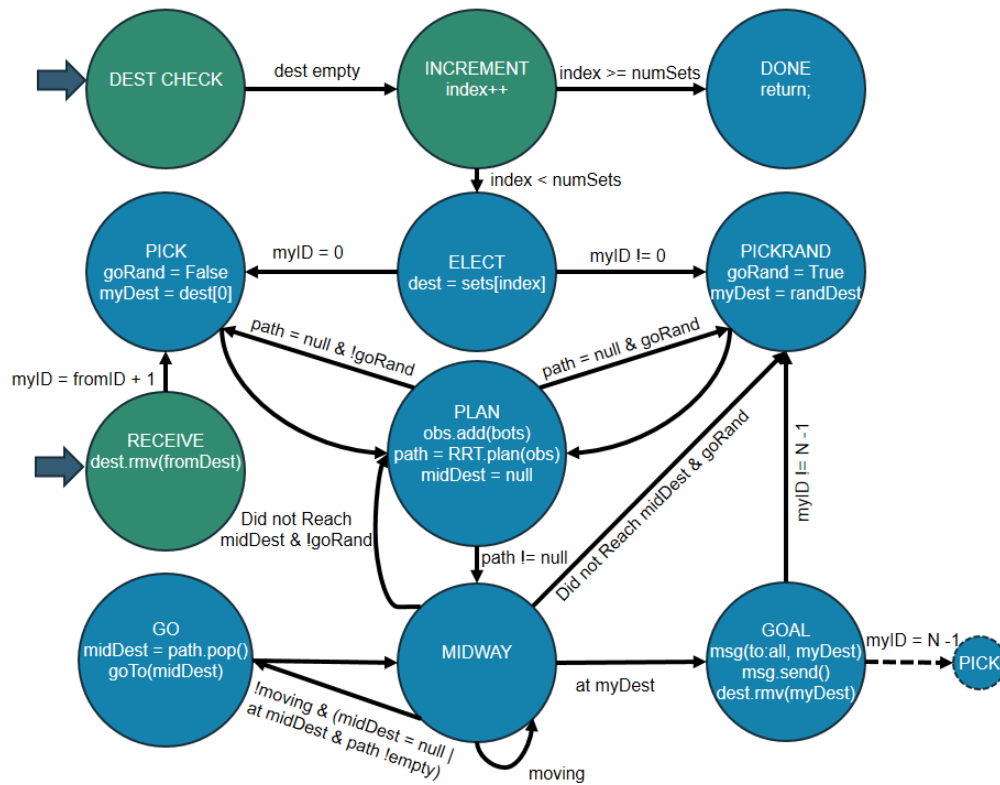


Figure 5-2 State machine diagram for the MDTs App

Similar to Follow App, the MDTs App is implemented using an infinite while loop, with control changing between stages as conditions are met. The blue circles in Figure 5-2 represent stages in a switch-case statement, while the green circles represent code executed outside one of these stages. The DEST CHECK circle represent a check that is

performed at the beginning of each while loop to determine if the *dest* array is empty. If *dest* is empty, this means all waypoints in that set have been visited, and the *index* variable is incremented in the INCREMENT circle. If *index* is greater than or equal to *numSets*, then all waypoint sets have been visited and control is transferred to the DONE stage, which returns from the while loop and ends execution. Otherwise, *dest* becomes *sets[index]* and control transfers to the ELECT stage.

To the start the program, *index* is set to 0, *dest* is set to *sets[0]*, and the stage is set to ELECT. In the ELECT stage, a simple leader election is implemented by selecting the bot with *myID* equal to 0. This robot is sent to the PICK stage, where it will select the first waypoint from *dest* as its destination and set the *goRand* variable to false. All other robots are sent to the PICKRAND stage, where they select a random destination within the environment, and set the *goRand* variable to true. From both PICK and PICKRAND, control is sent to the PLAN stage.

In the PLAN stage, an attempt to find a path to the robot's destination is performed using rapidly exploring random tree (RRT) path planning. RRT path planning is an algorithm capable of planning paths around objects and is widely used in robotic applications. Before planning the path, all other robots are added as obstacles to the obstacle list. StarL obstacles have the option of setting a time frame variable, where the object will delete itself after a certain amount of time has passed. This is set to a relatively short period of time for the robot obstacles since they are constantly moving and therefore not in the same place for long periods of time. After adding the robots as obstacles, an `RRT.plan()` method is called and *path* is returned. If *path* is not null, this indicates that a route was found and *path* contains a sequence of positions ending with the robot's destination *myDest*. The *midDest* variable is set to null to indicate motion towards the first waypoint in the stack has not started, and the stage is set to MIDWAY. If

*path* is null, this means a route was not found. If the robot was not going to a random destination, it returns to the PLAN stage. In this case it is likely another robot is blocking the waypoint it is trying to reach. New paths will be planned until the other robot moves out of the way and a valid path is found. If a route was not found and the robot was going to a random destination, control goes back to PICKRAND, and another random destination is selected.

The purpose of the MIDWAY stage is to send the robot to each destination in *path*. First a check is performed to see if the robot is moving. If it is moving, this means the robot is headed towards *midDest*, and control returns to the MIDWAY stage. If the robot is not moving, there are several possible scenarios. The first is that motion towards *midDest* has not been started. In this case, *midDest* will be null and control is transferred to the GO stage. In the GO stage *path* is popped, and *midDest* is set to the value returned. Then the `goTo` method is called to send the robot to *midDest*.

The next two possibilities both occur when the robot has arrived at *midDest*. In one case *path* will not be empty, which means there are more points to be visited. Control is then sent to the GO stage, so the robot can move to the next point in *path*. In the second case, *midDest* and *myDest* are the same waypoint. This means that *path* will be empty, and the robot has arrived at its destination. When this occurs, control is transferred to the GOAL stage.

Before describing the GOAL stage, the final scenario in which the robot is in the MIDWAY stage and is not moving is considered. In this scenario, the robot collided with an obstacle or another robot before reaching *midDest*. In this case, *midDest* will not be null since motion has started towards it, and the robot will not be at *midDest*. If the robot was going to a random destination, it is sent back to PICKRAND so another destination

can be chosen. If it is going to a goal waypoint, it is sent back to PLAN, so a new *path* can be found.

When the robot reaches a goal waypoint from one of the waypoint sets, it enters the GOAL stage. In this stage, it first sends a message to all other robots with *myDest* as the contents, and then removes *myDest* from *dest*. If the robot's *myID* number is not equal to  $N - 1$  (i.e. it is not the robot with the largest *myID*) it is sent to PICKRAND so it can begin moving to random destination. If *myID* is equal to  $N - 1$  the robot is sent to PICK. At this point all robots have visited at least one waypoint in the set of waypoints currently being visited. The robot with highest *myID* will then visit all waypoints remaining in the set while all other robots move to random destinations.

The green RECEIVE circle in Figure 5-2 represents the code executed when a message is received by a robot. All messages in the MDTs App indicate that a robot has reached a goal waypoint. Upon receipt of a message, the robot will remove the destination sent with the message from its *dest* variable. Then, if the robot's *myID* is equal to *fromID* + 1, it is sent to PICK. This ensures that the robots visit the waypoints from each set in the order of their *myID* numbers.

After all points in any set  $W_i$  have been visited, the *dest* variable will be empty and *index* will be incremented. All robots will be sent to the ELECT stage, and the entire process will start over with the new waypoint set.

## Chapter 6

### Conclusion and Future Work

In this work a new video localization system for robots was described, a model and controller for planar motion of a quadcopter was developed, and two new StarL applications were presented. The video localization employed a Microsoft Kinect Version 1, which provides standard RGB and depth images. The system is capable of tracking circular robots (or robots with circular shapes attached) as they move within the camera's field of view. The localization script was written in Matlab and used the built-in `imfindcircles` function to detect the circular robots. The system successfully tracked iRobot Create2s, Parrot Cargo Minidrones, and Parrot ARDrones with a precision of approximately 0.5 centimeters. Robot yaw angles were also estimated by the system using color detection with a precision of about 0.85 degrees (based on a 95% confidence interval). Compared to commonly used multi-camera optical tracking systems, this system is significantly less expensive and does not require repeated calibration.

A model for planar motion of a quadcopter was developed by considering the forces acting on the drone and applying rotation matrices to find the force components in the  $x$  and  $y$  directions of an inertial coordinate frame. By assuming no vertical acceleration of the quadcopter and no rotation about the quadcopter's  $\hat{z}$  axis (i.e.  $\psi = 0$ ) simple equations for acceleration in the  $x$  and  $y$  directions were derived. These equations we used to model the drone in Simulink, where integrals were taken to find  $x$  and  $y$  velocities and positions. PID controllers were added to the Simulink model and tuned using the available graphical tools. The PID parameters provided by the model were used with a PID controller class written in Java and integrated into the StarL framework to successfully move Parrot Cargo Minidrones to waypoints from within StarL applications.

Two new StarL applications were presented. The first was Follow App, which is a simple application where robots begin by moving to unique positions from a waypoint sequence. Upon arrival, the robots wait until all other robots have reached their respective destinations, and then move to the next waypoint in the sequence. This application was used to demonstrate the capabilities of the localization system and control of the Parrot Minidrone, and provides an easy to understand example for how StarL applications are structured.

The second application was the MDTTS App, which aims to solve a problem where robots must visit several sets of waypoints in order, with each robot visiting at least one waypoint in each set. To solve the problem, the robots are assigned unique integer ID numbers and visit the points in each waypoint set in order of these IDs. While one robot is moving towards a goal waypoint, all other robots move to random locations within the environment. This is done to avoid deadlock situations where one robot is in the way of another. Since robots are always moving to random locations, if a robot happens to be occluding a waypoint or path of a robot moving to a goal point, it will eventually move out of the way.

There are several ways the work described in this thesis could be extended in the future. For the video localization system, more cameras could be added to the ceiling to extend the allowable area for robot motion. This would enable the use of more robots, and the extra space may allow for more interesting distributed algorithms to be tested in hardware. Another possible extension is to use Kinect Version 2 cameras, which provides a higher resolution. When this project started, Matlab did not provide support for the Kinect Version 2, but now does in the 2016a release. The higher resolution should provide increased accuracy and precision, but the extra pixels will slow down the circle detection function, so this tradeoff would need to be considered. Another extension that



will be important if larger numbers of robots are used is increasing the system's sampling rate. One possibility is to write a circle detection function that takes advantage of the powerful GPU available in the lab. Another way speed might be improved is to integrate other types of video localization techniques which do not require as much computation time such as frame difference or color detection.

An obvious extension for the quadcopter model is to include acceleration in the vertical direction. This would allow the quadcopter to move to three-dimensional waypoints, which would provide the opportunity to test more types of distributed algorithms. Vertical acceleration commands are not accepted by the Parrot quadcopters (vertical speed commands are sent instead) like the pitch and roll angles are, so this makes estimating and controlling vertical acceleration more difficult. It would also complicate the acceleration equations in the  $x$  and  $y$  directions, as the gravitational force and vertical component of the propeller forces do not cancel each other during vertical acceleration. Despite the challenges, developing this model would allow quadrotor control to be simulated, and make it easier to tune a controller for three-dimensional motion.

Extensions to StarL include the development of more applications to test and demonstrate novel distributed algorithms, both in simulation and hardware. For the MDTS App, a solution that aims to optimize the total distance travelled or the time taken for the robots to visit all waypoints could be developed. Approaches that could be considered would include greedy strategies, where robots are selected based on their distances from points currently being visited, and possibly more sophisticated strategies that plan farther into the future. Regardless of the approach, care must be taken to ensure that deadlocks cannot occur in the environments in which the robots operate.

## References

- [1] Parrot for developers. <https://github.com/Parrot-Developers>.
- [2] T.J. Atherton and D.J. Kerbyson. Size invariant circle detection. *Image and Vision Computing*, 17(11):795 – 803, 1999.
- [3] Tim Bailey and H. Durrant-Whyte. Simultaneous localization and mapping (slam): part ii. *Robotics Automation Magazine, IEEE*, 13(3):108–117, 2006.
- [4] M. Betke and L. Gurvits. Mobile robot localization using landmarks. *Robotics and Automation, IEEE Transactions on*, 13(2):251–263, April 1997.
- [5] Leonardo Bobadilla, Taylor T. Johnson, and Amy LaViers. Verified planar formation control algorithms by composition of primitives. In *AIAA SciTech*, Kissimmee, Florida, January 2015. AIAA.
- [6] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.
- [7] G. Carrera, A. Angeli, and A.J. Davison. SLAM-based automatic extrinsic calibration of a multi-camera rig. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2652–2659, May 2011.
- [8] Melvin Cohen and Godfried T Toussaint. On the detection of structures in noisy pictures. *Pattern Recognition*, 9(2):95–98, 1977.
- [9] Rajesh N Dave. Fuzzy shell-clustering and applications to circle detection in digital images. *International Journal Of General Systems*, 16(4):343–355, 1990.
- [10] M.W.M.G. Dissanayake, P. Newman, S. Clark, H.F. Durrant-Whyte, and M. Csorba. A solution to the simultaneous localization and map building (slam) problem. *Robotics and Automation, IEEE Transactions on*, 17(3):229–241, June 2001.

- [11] Parasara Sridhar Duggirala, Taylor T. Johnson, Adam Zimmerman, and Sayan Mitra. Static and dynamic analysis of timed distributed traces. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS 2012)*, San Juan, Puerto Rico, December 2012.
- [12] H. Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *Robotics Automation Magazine, IEEE*, 13(2):99–110, June 2006.
- [13] Dieter Fox, Wolfram Burgard, Hannes Kruppa, and Sebastian Thrun. A probabilistic approach to collaborative multi-robot localization. *Autonomous Robots*, 8(3):325–344, 2000.
- [14] Martin von Gagern. Direct way of computing clockwise angle between 2 vectors. <http://stackoverflow.com/questions/14066933/direct-way-of-computing-clockwise-angle-between-2-vectors>, 2013. [Online; accessed 05-July-2015].
- [15] Gabriel M Hoffmann, Haomiao Huang, Steven L Waslander, and Claire J Tomlin. Quadrotor helicopter flight dynamics and control: Theory and experiment. 2007.
- [16] Seungho Jeong and Seul Jung. Vision-based localization of a quad-rotor system. In *Ubiquitous Robots and Ambient Intelligence (URAI), 2012 9th International Conference on*, pages 636–638, November 2012.
- [17] Taylor T. Johnson. Fault-tolerant distributed cyber-physical systems: Two case studies. Master’s thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, May 2010.
- [18] Taylor T. Johnson. *Uniform Verification of Safety for Parameterized Networks of Hybrid Automata*. PhD thesis, University of Illinois at Urbana-Champaign, Electrical and Computer Engineering, Urbana, IL 61801, 2013.
- [19] Taylor T. Johnson and Sayan Mitra. Safe flocking in spite of actuator faults. In Shlomi Dolev, Jorge Cobb, Michael Fischer, and Moti Yung, editors, *12th International*

*Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2010)*, volume 6366 of *Lecture Notes in Computer Science*, pages 588–602. Springer Berlin / Heidelberg, September 2010.

[20] Taylor T. Johnson and Sayan Mitra. Safe flocking in spite of actuator faults using directional failure detectors. *Journal of Nonlinear Systems and Applications*, 2(1-2):73–95, April 2011.

[21] Taylor T. Johnson and Sayan Mitra. Parameterized verification of distributed cyber-physical systems: An aircraft landing protocol case study. In *ACM/IEEE 3rd International Conference on Cyber-Physical Systems*, April 2012.

[22] Taylor T. Johnson and Sayan Mitra. A small model theorem for rectangular hybrid automata networks. In *Proceedings of the IFIP International Conference on Formal Techniques for Distributed Systems, Joint 14th Formal Methods for Open Object-Based Distributed Systems and 32nd Formal Techniques for Networked and Distributed Systems (FMOODS-FORTE)*, volume 7273 of *LNCS*. Springer, June 2012.

[23] Taylor T. Johnson and Sayan Mitra. Invariant synthesis for verification of parameterized cyber-physical systems with applications to aerospace systems. In *Proceedings of the AIAA Infotech at Aerospace Conference (AIAA Infotech 2013)*, Boston, MA, August 2013.

[24] Taylor T. Johnson and Sayan Mitra. Anonymized reachability of rectangular hybrid automata networks. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*, 2014.

[25] Taylor T. Johnson and Sayan Mitra. Safe and stabilizing distributed multi-path cellular flows. *Theoretical Computer Science*, 579:9 – 32, 2015.

[26] Taylor T. Johnson, Sayan Mitra, and Cedric Langbort. Stability of digitally interconnected linear systems. In *Proceedings of the 50th IEEE Conference on Decision*

and Control and European Control Conference (CDC ECC 2011), pages 2687–2692, Orlando, Florida, USA, December 2011.

[27] Taylor T. Johnson, Sayan Mitra, and Karthik Manamcheri. Safe and stabilizing distributed cellular flows. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 577–586, Genoa, Italy, June 2010. IEEE.

[28] Vladimir Kolmogorov and Ramin Zabih. Multi-camera scene reconstruction via graph cuts. In Anders Heyden, Gunnar Sparr, Mads Nielsen, and Peter Johansen, editors, *Computer Vision (ECCV)*, volume 2352 of *Lecture Notes in Computer Science*, pages 82–96. Springer Berlin Heidelberg, 2002.

[29] J.J. Leonard and H.F. Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *Robotics and Automation, IEEE Transactions on*, 7(3):376–382, June 1991.

[30] Liang-Fu Li, Zu-Ren Feng, and Qin-Ke Peng. Detection and model analysis of circular feature for robot vision. In *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, volume 6, pages 3943–3948 vol.6, Aug 2004.

[31] Yixiao Lin and Sayan Mitra. Starl: Towards a unified framework for programming, simulating and verifying distributed robotic systems. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM, LCTES’15*, pages 9:1–9:10, New York, NY, USA, 2015. ACM.

[32] R. Mahony, V. Kumar, and P. Corke. Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor. *Robotics Automation Magazine, IEEE*, 19(3):20–32, 2012.

[33] J. McLurkin, A. McMullen, N. Robbins, G. Habibi, A. Becker, A. Chou, Hao Li, M. John, N. Okeke, J. Rykowski, S. Kim, W. Xie, T. Vaughn, Yu Zhou, J. Shen, N. Chen,

- Q. Kaseman, L. Langford, J. Hunt, A. Boone, and K. Koch. A robot system design for low-cost multi-robot manipulation. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 912–918, 2014.
- [34] Daniel Mellinger, Nathan Michael, and Vijay Kumar. Trajectory generation and control for precise aggressive maneuvers with quadrotors. *The International Journal of Robotics Research*, page 0278364911434236, 2012.
- [35] M. Rubenstein, C. Ahler, and R. Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3293–3298, May 2012.
- [36] J. Ryde and Huosheng Hu. Fast circular landmark detection for cooperative localisation and mapping. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2745–2750, April 2005.
- [37] Erol Sahin. *Swarm Robotics: SAB 2004 International Workshop, Santa Monica, CA, USA, July 17, 2004, Revised Selected Papers*, chapter Swarm Robotics: From Sources of Inspiration to Domains of Application, pages 10–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
- [38] A. L. Salih, M. Moghavvemi, H. A. F. Mohamed, and K. S. Gaeid. Modelling and pid controller design for a quadrotor unmanned air vehicle. In *Automation Quality and Testing Robotics (AQTR), 2010 IEEE International Conference on*, volume 1, pages 1–5, May 2010.
- [39] Stephen Se, David Lowe, and Jim Little. Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks. *The International Journal of Robotics Research*, 21(8):735–758, 2002.

- [40] Y. Sugiyama, A. Shiotsu, M. Yamanaka, and S. Hirai. Circular/spherical robots for crawling and jumping. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 3595–3600, April 2005.
- [41] HK Yuen, J Princen, J Illingworth, and J Kittler. Comparative study of hough transform methods for circle finding. *Image and Vision Computing*, 8(1):71–77, 1990.
- [42] Adam Zimmerman. StarL for programming reliable robotic networks. Master’s thesis, University of Illinois at Urbana-Champaign, 2012.

## **Biographical Information**

Nathan Hervey earned his Bachelors of Science in Mechanical Engineering from Texas A&M University in May of 2009. In the fall of 2011 he began his studies in bioengineering at the University of Texas at Arlington with Dr. George Alexandrakis as his advisor. His research focused on new analysis technique for functional near-infrared spectroscopy data. He obtained his master's degree in bioengineering in May 2014, and then began work on a master's degree in computer science under the advisement of Dr. Taylor Johnson. He received this degree in May 2016, and is excited to start work as a software developer in industry.