THE UNIVERSITY of ADELAIDE

Multi-Drone Tracking and Formation Control Platform

Name
Alexander Woolfall

Department
Electrical and Electronic Engineering

Platform Contributor
Jacob Stollznow

Supervisors
Dr. Hong Gunn Chew Dr. Duong Doc Nguyen

Date of Submission
5th Jun 2020

Abstract

Unmanned Aerial Vehicle (UAV) swarming capabilities and formation control platforms are by no means a novel concept. Many of these systems focus on overcoming the complications of flying a subset of drone types in outdoor environments making use of custom on-board processing to accomplish the drone's control tasks.

This thesis presents a new UAV formation control platform that aims to alleviate the complications of flying many different drones types simultaneously in indoor environments. The developed platform is capable of robust, stable flight of a diverse set of drones through a centralised control architecture all controlled via a common programming interface. The platform incorporates in-built collision avoidance measures to ensure the safe flight of physical drones within the system and makes use of Optitrack motion capture for external positioning.

Additionally this thesis describes the introduction of a point set registration component to convert point cloud output from Optitrack into rigidbody objects representing drones within the platform. This component makes use of unit quaternions and the Iterative Closest Point (ICP) algorithm alongside application assumptions to decrease sources of instability within the platform.

Contents

List of Figures

Acronyms

API Application programming Interface.

GUI Graphical User Interface.

ICP Iterative Closest Point.

ROS Robot Operating System.

SVD Singular Value Decomposition.

UAV Unmanned Aerial Vehicle.

VRPN Virtual-Reality Peripheral Network.

1 Introduction

The controlled flight of UAV drones in indoor environments introduce a range of issues seen to a lesser degree in outdoor operation. The safe flight of a set of drones within a confined space require constant and alert observation by an operator and strict flight pattern protocols for the drones. In the aim of protecting the often expensive lab equipment and safety of those within the area researchers must dedicate considerable efforts to ensuring safe flight patterns.

The vast diversity of drone types also provoke additional time consuming but often menial tasks for operators attempting to implement their designs. Each drone type generally requires the use of their own control scheme with unique yet impactful caveats in design and operation. These caveats are substantial enough to absorb considerable efforts simply in writing the implementation of an algorithm or UAV program and often results in a design tailored specifically for the flight of that one drone type.

1.0.1 Motivations

Researcher needs are the primary motivation for the development of a multi-drone control platform.

Researchers require the need for extensive yet manageable logging and visualisation feedback as a method of deconstructing the actions their programs are undertaking.

Secondary motivation for the development of this platform comes from the desires of operators during UAV demonstrations and program testing. These desires include protecting the integrity of physical hardware within the system through the implementation of a collision avoidance system, and the reduction in preparation and setup times before flights.

A unified drone control platform that allows for the simultaneous flight of many types of drones, whilst also boasting integrated collision avoidance and a focus on reducing preparation costs.

Researchers are unable to make the platform themselves as the development time is considerable and its development time does not directly relate to progress in their own research projects. The combination of these factors form the motivation for undergoing the development of this multi-drone formation and control platform.

1.0.2 Significance

The platform holds significance in improving the experience for demonstrators and researchers in implementing their programs or algorithms onto physical drones. With the ability to seamlessly transfer programs between drone types with little to no additional efforts. The integrated safeguarding and drone collision avoidance will also ensure that all applications respect safety concerns without the need for additional efforts required by an end user. The final aim of this platform that holds significance for an end user is the focus on vastly decreasing preparation and setup times required to fly.

1.0.3 Platform Objectives

The core platform objectives are as follows split into the categories of required objectives and those that are considered to be extension.

Required:

1. The seamless integration of more than one UAV drone type.
2. A generalised and robust API capable of the control the above drone types as a swarm formation or as individual drones. Additionally it is imperative to researchers that the user facing API must support MathWorks Matlab.
3. The platform must contain in-built drone safeguarding and collision avoidance measures in order to reduce the risks associated with the flying of physical drones.
4. Additionally the platform must feature researcher focused capabilities such as extensive feedback and visualisation systems with the intuitive control of drones.

Extension:

1. The user API will be extended to support more languages than just Matlab such as Python and C++.
2. To further reduce setup and preparation times, and reduce initialisation complexity in comparison to current UAV drone swarm platforms such as Crazyswarm.

1.0.4 Platform Implementation Contributions

Although both platform contributors contributed in at least minor cases to all elements of the platform, many platform sections can be identified as having one major contributor. These contributions are displayed in the following table:

Contributions Table	
Platform Section	**Major Contributor/s**
User API	A.Woolfall
Drone Management Server	A.Woolfall
Debugging GUIs and Logging	J.Stollznow
Post-Flight Analysis	J.Stollznow
Safeguarding Efforts	J.Stollznow
Point Set Registration of Rigidbodies	A.Woolfall
Teleoperation	J.Stollznow
Testing Scripts	J.Stollznow, A.Woolfall
Rigidbody Class and Crazyflie Wrapper	J.Stollznow, A.Woolfall
Drone Wrapper Framework	A.Woolfall
Virtual Drone Wrapper (Vflie)	A.Woolfall
Documentation	J.Stollznow, A.Woolfall

2 Multi-Drone Platform

2.1 Background

2.1.1 Robot Operating System

Robot Operating System (ROS) is a robotics focused process communication framework designed for operation on Linux machines with experimental support for Mac OS. ROS's use in this platform is primarily for it's ability to simplify the communications between any number of separately running processes on one machine and it's communication capabilities across networks [1].

The ROS framework facilitates the communications between ROS 'node' processes around a centralised ROS core process. These ROS nodes communicate between each other through two primary methods, the first is a publisher and subscriber communication, and the second a service server/client method [2]. The publisher/subscriber method allows a node to advertise that it will communicate across a ROS 'topic' analogous to a radio station identifying itself with a selected radio frequency. From this point the ROS topic is able to push messages to this selected topic for any other node to read from as a subscriber. The second method of communication, the service server/client model, allows a ROS node to act as a server which accepts and returns from requests sent by client nodes [1]. This method is analogous to a web page accepting and responding to requests sent by web browsers. Our extensive usage of the publisher and subscriber model along with the functionality of ROS 'command queues' gives our system the ability to process communications to drones in an asynchronous, more efficient parallel processing.

Along side the above functionality ROS provides a basis for which a large selection of pre-built processing modules are provided. Such functionality included as a ROS package that this platform uses extensively is the

Figure 1: Images of the Optitrack motion capture system used during the development of this platform

vrpn_client_ros package that further simplifies communications with a data streaming source Virtual-Reality Peripheral Network (VRPN), and crazyflie_ros which simplifies communication with our development drone the Bitcraze Crazyflie. Both of these systems and their alternatives will be described in a later section. The final major functionality provided by ROS that is used for this platform is it's ability to facilitate a form of data visualisation and logging through the inbuilt ROS 'rviz' and logging systems [1].

2.1.2 Optitrack

Optitrack is a motion capture system operating alongside the motion capture software 'Motive'. Optitrack tracks the positions of reflective balls to produce 3D frame by frame point clouds at an update frequency of up to 120Hz. Through the use of a number of cameras, the system is capable of tracking specially made reflective markers accurately to within 0.3mm [3]. Additionally Optitrack features the ability to assign rigidbodies and subsequently track both the position and orientation of these bodies to varying degrees of success, more

on this is discussed in point set registration sections of this thesis [4]. Similar technologies to Optitrack include VICON and Qualisys motion capture systems.

2.1.3 Motion Capture Streaming Options

VRPN is a device-independant communications system which in our case allows for the easy communication of motion capture rigid bodies. VRPN allows the communication of time stamped rigid body poses, that being both position and orientation over a client/server network [5].

Our initial decision to operate using VRPN over the Optitrack native streaming service Natnet was its ease of use, the aforementioned functionality, and it's independence from the Optitrack environment as it is able to operate using a competitor product such as VICON. Optitrack's native data streaming system, Natnet, provides many more features when interacting with the Optitrack system. Natnet supports streaming of not only rigid body objects but marker point clouds as well. Additionally Natnet features the ability to control Motive functionality such as initialising recordings and querying rigid body properties [4].

Moving forward with the platform, NatNet has become the primary source of motion tracking information due to its ability to stream point cloud information for use in custom point set registration of platform rigidbodies.

2.1.4 UAV drones used during development

The two commercial drones looked at during the development of the platform are the Bitcraze Crazyflie 2.0 and the Ryze Tello EDU.

The crazyflie is an open source, light weight drone that has common use with researchers due to its open ended design. The platform is capable of considerable expansion using any number of custom built extensions from on

Figure 2: An image of the Bitcraze Crazyflie 2.0

drone cameras to custom built lighting rigs [6]. Due to it's accessibility and the ease at which it can be integrated, this will be the primary UAV drone used during the development of the platform. The crazyflie is controlled via a radio dongle attached to a centralised computer and is claimed to be capable of flying up to four drones without beginning to lose stability during flight. The out of the box functionality provided by the drone, although somewhat undocumented, is the ability to follow both velocity and position based commands, and to perform commands such as land or takeoff.

The Ryze Tello EDU, the educational variant of the Tello, is a vastly different drone in terms of it's method of control and its expandability. The Tello does not feature near limitless expansion but does however feature more out-of-box stability and some additional functionality compared to the crazyflie such as camera based positioning and loss of connection safety measures. The control method by which the Tello responds is through messages sent over a UDP WiFi connection. Commands accepted by the Tello over this connection are limited to relative positional based movements, takeoff, land, and interestingly the ability to perform a in flight 'flip' [7].

The Tello was planned to be implemented into the platform towards the end of the project however due to global events at the time, the Tello was

Figure 3: An image of the Ryze Tech Tello EDU

unable to be tested fully before the submission of this report.

2.2 Related Works

The creation of a UAV drone swarming platform is by no means a new concept. The coordinated flight of a set of drones to form three dimensional shapes, to organise spectacular light based displays, or to enable a platform for large scale demonstrations and testing is commonly available in a wide range of applications. Where the proposed platform differs from the following implementations is generally not in the raw performance of the tasks at hand but instead in the combination of technologies into a package suited to the projects requirements and constraints.

2.2.1 Crazyswarm

The platform's primary development drone, the Bitcraze Crazyflie 2.0 has a considerable history with researchers and drone hobbyists alike and as such it is not unexpected that drone formation and swarming platforms have been developed for the drone. The most prominent platform designed for the coordinated flight of many crazyflie drones is the CrazySwarm platform primarily maintained by Wolfgang Hoenig [8]. The CrazySwarm platform features the ability to coordinate the flight of up to 49 crazyflie drones using three crazyflie radio dongles on one machine. The crazyswarm platform also features a robust user Application programming Interface (API) written in the python programming language to efficiently control the swarm of crazyflie drones and considerable drone firmware alterations to achieve the impressive flight numbers [8].

The CrazySwarm platform holds a considerable number of features that our proposed platform also aims to incorporate and as such it's implementation has been widely considered during development. The first of which is the user API which has formed the basis for that of the proposed platform displaying necessities for drone identifications, initial (home) positions, takeoff, go to, and land commands, and time and delay helper functions [9].

Although many of these features and takeaways fit our proposed platform, the Crazyswarm platform itself could not be utilised in our implementation for a number of reasons. The first of which is the API which currently only supports use through the Python programming language whereas our platform requires the implementation of a Matlab control API. This complication by its self however does not disqualify the platform as workarounds, although possibly obtuse, can be developed to enable control through any language. The second complication in fact comes from the platforms ability to fly an astonishing 49 crazyflie drones at once despite the limited bandwidth of the crazyflie radio. This task is accomplished through pushing a large portion of platform processing off of a centralised system and instead onto the drones themselves [8]. This in turn allowed for much less bandwidth to be used to control a much larger set of drones achieving the goals of the Crazyswarm platform but compromising the goals of our own. The effects of a decentralised approach on achieving our platforms goals will be discussed in the following Distributed Crazyflie Platform subsection.

2.2.2 Distributed Control Platforms

A common occurrence amongst UAV control structures is to divert processing off of a centralised control station and instead perform a majority of it on board the flying drones within the system. This architecture is prominently seen in UAV formation platforms focusing on the controlled flight of a number of drones in outdoor environments and is particularly useful in this scenario due to the often vast distances between each drone and what would be a central control station. [10] [11]

The second case for a decentralised control platform is in attacking the problem of API dissimilarities and incompatibilities between drones. The PX4 autopilot control software aims to alleviate issues surrounding the controlled

flight of a multitude of different drone types through the standardisation of control protocols [12]. This is accomplished by flashing a common firmware onto a variety of drones that enables the control through a similar to identical source. This approach although beneficial in many ways does not allow the easy implementation of additional drone types. The setup and preparation times associated with flashing new firmware, and the inability to operate using more closed off drone types means that this method is infeasible. One of which drones is the Tello EDU which does not natively support the flashing of new firmware on the device. Given that, the PX4 autopilot platform may become one of the first drone types alongside the crazyflie and Tello to be introduced to the proposed platform through its own wrapper layer going forward.

2.3 Proposal

The proposed multi-drone formation and control platform aims to achieve the following deliverables:

1. A robust, straightforward, and unified user API supporting the development of a wide variety of user generated programs.
2. At a minimum user API support in both Matlab and C++.
3. A drone management server featuring the ability to control and provide logging information for a large number of diverse drones simultaneously.
4. A visual feedback and event logging system capable of fulfilling the requirements of researchers interested in the usage of this platform.
5. A drone safeguarding and collision avoidance module in built to the platform which limits the possibility of drone-on-drone or static bounds collisions within the system.
6. Considerable documentation on the platform's usage. From the user facing API to how to implement additional drone wrappers for the platform. This will be the primary information source for users when developing using the platform.
7. A series of example programs depicting the appropriate use of the platform as well as a few example drone wrapper classes to demonstrate the wrapping of additional drones. The supplied wrappers will include a crazyflie 2.0 wrapper making use of high level functions, a Tello EDU wrapper, and a wrapper relating to virtual drones (named vflies hereafter).

The platform is designed in a way such as to retain as much modularity as possible amongst the above stated deliverables. The modularity of such a system gives capability to users in modifying the platform to achieve desired

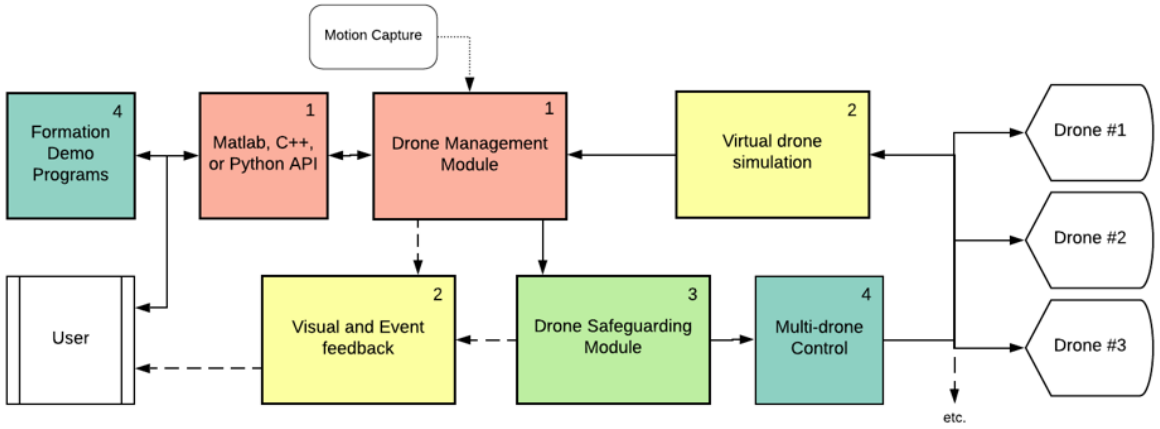

Figure 4: Overview of the proposed multi-drone formation and control platform

functionality. An example of why platform modularity is important is if UAV research requires the replacement of any number of platform capabilities, that being the platform's in-built safeguarding measures, or multi-drone control structure.

With this in mind, the proposed platform's functionality can be best described through four major feature sets. The four feature sets in question are titled as follows:

1. Drone management server and user facing API.
2. User Feedback and Drone Virtualisation Module.
3. Drone Safeguarding and Collision Avoidance.
4. Diverse Control and Demonstration Programs.

Module interactions is primarily conducted through ROS communications and relevant sections such as user programs and drone control are executed asynchronously.

2.3.1 Drone management server and User facing API

The drone management server can be considered as the centre of the proposed platform. This module's primary task is to receive messages from the user facing API and translate those messages into control signals to the relevant drones. The drone management server also facilitates all communications between drones regarding safeguarding constraints and acts as the central data point for the redirection and pre-processing of motion capture information.

The user facing API is a general term regarding the interface an end user will make use of to control drones within the platform. This interface was developed in two languages from the beginning of the project with the possible addition of Python as extension. The first two languages that were developed for is Matlab due to it's usefulness in research applications and requirement of the platform and C++ due to the relative ease of it's development due to the reuse of code from development on the drone management server.

The API features the ability to control individual drones through positional, or velocity based controls and also features the ability to call higher level functionality on the drones such as automated landing, home positioning, and takeoff. Furthermore the API features various settings by which the user can control the operation of the drone server such as by setting the update rates, or enabling/disabling functionality.

2.3.2 User Feedback and Drone Virtualisation Module

The user feedback module was developed with the primary focus of accommodating the wishes of researchers in using the platform and monitoring the performance of drones within the system. The user feedback module was constructed alongside the drone management server and user facing API and contains two main parts, an interact able GUI and a 3D real-time graphical

visualisation of the platform. The GUI gives an end user the ability to read important values from active drones within the system such as position and flight state, to read and manage logging information on a per drone basis, and to call important commands to the platform. The 3D real-time visualisation window displays real-time positions and orientations for all rigidbodies active on the platform.

The second part of this feature set is the introduction of a virtual drone type to be used in tandem with physical drones within the system. This virtual drone enables us to actively develop the drone platform without the requirement of using the physical UAV environment. In addition to this the introduction of a virtual drone type as part of this feature set will enable researchers to relieve some dependence on physical drones in testing their developments or allow testing using far more drones than is physically available at their disposal.

2.3.3 Drone Safeguarding and Collision Avoidance

The drone safeguarding and collision avoidance feature set aims to accomplish one of the core requirements of the proposed platform. This feature set, as the title suggests, ensures that no collisions occur within the system between two tracked objects, or with the bounds of the environment as well as ensure other safety measures are put in place. The collision avoidance measures was accomplished through the implementation of a velocity obstacles algorithm. Two viable options of which are the collision cones approach which manipulates a UAV's trajectory to avoid potential collisions [13], and a potential fields approach which treats obstacles as negative attraction fields which act to 'push' the UAV into safe flight patterns [14]. Other safeguarding measures include the safe landing of drones upon drone server shutdown, automatic timeouts in case api programs suddenly disconnect, and platform emergency functionalities.

2.3.4 Diverse Control and Demonstration Programs

The final feature set of the proposed platform centres around ensuring the applicability of the platform to a diverse set of drones and in creating example programs and documentations for the platform.

One of the platform's goals is to provide a generalised pipeline to allow the streamlined control of any number of drone types. This feature set aims to accomplish this functionality through the design of a robust drone wrapper framework by which the drone management server is capable of communicating with. The implementation of new drone types involve users working with this framework to easily add functionality to the platform.

As the platform is designed to be used by researchers and other end users alike it is imperative that the platform is welcoming to new users. To ensure that this is the case, the final feature set of the platform will put considerable efforts into developing a range of feature rich example applications as well as easily accessible documentation of platform end points.

2.4 Method and Section Analysis

2.4.1 User API and Drone Formation Scripting

Layout of the API

The user API was made such that it was as easy as possible to create new scripts for the platform. Drone formation scripts are created in C++ simply by including the user_api header, and in MatLab by including the user API files in the application's PATH. From here drone scripts can be developed by calling the functions which make up this API. The following paragraphs will refer directly to the C++ user API however they do apply to the MatLab API as well as it was a goal to keep the API as similar as possible between languages.

The First step of any drone script is to initialise the API program. This is done by calling the $mdp :: initialise(...)$ function passing in the desired update rate and the name of the program. To conclude a program, the script will call the $mdp :: terminate()$ function which will automatically send all in flight drone's to their designated home locations and land.

All drone's active on the platform will be identified by a user script through a $mdp :: id$ class. This class contains the numeric id of the drone on the server, and an identifying name which has been supplied as the rigidbody tag through Optitrack. To receive a list of all active drones on the server, the script will call the function $mdp :: get\_all\_rigidbodies()$ which will return a list of $mdp :: ids$ representing each active drone. This function refers to rigidbodies as it will also return ids representing other rigidbodies identified by the platform such as any declared and marked non-drone obstacles.

The movement of drones has been distilled by the API down to calls to two functions, one for setting a drone's new desired position, and another for setting a drone's desired velocity. $mdp :: set\_drone\_position(...)$ takes a $mdp :: id$ referencing a specific drone, and an instance of $mdp :: position\_msg$ indicating

the desired end point and duration the drone should take to arrive. $mdp :: set\_drone\_velocity(...)$ similarly takes a $mdp :: id$ referencing a specific drone, and an instance of $mdp :: velocity\_msg$ indicating the desired velocity and duration the drone should hold this velocity for. By default a new call to one of these functions will override all previous calls, for instance if a drone is performing a velocity command when it receives a new position based command, it will abandon the previous velocity command immediately and perform the new position command. It is possible however to allow the system to instead queue up events such that one is only ever performed once the last has completed. This option can be enabled during script initialisation.

The API also allows the script to read the current position and velocity of all rigidbodies tracked by the drone management server. This information is retrieved by calling either $mdp :: get\_position(...)$ or $mdp :: get\_velocity(...)$ and passing in an instance of $mdp :: id$ referencing the desired rigidbody.

The user API also features the ability to make use of common drone flight functionality such as calling for the drone to hover, takeoff from the ground, or land. These functions can be called through $mdp :: cmd\_takeoff(...)$, $mdp :: cmd\_land(...)$, and $mdp :: cmd\_hover(...)$. The platform keeps track of a home position for each drone. This home position is designated on drone initialisation as the starting x and y coordinates of the drone. The user API can interact with this location by either setting this to a new position, retrieving the home position, or calling a specified drone to return to and land at it's home position. This functionality can be used through the following functions: $msp :: set\_home(...)$, $mdp :: get\_home(...)$, and $mdp :: goto\_home(...)$

During operation, drones on the platform are constantly identified to be in one of the following five flight states: LANDED, HOVERING, MOVING, DELETED, and UNKNOWN. The state a drone is in is identified by the rigidbody class and will be discussed in that section. The drone's identified state is able to be retrieved by the user API at any point during operation. This can be

done with a call to the function $mdp :: get\_state(...)$ passing in a reference to a rigidbody id $mdp :: id$. The API's get state function operates by looking for a ROS parameter representing the current state of the requested drone. By using this method instead of ROS services, the API is able to return the drone's current state near immediately with minimal retrieval delays.

In order to avoid the user having to implement their own quantised control loop through the use of operating system sleep functions, the user API has included the $mdp :: spin\_until\_rate()$ function. This function returns once 'rate' time has passed since the last call to $mdp :: spin\_until\_rate()$. This 'rate' value is given by the user during script initialisation as a number in Hertz generally between 10Hz to 100Hz. For example, if the scripts designated rate is 10Hz (0.1 seconds between quantised 'frames') and it has been 0.06 seconds since the last call to $mdp :: spin\_until\_rate()$, calling $mdp :: spin\_until\_rate()$ will cause the script to wait for 0.04 seconds to hit the desired update rate of 10Hz and then return such that the script can continue operation. If a time period of 0.13 seconds has passed since the last call to $mdp :: spin\_until\_rate()$, then the function will immediately return as the desired update rate has already passed.

The combination of getting a drone's state and spinning allows the creation of a very useful control function $mdp :: sleep\_until\_idle(...)$ which spins the program using $mdp :: spin\_until\_rate()$ until the drone specified enters the HOVERING, LANDED, or DELETED states. The result is functionality for a script to be able to wait until the previous command has completed before continuing execution. For example if the script makes a call to set a drones position over 4 seconds and then makes a call to $mdp :: sleep\_until\_idle(...)$ on that same drone, then the script will pause execution until the drone has entered the idle state after completing the 4 second set position call. With this system a simple waypoint program can be created by interleaving drone set position commands with calls to the sleep until idle function.

An example of a drone formation script using this user API and making

use of both dynamic and waypoint flight can be seen in appendix A of this document.

Categories of user API commands

From this number of functions, user API commands can be split into a number of different categories based on the method by which they achieve their functionality:

1. Set commands which interact with the drone server; set_position, set_home, cmd_takeoff, etc.
2. Get commands which interact with the drone server; get_home.
3. Get commands which look at ROS topics or parameter server; get_position, get_velocity, get_state, etc.
4. user API commands which do not require communicating outside of the application; spin_until_rate.

Category 1 Drone Server Communication Issue and Resolution

For the various commands that interact with the drone server, in order to either retrieve or send data it was decided to make use of ROS topics and services. For the first category, all commands send a unified message over a single API topic to the drone server, adjusting the parameters of this message to distinguish between the different commands. Originally the user API only existed for C++ and these messages were constructed as custom ROS message contained a string identifying the command type, for example ’POSITION’ or ’VELOCITY’, and a number of other data fields to specify command parameters. MatLab however has a somewhat complex support for ROS custom messages. In order

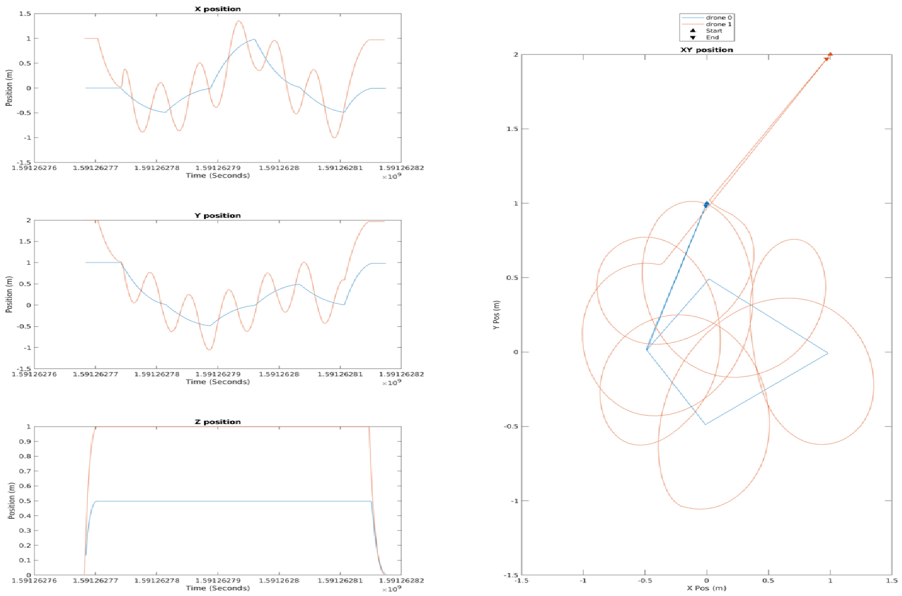

Figure 5: This image displays the automatic post-flight analysis of the 'do_donuts' user API script available in Appendix A. In this script, drone 0 (blue) is given four waypoints to follow while drone 1 (orange) is instructed to circle around the current position of drone 0.

to enable their use, message schematics must be manually added to each user's individual MatLab installation. The amount of work this will push onto the user before they are able to operate the user API through MatLab goes against one of the core requirements of this platform. Hence the alternative to encode the first category of user API commands into standard ros messages, which are automatically enabled through MatLab's ROS library, was used instead. The details by which user API commands are encoded in ROS standard messages can be seen through the source code sample available in Appendix B.

Category 2 Data Retrieval Issue and Resolution

The initial implementation of category 2 commands made use of ROS services to interact with the drone server and retrieve the relevant information. ROS services involve a server and a client, the server in this case is the drone

management server and the client is the user API. Upon calling a ROS service a message is sent to the server, who then performs a task on the server and sends back a response to the client. The problem encountered when making use of this functionality for data retrieval functions on the user API is the amount of time this interaction took. In order to retrieve the position of one drone using this system, the call to $mdp :: get\_position$ may take upwards of 4 milliseconds to complete. When attempting to get the positions of a number of drones this delay may be magnified intensely. The result of this issue is slow execution of drone scripts which require frame by frame location data of drones within the formation, and will cause immense issues to algorithms that are time sensitive.

The solution to this issue was to instead allow the user API to retrieve location data directly from the ROS topics used by the drone server. For the MatLab API this modification was as simple as swapping ros functions from services to topics.

The modification to the C++ API however proved to be much more difficult due to it's handling of ROS topic subscriptions. In C++, the subscription to ROS topics receives messages in a way that can be loosely described as asynchronous. This meant that it was not possible to simply poll the relevant ROS topic for the latest position of the specified drone. To resolve this issue, the C++ API was modified to store drone data internally which can serve to store the last known location and velocity asynchronously received from the ROS topics. Each time a call to $mdp :: get\_all\_rigidbodies()$ is made, a list of this drone data is updated such that each active drone has it's own drone data struct, and subscribers to both it's pose and velocity ROS topics. when a call to a category 2 command is made, the latest asynchronously queued topic message is handled for all drones and the most current data is retrieved. The performance of this method far exceeds the service method and is able to retrieve up to date information near immediately allowing for more dynamic flight of drones through the user API.

With this modification the code in Appendix A is able to function. If this modification was not made, then setting the position of one drone based upon the position of another each frame would result in stuttering and largely inconsistent movement.

2.4.2 Drone Management Server

The drone management server is the central system operating the multi-drone platform and contains the list of active rigidbodies currently in operation. The server also has a primary role in receiving user API commands and distributing these commands to the relevant rigidbodies operating on the platform. Platform initialisation, shutdown, and platform-wide operations such as emergency are performed on the level of the Drone Management Server.

The drone management server holds a list of rigidbody classes where each rigidbody class represents a single drone in operation on the platform. whereas the rigidbody conducts all functionality relevant to a single drone, the drone management server conducts functionality that impact multiple at once. The adding and removing of drones is handled through services existing as part of the drone management server class.

The drone management server has been through many iterations since the start of development, originally responsible for the processing of almost all user API messages. However since then much of this functionality has since been shifted onto the rigidbodies themselves to make use of asynchronous operation. The server's main task in the handling of user API messages has since changed to instead act as an intermediary step in translating the encoded standard ROS messages into an easier to handle custom message to eventually be received by the rigidbody classes. The case study presented in this thesis on point set registration of rigidbodies from marker point clouds is also implemented as part of this class due to its centrality to the multi-drone platform.

2.4.3 Rigidbody class

The rigidbody base class represents a single rigidbody operating on the platform. This rigidbody class holds the base functionality that applies to all rigidbodies no matter what their drone 'type' is, where examples of differing drone types are Crazyflie, Tello, or a virtual drone (vflie). Drone types are defined by the platform as inheriting the functionality and building upon the functionality present in this class. The rigidbody class holds the vast majority of platform functionality when it comes to the generic flight of drones. Due to the sheer amount of functionality in this class, many details will be left out of this report in favour of describing it's role as part of the platform and explaining some challenges which were overcome during it's development. As such following is a list of the major functionality that this rigidbody class handles:

- Per drone initialisation of various ROS structures
- Safe shutdown of the drone
- Timely reading and conversion of API commands received from the drone management server into functions implemented by drone wrapper classes.
- Handling and conversion of 'relative' movement options of the user API
- Validation and trimming of input API commands such that drone wrappers are not overwhelmed by vast numbers of incoming identical commands.
- Handling of per drone data including current pose and velocity, home positions, identification data, etc.
- Recalculation of drone velocities when new motion capture information is received.
- Publishing of a number of per drone information streams to relevant ROS topics.

- Queuing and de-queuing of API commands such that multiple commands may be completed one after another in order.
- General logging of drone activity.
- Handling and detection of drone flight states.
- Command safety timeouts such that a drone cannot be locked hovering in flight.

Flight States

Prior to the introduction of the current drone flight states system, the method to determine whether a drone is currently in the process of completing a command was based entirely on that commands inputted duration. This system however saw increasing issues particularly in the handling of packet loss over the crazyflie's radio communications. With the old system it was impossible to determine whether a drone had received it's last command nor was it possible to accurately determine when a drone had completed it's last. The result of these inefficiencies in the example of waypoint programs were missed waypoints, early transitions between waypoints, and in rare circumstances the missing of vital land commands which resulted in infinitely hovering drones (this particular circumstance was dealt with by the platform's safety timeout functionality).

The new system of flight states instead directly looked at a current drone's position and velocity values to determine what the physical state the drone is during flight. The possible states a drone can be in at any one point is identical to the states discussed in the user API section above. With this state system, missed commands due to packet loss can be identified through witnessing a differing physical state to what is to be expected, for instance if a drone receives a set_position command but remains in a HOVERING state,

then it can be determined that the position message was lost and another can be sent. Additionally this system resolves the issue of early transition by, instead of guessing the time of the following command via the last commands duration, the detection of a HOVERING state following a successful movement message will indicate the availability for the following command. The result of this system is more accurate and reliable API commands being conducted on drones pushing forward to reliability and usability of the drone platform.

API Message Trimming

Another issue discovered during the implementation of the crazyflie drone was related to overwhelming the drone's firmware with movement commands. Pushing new commands to the drone at 100Hz, like what can be seen in a simple implementation of a teleoperations controller, resulted in the crazyflie stuttering in it's movement and overshooting end goals by over 2 meters during testing. The issue was distilled down to be caused by delays in processing on the crazyflie's internal firmware. When a flight message such as 'go to position' is sent to the drone, the on-board firmware began processing of the flight command at which point the drone defaults to follow its current trajectory. This processing takes a certain amount of time before the drone can begin performing the necessary movements to fulfil the specified task. It was expected that this processing time meant that little to no time was given to the drone to perform these movements before the next command was sent and processing began on that.

The solution to the problem was the introduction of a check in the rigidbody class that determined whether an incoming command was identical or nearly identical to the drones last enacted command. If the new message is considered to be identical based on a reasonable number of criteria, the new command is discarded and the drone continues operation on the last

accepted command. As a result, the user is able to send user API commands at any throughput desired without worrying about overwhelming drones with commands. In addition to improving reliability of the platform, one less aspect of physical drone flight is required to be managed by an end user.

2.4.4 Drone Wrapper Classes

The drone wrapper framework serves as a method to extend functionality of the multi-drone platform and add additional drone types ready for flight. Drone Wrappers are created by inheriting the rigidbody class and implementing abstract functions to perform the desired task through the drone's native API. In addition to being an end point to the drone platform, it also features a number of helper functions to aid in the introduction of new drone types. Modifying the source code for a platform is not an easy task to be done by an end user, hence this process of adding additional drone types has made this task as easy as possible.

To add a new drone to the platform, the user will write C++ implementations based on a supplied drone wrapper template class inheriting from rigidbody. This template includes a number of functions that allow better communication with the platform such that all relevant sub systems are notified of the new drone type. Once completed, the user will then place the drone implementation C++ file into the specially marked '/wrappers' folder at which point it will automatically be added to the drone platform during compilation.

This compile time functionality is achieved through the use of a python script which runs automatically during compilation of the multi-drone platform. The script searches through all the files located in the /wrappers folder, identifying and adding drone types based on the user supplied tag. From this information, the script generates additional C++ code that is used by the drone management server and drone adding programs. The addition of this func-

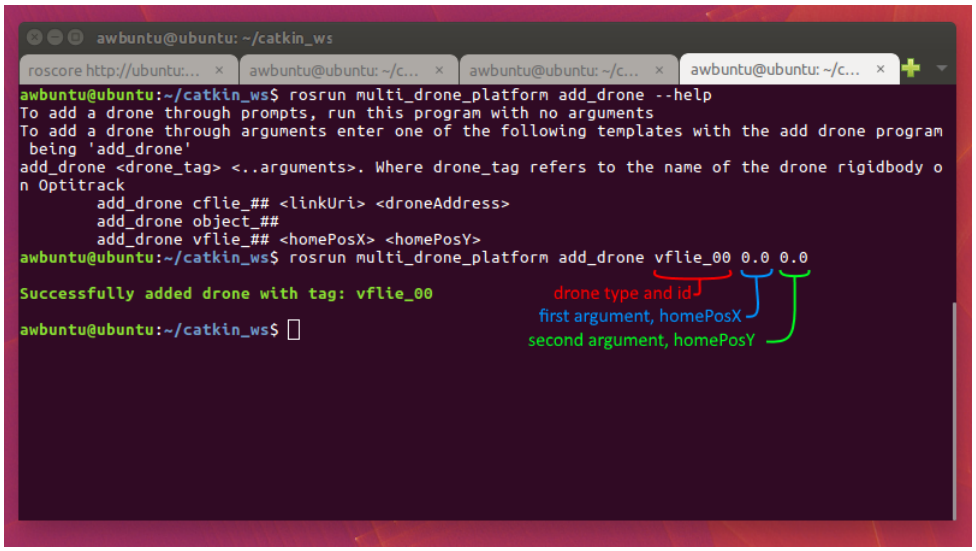

Figure 6: This image demonstrates the declaring of drones during run time. Here a virtual drone (vflie) is being declared using the arguments specified in the drone wrapper.

tionality means an end user does not need to know the details of ROS' C++ compilation environment, nor the details of the compilation tools CMake in order to extend functionality of the platform in the form of new drone types. As such this vastly improves the usability of the platform and the ease at which it's functionality can be extended.

Drone wrappers also allow the user to specify input arguments which are required by the drone upon initialisation. For example in the virtual drone implementation, it is not known at startup where in the world this drone lies. Hence the input arguments which must be specified at initialisation for this drone type include at a minimum the drone's X and Y home coordinates. Similar to above, these arguments are automatically integrated into the platform upon compilation using a python script and included as requirements for declaring a drone to the platform at run time.

As each drone is represented by one drone wrapper class on the drone management server, there do exists issues with this approach when adding drones that require their own server program to operate. In the example of the

crazyflie drone type, every crazyflie drone connects to central crazyflie server in order to contact the physical drone through a single radio. In a case such as this, the crazyflie server is launched as a separate ROS node alongside the multi drone platform. Each individual crazyflie drone then has its wrapper class serve as a communication link between the multi-drone platform and this external crazyflie server, for example a call to set position on the crazyflie wrapper will be redirected over ROS to the crazyflie server ROS node. After which point the crazyflie server can deal with distribution of crazyflie related movement commands.

2.4.5 Virtual Drone Implementation (Vflie)

The multi-drone platform's virtual drone implementation is implemented in such a way that it does not require any additional functionality within the system compared to a standard drone type wrapper allows. As such it serves in a way to test the robustness of the drone wrapper framework along side the implementations of both the crazyflie and in part the Tello drones.

The virtual drone operates off of buffers internal to the drone wrapper representing it's desired position and orientation at any given time. By iterating on these buffers as dictated by incoming movement commands, it can simulate the movement patterns an ideal physical drone will take within the environment. To improve believably, the virtual drone makes extensive use of linear interpolation functions comparing the time between each frame to present smooth movements across space. Similar to a standard drone, the multi-drone platform receives current pose information from the virtual drone at a rate dictated by Optitrack generally between 100 and 120Hz. The virtual drone uses this consistent update rate to perform the necessary movement tasks and update it's own position and orientation information in sync with rigidbodies from Optitrack. Through pushing computational efforts to the same stream as

position updates, the virtual drone is able to perform a majority of its tasks asynchronously to the operation of the drone management server.

The result is a virtualisation of a single drone that performs the tasks being sent to it in what is considered a near 'perfect' manner. This enables the vflie drones to be used to test drone formation scripts purely on the merits of it's algorithms involving little to no complications that may arise from running the same script on a physical drone.

2.4.6 Safeguarding Efforts

Safeguarding efforts have been implemented into the platform in order to avoid damages to possibly expensive physical equipment.

The first of these safeguarding efforts is the introduction of two stage movement command timeouts on all active drones. The first stage occurs when a drone does not receive a movement command beyond the completion of its last. Once this stage has been triggered, the drone automatically reverts to the hovering state and awaits another command from the user API. If this command is not received over a certain time frame, the drone triggers stage 2 of the timeout measures. At this point the drone will assume that an error has occurred that resulted in it locking in a flying state and will proceed to automatically land. This small addition adds immense safety in case of incorrect or incomplete execution of drone formation program and goes a long way to improving the usability of the platform for researchers or hobbyists.

A drone safeguarding module was added which makes use of artificial potential fields to ensure no collision can take place between two tracked objects on the platform. By modifying movement commands on the fly depending on the movement and positions of other rigidbodies, the platform should be able to avoid costly collisions between drones mid flight. The implementation of this module is being conducted by the platform's co-contributor and will be

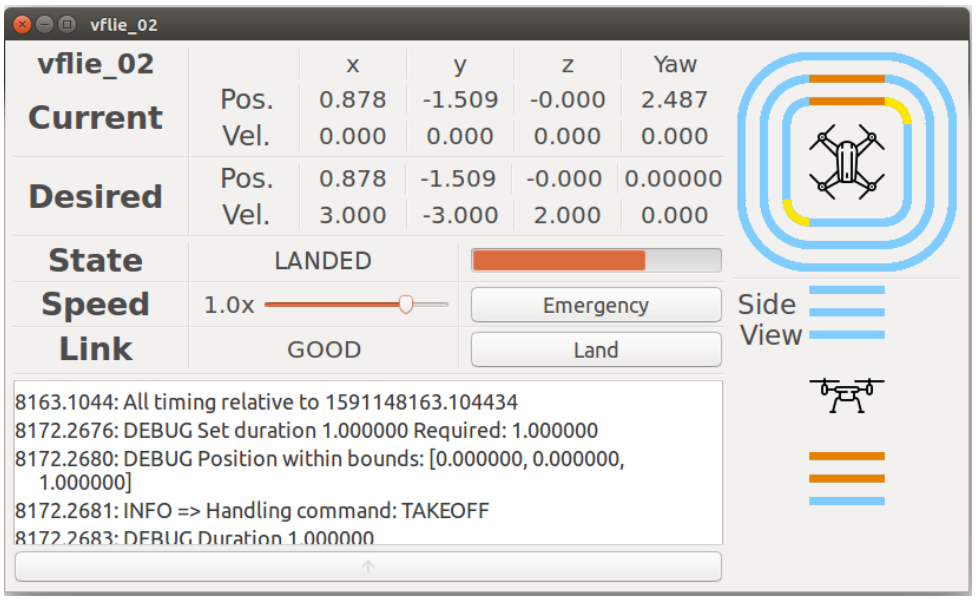

Figure 7: An example of the drone Live View debugging window.

discussed in detail as part of their thesis.

Lastly command safety limits were included in the platform such that commands from the user API cannot approach absurd parameters such as a position movement of 10 meters in 0.1 seconds.

The combination of safety efforts on the platform has allowed for drastic improvements in a user's quality of life when operating the platform and has come a long way in fulfilling platform requirements.

2.4.7 Debugging GUIs and Logging

Debugging on the platform is imperative for its usability by researchers in understanding the operation of their developed algorithms. The multi-drone platform therefore has been developed to provide reasonable logging and debugging functionality that can both be viewed in real time and post session.

The first debugging effort available is the platforms developed logging functions. Available to all drone wrappers is the ability to incorporate drone specific logging information in the debugging pipelines developed for the plat-

form. Through using these pipelines logging information can be categorised based on severity and isolated to the affected platform component.

During platform run time, the user has the ability to launch drone debugging (Live View) windows able to display current flight information on a per drone basis. The combination of a number of Live View windows allows for a wide look over flight activity for all drones currently operating on the platform. In addition to displaying flight information, the Live View windows are able to display instances of safeguarding events, where in relation to the drone these events occurred and to what extent.

2.4.8 Documentation

Documentation has been developed for the platform making use of the Doxygen code documentation tool. Using this we were able to create professional programming documentation for all components of the platform expected to be interacted with by an end user. With this documentation, along with additional platform operational guides to be created at a later date, knowledge of the correct operational usage of the platform should be thoroughly transferable. All platform documentation will be included with the source under the documentation directory and will include information regarding platform installation, declaration of drones during run time, the addition of drone wrapper classes, and the creation of drone formation programs.

3 Point Set Registration of Drones

Going forward with development on the platform it was soon identified that the current motion capture communications system being used had been the cause for several problems facing the multi-drone platform. The first of which is the sporadic loss of rigidbody tracking when there is not enough observation on the relevant markers by Optitrack cameras. The second issue witnessed in somewhat rare occurrences was the swapping of rigidbody identities amongst the Optitrack tracked objects. Additionally the previously used communications method, VRPN, did not have the capability to perform native actions on the Optitrack system, nor did it feature the ability to communicate anything other than identified rigid body configurations.

It is suggested for the reliable tracking of rigidbodies on motion capture systems that markers constructing these rigidbodies hold entirely unique configurations between objects [3]. Our platform, aiming to significantly reduce preparation and setup times, does not allow the expansive setup time this requires and as such we have experienced countless related issues such as skewed orientations, flickering position information, and even the swapping of rigid body identities amongst tracked objects. The last of which requiring completely re-initialising of rigid bodies on Optitrack and restarting of the platform.

Given these issues it has been decided that rigid body tracking and point set registration of raw point clouds will be done on the multi-drone platform in order to better deal with the marker occlusion issues and unique marker configurations as mentioned above.

The construction of rigidbody objects given a point cloud of marker positions will be done through a variation of the Iterative Closest Point (ICP) algorithm. The core conceit of this algorithm is to iterate an initial estimate of an object's state to eventually overlap with point formations in an Optitrack supplied marker point cloud update frame. [15] Once completed the result is

identifying the translation and rotation of the objects rigidbody in relation to a specified origin point. This research and implementation of this will be spoken about with more detail in the following sections.

3.1 Background

3.1.1 Least Squares Best Fitting

Least squares fitting is a method of fitting a curve to a data set via minimisation of the squares of the residuals each point holds from the curve. Two variations stem from the least squares method being linear least squares fitting which deals with the much simpler case of minimising the the squared distances from a standard curve amongst the data set, and non-linear least squares fitting where unknown parameters may be of non-linear i.e. trigonometric nature.

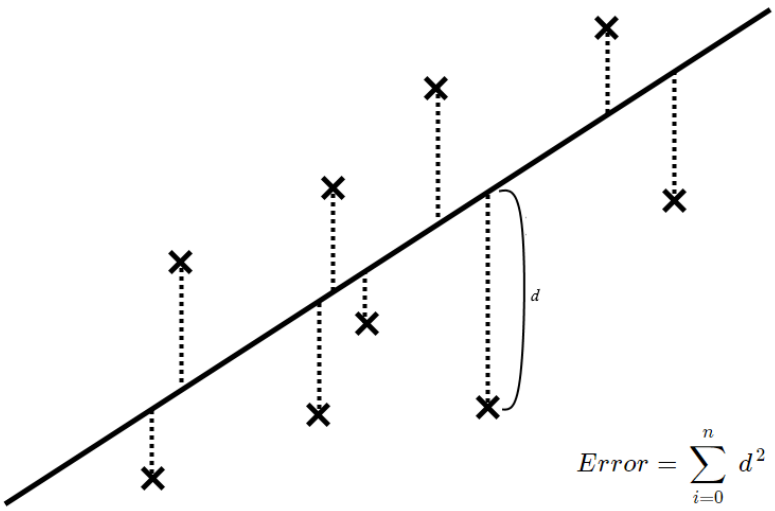

Figure 8: Calculation of the Least Squares error value.

The calculation of best fitting rotation for a rigidbody in 3d space is a minimisation problem that can be solved through one of the variants of the least squares method depending on assumptions made before calculation. The point set registration application of least squares instead of lying in the more common two dimensions of fitting a curve to a data set involve a best fit along a much higher number of degrees fitting amongst the three degrees of rotation, translation, and possibly scale.

3.1.2 Quaternions

A quaternion is a representation of rotation in 3d space and exists as a 4 element vector containing one real component and three complex components. The way in which these four components map onto three dimensional rotation will not be discussed here, however relevant properties of quaternions in relation to this application are as follows.

First, the cumulative rotation of two unit quaternions can be obtained simply by multiplying those two quaternions together. Drastically simplifying the same operation attempting to be done between two euclidean rotations.

Secondly, in order to apply a unit quaternion rotation to a three element vector representing a position, simply pre-multiply the quaternion with that vector. The resulting three element vector is the position rotated about origin by the rotation represented by the input quaternion. [16]

3.1.3 Point Set Centroids and Quaternion Rotations

A point set's centroid is the average locations of all points in the set. If $p_i = (x_i, y_i, z_i)$ is a vector representing one point in the point set P, then the centroid of that point set P is found as:

$$centroid_P = \frac{1}{N} * \sum_{i=0}^{n} p_i$$

where N is the number of points in point set P.

Point set centroids are important both in calculating the best fit translation to minimise the error term and to perform rotations on the corresponding point set.

In order to calculate the best fit rotation to merge the source point set with the destination point set, both point sets must be translated to centre over

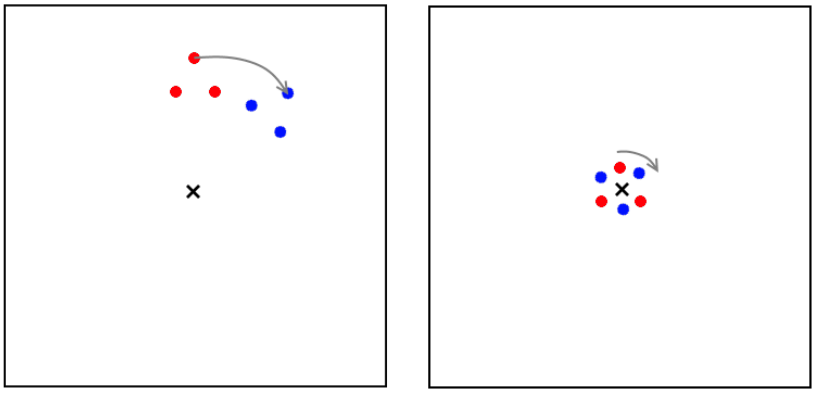

Figure 9: In both boxes origin is represented by a black x and a rotation of 45 degrees is applied as indicated by the arrow from red to blue. The left box shows the result of a rotation when translation to origin has not occurred. The right box shows the result of rotation when translation to origin has occurred.

the origin point in world space. From here a local rotation can be applied to each of the points in the point set by multiplying the point's 3 element vector (x, y, z) by a quaternion representing the desired rotation. The reason for this operation to occur in order to perform a local rotation on a point set is described in Figure 10.

The translation to re-position the point set over origin is done performing a vector subtraction between a point set's centroid vector and each of the points in that set:

$$p_i(originaligned) = p_i - centroid_P$$

Rotations of this fashion always occur relative to the world space origin point, hence a rotation applied to an unmodified point set will have vastly different outcomes to a re-located set. Once the desired rotation has been made, the point cloud can be moved back to its original position resulting in what appears to be a rotation applied locally around the point cloud's centroid.

This rotation pattern is used to apply the initial estimate rotation onto the marker template at the beginning of each iteration, and to prepare point cloud locations for use in best fitting rotation calculations as discussed following.

3.1.4 Eigenvalues and Eigenvectors

When given a square transformation matrix T if a vector v transformed by T results in a vector v' which is nothing more than a scaled multiple of v, then v is an eigenvector of matrix T and the scalar value which relates v and v' is the eigenvector's corresponding eigenvalue λ.

$$T(v) = \lambda v'$$

Eigenvectors and eigenvalues have many uses in computer graphics particularly in the application of transformations onto images such as shears, rotations, and scaling. The classical method for finding eigenvectors of a matrix is to solve the set of linear equations formed from the equation $Tv = \lambda v$, where eigenvector $v = (x, y, z)^t$ and T is a 3 by 3 square matrix. The eigenvalues of transformation matrix T are found as the roots of the characteristic polynomial of T.

3.1.5 Kd-Trees

A KD-Tree is a multi-dimensional variation of the more commonly known computer science data structure, the binary search tree. The way in which KD-Trees build upon the binary search tree allows for very fast computation of nearest neighbour points and other associative searches [17]. As this platform will make use of 3 dimensional KD-Trees (3D-Tree), this variation will be used in following examples.

A KD-Tree, similarly to a binary search tree, is formed from a set of linked nodes where each node in the tree represents one point in the data set. Each node in this tree holds links to two children nodes, identified as it's left and right nodes. The left node will always hold a data element whose value is less than the current node, and the right node will always hold a data element

whose value is greater than the current node. Unlike a binary search tree is how these data elements are sorted. In a KD-Tree, each data element is a vector of K elements where at each node the sorting is done on the element of this vector relating to the current depth within the tree using the following equation: $ComparisonElementIndex = depth\%K$. For a 3 dimensional KD-Tree where each data element is a vector (x, y, z), the 0 index element is x, the 1 index element is y, and the 2 index element is z.

The top most node has it's children sorted by the x element, these child nodes have their children sorted by the y element, the next depth has their children sorted by the z element, and repeating so forth.

From this structure nearest neighbour searches, as required by this point set registration implementation, can be simplified drastically by 'trimming' branches searching down this tree. The result is an algorithm able to find the nearest neighbouring point in a set to an arbitrary input point with a big O complexity of $O(log_2 n)$ where n is the number of nodes in the KD-Tree.

3.2 Literature Review

[15] This article is the first article to propose the Iterative Closest Point (ICP) method for registration of 3d shapes from two point cloud systems. It also does some analysis into other methods of 3d registration proposed prior to its publication date in 1992 particularly in details around the use of single value decomposition or quaternion based solutions to least-squares problems. Relating to the platform's application of this algorithm, this article suggests the use of unit quaternions to solve the least-squares problem of finding the desired rotation for problems in two and three dimensions. Unused in our case, the suggested method for all dimensions above three would be to use the singular value decomposition based on the covariance matrix of the two point sets.

[18] This article merges many of the different variations of the ICP algorithm to improve both its computational time and the robustness of it's results. The article also makes some computational time comparisons between the robust ICP, standard ICP, and two other variations. Comparisons showed that for low amounts of points none of the modified ICP variants gave benefits in computation time over standard ICP. The point cloud that the multi-drone platform receives from Optitrack is very unlikely to contain noise resulting in additional marker locations, and if so will not cause the addition of any where near as many points that will make the additional robustness of these variations required for our use case.

[19] The article 'Efficient Variants of the ICP Algorithm' makes a comparison between various methods for accomplishing the different stages of the ICP algorithm. The article explores methods for matching of points across the two sets, assignment of an error metric, minimising this error metric, and also for the additional steps of selecting a subset of points for each mesh, weighting corresponding point pairs, and rejection of certain point pairs. The comparisons made in this article are focused around registration of exceedingly large

point sets of which 2,000 are selected, whereas the expected number of points to be registered per drone as part of the drone platform are expected to be in the range of 4 to 10. With this being said relevant sections in this article to the multi-drone platform's application relate to matching of points (indicating a mix of projection and kd-tree to find the nearest point), error metrics and minimisation, and comparisons between high speed ICP variants. The error metrics discussed in this article commonly centre around generating a 'sum of squared distances' to indicate if any additional iterations are required, and differ in where these distances are between. The most appropriate approach discussed here appears to be the simplest, that being the sum of squared distances between two corresponding points in the two sets. Given the very small size of the point clouds in the drone case and the expectation that the source and destination clouds will be very close in pose between frames, a corresponding destination point for each source point should be easy to find. The methods discussed in this article will be used to form the basis of this platform's implementation of ICP.

[20] This article provides an alternative method of point set registration that does not follow the same lines of Iterative Closest Point. This new method makes use of instantaneous kinematics to produce the optimum translation and rotation that will transform the source cloud to overlap the destination cloud. This method has been claimed to provide faster convergence rates than the standard ICP however at the cost of being much more complex in its implementation. In this method instead of computing the centroids for translations and computing the rotations from various means, a velocity field is generated from each point in the source point cloud to a point on the destination point cloud. From this estimated motion towards the destination cloud a euclidean displacement is generated and performed. Like the ICP method, this is repeated until an error tolerance is reached and the clouds are determined to be matched. The generation of a velocity field representing the linear 'movement' of each

point from the source cloud to the destination cloud appears to be a very logical and innovative approach to point set registration. However it's presentation in this report is exceedingly complex compared to the ICP methods found in other reports.

[21] The article 'Least-Squares Fitting of Two 3-D Point Sets' presents an algorithm for finding the rotation matrix and translation vector which minimises the least-squares problem using singular value decomposition. This algorithm suggests the decoupling of translation and rotation segments to be solved separately. By doing this the translation can be found simply as the difference between the centroid of the rotated source set and the centroid of the destination set. The rotation estimate can be found independently using the Singular Value Decomposition (SVD) of a 3 by 3 matrix constructed from the centroids of the two sets. In comparing this algorithm to an iterative method and a quaternion based method it was found that this SVD method provided comparable results to the quaternion based method with slightly greater performance found in the quaternion method when the number of correspondence points are low.

For the drone platform's use case, the number of correspondence points are expected to be for the most part within 4 to 10 points. That being said however, the differences indicated in this report mean that the use of either this SVD method or the compared quaternion method can be used without worrying about performance impacts.

[22] The use of a point-to-plane error metric when determining the next iteration allows the ICP algorithm to converge much faster than the use of a point-to-point error metric. The issue with implementing a point-to-plane error metric however is the requirement to solve a non-linear least-squares problem each iteration which traditionally a very slow process. This article suggests a small simplification to the least-squares problem such that it can be solved much faster as a linear least-squares problem given the destination rotation is

within a small margin of the initial estimate. An implementation of this method was attempted for the drone platform however a stable solution was unable to be found. The convolution of translation movements and rotation elements into the one problem set meant debugging and categorising incorrect events was exceedingly difficult.

[16] This article proposes the quaternion alternative to SVD methods for solving the least-squares problem for rotation. This method suggest that the best fitting rotation can be found through first calculating a matrix from a sum of products of every corresponding point in each data set. The unit quaternion which represents the best fitting rotation can then be found as the eigenvector which corresponds to the maximum eigenvalue of this calculated matrix. This article also proposes a simplification to the case where there are only 3 corresponding points amongst the two sets. This simplification however cannot be used in the drone platform as it is unlikely (and not recommended) that only three markers are used to identify drones due to the risk of missing markers between frames. That being said this method seems fairly simple to implement whilst also providing relatively fast performance for small point sets as indicated in earlier articles.

Another article which builds on the quaternion algorithm has to do with the use of 'dual number quaternions' to improve both the speed and accuracy of the original algorithm [23]. Additionally there exist various other point registration algorithms other than ICP such as a method based on 'multivariate Epanechnikov kernels' [24]. Although this method is much newer and performs faster than the ICP algorithm, it's implementation and theory is far more complex.

The solution which will be implemented going forward is the ICP algorithm based around the use of unit quaternions [16] to determine the best fitting rotation. This algorithm by far appeared the best compromise between performance, simplicity, and reliability as indicated by the above articles.

3.3 Method

Iterative Closest Point (ICP) is an algorithm for finding the most optimal translation, rotation, and scale which fits a source point template to a destination point cloud. As the algorithm's name suggest, ICP uses an iterative approach to come to an optimal transform where each iteration of the ICP algorithm consists of four major calculations:

1. An initial estimate of position and rotation is supplied to the algorithm. This estimation will be iterated upon to find the best fitting pose. For the drone platform's implementation, this initial estimate will be the best fitting pose as determined by the ICP iteration from the previous frame. As each frame will be captured at 100-120Hz as determined by Optitrack [3], this estimate should be sufficiently close to the drone's actual.

2. The source point template, that being the relative locations between markers identifying the drone, is transformed by the initial estimate pose and for each point in the source template a corresponding point is chosen from the destination (Optitrack supplied) point cloud. For the drone platform's implementation, this selection will be done using kd-trees and the nearest neighbour algorithm. When finding a source point's nearest neighbour in the destination set, the euclidean distance between source and destination point is also returned. The distances between all corresponding points is added together to form the iteration's error value.

3. The centroids of both the source point set and the corresponding destination points is calculated. From here the best fitting rotation and translation can be found for this iteration by solving a least squares minimisation problem between the points of each set. This calculation will be done using the unit quaternion method identified during literature review.

4. Lastly the pose estimation is updated ready for the next iteration of the

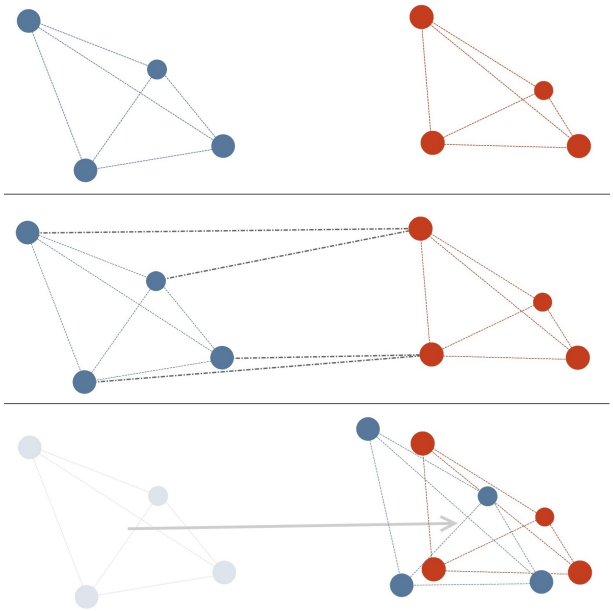

Figure 10: Demonstration of the one iteration of the ICP algorithm. For each part, the blue points on the left represent the source point cloud and the red points on the right represent the destination point cloud. The middle panel displays each source point matching with a point in the destination cloud using the nearest neighbour method. The bottom panel displays the resulting translation after conclusion of the first iteration. The second iteration will result in the two point clouds overlapping with successful registration.

ICP algorithm and the calculated error value is compared to determine whether it is within a pre-determined error tolerance level. If so the iterative process is stopped and the best fitting pose is returned, otherwise the next iteration is performed.

3.3.1 KD-Tree Implementation

There are many implementations for KD-Trees that exist in C++ that may be included as a third party library. These implementations however tend to contain far more functionality then is necessary for it's application in this platform. In order to find the corresponding point in the destination set for each point in the source set, the requirements for KD-tree functionality as needing by this platform's ICP implementation is as follows:

1. The KD-Tree must support 3 dimensions, all other dimensions above or bellow 3 are not required for this application.

2. The KD-Tree must be able to add nodes (where one node corresponds to one point in the destination set) either individually or as a set during initialisation.

3. The KD-Tree must deconstruct successfully, removing all nodes without leaving uncleaned memory behind. The KD-Tree will be reconstructed in full at the beginning of each frame hence it is not necessary however for the KD-Tree to be able to remove nodes from the tree one at a time.

4. The KD-Tree must have the functionality to, when given an arbitrary point in space, locate the nearest neighbouring point in the tree. Whilst doing so, the distance between these points must be calculated.

Given this specific set of requirements it was decided to instead develop a custom implementation of a three dimensional KD-Tree instead of importing a generalised variant from a third party library. The resulting code was developed using this article 'Multidimensional Binary Search Trees Used for Associative Searching' on KD-trees insertion and deletion algorithms [17], and this article 'An Algorithm for Finding Best Matches in Logarithmic Expected Time' on KD-tree's recursive nearest neighbour algorithm [25]. The resulting code comes in at just over 100 lines to perform the necessary functionality required by the drone platform's ICP implementation. Insertion of one node into the KD-Tree is completed in $O(log_2 n)$ time where n is the current number of nodes in the tree. Nearest Neighbour searching is completed in $O(log_2 n)$ and deconstruction completes in $O(n)$.

3.3.2 Calculating Best Fitting Rotation and Translation

Once corresponding sets are found using the nearest neighbour method, the centroid of each set is calculated. This centroid is then used to translate both the source point set and its corresponding destination points such that their centroids now lie over the origin.

Upon translating the source s and destination d point sets over origin by negating each point in the set by it's centroid, the best fitting rotation can be found. The article [16] presented a 3 by 3 matrix M whose elements are constructed via the following equation whereby s_i represents a point in the source point set and d_i represents a corresponding point on the destination point set.

$$M = \sum_{(} i = 1)^{n} s_i d_i^t$$

In representing this matrix M in the following fashion, a new 4 by 4 matrix N can be constructed from it's parts [16].

$$M = \begin{bmatrix} S_{xx} & S_{xy} & S_{xz} \\ S_{yx} & S_{yy} & S_{yz} \\ S_{zx} & S_{zy} & S_{zz} \end{bmatrix}$$

$$N = \begin{bmatrix} (S_{xx} + S_{yy} + S_{zz}) & S_{yz} - S_{zy} & S_{zx} - S_{xz} & S_{xy} - S_{yx} \\ S_{yz} - S_{zy} & (S_{xx} - S_{yy} - S_{zz}) & S_{xy} + S_{yx} & S_{zx} + S_{xz} \\ S_{zx} - S_{xz} & S_{xy} + S_{yx} & (-S_{xx} + S_{yy} - S_{zz}) & S_{yz} + S_{zy} \\ S_{xy} - S_{yx} & S_{zx} + S_{xz} & S_{yz} + S_{zy} & (-S_{xx} - S_{yy} + S_{zz}) \end{bmatrix}$$

From mathematical reasons relating to properties of quaternions discussed in this article [16], the unit quaternion that represents the solution to the least squares problem for rotation can be found as the eigenvector which corresponds to the maximum eigenvalue of matrix N. i.e. the unit quaternion

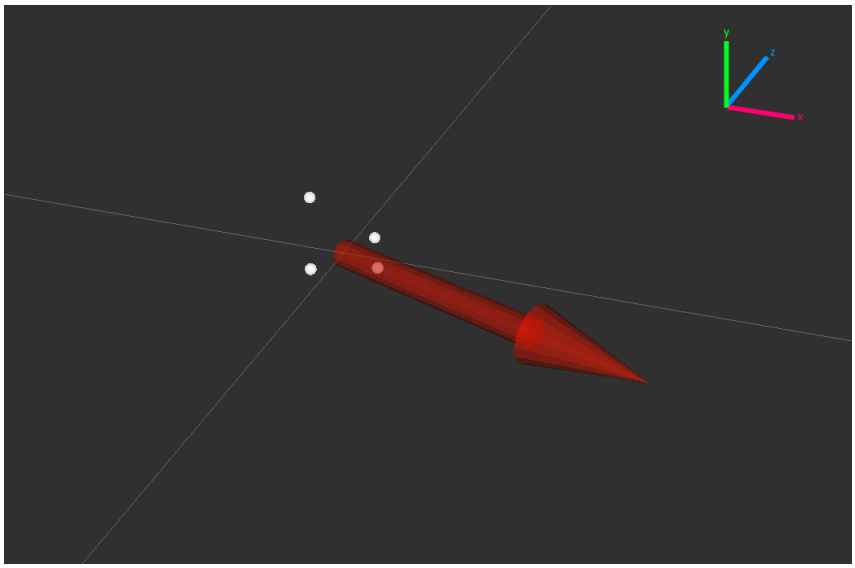

Figure 11: Demonstration of the platform's ICP implementation correctly registering the pose of the vflie marker cloud. The initial estimate in this example is a translation at origin and a rotation facing the positive x direction, where the real rotation of the drone is approximately 15 degrees off axis.

q who maximises the equation $q^t N q$ represents the best fitting rotation between the two point sets[16].

For mathematical calculations above, the platform made use of the Eigen third party C++ library [26]. Eigen provides a robust framework for mathematical calculations and considerable performance improvements for relatively complex mathematics tasks particularly those that involve matrices. The Eigen library has support for vectorisation and a variety of SIMD (Single Instruction, Multiple Data) computer instruction sets which have the possibility to vastly improve performance in tasks that involve executing the same instruction on a set of data. AS the platform has a clear requirement for performance in achieving the point set registration result, the use of a third party mathematics library can be highly benefitial.

Upon calculating the rotation the best fitting translation between the source and destination point sets for this iteration was found as the difference between the destination centroid and the rotated source centroid. Here the

best fitting solutions are combined with current estimates for translation and rotation to form the estimate for the next iteration of the ICP algorithm.

3.3.3 Implementation into the Drone Platform

Every frame, the pose of each drone must be recalculated using it's previous pose as the initial estimate. Additionally the way natnet_ros publishes point cloud frame data meant that the pose of all drones must be calculated at the same time. Hence the implementation for the ICP module was placed on the drone server instead of on the rigidbodies themselves.

In order to perform ICP pose calculations on a drone, the drone's marker template must be known before hand. The easiest way to do this is to require marker positions for each drone type to be hard coded into the wrapper class, which is then polled when need. This approach however goes against one of the platform's core goals in providing an easy to operate and flexible platform. The solution chosen was to instead rely on Optitrack's rigidbody system for initialisation of the platform's ICP implementation. For one frame on initialisation of a new rigidbody, it's first pose will be received from the marked rigidbody on Optitrack. Optitrack will also publish all the markers related to that rigidbody on a separate topic, this marker list (transformed back to origin based on the drone's rigidbody) will be saved as that drone's marker template. After this point the drone's pose will be determined based on the platform's ICP implementation. This method of collecting marker template information, and home position information is far simpler and more flexible for an end user requiring very little if any setup.

3.4 Analysis

3.4.1 Testing with the use of Vflie

Vflies work by simulating movement on wrapper internal vectors representing the drone's position and orientation. By modifying the vflie's wrapper such that it outputs a marker point cloud instead of it's internally calculated pose we are able to test the ICP implementation against a drone's 'true' position and orientation.

This pointer cloud was constructed from a hard coded set of 4 points centred around origin labelled as the drone marker template. Each frame the vflie's simulated translation and orientation is sampled and this transformation is applied to the vflie's marker template such that it is transformed to the drones position and orientation in world space. After which the resulting marker cloud is published to the relevant ROS topic to eventually be picked up by the ICP implementation model.

Upon receiving this marker cloud, the ICP module can determine the drone's position and orientation as if the cloud was being sent by Optitrack. The resulting Pose discovered by the icp implementation will then be published as the rigidbody's current pose. Testing in flight can then be done simply by running a test program through the drone platform.

Although this is a reasonable alternative to test the ICP implementation without the need of the Optitrack system, it does have considerable shortcomings. Due to the way natnet_ros publishes its point cloud information, emulating this process meant that only one vflie instance can publish its marker cloud at any one time.

Moving forward and once access to lab is restored, testing of the ICP implementation with the use of multiple marker sets on an Optitrack system will be conducted. As it currently stands there should not be any aspect of the

implementation that would complicate the registration of multiple drones on the Optitrack system however without being able to properly test this use case that is yet to be determined.

3.4.2 Evaluation of the Implementation

As it is only ever expected that a drone make a small movement from it's previous pose each frame (100Hz), the iteration count required to come within error tolerance generally does not exceed 2 iterations. This is where the implementation comes to a compromise between lowering the number of iterations required to reach error tolerance and temporal performance of the system. In order to determine error tolerance levels, the squared distance from each point to it's corresponding point is added. The process of first determining a source point's corresponding destination point requires iteration of the point cloud's kd-tree in a nearest neighbour calculation which has a big O complexity of $O(mlog_2n)$ where m is the number of source points and n is the number of destination points. And secondly a calculation for euclidean distance between each of those points. Considering the fact that these calculations make up a majority of the next iteration of the ICP algorithm, and iterations will only bring the pose closer to actual, the iteration is completed and the pre-iteration error value is compared.

Testing showed that on average the time to complete one iteration for a drone is approximately $0.8ms$. This number may seem exceptionally fast for the registration of a drone however considering the current implementation does not make any use of threading this value severly limits the number of drones on the platform. Given Optitrack frames are received at 100Hz, this gives the platform 10ms to complete all the tasks it needs to between Optitrack frames. If we allocate this entire period to registration of drones, this will limit the platform to at most 12 drones before time spent in registration begins to

impact platform performance. As mentioned earlier a possible solution to this problem is to split registration across multiple threads of which ICP appears to be a suitable task for.

In stress testing the implementation using a vflie with a marker template of 4 points, registration does fall into local minima issues when the drones real rotation is about 30 degrees off of the initial estimation pose. The result of this event occurring is an accurate translation element (as it is mostly based on each set's centroid) but a incorrect rotation element based on the template configuration. This issue is expected to alleviate with the introduction of more points in the marker template as it will then be more likely for each iteration to lead into the next until an actual minimum is achieved. That being said, the pose estimate each frame is not expected to reach nearly as high as 30 degrees from the actual drone rotation considering Optitrack's update rate of up to 120Hz.

An issue that was discovered in the making of this implementation is a limitation associated with the use of the natnet_ros package to simplify introduction of NatNet into the platform. Natnet_ros works by internally receiving information per frame from Optitrack's NatNet API and then packaging and sending that data over ROS topics to be picked up by the drone platform. The issue arises when we desire to publish virtual markers as part of a vflie's marker set to append the list of markers sent by natnet_ros. This task of appending a frame of point cloud data with virtual drone markers before that frame is read by the ICP implementation is not possible with the current setup. Hence the only option during testing at least was to disable natnet_ros' publishing of point cloud data and instead constructing the same pipeline out of one vflie. This issue cannot be resolved without moving the vflie marker publishing functionality out to the drone server and switching to Optitrack's supplied C++ NatNet API. This however is only an issue during testing as vflies are a special case in which they have the ability to independently publish their own poses

without needing to interact with some form of point set registration. As such in normal platform operation the vflie drone type will not be monitored by the platform's ICP system and instead they will publish their own poses directly to the relevant drone pose topics.

Regarding a possible extension to predict missing marker location based on a drone's recorded marker template and its last known pose and velocity. If for instance two of the four markers making up a drone's destination cloud are missing in a frame, the missing two markers can be estimated by projecting the drone's marker template over the destination cloud using the drone's known position and velocity. Using the 2 true marker positions and 2 predicted marker positions, the drone's new pose and rotation can be found however with some inaccuracy. Although this inaccuracy will exist due to marker estimation, following frames should be able to partially correct for the last such that an approximate position and rotation can be retained with only two new marker positions for a number of frames. This method is only possible as we can make assumptions on the flight dynamics of physical drones between frames, and the platform is able to approximate current position and orientation based upon calculated velocities. The extent to how much this can improve registration stability in lossy environments is an interesting extension to the platform's ICP module.

3.5 Conclusion

The point set registration implemented as part of this section is a first step to improving the reliability of the drone platform especially in circumstances where motion tracking exhibit lossy tracking environments. The solution provided real-time performant registration of a rigidbody from a template marker set able to form the basis for future work into using platform assumptions to improve registration reliability.

That being said the implemented point set registration method requires additional work before it can be relied on to be the sole provider of rigidbodies for the platform. The inability to perform testing using physical marker sets and an Optitrack environment means some unforeseen issues may still be present in the current solution. Additionally rotational flickering and local minima issues are witnessed in edge case scenarios that most likely will not be present during standard operation, however if this were to happen during the flight of a drone which required accurate motion capture to operate (such as the crazyflie and it's high-level commands) the result may cause loss of control in the drone during flight.

4 Platform Analysis

4.0.1 Platform Difficulties and Possible Improvements

The reliance on ROS did make development of the platform much easier, however it also means some additional difficulties are passed onto the user. These issues generally include some ROS environment setup, and launching of platform elements through ros terminal commands such as 'rosrun'. An improvement on this will be to automate away ros related setup and launch difficulties through for example bash programs.

The introduction of new drones to the platform is made as easy as possible for end users however there are aspects of this which the platform can not possible improve upon. For example in the introduction of the crazyflie drone, the biggest issue that still persists today has to do with aspects of the crazyradio and its communications with the drone itself. The crazyflie drone implementation in the platform today is adequately stable however getting it to that state required considerable work.

This problem can be better described by comparing the crazyflie to the Tello EDU. The crazyflie uses one USB dongle radio to communicate with a number of crazyflie drones, each of these drones have to be manually configured to work on the same network this step has to be done by the user. The Tello however requires a WiFi network to communicate with the drone platform. The Optitrack communications also require a WiFi connection to communicate meaning to communicate both with the Tello and the Optitrack software the user must manually add the drone to Optitrack's network or configure Linux to simultaneously communicate across two separate networks using two network adaptors. This unique, drone specific configuration cannot at least as far as I can tell be simplified by the platform.

Unfortunately it was not possible to fully test the platform towards the

end of development on physical drones due to the restrictions put in place with the threat of the Covid 19 virus. That being said all development done after these restrictions were rigorously tested on virtual drones which showed promising enough results to be considered functionally complete. Going forward there are plans to test all features developed during this period on physical drones when the chance arises and any changes that need to be made as found during this stage will be.

4.0.2 What Went Well

Once the platform has been setup, the writing of user api scripts to control drones on the platform is a really easy and rather fun activity. The user API, at least for all applications that were thought of during development, feels robust and is very stable even for highly dynamic systems.

The drone type wrappers provide easy modifications of the platform to add new drone types. The ease and robustness of this framework allowed even the development of the virtual drone type to be created almost entirely as a drone wrapper with little to no modification of the core platform.

All drones receive and act upon user API commands asynchronously from one another and the drone server. This means that delays or heavy processing that may be done by any one drone type do not impact the performance of any other drone in the system. Additionally the complexities of this detail are handled completely by the platform and do not require any attention by an end user.

5 Conclusion

The developed platform provides a safe, reliable, and enjoyable environment for the demonstration and testing of multi-drone formations programs.

Contributions to this platform have vastly improved accessibility through the introduction of a robust and reliable user API usable with both the MatLab and C++ programming languages. The development of a thorough drone wrapper framework allows the easy expansion of platform functionality and the late inclusion of abstract drone types. The introduction of the virtual drone class has allowed for extensive testing to be done on the platform regardless of inaccessibility of the Optitrack motion capture system or physical drones in the advent of the Covid 19 virus outbreak. As well as facilitate robust and costless testing functionality to researchers in the development of their drone formation algorithms. Lastly the development of an internally operating point set registration algorithm has allowed the first steps to be taken to improving the reliability of the platform in case of any motion capture environment issues that may otherwise invalidate operation.

With the addition of contributions made by co-contributors to the platform from safeguarding measures to debugging and logging capabilities, the developed platform provides the functionality as described by the initial project requirements.

References

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system,"

[2] "Documentation - ros wiki." http://wiki.ros.org/, 2018. [Accessed 30 Oct 2019].

[3] A. M. Aurand, J. S. Dufour, and W. S. Marras, "Accuracy map of an optical motion capture system with 42 or 21 cameras in a large measurement volume," *Journal of Biomechanics*, vol. 58, pp. 237 – 240, 2017.

[4] "Optitrack documentation wiki." https://v21.wiki.optitrack.com/index.php?-title=OptiTrack_Documentation_Wiki, 2019. [Accessed 24 Oct 2019].

[5] R. M. Taylor, II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser, "Vrpn: A device-independent, network-transparent vr peripheral system," in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '01, (New York, NY, USA), pp. 55–61, ACM, 2001.

[6] W. Giernacki, M. Skwierczyński, W. Witwicki, P. Wroński, and P. Kozierski, "Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering," in *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*, pp. 37–42, IEEE, 2017.

[7] "Tello sdk 2.0 user guide." https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf, 2018. [Accessed 30 Oct 2019].

[8] J. A. Preiss, W. Honig, G. S. Sukhatme, and N. Ayanian, "Crazyswarm: A large nano-quadcopter swarm," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3299–3304, IEEE, 2017.

[9] "Python api - crazyswarm 0.3 documentation." https://crazyswarm.readthedocs.io/en/latest/api.html, 2018. [Accessed 30 Oct 2019].

[10] J. Noronha, "Development of a swarm control platform for educational and research applications," Master's thesis, Iowa State University, 2016.

[11] "Flytos: Operating system for drones." https://flytbase.com/flytos/, 2019. [Accessed 30 Oct 2019].

[12] L. Meier, D. Honegger, and M. Pollefeys, "Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *2015 IEEE international conference on robotics and automation (ICRA)*, pp. 6235–6240, IEEE, 2015.

[13] A. Chakravarthy and D. Ghose, "Obstacle avoidance in a dynamic environment: A collision cone approach," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 28, no. 5, pp. 562–574, 1998.

[14] B. Salau and R. Challoo, "Multi-obstacle avoidance for uavs in indoor applications," in *2015 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pp. 786–790, IEEE, 2015.

[15] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Sensor fusion IV: control paradigms and data structures*, vol. 1611, pp. 586–606, International Society for Optics and Photonics, 1992.

[16] B. K. Horn, "Closed-form solution of absolute orientation using unit quaternions," *Josa a*, vol. 4, no. 4, pp. 629–642, 1987.

[17] J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[18] T. Zinsser, J. Schmidt, and H. Niemann, "A refined icp algorithm for robust 3-d correspondence estimation," in *Proceedings 2003 International Confer-*

ence on Image Processing (Cat. No.03CH37429), vol. 2, pp. II–695, IEEE, 2003.

[19] S. Rusinkiewicz and M. Levoy, "Efficient variants of the icp algorithm," in *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, pp. 145–152, IEEE, 2001.

[20] H. Pottmann, S. Leopoldseder, and M. Hofer, "Registration without icp," *Computer Vision and Image Understanding*, vol. 95, no. 1, pp. 54–71, 2004.

[21] K. S. Arun, T. S. Huang, and S. D. Blostein, "Least-squares fitting of two 3-d point sets," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-9, no. 5, pp. 698–700, 1987.

[22] K.-L. Low, "Linear least-squares optimization for point-to-plane icp surface registration," *Chapel Hill, University of North Carolina*, vol. 4, no. 10, pp. 1–3, 2004.

[23] M. W. Walker, L. Shao, and R. A. Volz, "Estimating 3-d location parameters using dual number quaternions," *CVGIP: Image Understanding*, vol. 54, no. 3, pp. 358–367, 1991.

[24] L. Hongbin and L. Bin, "Research on a novel 3d point cloud robust registration algorithm," in *2010 International Conference On Computer Design and Applications*, vol. 5, pp. V5–441, IEEE, 2010.

[25] J. Friedman, J. Bentley, and R. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[26] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3." http://eigen.tuxfamily.org, 2010.

A Drone Platform User API Example Script (C++)

```
// this script takes in two drones, one which will follow a set of waypoints (↩
    base drone), and another which will circle (perform donuts) around the ↩
    base drone during it's flight. This script is intended to demonstrate ↩
    dynamic flying conditions, waypoint navigation, and showcases the user ↩
    API.

// include the multi drone platform's user API header file
 #include "user_api.h"

void do_donuts(mdp::id baseDrone, mdp::id donutDrone) {
    // first takeoff the two drones simultaneously
    mdp::cmd_takeoff(baseDrone);
    mdp::cmd_takeoff(donutDrone, 1.0);

    // then wait until both drones have finished taking off
    mdp::sleep_until_idle(baseDrone);
    mdp::sleep_until_idle(donutDrone);

    // this array holds the waypoints the base drone will follow, each ↩
        position has an absolute x, y, and z component.
    std::array<std::array<double, 3>, 5> positions = {
            {{-0.5, 0, 0.0},
            {0.0, -0.5, 0.0},
            {1.0, 0.0, 0.0},
            {0.0, 0.5, 0.0},
            {-0.5, 0.0, 0.0}}
    };

    // create a position message to be used by the base drone, then fill it ↩
```

```
    with the first waypoint
mdp::position_msg baseMsg;
baseMsg.relative = false;
baseMsg.keepHeight = true;
baseMsg.position = positions[0];
baseMsg.duration = 6.0;
baseMsg.yaw = 0.0;

// get the current position of the base drone and place it into ↩
    baseDronePos
auto baseDronePos = mdp::get_position(baseDrone);

// create a position message to be used by the donut drone, then fill it ↩
    with the x and y location of the base drone (where z is up). As keep ↩
    height is set to true, this drone will remain at its current z height ↩
    but move to the specified x and y
mdp::position_msg donutMsg;
donutMsg.relative = false;
donutMsg.keepHeight = true;
donutMsg.position = {baseDronePos.x, baseDronePos.y, 0.0};
donutMsg.duration = 3.0;
donutMsg.yaw = 0.0;

// set the position of the donut drone to above that of the base drone ↩
    using the constructed message
mdp::set_drone_position(donutDrone, donutMsg);
// wait until the donut drone arrives
mdp::sleep_until_idle(donutDrone);

donutMsg.duration = 0.5;
```

```
// wait for next quantised frame
mdp::spin_until_rate();

// perform the following tasks until base drone runs out of waypoints to follow
for (size_t i = 0; i < positions.size(); i++) {
    // update base drone's position message with the new waypoint
    baseMsg.position = positions[i];
    // set base drone's positon to the updated position message
    mdp::set_drone_position(baseDrone, baseMsg);
    // wait until next quantised frame
    mdp::spin_until_rate();

    // while the base drone is moving towards it's next waypoint do the following in the mean time
    while (mdp::get_state(baseDrone) != mdp::drone_state::HOVERING) {
        // get the current position of the base drone
        baseDronePos = mdp::get_position(baseDrone);

        // calculate using sin and cos the circle around the base drone that donut drone should follow
        double timeNow = ros::Time::now().toSec();
        double donutX = sin(timeNow) * 0.6;
        double donutY = cos(timeNow) * 0.6;

        // update donut drone's position message with it's new desired location
        donutMsg.position = {baseDronePos.x + donutX, baseDronePos.y + donutY, 0.0};
        // set the position of the donut drone to the updated message
        mdp::set_drone_position(donutDrone, donutMsg);
```

```
            // wait until the next quantised frame
            mdp::spin_until_rate();
        }
    }

    // once completed, send both drones to go to their home positions ↩
        simultaneously
    mdp::go_to_home(baseDrone);
    mdp::go_to_home(donutDrone);

    // wait until both drones have finished going home and landed
    mdp::sleep_until_idle(baseDrone);
    mdp::sleep_until_idle(donutDrone);
}

int main() {
    // first initialise the user api program
    mdp::initialise(10, "donut_test_program"); // update rate (quantised ↩
        frames) set to 10Hz

    // get a list of all the active drones on the platform
    auto drones = mdp::get_all_rigidbodies();

    // if there are over two drones active, then run the do_donuts script
    if (drones.size() >= 2)
        do_donuts(drones[0], drones[1]);

    // conclude the api program and return.
    mdp::terminate();
    return 0;
```

```
}
```

B Encoding of user API into standard ROS messages

```
class id {
    private:
        bool owner = false;
        std_msgs::Header* data;

    public:
        id() {owner = true; data = new std_msgs::Header;}
        id(std_msgs::Header* header) {data = header;}
        ~id() {if (owner) {delete data;}}

        uint32_t& numeric_id() {return data->stamp.sec;}

        std_msgs::Header get_data() {return *data;}
};

class input_msg {
    private:
        geometry_msgs::TransformStamped* data;

    public:
        input_msg(geometry_msgs::TransformStamped* transform) {data = ↩
            transform;}
        ~input_msg() {}

        id drone_id() { return id(&data->header); }
        std::string& msg_type() { return data->child_frame_id; }
        geometry_msgs::Vector3& pos_vel() { return data->transform.↩
            translation; }
```

```
        double& relative() { return data->transform.rotation.x; }
        double& target() { return data->transform.rotation.y; }
        double& yaw_rate() { return data->transform.rotation.z; }
        double& yaw() { return data->transform.rotation.z; }
        double& duration() { return data->transform.rotation.w; }

};
```