Final Year Project Report

Full Unit - Final Project Draft

Intelligent and Autonomous Micro Aerial Vehicles

Gerald Elder-Vass

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science (w\Artificial Intelligence)

Supervisor: Sara Bernardini

Department of Computer Science
Royal Holloway, University of London

March 17, 2018

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 18180

Student Name: Gerald Elder-Vass

Date of Submission:

Signature:

Table of Contents

Abstract

The goal of this project is to develop techniques to underpin autonomous and intelligent behaviour in micro aerial vehicles.
In recent years, there has been growing interest in unmanned aerial vehicles (UAVs), also called drones. A drone is any aircraft that is operated without the possibility of direct human intervention from within or on the aircraft. The lack of an on board pilot allows drones to exist in a variety of sizes, from micro UAVs (called Micro Aerial Vehicles, MAVs) to large aircraft. UAVs are particularly useful in those situations in which it would be difficult or dangerous to send humans, e.g. in sites with nuclear radiation contamination or in close proximity to wildfires. In addition, MAVs, thanks to their small size and aerodynamics characteristics, can be used in scenarios that are inaccessible to large planes, such as cluttered outdoor settings and indoors, where they can provide unique viewing angles at low altitude. In this project, we focus on the development of autonomous MAVs, which are MAVs that function without constant human intervention. Instead of being tele-operated by a human operator, these drones are capable of flying autonomously and executing a high-level action plan. In comparison with other types of robots, MAVs present unique characteristics that make devising algorithms for autonomy particularly challenging. First, these vehicles are difficult to control as they are inherently unstable systems with fast dynamics. Second, given their small dimensions and light weight, MAVs have a restricted payload, i.e reduced computational power as well as noisy and limited sensors. Third, the life of a MAV battery is usually short and allows continuous flight for a limited period, ranging from a minimum of a few minutes to a maximum of around two hours. Despite these challenges, autonomy is considered crucial in unlocking the full potential of drones in a number of critical applications, such as disaster response, surveillance operations, exploration, weather observation and other important military and civilian missions. In this project, the student will first choose a specific task for the drone that demonstrates autonomous behaviour and then will implement such a behaviour in simulation and with a real drone (if available). Examples of tasks that require autonomy are the following: tracking of a moving target, navigation with obstacle avoidance, landing on a moving platform, circling around an object, executing figure flying and searching for a lost target.

Rationale

The aim of this project is to develop a ROS package which will control a drone causing it to follow an object as it moves around a simulated environment. This project will use techniques for analysing drone camera output and using the resulting data to make decisions on how to move around the environment in order to keep vision of the object. Autonomous navigation is a large and complex area and the first steps are the hardest, and so this project will be focusing on how the AR Drone can be controlled with communication via ROS topics in a simulation controlled by Gazebo.
The behaviour of following an object can be used in many environments for a number of similar purposes. Tracking and following an object can be used for filming and recording actions in hands-free situations or for providing vision in areas unsuitable for humans (e.g. high radiation zones/war zones).
An example of a similar usage of drones in sport is the HEXO+ drone[1] which uses GPS tracking on smart phones to follow targets[2]. Using GPS tracking allows the drone to track and follow a smart phone however there are situations in which having a smart phone or tracking device on the target is not ideal. Smart phones can be bulky and disruptive to ability (especially modern smart phones which are progressively increasing in size), in many activities where a user wants to be tracked using the drone the smart phone may add unnecessary weight or be impractical for the user to have on their person. In this case the drone would need another form of target tracking in order to avoid inhibiting the user's ability while providing a similar level of feedback. This, in part, inspired this project to use computer vision and object recognition techniques in order to track objects in motion.
Similar work has been completed for ground based drones, although this is not effective for tracking objects in scenarios with any uneven surface or on surfaces for which the drone cannot get enough traction to move itself effectively. Furthermore it limits the objects which can be track to ground based objects which have no chance of accelerating vertically - which means tracking a ball rolling along the floor is viable, but tracking a ball which may be picked up or kicked is unviable as the drone would lose sight of the ball and be unable to find it. With an aerial drone the object can not only be tracked and followed in three dimensions, the drone can easily rotate while circumnavigating the object without losing sight of it as it can move horizontally while rotating. An example of a ground based ball tracking drone is this video of the Jumping Sumo drone[3].

Review of another Auto Pilot System

During research for this project I came across a GitHub repository for an Auto Pilot function (named Auto Pylot) for the Parrot AR Drone 1.0 & 2.0[4]. After reviewing the package structure I discovered some limitations of the repository, my primary concerns and thoughts are discussed here:

The repository is rather unclear on how it should be used or what systems it has been designed to be used with (other than the drone itself). It provides functionality for use with physical controllers to interact with the drone and activate the auto pilot functionality but this suggests the code is unusable without a physical controller. (This is understandable as you want to be able to stop the drone should something go wrong, but in some situations you may not be able to have a controller if you're using the drone to record yourself.)

The source code does not provide the option for developed code to be tested in a simulated environment. In software development where the product will be used in real world

scenarios (especially with physical machines such as drones and cars), it is important that code is properly tested in a range of environments in case the behaviour of the code changes dramatically due to the environment changes around it. You can imagine the consequences of developing a self driving car or an object tracking drone directly in the real world - collisions and mistakes would initially be common and their effects could have huge impacts on the world around them. It is important that when code is developed in a simulated environment that it is directly transferable into the real world scenarios - this means if a drone can be developed in a simulated environment there should be no changes made to the code to use it in the real world, optimally a developer would simply change the I/O feeds of the system.

Finally it is unclear how a developer or user can use the source code provided and build on it to develop their own behaviours, the repository code does not appear to be easily re-usable. While the code clearly provides an effective tracking of a green ball and the following of the ball via the AR Drone - it appears that this is all the code can do (with the exception of the controller support and toggling the auto pilot functionality). Should developers want to create a drone behaviour which tracks different objects or even just a different coloured ball - the developers would either have to understand the inner-workings of this repository or re-write a large amount of the code in order to change the behaviour. Developers also wouldn't be able to control how the drone follows the object nor the speeds at which the drone reacts (although typically the speed is based on how far the ball is so this wouldn't impact the developer as much). A related criticism is that it is unclear how the code base interacts, both with itself and with the AR Drone, especially for someone who is not overly familiar with the C++ language, which makes it hard to use this code for insight into how the system works - as a result the work in this repository hasn't affected how I have developed my packages other than the video of it in action which represents an identical goal and that my packages should be much more readable and usable.

These flaws further inspire the development of a package which is re-usable and easily understandable by developers and users alike. It is also important to create simulation environments which can test the packages and the drone's behaviour effectively before the behaviours are used in real world scenarios.

Project Aims

After researching the area of drone vision and other systems as discussed above, the aims of this project are:

- Develop a package which provides the autonomous navigation of an AR Drone to follow an object.
- Provide easily readable and reusable code.
- Develop code which functions effectively for usage in both simulated and real-world environments.
- Provide functionality that allows users to change how the AR Drone follows it's target.

Chapter 1: **Theory**

1.1 ROS Report

1.1.1 Overview of ROS

ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more.[1]
ROS[5] is a framework for developing robotic applications, it uses topics, nodes, services and messages in order to create an environment where processes have access to the information they need and can act on the robot itself. ROS offers subscribers and publishers to ensure communication is not wasteful and that nodes do not receive unnecessary messages. ROS Packages can be downloaded to expand the functionality from the ROS Package Index[6].

1.1.2 Understanding of ROS

ROS consists of nodes, topics, messages and services to provide it's functionality.
A message is the simple transfer mechanism for information, each message has a type which consists of other message types and primitive types. Messages are published to topics from the terminal and from Python scripts/C++ programs.
A service is a request for information mechanism, similar to a function call in a standard programming language. A service consists of sending a message with the information required to generate the response which is then returned to the initial node.
A topic represents a subject or element which information can be expressed about. An example of a topic could be the velocity of a vehicle.
A node represents a script or program running which can subscribe and publish to topics, it can also request services. If a node has subscribed to a topic then every message sent to that topic will call a callback function with the message as the parameter. If a node is publishing to a topic then the message published will be sent to all subscribers. Publishing a message is like declaring information to nodes which have stated they are interested by Subscribing, such as publishing the location of an object which has been identified.

1.1.3 Applications of ROS

ROS will be used to ensure that data streams are correctly directed throughout the system, such as camera data to an analysis node or drone commands to the drone itself. Only nodes which need specific pieces of information will be informed of any data changes, this saves wasteful data transfer and processor cycles on messages which are irrelevant to the node's behaviour. ROS should be used to minimize messages and data streams to increase performance as robots tend to need to make rapid decisions based on the most recent accurate data. ROS nodes are designed such that there can be multiple nodes of the same type with different names to ensure that they can interact with multiple targets through the same scripts and programs. This means if you were analysing an image for two different objects, you could write one node which identifies an object and use the Parameter Service (see below - ROS Parameter Server) to specify which node instance should identify which object.

[1] A description of ROS taken directly from the ROS website [5]

1.1.4 Advanced Features of ROS

ROS provides a number of advanced features such as launch files and a Parameter Service. ROS launch files are used to run multiple processes or ROS commands from a single file, which makes it clear how a number of systems are interacting. The launch file specifies other ROS launch files and ROS nodes to run, it can even call various ROS commands such as *rostopic* and *rosparam*. An example launch file could start a simulation and spawn various objects into the simulation.
Launch files are written in XML and ROS provides a number of unique tags which allow the user to control how the system is initially setup without having to open multiple terminals or issue multiple commands.

ROS provides interaction with a Parameter Service via the *rosparam* command. The Parameter Service is a system for passing arguments/parameters to the ROS system, which allows a user to change the behaviour of a node by changing the parameter values. The rosparam command is also made available as an XML tag which means that users can create launch files which change the parameters and therefore behaviours of various nodes.
This feature allows developers to create nodes which can be used for a wider range of scenarios instead of providing a minimal level of functionality and expecting users to implement their own higher level nodes. If all packages were developed effectively using the ROS Parameter Service then if enough relevant packages had already been developed - a user would be able to combine them in a way which achieves a goal without having to write any more than a launch file. An example of this would be a user developing a launch file which defines ROS nodes in order to control a vehicle to navigate to a destination - the user could write a launch file and set parameters which specify the type of vehicle, how the route should be calculated, how the position of the vehicle should be evaluated and how responsive the vehicle is to following the calculated route to the destination.

1.2 Gazebo Report

1.2.1 Overview of Gazebo

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. At your fingertips is a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.[2]
Gazebo[7] is a simulation software with an accurate physics engine allowing users to create simulations of large environments. Gazebo allows users to change the world to meet specific requirements which ensures that users are able to create the perfect environment to develop and test their work. It is popular for testing and demonstrating robotic behaviour due to the complete control a user has of the world and it's easy interaction with ROS via a gazebo_ros package.

1.2.2 Applications of Gazebo

I will be using Gazebo to create environments for the AR Drone to interact with. I will make use of Gazebo's utilities to begin testing with simplistic environments where objects cannot be mistaken for other objects and show control over the drone's motion. This allows me to build up more complex real-world simulations after solving the problems involving computer vision before; in a simple simulation.
Gazebo offers a range of functionality which will greatly improve testing and development, as opposed to developing directly in the real world. It will also prove useful in decreasing the time to setup simulations due to the ability to quickly develop objects and change attributes such as colour and size in moments, which can then be saved for the next simulation.

[2] A description of Gazebo taken directly from the Gazebo website[7]

1.3 OpenCV Report

1.3.1 Overview of OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source BSD-licensed library that includes several hundreds of computer vision algorithms.[3]
OpenCV[8] is a software library for computer vision. It offers a range of algorithm implementations to allow users to manipulate and analyse images in many ways. Due to the range of computer vision algorithms available; the library can be used to analyse cameras in real time or perform manipulations on images where time is not a major constraint.

1.3.2 Applications of OpenCV

OpenCV will be used in the project to analyse the output of the drones cameras. It will be used to recognise shapes and templates from the raw camera imagery in order to provide information for the ROS nodes to decide how to react. The processing of camera data will be performed in a ROS node and published to various topics which will ensure there is a high level of cohesion between nodes. ROS nodes will compute locations, accuracy and confidence in camera data and publish custom messages which relay the analysed information. We can then use this data via subscribers in other nodes to make the decisions which control the drones behaviour and publish control messages to the drone topics.
This separation of perception of the world and the decision making process allows us to easily debug and test nodes while they are being written and make any suitable changes once development has been completed.

1.3.3 What is Template Matching?

Template Matching (sometimes called Pattern Matching) the process of identifying an image (called a template) in a larger image (referenced here as a source image). Typically a basic implementation of Template Matching will compare the template to every possible location in the source image. This can make running times vary dramatically on a number of parameters. For example, if the template image is small; each comparison will be faster as there are fewer pixels to compare per match attempt but there will be more match attempts as there are more places in the source image in which the template could match. If the source image is smaller then there are fewer places the template could match, which makes the check faster. Template Matching will typically search for the best fit of the template image in the source image. This means, if the source image size does not change between comparisons (as holds when using a camera), the run time of the check will be constant. Of course the Template Match could fail to find a match in the source image which would also have an identical run time, it would simply return without result (or an empty list). In some cases Template Matching will search for multiple instances of the template in the source image which means the entire source image will need to be checked, once again, this leads to an identical run time as above. In rare cases Template Matching could simply match the first area in the source image which is above a certain threshold of similarity/certainty, in which case the run times vary between the duration of a single template check and the run time as above for checking the entire source image.
In short, no matter how the Template Matching algorithm is running, there's no reason to worry about which mode it's run in or whether or not it will find a match at all as the

[3] A description of OpenCV taken from the OpenCV website[8]

run time is not going to change in either scenario. One just has to ensure the run time is acceptable for the task at hand and that it will not change.

1.3.4 Implementation of Template Matching

OpenCV's implementation of Template Matching has six modes of execution based on the algorithm's available for the task. The fastest of these methods is CV_TM_CCORR however this is not of an acceptable reliability as shown in many online examples[10]. The fastest method with an acceptable reliability is CV_TM_CCOEFF. The calculation for Template Matching for each method can be seen in the relevant OpenCV documentation[11].

1.3.5 Limitations of Template Matching in OpenCV

Unfortunately template matching will only use the exact template supplied regardless of which method of comparison used. This means that if the task is to search for an object or within a source image, one would need to know exactly how far away they expect the object to be and have a template of the object at the given distance and angle of vision (e.g. from above if looking from above).

1.3.6 Planned Usage of OpenCV Template Matching

OpenCV's Template Matching can still be used considering the limitation outlined above. My solution to this problem is to use a solid colour circle to represent a target where different colours can be used to represent different objectives. At a basic level for example, a blue circle should be avoided where a red circle could represent a landing area or waypoint. To solve the issue of distances and the size of the template, a single template of relatively large size will be stored locally, copies will be made on various scales and multiple ROS nodes will process the Template Match for each scale of the template and publish to a topic with any matches they make. This will allow me to analyse the camera images in real time while decreasing the probability of missing an object.

1.3.7 What is Hough Circle Detection?

Hough Circle Detection[12] is an algorithm designed to identify circles, or partial circles, from a grey scale edges image. In an edges image, each pixel is either black or white where the white pixels represent the edges of a shape. Hough Circle Detection is based on the number of white pixels (edges) the same distance from a point on the image. Using a three-dimensional array with dimensions; image width, image height and radius range, the algorithm will use every possible pixel as a potential center of a circle and for each radius in the radius range to check all of the edges which are exactly 'radius' distance away from the center pixel. When an edge is detected at this point, the array value associated with this particular circle center and radius is incremented which acts as a counter for the number of edges which suggest a circle has been found. Once this has been performed across the entire image for all radii in the range, all of the array values which are above a certain threshold (a confidence value) will be treated as located circles, where the higher the counter is: the more certain the algorithm is that a circle should exist at that center with that radius. This method of detection can potentially take a long time, scaling with image size and radius range.

1.3.8 Planned Usage of OpenCV Hough Circle Detection

In the final deliverable I plan to use Hough Circle Detection to locate a ball in an image from the drone's camera. I will be using the OpenCV implementation[13] of the algorithm, specifying the parameters for it's order of operation. As stated before, the runtime of this algorithm scales with image size and radius range. In this project the image size will always be 640x320 pixels as obtained from the drones camera output and I will use a range of 3-100 pixels as the radius range, this allows me to detect a ball when it is far away and very near. While this range includes a large number of values, it has been limited to this size in an attempt to reduce runtime, if the drone is tracking the ball effectively then the ball should never move far away enough for its radius to be smaller than 3, likewise the ball should never move near enough for its radius to be greater than 100.
One could consider using a range of radii which steps between each value, such as starting with a radius of 10 pixels and then checking a radius of 12 pixels (a step size of two). However while this would improve the time efficiency of the algorithm, it would not accurately identify circles with a radius of a 'stepped' value - if the step was two and the smallest radius was 10, then the algorithm would fail to detect a circle with a radius of size 11 if the edges were particularly fine in width. The larger the step difference got - the less reliable the detection algorithm would be.

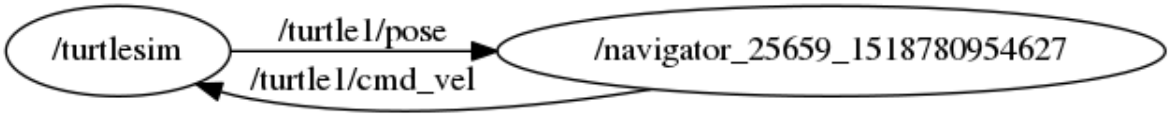
Chapter 2: **Programs**

2.1 Turtle Navigator

The first deliverable program written was the Turtle Navigator. The Turtle Navigator is a ROS package I developed in order to move the turtle from the ROS turtlesim tutorial to a target location. The purpose of this package was to show my understanding of the ROS framework. I made use of all aspects of ROS to show my understanding of them and how they are used together.
The publisher and subscriber system was very easy to use and develop for, the topics were already created by the turtlesim tutorial which meant I could interact with the topics easily. As the purpose of this deliverable was to experiment with and demonstrate my understanding of the ROS system; the navigational piece of code calculated the mathematics for Z-axis rotation was taken from ROS TurtleSim Go To Goal Tutorial, although the rest of the code is my own - it tends to have a similar feel to the tutorial as this was how the previous tutorials suggested how a node script should be written in Python.
I developed my own service called calcDistance which would calculate the distance between two points allowing me to change how the turtle moved based on distance. The implementation is a simple Euclidean distance calculation as you can see below.

```
#!/usr/bin/env python

from turtle_navigator.srv import *
import rospy
import math

def handle_calc_distance(req):
    dx = req.target.x - req.source.x
    dy = req.target.y - req.source.y
    distance = math.sqrt((dx ** 2) + (dy ** 2))
    return calcDistanceResponse(distance)

def calc_distance_server():
    rospy.init_node("calc_distance_server")
    s = rospy.Service("calcDistance", calcDistance, handle_calc_distance)
    rospy.spin()

if __name__ == "__main__":
    calc_distance_server()
```

This diagram is generated using the *rqt* command which generates graphs showing the flow of information between nodes and topics for ROS systems. This diagram represents the communication between nodes for the Turtle Navigator package.



2.2 Gazebo AR Drone Simulation (Empty)

Next I needed to see how a user could interact with the AR Drone, so using an empty world I created an AR Drone in Gazebo. Eventually, using the documentation, I figured out how to make the drone take off which led the the drone continually rising so high that it was no longer visible in the simulation. I discovered that some drivers were missing which made this step in the development process save me a lot of time; later I could have written a more complex package which failed for driver reasons but luckily it was spotted here first. After solving the driver issue I managed to get the drone to take off and hover a few meters from the floor.
I was then able to experiment with changing the drone's velocity and getting feedback from the front and bottom cameras. I discovered that the drone's control was not completely stable; after issuing a command for the drone to move forwards for a short duration and then stop it would attempt to hover but drift slightly in a random direction. I decided that this was simply due to balancing issues in the drone as opposed to a serious issue with the software. Once the drone is able to react to its environment, such as following an object, this drift should effectively disappear, or at least be accounted for, as when the drone drifts to one side it will realise that it is further from the object than it should be and correct itself.
Finally I had the drone land smoothly from various heights and places to see how it would react; in each case the drone smoothly descended until coming to a halt as it rested on the ground.

2.3 Gazebo AR Drone Take Off and Land

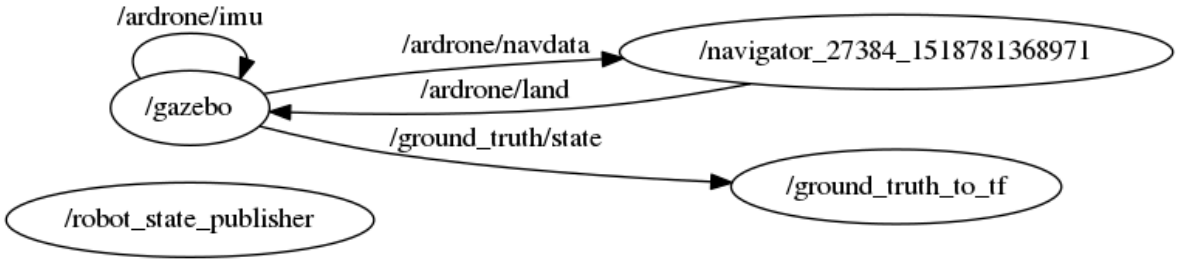
Now I was ready to start experimenting with ROS packages and the AR Drone. I once again created an empty world in Gazebo and spawned a drone in the center. The purpose of this package would be to show interaction with the drone from a package and so the behaviour would be trivial. I simply had the package wait for user input before taking off or landing on the spot. While this deliverable was beyond trivial, it fulfilled it's purpose of ensuring I could interact with the drone properly from a ROS package. This leads the way to develop ROS packages with more interesting behaviours.
While the code for this package would appear to be incomplete, it is the core code for navigation in an environment where we get feedback from the simulation environment. It's best to consider this as an AR Drone equivalent to the Turtle Navigator package. Unfortunately, this kind of feedback is not available from the simulation as the only input the AR Drone will receive is image information from the two cameras, but I've left the code in this way to show how the transition was made and that I can create subscribers and publishers for the new, more complex nodes.

```
<-- Import Statements -->

class Navigator(object):

  def __init__(self):
    rospy.init_node("navigator", anonymous=True)

    self.rate = rospy.Rate(10)
    self.sleepDuration = rospy.Duration(1)
    self.navdata = Navdata()
```

```
    self.emptyMessage = Empty()
    self.landPublisher = rospy.Publisher("/ardrone/land",
                                         Empty,
                                         queue_size=5)
    self.takeOffPublisher = rospy.Publisher("/ardrone/takeoff",
                                            Empty,
                                            queue_size=5)
    self.velPublisher = rospy.Publisher("/ardrone/cmd_vel",
                                        Twist,
                                        queue_size=10)

    self.positionSubscriber = rospy.Subscriber("/ardrone/navdata",
                                               Navdata,
                                               self.positionUpdate)

  def land(self):
    self.landPublisher.publish(self.emptyMessage)
    rospy.sleep(1.)

  def takeOff(self):
    self.takeOffPublisher.publish(self.emptyMessage)
    rospy.sleep(1.)

  def navTo(self, tx, ty, tz):
    pass

  def atTarget(self, tx, ty, tz):
    return True

  def positionUpdate(self, navdata):
    self.navdata = navdata
    print("Navdata: " + str(self.navdata.magX))

  def callback_bottom_cam(self):
    pass

  def callback_front_cam(self):
    pass

  def navigate(self):
    while not rospy.is_shutdown():
      tx = input("X: ")
      ty = input("Y: ")
      tz = input("Z: ")

      self.takeOff()

      while not self.atTarget(tx, ty, tz):
        self.navTo(tx, ty, tz)
        self.rate.sleep()

      self.land()

if __name__ == "__main__":
```

```
nav = Navigator()
nav.navigate()
```

Once again the diagram below was generated using the *rqt* command. The graph shows a more complex interaction between nodes and topics as the simulation world has more information to publish. There is a missing connection on this graph as the navigator node should also be publishing to the /ardrone/takeoff topic (which is subscribed to by the /gazebo node), but evidently the graph drawing tool only draws on arrow between nodes.



2.4 OpenCV Work

Before I could jump into using the camera (other than simply viewing it while running a simulation), I would need to be able to interpret the images the camera gave me in some way. To do this I wrote a simple Python program which would use an image and a template from the image to create a series of differently sized templates using OpenCV's basic scaling operations. Each template would be matched in the original image. You can see the results below (the scales of images are [0.5, 1, 1.5, 2], the co-ordinates are the location the template was matched to, and the time is in seconds).

```
Template: (2444, 528) Time to find template: 0.13109087944
Template: (2369, 598) Time to find template: 0.142675161362
Template: (1635, 462) Time to find template: 0.124691963196
Template: (1957, 657) Time to find template: 0.122797012329
```

Interestingly the original template, of scale 1, took the longest to match however the match was much more accurate than any of the others.
Based on the results I have concluded that the most accurate way of locating the template in the image is to find the most similar scale to the image and trust the location calculated from that template match. While this sounds obvious to the casual reader; the usage of the template matching is not to work out how far away an object is, the purpose is to locate an object so the drone can steer towards or away from it if necessary. While distance to an object is going to be important later on, for now direction is enough.
Below I've included the code for the template matching exercise:

```
import time
import cv2
import numpy as np

img = cv2.imread("trampolining.jpg", 0)
img2 = img.copy()
template = cv2.imread("template.jpg", 0)
```

```
scales = [0.5, 1, 1.5, 2]
templates = [None for i in range(len(scales))]
for index in range(len(scales)):
  temp = template.copy()
  templates[index] = cv2.resize(temp, None,
                                fx=scales[index], fy=scales[index],
                                interpolation=cv2.INTER_CUBIC)


def apply_template_matching(img, template):
  res = cv2.matchTemplate(img2, template, cv2.TM_CCOEFF)
  min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

  top_left = max_loc

  w, h = template.shape[::-1]
  center = (top_left[0] + (w / 2), top_left[1] + (h / 2))

  return center

def time_matching(img, template):
  start = time.time()
  center = apply_template_matching(img2, template)
  end = time.time()
  runtime = end - start
  print("Template: " + str(center[0]) + ", " + str(center[1]) +
        "\tTime to find template: " + str(runtime))

if __name__ == "__main__":
  for template in templates:
    time_matching(img2, template)
```

2.5 Gazebo AR Drone Land On Landing Pad

I was finally ready to begin working on a drone simulation using template matching for recognising a landing pad. The simulation was minimalistic, it consisted of a landing pad and the drone itself, the landing pad was simply a red oblong large enough for the drone to land on with a bit of extra space in case the landing was off by a small amount. For this project I created two important nodes; the landing pad identifier node and the controller node. The landing pad identifier is a node developed to convert the Image messages from the ROS topics the bottom drone camera publishes to into Open CV compatible images, using a ROS package called cv_bridge[15] and then applies template matching with a pre-defined image of the landing pad to work out if it is below the drone or not. Unfortunately the way the template matching system works suggests that the template doesn't have to match an exact size, this means that if the landing pad is a red square from above - when the drone passes over a partial of the landing pad, it believes the landing pad is under the drone. To make analysis more effective I implemented a confidence level which ensured the landing pad wouldn't be mis-identified. The second node is for moving the drone, it initially sets up a subscriber to the topic the landing pad identifier publishes to in order to decide when the drone should land. The second node has to publish movement commands and wait until

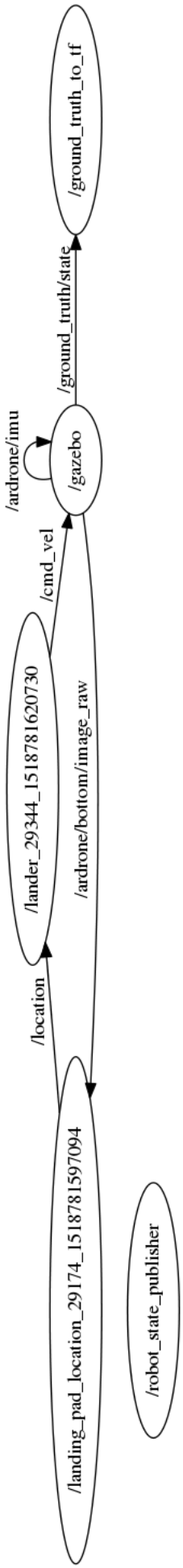
actions have been performed in order to land on the target effectively.

Here's the code for the landing pad identifier (this is the most relevant code):

```
<-- Import Statements -->

class Recogniser(object):
  NODE_NAME = "landing_pad_location"
  LANDING_PAD_IMAGE = "src/drone_land_on_red/scripts/found.jpg"
  ARDRONE_BOTTOM_CAMERA = "/ardrone/bottom/image_raw"
  IMAGE_CONVERSION_TYPE = "bgr8"
  IMAGE_COMPARISON_TYPE = cv2.TM_CCOEFF
  LOCATION_TOPIC = "/location"
  CONFIDENCE = 53500000.0 # How confident the images need to match
                            to declare we can see the landing pad

  def __init__(self):
    rospy.init_node(Recogniser.NODE_NAME, anonymous=True)
    # Camera goes a little crazy on takeoff so we will ignore
       the camera analysis until takeoff stabilises.
    self.completedTakeOff = False

    self.template = cv2.imread(Recogniser.LANDING_PAD_IMAGE, -1)
    if self.template is None:
      print("Error loading " + Recogniser.LANDING_PAD_IMAGE)

    self.bridge = CvBridge()
    self.image_sub = rospy.Subscriber(Recogniser.ARDRONE_BOTTOM_CAMERA,
                                      Image,
                                      self.callback)

    self.location_pub = rospy.Publisher(Recogniser.LOCATION_TOPIC,
                                        Location,
                                        queue_size=10)
    print("Setup complete!")

  def callback(self, data):
    # Analysis of the images from the ardrone bottom camera
    cv_image = None

    try:
      # Convert the raw image to a cv2 image
      cv_image = self.bridge.imgmsg_to_cv2(data,
                                           Recogniser.IMAGE_CONVERSION_TYPE)
    except CvBridgeError as e:
      print(e)

    if cv_image is None:
      print("cv_image is None!")
      return

    # Perform template matching
    res = cv2.matchTemplate(cv_image,
                            self.template,
```

```
                            Recognser.IMAGE_COMPARISON_TYPE)
    # max_loc are the indices of the highest value in the
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)
    depth, w, h = self.template.shape[::-1]
    center = (max_loc[0] + (w / 2), max_loc[1] + (h / 2))

    # Test if we have completed takeoff
    if (not self.completedTakeOff and max_val < Recogniser.CONFIDENCE):
      self.completedTakeOff = True

    # If we have completed our takeoff and we recognise the landing pad:
       publish the declaration and location
    if (self.completedTakeOff and max_val > Recogniser.CONFIDENCE):
      self.location_pub.publish(center[0], center[1], max_val)


<-- Node Running Code -->
```

As you can see from the *rqt* graph generated from this node (and the lander node), the communication between topics is simple and clear. The /gazebo node publishes the camera data to the landing_pad_location node via the /ardrone/bottom/image_raw topic, which in turn publishes Location data to the /lander node via the /location topic and finally the /lander node publishes velocity commands back to the /gazebo node via the /cmd_vel topic.


/ground_truth_to_tf
/ground_truth/state
/ardrone/imu
/gazebo
/cmd_vel
/lander_29344_1518781620730
/ardrone/bottom/image_raw
/location
/landing_pad_location_29174_1518781597094
/robot_state_publisher

2.6 Final Deliverable

2.6.1 Final Deliverable - Ball Finder

The key part of the final deliverable is the ROS node written to identify the ball from an image. This node subscribes to the /ardrone/front/image_raw topic from the AR Drone. Using some basic image manipulation techniques it identifies the most important areas of the image and then uses the Hough Circle Detection algorithm (implemented by OpenCV) to locate any circles which appear in the image.
The image received from the AR Drone topic is translated into a bgr8 image using cv_bridge so the image can be manipulated with OpenCV. The image is then converted to HSV encoding which makes identification of colour much easier. HSV stands for Hue Saturation Value, which means that pixels are stored with the respective values making the colours easier to identify as we only care about the pixel Hue while the Saturation and Value lies within a large range (avoids identifying Black and White as certain colours). This is much better than BGR (or RGB) colour encoding because when a colour gets brighter or darker with BGR, all three pixel attributes may change value, whereas HSV will only change one attribute.
The colour ranges for the HSV pixel selection are obtained using the *rosparam* (see Advanced Features of ROS) feature which allows launch files and terminal commands to change the parameters in the ROS system - this means that you can specify the colour range for the node to detect. With the user able to choose which colours to detect, the navigation node(s) can follow different colour balls in different simulations.
We use this HSV image to create a mask by matching any pixel within a certain colour range to 1 and any other pixel to 0. This mask is applied to the HSV image to remove all unnecessary pixels, and then converted to gray scale for the Hough circles algorithm (which requires gray scale images). This algorithm identifies the center of circles and their radii. Finally if a circle has been found, the details of it's location and radius are published to the circles topic to be used in the navigation nodes.
The methodology for this node was inspired by an article[14] which outlined how circles can be identified from a camera stream. The code was not used, or referenced in this project - only the methodology was used to inspire the HSV image method for generating an edges image to be used in the OpenCV implementation of the Hough Circles Detection algorithm.
The circles topic accepts messages of the Circle type which, is defined below and, declares an x and y location and the radius of the discovered circle:

```
Cicle.msg
---
int32 x_loc
int32 y_loc
uint32 radius
```

Here's the most relevant code for ball-finder.py:

```
<-- Import Statements -->

class Recogniser(object):
  NODE_NAME = "ball_finder"
  ARDRONE_FRONT_CAMERA = "/ardrone/front/image_raw"
  IMAGE_CONVERT_IMG_TO_BGR = "bgr8"
  IMAGE_CONVERSION_TO_HSV = cv2.COLOR_BGR2HSV
  IMAGE_CONVERSION_TO_GRAY = cv2.COLOR_BGR2GRAY
```

```
  CIRCLES_TOPIC = "/circles"

  LOWER_BLUE = None
  UPPER_BLUE = None

  def __init__(self):
    rospy.init_node(Recogniser.NODE_NAME, anonymous=True)

    Recogniser.LOWER_BLUE = np.array(rospy.get_param("lower_colour_list",
                                                     [110, 50, 50]))
    Recogniser.UPPER_BLUE = np.array(rospy.get_param("upper_colour_list",
                                                     [1130, 255, 255]))

    self.circle_pub = rospy.Publisher(Recogniser.CIRCLES_TOPIC,
                                      Circle,
                                      queue_size=3)
    rospy.sleep(0.5)

    self.bridge = CvBridge()
    self.image_sub = rospy.Subscriber(Recogniser.ARDRONE_FRONT_CAMERA,
                                      Image,
                                      self.callback)

    print("Setup complete!")

  def callback(self, data):
    try:
      cv_image = self.bridge.imgmsg_to_cv2(data,
                                           Recogniser.IMAGE_CONVERT_IMG_TO_BGR)
      hsv_image = cv2.cvtColor(cv_image, Recogniser.IMAGE_CONVERSION_TO_HSV)

      mask_image = cv2.inRange(hsv_image,
                               Recogniser.LOWER_BLUE,
                               Recogniser.UPPER_BLUE)

      masked_original_image = cv2.bitwise_and(hsv_image,
                                              hsv_image,
                                              mask=mask_image)

      masked_gray_image = cv2.cvtColor(masked_original_image,
                                       Recogniser.IMAGE_CONVERSION_TO_GRAY)
      circles = cv2.HoughCircles(masked_gray_image, cv2.HOUGH_GRADIENT,
                                 1, 20,
                                 param1=50, param2=30,
                                 minRadius=3, maxRadius = 100)

      if not (circles is None): # Send the first circle found
        ball = Circle(circles[0][0][0], circles[0][0][1], circles[0][0][2])
        self.circle_pub.publish(ball)

    except CvBridgeError as e:
      print(e)

<-- Node Running Code -->
```

2.6.2 Final Deliverable - Ball Follower (Stationary)

To demonstrate the effective behaviour of the final deliverable, the next step was to create a simulated world in which a ball would statically float just in front of the drone. The drone would have to; take off, recognise where the ball was floating and move to watch the ball (position the ball in the center of it's vision). In the simulation world the drone passively tends to drift which meant that even if the node wasn't started immediately after take off, the drone would still recognise the ball and move appropriately. This node subscribes to the /circles topic which was discussed in the previous section (see Final Deliverable - Ball Finder).
The core concept of this node is to identify which direction the drone needs to move in order to position the ball in the center of the front camera's image with the correct size (which translates to the right distance away from the ball). The code could be a little confusing due to the naming of variables between the 2D and 3D orientation. Keep in mind that if the ball is on the left of the front camera's image, the drone needs to move to the left in order to center it: This means the difference between the ball and the center of the image (dx) is negative, but the drone needs to move along the Y-axis in the positive direction (which is left relative to the drone) due to the orientation of axes in the simulation environment.
As you can see from the code, the direction in which movement is required is determined by the getYZReaction method (which also scales the movement down so it doesn't fly quickly in any direction) and the getXReaction method. These methods differ because the movement along the Y and Z axes are based on the circle position whereas the movement along the X axis is based on the circle radius (the distance from the ball).
To be clear, in the simulation axes are orientated as follows:

```
X: Longitudinal - Forwards and Backwards movement - positive is Forwards.
Y: Lateral - Left and Right movement - positive is Left.
Z: Vertical - Upwards and Downwards movement - positive is Upwards.
```

Here's the code for the Ball Follower (Stationary):

```
<-- Import Statements -->

class Navigator(object):

  NODE_NAME = "drone_strafer"

  CMD_VEL_TOPIC = "/cmd_vel"
  CIRCLE_TOPIC = "/circles"

  # This is the position (on the front camera image)
  #   the drone will try to follow the ball.
  CENTER = (320, 180)

  # The lower this value is, the more accurately the drone will
  #   follow along the YZ axes.
  IGNORE_DISTANCE = 20

  # This represents how closely (distance-wise) the drone will
  #   follow the ball.
  RADIUS = 20

  # The lower this value is, the more accurately the drone will
```

```
#   follow along the X axis.
IGNORE_RADIUS = 1

def __init__(self):
  rospy.init_node(Navigator.NODE_NAME, anonymous=False)

  self.lastCircle = Circle(320, 180, 20)
  self.lastReaction = self.getCleanTwist()

  self.pub_cmd_vel = rospy.Publisher(Navigator.CMD_VEL_TOPIC,
                                     Twist,
                                     queue_size=3)
  self.sub = rospy.Subscriber(Navigator.CIRCLE_TOPIC,
                              Circle,
                              self.callback)

  rospy.sleep(0.5)
  print("Setup complete!")

def callback(self, data):
  self.lastCircle = data

def navigate(self):
  while not rospy.is_shutdown():
    # dx/dy are difference in locations on image
    # drone reaction: +x = forwards, -x = backwards
    #               : +y = left, -y = right
    #               : +z = upwards, -z = downwards

    # negative if needs to move left
    image_diff_x = self.lastCircle.x_loc - Navigator.CENTER[0]

    # negative if needs to move upwards
    image_diff_y = self.lastCircle.y_loc - Navigator.CENTER[1]

    # negative if needs move forwads
    radius_diff = self.lastCircle.radius - Navigator.RADIUS

    reaction = self.getCleanTwist()
    reaction.linear.z = self.getYZReaction(image_diff_y) * -1
    reaction.linear.y = self.getYZReaction(image_diff_x) * -1
    reaction.linear.x = self.getXReaction(radius_diff) * -1

    if not self.repeatReaction(reaction):
      self.lastReaction = reaction
      self.pub_cmd_vel.publish(reaction)

def getCleanTwist(self):
  <-- Return a clean Twist object with hover mode disabled -->

def getYZReaction(self, difference):
  if abs(difference) < Navigator.IGNORE_DISTANCE:
    return 0
```

```
    # -1 or 1 based on move left or right/up or down
    return 0.1 * self.getSign(difference)

  def getXReaction(self, difference):
    if abs(difference) < Navigator.IGNORE_RADIUS:
      return 0

     # -1 or 1 based on move forwards or backwards
    return 0.1 * self.getSign(difference)

  def getSign(self, num):
    if num > 0:
      return 1

    return -1

  <-- Avoid repeating the same action code. -->

<-- Node Running Code -->
```

2.6.3 Final Deliverable - Gazebo Plugin Ball Mover

In order to have the drone track and follow an object (a ball in this case), the ball needs to have some kind of motion. Regardless if the ball's path is constant or random, any clear movement or motion is acceptable for testing the tracking abilities of the drone. Gazebo plugins are .so library files, typically written in C++ and compiled using the (C)Make system. For my purposes I've written a simple C++ plugin which sets the ball position in a circular motion based on the time of the simulation using the sin and cos functions of the mathematics library. Whenever the plugin has it's *OnUpdate* method called it will get the simulation time and calculate a new position for the ball. The motion of the ball is a small circle about the X-axis a short distance from the center of the world where the drone starts.
I used CMake and wrote my own CMakeLists.txt file for building the plugin and referenced the plugin .so file from the .world files where it would be used to control the ball movement. In theory the same plugin could be used to move any other object in an identical path, this is simply being applied to the ball and nothing else.
Here's the plugin code:

```
#include <functional>
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/common/common.hh>
#include <ignition/math/Vector3.hh>
#include <ignition/math/Pose3.hh>
#include <math.h>

namespace gazebo
{
  class BallMover : public ModelPlugin
  {
    public: void Load(physics::ModelPtr _parent, sdf::ElementPtr /*_sdf*/)
    {
      // Store the pointer to the model
```

```
      this->model = _parent;

      // Listen to the update event. This event is broadcast every
      // simulation iteration.
      this->updateConnection = event::Events::ConnectWorldUpdateBegin(
          std::bind(&BallMover::OnUpdate, this));
    }

    // Called by the world update start event
    public: void OnUpdate()
    {
      // Get Sim Time
      common::Time time = this->model->GetWorld()->GetSimTime();
      double sinTime = sin(time.Double() / 2.0);
      double cosTime = cos(time.Double() / 2.0);

      // Create pose
      ignition::math::Pose3d pose = ignition::math::Pose3d(3,
                                                           sinTime,
                                                           1.5 + cosTime,
                                                           0, 0, 0);

      // Set position to pose
      this->model->SetWorldPose(pose);
    }

    // Pointer to the model
    private: physics::ModelPtr model;

    // Pointer to the update event connection
    private: event::ConnectionPtr updateConnection;
  };

  // Register this plugin with the simulator
  GZ_REGISTER_MODEL_PLUGIN(BallMover)
}
```

2.6.4 Final Deliverable - Launch Files

In order to create a simulation environment in which the drone can track and react to a ball (even a stationary one) there are a number of nodes and process that need to be started before my own nodes can start to interact with the system. The obvious processes which need to start are the ROS master process (which is a simple call to the *roscore* command, a process which controls interaction between parts of the ROS system), the Gazebo simulation process and the loading of an appropriate simulation world. Users can also edit this launch file to control data like the colour of the ball in the simulation or the colour of ball the identification node should be looking for using the *rosparam* system (see Advanced Features of ROS). I've started most of these processes through the gazebo_ros package which provides functionality to start simulations and the ROS master process at the same time, furthermore I've passed in the world file containing the ball the drone will follow which in turn contains the link for the ball's motion plugin. Finally through the tum_simulator package I've called the spawn_quadrotor launch file which, as one might expect, spawns the drone in the simulation world.

I've also included the XML tags for using the ROS Parameter Service to change the detection colour range of the ball. This doesn't need to be here and it's not having any real effect on the system as I've set the values to the default values, however it shows how useable and easy the system is as the user would simply need to change the values in the list between the *rosparam* tags to change the behaviour of the ball identifier node.
Here's the launch file for a moving ball simulation world:

```
<?xml version="1.0"?>
<launch>
  <rosparam param="lower_colour_list">[110, 50, 50]</rosparam>
  <rosparam param="upper_colour_list">[1130, 255, 255]</rosparam>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name"
         value="$(find drone_follow_ball)/worlds/ardrone_ball_moving.world"/>
  </include>

  <include file="$(find cvg_sim_gazebo)/launch/spawn_quadrotor.launch">
    <arg name="model"
         value="$(find cvg_sim_gazebo)/urdf/quadrotor_sensors.urdf.xacro"/>
  </include>
</launch>
```

2.6.5 Final Deliverable - Gazebo AR Drone Tracking Moving Ball

The final program under development is the follow_ball.py script, which creates a node designed to interpret the output of the ball finding node and creates appropriate reactions for the drone in order to keep the ball in the center of its vision.
The ball finder node publishes located circles to the /circles topic which we use as the last known location of the ball. Our navigation node (follow_ball.py) subscribes to two topics, the first being /circles so we can track and react to the ball, the second topic is a /toggle_lateral topic. The /toggle_lateral topic is effectively a listener which toggles the drone's movement style between using horizontal motion (lateral movement) and using Z-axis rotation (yaw).
Most of the navigator script is setup for creating Publishers, Subscribers and class variables. Whenever a Circle message is published to the /circles topic, the navigator node will overwrite the previous instance of it's circle field. The node then regularly calculates how the drone should react to the last known position of the ball in an attempt to place the ball in the middle of it's front camera, with a small leeway where when the ball is close enough to the middle of the drone's vision, the drone will attempt to stabilise.

Initially in development the drone would display exaggerated and apparently wild movement as it tried to follow the ball due to an issue in the documentation of the /cmd_vel topic and it's usage. The /cmd_vel topic does not control the drone velocity (as one would expect). The /cmd_vel topic is used to control the drone tilt along the X and Y axes, but velocity along the Z axis and a relative rotation rate for the yaw. This means that when I was initially trying to set the drones X axis velocity to 1 ms^{-1} which in reality set the drones X axis tilt to the maximum tilt for the drone - this caused the drone to move at an alarming rate in the opposite direction along the X axis. The input for the linear X and Y axis values represents the proportion of maximum tilt to apply to the drone, where -1 represents maximum tilt in the positive direction and 1 represents maximum tilt in the negative direction. The input for the angular Z axis value represents the proportion of 95 degrees of rotation to apply in the clockwise direction. This issue has been raised on GitHub[20]
I overcame this issue by defining values where the drone would reach maximum acceleration

(or tilt). By defining a distance the circle must be from the center of the drone's vision before the drone would attempt to maximise it's acceleration, I could use this value to scale down the actual difference to achieve a percentage of the maximum tilt which would result in a smaller tilt, a slower acceleration, when the ball wasn't far from the center of the drone's vision.

As a final extension, I've added the functionality to avoid the drone repeating commands by saving the last published command and comparing it to any command it attempts to publish next. While this isn't strictly necessary, it avoids publishing an excess of messages or any useless messages being published to the /cmd_vel topic. Obviously if you try to tell the drone to move upwards at a rate of $1ms^{-1}$ twice in a row, there will be no change in behaviour since the drone is already performing that task. This optimisation feature doesn't save a huge amount of time nor space within the context of this project but it will reduce the number of operations on the processor which means others could use this package for more complex features and it means the package will function a little more efficiently on machines with smaller processors.

An interesting feature of the this node is; provided the ball's velocity does not exceed the top speed of the drone, the drone will always keep the ball in its vision. Furthermore if the ball travels at a constant velocity in any direction at a speed lower than or equal to the top speed of the drone, the drone will match its velocity at a perfect and constant offset from the optimal position in the image. For example, if the ball is moving at $1ms^{-1}$ and the top speed of the drone is $2ms^{-1}$ and the maximum speed is achieved when the pixel difference is 200 pixels. Then when the ball is 100 pixels away from the optimal position (along the relevant axis) the drone will be moving at half of it's maximum speed, which happens to be $1ms^{-1}$! This lead me to formulate an equation which can track the position of a ball within the drone's vision. The equation is as follows:

$$dx = \frac{v_1 mx}{v_2}$$

Where:

dx = The difference in pixels the ball will be from the optimal position in the drone vision
v_1 = The velocity of the ball
v_2 = The maximum velocity of the drone
mx = The difference in pixels the ball must be for the drone to achieve maximum velocity

Here's the core code for the navigator node (follow_ball.py script):

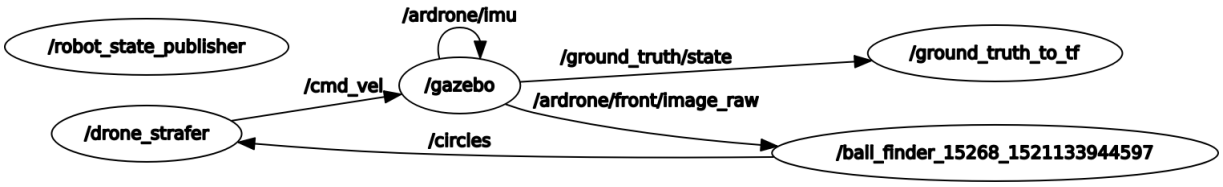
```
<-- Import Statements -->

class Navigator(object):

  NODE_NAME = "drone_strafer"

  CMD_VEL_TOPIC = "/cmd_vel"
  CIRCLE_TOPIC = "/circles"
  TOGGLE_LATERAL_TOPIC = "/toggle_lateral"

  CENTER = (320, 180)
  IGNORE_DISTANCE = 5
```

```
RADIUS = 20
IGNORE_RADIUS = 1

# Forwards/Backwards speed is maximised at radii difference of 5
MAX_X_SPEED_DIFF = 5.0
# Left/Right speed is maximised at pixel dx = 100
MAX_Y_SPEED_DIFF = 100.0
# Left/Right rotation speed is maximised at pixel dx = 100
MAX_Z_ROTATION_DIFF = 100.0
# Up/Down speed scales with pixel dy = 25
SCALAR_Z_SPEED = 25.0

def __init__(self):
  rospy.init_node(Navigator.NODE_NAME, anonymous=False)

  self.lastCircle = Circle(320, 180, 20)
  self.lastReaction = self.getCleanTwist()
  self.lateral = True

  self.pub_cmd_vel = rospy.Publisher(Navigator.CMD_VEL_TOPIC, Twist, queue_size=3)
  self.sub_toggle = rospy.Subscriber(Navigator.TOGGLE_LATERAL_TOPIC,
                                     Empty,
                                     self.toggleCallback)
  self.sub = rospy.Subscriber(Navigator.CIRCLE_TOPIC, Circle, self.callback)

  rospy.sleep(0.5)
  print("Setup complete!")

def toggleCallback(self, data):
  self.lateral = not self.lateral

def callback(self, data):
  self.lastCircle = data

def navigate(self):
  """Navigate to the last known circle position (while rospy is up).
     Avoids repeating reactions."""
  while not rospy.is_shutdown():
    # image_diff_(x/y) are difference in locations on image
    # drone reaction: +x = forwards, -x = backwards
    #               : +y = left, -y = right
    #               : +z = upwards, -z = downwards

    # positive if needs to move left
    image_diff_x = Navigator.CENTER[0] - self.lastCircle.x_loc
     # positive if needs to move upwards
    image_diff_y = Navigator.CENTER[1] - self.lastCircle.y_loc
     # positive if needs move forwards
    radius_diff = Navigator.RADIUS - self.lastCircle.radius

    reaction = self.getCleanTwist()

    reaction.linear.x = self.getXReaction(radius_diff)
    reaction.linear.z = self.getZReaction(image_diff_y)
```

```
      if self.lateral:
        reaction.linear.y = self.getYReaction(image_diff_x)
      else:
        reaction.angular.z = self.getZRotation(image_diff_x)

      if not self.repeatReaction(reaction):
        self.lastReaction = reaction
        self.pub_cmd_vel.publish(reaction)

  def getCleanTwist(self):
    <-- Docstring -->
    twist = Twist()
    twist.linear.x = 0.0
    twist.linear.y = 0.0
    twist.linear.z = 0.0
    twist.angular.x = 1.0 # Disable hover mode
    twist.angular.y = 1.0 # Disable hover mode
    twist.angular.z = 0.0
    return twist

  def getXReaction(self, difference):
    <-- Docstring -->
    if abs(difference) < Navigator.IGNORE_RADIUS:
      return 0

    return difference / Navigator.MAX_X_SPEED_DIFF

  def getYReaction(self, difference):
    <-- Docstring -->
    if abs(difference) < Navigator.IGNORE_DISTANCE:
      return 0

    return difference / Navigator.MAX_Y_SPEED_DIFF

  def getZRotation(self, difference):
    <-- Docstring -->
    if abs(difference) < Navigator.IGNORE_DISTANCE:
      return 0

    return difference / Navigator.MAX_Z_ROTATION_DIFF

  def getZReaction(self, difference):
    <-- Docstring -->
    if abs(difference) < Navigator.IGNORE_DISTANCE:
      return 0

    return difference / Navigator.SCALAR_Z_SPEED

  <-- Avoid repreating the same action code. -->

<-- Node Running Code -->
```

Once again the communication between topics is minimal and clear. The generated graph is

similar to the Landing Pad graph (see Landing Pad Graph) as the core concept is the same - an identifier node listens to data from the drone's camera and publishes it's findings to a topic, a decision node calculates a reaction for the drone to perform and publishes this data back to the /cmd_vel topic so the simulation can update the drone's movement.



Chapter 3: **Analysis of Final Deliverable**

3.1 Achievements

In this project there were a number of major achievements during development and with the final deliverable(s). The largest achievement of development was getting all of the packages and pieces of software to communicate effectively, which was no small feat as there was considerable difficulty in finding packages which interacted with one another in order to provide the support I needed to develop the final deliverable(s). This was made especially difficult by the fact that many packages were not supported for the ROS kinetic distribution on the ROS Package Index, leading me to scour GitHub repositories to find stand-alone packages which had been developed specifically for the kinetic distribution.

The most interesting achievement of this project was the identification of the ball from an image (and the lack of mis-identifying the ball when it was (or wasn't) present). The HSV image conversion provided an interesting insight into the method of colour encoding and it allowed the accurate identification of the ball using the Hough Circle Transform algorithm. This method of image encoding is an important piece of software because, as demonstrated in this project, there are a range of everyday objects which have (effectively) a single colour but the as light hits objects at different angles and in different amounts the objects can be very hard to identify when the colours differ by many values when stored in the traditional RGB (or BGR) encoding system. The HSV encoding can massively reduce the effects of shadowing on the identification of colours in images and image streams.

A minor but still interesting achievement of this project was the ability for the navigation node to swap between lateral movement and yaw rotation in order to keep the ball in sight. While this achievement was minor, it creates the potential for a stationary tracker drone if you were to disable the X-axis tracking; the drone would act as a camera which rises and falls with the target object, this changes the use cases of the drone. If you were trying to record or track an activity where there isn't a large amount of lateral or longitudinal movement but interesting ranges of vertical movement then this would be much more appropriate. Sporting cases in which this applies could include: Trampoline Gymnastics, Gymnastics and Skateboarding.

Another minor achievement of this project is the development of a Gazebo plugin. While the development of a plugin for such a specific use case (moving a ball in a circle) isn't particularly useful (- as not many things move in this way) it creates the opportunity to develop plugins which move an object (not just a ball) in more interesting and complex ways. Whether an object should move randomly, follow a path or simple change velocities in various situations, a Gazebo plugin can be developed from the code in the project package which provides this functionality without having to concern themselves with the inner workings or confusing code snippets of plugin development and Gazebo itself.

One achievement of the project I am very happy with is the usage of the *rosparam* XML tag and the ROS Parameter Service which allows the ball_finder.py script to be reused in other simulations without editing it at all. Users will be able to define the colour range of the ball for the node to identify from their launch files or via the terminal which means the user doesn't need to mess around with the code in order to change it's behaviour. This makes this aspect of the package highly re-usable for other use cases.

Obviously the biggest achievement of this project is the successful creation of a ROS package

which can accurately track and follow a target smoothly. When running the navigator node (follow_ball.py script) you can use the image_view package to view the camera output of the drone which tends to follow the ball rather accurately and smoothly when the ball is relatively near the optimal position. This means that while the drone is following the ball accurately; the drone doesn't allow the ball to escape vision. The only erratic or wild behaviour of the drone is when the ball is on the very edge of vision or when the ball finder node has identified the ball before the navigator node has started, as the navigator will move the drone to the ball's last known position - which it may no longer be near. For this reason it would be more effective to use scripts which initialise the navigator before initialising the identifier(s), which could be achieved through a launch file or bash script in order to make the package more consistent, this wouldn't reduce the re-usability of the navigator node as it would still be usable as a stand-alone node.

With respect to the Project Aims (see Project Aims), the package developed (drone_follow_ball) meets the aims to some extent. I have successfully developed a package to autonomously navigate an AR Drone to follow an object - in this case a coloured ball.
The package has easily readable and understandable code which users will be able to edit to suit their own needs, and the package is effectively reusable through the use of the *rosparam* XML tags to change the colour of the ball to be tracked.
The drone's following behaviour can be changed through the /toggle_lateral topic to switch between lateral movements and yaw rotation to look at the target object.
Finally the package code has been developed and thoroughly tested in a simulated environment using Gazebo. Although the project did not have time to test against real world environments, it would not be particularly difficult to create ROS nodes or scripts which subscribe to the /cmd_vel topic and use these to update the real drone's movement. Similarly another node could effectively collect the real drone's camera data and publish it to the /ardrone/front/image_raw topic, which would feed the drone's camera output to the system. To complete this development, the developer would simply need access to an AR Drone (to which I do not), and to write an appropriate launch file which starts these I/O nodes instead of the simulation.
Due to the development process and ROS package system I believe that this project aim has been adequately met as the code will function in the same way regardless of whether the data being fed into the system is from a simulation or a real drone.

3.2 Difficulties

During development there were a number of difficulties. Due to the system being comprised of three core technologies; ROS, Gazebo and OpenCV - which are all Open Source projects, there were some difficulties concerning the various versions of each piece of software and how it interacted with the others. Initially only one version of ROS was available on my operating system (Ubuntu 16.04) which was ROS kinetic. While these two systems interacted effectively, as soon as I started looking into packages available for kinetic I discovered that a large range of packages were not supported on this version of ROS as it was relatively recent at this time (ROS indigo had just been released at the start of this project, making kinetic the penultimate release at this time.). Most packages, which online forums had discussed, were for the ROS fuerte distribution which is a number of versions behind kinetic (named alphabetically, so kinetic is five versions ahead of fuerte), this meant that code was no longer relevant as there was a large system upgrade when ROS hydro was released. The version of ROS I was using dictated the version of Gazebo I would need in order for them to be compatible and communicate effectively. ROS kinetic requires Gazebo 7.x which is the version I installed to get my simulations running. Thankfully the Open CV library only depended on

the Python version of the source files. I was using Python 2.7.12 and so I used OpenCV 2.4.9.

The biggest issues in dependencies was in installing ROS packages, as stated previously many version were not supported. Instead I had to scour GitHub and similar sites in order to locate versions of packages which were compatible with ROS kinetic. I had to locate unique packages for the tum_simulator, ardrone_autonomy, hector_quadrotor and cv_bridge, which were all important packages for controlling the drone and converting ROS Image messages into Python objects compatible with the OpenCV Python binding.
Incompatible packages are the cause of a number of side effects during development, some were entertaining, some were particularly frustrating. Initially before I discovered the need for the ardrone_autonomy package the drone wouldn't stop taking off, it would rise up in the air forever but wouldn't register as completing take off so no commands issued via the terminal would affect it (except for emergency landings).
A similar issue was that the drone had a hover mode which didn't exactly hover on the spot, it would regularly drift in a constant direction which caused minor issues in the manual testing process before any behaviours would be attempted.

Another issue was with the package.xml file, after the ROS upgrade in the hydro version there became two formats for the package.xml file and the CMakeLists.txt. While developing my packages I discovered that there were also two ways of creating the base content for a package, one used the ros-create-pkg function and one used the catkin-create-pkg. The difference between these packages is that the ROS package creation generates the base content for the legacy package format. As a result I wasn't able to effectively use functions like dependencies on more recent packages nor create my own custom messages and services. Eventually I worked out what was wrong and managed to fix the issues but this took a considerable amount of time and research considering I was unaware that the two functions existed at the time. The reason there were a large number of delays in the project was because there were multiple times where there were more than one dependency or incompatibility issue which wasn't returning error codes or stack traces, which meant the only debugging process was searching huge outputs for warnings using tools such as grep and working out issues within the simulated environment.

The final difficulty in development was an issue with the /cmd_vel topic. The documentation for this topic suggests that the topic is used to set the velocity of the AR Drone - which is what one would assume it does based on its name, however this topic does not perform the exact behaviour specified in the documentation. The topic accepts Twist messages, which specify three linear and three angular components, where the linear components represent direction and motion along the XYZ axes, while the angular components are only partially used for rotation.
The linear components are used accordingly:
The X and Y components represent the amount of tilt to apply along the relevant axis, where -1 represents maximum tilt towards the positive end of the axis, 1 represents maximum tilt towards the negative end of the axis and 0 represents no tilt along the axis.
The Z component represents the velocity of the drone in meters per second where -1 represents movement towards the negative end of the axis, 1 represents movement towards the positive end of the axis and 0 represents no movement along the axis.
The angular components are used accordingly:
The X and Y components (roll and pitch respectively) have almost no effect on the drone as the drone would not be able to sustain flight at an unstable roll or pitch without moving because the roll and pitch are used to control motion along the X and Y axes. When all linear and angular components of Twist are set to 0; the drone enters hover-mode, which can be disabled by changing these values (angular X and Y) to arbitrary non-zero values.
The Z component is used to change the yaw of the drone, where 1 represents a rotation rate of roughly 95 degrees in the clockwise direction per second.

I discovered these /cmd_vel issues while developing the final deliverable to follow a moving ball when the drone started moving erratically trying to correct its positioning at an alarming rate. After some searching around this issue I discovered the relevant GitHub issue ticket[20], which supplied some possible solutions to this problem involving calculating the appropriate values based on acceleration, gravity and the maximum tilt angle, however after some time searching I couldn't identify with confidence the maximum tilt angle or decide how aggressively the drone should accelerate in certain situations which lead to my practical solution. My solution would be to choose a pixel difference at which the drone would maximise it's acceleration to get the ball to the optimal position and scale the desired tilt with this value. For example; if the maximum tilt occurred when the pixel difference was 100 and the current pixel difference was only 25, the navigator node would calculate a tilt of 0.25 or 1/4. This ensured that the drone would smoothly increase its tilt and velocity in order to match the ball speed or even overtake it so that the ball would be in the center of the drone's vision again. You can imagine that there is a speed at which the ball can travel in one direction in which the navigator will have the drone match the speed perfectly based on the maximum tilt pixel difference (see Constant Speed Calculation).

3.3 Effectiveness

The final deliverable in this section will refer to the drone_follow_ball package which contains a series of scripts, launch files, world files, a message file, a Gazebo plugin and various other minor files. The world files successfully create the simulation worlds with objects initialised in the correct position and uses the Gazebo plugin on the correct model. The gazebo plugin is also very successful as it ensures the ball follows a constant path, while variety in the path would potentially test the drone's tracking ability to a greater extent, a user or observer would have to constantly reposition the camera whenever there was a chance the drone and/or ball would move out of sight. Issues could also arise from the ball or drone reaching the 'bounds' of the simulation, although this is unlikely to happen during a simulation.
The ball_finder.py script locates the ball very successfully and at a relatively high speed, it also publishes it's findings to the /circles topic efficiently due to the Circle message type which is defined by this package.
The follow_stationary_ball.py isn't as effective as it could be, although this script was, at it's core, a test to drone control via basic analysis of an image with a stationary object. This node served a core purpose of tracking and reacting to an object as the drone appeared to passively drift away from it's take off position, regardless of whether the drone was in hover mode or not. This node was used to clearly show that this passive drift effect wouldn't affect the drone's behaviour once it started reacting to the camera output as anytime the drone drifted away from the object, it would detect the object had moved (in reality the drone itself had moved) and move to correct this position change. For that purpose the script is effective in it's main goal, the script was not continued as it represents a key milestone in development of the final deliverable. It's worth noting that the follow_ball.py script would also perform the same task even more effectively as tracking a moving ball and tracking a stationary ball are effectively the same task - as in, a drone which can track a moving ball can track a stationary one, but not vice versa. It's also important to note that at the point of development of the stationary ball follower I was unaware of the issues surrounding the /cmd_vel topic which is another valid reason for the script not being re-used to develop the final ball following node.
The final script developed, follow_ball.py is the core of the final deliverable - reacting to movements the ball makes and steering to keep it in vision. It performs this task very well, it's only limitations are the top speed of the drone versus the top speed of the ball and in the event the ball detection publishes messages of a ball which is no longer present to the

/circles topic then the drone would attempt to find the ball in that direction, which could cause the drone to fly far from the ball as it is no longer in said position. The feature for toggling the lateral movement with the yaw rotation for following the ball is rather effective, although some minor tweaking of the rate of rotation could be done to remove a small amount of camera shaking in yaw mode - though this is a simple as changing a value at the top of the class which represents where maximum rotation rate occurs. The navigator node is an effective method of observing the ball at an almost constant position, furthermore the node would be effective for any drone type which accepted Twist objects and had a front camera, the use of the /cmd_vel topic can easily be changed via the class variables should another simulated drone use a different topic in the ROS system.

3.4 Improvements

The are a small number of improvements and extensions which would have improved the final deliverable package.
A feature of ROS I would have liked to have used more is the *rosparam* functionality, which allows users and launch files to specify the values of parameters, similar to how scripts and programs can be called with arguments which affect behaviour. The *rosparam* feature could be used to change the behaviour of the launch files in a way which affects the motion of the ball, for example a param could be used to make the ball stationary/follow a circle path/follow a square path/move randomly which would mean that there would only be one launch file for all configurations instead of one launch file per configuration. The only difference in the launch files would be a *rosparam* XML tag which specifies a default behaviour, such as linking to the ball_mover.so library file.
The *rosparam* feature could be further utilised from within the node scripts in order to specify the pixel differences which affect maximum acceleration and tilt for the drone or the position in which the drone should attempt to keep the ball in vision. This means that a user would be able to change the way in which the drone follows the ball without editing the node source code at all.
Although I have implemented a feature which allows users to control the colour of the ball to be detected in the ball_finder.py, an even further advancement of this feature would be having multiple drones which each follow a ball of a certain unique colour. In this case the Circle message specification could be changed to include a ball colour which would allow drones to communicate indirectly by publishing the locations of all the balls they can see and what colour each ball is so that other drones could use this information to track their own ball.

Another area in which the project could be improved would be by implementing my own variation of the Hough Circles Transform algorithm - or at least learning in more detail how the OpenCV parameters for the method call affect the resulting detection of circles, this change could allow me to run ball detection with a higher time/space efficiency. However this change is relatively minor and it may not be worth re-writing an implementation unless the time or space saved makes a significant difference to the package performance.

A major change with a minor difference could be made to the follow_ball.py script which would subscribe to the /navdata topic, specifically the altitude field, in order to ensure the drone doesn't drop below a certain altitude or make contact with the ground to avoid damaging the drone in a real world scenario.

The development process could have been improved by researching the inner workings of some of the Open Source software. This would have been practical if the project had a larger scope, it would have been more beneficial to understand the processes involved in creating

the simulation environment, the ROS /clock topic and it's interaction with timings in the simulated world and the functionality of OpenCV with various alternatives or implementations of better algorithms which may suit the deliverable more effectively.

As stated in the achievements section (see Achievements), an improvement which could be made to this package would be the development of ROS nodes or scripts which feed the data from /cmd_vel to a real drone and feeds the drone's front camera data into the /ardrone/front/image_raw topic. This would allow the development of a launch file which initialises the appropriate ROS parameters and ROS nodes without starting the simulation environment. But as stated earlier, for this improvement I would require regular access to a Parrot AR Drone to test the I/O nodes/scripts and an environment in the real world to test the behaviours in (such as a clear room or open outdoor space with minimal obstructions).

Chapter 4: **Software Engineering**

Due to the nature of this project, nodes tend to take input via subscribing to topics and output via publishing to topics - this makes unit testing very difficult as any unit test would have to initialise the roscore system and create various topics to feed data into nodes, the nodes themselves would have to be started and run in another process and then the test nodes would have to subscribe to data before finding a way to report to the tester which tests had passed or failed. This creates a lot of overhead and makes the test process difficult and tedious, instead throughout this project nodes have been manually tested in most cases - during the development of a node when any feature was added or changed, the simulation would be started and I would manually confirm whether the change was successful or not.

In some cases, however, stand-alone test scripts have been used to ensure complex parts of the system are working effectively. Notably in the final deliverable there is a test_ball_recog.py script which uses an image file (front_image.jpg) in order to test the Hough Circles Transform implementation in OpenCV with the image from the drone's front camera. This test script also opens up various windows to show the steps involved in recognising the ball from the image - this is particularly useful for understanding the process of identifying the ball from the drone's output.
Accompanying this script is a save_front_camera_image.py which saves a snapshot from the drone's front camera in a way the image can be used for the ball recognition test script.

I also tested the effectiveness and efficiency of the OpenCV pattern matching functionality. This test was a script which used varying sizes of templates against a constant larger image to define how effective the pattern matching implementation would be when the drone was a range of distances from the landing pad. You can see this test in the opencv/ directory along with the default template and source image - the test accurately identifies the template in the image for each template size and shows the time taken to identify each template and the location of the best match for the template. This is a particularly useful test as it allowed me to account for how accurate the recognition node would identify the template when the drone was: in flight, far away or stationary. It also gives a sense of time where it takes much longer to match the template when the template is small as there are more comparisons to do with the original image, this means that the drone should fly relatively low so that (in this specific case) the identification script will be fast enough to identify the pad before the drone moves on.

Throughout the process of development while using manual testing, some nodes and scripts could not be tested via visual confirmation in the simulation environment - for example, the ball identification node was written before the drone navigation node (for obvious reasons) which meant that the identification node would be publishing data to a topic which was not being read or used in the simulation.
To solve this issue I made use of the *rostopic echo <topic>* feature of ROS which allowed me to check the output of the node by having it displayed directly on the screen. This provides me with all data published to *<topic>*, allowing me to check the output of the identification node. When the ball is stationary; the ball reports to be in the center of the drone's vision until the drone passively drifts away - which is the correct behaviour for this node - in the event the ball is moving the node accurately identifies it's position with the associated image.

Some areas of the package were particularly easy to test but only required testing once or twice to ensure the functionality was being performed as expected. The *rosparam* command provides functionality for getting the value of certain parameters, which means I could test that the launch files were setting parameters properly using the rosparam XML tags by running a simple command.

```
rosparam get <parameter_name>
rosparam get lower_colour_list
```

Chapter 5: **Project Diary**

October 3 - I completed the basic ROS tutorials today and created my own catkin workspace in the my GitHub repository. I created a package for my learning ros code, I created two scripts for practising and learning ROS. One of which is to demonstrate a basic publisher (draw_circle.py) and the other is for demonstrating the use of publishers and subscribers in a single script.

October 12 - Meeting with advisor today to discuss how to proceed with next deliverable. Next deliverable will be a piece which encompasses all aspects of ROS to prove understanding and will involve the user specifying a location for the turtle to navigate to until a proximity is reached.

October 16 - Completed early deliverable for understanding ROS. Package which navigates a turtle from turtlesim to a target location specified from the user. Makes use of nodes, topics, publishers, subscribers, messages and services. Starting work on Gazebo simulation software.

October 22 - Spent the last few days investigating ROS packages and looking into how Gazebo and ROS can interact, started selecting packages to be used for the next deliverable for creating a Gazebo simulation with ROS interaction (possibly including a basic drone simulation).

October 23 (13:35) ROS site has gone down, temporarily blocked in proceeding with Drone Simulation. Investigating basic behaviours and drone control.

October 28 - After much searching through packages available to me, I finally found a package which simulates the Parrot AR Drone on the kinetic version. Managed to get some basic simulations working and started testing services and publishing to the new topics.

November 4 - Tested the usage of a range of topics in a gazebo simulation of ar drone(s). Started writing a topic which will steer the drone towards a target (currently just a red 'landing pad') and attempt to land on it.

November 13 - Spent the previous week (admittedly not had much time) looking into a range of packages to deal with computer vision on the ardrone. Due to the limited set of ROS packages available on Kinetic. I've also looked into code snippets and other, more generic, Python packages to do this. While some of these are viable options I've decided, for now, to develop the rest of the term deliverables in the Gazebo environment using solid colour panels to identify targets and landing pads. This means that at a later date I will be able to change the identification of targets and obstacles without changing the code! (Object Oriented Programming at it's finest!)

November 22 - Spent a long time investigating image recognition in ROS/AR Drone/Python. After a meeting with supervisor I was given some places to look and began working on HOG, looking into the practicalities of it. Found a range of other packages which are used for image recognition but they are considerably dated. (It would appear very little if not any work has been done on the Git repo due to testing and work done on my local machine, considering a lot of time has been spent setting up environments and testing if packages work I haven't committed these are they aren't useful until they work.)

November 24 - Started learning to use OpenCV (Python binding) and found a package to convert from ROS images to OpenCV images which allows me to think about object detection from the Ar Drone cameras.

November 28 - Written report on OpenCV, started work on Interim Report, started work on Interim Viva. Begun work on OpenCV test programs to show effectiveness. Found a ROS package to map ROS image messages to OpenCV images: `http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages`

January 19 - Started project work on a landing pad, spent time creating a landing pad in a gazebo world. Created launch file for program to create quadrotor and started creating a script to recognise the landing pad.

January 30 - Image recognition is proving harder due to incompatible data types (camera feed vs CV implementation) spending time working on converting images (cv-bridge is proving difficult). System seems to be having issues showing my camera output which is causing issues and slowing development. Started looking into object avoidance. Started looking into moving objects in gazebo sim for the moving landing pad deliverable (based on feasibility).

February 10 - For the last few days I've been toiling away at getting open cv to compare a stock image of a red square (the landing pad) with the ardrone bottom camera output. There have been two major issues with having this work: Firstly the ros package I created was using the old formatting system, which caused the files to behave in an unorthodox way. At the same time there were repeated Assertion Errors being produced in the pattern matching process where the depth of the images did not match (I assumed this was just about how the drone image and the .jpg image were storing their colour descriptions for each pixel). Eventually I solved these two issues by creating a new new-style package (called drone-land-on-red) which fixed the irrational behaviour of the ros system with the file structure (also fixing another issue creating custom messages). And I used the cv-bridge to convert an image of the landing pad and save it into a new .jpg file which was then used for comparison with the camera output. (This might sound like I saved an image and compared it to itself but I saved the image so that when the drone wasn't in the same position I could check whether or not it was in the same position (or similar)). This (seemingly small) piece of work has been a major hurdle in development for some time now which has caused the progress of the project to grind to a halt. Now it is solved I have started using the template matching ability of open cv to start identifying the landing pad and should have this deliverable complete tomorrow. I will spend the next week (after completing this deliverable) starting the final deliverable (to be explained after this post as it's getting quite long), writing a report on cv-bridge and how it allows the open cv library to be used in conjunction with the ardrone and finally I will start writing the Final Project Report Draft.

February 10 - **Final Deliverable Specification** I have decided the final deliverable will be a ball tracking drone. A simple object (likely a coloured ball) will move around (either randomly or following a path) to which the drone will identify where the ball is (if it can see it) and readjust its position to follow it. Should the drone no longer be able to see the ball it will attempt to find it by following its last known position. Potential extension: If another ball is discovered or the ball's colour changes, change behaviour (e.g. avoid the ball or look away from the ball)

February 12 - Created lander.py and wrote code for the drone to land on the red landing pad. Solved minor issues surrounding image loading with the open cv library and the cv-bridge package. The final issue in developing this deliverable is that it appears gazebo is stopping publishes to the /clock topic which causes the nodes not to publish to nodes correctly. Currently searching for the exact nature of the problem and a solution.

February 23 - Started development on the final deliverable package. Created scripts to take snapshots on the front camera for ardrone for image development purposes. Created a script to identify ball from the taken image, this will be used in part to recognise the ball in a node

where it is present for the deliverable.

March 10 - After experiencing difficulties with the drone simulation following the moving ball correctly, I have discovered the following GitHub Issue: `https://github.com/AutonomyLab/ardrone_autonomy/issues/116` Which explains that the cmd_vel topic (used to specify the velocity of the drone) is not specifying velocity as documented (and expected) it actually specifies the tilt angle percentage for the drone (which causes a lot of issues with scaling movements). Furthermore the z component of movement is actually a velocity.

March 12 - The final deliverable is now complete, consisting of two nodes, a simulation launch file and a gazebo plugin. The simulation world contains a drone and moving ball (using the plugin), the nodes are for tracking the ball and following the ball. There is an option to substitute the lateral movement with look/rotation through the use of a topic. The drone accurately follows the ball in the correct manner. This main behaviour node is called follow_ball_scaling.py as it follows the ball with movement scaling with how far the ball is from the optimal position. I will also consider developing a second final deliverable which uses relative motion to decide how to move meaning the drone will try to match the ball's velocity instead of trying to center it on the front camera. This may be much more difficult due to the previously discussed issue outlining the motion of the drone in respect to the /cmd_vel topic controlling tilt and not actual velocity difference, but I will look into this deliverable to grasp it's feasibility.

March 15 - As a minor extension of the final deliverable I've added support for the ball_finder.py script to identify different coloured balls using the ROS Parameter Service and rosparam XML tags in launch files. I've decided against using relative velocities to follow the ball smoothly as this functionality is low-reward and will require a large investment of time due to the difficulties with the /cmd_vel topic issues previously discussed. I've recorded a video of the drone functioning as intended with the image_view to show the drone camera output, a call to rostopic echo on the /circles topic to show the ball detection working effectively and calls to the /toggle_lateral topic showing the difference between lateral movement and yaw rotation. Finally I've completed large amounts of the report and added rqt_graphs for the final deliverable.

Chapter 6: **Professional Issues**

Drones are becoming a popular part of both the computing and recreational worlds. This means that there are a wide range of applications for drones and professional issues surrounding one area of drones could have big consequences if used in another scenario. Here's an extreme scenario:
The final deliverable for this project was a package which controlled a drone to make it follow a target, attempting to keep the target in vision. While this could be used innocently enough such as an unmanned video camera for activities with a long range of motion involved such as filming a skateboarding run or a football game where the drone can follow the action from a reasonable distance without being too far from the action. However a prime example of mis-use of this package would be any military application or reconnaissance missions, should a drone with sufficient battery capacity and a relatively high end camera be using this package it could track a target from relatively unseen distance and relaying information or video to a controller. In an even worse case this could be applied to a drone with weaponry - this abuse of technology could mean that human controllers would be able to end the lives of people without being anywhere near the act.
In any situation, drones should be used in public situations and personal use without recording any data without prior permission from everyone involved. An improper use of drones in a recreational environment could be an individual using a drone to observe others or track movement of individuals, this is an invasion of privacy and could, in theory, be controlled by limiting battery life or operational distance.
Ethical Issues with use of Drone Aircraft [19] suggests an event where a drone which is NOT FULLY autonomous, a drone in which a human has influence over the action the drone takes, the human should be held accountable and should be the one to consider any ethical issues involved in the actions they influence on the drone. The suggestion here is that the one who influences the drone's behaviour is the one held responsible and the one who should consider the ethics. Where a drone is fully autonomous, the influencer of the behaviour is the individual or group who wrote the decision code behind the drone behaviour. In this project the individual who influences the behaviour of the drone is me, and as such I have considered the applications of the final deliverable package. The package I have developed in this project is limited in usage, it can clearly be used to track objects and land on targets but it needs enough information about the objects it needs to track and targets to land on in order to effectively perform these operations. It would also be greatly limited by battery life and range of effect. While this project hasn't looked into using physical drones with ROS packages and processes, it is possible to do and a user who develops on my code would have to look into areas such as where the processing of the ROS data is done in order to use it. I raise this point because if ROS is installed and processed on the physical drone itself then the drone would have an effective range of however far it can travel on a full battery (while being able to return to a charging station), however if another machine is required to process the ROS nodes then the effective range would be further limited by the broadcasting range of the processing machine.

Chapter 7: **Installation and Usage Instructions**

Operating System

Install Ubuntu 16.04 on your machine. (Ubuntu 15.10 and Debian Jessie are compatible with ROS kinetic but have not been used in this project) If your OS does not come with Python 2.7 installed you will need to install it manually, however Ubuntu 16.04 comes with Python installed.

Install ROS

Follow the ROS kinetic installation instructions at `http://wiki.ros.org/kinetic/Installation/Ubuntu`
Complete the installation and configuration tutorials to initialise your catkin workspace - `http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment`

Install ROS Packages

Install the following packages ensuring the kinetic version of ROS is the target distro.
For ROS package links you can find the GitHub repo at the top of the page.

tum_simulator - Follow instructions at `http://wiki.ros.org/tum_simulator#Installation`
ardrone_autonomy - GitHub Repo at `https://github.com/AutonomyLab/ardrone_autonomy#installation`
hector_quadrotor - ROS package at `http://wiki.ros.org/hector_quadrotor`
cv_bridge - ROS package at `http://wiki.ros.org/cv_bridge`

Install OpenCV for Python

Installation from Pre-built Binaries:

```
sudo apt-get install python-opencv
```

Other installation methods available from `https://docs.opencv.org/trunk/d2/de6/tutorial_py_setup_in_ubuntu.html`

Install Gazebo

Installation tutorial with ROS Integration: `http://gazebosim.org/tutorials?tut=ros_installing`
Alternatively,
Accept software from packages.osrfoundation.org

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable
    `lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'
```

Setup keys

```
wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
```

Install Gazebo (7 for ROS kinetic)

```
sudo apt-get update
sudo apt-get install gazebo7
```

Install Project Package

Clone the project repository into your src/ folder in your catkin workspace and build your workspace with *catkin_make*.

Using the Project Package

In order to open a simulation world in Gazebo you can use the following commands:

```
roslaunch drone_follow_ball ardrone_ball.launch
roslaunch drone_follow_ball ardrone_ball_moving.launch
```

The ardrone_ball_moving.launch file will create a simulation environment where the ball follows a path controlled by the Gazebo plugin ball_mover.cc while the other launch file will create a stationary ball. Both launch files create an AR Drone, stationary at the center of the world in a resting state.
In order to track the ball (see Final Deliverable - Ball Finder) you should run the following command:

```
rosrun drone_follow_ball ball_finder.py
```

In order to have the drone follow the ball (see Final Deliverable - Tracking Moving Ball) you can issue the following commands:

```
rosrun drone_follow_ball follow_stationary_ball.py
rosrun drone_follow_ball follow_ball.py
```

The first command was a checkpoint of development which should not be used except for testing purposes. When using the follow_ball.py script you can use the following command to toggle the lateral movement on and off again:

```
rostopic pub -1 /toggle_lateral std_msgs/Empty
```

To control the drone you will need to issue commands to publish to relevant topics from the tum_simulator and ardrone_autonomy packages:

```
rostopic pub -1 /ardrone/takeoff std_msgs/Empty
rostopic pub -1 /ardrone/land std_msgs/Empty
```

While using this package you can check the I/O operations through the following methods: To view the camera output from the drone (and input to the other topics):

```
rosrun image_view image_view image:=/ardrone/front/image_raw
rosrun image_view image_view image:=/ardrone/bottom/image_raw
```

To view information published to topics (where the topic you want to view is <topic_name> including the preceding forward slash, example for the /circles topic included):

```
rostopic echo <topic name>
rostopic echo /circles
```

For a wide overview of communication between topics you can generate an rqt_graph of the currently running topics and nodes using the command:

```
rosrun rqt_graph rqt_graph
```

Bibliography

[1] The HEXO+ Self-Flying Drone, using smart phone GPS tracking to record activities - https://hexoplus.com/

[2] The HEXO+ Drone promotional video showing the drone in action - https://www.youtube.com/watch?v=llr4tDxXiew

[3] A Jumping Sumo drone following a ball using OpenCV - https://www.youtube.com/watch?v=XUTgYqPHDV8

[4] An auto piloting system for the Parrot AR Drone 1.0 & 2.0 from Science Center at Washington & Lee University by Professors Joshua Stough and Simon D. Levy https://github.com/simondlevy/ARDroneAutoPylot

[5] Robot Operating System - http://wiki.ros.org

[6] ROS Package Index - http://www.ros.org/browse/list.php?package_type=package&distro=kinetic

[7] Gazebo - http://gazebosim.org/

[8] OpenCV - https://opencv.org

[9] OpenCV Python Tutorials - https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html

[10] OpenCV Python Tutorial showing the effectiveness of various methods of Template Matching - http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html

[11] The specific calculations for each method of Template Matching in the OpenCV library - https://docs.opencv.org/3.3.1/de/da9/tutorial_template_matching.html

[12] Wikipedia page for Hough Circles Algorithm - https://en.wikipedia.org/wiki/Circle_Hough_Transform#Find_circle_parameter_with_unknown_radius

[13] Hough Circles Transform in OpenCV - https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_houghcircles/py_houghcircles.html

[14] Methodology for detecting circles with HSV and Hough Circles (Code from this tutorial was not used or even looked at in this project, only the methodology was used for inspiration - I have used a similar but not identical methodology) - http://anikettatipamula.blogspot.co.uk/2012/12/ball-tracking-detection-using-opencv.html

[15] ROS Package converting ROS Images to OpenCV images - https://github.com/ros-perception/vision_opencv

[16] ROS Package for simulation AR Drone - http://wiki.ros.org/tum_simulator

[17] ROS Package for simulation AR Drone interaction - http://wiki.ros.org/ardrone_autonomy

[18] ROS Turtle Sim Package for learning ROS - http://wiki.ros.org/turtlesim

[19] Ethical Issues with use of Drone Aircraft - http://csiflabs.cs.ucdavis.edu/~ssdavis/188/Ethical%20Issues%20with%20use%20of%20Drone%20Aircraft.pdf

[20] Issue discovered with controlling the AR Drone via the /cmd_vel topic - https://github.com/AutonomyLab/ardrone_autonomy/issues/116