Aerospace
Engineering.

Aerospace Engineering
Individual Investigative Project

Collision Avoidance Systems for UAVs in a Search and Rescue Swarm

Luke Cowan

May 2016

Supervisor: Dr Lyudmila Mihaylova

Dissertation submitted to the University of Sheffield in partial fulfilment of the requirements for the degree of Master of Engineering

Abstract

Collision Avoidance Systems (CAS) are used in a variety of scenarios; ground robots, air traffic control, and multiple Unmanned Ariel Vehicles (UAVs). The need for CAS is vital for UAVs flying in a swarm as each UAV needs to know where each other UAV is located and then be able to avoid them if the distance between the UAVs become too close. This report creates a Collision Avoidance System for a swarm of UAVs using the Velocity Obstacle method. The Velocity Obstacle method has been developed to be used in a 2-dimensional and also a 3-dimensional space.

Acknowledgments

I would like to thank my supervisor Dr Lyudmila Mihaylova for overseeing this project and giving me access to the Parrot ARDrone, RPLIDAR, and a Raspberry PI.

I would also like to thank James Douthwaite for continuing my work and developing it further.

Table of Contents

Nomenclature

A	Position of Agent A
$\tilde{A}$	Position of tip of Velocity Vector A
A_x ,A_y	x and y coordinates of Agent A
B	Position of Agent B
$\hat{B}$	Radius of Agent B + Agent A
B_x , B_y	x and y coordinates of Agent B
C	Position of Left Hand tangent point
$CC_{A,B}$	Collision Cone AB
c_{xz} c_{yz}	z-intercept in the xz and yz plane
D	Position of Right Hand tangent point
$Grad_{ABxz,}$ $Grad_{AByz}$	Gradient of line AB in xz and yz plane
$R_{x,y,z}$	Rotation Matrix around x, y, z axis
$T_{x,y,z}$	Translation values for x, y, z points
u, v, w	Velocity components in Cartesian Coordinates
V_A	Velocity A
V_B	Velocity B
V_{AB}	Relative Velocity AB
VO	Velocity Obstacle
x, y, z	Position components in Cartesian Coordinates
θ, ϕ, β, γ	Angles for creating Collision Cone
ε, ζ, η	Barycentric Coordinates
κ, ω	rotation angles
λ_{AB}	Length between AB
λ_{AC}	Length between AC
λ_{AD}	Length between AD
$\psi_{AB_{xz}}, \psi_{AB_{yz}}$	Angle between AB in xz and yz plane
Λ_{AB}	Line of V_{AB}

1. INTRODUCTION

1.1 Background to the Project

Unmanned Aerial Vehicles (UAVs) are an ideal solution for use in Search and Rescue scenarios where humans can't easily access the area. For instance, in the aftermath of floods, earthquakes, and hurricanes. In these situations, a team of people can quickly and easily launch a UAV into the skies to assess the damage from an aerial view. Using UAVs is considerably cheaper than using a helicopter to do the same job. For a search and rescue situation a quadcopter would be suited to flying close to buildings and objects and could use the array of sensors on board to scan the area for anyone who may be trapped, it could scan a building's infrastructure to see if the building is safe for humans to enter, and the UAV could drop medical supplies or any survival equipment. There are many possible applications for UAVs in different Search and Rescue scenarios. Another use of UAVs is to use them in a swarm, a collection of UAVs all flying together in one network allowing faster surveillance of an area as the UAVs work together to cover more ground when searching for someone. However, there is a problem with swarms, this is that the UAVs could be flying in a close formation or they could have the need to keep a maintained separation between each agent and to be able to achieve this each UAV agent needs to have an on board Collision Avoidance System.

Collision Avoidance Systems (CAS) are ever more present in today's aircraft, from use in civil aviation to military and also in the growing UAV market. CAS are put in place to try and minimise the likelihood of collisions of aircraft. These collisions could be with other aircraft or with stationary objects, for example: buildings, trees, mountains and hills etc. CAS are needed with UAVs because of the nature of being unmanned and autonomous in most cases, whereas civilian aircraft have a pilot who could carry out an evasive manoeuvre to avoid any potential collisions. For a UAV to fly in civil airspace several requirements have to be met to ensure it is as safe as possible. From the Civil Aviation Authority (CAA) there are two methods to avoid collisions depending whether the UAV is being flown within the 'Line of Sight' of the pilot or not. When the UAV is being flown in the line of sight then the pilot must use the 'See-and-Avoid' method, where the pilot maintains

observation of the UAV and the airspace around it to avoid any obstacles. The accepted maximum distance for Line of Sight is 500m horizontally and 400ft vertically, Section 3.11 of (Authority, 2015). When the UAV is beyond line of sight then the pilot cannot see the UAV and act accordingly to any obstacles, this calls for a 'Sense-and-Avoid' system to be placed onto the UAV. The Sense-and-Avoid system is a sensor or multiple sensors that has capability to sense its surroundings and detect if there is an object in the way that could cause a collision and then send a control signal to manoeuvre away from it. The use of Sense-and-Avoid must show a level of performance that is equivalent or a better standard that a pilot could do without the use of cooperative communication between other UAVs or have previous knowledge of the UAVs flight plans (Geyer, et al., 2008). Overall the demand for a Collision Avoidance System is rapidly increasing as more and more UAVs are being used in the same air space. This project will be looking into different types of sensors used for CAS and developing a chosen sensor to be applicable on a Parrot AR Drone 2.0 as the test UAV.

1.2 Project Aims and Objectives

1.2.1 Aims

The aim of this project is to investigate methods for Collision Avoidance Systems of UAVs in a swarm. In particular, creating and testing a simulation of a Collision Avoidance Systems to be implemented in real time.

1.2.2 Objectives

Objective One: To evaluate current methods to establish the foundations of creating a new Collision Avoidance System.

Objective Two: To create a CAS in two dimensions.

Objective Three: To extend the two dimensional case to three dimensions.

Objective Four: To evaluate the performance of the two CAS.

Objective Five: To investigate the possibilities of implementing a CAS in real time on a UAV.

1.3 Report Overview

A review of the literature surrounding this subject has been completed and presented in section 2. This review was influential in determining the structure of the CAS that would be created and what novelties could be found in this ever changing field.

Section 3 sets the scene to the use of Velocity Obstacles and describes how one is created and also the different current uses.

The main content in this report is in sections 4 and 5. These are the technical chapters describing the process of creating the Velocity Obstacle algorithm for the two and three dimensional case respectively. Section 5 has an additional sub section about other possible methods for creating the algorithm in three dimensions. As this is a novel concept there is a lot of scope into developing the most efficient program.

In section 6 each part of the algorithms is tested to verify that they behave as expected. Also the algorithm as a whole is tested, evaluated, and validated via a number of different tests to see where the performance of the algorithm can be enhanced.

Additional work has been completed in sections 7 and 8. These sections show one way how the ARDrone can be controlled via a Raspberry Pi and how it is possible to use a LiDAR as a sensor for a CAS.

Finally, in section 9 the Conclusion, Project Management, and Self Review are all discussed. These sections summarise the findings from the report along with future possibilities of how the algorithms can be improved and any future work that can develop from this report. In addition to this there is a review on the project management and a self-review of the work completed.

2. LITERATURE REVIEW

There have been a number of studies into different forms of Collision Avoidance Systems (CAS) for both aircraft and Unmanned Aerial Vehicles (UAVs). CAS have also been used in scenarios other than in aviation, for example, Collision Avoidance in cars and small robots. In all these different cases similarities can be drawn from one another and through these similarities this project has combined them to form a new Collision Avoidance System.

2.1 Collision Avoidance Systems

To be able to form a new CAS one must have extensive research into the already existing forms of CAS. A starting point was to look at 'A Survey of Collision Avoidance Approaches for Unmanned Aerial Vehicles' by (Albaker & Rahim, 2009).

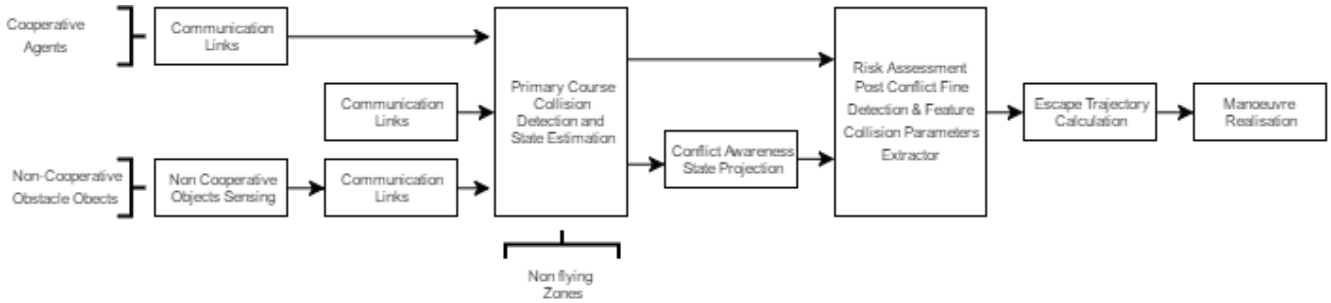


Figure 2.1 Collision Avoidance System Units copied from (Albaker & Rahim, 2009)

In this paper were several useful figures, the first figure, Figure 2.1, shows the different units that form a Collision Avoidance System. It can be seen that different procedures are needed to take place for cooperative and non-cooperative agents. Cooperative agents usually utilise a transponding method where the target agent transmits information about its position and heading. This method is currently employed on all aircraft that fly in controlled airspace, however this method of sensing does not work for any targets that are non-transponding. For example, UAVs that are flying in controlled airspace when they shouldn't be, or other targets like birds, trees, and pylons. Non-cooperative agents need to be sensed in other ways than transponders. Targets can be detected either by passively gaining information about a target, e.g. Optical or Acoustic Sensors, or by actively sending out a pulse to find the target, e.g. Radar or LiDAR (Lacher, et al., 2007). For use of Collision Avoidance in a Swarm it is ideal to use cooperative agents so that each

agent knows exactly where each other agent is. This will reduce the need for complex sensors to locate the agents.

Another interesting point made by (Albaker & Rahim, 2009) is that an automated CAS should have these five key functions: Sensing, Detection, Awareness, Escape Trajectory Estimation, and Manoeuvre Realisation. These functions are then further explored into different sub divisions as shown in Figure 2.2.

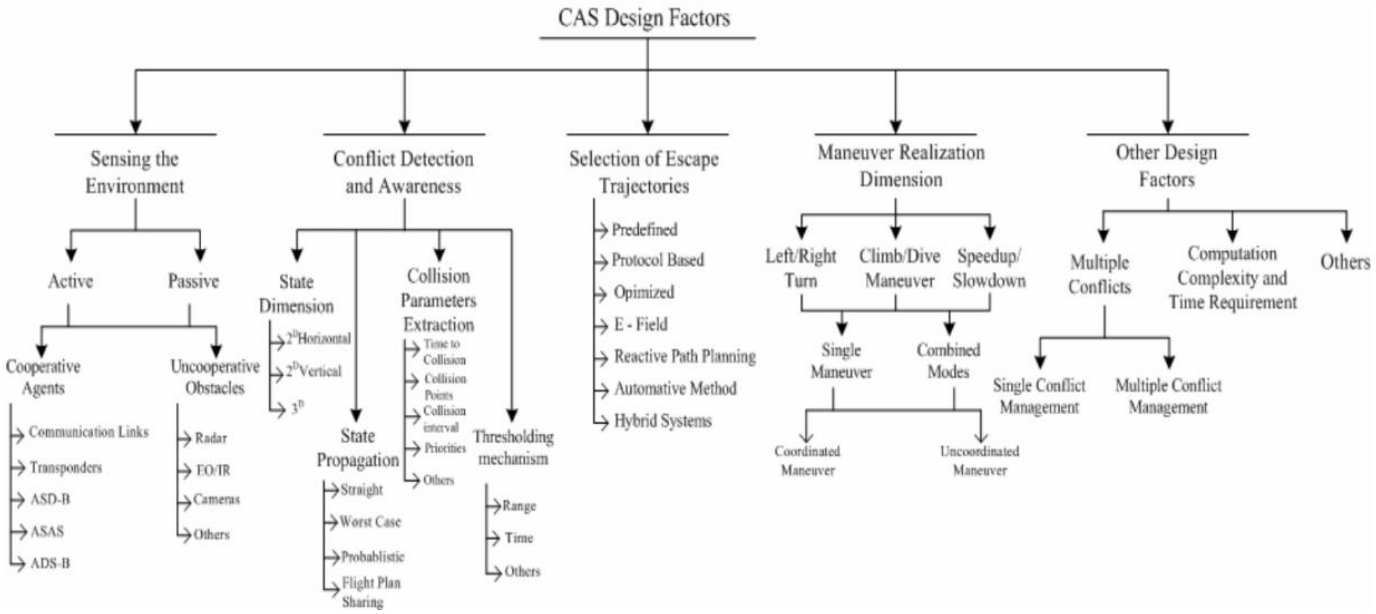


Figure 2.2 CAS Design Factors

The final part of (Albaker & Rahim, 2009) relevant to this study is the list of existing operational Collision Avoidance Systems or ones that have been evaluated in the field. This list shows that CAS are not only just for aircraft but also for ground and maritime vehicles to. These operational CAS are:

- Airborne Information for Lateral Spacing (AILS)
- Country Technical Assistant Service (CTAS)
- Ground Proximity Warning System (GPWS)
- Precision Runway Monitor (PRM)
- Traffic Alert and Collision Avoidance System (TCAS)
- Traffic and Collision Alert Device (TCAD)
- User Request Evaluation Tool (URET)

The main CAS that occurs in many different papers is the Traffic Alert Collision Avoidance System (TCAS), this is because TCAS transponders are installed on most civil aircraft (Geyer, et al., 2008). (Lacher, et al., 2007) proposed some challenges associated with TCAS. In their paper on Unmanned Aircraft Collision Avoidance – Technology Assessment and Evaluation Methods, one of the challenges of TCAS was the fact that collision avoidance technologies must be backward compatible with legacy TCAS and they need to be able to work with transponding and non-transponding aircraft. Another point made was that there was a lack of development of collision avoidance technology meaning that TCAS took almost 20 years to be researched and developed.

2.2 Collision Avoidance System Algorithms

To implement a good CAS one must use an algorithm to provide avoidance manoeuvres to the UAV to be able to avoid any potential collisions. Due to the nature of CAS being implemented into a range of scenarios (Aviation, Maritime, and Ground Vehicles) there are a number of different algorithms that have been produced. This means that a number of different theories could be used to create a new CAS that would have its own advantages and disadvantages. The first Collision Avoidance algorithm researched was that of the Velocity Obstacle Method. This method is used in a lot of collision avoidance for robots in a 2D environment. Therefore, this method is a good starting point to build on at CAS. One paper that provides some impressive concepts is (Fiorini & Shiller, 1998) Motion Planning in Dynamic Environments using Velocity Obstacles where they select avoidance manoeuvres that are able to avoid static and moving obstacles by selecting robot velocities outside of the Velocity Obstacle. Velocity obstacles are a set of velocities that would result in a collision with an obstacle that moves at a given velocity.

Another Velocity Obstacle method is one designed by (Berg, et al., 2008). They proposed a new concept - ‘Reciprocal Velocity Obstacle’, their method considers that each agent navigates independently and without any communication between agents. They also take into account the behaviour of other agents by assuming that other agents have the same collision avoidance algorithms. Berg and Manocha also wrote another paper with Wilkie about Generalized Velocity Obstacles (Wilkie, et al., 2009) which provides more examples of velocity obstacle methods to be used with car-like robots. Through reading these papers a simple velocity obstacle

method could be applied to a swarm of UAVs as the velocity and heading information can be shared between agents.

A different idea for a CAS is that of (John & John, 2015) where a CAS was developed for air traffic management. The basic idea behind this method is that each aircraft is given a box which denotes its safe zone and no aircraft are allowed to have their safe zone overlap with another aircraft's safe zone. This method can be scaled down to be used by UAVs in a swarm where each UAV has a safe zone around it.

(Jikov, et al., 2015) and (Chen, et al., 2015) use the probability of collisions to create an algorithm to avoid any collisions. They implement a Kalman filter with these probabilities to provide avoidance manoeuvres. (Chen, et al., 2015) compares two different detection methods, geometric and probabilistic methods for collision detection on the predicted trajectory. The conflict detection probability is maximised to study the threshold selection for each method. (Jikov, et al., 2015) uses List Viterbi Algorithm to find an optimal and collision free path for the aircraft to follow.

2.3 Sensors used for Collision Avoidance

Now that a CAS algorithm has been chosen there needs to be some sensors that are able to sense if there is an obstacle in a collision course. In (Geyer, et al., 2008) there is a section on different sensors used in Collision Avoidance Systems and it gives some advantages and disadvantages to each sensor. This paper was found to be very useful in understanding each sensors that could be used. The sensors listed in the paper are:

- Electro-Optics
    - Three cameras that can identify potential targets and track them to decide if any represent a threat.
- Infrared
    - Infrared cameras that rely on moving targets for detection. They can't detect low relative velocity targets i.e. targets on a head on collision.
- Acoustic
    - System detects noise coming from an aircraft.
- LiDAR
    - A pulse of laser is emitted and the time taken for the pulse to return is measured and the distance to an object is calculated from that.
- Radar
    - Used for long range collision avoidance on the ground.

(Lacher, et al., 2007) also have a section in their paper on Collision Avoidance about the different sensors that are possible to use. They produced a table that sums up the trade-offs of different sensors. This table is reproduced in Table 2.1.

	Modality	Range	Bearing (Azimuth)	Bearing (Elevation)	Trajectory
Mode A/C Transponder	Cooperative	Accurate; 10s of miles	Calculated	Calculated based on pressure altitude	Derived
ADS-B	Cooperative	Accurate; 10s of miles	Calculated based on GPS	Calculated based on pressure altitude	Provided
Optical	Non-Cooperative, Passive	Not sensed	Accurate	Accurate	Derived
Thermal	Non-Cooperative, Passive	Not sensed	Accurate	Accurate	Derived
Laser/Lidar	Non-Cooperative, Active	Accurate; 1000ft	Narrow	Narrow	Derived
Radar	Non-Cooperative, Active	Accurate; 1 mile	360 degrees	360 degrees	Derived
Acoustic	Non-Cooperative, Active	Accurate; 100ft	360 degrees	360 degrees	Derived

Table 2.1 Reproduced from (Lacher, et al., 2007) Showing different types of sensors

2.4 LiDAR as a Collision Avoidance Sensor

The final section in this Literature review is focused on the use of LiDAR as a Collision Avoidance Sensor. There have only been a few papers written on the use of LiDAR for Collision Avoidance. One of these papers is the Use of LIDAR for Obstacle Avoidance by an Autonomous Aerial Vehicle (Ladha, et al., 2011) which uses the concept of Obstacle Growth Algorithm where once an obstacle is detected then every point in the scanning range is considered to be a point obstacle. This point obstacle is then extended by half the clearance width of the vehicle, this is then used to make sure that the UAV will clear the obstacle with enough space.

A different paper on LIDAR Obstacle Warning and Avoidance System for Unmanned Aircraft (Sabatini , et al., 2014) uses LIDAR to be able to detect power cables showing the ability of LIDAR to detect thin objects as well as larger obstacles such as trees, walls, or other UAVs.

3. THE DEVELOPED COLLISION AVOIDANCE ALGORITHMS FOR A SMALL SWARM OF UAVS

Behind any Collision Avoidance System there is an algorithm which governs the behaviour of the system. These algorithms can be fine-tuned to give the best desired response specific to the needs of the system. There are different algorithms more suited to different environments and the algorithm of choice is the Velocity Obstacle method implementing the use of Collision Cones. The reason for this choice is detailed later in this section.

3.1 Collision Cones

Collision Cones are geometric representations of all the possible velocity vectors that could result in a collision. This section uses the main principle set up in (Fiorini & Shiller, 1998). For simplicity each agent is represented as a circle and the position and velocities of each agent are known. The first step in creating a collision cone is to reduce the size of agent A to a point and then increase the size of agent B by the radius of A thus representing the sizes of two agents at one point.

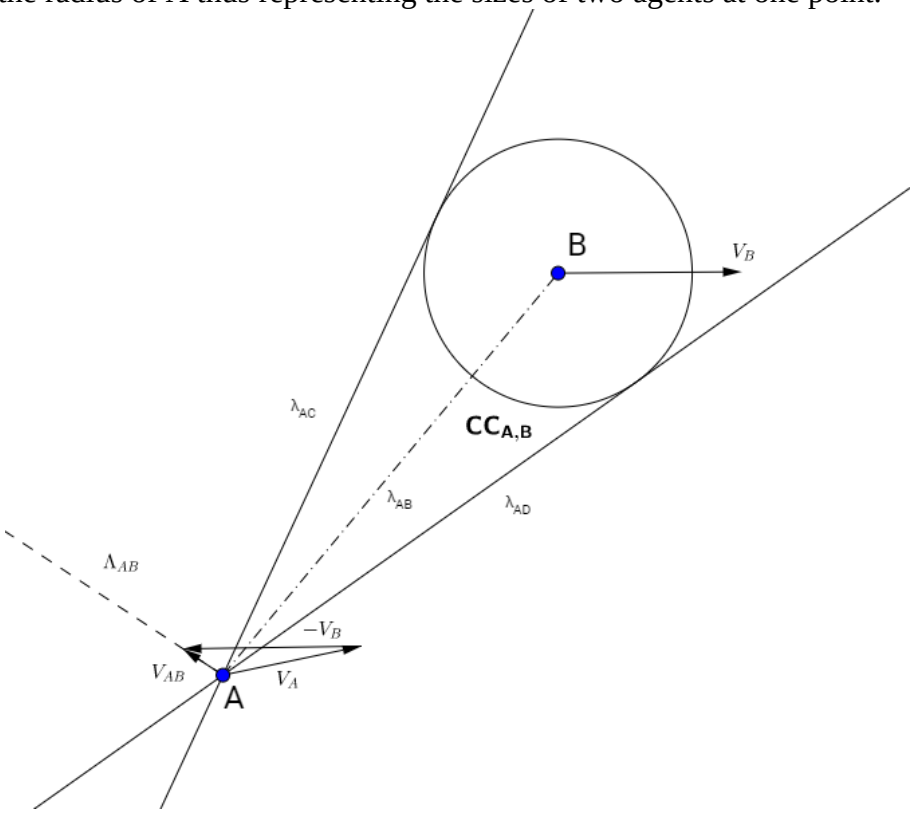


Figure 3.1 Collision Cone $\boldsymbol{CC_{A,B}}$ *showing the relative velocity* $\boldsymbol{V_{AB}}$

The next step is to create two tangents, λ_{AC} and λ_{AD}, to the circle from point A, this is where the collision cone gets its name from. The collision cone, $\mathbf{CC_{A,B}}$, is the set of colliding relative velocities between A and B. $\mathbf{CC_{A,B}}$ can be defined by the following equation 3.1.

$$CC_{A,B} = \{V_{A,B} | \lambda_{AB} \cap B \neq \emptyset\} \quad (3.1)$$

Where $\boldsymbol{v}_{A,B}$ is the relative velocity of A with respect to B, $V_{A,B} = V_{\boldsymbol{A}} - V$, and λ_{AB} is the line of $\boldsymbol{v}_{A,B}$. This shows that any relative velocity that lies within the collision cone will result in a collision between the two agents.

It is easier for the algorithm to work with absolute velocities of agent A instead of relative velocities. This is achieved by adding the velocity of agent B, $\boldsymbol{v}_B$, to each velocity in $CC_{A,B}$. Similarly, by translating the collision cone by $\boldsymbol{v}_B$.

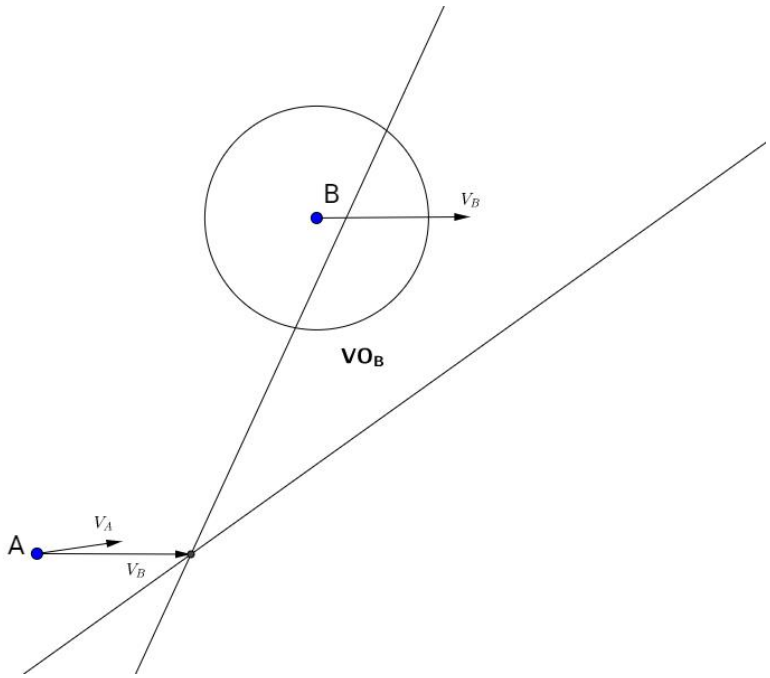


Figure 3.2 Velocity Obstacle $\boldsymbol{VO_B}$

This is now referred to as a Velocity Obstacle, VO, and is defined as:

$$VO = CC_{A,B} \oplus \boldsymbol{v}_B \quad (3.2)$$

Where $\oplus$ is the Minkowski vector sum operator (Oaks, 2005). Now the algorithm can calculate a suitable absolute velocity for agent A that does not lie inside the VO.

3.1.1 Multiple Velocity Obstacles

To avoid multiple agents, multiple velocity obstacles are created, e.g. if there were 3 agents A, B_1, and B_2, agent A would have a VO between itself and B_1 and also between itself and B_2. This can be expressed as the union of all the VOs:

$$VO = \cup_{i=1}^{n} VO_{B_i} \tag{3.3}$$

Where n is the number of agents. This means that now the algorithm will produce an absolute velocity of agent A that will not coincide with any of the VOs. There can be an occasion where there are numerous agents flying around in a swarm creating a lot of velocity obstacles. It can be advantageous to factor in a priority system where the agents that would collide first have top priority. This can be achieved by setting a time horizon dependant on the avoidance dynamics. This time horizon can then be subtracted from the VO to create a new VO_h which factors in the priority system.

3.2 Use of Velocity Obstacles

The principle of Velocity Obstacles can be applied in a variety of different scenarios all involving different objects moving. Some examples of these scenarios are; using Velocity Obstacles to model the behaviours of large crowds of people moving through tight spaces, for example, building evacuations, exiting a stadium, and movement through a congested City (Guy, et al., 2009). VOs can be used to model automatic cars joining a main road from a slip road (Fiorini & Shiller, 1998). The main use of VOs is for collision avoidance in small robots and these scenarios have been covered in many papers, such as (Berg, et al., 2008) covering VOs with added acceleration constraints, (Wilkie, et al., 2009) expanding VOs to take account of the constraints of a car-like robot, and (Berg, et al., 2011) creating VOs for *n* amounts of robots all avoiding each other.

The use of Velocity Obstacles was chosen for use in the scenario of multi-agent swarm UAVs as the concept has not been as widely used as it has in ground robots. Also, the use of geometry to predict whether a certain velocity will result in a collision makes it simpler to fine tune the algorithm and to debug as it can be easily visualised how the VO should behave.

4. VELOCITY OBSTACLE ALGORITHM IN TWO DIMENSIONS

There are two algorithms created that implement the Velocity Obstacle method one for a two dimensional case and one for an experimental three dimensional case. This chapter goes into detail on how the 2D algorithm was created step by step, the source code for the algorithms can be found in Appendix A. The algorithms are based on the work done by Fiorni and Shiller but have been adapted for the use of a multi-agent UAV swarm scenario. The algorithms were written in MATLAB and contain several scripts to run. The first script is *Agent_Collision_Avoidance.m*, this makes use of the work by (Douthwaite, et al., 2016) of being able to set up multiple agents which gives them an initial state of a vector containing *x, y, z* position and *u, v, w* velocities. This script combines all the positions and velocities for each agent into a packet and then calls the function *Agent_Avoid()* with the required input of the packet and the size of the agents. *Agent_Avoid()* is where the iterations for the simulations take place with each of the agent's packets being placed into the next function, *Avoid_Algorithm_2D* for the 2D case. These are the main algorithms where the VOs and the new velocity are created.

There are a couple of assumptions in this algorithm. The first is that the agent can stop its movement and then translate by any velocity vector within ±5m/s in any direction. Secondly, each agent is represented as a circle even if the real agent is not spherical. The radius of the circle would be the widest part of the real agent. Finally, in each iteration of the simulation the new position of the agents is the sum of the current position and its velocity vector.

4.1 Creating the Cone in 2D

To create the Collision Cone shown in Figure 4.1, first the distance between the two agents, λ_{AB}, is to be calculated, shown by equation 4.1.

$$\lambda_{AB} = \sqrt{(B_x - A_x)^2 + \left(B_y - A_y\right)^2} \tag{4.1}$$

Now the lengths of the two tangents that create the cone can be calculated using equation 4.2. Using circle theorem it can be shown that the lengths of two tangents from a point to a circle are equal, therefore $\lambda_{AC} = \lambda_{AD}$.

$$\lambda_{AC} = \lambda_{AD} = \sqrt{\lambda_{AC}^2 - \widehat{B^2}} \quad (4.2)$$

The next step is to calculate the angles, θ, ϕ, β, and γ so the positions of the tangent points *C* and *D* can be found.

$$\theta = \sin^{-1}\left(\frac{\hat{B}}{\lambda_{AC}}\right) \quad (4.3)$$

$$\phi = \cos^{-1}\left(\frac{|B_x - A_x|}{\lambda_{AB}}\right) \quad (4.4)$$

$$\gamma = \theta + \phi \quad (4.5)$$

$$\beta = \gamma - 2\theta \quad (4.6)$$

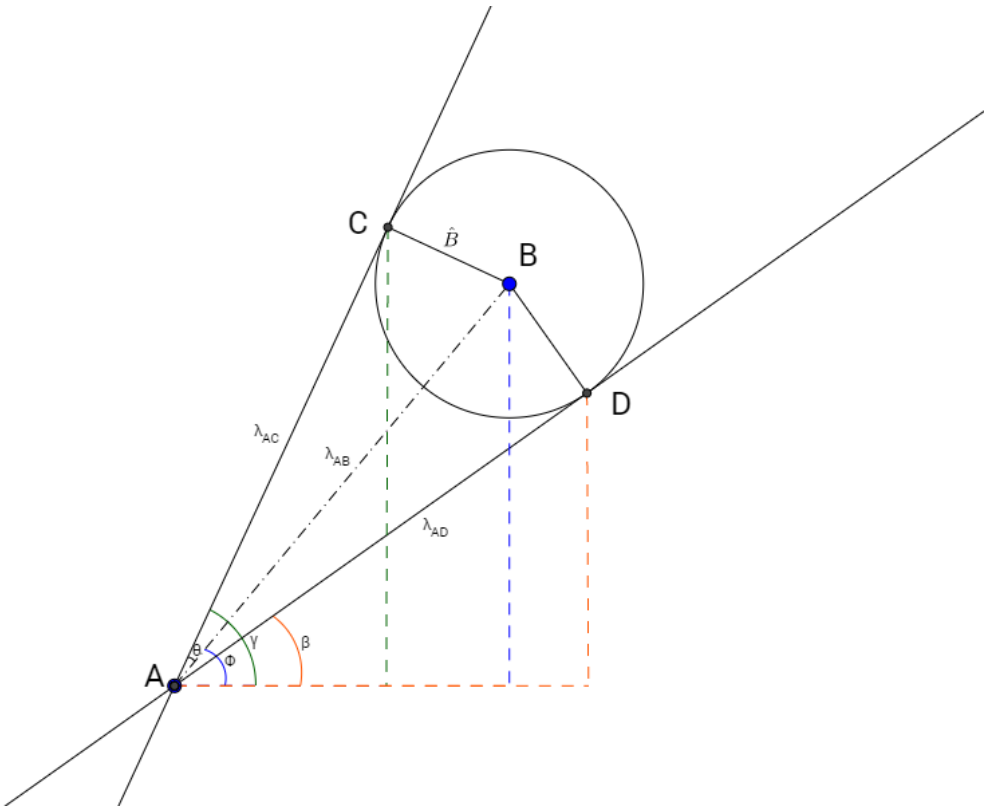


Figure 4.1 Collision Cone showing the location of the angles

To calculate the positions of the tangent points *C* and *D*, the location of *B* relative to *A* needs to be known. There are four main regions that *B* can be relative to *A* and four more boundary cases. These regions are shown in Figure 4.2 and 4.3, they are also expressed by the equations 4.7 through to 4.10.

$$Quad\ 1 = (B_x \geq A_x) \wedge (B_y \geq A_y) \quad (4.7)$$

$$Quad\ 2 = (B_x \geq A_x) \wedge (B_y < A_y) \quad (4.8)$$

$$Quad\ 3 = (B_x < A_x) \wedge (B_y \leq A_y) \quad (4.9)$$

$$Quad\ 4 = (B_x < A_x) \wedge (B_y > A_y) \quad (4.10)$$

The boundary cases have been created as there becomes a case where points *C* and *D* are not in the same quadrant, therefore different equations are needed to obtain the locations of *C* and *D*.

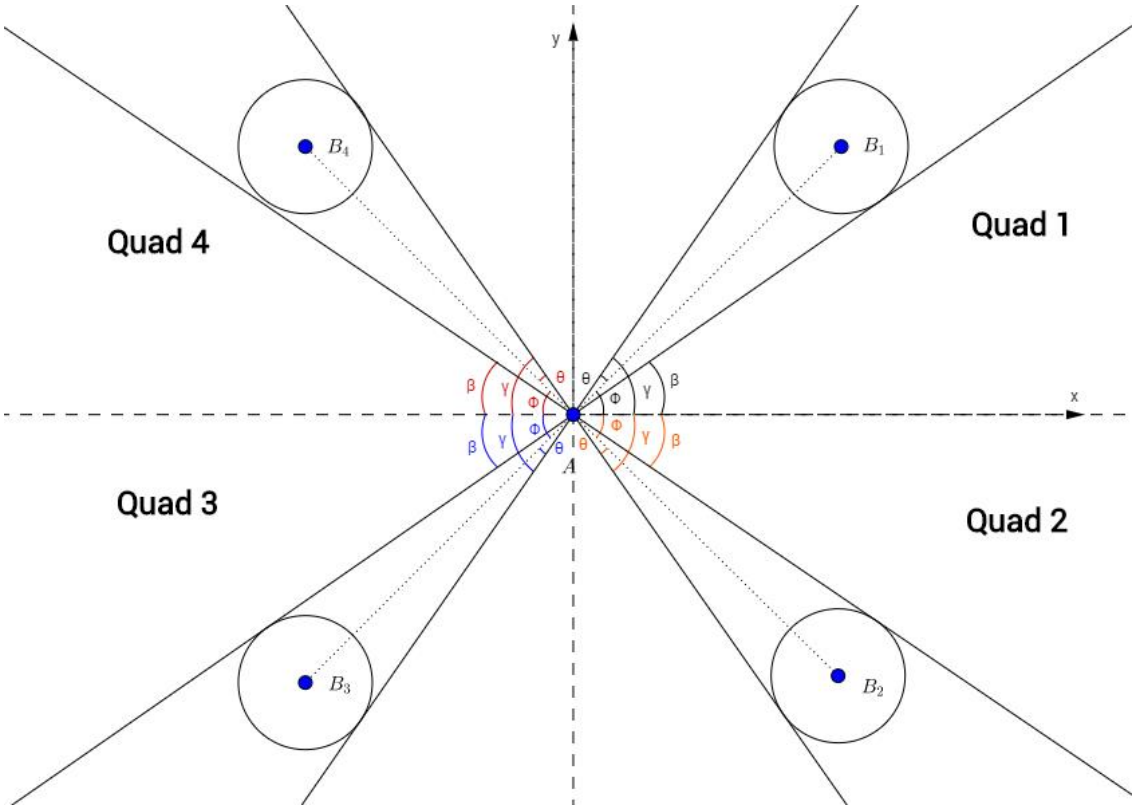


Figure 4.2 The 4 quadrants B could be in with the corresponding angles

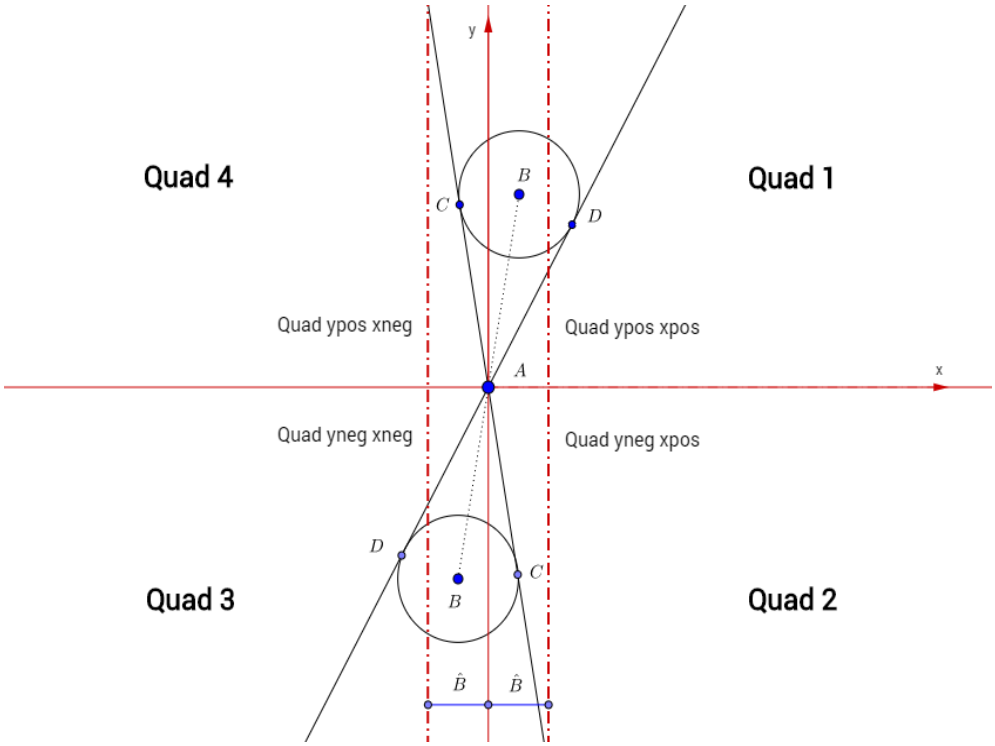


Figure 4.3 Examples of two of the Boundary Cases

Depending on which quadrant *B* is in relative to *A*, the equations needed to calculate the positions of tangent points *C* and *D* are different because of the way the angles are always relative to the x axis. The method to calculate the positions is by using trigonometry to find the distance from *A* to *C or D*, then add that distance to the position of *A*.

This method only gives a cone up to the tangent points, but it is desired to have the cone extend further. In theory the cone would stretch to $\pm \infty$ but this is impractical due to the agents having physical limitations as to how fast they can accelerate. In this case the cone was extended by an extra 3 m/s in the x direction. To extend the cone the gradient of the line is found followed by the y-intercept. This allows the equation of the line in the form $y = mx + c$ to be obtained. From there an x-coordinate is chosen at the desired extension distance and put into the equation of the line to obtain the new y-coordinate that would result with a point on the line. The final part of creating the Collision Cone is to turn it into a Velocity Obstacle by adding the velocity vector $\boldsymbol{V_B}$ to each point on which makes up the Collision Cone.

4.2 Calculating a New Velocity in 2D

The next section of the algorithm is to calculate if the current velocity vector lies within the VO. If it does, then a new velocity vector needs to be chosen. To start, all the possible velocities from agent A are calculated shown in Figure 4.4.

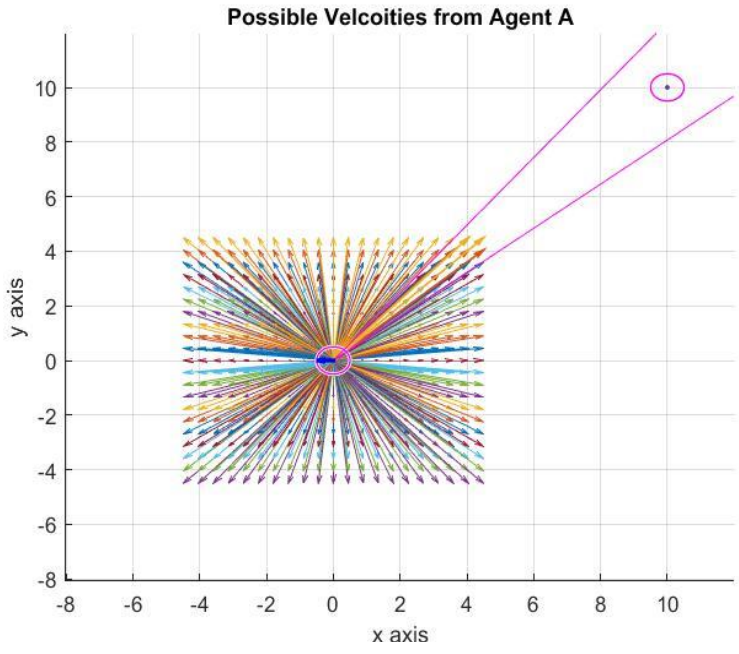


Figure 4.4 Possible Velocities from Agent A

There is an arbitrary constraint that agent A can only move a maximum of 5 m/s in any direction in the x-y plane. This can be changed depending on the constraints of the specific agent being used. For simplicity it has been chosen that the possible velocities for agent A range from -5 m/s to +5m/s in 0.5m/s intervals. The resolution of the potential velocities can be increased, however, this increases the computational load on the system. The resolution could be an initial parameter on the makeup of that particular agent.

To create all the potential velocities a nested for loop is used and for each iteration the variable p is given the x and y coordinates of the position of the tip of the velocity vector. Now the position is known, it needs to be checked to see if it lies within the VO. To do this the barycentric coordinates of the point needs to be calculated by using equation 4.11, where A, C, D are the vertices of the VO and ε, ζ, η are the barycentric coordinates of the point p.

$$p = \varepsilon A + \zeta C + \eta D \tag{4.11}$$

The barycentric coordinates of the point p lies within the triangle ACD if $0 \leq \varepsilon, \zeta, \eta \leq 1$. The barycentric coordinates ε, ζ, η can be calculated from the coordinates of the vertices ACD using equations 4.12 and 4.13, then rearranging to find ε, ζ, η in equations 4.14 to 4.16, where x, y are the Cartesian coordinates of point p and $x_{1,2,3}$, $y_{1,2,3}$ are the Cartesian coordinates of points A,C,D respectively.

$$x = \varepsilon x_1 + \zeta x_2 + \eta x_3 \tag{4.12}$$

$$y = \varepsilon y_1 + \zeta y_2 + \eta y_3 \tag{4.13}$$

$$\varepsilon = \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)} \tag{4.14}$$

$$\zeta = \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)} \tag{4.15}$$

$$\eta = 1 - \varepsilon - \zeta \tag{4.16}$$

For each point in the iteration the barycentric coordinates can be calculated and then tested to see if they are greater than or equal to 0 and less than or equal to 1. If the point satisfies the criteria, then this point is a collision point. The collision velocity is calculated as the difference in position of the collision point and agent A. This collision velocity is then stored in a matrix of collision velocities. If the point tested doesn't satisfy the criteria the same process is repeated but the velocity is stored in a matrix of non-collision velocities.

For when there are multiple agents in the simulation, Vel_Collide is outputted from the *Avoid_Algorithm* function as Vel_Collide_Prev and is then used the next time *Avoid_Algorithm* is called. These two values, Vel_Collide and Vel_Collide_Prev are then combined together in a union to form a new Vel_Collide matrix. Figure 4.5 shows how the VOs overlap and the union of collision velocities is needed.

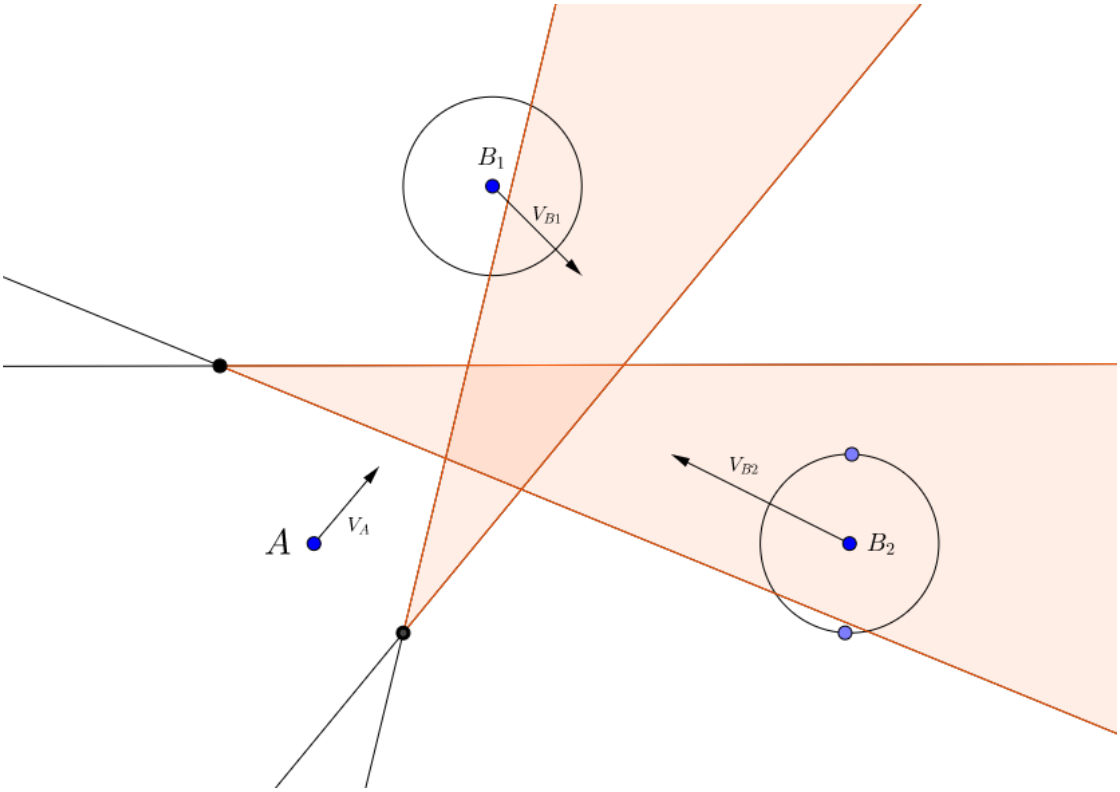


Figure 4.5 The Union of two Velocity Obstacles

The final step is to calculate a new velocity if the original velocity will result in a collision. First the algorithm checks whether the original velocity is not a member of the collision velocities. If it is not a member, the velocity of A is set to be the original velocity. This is to make the agent head in the same direction as it did when the simulation was started. If the original velocity is a member of the collision velocities, it then checks to see if the current velocity is a member. If the current velocity is a member, then the algorithm works out the nearest velocity to the current velocity that is not a member of the collision velocities.

To find the new velocity the algorithm goes through each value in the non-collision velocity matrix and calculates the distance between the no collide velocity and current velocity and stores this in a matrix. The MATLAB function *min()* is used to obtain the minimum value in the matrix created from the difference in velocities.

The *min()* function outputs the index position in that matrix of the minimum value, therefore the new velocity is at that index position in non-collision velocity matrix.

This Collision Avoidance Algorithm is called on several times depending on how many agents there are in the simulation. For example, if there are 3 agents then the algorithm needs to be called 6 times as each agent needs to create a VO with all the other agents.

5. VELOCITY OBSTACLE ALGORITHM IN THREE DIMENSIONS

Previous works of Velocity Obstacles have only looked into the two dimensional case (Large, et al., 2004) (Fiorini & Shiller, 1998) (Shiller, et al., 2001). This report looks into the concept of creating VOs for quadcopters in a three dimensional space. There has been limited work completed in this topic, one example is (Yazdi, et al., 2015) where they have used the Velocity Obstacle method for UAVs in a larger airspace.

This chapter describes a different method to calculate a new velocity. The 3D algorithm uses the same concept of creating a VO with each agent, however this time the cone is in a three dimensional space. The algorithm for calculating the collision velocities and a new velocity are a novel concept created for the use in this report. An example of the algorithm is presented in Appendix B.

5.1 Creating the Cone in 3D

The method for creating the cone in £D differs slightly from the 2D case as the addition of the z axis creates additional angles that need to be calculated. One way

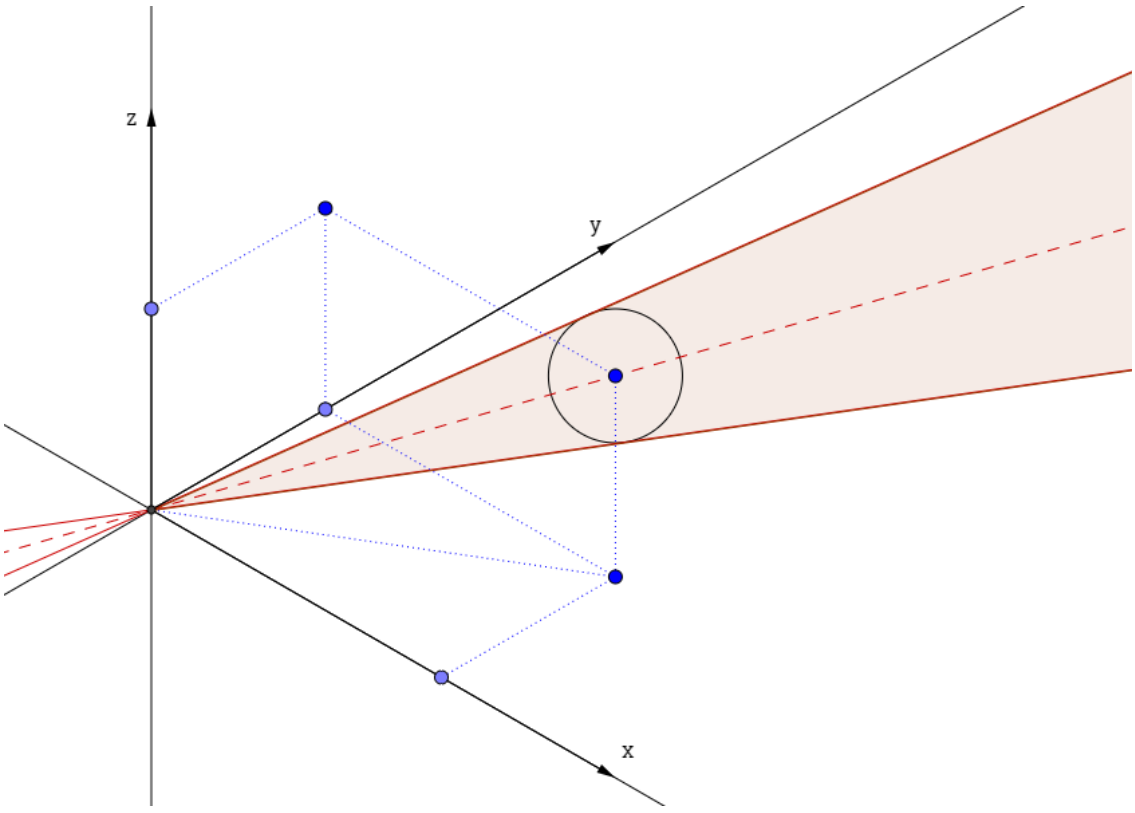


Figure 5.1 Figure of the 3D cone

to visualise and manipulate the cone is to use the xz plane and the yz plane shown in Figures 5.1, 5.2, and 5.3.

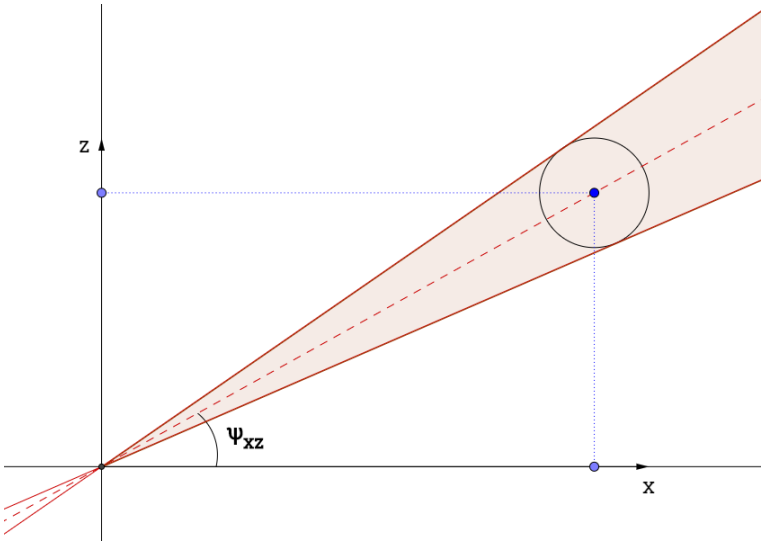


Figure 5.2 3D cone in the xz plane

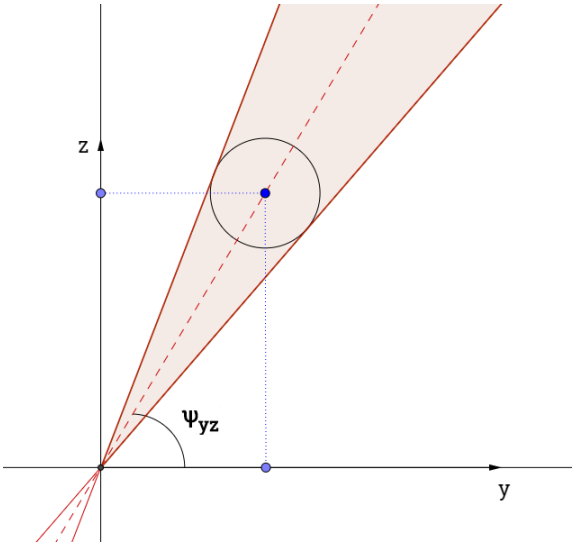


Figure 5.3 3D cone in the yz plane

The x, y, and z coordinates can be found by using the agent B's position, which is the circle shown in Figures 5.1 to 5.3, and the angle ψ_{xz} or ψ_{yz}, depending on which coordinate is being calculated. Similarly to the 2D case, the position of agent B relative to agent A is needed to be known in order to calculate the angles ψ_{xz} or ψ_{yz}. There are 10 possible locations agent B could be relative to agent A. There are 4 quadrants in the xz plane and another 4 in the yz plane. The last 2 locations have been called Quad0_xz and Quad0_yz, these locations are when agent B shares the same x and z coordinates or y and z coordinates. These locations are shown in Figure 5.4 and 5.5.

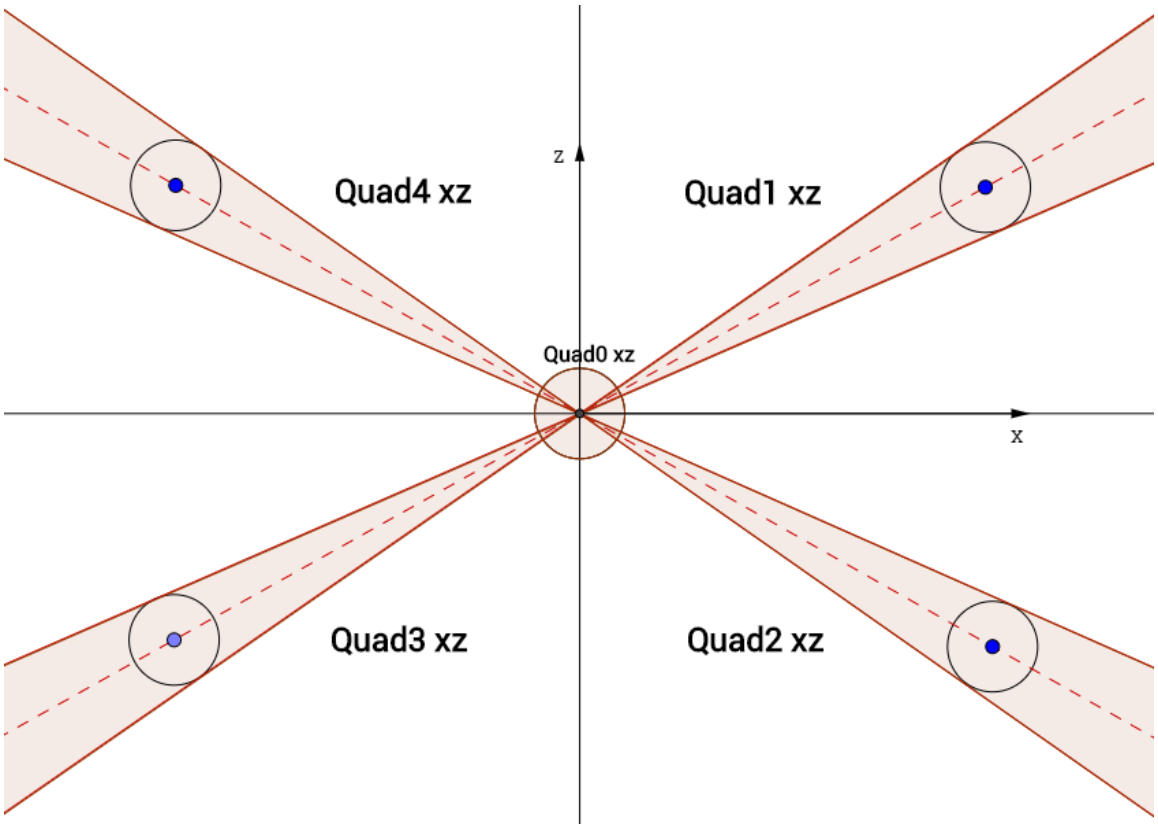


Figure 5.4 Locations of Agent B relative to Agent A in the xz plane

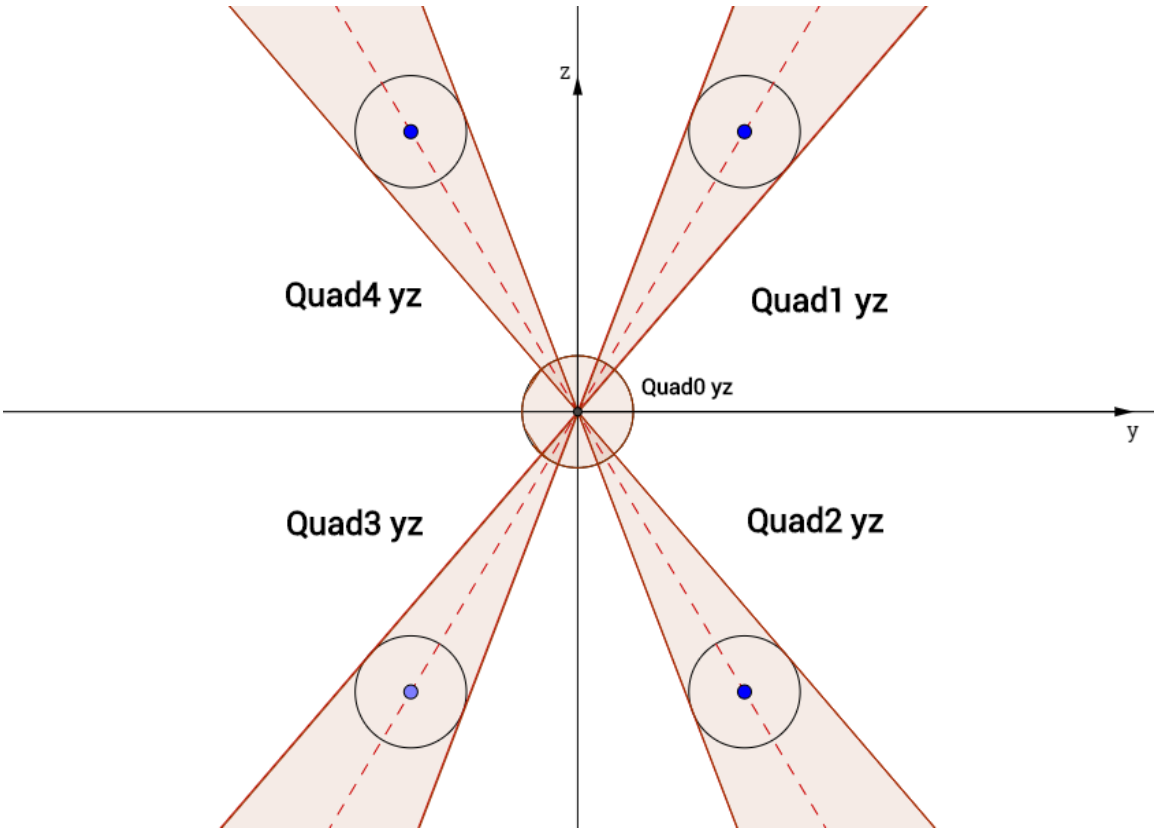


Figure 5.5 Locations of Agent B relative to Agent A in the yz plane

Now the position of agent B relative to agent A is known, the position of B can be extended to a new point that lies on the line AB. This is to extend the cone past agent B allowing a longer cone to be generated. The reason the cone is extended is because when the two agents are close together the cone is very short, so there are potential velocities missed in the calculation for choosing a new velocity. With an extended cone there become more choices of velocities for the algorithm to pick. The method to extend the cone is to calculate the gradient of the line AB in the xz plane and the yz plane. This is done with equations 5.1 and 5.2. Then the point at which the lines intercept the z axis are calculated from equations 5.3 and 5.4.

(5.1) $$Grad_{AB_{xz}} = \frac{A_z - B_z}{A_x - B_x}$$

(5.2) $$Grad_{AB_{yz}} = \frac{A_z - B_z}{A_y - B_y}$$

(5.3) $$c_{xz} = A_z - Grad_{AB_{xz}} * A_x$$

(5.4) $$c_{yz} = A_z - Grad_{AB_{yz}} * A_y$$

The equation of the line AB in the xz and yz planes are now known. Depending on which quadrant agent B is in determines if the x coordinate should be increased or decreased by the desired extension distance. There is now enough information to calculate the y and z coordinates of the new position of point b.

(5.5) $$b_z = Grad_{AB_{xz}} * b_x + c_{xz}$$

(5.6) $$b_y = \frac{b_z - c_{yz}}{Grad_{AB_{yz}}}$$

There are several scenarios when this method will not work as the gradients in the xz or yz planes could equal $\pm \infty$ or NaN (Not a Number). These scenarios are:

- When the x AND y coordinate of A and B are the same, i.e. when agent B is directly above or below agent A.
- When just the x coordinate of A and B are the same, i.e. when agent B is just in the yz plane of agent A.
- When just the y coordinate of A and B are the same, i.e. when agent B is just in the xz plane of agent A.
- When just the z coordinate of A and B are the same, i.e. when agent B is just in the xy plane of agent A.

In each of these scenarios the previous method to extend to the new position is modified to calculate the gradient of the line AB in a 2D plane as one of the planes would be constant in the scenario.

5.1.1 Creating Discs inside the Velocity Obstacle

The next stage in the avoidance algorithm in three dimension is to calculate the velocity of the collision points that fall in the cone. This is achieved by dividing the cone up into *n* amount of discs which are evenly distributed along the cone. Figure 5.6 shows how the discs stack on top of each other to represent the cone.

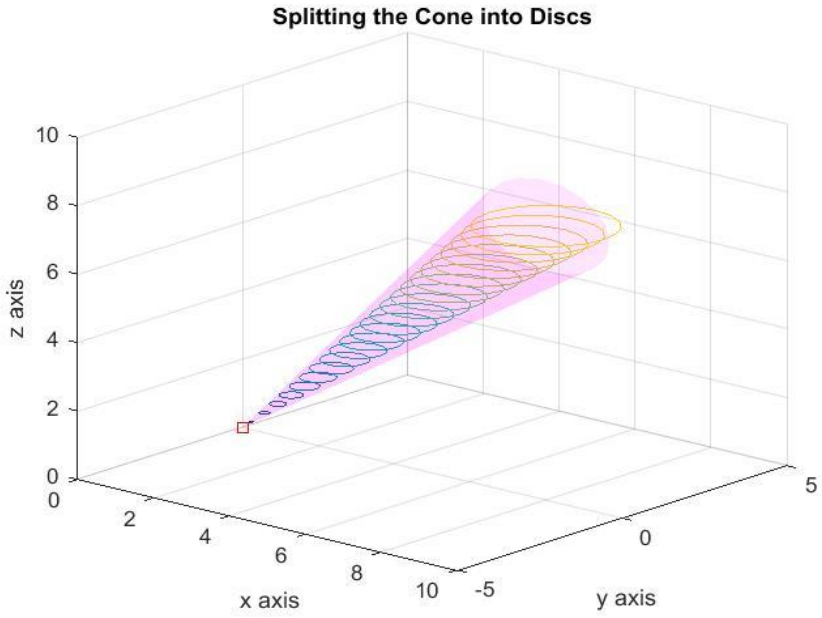


Figure 5.6 Splitting the cone into discs parallel to the xy plane

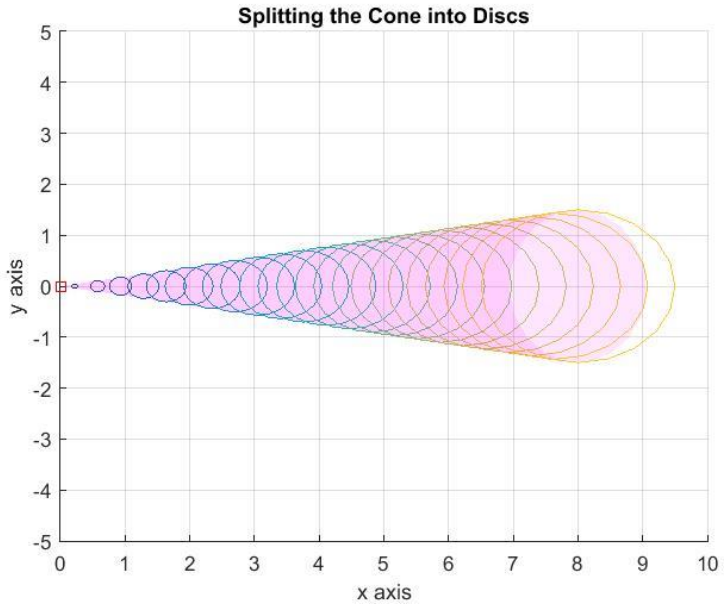


Figure 5.7 View of the discs in the xy plane

The discs do not have the same radius as the cone in the xz plane, but when observed in the xy plane as in Figure 5.7 it is shown that the radius does match with the cone. This is because of the angle of agent B to agent A, it would create an ellipse if the cone was cut along the xy plane. This does not affect the role of the Collision Cone, as the radius of agent B has been increased to 1.5 times the size that it needs to be. This is to allow agents to pass each other with sufficient room and to avoid grazing another agent. The algorithm creates a for loop from 0 to the length of the cone at *n* intervals, where *n* can be changed for the desired amount of discs created in the cone. It then calculates the centre of each disc, point *C*, in x, y, and z coordinates by using the angles ψ_{xz} , ψ_{xz} and the length from A to *C*, shown in Figure 5.8.

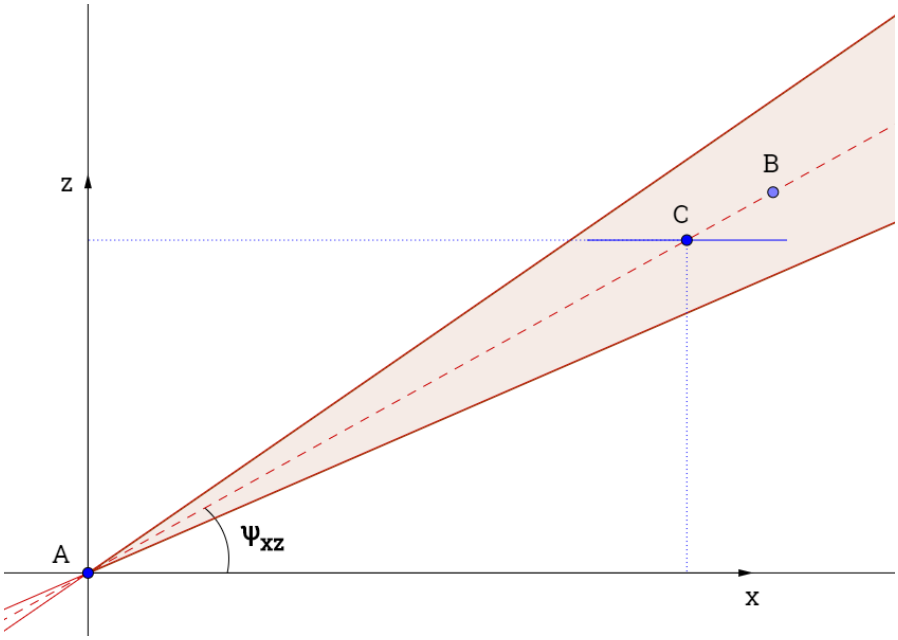


Figure 5.8 Calculating the Centre point C of a disc

Once the location of the centre points has been calculated, the algorithm then uses a nested for loop to create circles with reducing radii inside the disc and also points around the circumference of the circle at set intervals, shown in Figure 5.9, which can be changed to increase the number of points within the cone.

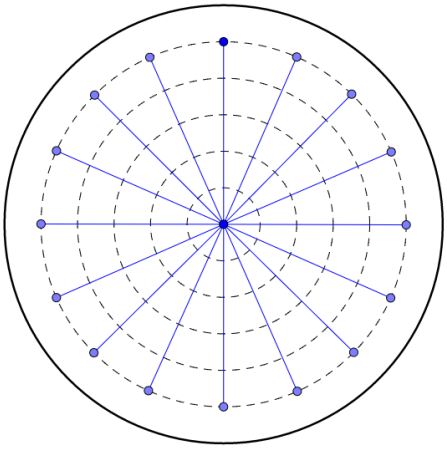


Figure 5.9 Circles inside the Cone with decreasing radii

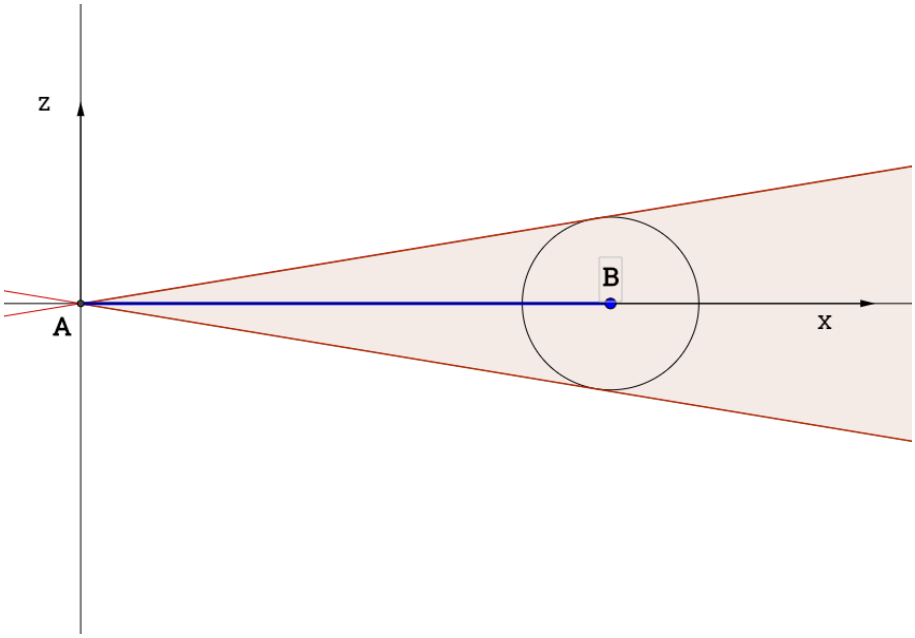


Figure 5.10 When the z component of A and B are the same

There is a major problem with this method of creating the discs. The value of z is constant for each disc and the distance between each disc is dependent on the change in the z component of agent A and B. This means when agent A and B are horizontal to each other, i.e. when the z components are the same, there would only be one disc, shown in Figure 5.10. As there is only one disc in this case a lot of space above and below the disc is left, therefore this method of creating discs inside the VO is not effective and a new method is required.

5.1.2 Another Method to Create Discs Inside the Velocity Obstacle

This other method for creating discs inside the Velocity Obstacle uses translation and rotation matrices. The reason behind using these matrices is that it is desired to have the discs perpendicular to the line from agent A to agent B, shown in Figure 5.11. This way wherever agent B is relative to agent A there will always be *n* amount of discs, depending on the precision. To create the discs it is easiest to translate the VO to the origin then rotate it either to be in the positive or negative direction depending on whether agent B's z coordinate is greater or less than agent A's. This is so that the z coordinate is constant when the discs are created.

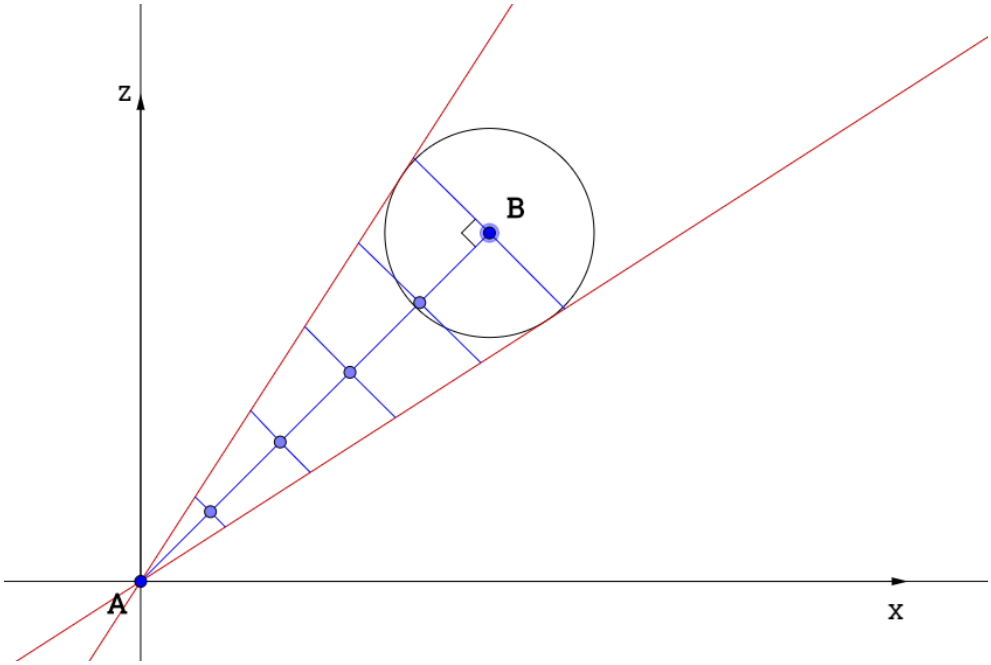


Figure 5.11 Discs Perpendicular to the line AB

There is a set procedure required to complete this method, (Murray, 2013).

1. Translate agent A to the origin
    a. Translate agent B by same amount
2. Rotate agent B by κ around the z axis
3. Rotate agent B by ω around the x axis
4. Create *n* amount of discs along the Velocity Obstacle which should only be along the z-axis
5. Rotate agent B and all the points in the discs back to the original orientation

6. Translate agent A and B and all the points in the discs back to the original positions

Figure 5.12 shows steps 2 and 3 in the above procedure. The angles are calculated by the following equations.

$$\kappa = \tan^{-1}\frac{B_x - A_x}{B_y - A_y} \tag{5.7}$$

$$\omega = \tan^{-1}\frac{\sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}}{B_z - A_z} \tag{5.8}$$

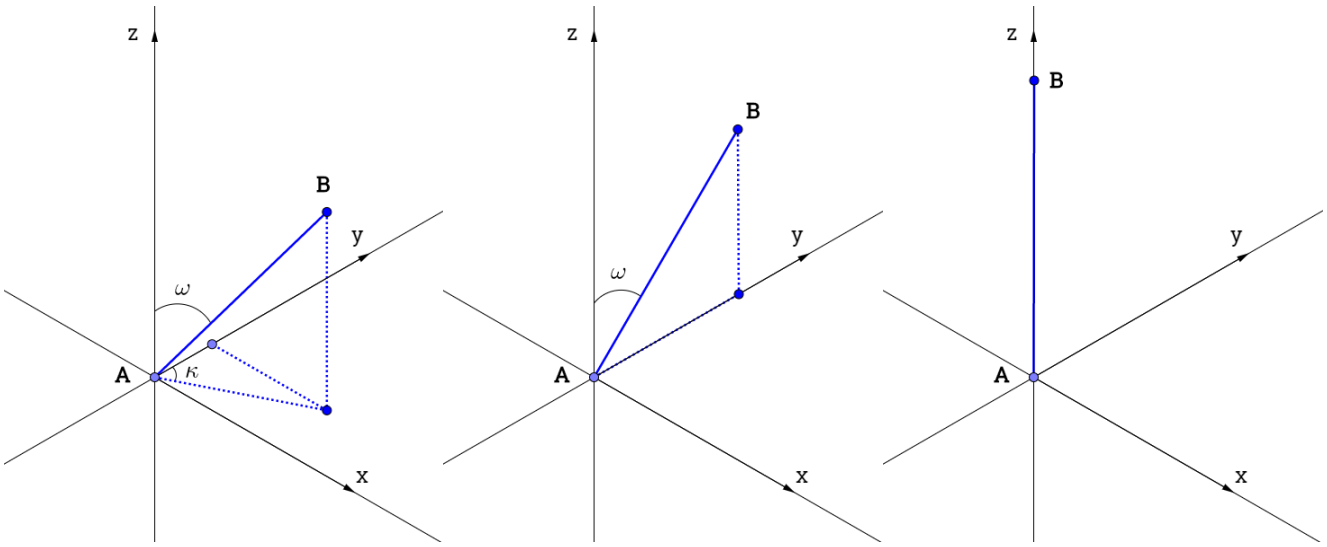


Figure 5.12 Rotations around the z and x axis

The translation matrix is a 4x4 matrix with the required translation in the top 3 positions of the last coloum and the rest of the matrix looks the same as the identity matrix (Murray, 2013). This has the affect of adding the translation values to a point if the translation matrix was multiplied by a point, shown in Figure 5.13.

$$\begin{aligned} G_x &= H_x + T_x \\ G_y &= H_y + T_y \\ G_z &= H_z + T_z \end{aligned} \qquad \begin{bmatrix} G_x \\ G_y \\ G_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} H_x \\ H_y \\ H_z \\ 1 \end{bmatrix}$$

Figure 5.13 Example of a Translation Matrix

There are two ways in which the line AB can be rotated to lie just in the z axis. The first is to rotate the line AB around the z axis to lie along the y axis and then rotate again, this time around the x axis. The second is to rotate the line AB around the z

axis to lie along the x axis and then rotate again, this time around the y axis. The algorithm uses the first method, therefore only requiring rotation matrices around the z and x axis, R_x, R_z. There is one caveat and that is depending whether agent A's y coordinate is less than agent B's. If it is then R_x is as described in equation 10. If it is not, then the signs on the two *sin(ω)* switch.

(5.9) $$R_z = \begin{bmatrix} \cos(\kappa) & \sin(\kappa) & 0 & 0 \\ -\sin(\kappa) & \cos(\kappa) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(5.10) $$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\omega) & \sin(\omega) & 0 \\ 0 & -\sin(\omega) & \cos(\omega) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now the line AB has been translated and rotated to lie just along the z axis, the centre coordinates of the discs which will make up the cone can be calculated. As the line AB is just on the z axis the x and y coordinates of the centre point will be 0. The z coordinates of the centre points are determined by the precision and the length of the line AB, i.e. the centre points will be evenly spaced along the line AB.

For each centre point the algorithm uses a nested for loop to create circles with reducing radii inside the disc and also points around the circumference of the circle at set intervals, shown in Figure 5.9, which can be changed to increase the number of points within the cone.

5.2 Calculating a New Velocity in 3D

Now that the points inside the cone have been calculated, the algorithm can work out whether the current velocity is a collision velocity and if it is, calculate a new velocity which is not a collision velocity. The method used is slightly different again to the 2D case. This time all the points in the cone are stored in the variable *Vel_Collide* and the no collide velocities are calculated by comparing all the points that are up to the size of the radius of B away from $\widetilde{A}$, where $\widetilde{A}$ is the position of agent A + the velocity vector of agent A, i.e. $\widetilde{A}$ is the point at the tip of the velocity vector. This is done because the Velocity Obstacle's radius at any point along the line AB is less than or equal to the radius of B. This means that there will always

be a point less or equal to the radius of B away from $\widetilde{A}$ for there to be a non-collision velocity vector to this point.

The first condition for the algorithm to calculate a new velocity is to determine whether the original velocity at the start of the simulation is not a member of the collision velocities and is not within the set of collision velocities. If it is not a collision velocity, then the new velocity is the original velocity, this is to allow the agent to have the velocity it was initially set at. If it is a collision velocity the algorithm, then checks whether the current velocity is in the set of collision velocities. If it is not, the new velocity stays as the current velocity. However, if the current velocity is a collision velocity the algorithm has to find the closest possible velocity to the current velocity that is not a collision velocity.

To find the closest velocity to the current velocity that is not a collision velocity the algorithm creates a nested for loop to calculate all the points that are less than or equal to the length of the radius of B away from $\widetilde{A}$. These points are the potential new velocities that agent A can achieve near the current velocity. These positions are compared to the values in the matrix of collision velocities to determine whether or not these points are collision velocities. If the points calculated are not a collision velocity, then they are put into a matrix of non-collision velocities.

The next stage is to determine which value in the non-collision velocities is closest to the current velocity. This is done in the same way as the 2D case by finding the distance between the non-collision velocities and the current velocity. Then the point that gives the shortest distance is chosen to be the point to which the new velocity vector goes to.

5.3 Other Possible Methods for the 3D case

There are a couple of additional original methods that could solve the Velocity Obstacle in three dimensions. These ideas have not yet been implemented or tested and are purely theoretical. The aim of looking for other possible solutions is to determine the best solution in terms of computational time as this is a large factor in implementing the collision avoidance algorithms in real time.

One solution is based on using rotation and translation matrices as the previous method describes, but instead of creating all the points in the discs and rotating and

translating the cone back to original position the velocity vector would be translated and rotated to the same orientation as the cone, i.e. just along the z axis. This is to avoid rotating and translating tens of thousands of points created in the cone which would take up computational time.

The other solution is similar to the two dimensional case where position of agent A and the two tangents points C and D create a triangle, then the barycentric coordinates of the point in question for being a collision point or not are tested to determine whether or not it is in the triangle. This method can be extended to the three dimensional case where the cone can be represented as a number of 2D triangles, all with the same centre point but the plane of each triangle pane is rotated about the centre line to create a cone. This method would require the position of agent A and B then the tangent points of each pane. This requires substantially less computational power than creating tens of thousands of points.

These methods have not been created yet due to time constraints in the project but they show how the algorithms are constantly being developed to produce better performances. These methods could be used for future work to develop this project further.

6. VELOCITY OBSTACLE TESTS

This chapter looks at the variety of tests used to check that the algorithm works correctly and as desired. There are tests on specific sections of the algorithm, such as testing to see if the Velocity Obstacle is created in the right quadrant, and also that the new velocity created is acceptable. There are also tests on the validation of the algorithm, i.e. how well the algorithm works for a variety of different scenarios.

6.1 Algorithm Tests for the 2D Case

6.1.1 Testing the Velocity Obstacle

The Velocity Obstacle was tested for each possible quadrant to see if it had translated by the velocity of B, if it had the right orientation with respect to agent A, and if it had been extended further to include the points beyond the location of agent B. Figure 6.1 shows multiple VOs each with a different starting position.

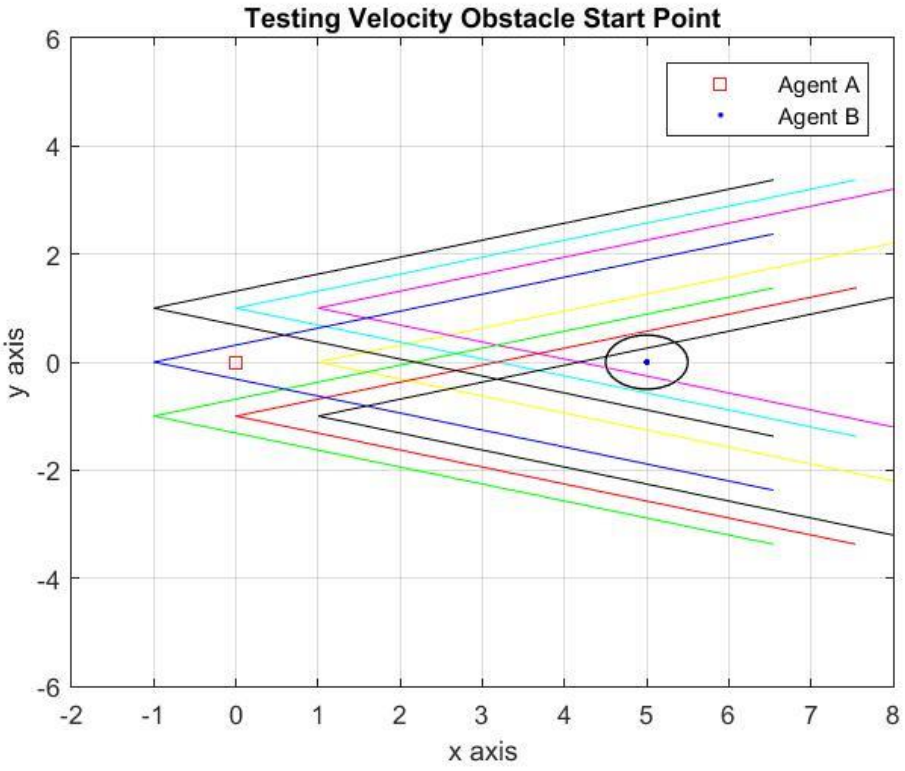


Figure 6.1 Testing the Velocity Obstacle Being Translated

This is to show different velocities of B and how the VO is translated by the velocity of B. The velocities used are each row of Figure 6.2.

```
Vel_B = [1, 0, 0;
         1, 1, 0;
         0, 1, 0;
         0,-1, 0;
        -1,-1, 0;
        -1, 0, 0;
        -1, 1, 0;
         1,-1, 0];
```

Figure 6.2 Velocities used to test the Velocity Obstacle being translated

Figure 6.3 shows the orientation of the Velocity Obstacle for varying positions of agent B. The positions of agent B have been chosen so that each quadrant has been tested. The positions of agent B are shown in Figure 6.4.

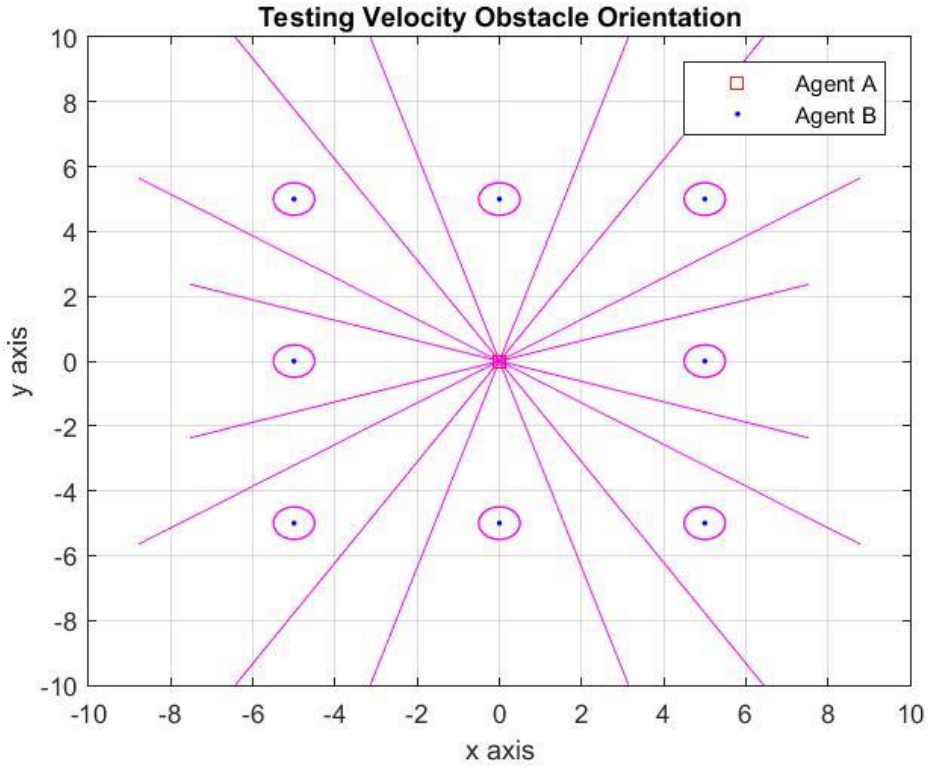


Figure 6.3 Testing the Velocity Obstacle Orientation

```
Pos_B = [5, 5, 0;
         5, 0, 0;
         0, 5, 0;
         5,-5, 0;
        -5, 5, 0;
        -5, 0, 0;
         0,-5, 0;
        -5,-5, 0];
```

Figure 6.4 Matrix of Position of Agent B

6.1.2 Testing the Barycentric Coordinates

The collection of points surrounding agent A were tested to determine whether they were located within the triangle created by points A, C, and D by using the barycentric coordinates of the test point. The first test was to see that the 10x10 grid of points around agent A was created.

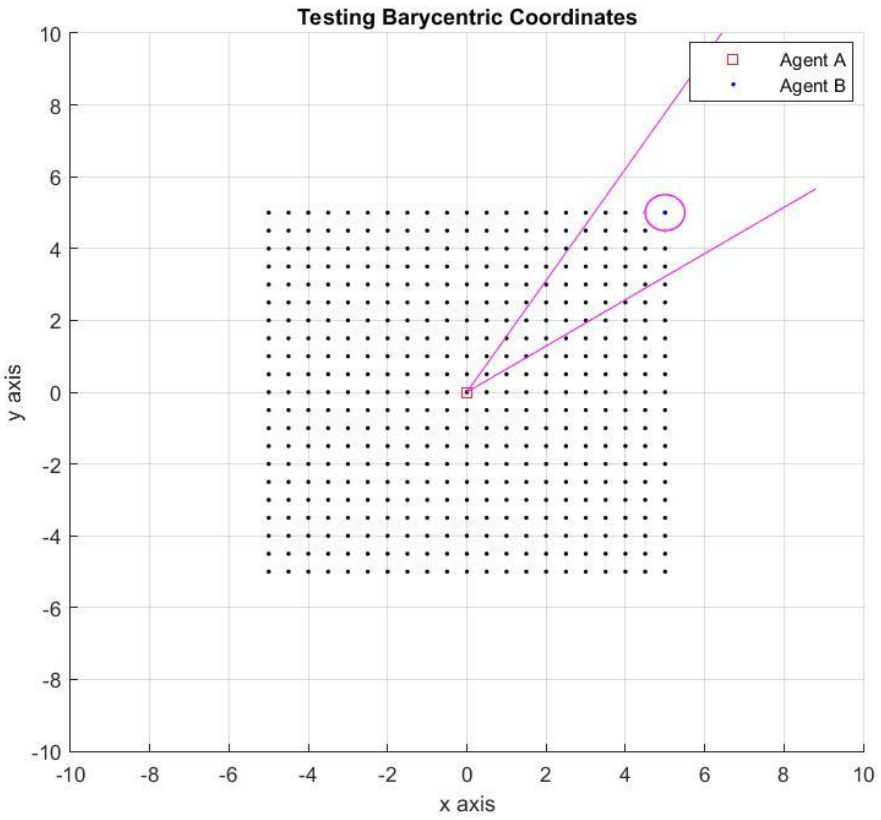


Figure 6.5 Testing all the points in a 10x10 grid

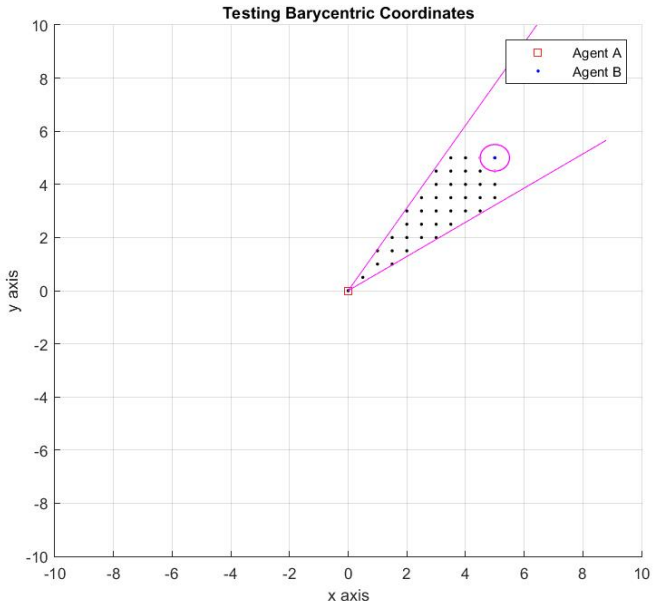


Figure 6.6a Plot of points inside triangle

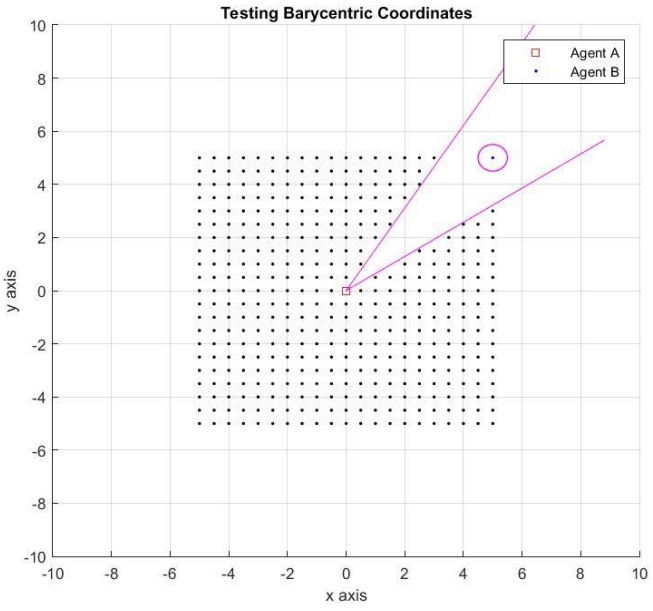


Figure 6.6b Plot of points outside triangle

Figure 6.5 confirms that all the points have been created. In this test the precision of the grid was set at 0.5. These points represent all the possible locations of the tip of a velocity vector.

The next test was to see if the barycentric coordinates of each of those points were inside the triangle. Figure 6.6a shows all the points which lay inside the triangle and Figure 6.6b shows all the points outside the triangle. If the Figure 6.6a and 6.6b were to be combined, the result would be Figure 6.5. This confirms that the algorithm to calculate whether or not a point lies within a triangle works.

6.1.3 Testing the Velocity Vector

The velocity vectors calculated from all the points from the previous test were tested to verify if they were a collision velocity or not. This was achieved by creating a vector plot of the collision velocities with a plot of the VO and verifying that the collision velocities lie within the VO. Similarly, the same was done with the non-collision velocities to verify that they did not lie within the VO.

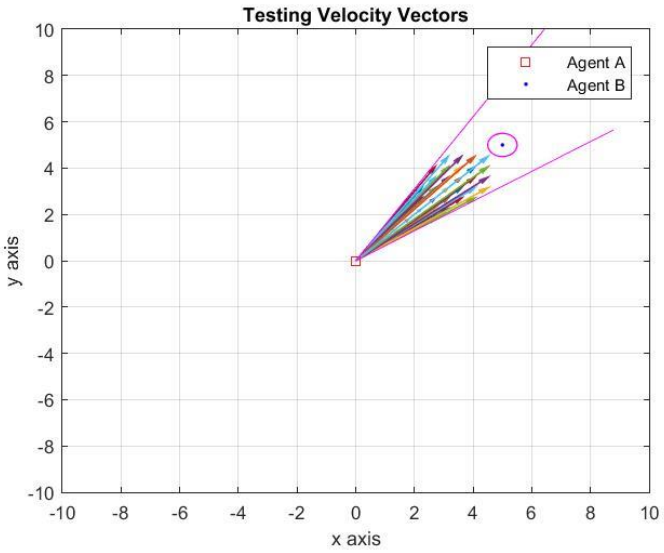


Figure 6.7a Plot of collision velocity vectors

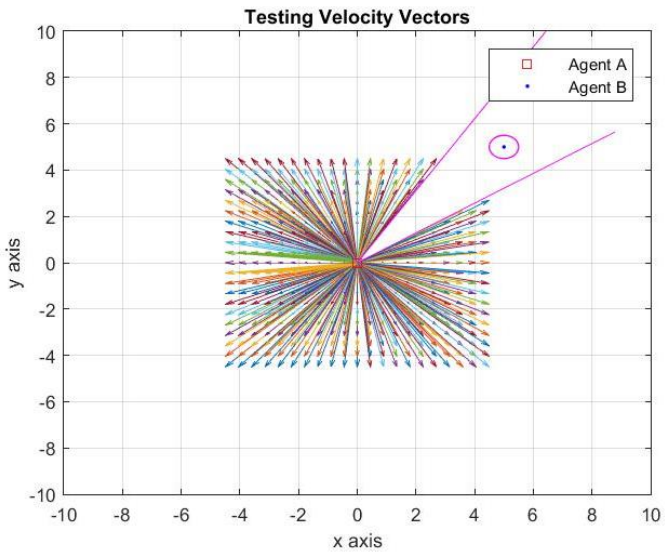


Figure 6.7b Plot of non-collision velocity vectors

Figures 6.7a and b are similar to Figures 6.6a and b, however these figures use the *quiver* plot function in MATLAB to represent the velocity vectors. There was one problem with the *quiver* function as it doesn't graphically line up to where the tip of the arrow should be. This is only an issue for graphical representation as the actual value of the velocity vector is correct in the algorithm.

Figure 6.8 shows how the velocity vectors are also correct for multiple agents.

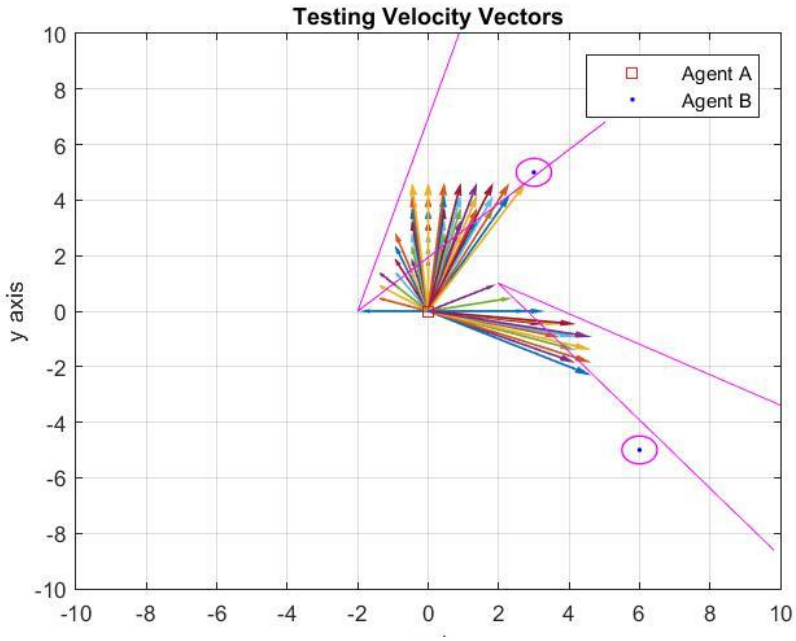


Figure 6.8 Velocity vectors using multiple Velocity Obstacles

6.1.4 Testing the New Velocity Vector

The new velocity vector chosen was tested to verify that it was the closest possible velocity vector to the original velocity vector, and that it was not going to cause a collision. Figure 6.9 shows how the original velocity lies inside the VO and then the new velocity is the nearest point that is not in the VO. This figure shows the *quiver* doesn't reach the point completely.

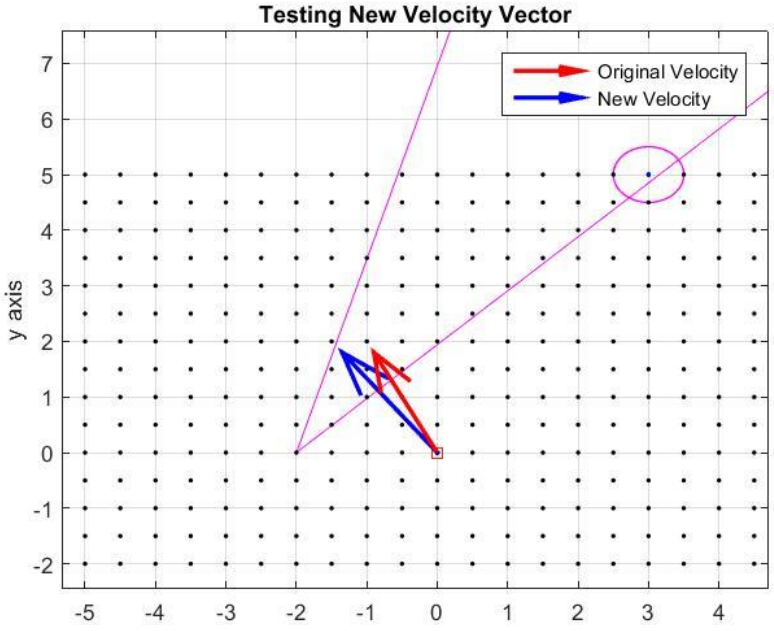


Figure 6.9 Choosing nearest velocity not inside the Velocity Obstacle

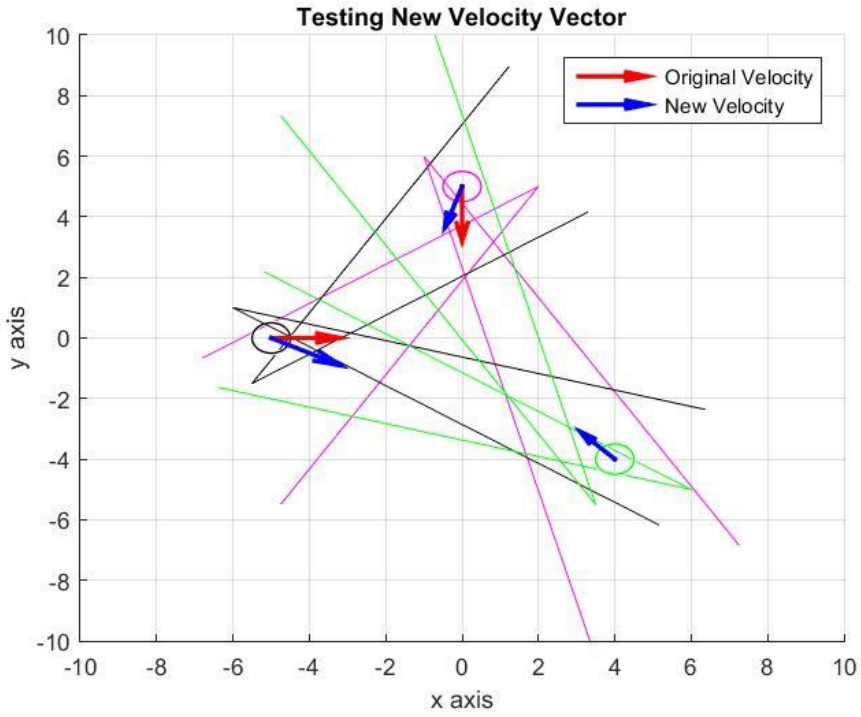


Figure 6.10 New Velocities for Multiple Agents

Figure 6.10 shows multiple agents each creating a VO with each other agent. The coloured circles represent each agent and the matching coloured cones are the VO for that agent.

6.2 Algorithm Tests for the 3D Case

6.2.1 Testing the Velocity Obstacle

The Velocity Obstacle was tested for each possible quadrant to see if it had been translated by the velocity of B, if it had the right orientation with respect to agent A, and if it had been extended further to include the points beyond the location of agent B.

Figure 6.11 shows just a selection of the velocities tested to verify that the VO shifts by the velocity of agent B. For these tests the position of agent A and agent B were kept the same, however the velocity of agent B was changed to cover all quadrants. Figure 6.12 shows the VO when agent B is located in all the possible quadrants. Figure 6.12 is split into 4 figures which show the VOs from the different planes xy, xz, yz, and the xyz plane. It also shows how the VO extends further than agent B as in the two dimensional case.

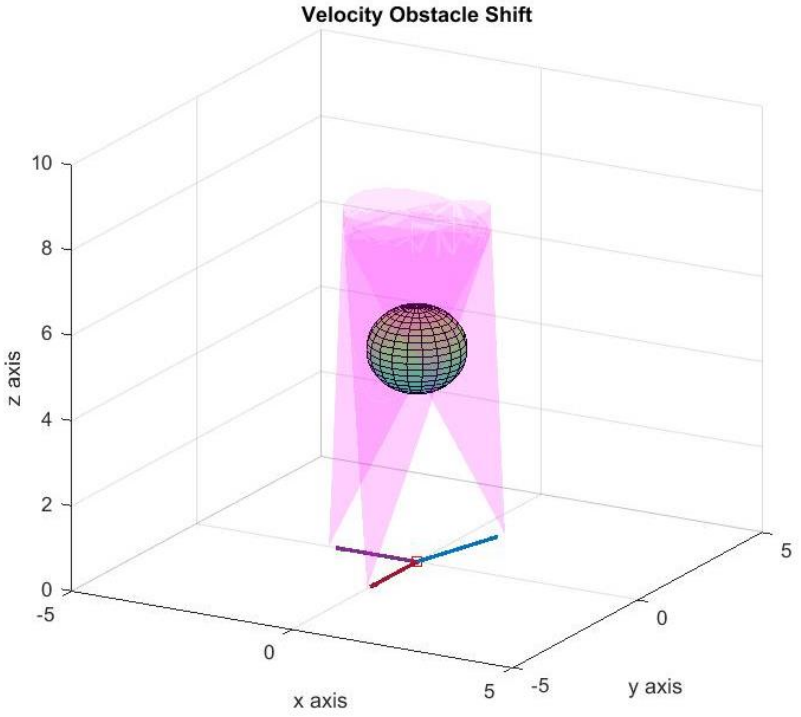


Figure 6.11 Showing the Velocity Obstacle being shifted by the Velocity of B

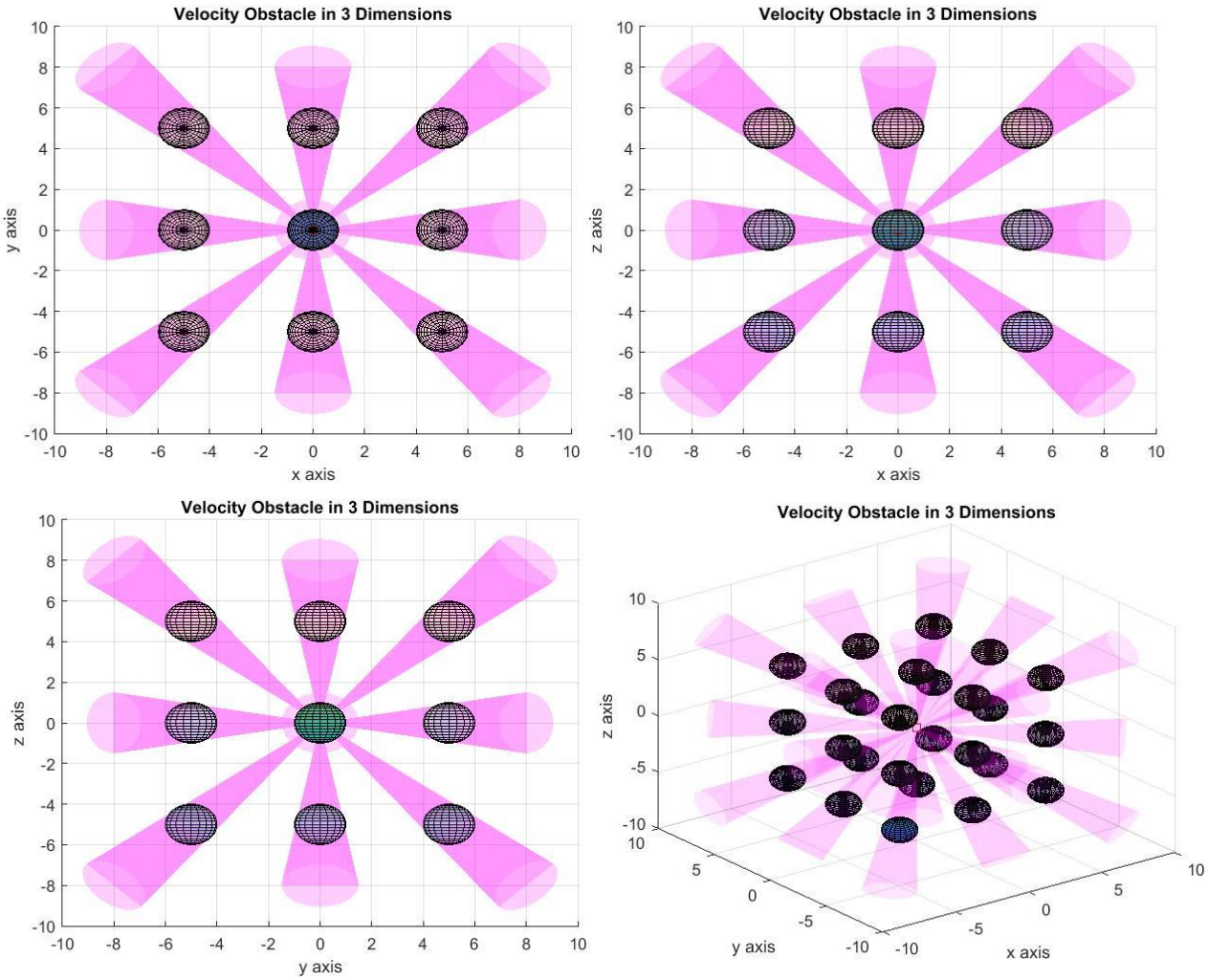


Figure 6.12 Different views of Velocity Obstacles in all quadrants

6.2.2 Testing the Translation and Rotation Matrices

The translation and rotation matrices were tested to verify that the line between agent A and agent B can be translated to the origin and then rotated so that the line AB lies along the z-axis. Figure 6.13 shows the blue line as the original line between agent A and agent B, the green line is the shifted line to the origin, and the red line is the final translated and rotated line.

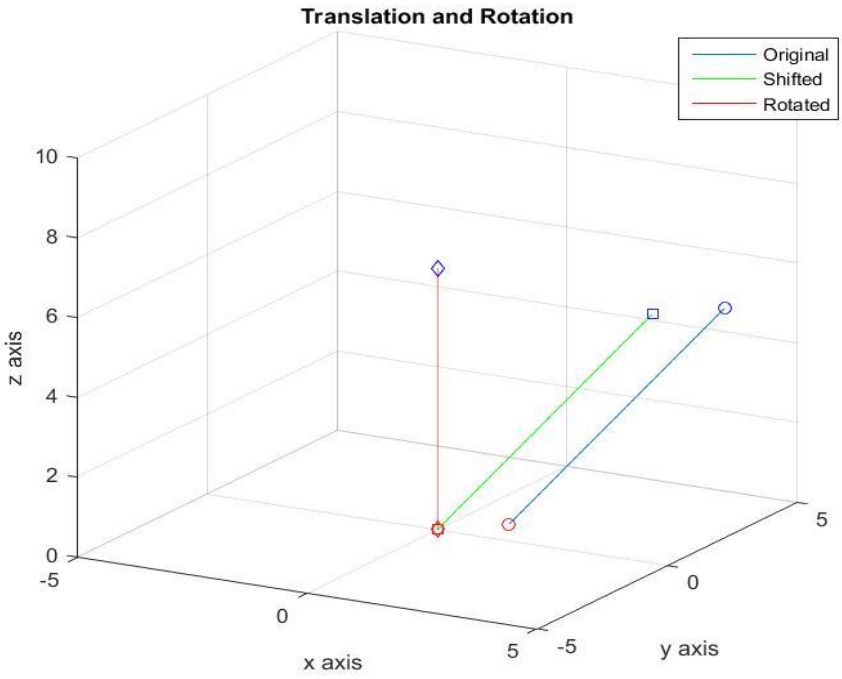


Figure 6.13 Line between the two Agents translated and rotated

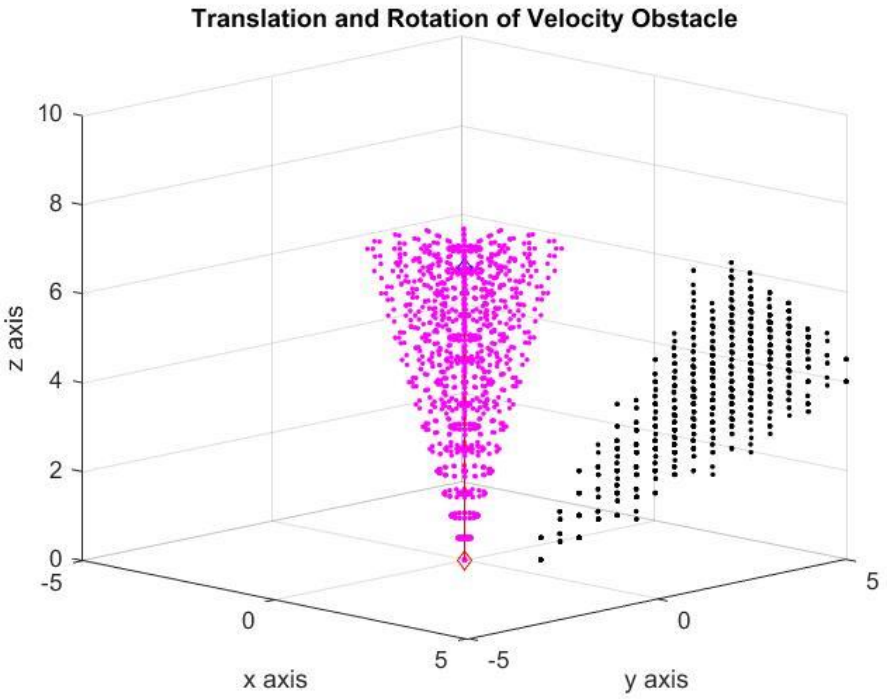


Figure 6.14 Collision points translated and rotated

Different positions of agent A and B were used to test that the translation and rotation matrices can be used to move any line between A and B to the origin and along the z axis. The collision points were then plotted, first just in the z axis, then translated and rotated to test that the translation and rotation matrices work for all the collision points, shown in Figure 6.14.

6.2.3 Testing the Points Inside the Discs

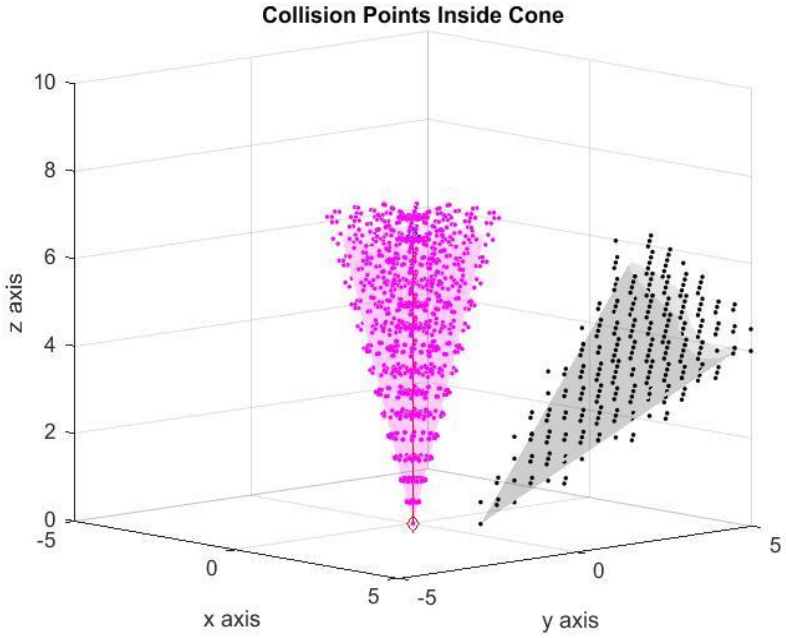


Figure 6.15 Collision Points inside Velocity Obstacle

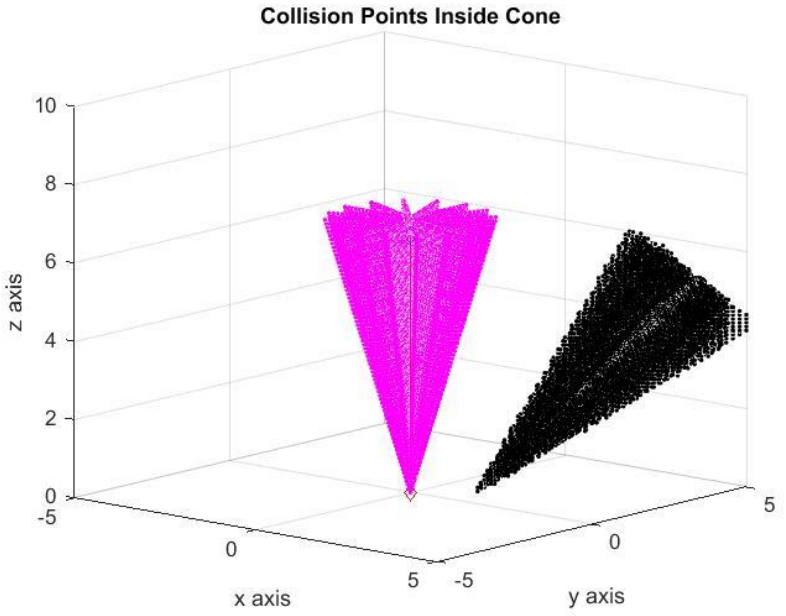


Figure 6.16 Precision of Collision Points Increased

This test verifies that the collision points are in fact inside the collision cone by overlapping the Cone plotting function with all the collision points. Figure 6.15 shows how the points lie perfectly inside the cone when it lies just in the z axis but when translated and rotated the points are not. This is due to the rounding of the location of the points to the nearest 0.1 as this is the precision set by the algorithm. In this test the precision of the number of points was reduced to be able to see the cone plot, otherwise the results would be as shown in Figure 6.16.

6.2.4 Testing the New Velocity

The new velocity vector chosen was tested to verify that it was the closest possible velocity vector to the original velocity vector and that it was not going to cause a collision. Figure 6.17 shows the red arrow as the original velocity vector, which lies inside the cone and the blue arrow as the new velocity vector which is outside the cone. This verifies that the algorithm chooses a correct new velocity.

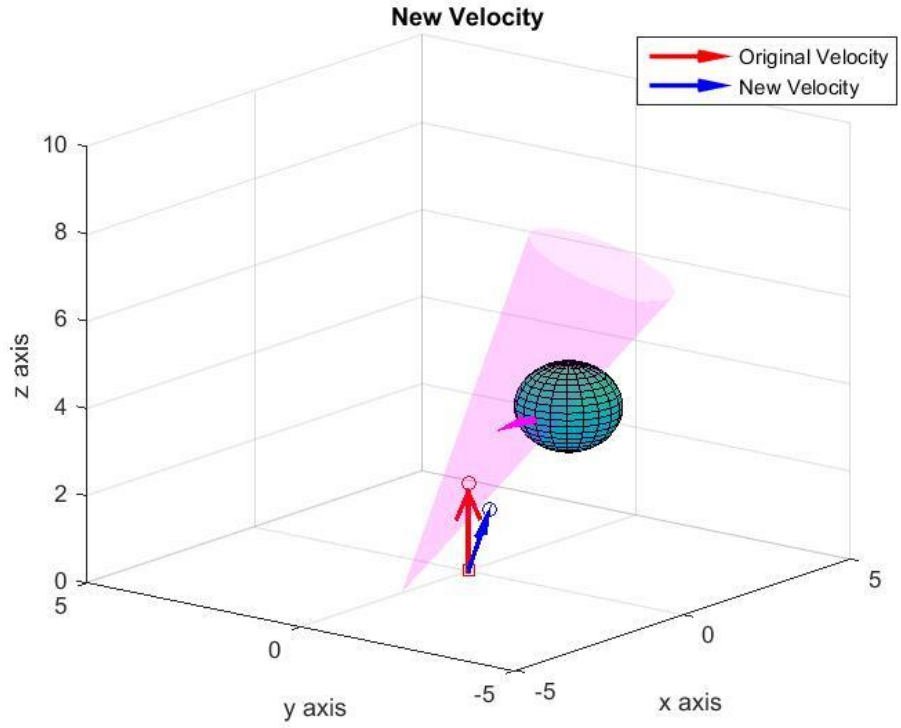


Figure 6.17 New Velocity in 3D Space

The next test was to see how the algorithm behaves when multiple agents are placed into the simulation. Figure 6.18 shows how the algorithm choses a new velocity for each VO with each agent. If the first new velocity is within the second VO, then the algorithm choses a new velocity that is not within either VO. The algorithm will do this for ‘n’ agents, updating the new velocity each iteration. As Figure 6.18 shows the original velocity vector as the red arrow, then after the first iteration the

algorithm produces the green velocity vector, and finally the algorithm produces the blue velocity vector which is outside both of the VOs.

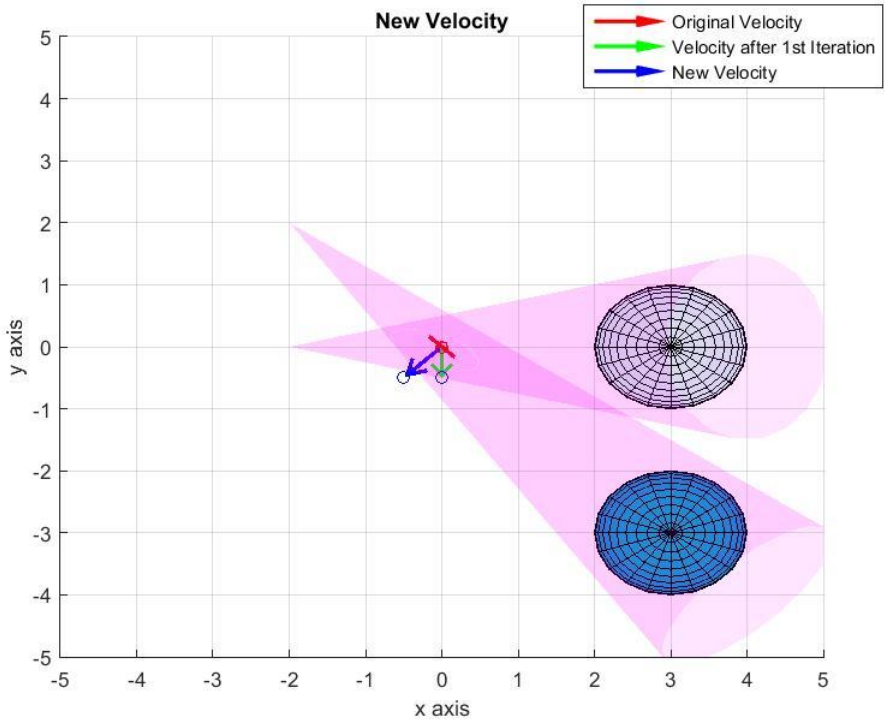


Figure 6.18 Updating the Velocity Each Iteration

6.3 Algorithms Performance Validation and Evaluation

There are several different methods to validate and evaluate the performance of the algorithms created. The algorithms can be validated by running a simulation multiple times to produce a mean completion rate, i.e. the probability of a successful simulation without any collisions. The algorithms can also be evaluated by recording the computational time of the algorithm to calculate a new velocity. The performance of the algorithm can then be analysed to verify if it meets a set of requirements that have been desired by the user. This section only contains tests for the two dimensional case as the three dimensional case had too many bugs to provide a reliable simulation at the time of writing.

6.3.1 Probability of a Successful Simulation

To find the probability of a successful simulation the algorithm is run multiple times with different inputs for the positions and velocities of agent A and B. Each time the algorithm is run for one set of position and velocity values it is also run

several times according to the *runTime* variable. Inside the run time for loop the algorithm is called, then the position of agent A and B are updated by the velocity vector of the agent. This is so that the algorithm simulates movement of the two agents over the run time. The position of the agents is set by choosing random values between ± 10m, and the velocity of the agents is set by choosing random values between ± 5m/s. These values were chosen as the 5m/s was decided to be the max velocity the algorithm could achieve in calculating a new velocity, and the 10m was decided as the simulation area because of the previous tests all being within this area.

To verify that the algorithm is successful the validation algorithm logs the position and velocity of each agent at each iteration of the run time. This produces two matrices, one for agent A and one for agent B, with a size of number of scenarios by six times the run time, for example, if there are 1000 scenarios each with a run time of 20 iterations, this will produce a matrix of size (1000 x 120). The reason the number of columns is six times the run is because for each run time the x, y, z positional coordinates and the u, v, w velocity components are placed into a column. The validation algorithm also logs if there is a collision. A collision happens if the difference in position of agent A and B is less than or equal to the size of the agent. It logs the collision with an index value which will give the index of which scenario (row) and which run time iteration (column). Therefore, the position and velocities of the agents will be known for that collision. To calculate the probability of a successful simulation the equation 6.1 is used.

$$P(success) = 1 - \frac{Number\ of\ Collisions}{Number\ of\ Scenarios} \tag{6.1}$$

Where P(success) is the probability of success. With increased number of test scenarios there will be an increase in the accuracy of the probability.

While running the simulation for the two dimensional case for 10,000 random positions over a run time of 20 iterations, the algorithm computes whether or not a collision occurs between the two agents. This is determined by calculating the absolute magnitude between the two agents and if this distance is less than or equal to the diameter of the A then a collision would occur. From the 10,000 tests there were only seven occasions when a collision occurred. These occurred when the random positions of the two agents were created. It happened to be that the spawn locations had an absolute magnitude between each other of one, therefore resulting

in a graze. However, the agents moved away from each other after they had been spawned in. These results give a 99.93% success rate for a simulation for random initial conditions.

The next test is to manually set the agents on a collision course and calculate the percentage of successful avoidance. This was achieved by positioning the agents in a circle and giving each agent a velocity which would take it to the antipodal position, shown in Figure 6.19.

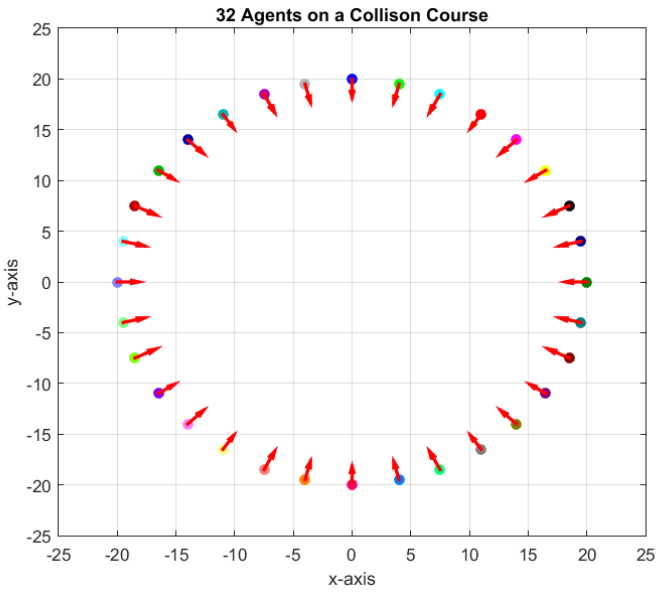


Figure 6.19 Circle of Agents with Initial Velocities to their Antipodal positions

This test was run for varying amounts of agents and the probability of a successful avoidance for each simulation is displayed in Figure 6.20. To calculate the

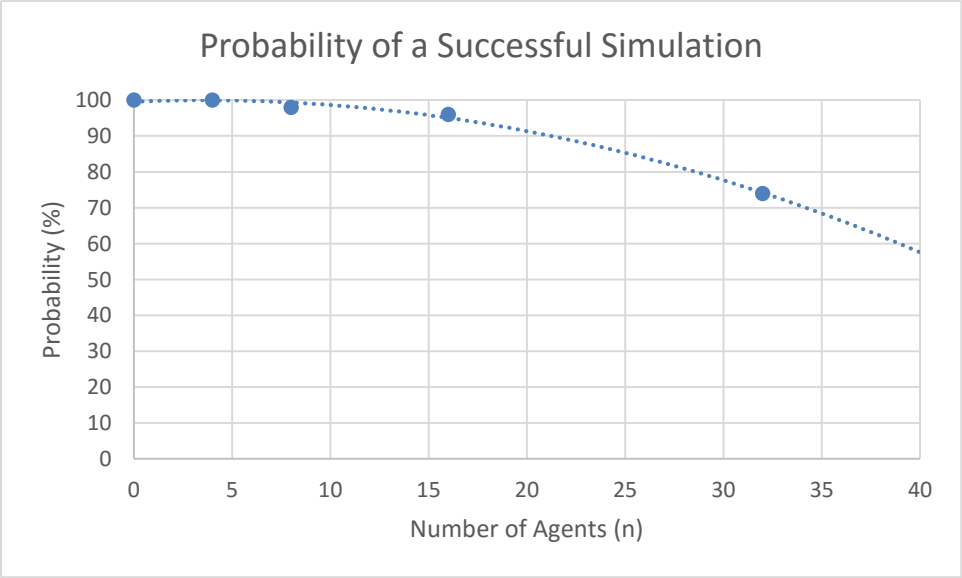


Figure 6.20 Probability of a Successful Simulation with Varying Amounts of Agents

probability of a successful simulation each test had a run time equal to 20 iterations and the tests were repeated 10 times all starting with the same radius circle. This gives a possible 200 iterations where a collision could occur. To determine if a collision happened or not the positions of the agents are compared to each other and if the absolute magnitude between the positions is less than or equal to the size of the agents then a collision occurs.

Figure 6.20 shows that as the number of agents in one given area increase the probability of a successful simulation decreases. This is due to the fact that increasing the swarm density decreases the performance and increases the complexity of the algorithm (Kernbach, 2013) resulting in errors in the algorithms. These errors could be that because of the high swarm density there isn't any feasible non collision velocities that the agent could achieve, therefore resulting in a collision.

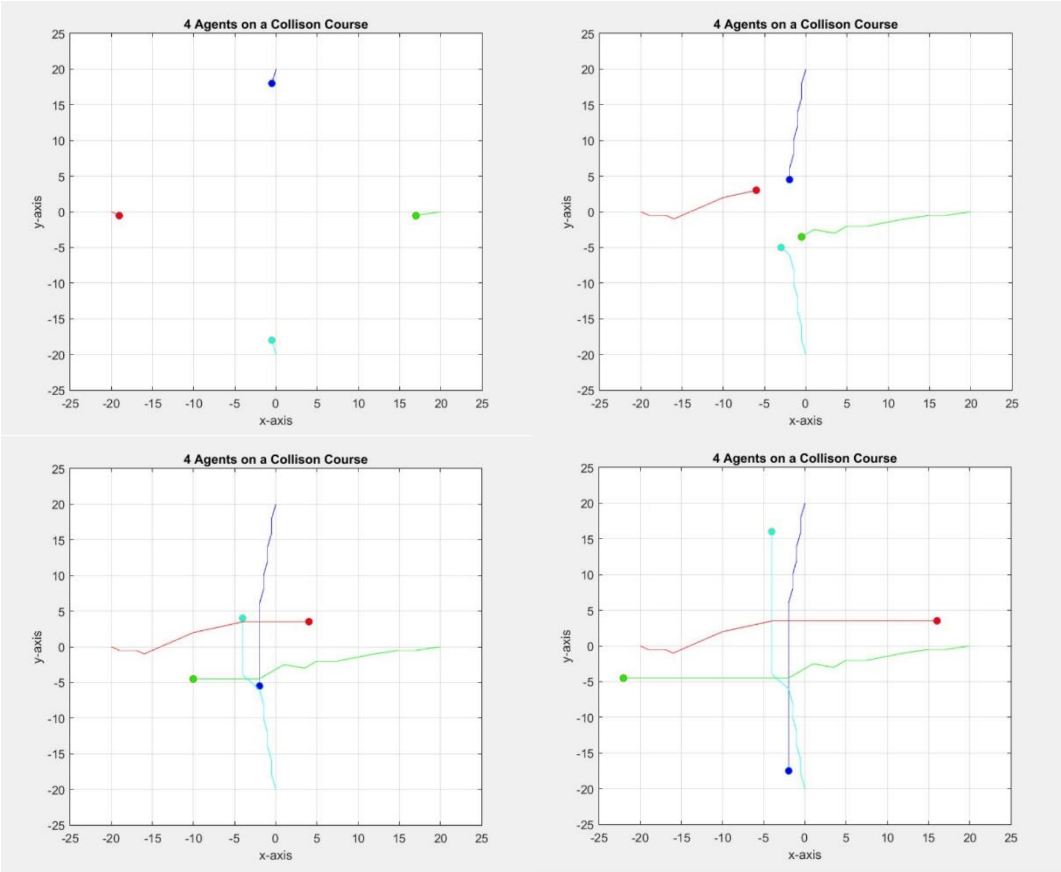


Figure 6.21 Agents Collision Avoidance Path

Figure 6.21 shows the path the four agents took to avoid each other and how they return to the original velocity when clear from a collision. This simulation was run using a time step of 1s, meaning that for every iteration of the simulation the agent's position was updated with the magnitude of the velocity. For example, if the

velocity was 1m/s, at the end of the iteration the agent would move by 1m. This results in large step changes in the position of each iteration which is not realistic. To produce a smoother result, the time step was changed to 0.1s, i.e. each iteration the agent would move a factor of 0.1 times the magnitude of the velocity. To get

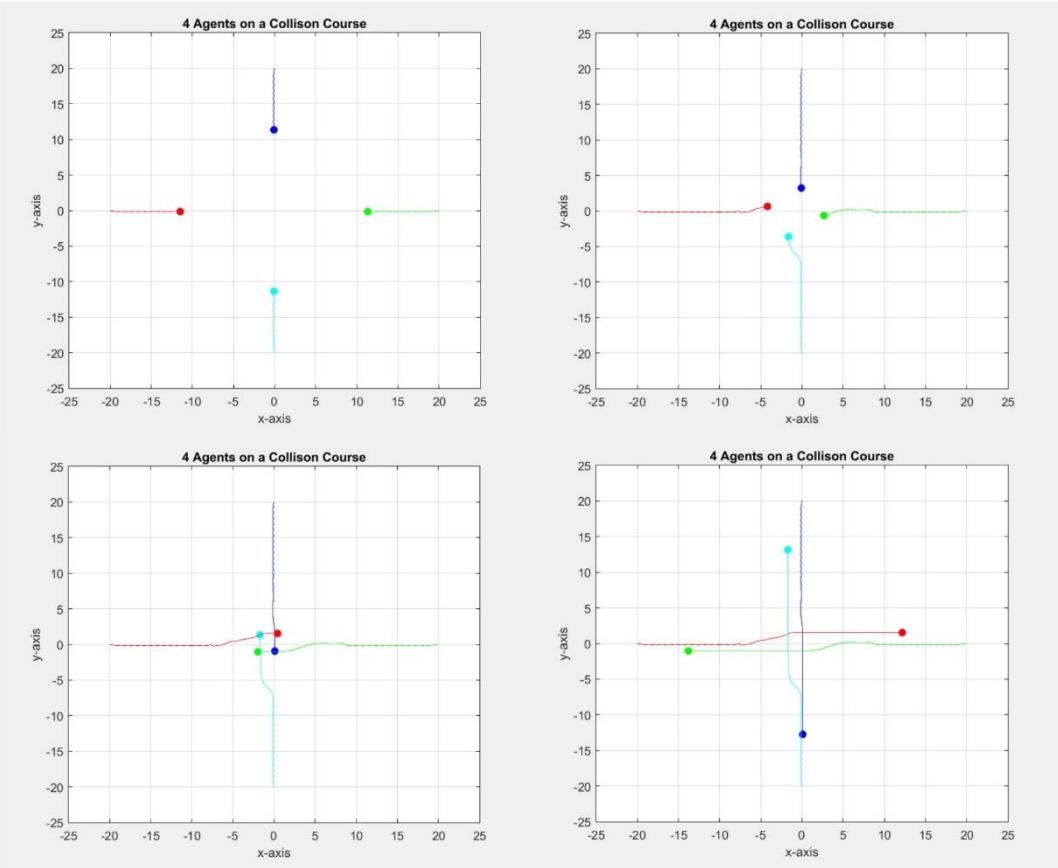


Figure 6.21 Collision Avoidance with a Time Step of 0.1s

the agent to move the same distance the run time would have to be increased by a factor of 10. Running the same scenario of four agents as in Figure 6.20 but with the change in time step and run time the result is as shown in Figure 6.21.

The results produce a much smoother path as with each iteration the agent moves a small fraction towards the desired position and then revaluates a velocity that will not cause a collision. With a decrease in the step size there is an increase in the number of decisions made, however these decisions provide the best path for the agent to follow to avoid a collision.

6.3.2 Computational Time

A MATLAB script is created to evaluate the avoidance algorithm's computational time to calculate a new velocity for the algorithm. This script uses the MATLAB *profile* function which tracks the execution time of a program. The test carried out runs the avoidance algorithm for 2, 4, 8, 16, and 32 agents while recording the time it takes to complete one iteration of the simulation. The positions and velocities are set up the same as in section 6.3.1 and on a collision course.

For each set of agents, the test was run ten times to find the average time it took to complete one iteration. Figure 6.22 shows the result of these tests and how the computational time rises exponentially with an increase of agents. This is because at each iteration the avoidance algorithm is run for each combination of possible inputs. For example, let there be three agents, A, B, and C, the avoidance algorithm will create a VO between pairs: AB, AC, BA, BC, CA, CB. This means for a three agent simulation the avoidance algorithm will be called six times. The number of times the algorithm is called depending on the number of agents can be summarised by equation 6.2, where n is the number of agents.

$$n \times (n-1) \tag{6.2}$$

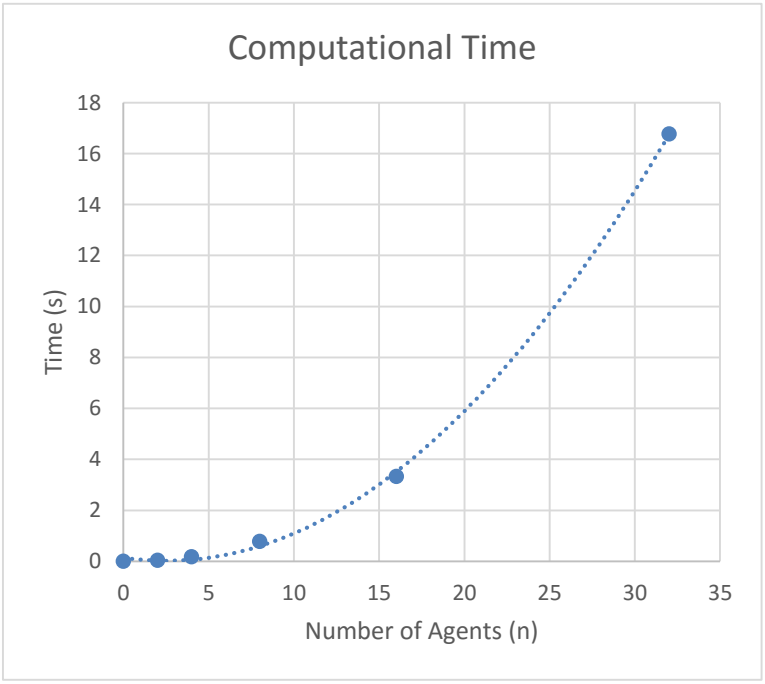


Figure 6.22 Computational Time of Simulation with Varying Number of Agents

7. CONTROLLING THE ARDRONE

This Chapter looks into how the Parrot ARDrone 2.0 can be controlled for possible use as an agent. One such outcome is to be able to implement the collision avoidance algorithms from the previous chapters onto the ARDrone. The first that needs to be achieved is to understand how the ARDrone can be controlled autonomously.

The ARDrone comes with a Software Development Kit (SDK) which enables the user to be able to design and develop code that can control the ARDrone. The ARDrone can be controlled via a device supporting Wi-Fi, e.g. a computer/laptop or a Raspberry PI with a Wi-Fi dongle (Parrot, 2012). In this case a Raspberry PI was chosen to control the ARDrone. This was because a Raspberry PI can be easily attached to the ARDrone and it can also be powered off the USB port attached to the ARDrone, shown in Figure 7.1. It also means that the Wi-Fi connection will not be lost as the two would be next to each other at all times. The location of the Raspberry PI can be placed on top or underneath the ARDrone as shown in Figure 7.1. However, the user needs to be aware that the Pi needs to be as close to the centre of mass of the drone and that wires do not obstruct the propellers.

Figure 7.1 Raspberry PI Connected to ARDrone

7.1 Raspberry PI

Using a Raspberry PI with a Wi-Fi dongle allows the user to create a program using a variety of different languages, and be able to have that program launch on start up. This method is ideal as everything can be connected and all the user has to do is plug the power into the ARDrone and wait for the Raspberry PI to boot up, then the ARDrone would fly according to the code in the Raspberry PI.

One way to control the ARDrone is via a Node.js client. The Node.js client enables the user to create a JavaScript programme that can communicate with the ARDrone. Node.js can be easily installed onto a Raspberry PI and there are a variety of different tutorials available on how to download Node.js (adafruit, 2015) (elinux, 2015) (play, 2015).

Now that Node.js is installed on the Raspberry PI, the next step is to write a programme that uses Node.js and the SDK to control the ARDrone. (Copter, 2012) produced a useful guide on how to create basic scripts that can control the ARDrone. For this guide the directory node-ar-drone needs to be downloaded from Github (felixge, 2014). In this directory they have created several functions which can control the ARDrone. For example,

- client.takeoff() – Sets the internal *fly* state to *true* and the drone takes off.
- client.land() – Sets the internal *fly* state to *false* and the drone lands.
- client.clockwise(speed) – Makes the drone spin clockwise, speed can be a value from 0 to 1.
- client.up(speed) – Makes the drone gain altitude, speed can be a value from 0 to 1.
- And more…

These functions can be combined to create an autonomous program. In this case an autonomous program is defined as creating a list of commands for the drone to execute, so when the drone is in flight the user has no control over the behaviour of the drone. A simple example given in the README.md file of the node-ar-drone directory for autonomous flight is shown in Figure 7.2. This code tells the ARDrone to take off and hover, then after 5000ms to turn clockwise, and finally

after 3000ms to stop and land. An autonomous programme can be created which can control the ARDrone to fly anywhere.

```
var arDrone = require('ar-drone');
var client  = arDrone.createClient();

client.takeoff();

client
  .after(5000, function() {
    this.clockwise(0.5);
  })
  .after(3000, function() {
    this.stop();
    this.land();
  });
```

Figure 7.2 Example of code for autonomous flight

A programme was created to make the ARDrone fly a square course inside a room. This was done to test if it was possible to use the code from (felixge, 2014) to control the ARDrone. If this was the case, then there becomes the possibility to be able to combine the collision avoidance algorithms with being able to control the ARDrone.

The pseudo code for making the ARDrone fly in a square pattern is shown in Figure 7.3. Comparing the example code and the pseudo code it can be seen that the example code uses time to complete actions rather than distances or angles as the pseudo code uses.

```
TakeOff
Fly Forward 2m
Turn Clockwise 90degs
Fly Forward 2m
Turn Clockwise 90degs
Fly Forward 2m
Turn Clockwise 90degs
Fly Forward 2m
Land
```

Figure 7.3 Pseudo Code for Square Flight Pattern

Therefore, some tests were required to calculate the time it takes to move forward 2m and turn 90 degrees. Then this time can be scaled to give the required distance and angles for the desired code.

Figure 7.4 90 degrees Turn Time Test

The test carried out to calculate the time to turn 90 degrees was to set out two orange cones at 90 degrees as markers for the start position and the target position. The first time chosen was 1000ms, this resulted in the top most yellow cone in Figure 7.4. As this cone is half way between the two orange cones then next value chosen was 2000ms, this resulted in the bottom most yellow cone. The next value chosen was 1500ms and this resulted in the second top most yellow cone. The final value chosen was 1700ms and this resulted in the desired 90-degree rotation. The next test was to calculate the time to move 2m forwards. Figure 7.5 shows the set up in which the test took place. An orange cone was set up at 1m intervals starting from the right hand side. The first test used a value of 1000ms, this resulted in the right most yellow cone. The next test used 1500ms, this resulted in the second right most yellow cone. The third test used 2000ms, this resulted in the left most yellow cone. The final test used a time of 1800ms, this resulted in the second to the left yellow cone located at 2m.

Figure 7.5 2m Forward Time Test

8. USING THE RPLIDAR

In this chapter the use of a LiDAR is described as a sensor for collision avoidance. The use of a LiDAR is in a separate Collision Avoidance System to that of the algorithms discussed in previous chapters. This Collision Avoidance System is focused at avoiding collisions with obstacles such as walls. This is because of the LiDAR chosen for the project, the RPLIDAR, only scans in a two dimensional plane. This LiDAR was chosen because of the ability to automatically scan 360 degrees, lightweight, and also the cheapest on the market with those abilities. The RPLIDAR also comes with a dedicated sdk enabling the user to extract data (Slamtec, 2015). This data is outputted in the form of distance in millimetres and also angles in degrees. From this data it is possible to calculate where an obstacle is relative to LiDAR.

The LiDAR can be used in conjunction with a drone to detect if a wall or object is in the same plane as itself. From this idea the LiDAR could be used to calculate if an object is less than a certain distance away, which could be called a safe distance, if there is an object within this safe distance a program could control the drone it is attached to fly away from the obstacle. An example of some pseudo code is found in Figure 8.1.

```
While distance to object < safe distance
    new heading = heading of obstacle + 180 degrees
    fly on new heading
//As it's a quadcopter, it can easily translate
//rather than turning and going forward


if multiple objects at different locations
    either fly up or down to avoid
    or find heading where there is not an obstacle
       and fly on that heading
```

Figure 8.1 Pseudo Code for Collision Avoidance using a LiDAR

8.1 RPLIDAR Tests

Several tests were conducted on the RPLIDAR to examine how it behaves. The frame_grabber programme was downloaded from the RPLIDAR documentation (Slamtec, 2015). This programme connects with the RPLIDAR via one of the USBs on the host computer and creates a plot of the laser scans and shows any objects the laser reflects off, this is shown in Figure 8.2.

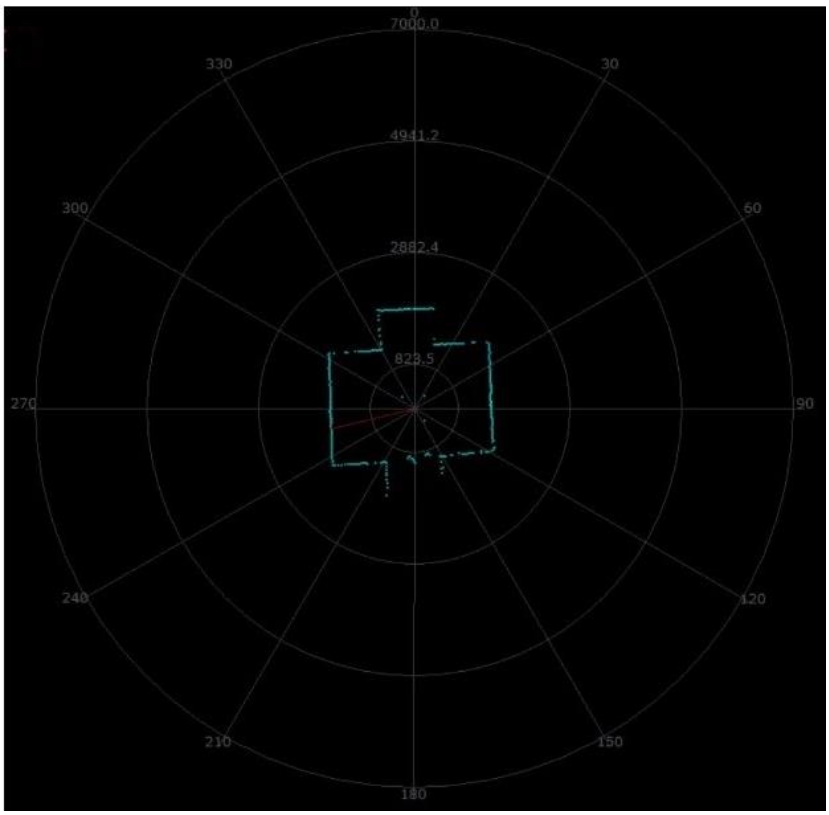


Figure 8.2 frame_grabber programme output

The scenario shown in Figure 8.2 is a mock-up of a room by using tables to create a boundary and in the centre is a chair with the host laptop on, which is running the software, and under the chair on the floor is the RPLIDAR. This scenario is shown in Figure 8.3, which shows how the tables and chair were set up in real life. From Figures 8.2 and 8.3 it can be shown how accurate the RPLIDAR is. Looking closely at Figure 8.2, the 4 chair legs can be seen in the middle and also the feet of the user while they used the host computer.

Another test carried out with the RPLIDAR was that of extending the tables to see the effect of the LiDAR of objects at varying distances. Figure 8.4 shows the table at the top pushed back to approximately 4m away and the tables on the right pushed into a semi-circular arc. It is easily shown that the further away an object is the

less precise the readings become. For the table pushed back 4m away the readings don't produce a straight line as they do for when an object is close by, like the tables on the left hand side. This lack in precision for objects far away is acceptable as it is only required to give accurate responses when an object is close by, for example, when the drone is close to a wall.

Figure 8.3 Creating a room using tables

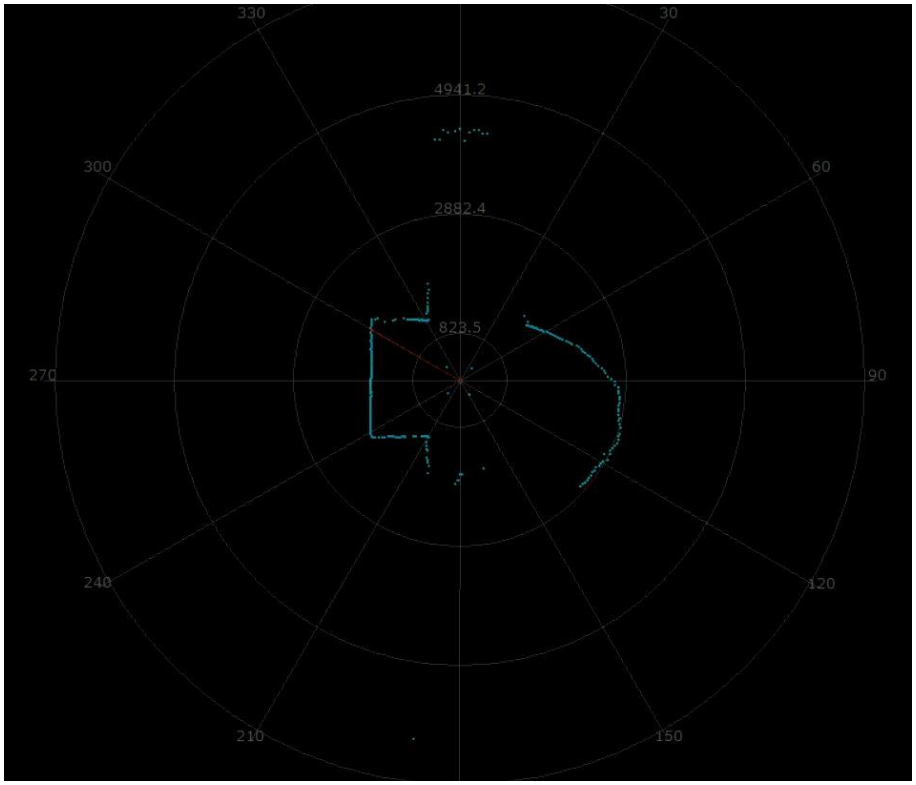


Figure 8.4 Extending the tables different distances

9. PROJECT CONCLUSION

9.1 Report Conclusion and Key Findings

In this report a detailed analysis of current Collision Avoidance Systems has been carried out with the intention of creating one for UAVs in a swarm for the use in a search and rescue environment. The Velocity Obstacle method was chosen as a basis for this Collision Avoidance System. The motivation for this was that it has not been widely used in conjunction with UAVs and this report looks into the possibilities of it being used for this scenario.

The first stage for creating a CAS using the Velocity Obstacle method was to first understand how they work. It was found that the concept of VOs was relatively simple in nature. In summary, if a velocity vector laid inside a collision cone created then a collision would occur between the two agents if no changes occurred. The challenges faced were that of determining if a velocity vector laid inside the collision cone and if it was to determine a new velocity vector for the agent that does not lie inside the collision cone.

This report first created a CAS using VOs in a two dimensional environment and then in a three dimensional environment. The three dimensional case was an experimental VO as it has not been studied as in depth as the two dimensional case has in the literature. Each case had extensive tests during the creation phase, e.g. testing the VO was created in the right place and orientation, testing the collision velocities are inside the VO, and testing the new velocity vector was not inside the VO. The 2D VO was also tested with a variety of different test cases to calculate the probability of collision with varying initial conditions and a review on the computational time was made.

Additional work to the collision avoidance algorithms was completed using the Parrot ARDrone and the RPLIDAR. A simple program was created to be able to send commands wireless to the ARDrone enabling it to take off spin around and to go forward. These commands are the base commands needed to program autonomous flight. The LiDAR was used to test the ability to detect walls and objects and to determine the feasibility of extracting data from the LiDAR.

9.2 Future Work

This report has created two CAS from concept to reality and due to the time limits of the project only a basic CAS could be created. This leaves a lot of scope for future work on this subject. The first port of call would be to optimise the performance of the algorithms by decreasing the computational time. This needs to be achieved mainly in the 3D CAS as the algorithm has not yet found the most efficient way of creating a VO but the foundations for this work to be enhanced have been laid.

Improvements to the algorithms can be made by including additional constraints such as giving agents priorities so that agents on an imminent collision course take a higher priority than that of agents who aren't on an imminent collision course. This could be achieved by adding a time horizon to the VO as described in (Fiorini & Shiller, 1998).

Another possibility is to include acceleration constraints to each Agent as described in (Berg, et al., 2011). In the current algorithms it is assumed that after each iteration the agent can move to exactly to the position desired, however a real agent will have constraints to the acceleration and therefore after each iteration the agent may not be in the position desired. By including these constraints into the algorithm the desired position can be more realistic to what the agent can achieve.

Similarly to the acceleration constraints the algorithms could incorporate the dynamics of the agent to be able to better model the movements and the available velocity vectors the agent can achieve. The agents could be modelled with having the constraint of a turning circle therefore the algorithm would need to be modified. This turning radius constraint is described in (Sezer & Gokasan, 2012).

To implement the avoidance algorithm onto a UAV an estimation of the positions and velocities of the agents needs to be made as the data packet being inputted into the algorithm would be raw data which needs to be filtered and estimated to get an accurate measurement of the packet information for the algorithm.

9.3 Project Management

The original objectives for this project were as follows:

Objective One: To evaluate current methods to establish the foundations of creating a new Collision Avoidance System.

Objective Two: To create a CAS in two dimensions.

Objective Three: To extend the two dimensional case to three dimensions.

Objective Four: To evaluate the performance of the two CAS.

Objective Five: To investigate the possibilities of implementing a CAS in real time on a UAV.

9.3.1 Meeting Objective One

Objective one was completed early on in the project and was detailed in the literature review. This formed the base knowledge for the rest of the project. From evaluating the current CAS, it was decided to use the Velocity Obstacle method.

9.3.2 Meeting Objective Two

Before creating a CAS using the Velocity Obstacle method it was important to understand how this could be achieved and in order to create the algorithm from concept. The 2D case was created before the 3D case as it has been previously researched so there was information to use to form the algorithm. The way the algorithm calculates if a velocity vector is within the cone and to calculate a new velocity are both novel ideas for the use of VOs.

9.3.3 Meeting Objective Three

Objective three of extending the CAS into three dimensions was a challenging step as it hasn't been researched as well as the two dimensional case, therefore using the same concept of if there is velocity vector is inside the VO, then there will be a collision. The method of determining if there is a collision velocity or not and to compute the new velocity is all an original idea. As it was an original idea and there wasn't enough time to optimise the method additional work is needed.

9.3.4 Meeting Objective Four

Evaluating the performance of the CAS created proved to be time consuming as errors would occur in the algorithm. This resulted in issues needing to be fixed which occurred several times as the algorithms were tested to their limits. However, it is desired to find all the bugs and errors while testing and evaluating the algorithms to make sure there is no case where they wouldn't work. In some cases, it wasn't obvious to why the algorithm would not work correctly. Nevertheless, the algorithm works the majority of the time for a low number of agents.

9.3.5 Meeting Objective Five

Objective four was never fully completed due to the majority of the time spent creating and testing the avoidance algorithms. However, some progress was made into understanding how it is possible to control the ARDrone wirelessly and autonomously. Research and experimentation of the RLIDAR were carried out to understand how the LiDAR can be used as a sensor for collision avoidance.

9.3.6 Time Management

The work over the two semester differed as in the first semester there was substantial research into CAS. However, there are a lot of different types of CAS and it took a while to find an achievable CAS to create in the time. Originally work started on creating a model of a LiDAR but this was quickly abandoned as it was no longer relevant to the CAS chosen. Work was also completed on creating a MATLAB script which converted GPS latitude and longitudes to polar and Cartesian coordinates to be used for a CAS but this was not followed up as it was not necessary for the final CAS. The work completed for semester one can be seen in the Gantt chart in Appendix C. Also in Appendix C the forecasted work can be seen.

Over the Christmas break it was decided to change the direction of the project to focus on Velocity Obstacles and using them to create a CAS. A new Gantt chart was created for the proposed work for the second semester, this can be seen in Appendix D. The work carried out over the second semester was well planned and all the work was completed before the deadlines. However, as mentioned in the

previous future work section, the work could have been developed further if more time was available.

9.4 Self Review

I have taken great pride in my work for this project as it has been my own creation from the concept through to the development of a functional Collision Avoidance System. I believe that I have worked to the best of my ability producing a simulation that works in avoiding obstacles and I have been able to complete the best part of the objectives within the time constraints. I have found the experience of researching, creating, and testing an algorithm a challenging but rewarding one. It has been very encouraging to have my work appreciated by academics and with them being very interested in what I have been doing, especially my supervisor and the PhD student I have been working with. It was a great honour to be told that they would like to use my results to create a paper together with me, to be published and presented at a robotics conference (Cowan, et al., 2016). I feel this is an achievement few people achieve and therefore I am very thankful for the opportunity.

Throughout the project there have been sections where I could have stopped thinking and completed the section but I pushed myself to create something in a niche topic with an original solution. This has given me a great sense of project ownership and it will be satisfying to see other students carry on with the work that I have started.

I would like to still keep working on the algorithms, especially the three dimensional collision avoidance as I believe this will become a requirement with the ever increasing use of UAVs. It would be something special to one day have my name associated with this method. It will also be very practical for use in my career in engineering, especially with UAVs.

I would like to say one last thank you to my supervisor Lyudmila Mihaylova and to James Douthwaite for helping with this project and also for supporting me with future work.

10. REFERENCES

adafruit, 2015. *Installing Node.JS.* [Online]
Available at: https://learn.adafruit.com/node-embedded-development/installing-node-dot-js
[Accessed March 2016].

Albaker, B. M. & Rahim, N. A., 2009. *A Survey of Collision Avoidance Approaches for Unmanned Aerial Vehicles,* University of Malaya: UMPEDAC Research Centre.

Authority, C. A., 2015. *Unmanned Aircraft System Operations in UK Airspace - Guidance. CAP 722.* s.l.:Safety And Airspace Regulation Group.

Berg, J. v. d., Lin, M. & Manocha, D., 2008. *Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation,* Pasadena, CA: Robotics and Automation.

Berg, J. v. d., Snape, J., Guy, S. J. & Manocha, D., 2011. *Reciprocal Collision Avoidance with Acceleration-Velocity Obstacles.* Shanghai, Robotics and Automation.

Chen, H., Jikov, V. & Rong Li, X., 2015. On Threshold Optimization for Aircraft Conflict Detection. *18th Internatonal Conference on Information Fusion,* pp. 1198 - 1204.

Copter, N., 2012. *Hacker Guide.* [Online]
Available at: http://www.nodecopter.com/hack
[Accessed March 2016].

Cowan, L., Douthwaite, J. & Mihaylova, L., 2016. *Collision Avoidance within Swarming UAV's using Velocity Obstacle Trajectory Generation.* Baden-Baden Gerany, The 2016 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI 2016).

Douthwaite, J., Mihaylova, L. & Veres, S., 2016. *Enhancing Autonomy in VTOL Aircraft Based on Symbolic Computation Algorithms,* Sheffield, UK: Proceedings of the 17th Conf. Towards Autonomous Robotic Systems (TAROS-16).

elinux, 2015. *Node.js on RPi.* [Online]
Available at: http://elinux.org/Node.js_on_RPi
[Accessed March 2016].

felixge, 2014. *node-ar-drone.* [Online]
Available at: https://github.com/felixge/node-ar-drone
[Accessed March 2016].

Fiorini, P. & Shiller, Z., 1998. *Motion Planning in Dynamic Environments using Velocity Obstacles,* Los Angeles: University of California.

Geyer, C., Singh, S. & Chamberlain, L., 2008. *Avoiding Collisions Between Aircraft: State of the Art and Requirements for UAVs.* Pittsburgh, (Pennsylvania 15213): Carnegie Mellon University.

Guy, S. J. et al., 2009. *ClearPath: Highly Parallel Collision Avoidance for Multi-Agent Simulation,* New York: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation.

Jikov, V. P., Rong Li, X. & Ledet, J., 2015. An Efficient Algorithm for Aircraft Conflict Detection and Resolution Using List Viterbi. *18th International Conference on Information Fusion,* pp. 1709 - 1716.

John, S. & John, K. L., 2015. *Case Study: Efficient Algorithms for Air Traffic Management Decision Support Tools,* Hamburg, Germany: Sensor Data Fusion: Trends, Solutions, Applications .

Kernbach, S., 2013. *Handbook of Collective Robotics: Fundamentals and Challenges.* s.l.:Pan Stanford.

Lacher, A., Maroney, D. & Zeitlin, A., 2007. *Unmanned Aircraft Collision Avoidance - Technology Assessment and Evaluation Methods,* McLean, VA, USA: The MITRE Corporation.

Ladha, S., Kumar, D. K., Bhalla, P. & Jain, A., 2011. *Use of LIDAR for Obstacle Avoidance by an Autonomous Aerial Vehicle,* Dubai, UAE: Birla Institute of Technology and Science Pilani.

Large, F., Vasquez, D., Fraichard, T. & Laugier, C., 2004. *Avoiding cars and pedestrians using velocity obstacles and motion prediction.* s.l., Intelligent Vehicles Symposium, 2004 IEEE.

Murray, G., 2013. *Rotation About an Arbitrary Axis.* [Online]
Available at: http://inside.mines.edu/fs_home/gmurray/ArbitraryAxisRotation/
[Accessed April 2016].

Oaks, E., 2005. *Minkowski Sums of Simple Polygons.* s.l.:School of Computer Science, Tel-Aviv University.

Parrot, 2012. *Parrot AR.Drone Developer Guide SDK 2.0.* [Online]
Available at: http://developer.parrot.com/ar-drone.html
[Accessed April 2016].

play, W. w. w., 2015. *Raspberry Pi + Node JS.* [Online]
Available at: http://weworkweplay.com/play/raspberry-pi-nodejs/
[Accessed March 2016].

Sabatini , R., Gardi, A. & Richardson, M., 2014. LIDAR Obstacle Warning and Avoidance System for Unmanned Aircraft. *International Journal of Mechanical, Aerospace, Industrial, Mechatronic and Manufacturing Engineering*, 8(4), pp. 710 - 721.

Sezer, V. & Gokasan, M., 2012. A Novel Obstacle Avoidance Algorithm: "Follow the Gap Method". *Robotics and Autonomous Systems,* 60(9), pp. 1123-1134.

Shiller, Z., Large, F. & Sekhavat, S., 2001. *Motion Planning in Dynamic Environments: Obstacles Moving Along Arbitray Trajectories.* s.l., Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference.

Slamtec, 2015. *RPLIDAR Development Kit.* [Online]
Available at: http://www.slamtec.com/en/Lidar/A1
[Accessed Januray 2016].

Wilkie, D., Berg, J. v. d. & Manocha, D., 2009. *Generalized Velocity Obstacles,* St. Louis, MO: 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems.

Yazdi, J. et al., 2015. *Three-Dimensional Velocity Obstacle Method for UAV Deconflicting Maneuvers,* Florida: AIAA Guidance, Navigation, and Control Conference .

APPENDIX A

```
function [VelA,Vel_Collide_union] =
Avoid_Algorithm_2D(PosA,PosB,VelA,VelB,Size,Vel_Collide_Prev,Vel_start,LineColour,preci
sion)
%% Avoid Algorithm 2D
% Calculate Distance between 2 agents
Lambda_AB = sqrt((PosB(1) - PosA(1))^2+(PosB(2) - PosA(2))^2);
% Calculate Radius of Agent 2, B_hat
B_hat = Size*1.5;
% Calculate length of tanget from point A to circle
Lambda_AC = sqrt(Lambda_AB^2 - B_hat^2);
Lambda_AD = Lambda_AC;
% Calculate Angles Phi, Theta, Gamma, Beta
% Phi
phi = acos(abs(PosA(1) - PosB(1))/Lambda_AB);
% Theta
theta = asin(B_hat/Lambda_AB);
% Gamma
gamma = theta + phi;
% Beta
beta = gamma - 2*theta;
% Shift A by velocity of B to find new base point a
x_a = PosA(1) + VelB(1);
y_a = PosA(2) + VelB(2);
% Find Location of Agent B relative to Agent A
% Extend the tangents points past Agent B
% Quad 1
if (PosB(1) >= PosA(1)) && (PosB(2) >= PosA(2))
    if (PosB(1) < PosA(1)+B_hat)
        %ypos_xpos zone
        x_c = PosA(1) + (Lambda_AC * cos(gamma));
        y_c = PosA(2) + (Lambda_AC * sin(gamma));
        x_d = PosA(1) + (Lambda_AD * cos(beta));
        y_d = PosA(2) + (Lambda_AD * sin(beta));
        Grad_AC_xy = ((y_c+VelB(2)) - y_a)/((x_c+VelB(1)) - x_a);
        Grad_AD_xy = ((y_d+VelB(2)) - y_a)/((x_d+VelB(1)) - x_a);
        x_c = x_c - 3 + VelB(1);
        x_d = x_d + 3 + VelB(1);
    else
        x_c = PosA(1) + (Lambda_AC * cos(gamma));
        y_c = PosA(2) + (Lambda_AC * sin(gamma));
        x_d = PosA(1) + (Lambda_AD * cos(beta));
        y_d = PosA(2) + (Lambda_AD * sin(beta));
        Grad_AC_xy = ((y_c+VelB(2)) - y_a)/((x_c+VelB(1)) - x_a);
        Grad_AD_xy = ((y_d+VelB(2)) - y_a)/((x_d+VelB(1)) - x_a);
        x_c = x_c + 3 + VelB(1);
        x_d = x_d + 3 + VelB(1);
     end
% Quad 2
elseif (PosB(1) >= PosA(1)) && (PosB(2) <= PosA(2))
    if (PosB(1) < PosA(1)+B_hat)
        %yneg_xpos
        x_c = PosA(1) + (Lambda_AC * cos(beta));
        y_c = PosA(2) - (Lambda_AC * sin(beta));
        x_d = PosA(1) + (Lambda_AD * cos(gamma));
        y_d = PosA(2) - (Lambda_AD * sin(gamma));
        Grad_AC_xy = ((y_c+VelB(2)) - y_a)/((x_c+VelB(1)) - x_a);
        Grad_AD_xy = ((y_d+VelB(2)) - y_a)/((x_d+VelB(1)) - x_a);
        x_c = x_c + 3 + VelB(1);
        x_d = x_d - 3 + VelB(1);
    else
        x_c = PosA(1) + (Lambda_AC * cos(beta));
        y_c = PosA(2) - (Lambda_AC * sin(beta));
        x_d = PosA(1) + (Lambda_AD * cos(gamma));
        y_d = PosA(2) - (Lambda_AD * sin(gamma));
        Grad_AC_xy = ((y_c+VelB(2)) - y_a)/((x_c+VelB(1)) - x_a);
        Grad_AD_xy = ((y_d+VelB(2)) - y_a)/((x_d+VelB(1)) - x_a);
        x_c = x_c + 3 + VelB(1);
        x_d = x_d + 3 + VelB(1);
    end
% Quad 3
elseif (PosB(1) <= PosA(1)) && (PosB(2) <= PosA(2))
    if (PosB(1) > PosA(1)-B_hat)
        %yneg_xneg
        x_c = PosA(1) - (Lambda_AC * cos(gamma));
```

```
        y_c = PosA(2) - (Lambda_AC * sin(gamma));
        x_d = PosA(1) - (Lambda_AD * cos(beta));
        y_d = PosA(2) - (Lambda_AD * sin(beta));
        Grad_AC_xy = ((y_c+VelB(2)) - y_a)/((x_c+VelB(1)) - x_a);
        Grad_AD_xy = ((y_d+VelB(2)) - y_a)/((x_d+VelB(1)) - x_a);
        x_c = x_c + 3 + VelB(1);
        x_d = x_d - 3 + VelB(1);
    else
        x_c = PosA(1) - (Lambda_AC * cos(gamma));
        y_c = PosA(2) - (Lambda_AC * sin(gamma));
        x_d = PosA(1) - (Lambda_AD * cos(beta));
        y_d = PosA(2) - (Lambda_AD * sin(beta));
        Grad_AC_xy = ((y_c+VelB(2)) - y_a)/((x_c+VelB(1)) - x_a);
        Grad_AD_xy = ((y_d+VelB(2)) - y_a)/((x_d+VelB(1)) - x_a);
        x_c = x_c - 3 + VelB(1);
        x_d = x_d - 3 + VelB(1);
    end
% Quad 4
elseif (PosB(1) < PosA(1)) && (PosB(2) >= PosA(2))
    if (PosB(1) > PosA(1)-B_hat)
        % ypos_xneg
        x_c = PosA(1) - (Lambda_AC * cos(beta));
        y_c = PosA(2) + (Lambda_AC * sin(beta));
        x_d = PosA(1) - (Lambda_AD * cos(gamma));
        y_d = PosA(2) + (Lambda_AD * sin(gamma));
        Grad_AC_xy = ((y_c+VelB(2)) - y_a)/((x_c+VelB(1)) - x_a);
        Grad_AD_xy = ((y_d+VelB(2)) - y_a)/((x_d+VelB(1)) - x_a);
        x_c = x_c - 3 + VelB(1);
        x_d = x_d + 3 + VelB(1);
    else
        x_c = PosA(1) - (Lambda_AC * cos(beta));
        y_c = PosA(2) + (Lambda_AC * sin(beta));
        x_d = PosA(1) - (Lambda_AD * cos(gamma));
        y_d = PosA(2) + (Lambda_AD * sin(gamma));
        Grad_AC_xy = ((y_c+VelB(2)) - y_a)/((x_c+VelB(1)) - x_a);
        Grad_AD_xy = ((y_d+VelB(2)) - y_a)/((x_d+VelB(1)) - x_a);
        x_c = x_c - 3 + VelB(1);
        x_d = x_d - 3 + VelB(1);
    end
end
% Extending the y coordinates of C and D by using equation of a line y = mx+x
c_AC_xy = y_a - Grad_AC_xy*x_a;
c_AD_xy = y_a - Grad_AD_xy*x_a;
y_c = Grad_AC_xy*(x_c) + c_AC_xy;
y_d = Grad_AD_xy*(x_d) + c_AD_xy;
% Positions of Points Shifted by VelB
Posa = [PosA(1)+VelB(1),PosA(2)+VelB(2),PosA(3)+VelB(3)];
PosB = [PosB(1),PosB(2),PosB(3)];
PosC = [x_c,y_c];
PosD = [x_d,y_d];
% Set up scan starting at -5m away
x = PosA(1) - 5;
y = PosA(2) - 5;
% Create size of variables
Vel_Collide = ones(100,2)*10;
Vel_No_Collide = ones(500,2)*10;
a = 1;
b = 1;
p1 = Posa;
p2 = PosC;
p3 = PosD;
for i = 0:10/precision
    for j = 0:10/precision
        p = [x,y];
        % Calculating Barycentric Coordinates
        alpha = ((p2(2) - p3(2))*(p(1) - p3(1)) + (p3(1) - p2(1))*(p(2) - p3(2))) /
((p2(2) - p3(2))*(p1(1) - p3(1)) + (p3(1) - p2(1))*(p1(2) - p3(2)));
        eta  = ((p3(2) - p1(2))*(p(1) - p3(1)) + (p1(1) - p3(1))*(p(2) - p3(2))) /
((p2(2) - p3(2))*(p1(1) - p3(1)) + (p3(1) - p2(1))*(p1(2) - p3(2)));
        zeta = 1 - alpha - eta;
        if ((alpha >= 0) && (eta >= 0) && (zeta >= 0) && (alpha <= 1) && (eta <= 1) &&
(zeta <= 1))
            P_Col = [x,y];
            V_Col = P_Col - [PosA(1),PosA(2)];
            Vel_Collide(a,:) = round(V_Col/precision)*precision;
            a = a + 1;
        else
            P_NoCol = [x,y];
            V_NoCol = P_NoCol - [PosA(1),PosA(2)];
```

```
            Vel_No_Collide(b,:) = V_NoCol;
            b = b + 1;
        end
        y = y + precision;
    end
    y = PosA(2) - 5;
    x = x + precision;
end
% Create union of past velocity obstacles
Vel_Collide_union = union(Vel_Collide_Prev,Vel_Collide,'rows');
% Set size of variables
Near_dist = ones(length(Vel_No_Collide),1)*10;
temp = zeros(length(Vel_No_Collide),2);
n = 1;
% Check if VelA is a Collision Velocity
while ismember(VelA(1:2),Vel_Collide_union,'rows') == 1
    if n > 1
        Vel_No_Collide(I,:) = [];
    end
    % Change Velocity to nearest value that isn't in the collision set
    for m = 1:length(Vel_No_Collide)
        temp(m,:) = VelA(:,1:2) - Vel_No_Collide(m,:);
        Temp = sqrt(temp(m,1)^2 + temp(m,2)^2);
        Near_dist(m) = Temp;
    end
    [Closest,I] = min(Near_dist);
    New_Vel = Vel_No_Collide(I,:);
    VelA(:,1:2) = New_Vel;
    n = n + 1;
end
% If Vel start not a collision velocity use vel start as new velocity
if ismember(Vel_start(:,1:2),Vel_Collide_union,'rows') == 0
    VelA(:,1:2) = Vel_start(:,1:2);
end
%%% Plotting functions
plot(PosA(1),PosA(2),'or'); hold on
viscircles(PosA(:,1:2),Size/2,'EdgeColor',LineColour,'LineWidth',0.8);
viscircles(PosB(:,1:2),Size/2,'EdgeColor',LineColour,'LineWidth',0.8);
NewVel = plot(VelA(1)+PosA(1),VelA(2)+PosA(2),'.b');
StartVel = plot(Vel_start(1)+PosA(1),Vel_start(2)+PosA(2),'.r');
plot([Posa(1),PosC(1)],[Posa(2),PosC(2)],'Color',LineColour);
plot([Posa(1),PosD(1)],[Posa(2),PosD(2)],'Color',LineColour);
drawnow
refresh
xlim([-20 20]);
ylim([-20 20]);
grid on
title('Testing New Velocity Vector')
xlabel('x axis')
ylabel('y axis')
legend([StartVel,NewVel],' Start Velocity ','  New Velocity')
end
```

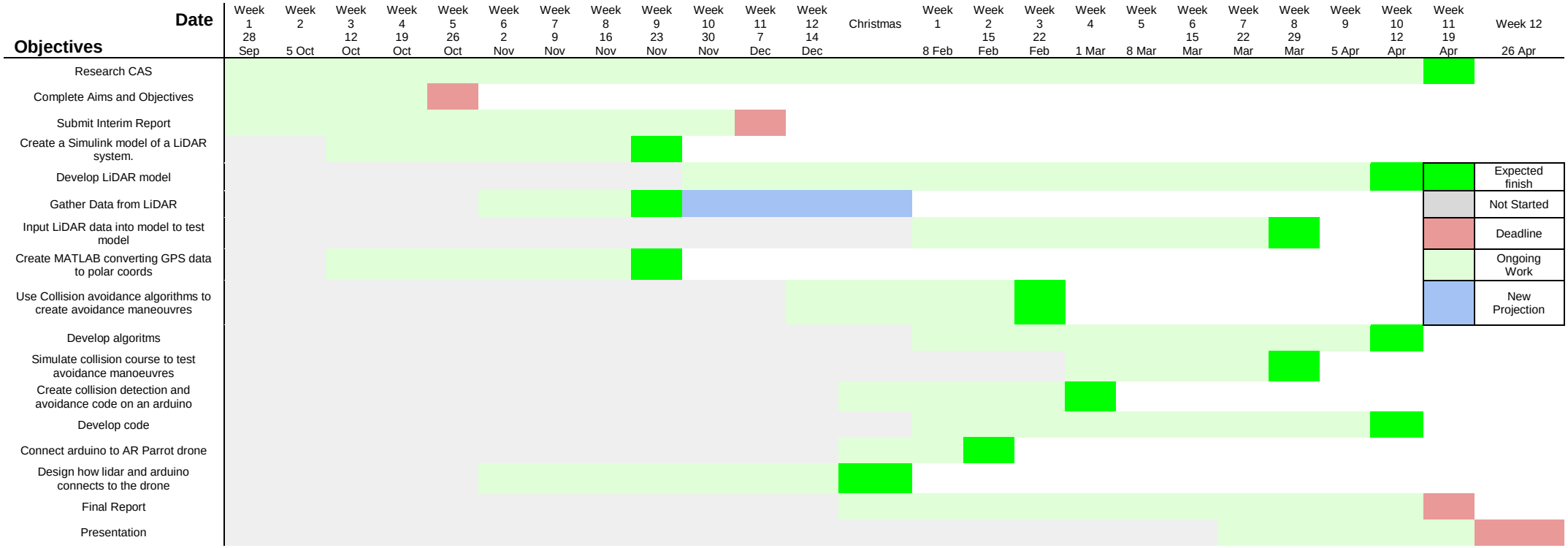
APPENDIX B

```
function [VelA,Vel_Collide,Vel_No_Collide] =
Avoid_Algorithm_3D_rotate(PosA,PosB,VelA,VelB,Size,Vel_Collide_Prev,Vel_Start,LineColou
r)
%% Avoid Algorithm 3D
%Positions of Points Shifted by VelB
Posa = [PosA(1)+VelB(1),PosA(2)+VelB(2),PosA(3)+VelB(3),1];
Posb = [PosB(1)+VelB(1),PosB(2)+VelB(2),PosB(3)+VelB(3),1];
Vel_No_Collide = [0,0,0];
Vel_Collide = zeros(20000,3);
Quad1_yz = 0;
Quad2_yz = 0;
Quad3_yz = 0;
Quad4_yz = 0;
x_b = Posb(1);
%Location of Agent B xz plane
if (PosB(1) > PosA(1)) && (PosB(3) > PosA(3))
    x_b = x_b + 3;
elseif (PosB(1) > PosA(1)) && (PosB(3) <= PosA(3))
    x_b = x_b + 3;
elseif (PosB(1) < PosA(1)) && (PosB(3) < PosA(3))
    x_b = x_b - 3;
elseif (PosB(1) < PosA(1)) && (PosB(3) >= PosA(3))
    x_b = x_b - 3;
end
%Location of Agent B yz plane
if (PosB(2) > PosA(2)) && (PosB(3) > PosA(3))
    Quad1_yz = 1;
elseif (PosB(2) > PosA(2)) && (PosB(3) <= PosA(3))
    Quad2_yz = 1;
elseif (PosB(2) < PosA(2)) && (PosB(3) < PosA(3))
    Quad3_yz = 1;
elseif (PosB(2) < PosA(2)) && (PosB(3) >= PosA(3))
    Quad4_yz = 1;
else %(PosB(2) == PosA(2)) && (PosB(3) == PosA(3))
    Posb = Posb + 0.0000000000000001;
end
% Boundary Conditions
if PosB(1) == PosA(1) && PosB(2) == PosA(2)
    if PosB(3) > PosA(3)
        z_b = PosB(3) + 3;
    elseif PosB(3) < PosA(3)
        z_b = PosB(3) - 3;
    end
    x_b = PosB(1);
    y_b = PosB(2);
    Posb = [x_b,y_b,z_b,1];
elseif PosB(1) == PosA(1)
    if Quad1_yz == 1
        x_b = Posb(1);
        y_b = PosB(2) + 3;
    elseif Quad2_yz == 1
        x_b = PosB(1);
        y_b = PosB(2) + 3;
    elseif Quad3_yz == 1
        x_b = PosB(1);
        y_b = PosB(2) - 3;
    elseif Quad4_yz == 1
        x_b = PosB(1);
        y_b = PosB(2) - 3;
    end
    Grad_AB_y = (Posa(3) - Posb(3)) / (Posa(2) - Posb(2));
    c_yz = Posa(3) - Grad_AB_y*Posa(2);
    z_b = Grad_AB_y*y_b + c_yz;
    Posb = [x_b,y_b,z_b,1];
elseif PosB(2) == PosA(2)
    y_b = Posb(2);
    Grad_AB_z = (Posa(3) - Posb(3)) / (Posa(1) - Posb(1));
    c_xz = Posa(3) - Grad_AB_z*Posa(1);
    z_b = Grad_AB_z*x_b + c_xz;
    Posb = [x_b,y_b,z_b,1];
elseif PosB(3) == PosA(3)
    z_b = Posb(3);
    Grad_AB_xy = (Posa(2) - Posb(2)) / (Posa(1) - Posb(1));
    c_xy = Posa(2) - Grad_AB_xy*Posa(1);
    y_b = Grad_AB_xy*x_b + c_xy;
```

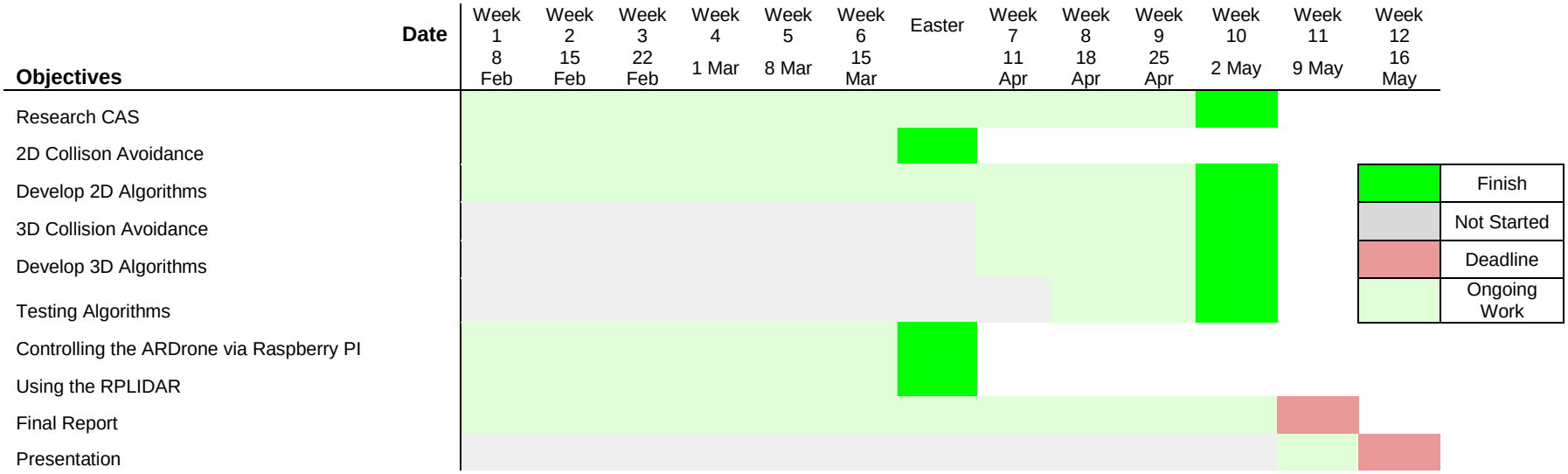
```
    Posb = [x_b,y_b,z_b,1];
else
    Grad_AB_z = (Posa(3) - Posb(3)) / (Posa(1) - Posb(1));
    Grad_AB_y = (Posa(3) - Posb(3)) / (Posa(2) - Posb(2));
    c_xz = Posa(3) - Grad_AB_z*Posa(1);
    c_yz = Posa(3) - Grad_AB_y*Posa(2);
    z_b = Grad_AB_z*x_b + c_xz;
    y_b = (z_b - c_yz)/Grad_AB_y;
    Posb = [x_b,y_b,z_b,1];
end
Lambda_AB_xz = sqrt((Posb(1) - Posa(1))^2 + (Posb(3) - Posa(3))^2);
% Radius of cone at Agent B
B_hat = Size*1.5;
if Lambda_AB_xz == 0
    theta_xy = 0;
else
    theta_xy = asin(B_hat/Lambda_AB_xz);
end
% Transformation Matrices
T = [1 0 0 -Posa(1);
     0 1 0 -Posa(2);
     0 0 1 -Posa(3);
     0 0 0     1  ];
T_back = [1 0 0 Posa(1);
          0 1 0 Posa(2);
          0 0 1 Posa(3);
          0 0 0    1  ];
% Agent A and B translated to origin
A = (T * Posa')';
B = (T * Posb')';
% Angles to rotate to z axis
omega = -atan(sqrt(B(1)^2+B(2)^2)/B(3));
kapa = -atan(B(1)/B(2));
%Rotation Matrices
Rz = [cos(kapa) sin(kapa) 0 0;
     -sin(kapa) cos(kapa) 0 0;
          0        0      1 0;
          0        0      0 1];
if Posa(2) < Posb(2)
    Rx = [1      0           0      0;
          0  cos(omega)  sin(omega) 0;
          0 -sin(omega)  cos(omega) 0;
          0      0           0      1];
else
    Rx = [1      0            0     0;
          0  cos(omega) -sin(omega) 0;
          0  sin(omega)  cos(omega) 0;
          0      0           0      1];
end
R = Rx*Rz;
R_back = Rz'*Rx';
B_new = (R* B');
Length = ceil(B_new(3));
r = zeros(Length,1);
j = 1;
f = 1;
precision = 0.1;
C = [];
% Creating points inside cone
for i = 0:precision:Length
    r(j) = (Length - i) * sin(theta_xy);
    % Centre of each disc
    Cx = 0;
    Cy = 0;
    Cz = Length - i;
    C = [C;Cx,Cy,Cz];
    for m = -r(j):precision:r(j)
        for n = 0:pi/8:2*pi
            x = m*sin(n);
            y = m*cos(n);
            z = 0;
            px = (C(j,1) + x);
            py = (C(j,2) + y);
            pz = (C(j,3) + z);
            V_Coll = [px,py,pz,2] - A;
            V_Coll_rot =  (R_back * V_Coll');
            V_Coll_trans = (T_back * V_Coll_rot)';
            V_Coll_trans = round(V_Coll_trans / precision)*precision;
            Vel_Collide(f,:) = V_Coll_trans(:,1:3);
```

```
            f = f + 1;
        end
    end
    j = j + 1;
end
% Union of collision velocities
Vel_Collide = union(Vel_Collide_Prev,Vel_Collide,'rows');
max_vel = 5;
% If Start velocity not a collision velocity uses start velocity as new
% velocity
if ismember(Vel_Collide,Vel_Start,'rows') == 0
    VelA = Vel_Start;
% If VelA a collision velocity calculate new velocity
elseif any(ismember(Vel_Collide,VelA,'rows')) == 1
    A_tilda = PosA + VelA;
    qx = A_tilda(1) - B_hat;
    qy = A_tilda(2) - B_hat;
    qz = A_tilda(3) - B_hat;
    Vel = [];
    f = 1;
    precision = 0.5;
    % Scanning area around the current velocity vector
    for i = 0:precision:2*B_hat
       for j = 0:precision:2*B_hat
          for k = 0:precision:2*B_hat
             Q = [qx,qy,qz];
             if (Q - PosA) <= (sqrt(max_vel^2 + max_vel^2))
                V = Q - PosA;
             end
             Vel = [Vel;V];
             if any(ismember(Vel_Collide,Vel(f,:),'rows')) == 0
                 Vel_No_Collide = [Vel_No_Collide;Vel(f,:)];
             end
             f = f + 1;
             qx = qx + precision;
          end
          qx = A_tilda(1) - B_hat;
          qy = qy + precision;
       end
       qy = A_tilda(2) - B_hat;
       qz = qz + precision;
    end
    Near_dist = [];
    % Calculating new velocity
    for m = 1:length(Vel_No_Collide)
        temp = VelA - Vel_No_Collide(m,:);
        Temp = sqrt(temp(1)^2 + temp(2)^2 + temp(3)^2);
        Near_dist = [Near_dist;Temp];
    end
    [Closest,I] = min(Near_dist);
    New_Vel = Vel_No_Collide(I,:);
    VelA = New_Vel;
end
%%% Plotting Functions
[X,Y,Z] = sphere;
plot3(PosA(1),PosA(2),PosA(3),'sr'); hold on
plot3(VelA(1),VelA(2),VelA(3),'ob');
plot3(Vel_Start(1),Vel_Start(2),Vel_Start(3),'or');
Cone(Posa(:,1:3),Posb(:,1:3),[0 B_hat],20,LineColour,0,0); hold on
alpha(0.1)
surf(X+PosB(1),Y+PosB(2),Z+PosB(3));
xlim([-5 5]);
ylim([-5 5]);
zlim([-2 8]);
grid on
drawnow
refresh
title('New Velocity')
xlabel('x axis')
ylabel('y axis')
zlabel('z axis')
end
```

APPENDIX C


Date
Objectives
Week 1 28 Sep
Week 2 5 Oct
Week 3 12 Oct
Week 4 19 Oct
Week 5 26 Oct
Week 6 2 Nov
Week 7 9 Nov
Week 8 16 Nov
Week 9 23 Nov
Week 10 30 Nov
Week 11 7 Dec
Week 12 14 Dec
Christmas
Week 1 8 Feb
Week 2 15 Feb
Week 3 22 Feb
Week 4 1 Mar
Week 5 8 Mar
Week 6 15 Mar
Week 7 22 Mar
Week 8 29 Mar
Week 9 5 Apr
Week 10 12 Apr
Week 11 19 Apr
Week 12 26 Apr
Research CAS
Complete Aims and Objectives
Submit Interim Report
Create a Simulink model of a LiDAR system.
Develop LiDAR model
Gather Data from LiDAR
Input LiDAR data into model to test model
Create MATLAB converting GPS data to polar coords
Use Collision avoidance algorithms to create avoidance maneouvres
Develop algoritms
Simulate collision course to test avoidance manoeuvres
Create collision detection and avoidance code on an arduino
Develop code
Connect arduino to AR Parrot drone
Design how lidar and arduino connects to the drone
Final Report
Presentation
Expected finish
Not Started
Deadline
Ongoing Work
New Projection

APPENDIX D


Date
Objectives
Week 1 8 Feb
Week 2 15 Feb
Week 3 22 Feb
Week 4 1 Mar
Week 5 8 Mar
Week 6 15 Mar
Easter
Week 7 11 Apr
Week 8 18 Apr
Week 9 25 Apr
Week 10 2 May
Week 11 9 May
Week 12 16 May
Research CAS
2D Collison Avoidance
Develop 2D Algorithms
3D Collision Avoidance
Develop 3D Algorithms
Testing Algorithms
Controlling the ARDrone via Raspberry PI
Using the RPLIDAR
Final Report
Presentation
Finish
Not Started
Deadline
Ongoing Work