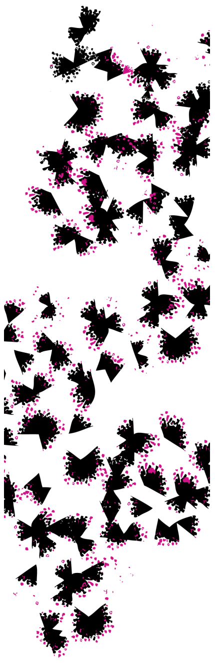
RAM ROBOTICS AND MECHATRONICS



Implementation of the Velocity Obstacle Principle for 3D Dynamic Obstacle Avoidance in Quadcopter Waypoint Navigation

A. (Alex) Kamphuis

BSc Report

September 2014

Report nr. 019RAM2014
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

CTIT

Summary

Previous work at the RAM flight arena has demonstrated the possibility of using multiple quadcopters to achieve a common task in a shared environment. In such a scenario we need a distributed architecture to perform navigation in the presence of dynamic obstacles.

This bachelor's assignment aims to control quadcopters for waypoint navigation, so that they avoid obstacles and other quadcopters. The proposed work makes use of the Velocity Obstacle Principle extended to a 3D environment in order to ensure collision free trajectories during the navigation.

The principle has been implemented in the form of a distributed architecture in which, for every robot, one instance of the algorithm is running. The method requires information on the object's state. The system's modular nature allows dynamic addition and removing of objects to or from the system.

Proof of principle is achieved through various simulations and performances are tested using a different test-runs.

The implementation makes use of a very generic and modular approach that enables future developments of complex behaviours. Using the Robot Operating System the algorithm can be used in any navigation system that has a position feedback and position control, and is independent from the nature of the robot.

Contents

Summary iii

1 Introduction 1

2 Architecture 2

2.1 System Overview 2

2.2 Hardware 2

2.3 Software 4

3 Theory 6

3.1 Quadcopter model 6

3.2 The velocity obstacle 6

3.3 Position controller 8

4 Implementation 9

4.1 Algorithms 9

4.2 Additions for actual implementation 11

5 Results 15

5.1 No other agents 15

5.2 Avoiding on a single position 15

5.3 Avoiding while moving 16

6 Conclusion and Discussion 17

6.1 Necessity of reference gain 17

6.2 Avoidance Strategy 17

7 Future Work 18

7.1 Specific for this work 18

7.2 Implementation of a time horizon 18

7.3 Vision 20

8 Tutorial 21

8.1 Requirements 21

8.2 Setting up your gear 21

8.3 Recieving position feedback 22

8.4 First flight test 23

8.5 Starting the velocity obstacle algorithm 23

8.6 Providing targets 25

8.7 Adjusting behaviour 25
8.8 Deeper understanding 25

Bibliography **26**

1 Introduction

UAV's offer great potential in many different situations. The most obvious being situations that are either too costly or too dangerous for humans. Furthermore they have proven themselves useful in recording, search, surveillance and recreation. Especially multicopters provide a versatile platform to apply in many different situations.

In the past years multicopters have dropped in price, and risen in capability. The increased availability of these machines has given rise to an increase in the need to control these machines easily.

The limits in speed, size and cargo capacity can on some occasions be overcome by using multiple multicopters. This does however result in an increasing complexity of the required control methods. Because the user is (in most cases) unable to keep up with the movement of large amounts of multicopters there is a need for a certain amount of autonomy.

In an constantly changing environment that involves humans, other robots and inanimate objects an important requirement to an autonomous UAV is obstacle avoidance. This can be achieved on different levels of autonomy.

Each level of autonomy has its own advantages and disadvantages that eventually determine the method of choice for a certain application.

This bachelor's assignment implements a method to avoid obstacles in a dynamic environment. The perspective is strongly modular and generic so that this work may be reused for any type of position feedback system and any type of moveable robot (not limited to UAV's).

2 Architecture

In this chapter the hardware and software architecture used during this assignment are explained.

2.1 System Overview

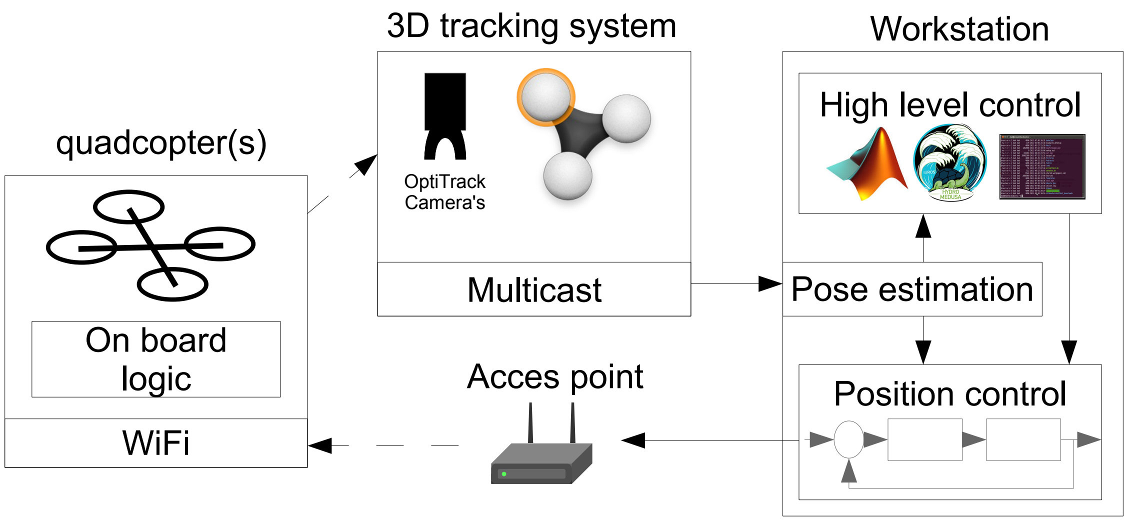


Figure 2.1: Overview of the system used

To perform experiments in this assignment the setup as seen in Figure 2.1 was used.

The quadcopter is a Parrot AR.Drone 2 (Parrot, 2012) which was tracked by an OptiTrack (NaturalPoint, 2011) system. The system uses Motive software (NaturalPoint, 2011) that sends pose information via multicast to another PC that we will call workstation.

The workstation runs Ubuntu 12.04 (Ubuntu, 2012) and ROS Hydro (Foundation, 2013). It runs multiple instances of the same position controller (Trouwborst, 2014), one for each robot.

The workstation is connected to a wireless access point that uses WiFi to communicate to one or more drones.

A detailed explanation of some of the system components can be found below.

2.2 Hardware

2.2.1 *NaturalPoint OptiTrack* & *Motive*

For state feedback of the systems controlled during this assignment a set of camera's was used to perform localization.

All tracked objects were uniquely labelled with infrared-reflecting buds so that the infrared camera's in Figure 2.2a can localize them. The information from multiple camera's is then interpreted by software called *Motive* of which a screenshot can be found in Figure 2.2b.

2.2.2 Quadcopter

The Parrot AR.Drone (Figure 2.3), the machine used for testing, is a commercially available quadcopter that can be controlled very simply with a smartphone or tablet. It is a remarkably stable platform due to a great deal of internal logic and control. Previous work at RAM has

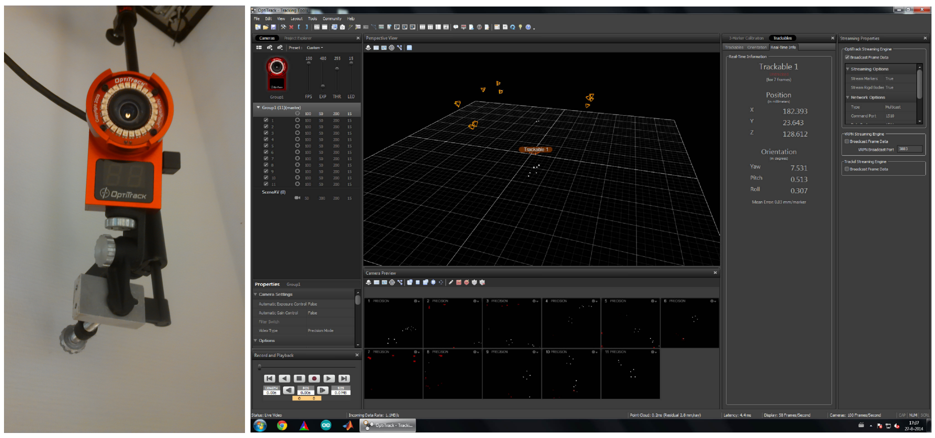

(a) OptiTrack camera **(b)** Motive Software

Figure 2.2: NaturalPoint soft- and hardware for localization

made position control for this device possible. Furthermore the sturdy nature of these robots make them ideal for testing indoors.

Figure 2.3: The Parrot AR.Drone, used for testing

2.2.3 Setup

The hardware set-up can vary for different systems but during this assignment the set up was as seen in Figure 2.4.

The dotted connection in Figure 2.4 is only used during initialization of the system.

The OptiTrack cammera's are connected through Power over Ethernet to the OptiTrack PC, which runs Motive software. During initialization a remote desktop application is used to set up the Motive software by calibrating the camera's and selecting the trackables. The OptiTrack pc then streams pose information on all selected objects to the Linux system via Multicast.

The Linux system connects to every drone consecutively, and uses the telnet protocol to send it settings. These settings make the drone connect to the wireless access point with a static IP-address. Afterwards the Linux PC, connected to the access point either by wire or WiFi, can reach the drones through the access point at the static IP's.

The OptiTrack camera's then use reflected infra-red light to detect the drones and generate pose information.

2.3 Software

2.3.1 `Matlab`

`Matlab`, MathWorks (2013), is a high-level programming language aimed at a technical audience. It offers a quick way to visualize data, create and debug algorithms.

In this assignment it was used to model the behaviour of various agents in a two dimensional environment after the implementation of different algorithms that make use of the velocity obstacle principle. It was used to demonstrate solveability before implementation of the algorithms on another platform. An elaborate explanation of how Matlab was used and the results can be seen in Section 4.1.

Furthermore this software has been used to interpret and plot data generated during experiments, and to create an graphical user interface that allows for easy sending of targets to two quadcopters.

2.3.2 ROS

The robot operating system (ROS) is a framework that can be used for the development of applications for robots. It is released under an open source license; therefore there is a large community that has developed numerous packages performing a variety of functions.

The robot operating system is built highly modular which nicely fits the goals of this assignment. There are several concepts in ROS that might need some explaining:

- **Node** - A piece of software that performs calculations and communicates with other nodes.
- **Topic** - A 'mailbox' in which nodes can place messages and read messages from.
- **Message** - A predefined data-structure that can contain anything from a boolean value to multiple variables and complex imagery structures.
- **Package** - A collection of nodes, messages and more ROS files that perform a task.

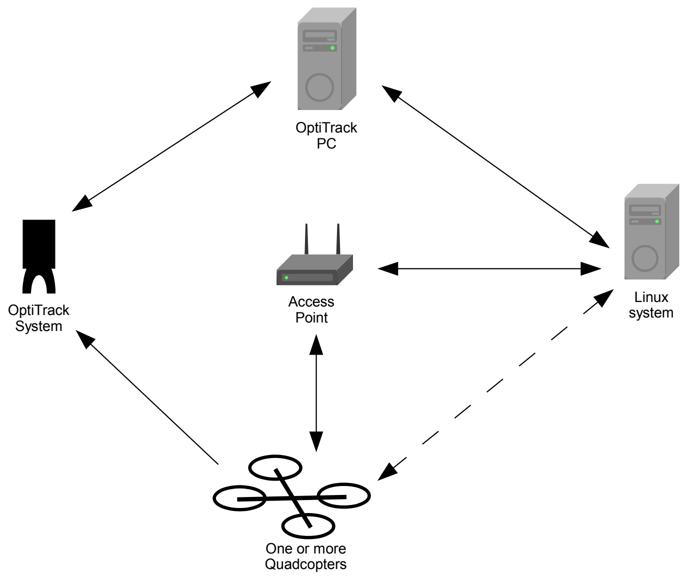


Figure 2.4: Hardware setup during experimentation

ROS's modular nature is very well illustrated by the topic/message system: whenever a node places (publishes) a message on a topic, it does not know if or by whom it is read (subscribed). Nodes can also subscribe to topics without knowing if or whom is publishing to it. Additionally nodes can publish and subscribe to arbitrarily many topics at the same time.

This makes it possible for us to create one node that performs the calculation of velocity obstacles, and run it many times, for every robot.

2.3.3 Setup

The software create in this assignment can be schematically represented as blocks as seen in Figure 2.5. This was the setup as used during this assignment. Instances of the software do not necessary need to be run on the same machine.

The dotted blocks are newly added, whereas the others are pre-existing (Gräve, 2010), (Monajjemi, 2012), (Trouwborst, 2014).

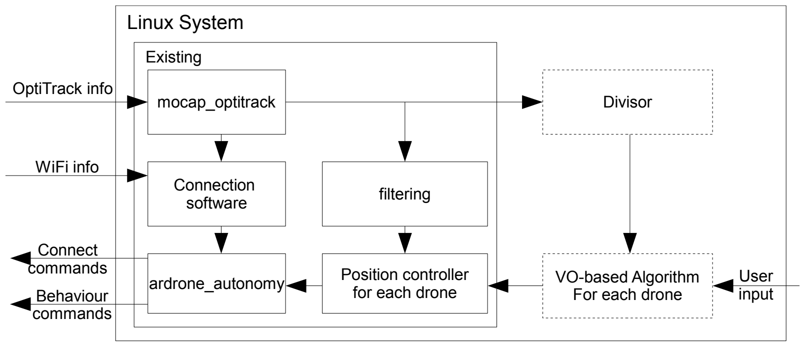


Figure 2.5: Software Overview

3 Theory

This chapter will treat some of the theoretical concepts used during this bachelor's assignment.

3.1 Quadcopter model

The dynamics of quadcopters have been studied and modelled extensively in many different ways ranging from very abstract to very specific. (Charles Richter, 2010), (Markus Hehn, 2011), (Daniel Mellinger, 2011)

Most of the pre-existing models are machine and controller specific, which this assignment tries to avoid. We do however need some sort of estimate as to what our 'perfect' controller (Figure 3.3) can do in combination with the quadcopter. The simplest way to restrict movements in a semi-natural way is to give limits to the velocity and acceleration.

Though it is highly unlikely that a quadcopter can accelerate or move at the same rate in any direction *that is what we assume.* The assumption was made for simplicity and generality.

Furthermore both quadcopters and objects are modelled as spherical in shape because this provides generality and simplicity. More complex shapes can be made by using multiple spheres to represent them.

This highly simplified model of the Quadcopters allows for easy extension to other platforms.

3.2 The velocity obstacle

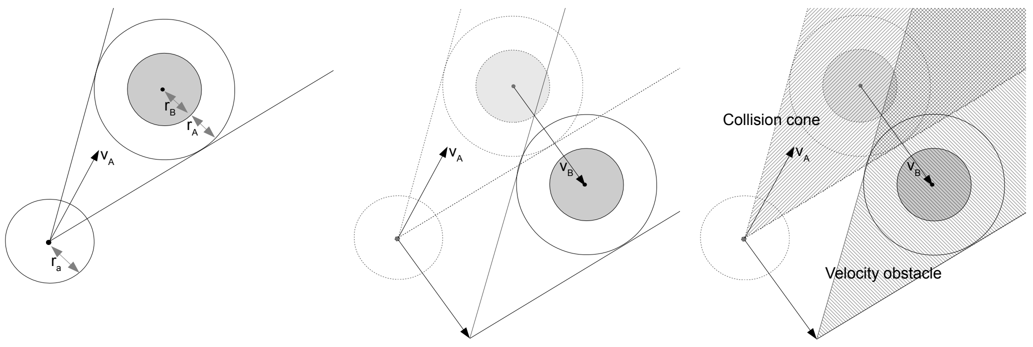
The velocity obstacle is a concept used for obstacle avoidance on a local scale. The concept and its use in this report are explained below.

3.2.1 Constructing a velocity obstacle

The construction of a velocity obstacle starts with the construction of a collision cone. *The collision cone for obstacle B on robot A can be described as the collection of velocities for robot A that would result in collision with object B.*

3.2.2 Collision cone

Let us assume a robot with radius r_A and an object with radius r_B. If one draws a circle around the object with radius $r_A + r_B$, it becomes possible to define the borders of the collission cone: these can be found by drawing lines tangent to the circle with radius $r_A + r_B$ through the center-point of robot A. This proces can be seen in Figure 3.1a.



(a) Collision Cone **(b)** Velocity obstacle **(c)** Marked cone and obstacle

Figure 3.1: Stages in constructing a velocity obstacle

In 3D space the cirlces in Figure 3.1a become spheres. The tangent lines to this sphere through the centerpoint of robot A touch the sphere in a circle. Let us define this circle as the base of the collision cone. Due to the axial symmetric nature the collission cone can be fully defined by two tangent lines touching the base directly across the base of each other.

The velocity v_A in Figure 3.1a is between the two tangent lines and would therefore result in a collission. Qualitatively this is easy to see: the centre of robot A will at some point in time lie within the circle with radius $r_A + r_B$, meaning the distance between the two centerpoints of A and B would have to be smaller then the sum of their radii, which is not possible.

For the sake of clarity the collision cone has been highlighted in Figure 3.1c.

Translation

If object B has a velocity itself however, the above conclusion might not be correct. To account for the movement of obstacle B one can simply translate the collission cone with the velocity of obstacle B as is done in Figure 3.1b. The result is defined as the velocity obstacle caused by obstacle B on robot A. Note that for a static obstacle its velocity obstacle is equal to its collision cone since then $v_B = 0$.

The velocity obstacle of B on A is a collection of all velocities of A that will eventually result in collision if B maintains its current velocity.

Now that a velocity obstacle is calculated for one object on the robot, other velocity obstacles can be added to it, so as to create a larger set of velocities that will end in collision.

Because all velocity obstacles calculated this way are similar in shape (cone-shaped in three dimensions, triangle shaped in 2D) it is sufficient to store only their position (apex of the cone or triangle), aperture (angle between the two tangent lines) and orientation.

3.2.3 Collision detection

To use the velocity obstacles calculated to make decisions on which velocities (not) to choose we need to have a way of detecting whether a point is in a velocity obstacle. Below a generic approach, that can be used in 2D and 3D, is described. For the sake of clarity it will be explained in 2D.

First the velocity obstacle is drawn in Figure 3.2. The apex of the cone labeled a and the tangent points *tp₁* and *tp₂* are given by their location.

We define a vector *base* running from a to the base-center (which can be calculated by taking the average of the two tangent points). Furthermore we define a vector v running from a to the point to be tested, x. Using the dot product of the two vectors the *test angle* can be determined.

The *aperture* can be calculated by using the inverse tangent function for the length of the *axis*, which runs from a perpendicular to the *base*, and half the *base*.

Because the velocity obstacle is infinitely large to the right side of Figure 3.2 point x is in the velocity obstacle if the *test angle* is smaller then the *aperture*

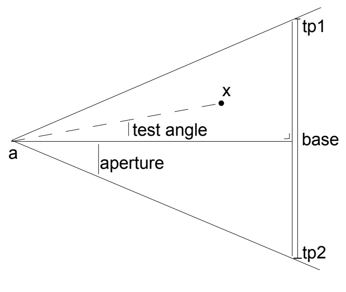
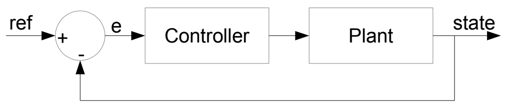
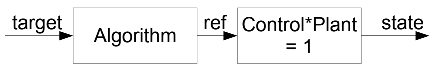


Figure 3.2: Angles for testing whether a point is in a velocity obstacle

3.3 Position controller



(a) A standard controller configuration



(b) Assuming a perfect controller

Figure 3.3: Schematics showing a pefect controller

For the sake of modularity (as mentioned in Chapter 1) an assumption was made on the existence of a position controller that accepts setpoints in Euclidian space. It is a deliberate choice to assume a *perfect* position controller.

This implies that whatever setpoint we send to the controller, it will make sure the robot gets there by the next timestep. During the test phase of this assignment it became clear that this assumption was not reasonable and so a workaround was implemented, on which more is written in Section 4.2.3.

The assumption of the perfect controller creates a certain distinction between a unit generating setpoints and the plant, because there is to be no estimation of the robot (model) in the calculation of a setpoint. This allows us to use the very same path planning algorythm for any type of robot, both flying and ground robots, even if the system is non-holonic and/or underactuated.

The traditional view of a controller can be seen in Figure 3.3a. When one assumes the controller to function perfectly the product of the controller and the plant (robot) equals one: there is no difference between the desired and actual state.

The obstacle avoiding algorythm to be made will produce setpoints to put in this controller based on a target, which is a waypoint for navigation. The over-simplified scheme representing this configuration can be found in Figure 3.3b.

4 Implementation

This chapter elaborates on the work done to achieve the results in Chapter 5. For a details on how to use this work please refer to Chapter 8.

4.1 Algorithms

The functions/algorithms explained here are set up for one robot-object combination. The actual implementation in `Matlab` and `C++` differ: implementation in `Matlab` is highly vectorized for speed, whereas implementation in `C++` is distributed. Nevertheless the algorithms presented below describe the flow of the code correctly and do not differ in functionality.

Simulation and actual implementation also differ in dimensionality; the actual `C++` implementation is in 3D whereas the `Matlab` simulation is in 2D for clarity and speed. This chapter describes the algorithms for both, and elaborates on the differences.

The top level script has the workflow illustrated in Algorithm 1. Furthermore it uses several function files that are described below.

The outermost loop is run once every timestep and updates the position using the Euler method for integration of velocity v_i at time t to position p_i at time $t = t + 1$.

Some terminology can be found in Table 4.1.

4.1.1 `calcPrefv`

This function generates, for robot i at position p_i, a vector $v_i\star$ pointing in the direction of the target g_i.

The vector is then scaled with the maximum allowed velocity $v_{max,i}$. This way, the norm of the vector representing the reference velocity to be assigned at each robot is equal to the maximum velocity $v_{max,i}$.

When a robot i nears its target g_i its reference velocity $v_i\star$ that is assigned to it is scaled down linearly can be seen in Figure 4.1 to avoid overshoot. Note that this way of assigning the reference velocity to the robots represents one simple and effective method, although other methods can be implemented and used to generate smooth trajectories to reach the goal.

Algorithm 2 reports the workflow that has been implemented to generate the reference velocity v_i for robot i.

Name	Explanation
i	The active robot.
n	The number of agents.
$vmax_i$	The unidirectional maximum velocity for robot i.
$amax_i$	The unidirectional maximum acceleration for robot i.
p_i	The current position of robot or object i
g_i	The target robot i
v_i	The current velocity of robot i.
tangent points	Points on a circle where the constructed tangent lines touch the circle.

Table 4.1: Some terminology used during the explanation of algorithms

Algorithm 1 Top level flow of algorithms

```
while time≤ simulation time do
    Calculate and store velocities of all agents;
    for all agents do
        for all other agents do
            Calculate and store velocity obstacle;
        end for
        Calculated preferred velocity;
        for All possible velocities do
            if Velocity is not in any velocity obstacle then
                Select this velocity;
                break the For loop;
            end if
        end for
        if Suitable velocity was not found then
            Do conflict resolution;
        end if
        Calculate new position;
    end for
end while
```

Algorithm 2 `findPrefv` - Calculates the reference velocity $v_i\star$ for robot i

```
$v_i \leftarrow g_i - p_i$;
$v_i\star \leftarrow v_i * \frac{v_{max,i}}{|v_i|}$;
$s \leftarrow \frac{1}{2} * a_{max,i} * t^2$;
$d \leftarrow |g_i - p_i|$;
if $d < s$ then
    $v_i\star \leftarrow v_i \star * \frac{d}{s}$;
end if
return $v_i\star$;
```

4.1.2 `calcVOs`

This function calculates velocity of all agents $j \neq i$ on robot i. In order to do so it calls a function `findTangents` to find the collision cone.

The workflow looks is as illustrated in Algorithm 3. .

`findTangents`

This function calculates the tangent points tp_1 and tp_2, defined here as the point at which tangent lines from the robot i at position p_i touch the circle around object j at position p_j which has a radius $r = r_i + r_j$.

The code has the flow illustrated in Algorithm 4. The various angles and values are illustrated in Figure 4.2.

The **if** statement in Algorithm 4 makes sure that $\frac{r}{d} \leq 1$ for the sin^{-1}-function. If $\frac{r}{d} > 1$ the function would put out imaginary values.

This situation only occurs when robot i is too close to another agent j. In the `C++` implementation a check for this error has been implemented.

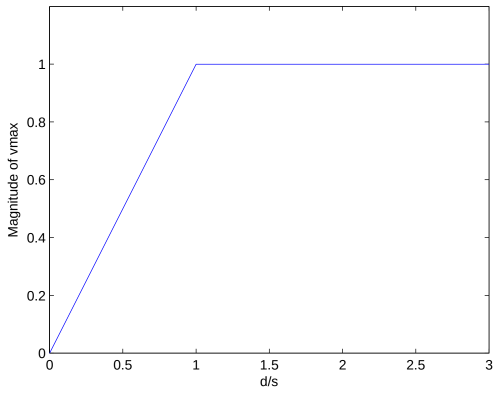


Figure 4.1: The magnitude of the preferred velocity

Algorithm 3 `calcVOs` - Calculates a velocity obstacle for agent $j \neq i$ on robot i

```
for all $j < n \quad \& \quad j \neq i$ do
    $r = r_i + r_j$;
    $[]tp_1, tp_2] \leftarrow$ findTangents;
    $CC \leftarrow [p_i, tp_1, tp_2]$;
    $VO = CC \oplus v_j$;
    store $VO$ in $VOs$;
end for
return $VOs$
```

4.1.3 Avoidance Strategy

A strategy that tries to pick an avoidance velocity v_a, i for robot i that will not result in collision using the workflow presented in Algorithm 5.

If the algorithm does not find a suitable avoidance velocity it slows down (because the for loop in Algorithm 5 ends with $\theta = \pi$).

This is a simple but effective example of an avoidance strategy. Other strategies can be implemented to achieve a certain behaviour, i.e. a strategy that picks velocities far away from velocity obstacles to be more safe.

Note that the algorithm chooses a velocity that is only $a_{max} * t_{step}$ larger then the current velocity v_i. The magnitude of the velocity v_i will therefore be similar in shape to Figure 4.3, which depicts the typical (theoretical) shape of the maximum velocity versus time. At time $t = 1$ a new target is given, to which the algorithm respons by accelerating in its direction linearly, untill v_{max} is reached at $t = 2$. At $t = 3$ the robot begins nearing the target and so slows down linearly (as mentioned in Section 4.1.1).

4.2 Additions for actual implementation

The actual implementation was done using `C++` because it offers higher performance and better integration with ROS then `Matlab`.

In addition to the algorithms above the following items were implemented to achieve a real life working example.

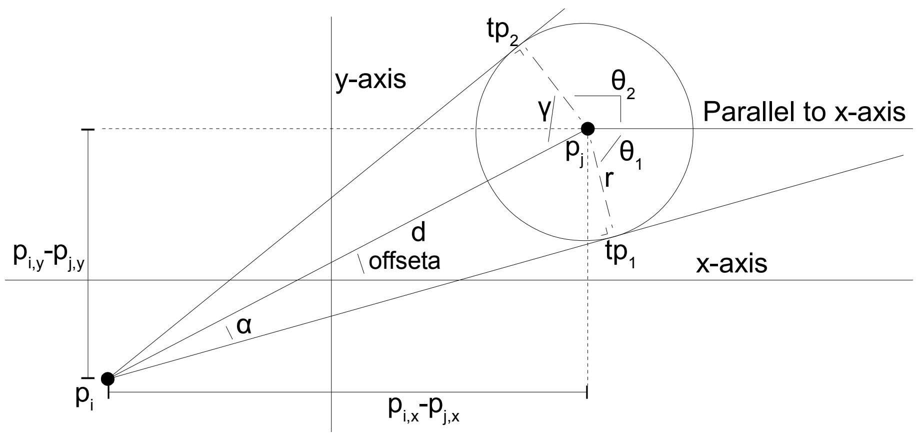


Figure 4.2: Naming of angles to find tangent points

Algorithm 4 `findTangents` - Calculates the tangent points for given p_i, p_j and r

```
$d \leftarrow |p_i - p_j|$;
if $r > d$ then
    $r = d$;
end if
$\alpha \leftarrow sin^{-1}(\frac{r}{d})$;
$\gamma \leftarrow \frac{1}{2}\pi - \alpha$;
$offseta \leftarrow atan2(\frac{y_a - y_c}{x_a - c_a})$;
$\theta 1 \leftarrow offseta - \gamma$;
$\theta 2 \leftarrow offseta + \gamma$;
$tp_1 \leftarrow c + \angle(r, \theta_1)$
$tp_2 \leftarrow c + \angle(r, \theta_2)$
```

4.2.1 Master divisor

The master divisor is a, relatively simple, node that collects information on all objects and sends it to all robots.

In this way all robots have the possibility to calculate velocity obstacles of all objects.

4.2.2 Extension to 3D

To make the implementation more generic support for 3D was implemented.

Though the principle stays the same, geometric properties change: collision triangles and circles from are cones and spheres in 3D.

To calculate where the tangent points tp_1 and tp_2 are in 3D an addition was needed to Algorithm 4. The addition works by rotating the structure of robot i and agent j so that it lies in the x-y plane, as depicted in Algorithm 6. Afterwards its rotates the structure back to find the tangent points absolute position.

The avoidance strategy was not changed to make evasive movements in the z direction, and thus will only evade by going sideways.

4.2.3 Compensating for controller/quadcopter

During preliminary testing it became clear that the assumption of a perfect controller (Figure 3.3) lead to a very slow robot.

Algorithm 5 Avoidance Strategy

```
$dv_i \leftarrow v_i\star - v_i$;
$ndv_i \leftarrow dv * \frac{amax_i * t_{step}}{dv_i}$;
**for** $\theta = 0...\pi$ **do**
    $mdv_i \leftarrow ndv_i$ rotated by $\theta$ radians;
    **if** $(mdv_i + v_i) \notin VOs$ **then**
        save $mdv_i$;
        break;
    **else**
        $mdv_i \leftarrow ndv_i$ rotated by $-\theta$ radians;
        **if** $(mdv_i + v_i) \notin VOs$ **then**
            save $mdv_i$;
            break;
        **end if**
    **end if**
**end for**
**if** $|mdv_i + v_i| > v_{max,i}$ **then**
    $v_{temp} \leftarrow mdv_i + v_i$;
    $v_{temp} \leftarrow v_{temp} * \frac{vmax}{|v_{temp}}$;
    $mdv_i \leftarrow v_{temp} - v_i$;
**end if**
```

Reason for this is that for a typical time interval (T_{step}) the robot is expected to accelerate with a finite amount $mathita_{max,i}$. The next setpoint that will be given to the controller is then no further then $T_{step} * a_{max,i}$ away.

If this value is small, it will in general result in small controller action.

When not close to the target the algorithm implements a reference gain to make the position controller put out a more aggressive command. The right value for this however depends on the use case, but moreover on the controller and quadcopter used. It is therefore a platform specific value, something this assignment has tried to avoid.

One might also increase the proportional gain of the position controller to achieve the same result, but at the risk of encountering an unstable controller.

4.2.4 Using velocity feedback

The reference gain described in Section 4.2.3 causes an overshoot for setpoint behaviour: The robot decelerates when near the target; the distance depends on v_{max} and a_{max}. The reference

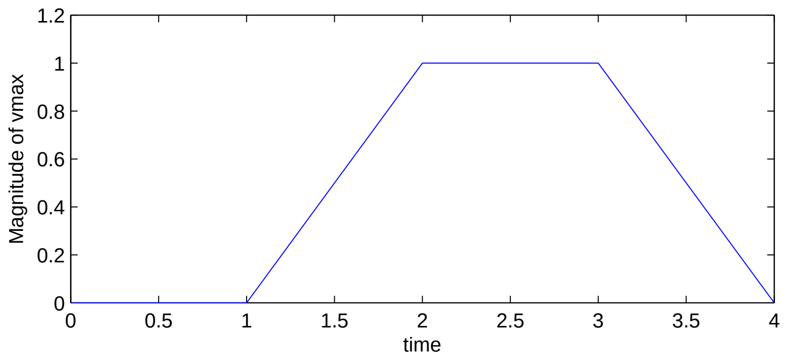


Figure 4.3: Typical shape of the magnitude of velocity versus time

Algorithm 6 Addition to extend `findTangents` to 3D

```
$da \leftarrow p_j - p_i$;                                                        ▷ Vector between objects
$r_x \leftarrow \sqrt{da_z{}^2 + da_y{}^2}$;                                      ▷ Projected distance in xy-plane
$tx \leftarrow tan2^{-1}(da_z, da_y)$;                                            ▷ Angle of $p_i$ with the xy-plane
$p_i \leftarrow$ rotate $p_i$ around the $x$-axis with $tx$ radians;              ▷ Rotation
$[tp_1, tp_2] \leftarrow$ findTangents                                            ▷ Original function
$[tp_1, tp_2] \leftarrow$ rotate $[tp_1, tp_2]$ around the $x$-axis by $-tx$ radians;   ▷ Rotate back
```

gain enlarges the velocity, but not the acceleration, making the robot change directions too slow.

A solution specific for this assignment's demo was found by decreasing the reference gain as speed increases. The reference gain has a maximum and minimum so it remains within safe boundaries; replacing the boundaries can result in unstable behaviour.

4.2.5 Hard switch when near target

To improve safety a 'hard switch' was implemented for when a robot is too close to an object.

When such a case is detected the algorithm puts out the maximum velocity away from the object, after which it resumes the normal flow.

5 Results

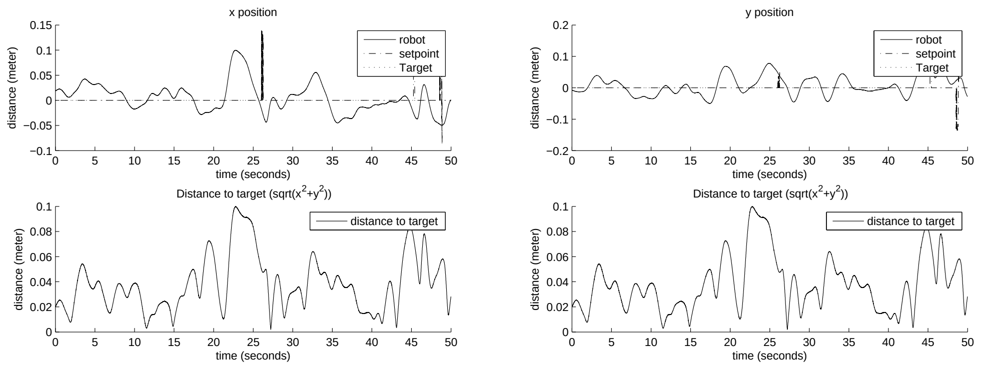
Plots in this chapter were made using `Matlab`'s ROS IO package and matlab_rosbag (Charrow, 2014).

5.1 No other agents

This section gives results of tests without any obstacles in Figure 5.1.

The behaviour in Figure 5.1a and Figure 5.1b is the response to static target with no other agents in the system. Fluctuations are caused by airflow and enlarged by the reference gain when out of range of the target.

The path plotted in Figure 5.2 was achieved when giving a target following a circular path in the xy plane.



(a) Hovering behaviour in x **(b)** Hovering behaviour in y

Figure 5.1: Static target behaviour without any obstacles

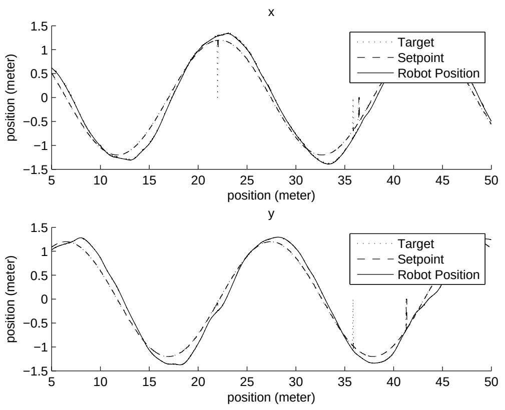
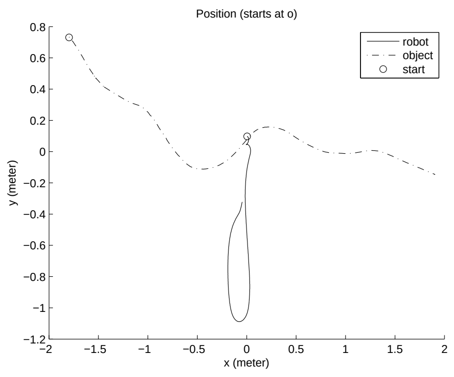


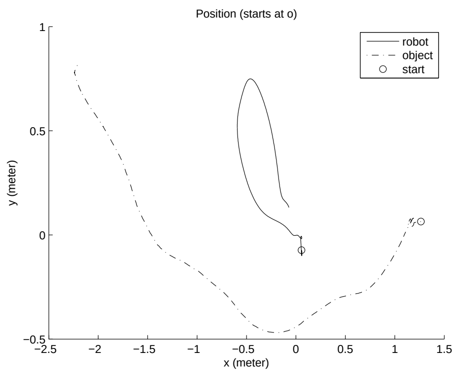
Figure 5.2: Tracking a moving target without obstacles

5.2 Avoiding on a single position

A single robot with target (0,0,1) was set to evade an object in the hand or on the hat of the user. The results are shown in Figure 5.3.



(a) first run

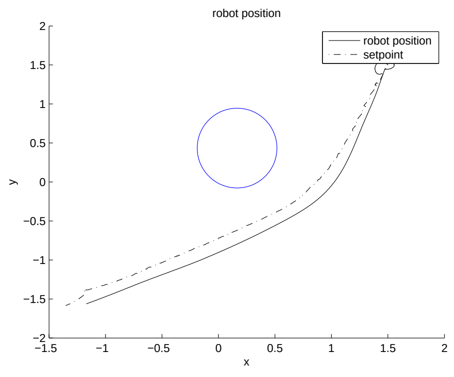


(b) second run

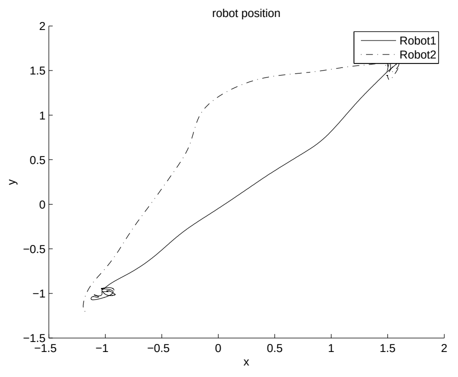
Figure 5.3: Test results for avoiding an object whilst having a non moving target

5.3 Avoiding while moving

Two different tests were done to test the behaviour when moving towards a target. Figure 5.4a shows the results for one robot evading an obstacle that remains in one position and Figure 5.4b shows the path of two robots that switch positions, both in meters.



(a) Path for a robot evading a static obstacle



(b) Paths for two robots with opposing targets

6 Conclusion and Discussion

This assignment has aimed to create a modular method to avoid obstacles that is not limited to UAV's.

A velocity obstacle based algorithm using ROS has been implemented for the Parrot AR.Drone, for two drones coexisting and for one drone avoiding a static or moving obstacle.

The results presented with the Parrot AR.Drone prove a working infrastructure. The infrastructure as-is however has a few shortcomings that are elaborated below.

6.1 Necessity of reference gain

For the Parrot AR.Drone with the controle made by Cees Trouwborst the setpoints put out were too close to the robots current position; making the controller+robot react slowly.

This was counteracted by implementing a reference gain. The most suitable magnitude of the reference gain will differ for systems with a different response, also determined by the controller's proportional gain. Setting the reference gain for optimal performance therefore is platform dependent.

Even though the structure of the algorithm is independent of the controller+robot combination it is run for, it does need an estimate of the system's performance, beyond the desired maximum acceleration and maximum velocity.

6.2 Avoidance Strategy

A simple algorithm for selecting a non-colliding velocity has been implemented in this assignment, that only works in the xy-plane. To make full use of the robot's capabilities also the vertical direction should be used.

Furthermore the current avoidance strategy does not implement a heuristic as to which velocities are most suitable: it picks a velocity that is as closely aimed at the target as possible. As a result the velocity is always close to a velocity obstacle (if there is one between the robot and target).

Using more information on the system a velocity can be chosen that is more likely to be safe, allow for higher speed, shorter path or less energy consumption.

In an ideal case a user will be able to select different strategies to alter the behaviour of the robot in flight.

7 Future Work

This chapter describes some work that could be done in te future.

7.1 Specific for this work

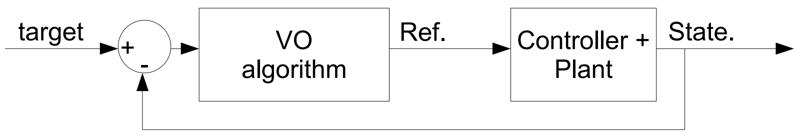
This section elaborates on the work that could be done to this assignment to significantly improve its results.

7.1.1 In operation characterization

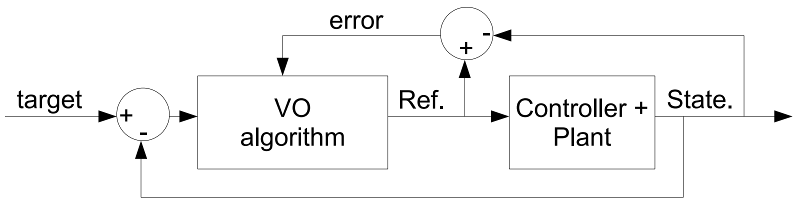
The need to implement a platform dependent reference gain makes this algorithm not as modular as necessary. However the algorithm has every piece of information required to perform characterization of the system it is controlling during flight.

The two elements of this information are the reference the velocity obstacle puts out and the state the controller and robot respond with, as can be seen in Figure 7.1b.

If the error between state and reference is characterized the velocity obstacle algorithm can predict what it has to put out to make the robot reach the desired state. In this way the algorithm is platform independent again.



(a) A simple overview of the system as is



(b) System expanded with character information

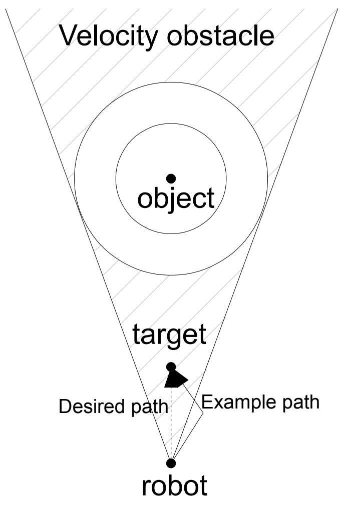
7.2 Implementation of a time horizon

The velocity obstacle method creates cones that are collections of speeds that are not allowed for the robot.

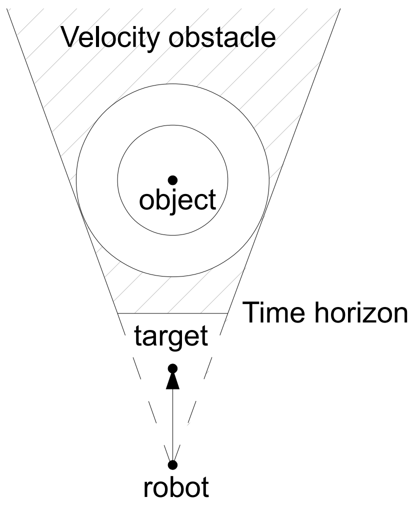
There are situations in which this causes problems. A simple example is drawn in Figure 7.2a, where the robot tries to achieve a target that is in between the robot and an object.

Because the target is in a velocity obstacle the robot will not select a velocity directly towards it even though one can see directly from this image that there is no risk of collision. The robot will take an alternative path labeled 'example path' in Figure 7.2a that remains outside the velocity obstacle.

A solution to this problem is to set a time horizon: a limit as to which velocities are allowed inside a velocity obstacle. The size of this limit could be related to the estimated time to collision at this velocity, or the distance to the object causing the velocity obstacle.



(a) Without time horizon



(b) With time horizon

Figure 7.2: A situation where a time horizon offers solution

7.2.1 A more suitable avoidance strategy

The avoidance strategy used in this assignment has its limits in use, but moreover in theoretical excellence.

The main cause for this is that it searches checks for a finite amount of velocities V whether they are in velocity obstacle, as can be seen in Algorithm 7.

The set V that is checked in the current algorithm is a small part of all achievable velocities $V_{achievable}$. It is however possible to calculate the all the avoiding velocities $V_{avoid}$by solving for the velocities that are in a velocity obstacle: $V_{avoid} = V_{achievable} - V_{inObstacle}$.

Then all velocities in the set V_{avoid} may be weighted to a certain heuristic that depends on desired behaviour to select the most optimal one, to ensure selection of the best velocity at all times.

The heuristic may depend on a range of different factors or a combination thereof such as but not limited to:

- vicinity to a velocity obstacle
- global direction
- local direction (curvature)
- speed (or another position derivative)

Algorithm 7 Simple representation of current avoidance strategy

```
for do $v_i \in V$
    if then $v_i \notin VOs$
        Use this velocity;
        break;
    end if
end for
```

7.3 Vision

The work done in this assignment aims to enable large groups of robots to co-exist. In such an environment each robot has its own control and velocity obstacle logic to avoid collisions with the environment and other robots. The distributed fashion of the system allows for exceedingly large groups of robots.

The level of autonomy that every robot in such a system has allows the user to either change targets for all robots as a group or for a single robot during flight. Because of the no-crash guarantee by all robots in such a system there is no need for additional or central intelligence.

This way robots can truly coexist with humans, objects and other robots.

8 Tutorial

This chapter explains how to run experiments with the velocity obstacle algorithm running.

You can skip ahead to Section 8.5.2 if you have already followed the similar tutorial in the ram_ba_package documentation.

8.1 Requirements

You need the following software on your system:

- ROS compatible operating system (such as Ubuntu 12.x or 14.x)
  get at: (Ubuntu, 2012)
- ROS (we have tested using Hydro, get at (Foundation, 2013))
- ram_ba_package (Trouwborst, 2014) and its dependencies which are amongst others:
  - ardrone_autonomy (Monajjemi, 2012)
  - mocap_optitrack (Gräve, 2010)
- ram_vo package (get at dedicated RAM-person)
- motive software (NaturalPoint, 2011)

Your system needs to consist of at least:

- Computer (to run ROS, and to run OptiTrack)
- OptiTrack camera's connected to a PC
- Wireless access point connected to the computer
- Parrot AR.Drone (labelled with infra red reflecting buds)

Different set ups are possible as well, at the SmartXP lab for instance, the OptiTrack software is running on a different PC then ROS. In that case also communication is required.

8.2 Setting up your gear

This section will explain how to set up the system until take-off.

8.2.1 Sending position feedback

Start the motive software and make sure all camera's are running, they should all show a different number indicating that they are connected.

Select '*perform camera calibration*' in the screen that pops up.

- Remove all obstacles from the flying area
- Put up a net and protective screens if you need them
- in the Motive software, find the camera calibration section, and click *block visible*
- Click *start wanding*
- Pick up the wand and wave it slowly around the flying area
- Stop waving when the calibration screen says *quality: very high*

Moving the wand around the flying area should result in the Motive software plotting coloured paths in ever camera's specific screen. If you do not see these coloured paths wave the wand slower.

After wanding click *calculate*. The Motive software starts optimizing its calibration. The longer you wait the better it gets. You can wait until the software tells you it is sufficient for quality *exceptional* or until the software tells you it is *ready to apply*. Then click *apply*.

Afterwards you can set the ground plane using the NaturalPoint groundplane-indicator. Place the indicator inside the flight arena at the position you want to be the origin. For use in the smart XP point its z-axis due south. Click *set ground plane*.

Remove the groundplane-indicator and place the objects you want to be tracked inside the flying arena. Point quadcopters with their front camera due east, in the SmartXP that is towards the big presentation screen.

Use the Motive software to drag a square around the objects you should now see in the screen. Select all the trackers on one object, and *right-click* to *create rigid body from selection*. Repeat this for all your objects/quadcopters.

For now you should remember the order you created the trackables: you will need it later.

Now using the menu bar open *view* > *data streaming*. Check broadcast frame data scroll down to find the multicast settings. Set it up to cast to the computer that runs ROS by:

- key in its IP adress
- switch interface: preffered to interface: 130.89.xx.xx for UTwente

You are done with the Motive software for now.

8.3 Recieving position feedback

You will need to launch a mocap_optitrack instance to recieve the position information and publish it in ROS. You can do that yourself, but the ram_ba_package will do it for you fast and easy.

- Open a terminal and start a roscore by keying in `roscore`
- Open a new terminal and type `rosrun ram interface.py`

The connection interface opens.

Turn on your drones and wait for your WiFi module to see al the SSID's they broadcast. Then hit *scan* in the connection interface.

You can now label the drones by giving them a name. This can be anything you want, but it makes sense to keep this short and simple, without spaces.

The tricky part is now: the trackable ID is the number you remembered when setting rigid bodies in Motive. Match the SSID of the robot you are configuring to the trackable ID and hit *OK*. For this tutorial we assume you named two drones: Lindsey and Matteo.

When you are done all drones have a name and a trackable ID. You can now forget the numbers, you need only the names.

You can confirm you are getting the right position info by opening a new terminal and entering `rostopic echo /Lindsey/unfiltered_pose`. This is the topic at which robot Lindsey's position comes in. Move the quadcopter around an look at how the position has changed to confirm that everything works OK.

Use the *assign* button to connect to the drones and make them connect to the access point. By default the access point should have an SSID named RAM_drones.

You can now check if there is a connection to the drones by opening a terminal and keying in `ping 192.168.2.9` (or any other IP the drone has been given).

8.4 First flight test

Double click on a drone's name in the connection interface. Accept the warning message and a controller window should open.

Move the sliders to get a nice setpoint to start, i.e. (0,0,1). *Check Publish Setpoint*, and hit *Take off*.

The drone takes off. play around a bit with the controller interface and hit *land* when you are done.

8.5 Starting the velocity obstacle algorithm

8.5.1 Simulation with Gazebo

The simulation is set up using launch file, it uses a premade package called tum_simulator (Huang, 2014). The package has the name of the quadcopter 'ardrone' hardcoded in it. The ram_vo package has a few remappers (simple nodes that transfer messages) to get your simulation going, but the tum_simulator will not work for more then one robot.

The launchfile you are looking for is called by running `roslaunch ram_vo single_simulation.launch`
A few windows will open:

- Gazebo, a physics simulator which also shows you the simulation
- gdb window with a controller
- gdb window with a VO_controller
- gdb window with a target_from_keyboard node

The simulation will have an empty world with just one quadcopter and one sphere in it. You can run the controller by typing `run` in its terminal.

Plug in a joystick. Use its buttons to send a take off command. The robot on screen takes off. If you want you can use the joystick to fly around. If you let go of the joystick, the controller will take over and take the robot back.

Run the VO_controller by typing `run` in its terminal.

Experiment with putting the robot on the other side of the sphere using the joystick, and then letting the velocity obstacle algorithm do the rest.

Note: there is a lot of remapping going on here. Use `rosrun rqt_graph rqt_graph` to get an overview. Trying to get this from source is difficult.

8.5.2 Real flying

There are two launch files in the ram_vo pakcage you are interested in at this moment: `double_testing.launch` and `demo_petje.launch`. You can use them to set up an experiment, but since they perform remapping topic names, you have to change them to fit your topic names or set your names to:

- first robot: Lindsey, trackable ID: 1
- second robot: Matteo, trackable ID: 2

- object you can carry: Petje, trackable ID: 3

For a test run with two robots use `double_testing.launch`, for a run with one robot (Lindsey, ID: 1) and one object (Petje, ID: 3) use `demo_petje.launch`.

Open a new terminal and key in `roslaunch ram_vo demo_petje.launch` (replace with other launchfile if you like).

A series of windows should open:

- RVIZ - realtime visualization of your experiment
- gdb terminals:
  - One or more velocity controllers (VO_controller)
  - Master_divisor
  - target_from_keyboard

Not much is happening yet, you have to start some executables using their gdb terminals.

Start the master_divisor by typing `run` in its terminal. It provides you with a list of the names of all robots it has spotted. It will now send all position and velocity info to all robots.

Start the target_from_keyboard node by typing `run` in its terminal. it will ask you for an x-coordinate for the new target. If you like you can key in a value and hit enter. The terminal will then ask for a y, and later for a z coordinate. After setting a z coordinate the target will be published. right now you can try this as much as you like, because there is no-one listening to the targets (yet).

Stop publishing silly targets. Start the VO_controller by typing `run` in its terminal. In the terminal you can now read the name of the robot it is running for, and some information on that robot. After waiting a while (default 2 seconds) so you can read what is said there, the node starts the actual work: recieving targets, recieving information and publishing setpoints.

During startup of the velocity obstacle controller the target will be at (0,0). So do not start two velocity obstacle controllers at the same time, because the robots will try to go to the same position.

You can either start the VO_controllers while the quadcopters are in the air running only the position controller, or while they are on the ground and hit takeoff on the position controller afterwards. It is easier to see what you are doing if they are flying first.

8.5.3 See what is happening

When a VO_controller is running you can go to the RVIZ window that the launchfile has opened. It will show you a few things:

- A green sphere: the robot as interpreted by the algorithm
- A red sphere: the object as interpreted by the algorithm
- A red semi-transparent hull around the object: The full size of the no-fly zone of the object
- A yellow dot: the current target for the robot
- A blue line: The curren velocity pointing from the robot to the target; its magnitude is of no importance
- A purple dot: The requested position, which is most of the time the setpoint put out to the position controller

If either the requested position or the robot is close to the target, and the target itself is not in a velocity obstacle, the algorithm will put out the target as a setpoint, rather then the requested position. You can not see this in RVIZ.

8.6 Providing targets

You can provide targets for the robots at */theirname/target*-topics. The package ram_vo provides ways to do that easily as well.

target_from_keyboard - allows you to seperately key in values for x, y and z, start using a launchfile.

double_setpoint_pub.m - Graphical user interface to select a setpoint out of 25 presets to publish (simultaneously if you want) to a maximum of two robots. Run in `Matlab`.

circle_s.m - Publish multiple targets following a circular path. Run in `Matlab`.

switch_positions.m - Publish two alternating targets. Run in `Matlab`.

8.7 Adjusting behaviour

Several parameters inside the sourcecode alter behaviour of the algorithm. They are listed here.

- amax - Maximum acceleration you want the robot to have. This has a large influence on how for the algorithm looks ahead. Feasible values lie between 5 and 20.
- radius - The radius of a robot/object. Set this larger then the actual radius to allow for a safety band. Feasible values range from 0.4 to 0.6.
- vmax - Maximum speed the robot is allowed to travel at. In practice this is only reached when covering distances in the order of meters. Feasible values range from 0.5 to 1.5.
- TSTEP - Size of a timestep, depends largely how for the algorithm looks ahead. Increasing this will also increase the speed of the robot, but this will be less safe (as situations may change within the timestep). Feasible values is 0.05. Use vmax and amax if you want to tweak.
- max_ref_gain - The maximum value for the reference gain. Refer to Section 4.2.3 for an explanation. Feasible values range from 5 to 18.

8.8 Deeper understanding

You can get a deeper understanding of what is happening by doing more application specific tutorials on ROS and on the software by Cees, as well as by looking at the algorithms in Chapter 4.

When you have completed these tutorials you should be able to understand what is happening if you open the ram_vo source code directly. The code is heavily commented so you will not be left in the dark. Furthermore documentation is available at the dedicated RAM person.

If you want a better view of what is happening you can open a new terminal and run `rosrun rqt_graph rqt_graph`. A new window will open that shows all topics and nodes that are currently used.

If problems or unclarities arise please contact me at kamphuis.alex@gmail.com, enjoy your flight!

Bibliography

Charles Richter, Adam Bry, N. R. (2010), Polynomial trajectory planning for quadrotor flight.

Charrow, B. (2014), matlab_rosbag.
https://github.com/bcharrow/matlab_rosbag

Daniel Mellinger, V. K. (2011), Minimum snap Trajectory Generation and Control for Quadrotors, *IEEE International Conference on Robotics and Automation.*

Foundation, O. S. R. (2013), ROS Hydro Medusa.
http://wiki.ros.org/hydro

Gräve, K. (2010), mocap_optitrack.
wiki.ros.org/mocap_optitrack

Huang, H. (2014), tum_simulator.
wiki.ros.org/tum_simulator

Markus Hehn, R. D. (2011), Quadrocopter Trajectory Generation and Control.

MathWorks (2013).

Monajjemi, M. (2012).

NaturalPoint (2011).

Parrot (2012), AR.Drone2.
http://ardrone2.parrot.com/

Trouwborst, C. (2014), RAM ba package.
https://github.com/ceesietopc

Ubuntu (2012), Ubuntu 12.04 Precise Pangolin.
http://www.ubuntu.com/

Source code is available for RAM members at the ce wiki.