

# Obstacle Avoidance with a Multicopter in a Dynamic Two-Dimensional Environment

by

Jacobus Stephanus Coetzee



*Thesis presented in partial fulfilment of the requirements for  
the degree of Master of Engineering (Mechatronic) in the  
Faculty of Engineering at Stellenbosch University*

Supervisor: Dr. WJ. Smit

March 2017

# Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: ..... March 2017 .....

Copyright © 2017 Stellenbosch University  
All rights reserved.

# Abstract

## Obstacle Avoidance with a Multicopter in a Dynamic Two-Dimensional Environment

JS. Coetzee

*Department of Mechanical and Mechatronic Engineering,  
University of Stellenbosch,  
Private Bag X1, Matieland 7602, South Africa.*

Thesis: MEng (Mechatronic)

March 2017

In order to allow a multicopter to fly autonomously in a dynamic two-dimensional environment, a prototype obstacle avoidance system was developed. Included in this prototype avoidance system was the development of a two-dimensional proximity sensor. A combination of obstacle avoidance algorithms (D\* Lite and the VFF) were coded in MATLAB and verified through various simulations. Pose and measurement uncertainties were also investigated and incorporated into the simulations' map building technique. In the end, the obstacle avoidance algorithm was successfully implemented in various outdoor test environments, managing to safely navigate the multicopter to its end destination in each case.

# Uittreksel

## Hindernis Vermyding met 'n Hommeltuig in 'n Dinamiese Twee-Dimensionele Omgewing

*("Obstacle Avoidance with a Multicopter in a Dynamic Two-Dimensional Environment")*

JS. Coetzee

*Departement Meganiese en Megatroniese Ingenieurswese,  
Universiteit van Stellenbosch,  
Privaatsak X1, Matieland 7602, Suid Afrika.*

Tesis: MIng (Megatronies)

Maart 2017

Ten einde hindernis vermyding met 'n hommeltuig toe te pas in 'n twee-dimensionele vlak, is 'n prototipe hindernis vermydings sisteem ontwerp. Ingesluit by die prototipe vermydings sisteem was die ontwerp van 'n twee-dimensionele afstand sensor. 'n Kombinasie van hindernis-vermydings algoritmes (D\* Lite en die VFF) was geprogrammeer en geverifieer in MATLAB deur verskeie simulaties te doen. Posisie asook meting onsekerheid was ondersoek en in ag geneem in die simulatie se kaart bou tegniek. Daarna was die hindernis-vermyding sisteem suksesvol geimplimenteer in verskeie buitemuurse toets omstandighede waar dit altyd die hommeltuig veilig by sy eind bestemming kon uitbring.

# Acknowledgements

I would like to express my sincere gratitude to the following people and organisations:

- My supervisor, Dr. W.J. Smit for his guidance and support with regards to my research.
- The NRF for granting me a bursary.
- My family and friends for their ongoing support.

# Contents

<b>Declaration</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>Uittreksel</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>viii</b>
<b>List of Tables</b>	<b>x</b>
<b>Nomenclature</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Objectives . . . . .	3
1.2 Scope . . . . .	3
1.3 Overview of Project Layout . . . . .	4
<b>2 Literature Review</b>	<b>5</b>
2.1 Map Building Techniques . . . . .	5
2.1.1 Uniform Occupancy Grids . . . . .	6
2.1.2 Adaptive Occupancy Grids - Quadrees . . . . .	8
2.2 Existing Obstacle Avoidance Algorithms . . . . .	9
2.2.1 Virtual Force Field . . . . .	10
2.2.2 A* . . . . .	11
2.2.3 Lifelong Planning A* . . . . .	12
2.2.4 D* Lite . . . . .	13
2.3 Conclusion . . . . .	15
<b>3 Measurement and Pose Uncertainty</b>	<b>16</b>
3.1 Overview: Measurement Uncertainty . . . . .	16
3.2 Assumptions: Measurement Uncertainty . . . . .	16

3.3	Inverse Sensor Model Derivation . . . . .	17
3.3.1	Ideal Inverse Sensor Model . . . . .	17
3.3.2	Inverse Sensor Model with Gaussian Noise . . . . .	18
3.4	Overview: Pose Uncertainty . . . . .	21
3.5	Assumptions: Pose Uncertainty . . . . .	22
3.6	Pose Uncertainty Derivation . . . . .	22
3.7	Simulation . . . . .	24
3.7.1	Setup . . . . .	24
3.7.2	Results . . . . .	25
3.8	Conclusion . . . . .	28
<b>4</b>	<b>Hardware and Software Integration</b>	<b>30</b>
4.1	Hardware Integration . . . . .	30
4.1.1	Pixhawk Flight Controller . . . . .	30
4.1.2	Intel Edison . . . . .	31
4.1.3	Proximity Sensor Design . . . . .	33
4.2	Software Integration . . . . .	34
4.2.1	Robot Operating System . . . . .	35
4.2.2	MATLAB . . . . .	36
4.2.3	Python . . . . .	36
4.2.4	Proximity Sensor Design . . . . .	37
4.2.5	Hardware in the Loop . . . . .	38
4.3	Conclusion . . . . .	39
<b>5</b>	<b>Proximity Sensor</b>	<b>41</b>
5.1	Test Results . . . . .	41
5.2	Conclusion . . . . .	46
<b>6</b>	<b>Obstacle Avoidance Implementation</b>	<b>47</b>
6.1	Algorithm Integration . . . . .	47
6.2	MATLAB Simulation Setup . . . . .	48
6.2.1	D* Lite . . . . .	48
6.2.2	Virtual Force Field . . . . .	50
6.3	MATLAB Simulation Results . . . . .	52
6.4	Conclusion . . . . .	54
<b>7</b>	<b>Flight Tests</b>	<b>57</b>
7.1	Assumptions . . . . .	58
7.2	Outdoor Test Setup . . . . .	58
7.3	Flight Test Results . . . . .	60
7.4	Conclusion . . . . .	69
<b>8</b>	<b>Conclusion and Recommendations</b>	<b>71</b>
8.1	Conclusion . . . . .	71

<i>CONTENTS</i>	<b>vii</b>
8.2 Recommendations . . . . .	72
<b>List of References</b>	<b>74</b>
<b>Appendices</b>	<b>76</b>
<b>A Pseudo Code For Obstacle Avoidance Algorithms</b>	<b>77</b>
A.1 A* . . . . .	77
A.2 LPA* . . . . .	78
A.3 D* Lite . . . . .	80
<b>B Datasheets</b>	<b>82</b>
B.1 Pixhawk Flight Controller . . . . .	82
B.2 Intel Edison . . . . .	85
B.3 Arduino Mega . . . . .	88
B.4 PulsedLight Lidar . . . . .	94
B.5 Pickup Sensor . . . . .	100
B.6 Piksi RTK . . . . .	107



# List of Figures

1.1	Ivanpah solar power plant in California . . . . .	2
2.1	Quadtree map building example . . . . .	8
2.2	Graph representation . . . . .	9
2.3	Virtual Force Field vector diagram . . . . .	10
2.4	Virtual Force Field gradient diagram . . . . .	11
3.1	Ideal inverse sensor model . . . . .	18
3.2	Probability density function (PDF) . . . . .	19
3.3	Gaussian inverse sensor model . . . . .	21
3.4	Incorporating measurement uncertainty into the map building process	21
3.5	Cumulative distribution function (CDF) . . . . .	23
3.6	Incorporating pose uncertainty into the map building process . . . .	24
3.7	CDF position approximation . . . . .	27
3.8	Simulated map outputs . . . . .	27
3.9	Sensor model comparison . . . . .	28
4.1	Overview of hardware and software integration . . . . .	31
4.2	Pixhawk flight controller . . . . .	32
4.3	Intel Edison breakout board with Arduino expansion board . . . . .	32
4.4	Arduino Mega . . . . .	33
4.5	Pulse width modulation (PWM) wiring setup for LIDAR . . . . .	34
4.6	Flow diagram of the proximity sensor code . . . . .	39
4.7	jMAVSim hardware in the loop (HITL) interface . . . . .	40
5.1	Sensor Setup . . . . .	42
5.2	Test results: First scanned environment . . . . .	43
5.3	Test results: Second scanned environment . . . . .	44
5.4	Test results: Third scanned environment . . . . .	45
6.1	Cost and heuristic function illustration . . . . .	50
6.2	Combined avoidance algorithm flow chart . . . . .	51
6.3	MATLAB simulations of static and dynamic environments . . . . .	53
6.4	D* Lite MATLAB simulation of cluttered static environment . . . .	54
6.5	D* Lite execution time . . . . .	55

*LIST OF FIGURES***ix**

6.6	D* Lite execution time using different obstacle configurations . . . .	56
7.1	Multicopter test setup . . . . .	59
7.2	Flight test obstacle placement . . . . .	59
7.3	Test results: First flight . . . . .	63
7.4	Test results: Second flight . . . . .	64
7.5	Test results: Third flight . . . . .	65
7.6	Test results: Fourth flight . . . . .	66
7.7	Test results: Fifth flight . . . . .	68

# List of Tables

3.1	Correspondences between probabilities and log odds ratios . . . . .	18
3.2	PDF distribution data obtained from Pixhawk position estimation error . . . . .	26
4.1	ROS LaserScan Message type: Message data and definitions . . . . .	38
6.1	Octile distance heuristic function inequality test . . . . .	49
7.1	Summary of flight test results . . . . .	62
7.2	PDF distribution data obtained from Pixhawk position estimation error during flight tests . . . . .	62

# Nomenclature

## Variables

$c^*(s, s')$	Cost of the shortest path
$d$	Distance
$E$	Error
$F$	Force
$g(s)$	Exact cost
$h(s, s')$	Heuristic cost
$k_m$	Key modifier
$k(s)$	Key
$L$	Length
$m$	Matrix
$M$	Number of measurements
$r$	Distance between cells
$rhs(s)$	Exact cost
$s$	Vertex
$S$	Graph
$x$	Pose estimate
$z$	Distance measurement
$\lambda$	Log-odds map update value
$\mu$	Average error
$\sigma$	Standard deviation

## Subscripts

$att$	Attraction
$i$	Cell / Matrix index
$j$	Cell / Matrix index
$last$	Last vertex a path was recalculated at
$m$	Modifier
$max$	Maximum

<i>rep</i>	Repulsion
<i>safe</i>	Safety distance
<i>start</i>	Current vertex a path should be calculated to
<i>t</i>	Current Time
<i>total</i>	Total
<i>x</i>	x-direction
<i>y</i>	y-direction

### Superscripts

<i>c</i>	Compliment
----------	------------

### Acronyms

A*	A-star
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
CSP	Concentrated Solar Power
DC	Direct Current
D* Lite	D-star Lite
ENU	East, North, Up
GPS	Global Positioning system
HITL	Hardware in the Loop
IMU	Inertial Measurement Unit
LIDAR	Light Detection and Ranging
LPA*	Lifelong Planning A-star
LPDDR	Low Power Double Data Rate
MATLAB	Matrix Laboratory
MAVLINK	Micro Air Vehicle Communication Protocol
MAVROS	MAVLINK extendible communication node for ROS
NED	North, East, Down
PDF	Probability Density Function
PWM	Pulse Width Modulation
STERG	Solar Thermal Energy Research Group
UART	Universal Asynchronous Receiver/Transmitter
UAV	Unmanned Aerial Vehicle
USB	Universal Serial Bus
PV	Photovoltaic
ROS	Robot Operating System

*NOMENCLATURE*

**xiii**

RTK Real Time Kinematic  
SD Secure Digital  
VFF Virtual Force Field

# Chapter 1

## Introduction

In recent years, the use of multicopters have become popular for both research and recreational activities. Currently, two active research fields are search-and-rescue and swarming (the use of multiple drones to complete a single task). Recreational activities, on the other hand, can include anything from drone racing, photography, surveying or even first person view flying. The use of these devices are increasing despite the fact that they carry inherent risk. For this reason additional safety measures are constantly being investigated; including autonomous obstacle avoidance capabilities.

Vehicles with autonomous capabilities are being researched and implemented around the world by multiple companies such as Google, Tesla, Otto, Ford and many more. Other companies like Honda are working on advancements in robotics with humanoid-like capabilities. Whereas Amazon and DHL are focusing their research on package delivery with the help of unmanned aerial vehicles (UAVs). However, for these vehicles and robots to be deemed safe while moving about autonomously, obstacle avoidance is required.

At Stellenbosch University, the Solar Thermal Energy Research Group (STERG) is currently investigating multiple different ways of reducing the costs surrounding the manual inspection and calibration of heliostats in a concentrated solar power (CSP) plant. One way of doing this is to increase routine inspections as it can have a significant impact on the maintenance costs throughout a plant's operating period, if problematic areas are identified beforehand. However, solar power plants usually cover a great area and contain a tremendous amount of equipment. One example of such a solar plant is the Topaz solar farm in California. This plant contains 9 million solar photovoltaic (PV) modules and spans over an area of 25.6 km<sup>2</sup>. Another example is the Ivanpah CSP plant in California, as seen in Figure 1.1. This plant consists of more than 300 thousand mirrors and spans over an area of 14.2 km<sup>2</sup>.

To maintain both the efficiency of the plant and the plant itself, mirrors as well as solar PV modules have to be kept clean and replaced when broken. Usually, these manual inspections require a considerable amount of time and labour. Therefore, the high complexity, size and remote locations of these

plants provide quite a few challenges namely: dangerous or difficult to access work areas, tight schedules and a large number of inspection points.



Figure 1.1: Ivanpah solar power plant in the Mojave Desert, California (Bright-Source Energy Inc, 2016). This plant uses around 173 500 software operated heliostats to follow the sun's trajectory with more than 300 000 mirrors, spanning over 14.2 km<sup>2</sup>.

Recently, multicopters flown by pilots have been used for inspection purposes on solar farms. Equipped with infrared sensors, these multicopters are flown to all the inspection points to check for damage and heat anomalies across the plant. This method, led to a decrease of up to 40 % in the plant's maintenance costs while taking less than 30 % of the time it would have taken if manual inspections were done (UAS, 2016).

If a fully automated inspection system can be implemented, it will be able to further increase the efficiency of the maintenance work, reduce the operating and maintenance costs as well as improve the safety and working conditions of the service technicians. Therefore, the proposed solution is to use autonomous UAVs, specifically multicopters, for regular inspections around CSP plants. However, for the system to be able to fly autonomously, obstacle avoidance is required. This study will therefore be aimed at the design, implementation and testing of a system that enables a multicopter to fly autonomously in a dynamic two dimensional (2D) environment.



To outline the steps followed on how the proposed solution was investigated, both the objectives and scope of the project are given in Section 1.1 and Section 1.2 respectively. The layout of this thesis then follows in Section 1.3.

## 1.1 Objectives

This study is aimed at designing, building and testing an obstacle avoidance system for a multicopter in a dynamic 2D environment. In order to reach these objectives, the following methodology was set:

- Examine literature regarding different map building and obstacle avoidance techniques.
- Select, code and simulate a suitable map building technique.
- Determine what effect pose and measurement uncertainty have on the map building technique.
- Select, code and simulate an obstacle avoidance algorithm along with the map building technique.
- Integrate the necessary hardware and software to get a working prototype of the obstacle avoidance system.
- Implement the prototype obstacle avoidance system on a multicopter in a test environment.
- Evaluate test results and give relevant feedback.

## 1.2 Scope

The scope of this project is aimed at designing, building and testing an obstacle avoidance system. This includes the proximity sensor design, algorithm development, simulations, component integration and testing. In essence, the obstacle avoidance capabilities will be a separate system that can be connected with the flight controller, enabling it to fly autonomously.

The scope of this project, however, does not include the design or modification of the flight controller itself. This means that the flight controller was used with its own flight stack i.e. no controller design was done. The controller was, however, tuned to improve flight stability and performance.

### 1.3 Overview of Project Layout

In Chapter 2, a literature review regarding the relevant path planning and map building techniques for this project is given. The effects that uncertainty in pose and proximity sensor measurements can have on the map building technique are then investigated in Chapter 3. This is followed by the hardware and software integration in Chapter 4, illustrating how a working prototype of the obstacle avoidance system can be obtained by correctly integrating the necessary components. Chapter 5 gives an overview of the proximity sensor test results obtained using the sensor setup as described in Chapter 4. The chosen combination of obstacle avoidance algorithms are then explained in Chapter 6 followed by the simulated results. Final flight test results as well as a discussion regarding these results can be found in Chapter 7. The conclusion and recommendations regarding this project are then given in Chapter 8.

# Chapter 2

## Literature Review

Obstacle avoidance can be broadly classified under two categories: global- and local path planning. Global path planning is mainly done offline (beforehand), whereas local path planning must be fast, reactive and is usually carried out online (while in motion) to ensure the safety of the vehicle by reacting to unforeseen obstacles (Mujumdar and Padhi, 2011). The key distinction between the two categories of obstacle avoidance is the amount of information available regarding the vehicle's immediate environment. If all the information regarding the obstacles is available, global path planning will be the best approach. On the other hand, if the information regarding the environment is imperfect, local information based on proximity sensor data will have to be used to navigate (Hoy *et al.*, 2015).

Many techniques for both local and global obstacle avoidance have been proposed in the recent literature (Goerzen *et al.*, 2010; Mujumdar and Padhi, 2011; Hoy *et al.*, 2015), all with varying levels of efficiency and comprehensivity. By identifying and compensating for the weaknesses within some of these techniques, it is possible to generate alternative methods for obstacle avoidance (Chapter 6).

In Section 2.1, two different map building techniques are discussed: uniform and adaptive occupancy grids. This is followed by an investigation regarding different path planning techniques in Section 2.2. These techniques include the Virtual Force Field (VFF) method, A\*, LPA\* and D\* Lite.

### 2.1 Map Building Techniques

In this section, two occupancy grid methods are investigated. Uniform grids are introduced in Section 2.1.1 whereafter a probability update formula is derived for this map building technique. This is followed by an explanation of quadtrees, an adaptive occupancy grid method in Section 2.1.2 along with some of the advantages this method holds over uniform occupancy grids.

### 2.1.1 Uniform Occupancy Grids

The occupancy grid algorithm was first introduced by Moravec and Elfes (Moravec, 1988), where two grids of the same environment is generated by discretizing it into smaller cells. One grid contains the probability that a cell is occupied whereas the other contains the probability that a cell is free. From these two maps a final map is generated by choosing the state with the highest confidence for each cell. To be able to update a map of the environment, the input parameters to the system must first be identified. For any map updating device, this will be its pose estimate ( $x_{1:t}$ ) as well as the sensor measurement data ( $z_{1:t}$ ) given from the first time-step up until time  $t$  (Joubert, 2012).

In an ideal world, hard assignment regarding the occupancy of a grid cell (one for occupied and zero for empty) can be used as the devices are perfect. However, sensor measurements are noisy, which implies that the position estimate as well as the proximity sensor measurements are not accurate. Therefore, it would be more realistic to assign a probability value to each grid cell, where the value represents the probability of the cell being occupied. If  $m_i$  is the outcome of cell  $i$  being occupied given the current pose and distance measurement from the multicopter and sensor respectively, the posterior probability distribution can be determined as follows (Joubert, 2012):

$$p(m_1, m_2, \dots, m_N | z_{1:t}, x_{1:t}) \quad (2.1)$$

where  $N$  represent the number of cells in the grid. The problem with Equation 2.1 is that it requires a large amount of computational power as it updates the entire grid for each new measurement that is taken. To overcome this problem, the state of each individual cell can be estimated by using Equation 2.2 (Joubert, 2012). This states that the probability of cell  $i$  being occupied can be calculated if both the pose estimate as well as the distance measurements received from the proximity sensor are available.

$$p(m_i | z_{1:t}, x_{1:t}) \quad (2.2)$$

If conditional independence is assumed (Joubert, 2012), the state of each cell can be multiplied, as seen in Equation 2.3, to calculate the posterior probability distribution as given in Equation 2.1.

$$p(m_1, m_2, \dots, m_N | z_{1:t}, x_{1:t}) = \prod_i p(m_i | z_{1:t}, x_{1:t}) \quad (2.3)$$

By incorporating the measurements into a global coordinate system, the pose information is included in the measurements themselves since measurements are given relative to the proximity sensors' current pose. Therefore, the pose information  $x_{1:t}$  may be omitted as it does not supply any additional information regarding the environment (Joubert, 2012).

$$p(m_i | z_{1:t}, x_{1:t}) = p(m_i | z_{1:t}) \quad (2.4)$$

By following Bayes' theorem and assuming independence between the different measurements, Equation 2.4 can be written as

$$\begin{aligned} p(m_i|z_{1:t}) &= \frac{p(z_t|m_i, z_{1:t-1})p(m_i|z_{1:t-1})}{p(z_t|z_{1:t-1})} \\ &= \frac{p(z_t|m_i)p(m_i|z_{1:t-1})}{p(z_t|z_{1:t-1})} \end{aligned} \quad (2.5)$$

From Bayes' theorem, it therefore follows that

$$p(z_t|m_i) = \frac{p(m_i|z_t)p(z_t)}{p(m_i)} \quad (2.6)$$

Substituting Equation 2.6 into Equation 2.5 gives

$$p(m_i|z_{1:t}) = \frac{p(m_i|z_t)p(z_t)p(m_i|z_{1:t-1})}{p(m_i)p(z_t|z_{1:t-1})} \quad (2.7)$$

Similarly, if  $m_i^c$  is the compliment of  $m_i$ , therefore, the event that cell  $i$  is empty, the following is also true

$$p(m_i^c|z_{1:t}) = \frac{p(m_i^c|z_t)p(z_t)p(m_i^c|z_{1:t-1})}{p(m_i^c)p(z_t|z_{1:t-1})} \quad (2.8)$$

Factors that could prove difficult to compute are eliminated by dividing Equation 2.7 with Equation 2.8, leaving

$$\frac{p(m_i|z_{1:t})}{p(m_i^c|z_{1:t})} = \frac{p(m_i|z_t)}{p(m_i^c|z_t)} \frac{p(m_i|z_{1:t-1})}{p(m_i^c|z_{1:t-1})} \frac{p(m_i^c)}{p(m_i)} \quad (2.9)$$

By taking the natural logarithm on both sides, a log-likelihood ratio is obtained. This is mainly done to avoid truncation errors as well as the multiplication of probabilities.

$$\begin{aligned} \log \left( \frac{p(m_i|z_{1:t})}{p(m_i^c|z_{1:t})} \right) &= \log \left( \frac{p(m_i|z_t)}{p(m_i^c|z_t)} \right) + \log \left( \frac{p(m_i|z_{1:t-1})}{p(m_i^c|z_{1:t-1})} \right) \\ &\quad - \log \left( \frac{p(m_i)}{p(m_i^c)} \right) \end{aligned} \quad (2.10)$$

The last term in Equation 2.10, also given in Equation 2.11, is known as the prior probability log-odds ratio of a cell (Joubert, 2012). In the absence of any prior knowledge regarding the environment, this term is set to zero.

$$\log \left( \frac{p(m_i)}{1 - p(m_i)} \right) \quad (2.11)$$

The second term on the right hand side of Equation 2.10 is the probability of cell  $i$  being occupied. This value takes into account all previous measurements

of cell  $i$ , whereas the first term on the right hand side of Equation 2.10 is known as the inverse sensor model (Section 3.3). This term contains only new information gathered at time  $t$ , enabling Equation 2.10 to integrate new measurements into the grid through addition.

To convert the log-odds ratio, given on the left hand side of Equation 2.10, back to a probability value, Equation 2.12 and 2.13 can be used (Joubert, 2012). This enables each cell to be updated by using all the previous sensor measurements, the current sensor measurements as well as prior knowledge regarding obstacles.

$$\lambda = \log \left( \frac{p(m_i|z_{1:t})}{1 - p(m_i|z_{1:t})} \right) \quad (2.12)$$

$$p(m_i|z_{1:t}) = \frac{e^\lambda}{1 + e^\lambda} \quad (2.13)$$

### 2.1.2 Adaptive Occupancy Grids - Quadrees

Unlike the uniform occupancy grid described in Section 2.1.1, the environment is not completely discretized into smaller cells, but divided into more manageable regions. In essence the quadtree algorithm is initialized with a single node and as objects are added to this node it will be split into four regions to better isolate the objects. The process of dividing the different nodes into four sections will continue till the object is completely isolated in the smallest grid size possible. Any object that does not fully fit inside a node's boundary will be placed in the parent node as illustrated in Figure 2.1.

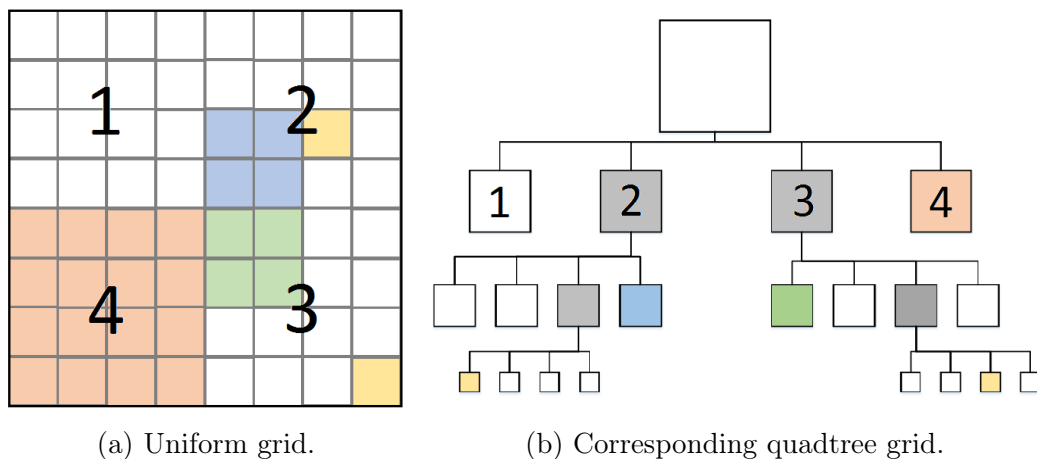


Figure 2.1: Quadtree map building example. From this image it can be seen how the environment is continuously split into four regions to better isolate obstacles. If a uniform grid (a) was used for the same obstacle configuration, 64 cells would have been required as opposed to the 21 required to represent the quadtree map (b).

One of the main advantages of quadtrees is the decreased amount of storage space required to save the same data as an uniform grid (Mujumdar and Padhi, 2011). In the case of the illustration in Figure 2.1, the quadtree representation only occupies 21 cells where the uniform grid would have occupied 64 cells for the same map, saving around 65% of storage space. If, however, a map is being built of a large cluttered environment, this method becomes computationally expensive as nodes have to be split until all the objects are isolated. This means that for some configurations of objects, it is possible for the quadtree map to split into the same amount of cells used for the uniform grids. In such a case, this method would give exactly the same results as a uniform grid, but would be more computationally intensive.

## 2.2 Existing Obstacle Avoidance Algorithms

In this section, four existing obstacle avoidance algorithms are discussed. Firstly, the Virtual Force Field (VFF) method is introduced in Section 2.2.1. This is followed by three global path planning algorithms: A\* (Section 2.2.2), Lifelong Planning A\* (Section 2.2.3) and D\* Lite (Section 2.2.4).

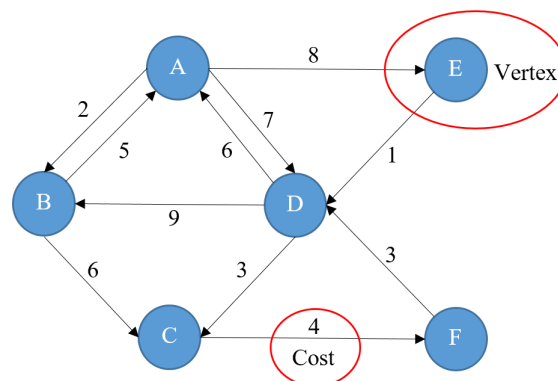


Figure 2.2: Graph representation. This is a visual representation of the information needed to calculate the most cost effective path between two vertices.

To better understand the discussions regarding the global path planning algorithms, some of the terminology used, first has to be clarified. The illustration shown in Figure 2.2 is referred to as a graph ( $S$ ). A graph usually contains all the information needed to plan a path from one point to the next. Normally a graph consists of two components: vertices and cost. A vertex ( $s$ ), also referred to as a node, is the meeting point of two or more lines to form an angle as seen in Figure 2.2. To make the transition from one vertex to the next, a value is introduced to represent the amount of effort needed to complete this task i.e. cost. Usually the cost is linked to the distance the vehicle

must travel as well as the task that has to be completed for instance changing direction or altitude. This value is normally calculated with a predetermined formula and may change as the graph is updated with new proximity sensor measurements. The exact cost of a path from the search algorithms' starting point to any vertex,  $s$ , is given by  $g(s)$  or  $rhs(s)$  and is respectively referred to as the  $g$ -value or the right hand side value. The estimated heuristic cost from any vertex,  $s$ , to another vertex,  $s'$ , is given by  $h(s, s')$  and is sometimes referred to as the  $h$ -value.

### 2.2.1 Virtual Force Field

This method is known as a global path planning algorithm with local collision avoidance capabilities. It was mainly used in ground robots and only recently found use in aerial vehicles. One of the most significant benefits of this method is the fact that no knowledge of the vehicle model is needed to obtain results (Paul *et al.*, 2008).

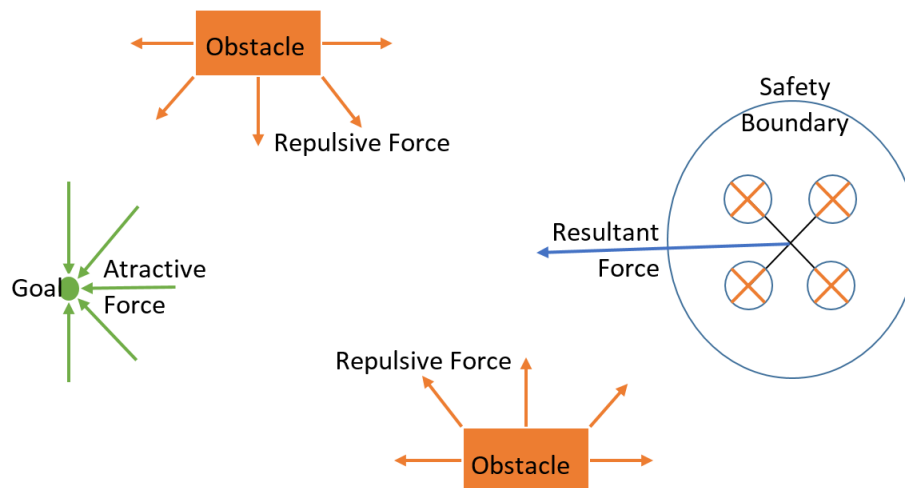


Figure 2.3: VFF vector diagram. By giving the goal position an attraction force and all the obstacles a repulsion force, it is possible to calculate a resultant force as well as a resultant direction as indicated (Coetzee and Smit, 2016).

The potential function usually consists of two forces. An attraction force that pulls the vehicle to the goal and a repulsion force to ensure that the vehicle does not collide with obstacles (Mujumdar and Padhi, 2011). The vehicle can therefore be seen as an electrical agent with a positive charge. If all the obstacles are given a positive charge as well, there will be a repulsion force on the vehicle. On the other hand, if the goal is given a negative charge, there will be an attraction force on the vehicle (De Filippis *et al.*, 2012). This method therefore calculates the resultant force along with the resultant direction by summing up all the known forces to plan its next position as seen in Figure 2.3.



An alternative way of implementing this method is by taking the gradient of the combined forces to plan the next position. This can be visualised as a marble on the floor. If the floor is slanted in a certain direction, the marble will start rolling until it reaches the bottom (De Filippis *et al.*, 2012), as illustrated in Figure 2.4.

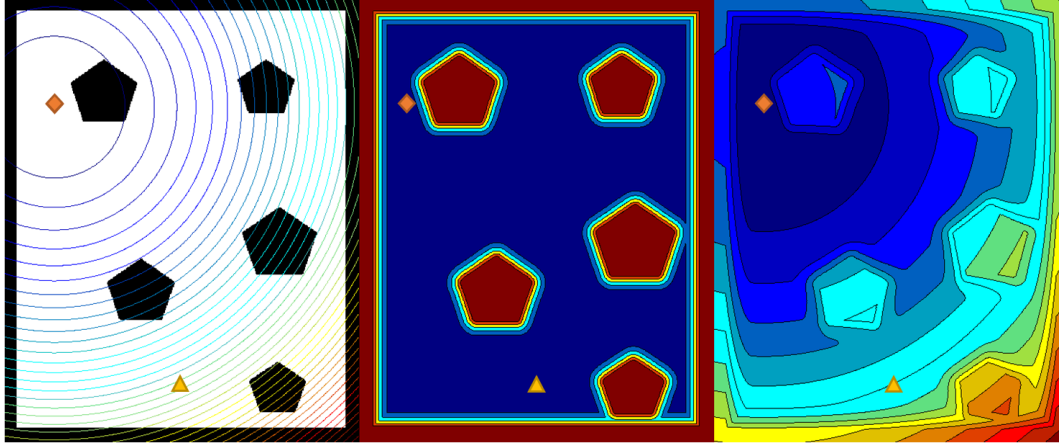


Figure 2.4: VFF gradient diagram. The triangle and the diamond respectively represents the start and goal position of the robot. The first image represents the attractive force around the goal position, while the second image represents the repulsive force from each obstacle. The last image represents the sum of these forces (Coetzee and Smit, 2016).

The VFF method, however, does not explicitly avoid moving obstacles and has a tendency to get caught in local minima. Therefore, the vehicle can get caught in a cluttered environment before reaching its goal (Mujumdar and Padhi, 2011). These issues can, however, be addressed with the proper implementation of a global path planning algorithm (Mujumdar and Padhi, 2011; Droeschel *et al.*, 2016).

### 2.2.2 A\*

The A\* global path planning algorithm is still one of the most popular incremental heuristic search methods that exists since it is fairly easy to understand and flexible to use in a wide range of scenarios (Choset, 2005). When the algorithm is initialized, the start vertex is added to a priority queue. After the algorithm has been initialized, the search for the shortest path is initiated. This is done by expanding the vertices in the priority queue. Since the start vertex is the only value in the priority queue at this point, the algorithm will start there. This vertex is then removed from the priority queue and all its neighbouring vertices are evaluated. The  $g$  and  $h$ -values for these vertices are then determined along with  $f(s)$  given by Equation 2.14 (Choset, 2005). These

newly evaluated vertices are then all placed in the priority queue and reordered in a non-descending order with regards to  $f(s)$ . The vertex with the smallest  $f(s)$  value is then removed from the priority queue and all its neighbours are evaluated. This process continues until the the goal vertex has a smaller  $f(s)$  value than any of the other vertices in the priority queue or until the queue is empty.

$$f(s) = g(s) + h(s, s_{\text{goal}}) \quad (2.14)$$

To find the shortest path after the algorithm is executed, each vertex has to keep track of its predecessor. This allows the the algorithm to give the shortest path by retracing its steps from the goal back to the start vertex. This algorithm can, however, not reuse any of its calculated data to recalculate the shortest path if necessary and therefore always have to plan from scratch. For more information regarding this search algorithm, refer to the pseudo code given in Appendix A.1.

### 2.2.3 Lifelong Planning A\*

Lifelong Planning A\* (LPA\*) is a incremental heuristic search method that repeatedly calculates the shortest path between any two vertices in a dynamic environment (Koenig and Likhachev, 2002a). When the algorithm is initialized, the  $g$ -values as well as the right-hand side ( $rhs$ ) values of all the vertices are set to  $\infty$ . The  $rhs$ -value of the start vertex (the current position of the robot) is then set to zero, making the vertex locally inconsistent i.e.  $g \neq rhs$ . If  $g = rhs$ , the vertex can be seen as locally consistent. When a vertex is locally inconsistent, it is inserted into a priority queue.

After the algorithm has been initialized, the search for the shortest path is initiated. This is done by expanding the vertices in the priority queue. Since the start vertex is the only value in the priority queue at this point, the algorithm will start there. This vertex is then removed from the priority queue and its neighbouring vertices are evaluated. The  $rhs$ -values of these vertices are then updated according to the cost formula. By updating the  $rhs$ -value, these vertices are made locally inconsistent, and are therefore also added to the priority queue. The queue is then re-ordered in a non-decreasing order of their keys. The key,  $k(s)$ , of a vertex is a vector with two components:

$$k(s) = [k_1(s); k_2(s)] \quad (2.15)$$

where

$$\begin{aligned} k_1(s) &= \min(g(s), rhs(s)) + h(s, s_{\text{goal}}) \\ k_2(s) &= \min(g(s), rhs(s)) \end{aligned} \quad (2.16)$$

The  $rhs(s)$ -value represents the cost from the current vertex to the start vertex whereas  $h(s, s_{\text{goal}})$  can be seen as the estimated heuristic cost from the current

vertex to the goal position. The smallest distance from the current vertex to the start vertex is calculated by  $k_2(s)$ . If multiple vertices with the same value for  $k_1$  are calculated, the vertex with the smallest  $k_2$  value will be evaluated first. Therefore, by using this key, the vertices are expanded in an order that tries to focus the search in the direction of the goal position. This continues until the goal vertex is locally consistent and the key value of the vertex to expand next is not smaller than the goal vertex. If  $g(s_{\text{goal}})$  is still  $\infty$  after the search, there is no valid path available (Koenig and Likhachev, 2002a).

During the vertex expansion, a locally inconsistent vertex  $s$  is called locally over-consistent if  $g(s) > rhs(s)$ . In this case the  $g$ -value will be set to the  $rhs(s)$  value, making the vertex locally consistent while expanding to its neighbours. Locally inconsistent nodes are called under-consistent in the case where  $g(s) < rhs(s)$ . The  $g$ -value will then be set to  $\infty$ , making the vertex either locally consistent or over-consistent, causing the node to be added to the priority queue again (Koenig and Likhachev, 2002a).

If  $g(s_{\text{goal}})$  does not equal  $\infty$ , the shortest path can be calculated by starting from the goal vertex and moving to any preceding vertex that minimizes the cost until the start vertex is reached. In other words, starting at the goal vertex, each of the neighbouring vertices are checked for the smallest  $g$ -value. When that vertex is identified, its neighbouring vertices are investigated and the procedure is repeated until the start vertex is reached (Koenig and Likhachev, 2002a).

At first glance, LPA\* will give the same results as A\* since the first search will yield exactly the same results. However, the subsequent searches will reuse some of the previously calculated data in LPA\*, whereas A\* will have to plan from scratch, leading to faster execution time. For more information regarding this search algorithm, refer to the pseudo code given in Appendix A.2.

### 2.2.4 D\* Lite

This algorithm is based on the same algorithmic progression as LPA\* and therefore has the same properties (Koenig and Likhachev, 2002b). The first version of D\* Lite that is explained, can therefore be seen as an extended version of LPA\*. The only difference between the two algorithms is the fact that the vertices are expanded from the goal position to the start position, changing the components of Equation 2.15 to:

$$\begin{aligned} k_1(s) &= \min(g(s), rhs(s)) + h(s, s_{\text{start}}) \\ k_2(s) &= \min(g(s), rhs(s)) \end{aligned} \tag{2.17}$$

For Equation 2.17 to work, the heuristic function has to be non-negative and backwards consistent i.e. obey  $h(s_{\text{start}}, s_{\text{start}}) = 0$  and  $h(s_{\text{start}}, s) = h(s_{\text{start}}, s') + c(s', s)$  for  $s \in S$  and  $s' \in \text{pred}(s)$ . If  $g(s_{\text{start}}) = \infty$  after the search, there is no valid path available. The shortest path, however, can then

be calculated in a similar way to the method used in LPA\* by working from the start vertex to the goal vertex. Unfortunately, there is a problem with implementing the code in the manner described above. Every time the robot or vehicle moves, the key values of the vertices in the priority queue will have to be updated as the heuristic values of the vertices change when the robot moves. This only affects the key values of the vertices in the priority queue and not which vertices are locally consistent and thus in the queue. It can, however, get computationally expensive as the priority queue often contains a large number of vertices that have to be recalculated after every move (Koenig and Likhachev, 2002b).

Since the vertices are evaluated from the goal position to the start position and the heuristic function is aimed at minimizing the distance to the start position, the keys are evaluated in a descending order unlike LPA\*. After the robot moves a certain distance along the calculated path, it may detect an obstacle. Now, if the vertex the robot is currently on is re-evaluated, the first component of the key will have decreased by, at most,  $h(s, s')$ . The second component of the key does not depend on the heuristic function and will therefore not be influenced. To maintain lower bounds,  $h(s, s')$  needs to be subtracted from all the key values. But if  $h(s, s')$  is subtracted, the order of the keys will not change. However, if the subtraction is not done, the new key values calculated will be smaller than the rest by  $h(s, s')$ . Therefore, the key modifier ( $k_m$ ) term is introduced in the key function to compensate for this (Koenig and Likhachev, 2002b).

$$\begin{aligned} k_1(s) &= \min(g(s), rhs(s)) + h(s, s_{\text{start}}) + k_m \\ k_2(s) &= \min(g(s), rhs(s)) \end{aligned} \quad (2.18)$$

where

$$k_m = k_m + h(s_{\text{last}}, s_{\text{start}}) \quad (2.19)$$

and  $s_{\text{last}}$  represents the vertex the path was last recalculated at while  $s_{\text{start}}$  represents the current vertex of the vehicle the new path should be calculated to. Now if the robot moves and it detects changes in the environment again, the  $k_m$  value will have to be increased accordingly. In other words, the order of the vertices in the priority queue will not change or have to be recalculated any more when the robot moves, saving computational power.

For these changes to be valid, the heuristic function has to be forward-backward consistent. This implies that the heuristic function has to obey  $h(s, s'') \leq h(s, s') + h(s', s'')$  for all  $s, s', s'' \in S$ . The vertices also need to be acceptable no matter where the goal vertex is, therefore, they have to obey  $h(s, s') \leq c^*(s, s')$  for all  $s, s' \in S$  where  $c^*(s, s')$  is the cost of the shortest path from  $s$  to  $s'$  (Koenig and Likhachev, 2002b).

D\* Lite is therefore able to re-plan faster than starting from scratch each time as it modifies previous search results when needed. In other words, it is more efficient in recalculating the shortest path from the current vertex to

the goal vertex, because it only recalculates the necessary vertices that have changed or have not been calculated before. For more information regarding this search algorithm, refer to the pseudo code given in Appendix A.3.

## 2.3 Conclusion

This chapter focuses on two map building techniques as well as a few existing obstacle avoidance algorithms. In Section 2.1 a discussion is given regarding two different map building techniques: uniform and adaptive occupancy grids. A probability map update formula was then derived for uniform occupancy grids and was found to be (Joubert, 2012):

$$\log \left( \frac{p(m_i|z_{1:t})}{p(m_i^c|z_{1:t})} \right) = \log \left( \frac{p(m_i|z_t)}{p(m_i^c|z_t)} \right) + \log \left( \frac{p(m_i|z_{1:t-1})}{p(m_i^c|z_{1:t-1})} \right) - \log \left( \frac{p(m_i)}{p(m_i^c)} \right) \quad (2.10)$$

To be able to plan a path from a given map, different obstacle avoidance techniques are briefly investigated in Section 2.2. The discussed algorithms include the Virtual Force Field in Section 2.2.1, A\* in Section 2.2.2 as well as an expansion of A\*: Lifelong Planning A\* (LPA\*) in Section 2.2.3. LPA\* is then further expanded to D\* Lite in Section 2.2.4.

To be able to implement D\* Lite correctly, the heuristic function has to adhere to  $h(s, s'') \leq h(s, s') + h(s', s'')$  for all  $s, s', s'' \in S$ . The vertices also need to be acceptable no matter where the goal vertex is, therefore, they have to obey  $h(s, s') \leq c^*(s, s')$  for all  $s, s' \in S$  where  $c^*(s, s')$  is the cost of the shortest path from  $s$  to  $s'$ . The key,  $k(s)$ , used to determine what vertex should be expanded next, is defined as (Koenig and Likhachev, 2002a)

$$k(s) = [k_1(s); k_2(s)] \quad (2.15)$$

where

$$\begin{aligned} k_1(s) &= \min(g(s), rhs(s)) + h(s, s_{\text{start}}) + k_m \\ k_2(s) &= \min(g(s), rhs(s)) \end{aligned} \quad (2.18)$$

and

$$k_m = k_m + h(s_{\text{last}}, s_{\text{start}}) \quad (2.19)$$

where  $s_{\text{last}}$  represents the vertex the path was last recalculated at while  $s_{\text{start}}$  represents the current vertex of the vehicle the new path should be calculated to.

## Chapter 3

# Measurement and Pose Uncertainty

In this chapter, pose as well as measurement uncertainty in the map building procedure is investigated. This is done to more accurately simulate how the map building algorithm will be affected in a dynamic environment. The focus is therefore placed on the effect that these uncertainties have on the map building algorithms' output and whether the same results can be obtained by using simplified assumptions.

### 3.1 Overview: Measurement Uncertainty

From Equation 2.10 it was found that each occupancy grid cell must be assigned a probability value based on a given sensor measurement at time  $t$ . Each sensor measurement taken consists of an angle as well as the distance to the first observed obstacle along that line. To update the occupancy grid correctly, each of the cells intersected by the measurement line must be updated accordingly. Equation 2.10, therefore requires the inverse sensor model,  $p(m_i|z_t)$ . It is referred to as the inverse sensor model as it is the state of cell  $i$  that affects the measurement and not the measurement that affects the state (Joubert, 2012).

In Section 3.2 the relevant assumptions made regarding the measurement uncertainty is highlighted. This is followed by a discussion of the ideal inverse sensor model, whereafter measurement uncertainty is incorporated into the model for a more realistic representation of actual sensor measurements in Section 3.3.

### 3.2 Assumptions: Measurement Uncertainty

Before the inverse sensor model is derived in Section 3.3, the relevant assumptions regarding measurement uncertainty must be clarified. Firstly, as the

sensor used is a LIDAR (refer to Chapter 4 and 5 for more information) the data will be received in the form  $(\theta, r)$ , where  $\theta$  is the angle of the ray and  $r$  is the distance to the first observed obstacle along that ray. A LIDAR can be classified as a narrow beam sensor as the beam spans less than  $1^\circ$ . Therefore, the angle uncertainty in the beam itself is negligible compared to the distance uncertainty along the measurement line. By using this assumption, it is possible to create a one-dimensional inverse sensor model that is a function of the measurement distance  $r$ . Secondly, measurement independence, as mentioned in the map update formula derivation in Section 2.1.1, is assumed. This enables the update process to be handled separately for each measurement (Joubert, 2012).

### 3.3 Inverse Sensor Model Derivation

In this section an ideal inverse sensor model is initially derived under the assumption of noise free measurements. This is then followed by a derivation of an inverse sensor model corrupted with Gaussian noise. To see what effect sensor noise will have on the chosen map building algorithms' output, the sensor model was incorporated into a simulation. The simulation results as well as the conclusions reached are discussed in Section 3.7.

#### 3.3.1 Ideal Inverse Sensor Model

When a new measurement is taken at time  $t$ , the ideal inverse sensor model is used to calculate a log likelihood value for each of the cells intercepted by the measurement ray. This log likelihood value, along with all the previous measurements, is then incorporated into Equation 2.10 to get an updated probability value for each cell. If a distance of  $Z$  is measured at time  $t$ , the ideal sensor model should return  $p(m_i|z_t) = 0$  for all the cells in front of an obstacle,  $p(m_i|z_t) = 1$  for the cells containing the obstacle and  $p(m_i|z_t) = 0.5$  for all the cells behind an obstacle as seen in Table 3.1. Therefore, the function representing an ideal sensor model can be seen in Figure 3.1 and is defined as (Joubert, 2012):

$$g(r) = \begin{cases} 0 & \text{if } r \in (-\infty, Z - \frac{L}{2}) \\ 1 & \text{if } r \in [Z - \frac{L}{2}, Z + \frac{L}{2}) \\ 0.5 & \text{if } r \in [Z + \frac{L}{2}, \infty) \end{cases} \quad (3.1)$$

where  $r$  represents the distance from the current position of the laser range finder to the cell being updated and  $Z$  represents the actual measurement given by the sensor. Since the center of the occupancy grid cells are used to calculate the distance to the sensor, the peak at  $Z$  can be missed if the calculated distance,  $r$ , has a slight offset from the measured distance. To compensate for this, another parameter,  $L$ , is introduced, signifying a band of

$r$  values that will all receive the same probability, ensuring that the peak is not missed and all the cells are updated correctly.

Table 3.1: Correspondences between the probabilities and the log odds ratios along with their respected interpretations (Joubert, 2012).

$p(m_i z_t)$	$\log\left(\frac{p(m_i z_t)}{1-p(m_i z_t)}\right)$	Interpretation
0	$-\infty$	definitely free
0.5	0	unknown
1	$\infty$	definitely occupied

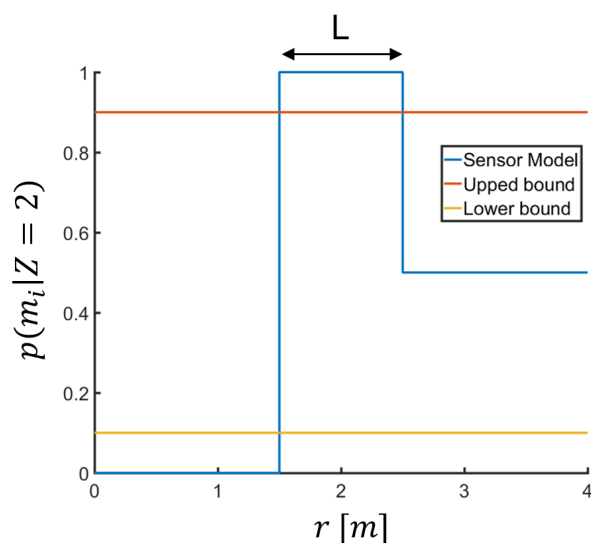


Figure 3.1: Ideal inverse sensor model (Equation 3.1) with  $L = 1$ . Both the upper and lower bounds are also indicated as a value of 0 or 1 would be mapped to  $-\infty$  or  $\infty$  (Table 3.1), implying that no new measurements can be added to a cell any more.

The inverse sensor model is, however, not used as shown in Figure 3.1. This is due to the fact that either a value of 0 or 1 will be mapped to  $-\infty$  or  $\infty$  respectively, implying that new measurements will not be able to change the value of the cell any more. To avoid this problem, the sensor model is given both an upper and a lower bound value (Joubert, 2012).

### 3.3.2 Inverse Sensor Model with Gaussian Noise

To be able to replicate actual sensor measurements more accurately in a simulated environment, normally distributed noise around the measured distance,



$Z$ , was assumed with a standard deviation of  $\sigma$ . The probability density function (PDF) of this Gaussian is therefore (Joubert, 2012):

$$f(r; Z, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(r-Z)^2}{2\sigma^2}} \quad (3.2)$$

and can be seen in Figure 3.2 where  $\sigma$  indicates the certainty of an obstacle

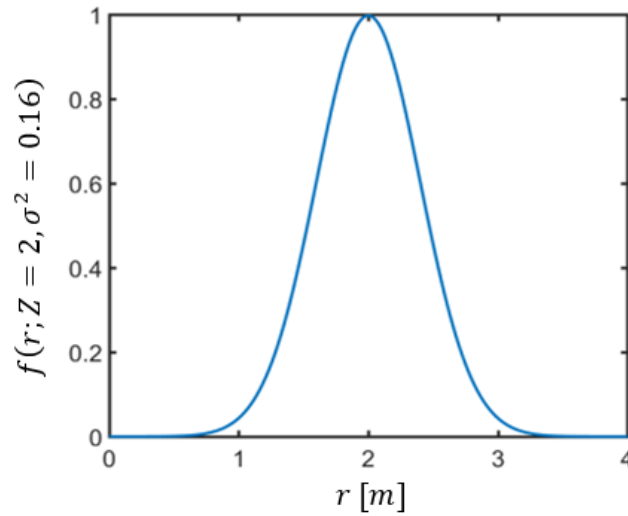


Figure 3.2: Probability Density Function (Equation 3.2) of a measurement at  $Z = 2$  m corrupted by Gaussian noise with a standard deviation of  $\sigma = 0.4$ .

being located at a distance  $Z$ . A large  $\sigma$  will increase uncertainty since the peak will be wider whereas a small  $\sigma$  will give a narrow peak, decreasing uncertainty. To combine the ideal sensor model with the Gaussian noise, a convolution between the functions given in Equation 3.1 and 3.2 has to be performed and is defined as (Joubert, 2012):

$$(f * g)(r) = \int_{-\infty}^{\infty} g(\tau) f(r - \tau) d\tau \quad (3.3)$$

Equation 3.1 is defined as  $g(r) = 0$  for  $r \leq 0$ , even though the sensor can not take measurements behind itself, to help with the convolution. Due to the fact that the ideal sensor model is piece-wise defined, the convolution will also be defined in the same manner. However, when the final formula is used to update a cell,  $r$  will always have to be larger than 0. For the sake of convenience, the following formula is defined to simplify further work (Joubert, 2012).

$$F(a, b) = \frac{1}{\sqrt{2\pi}\sigma} \int_a^b e^{-\frac{(r-\tau-Z)^2}{2\sigma^2}} d\tau \quad (3.4)$$

Using Equation (3.4), Equation (3.3) can be rewritten as

$$(f * g)(r) = kF(a, b) \quad (3.5)$$

where  $k \in [0, 0.5, 1]$ , depending on which interval  $r$  is defined in. By letting

$$u = \frac{r - \tau - Z}{\sqrt{2}\sigma} \quad (3.6)$$

Equation (3.4) can be simplified to

$$F(a, b) = \frac{-1}{\sqrt{\pi}} \int_{\frac{r-a-Z}{\sqrt{2}\sigma}}^{\frac{r-b-Z}{\sqrt{2}\sigma}} e^{-u^2} du \quad (3.7)$$

When introducing the error function, defined as

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad (3.8)$$

Equation (3.7) can further be simplified to

$$F(a, b) = -\frac{1}{2} \text{erf}\left(\frac{r-b-Z}{\sqrt{2}\sigma}\right) + \frac{1}{2} \text{erf}\left(\frac{r-a-Z}{\sqrt{2}\sigma}\right) \quad (3.9)$$

After convolution, the following piecewise defined function was generated (Joubert *et al.*, 2015).

$$(f * g)(r) = \begin{cases} 0 & \text{if } r \in (-\infty, Z - \frac{L}{2}) \\ -\frac{1}{2} \text{erf}\left(\frac{-Z}{\sqrt{2}\sigma}\right) + \frac{1}{2} \text{erf}\left(\frac{r-2Z+\frac{L}{2}}{\sqrt{2}\sigma}\right) & \text{if } r \in [Z - \frac{L}{2}, Z + \frac{L}{2}) \\ -\frac{1}{4} \text{erf}\left(\frac{r-2Z-\frac{L}{2}}{\sqrt{2}\sigma}\right) + \frac{1}{2} \text{erf}\left(\frac{r-2Z+\frac{L}{2}}{\sqrt{2}\sigma}\right) & \text{if } r \in [Z + \frac{L}{2}, \infty) \\ -\frac{1}{4} \text{erf}\left(\frac{-Z}{\sqrt{2}\sigma}\right) & \text{if } r \in [Z + \frac{L}{2}, \infty) \end{cases} \quad (3.10)$$

Figure 3.3 and 3.4 illustrates how an inverse sensor model, like the one given in Equation 3.10, works alongside the grid update equation. Note that the cells in front of the obstacle experienced a probability decreased while the cells containing the obstacle experienced an increase in probability and the cells behind the obstacle stayed unaffected as would be expected.

As with the ideal inverse sensor model in Section 3.3.1, the Gaussian inverse sensor model cannot be used as shown in Figure 3.3 since a value of 0 will be mapped to  $-\infty$ . This implies that new measurements will not be able to change the value of the cell any more (Joubert, 2012). To avoid this problem, the sensor model is given both an upper and a lower bound value. Simulation results regarding the inverse sensor model and its effects on the proposed algorithms' map building capabilities are discussed in Section 3.7.

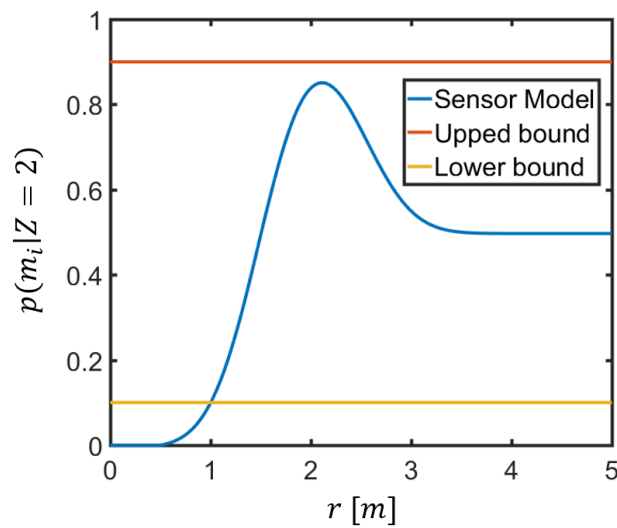


Figure 3.3: Gaussian Inverse Sensor Model (Equation 3.10) with  $L = 1$  m and  $\sigma^2 = 0.16$ . Both the upper and lower bounds are also indicated as a value of 0 or 1 would be mapped to  $-\infty$  or  $\infty$  (Table 3.1), implying that no new measurements can be added to a cell any more.

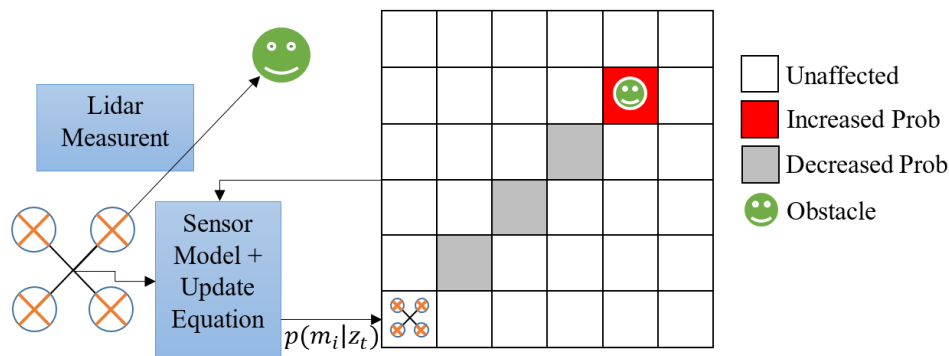


Figure 3.4: Map being updated by using previous map information along with new measurement information (Equation 2.10). White cells are unchanged, whereas the gray cells got a probability decreased and the red cell got a probability increase.

### 3.4 Overview: Pose Uncertainty

Until now, the occupancy grid update equation (Equation 2.10) defined in Section 2.1.1 was used under the assumption that the exact pose of the robot is always known. Unfortunately, this is not the case as sensor drift along with sensor integration can cause faulty pose estimations. This in turn can lead to a flawed map of the environment as small variations in the position of the robot

can have a significant effect on the position of distant objects when updating the map.

In Section 3.5 the relevant assumptions regarding the pose uncertainty are highlighted. This is followed by a derivation of a new map update formula in Section 3.6 that incorporates pose uncertainty directly into the update process. To see what effect the pose uncertainty will have of the chosen algorithms' map building capabilities, the new map update formula was incorporated into a simulation. The simulation results and the conclusion reached are discussed in Section 3.7.

### 3.5 Assumptions: Pose Uncertainty

As obstacle avoidance will only be implemented in a two dimensional environment, there are certain assumptions that need to be made. Firstly, the orientation of the robot will stay constant for the entire flight period. Therefore, the yaw angle of the multicopter is assumed to be constant for the entire flight and will not affect the measurements received from the distance sensor. Secondly, it can be assumed that the multicopter will fly to a specified height and remain at that height until the goal position is reached. Included in this assumption, is the fact that there are no obstacles at either the liftoff or the landing site. Thirdly, since the orientation and the height of the multicopter is assumed to be constant throughout the entire flight, only the horizontal uncertainty of the multicopter will be taken into account. Lastly, it is assumed that the uncertainty in the robots' pose is always available in PDF form (Joubert *et al.*, 2015). Therefore,  $\mu$  and  $\sigma$  can be generated at any given moment.

### 3.6 Pose Uncertainty Derivation

To incorporate pose uncertainty into the grid update equation, the cumulative distribution function (CDF) first has to be calculated. Given a PDF with mean  $\mu$  and variance  $\sigma^2$ , the CDF, as seen in Figure 3.5, can be generated with (Joubert *et al.*, 2015):

$$F_x(x; \mu, \sigma^2) = \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{x - \mu}{\sqrt{2}\sigma}\right) \quad (3.11)$$

To sample from a known one-dimensional PDF, a uniform random number between 0 and 1 must first be generated. This number can then be used to calculate the inverse of the CDF, also known as the inverse transform method (Rubinstein, 2008):

$$x = \sqrt{2}\sigma \operatorname{erf}^{-1}(2u - 1) + \mu \quad (3.12)$$

where  $u$  represents a uniform random number,  $\operatorname{erf}^{-1}$  represents the inverse error function and  $\mu$  as well as  $\sigma$  can be generated from the PDF.

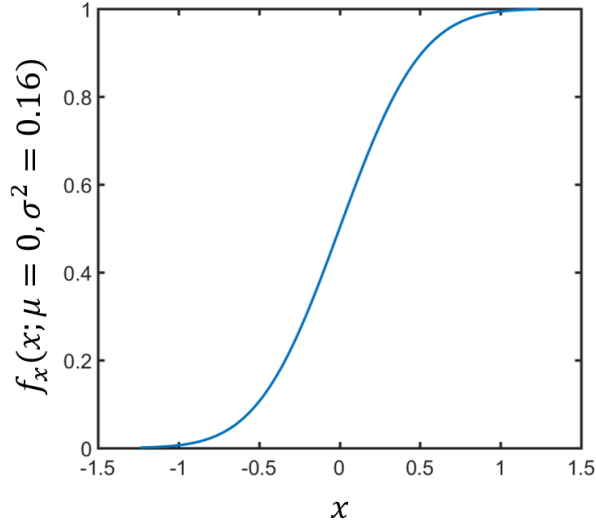


Figure 3.5: This cumulative distribution function (Equation 3.11) shown can be used to determine alternative positions for a vehicle (Equation 3.12) with an average error ( $\mu$ ) of 0 and a variance ( $\sigma^2$ ) of 0.16.

To use the sampled values to update the map, it is assumed that the same measurement can be made from any of the positions generated by Equation 3.12. Since the values were randomly sampled from a normal distribution, it makes sense to give an equal weight to each of them that sums to 1. To find the total value a cell will be updated with, each of the weighted probabilities affecting that cell have to be added together; refer to Figure 3.6 for a visual representation. Taking into account pose uncertainty, Equation 2.10 can be re-written as follows (Joubert *et al.*, 2015):

$$\log \left( \frac{p(m_i | z_{1:t})}{p(m_i^c | z_{1:t})} \right) = \sum_{j=1}^M w_t^{[j]} \log \left( \frac{p(m_i | z_t^{[j]})}{p(m_i^c | z_t^{[j]})} \right) + \log \left( \frac{p(m_i | z_{1:t-1})}{p(m_i^c | z_{1:t-1})} \right) - \log \left( \frac{p(m_i)}{p(m_i^c)} \right) \quad (3.13)$$

where  $M$  represents the number of measurements taken and  $w$  represents the weight of a given measurement. To accurately represent the PDF describing the multicopter's pose uncertainty, the number of samples needed from the CDF has to be calculated. With Equation 3.14 it is possible to calculate how many samples have to be generated from the CDF to get a maximum error,  $E_{\max}$ , between the sampled mean and the actual PDF mean of the data. This equation, however, has an accuracy of 95 % (Joubert, 2012). In other words, if  $\sigma = 0.16$  and  $E_{\max} = 0.1$ , 11 pose samples would have to be generated to be 95% sure that the mean of the sampled values is within a range of 0.1 from the actual PDF mean. Simulation results regarding the pose uncertainty of

the multicopter and its effects on the map building capability of the proposed algorithms are discussed in Section 3.7.

$$M = \left[ 4 \left( \frac{\sigma}{E_{\max}} \right)^2 \right] \quad (3.14)$$

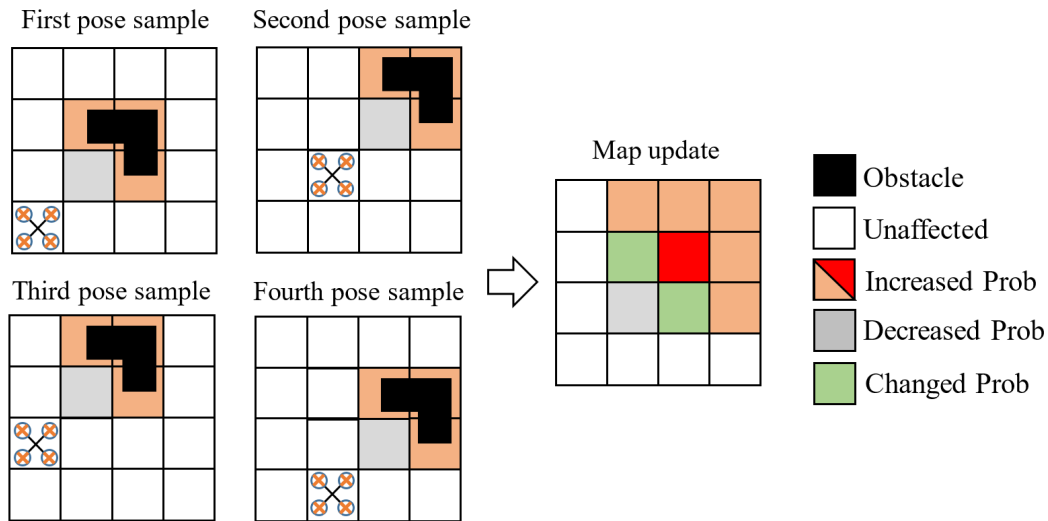


Figure 3.6: Illustration of how the map will be updated with pose uncertainty (Equation 3.13). It is assumed that the same measurement was taken from each of the generated positions (Equation 3.12), updating the map as shown.

## 3.7 Simulation

In the previous sections, analytical formulas for the uncertainty in both the proximity sensor (Section 3.1 - 3.3) as well as the pose estimation (Section 3.4 - 3.6) of the multicopter are derived. This section is aimed at simulating these analytical formulas to determine what effect they will have on the map building process and whether or not simplified methods can be used to generate similar results.

The simulation setup, as well as the data required for the simulations, are discussed in Section 3.7.1. This is followed by the simulation results in Section 3.7.2 as well as the conclusions and a discussion on alternative methods that can be used to obtain similar results in Section 3.8.

### 3.7.1 Setup

For an simulation to accurately represent what happens in the case of pose as well as proximity sensor uncertainties, these uncertainties first had to be quantified. In the case of pose uncertainty, it had to be determined how

accurate the flight controller estimates the position of the multicopter at any given time. In the case of the proximity sensor, two values had to be identified, namely the accuracy of a given distance along the measurement line, and how many measurements can be taken in a given time.

To get a better understanding of the error the flight controller makes in its pose estimation, tests were done using the flight controller alongside a Real Time Kinematic (RTK) GPS (Global Positioning System). The data from these two components were then compared using the RTK GPS as the ground truth measurement since it has a horizontal accuracy of 2 cm in clear skies (Appendix B.6). From the data comparison, the flight controllers' approximate position error was determined. Normal distributions were then fit to the data of which the results can be seen in Table 3.2. It should be noted that the data given in Table 3.2 represent the average error,  $\mu$ , as well as the standard deviation,  $\sigma$ , at the end of each individual test. With the data available to calculate both the average error and the standard deviation at each point during a flight, Equation 3.12 can be used to generate more accurate position estimates for the multicopter during simulation.

### 3.7.2 Results

Using the pose uncertainty setup as described in Section 3.7.1, Table 3.2 was generated. In this table the average error,  $\mu$ , the controller was making at the end of each flight is given in both the Northern and Eastern direction. The standard deviation,  $\sigma$ , illustrates how the controller deviated from the average error at that point in time. Unfortunately, the pose uncertainty update formula could only be used during simulation as the PDF of the error information is not known while flying the multicopter and can only be generated afterwards.

Since the PDF is not known, the effect of the error in the position estimation of the flight controller on the map building technique was investigated. It was found that the map being built by the multicopter will be in constant flux due to the error continuously changing. However, the map will still be traversable as the objects placed in the map will be correct relative to the position estimation of the multicopter at any given time. Therefore, if the error made by the position estimator is not 0 or completely constant, the effects will always be seen in the map building process as seen in Figure 3.8.

If the multicopter, however, has to fly in a known environment without a proximity sensor or any other obstacle detection capabilities, the position estimation error the multicopter is making will have to be taken into account. This is due to the fact that the error may change direction (Table 3.2) for each flight or size while flying. In other words, every time the multicopter is used it may think it is at the correct position, but can be making an average error of up to almost 2 m in any direction as seen in Table 3.2. Hence the pose uncertainty has to be taken into account by means of an extra safety boundary if a known map of the environment is used and the multicopter has no means

Table 3.2: PDF distribution values obtained through testing for both the Northern and Eastern error (Given in North, East and Down (NED) coordinate system). These values represent the average error,  $\mu$ , as well as the standard deviation,  $\sigma$ , at the end of each individual test.

Test #	North		East	
	$\mu$ [m]	$\sigma$ [m]	$\mu$ [m]	$\sigma$ [m]
1	-0.3813	0.2571	-0.1819	0.1864
2	0.5838	0.2951	-0.0847	0.0913
3	-0.6745	0.3395	0.5166	0.2779
4	-1.2974	0.3270	0.7304	0.3291
5	-1.7180	0.3713	-0.7105	0.4214
6	-0.2432	0.3578	-0.5450	0.2203
7	-0.4115	0.2894	0.6085	0.3592
8	0.2373	0.2820	-0.7810	0.2988
9	0.2327	0.3164	0.4611	0.2579
10	-0.9194	0.5784	-0.0911	0.1626
11	-0.2555	0.3311	0.6684	0.2082
12	-0.7241	0.3316	0.0481	0.3072

of obstacle detection. If it was possible to obtain the PDF while flying, the uncertainty would be directly incorporated into the map building process thus enabling the map to be reused as needed without adding additional safety boundaries as described above. Therefore, if the PDF is known while flying, Equation 3.12 will try to approximate the RTK position given the Pixhawk's position estimation as shown in Figure 3.7.

Using the sensor setup as described in Chapter 4, it was found that the measurements given by the LIDAR are time dependant. Therefore, the amount of measurements may vary depending on the distance of the obstacles. In other words, because it takes longer for the LIDAR to measure objects that are further away, more measurements will be generated in a specified amount of time if obstacles are close by. According to the PulsedLight LIDAR datasheet (Appendix B.4), the measurements given by the sensor have an accuracy of  $\pm 2.5$  cm. Since the grid size in the map building process can be adapted (default 0.5 m), the uncertainty in the LIDAR measurement does not have a significant effect unless the grid size is greatly decreased. This in turn will increase the computational time of the global path planner and decrease the step sizes taken i.e. how far the multicopter will move at a time.



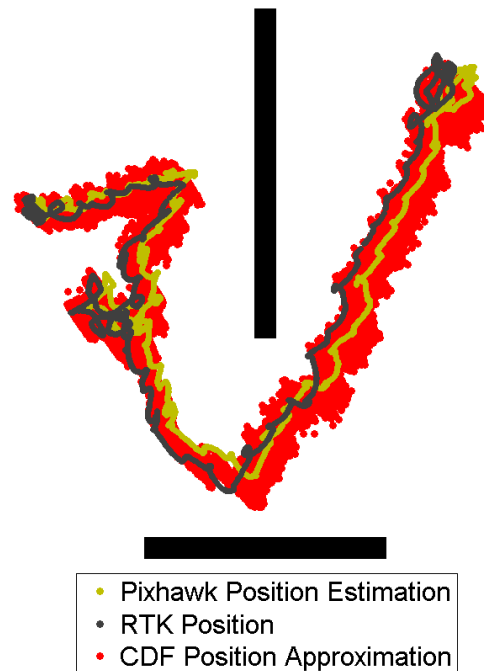


Figure 3.7: CDF position approximation. From this figure, it can be seen how the CDF approximates the RTK position of the multicopter given the Pixhawk's position estimation.

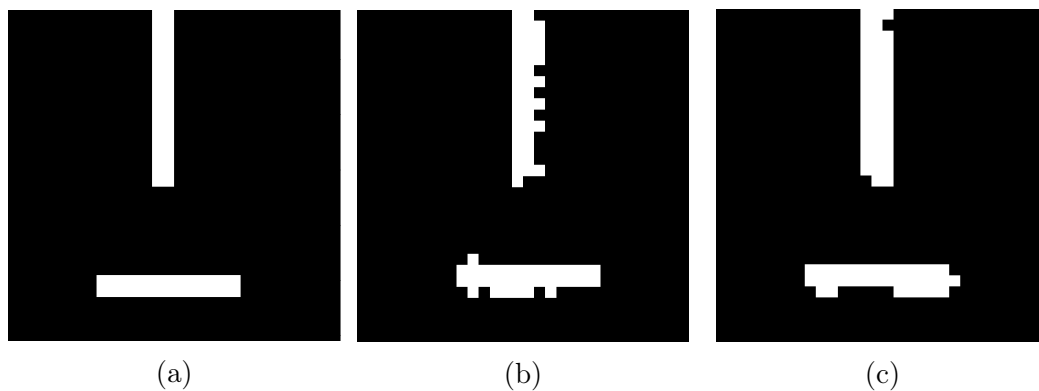


Figure 3.8: Simulated map outputs of Figure 3.7. Figure (a) represents what the map output should look like if the correct position is known while flying. Figure (b) represents the map output generated with the faulty position estimation data. Whereas Figure (c) represent the map that would be built if the position was approximated with the position error data.

Due to the small measurement uncertainty, an ideal-like inverse sensor model would give similar results to Equation 3.10 as seen in Figure 3.9. The upper and lower bounds should, however, still be implemented to ensure that new measurements can be incorporated into the map building process. Equa-

tion 3.10 was derived for a LIDAR, but can be updated for stereo vision or sonar if the derivation of the equation is revisited. For more information regarding the hardware and software used in the sensor setup as well as the map building results obtained from the sensor, please refer to Chapter 4 and 5 respectively.

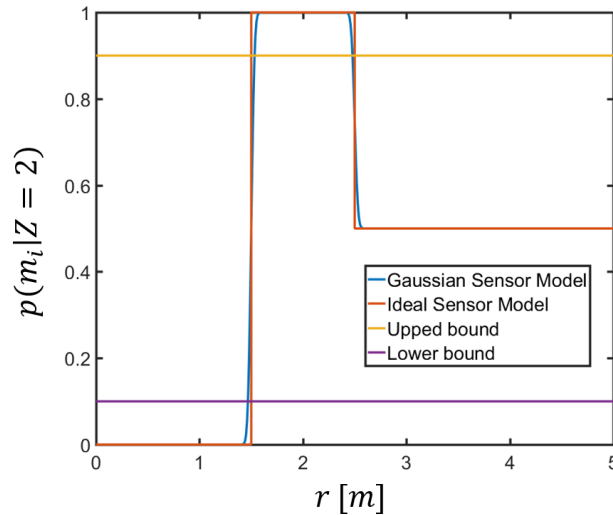


Figure 3.9: Sensor model comparison. When implementing the Gaussian inverse sensor model with a standard deviation of  $\sigma = 0.025$  (since the LIDAR has an accuracy of  $\pm 2.5$  cm) and  $L = 1$  m, the results obtained from the Gaussian inverse sensor model is almost identical to that of the ideal inverse sensor model. Both the upper and lower bounds are also indicated as a value of 0 or 1 would be mapped to  $-\infty$  or  $\infty$  (Table 3.1), implying that no new measurements can be added to a cell any more.

### 3.8 Conclusion

In this chapter analytical formulas for both the proximity sensor (Equation 3.10) as well as the pose uncertainty (Equation 3.13) of the multicopter are derived. These formulas are then simulated and the results evaluated to determine what effect they have on the map building process and whether or not simplified methods can be used to generate similar results.

For pose uncertainty it would be ideal if the PDF is known while the multicopter is flying. This would enable it to construct a more accurate map of the environment that can be reused without adding additional safety boundaries. The Monte Carlo method used, however, can get computationally intensive as each new proximity sensor reading requires a significant amount of samples from the CDF to accurately represent the PDF of the multicopters' pose uncertainty.

Unfortunately, as the PDF is not known while flying, the effects of the pose uncertainty were investigated. It was found that the map being built by the multicopter will be in constant flux due to the error continuously changing. However, the map will still be traversable as the objects placed in the map will be correct relative to the position estimation of the multicopter at any given time.

Using a LIDAR as the measurement instrument, it was found that the ideal sensor model would give similar results to the Gaussian inverse sensor model if the same upper and lower bounds were implemented on it. The Gaussian inverse sensor model can also easily be derived for other sensors like stereo vision and sonar if some of the assumptions made are revisited (Joubert, 2012). An ideal-like inverse sensor model can therefore be used alongside the LIDAR for all intents and purposes.

# Chapter 4

## Hardware and Software Integration

The aim of this chapter is to give the reader an overview of the hardware and software used to integrate different components and build a working prototype of the obstacle avoidance system. In Figure 4.1, all of the components, software and communication methods used can be seen. In Section 4.1 an overview is given regarding all the hardware used. This is followed by an overview of the software used in Section 4.2.

### 4.1 Hardware Integration

In this section, the hardware used for the setup, as can be seen in Figure 4.1, is discussed. This includes the Pixhawk flight controller (Section 4.1.1), the Intel Edison (Section 4.1.2) as well as the Arduino Mega and PulsedLight LIDAR used in the proximity sensor design (Section 4.1.3).

#### 4.1.1 Pixhawk Flight Controller

As set out in the scope of this project (Section 1.2), no work was done on the Pixhawk flight controller itself as the focus of the project was on combining existing components to create a system that enables multicopters to fly autonomously. The flight controller, as seen in Figure 4.2, uses a micro air vehicle communication protocol called MAVLINK. Using the MAVLINK protocol, it is possible to send and receive information from the flight controller with an external computer, usually called a companion computer. The flight controller also comes with an offboard mode enabling the flight controller to be controlled by the companion computer via a serial connection. However, one of the offboard mode safety features is that for the multicopter to stay in offboard mode, it must receive position or velocity commands at least every 0.5 s. If this is not done, the controller reverts back to the mode it was in

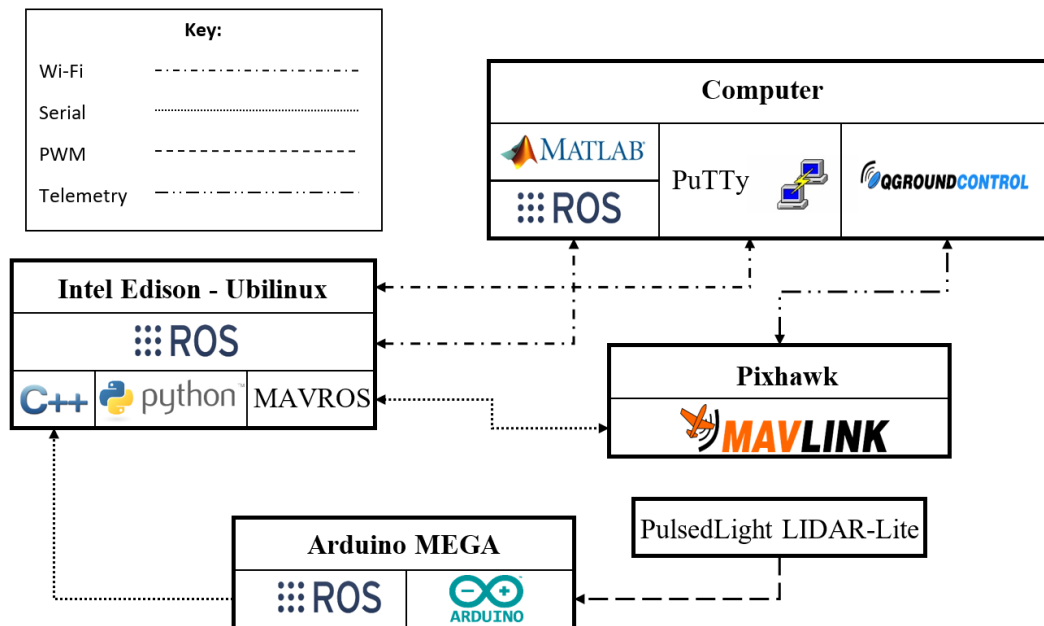


Figure 4.1: Overview of hardware and software integration. Using WiFi it is possible to communicate between the Intel Edison and the Computer. PuTTY can be used as the lowest level of communication i.e. to connect directly to the operating system on the Intel Edison. Or, if Robot Operating System (ROS) is running on the Intel Edison, it is also possible to connect MATLAB directly to it. The Pixhawk flight controller and the Arduino Mega both communicate with the Intel Edison using serial connections as indicated. The flight controller was also connected to Qgroundcontrol (the groundcontrol software) via a telemetry connection. Whereas the rotating Lidar was connected to the Arduino Mega with a PWM connection.

before it was switched to offboard control. For this reason, position control was always chosen before the multicopter was switched to offboard control during flight tests. This enabled the multicopter to keep its position whenever communication was lost, preventing it from crashing. For more information regarding the controllers' capabilities, refer to Appendix B.1.

### 4.1.2 Intel Edison

The Intel Edison, as seen in Figure 4.3, is an open source software development environment with a high performance dual-core CPU (Central Processing Unit) as well as a single core micro-controller. It is also integrated with both WiFi and Bluetooth while having 1 GB of LPDDR3 RAM (Low Power Double Data Rate Random Access Memory) and 4 GB of flash storage. The Intel Edison breakout board can be connected with multiple expansion boards, allowing seamless integration with other components like Arduino shields for



Figure 4.2: Pixhawk flight controller (Pix, 2015).

quick and easy prototyping. For this reason an Arduino expansion board was chosen to allow for maximum flexibility as it contains 20 digital input/output pins. These 20 pins include 4 pins that can be used as PWM (Pulse Width Modulation) outputs as well as 6 pins that can be used as analog inputs. It also contains 1 UART (Universal Asynchronous Receiver/Transmitter), 1 I2C (Inter-Integrated Circuit), a micro USB (Universal Serial Bus) device connector, a micro USB to connect to the UART, a standard size USB host type-A connector and a SD (Secure Digital) card connector. This entire setup can be powered by a 7-15 V DC (Direct Current) power supply.

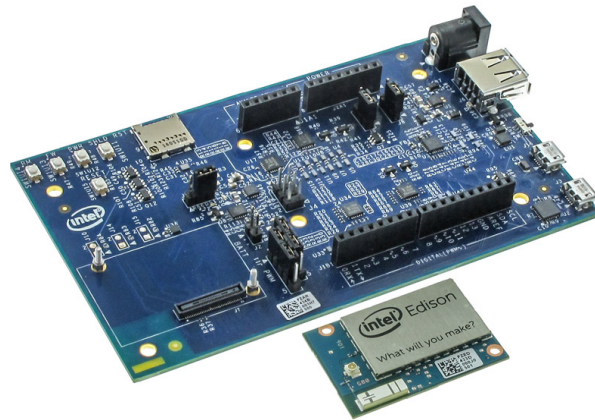


Figure 4.3: Intel Edison breakout board with Arduino expansion board (Int, 2016).

While running Ubilinux, a Linux image based on Debian, as its operating system, the Intel Edison was mainly used as a data relay between all the different components. By combining it with ROS (Robot Operating System), the Intel Edison was able to use MAVROS, a MAVLINK package for ROS, to create a serial connection with the flight controller, enabling it to be used as a

companion computer. By initiating a roserial server node, a serial connection was established with the proximity sensor (Section 4.1.3 and Section 4.2.4) by means of a USB cable. A WiFi connection was established to send all the information received from both the flight controller and the proximity sensor to the ground station computer. This enabled the Intel Edison to send all the ROS data to the ground station computer where it was processed and displayed in MATLAB (Section 4.2.2). For more information regarding ROS and the companion computer's capabilities, refer to Section 4.2.1 and Appendix B.2 respectively.

### 4.1.3 Proximity Sensor Design

To design a sensor capable of working as its own unit the Arduino Mega (Figure 4.4), an open source electronic prototyping platform, was used. By combining the Mega with roserial, it was possible to enable a ROS (Section 4.2.1) communication protocol over the Arduino's UART. This enabled the Arduino to be connected to any ROS system, allowing it to be a ROS node capable of publishing and subscribing to different topics while having access to the ROS system time.

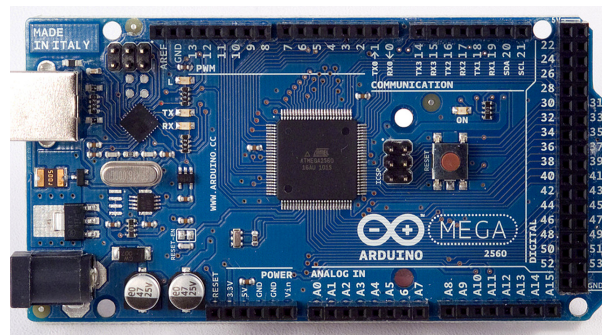


Figure 4.4: Arduino Mega (Ard, 2014).

To get a working 2 dimensional LIDAR system an Arduino Mega, Pulsed-Light LIDAR, continuous servo, pickup sensor and a button had to be integrated. By using a PWM (Pulse Width Modulation) setup, it was possible to connect the PulsedLight LIDAR to the Arduino board. For this type of connection, only 3 cables have to be used. One for power (5 V), one for ground and one for the LIDAR mode pin that has to be connected to both a monitor and a trigger pin on the Arduino board via a 1 k $\Omega$  resistor, as seen in Figure 4.5. The power of the LIDAR sensor also needed to be connected to a capacitor since the distance measurements are taken at high speeds, causing fluctuation in the power supply. The capacitor was therefore used to ensure a more stable power supply.

The trigger pin on the Arduino Mega was set to low ( $\sim 0$  V) in the software (Section 4.2.4). This, in turn, gave the monitor pin a low input with the help of the  $1\text{ k}\Omega$  pull-down resistor. Since the PWM setup was used, the LIDAR was continuously sending out signals to obtain distance measurements. When a signal is sent and the LIDAR is waiting for the signal to return, the mode pin is given a 5 V output. This in turn makes the monitor pin high ( $\sim 5$  V) on the Arduino board. The time this pin is high is then measured by the Arduino. As soon as the signal is received by the LIDAR, the pin is pulled low again and the Arduino stops timing. The time measured can then be translated back to a distance as  $10\text{ }\mu\text{s}$  is 1 cm (Appendix B.4).

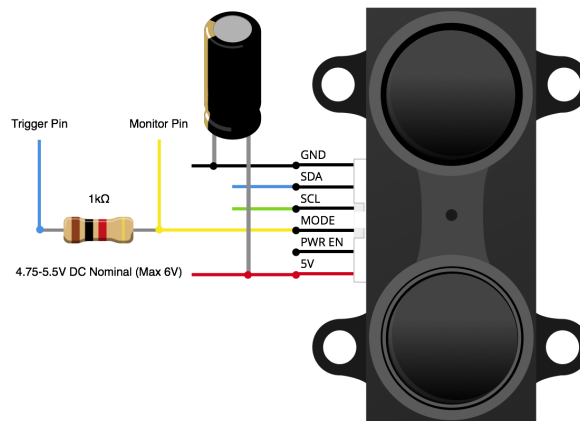


Figure 4.5: PWM wiring setup for PulsedLight LIDAR B.4.

By adding a continuous servo as well as a pickup, it was possible to create a rotating LIDAR sensor capable of taking continuous measurements. With both the distance and the angle known, it is possible to reconstruct the environment being scanned by the LIDAR. The continuous servo only requires 3 connections, one for power, one for ground and the last for a PWM signal. The servo can take a PWM input of  $900\text{--}2100\text{ }\mu\text{s}$ , where  $2100\text{ }\mu\text{s}$  and  $900\text{ }\mu\text{s}$  ensures a top speed in both the clockwise and counter-clockwise directions respectively. From the data sheet of the pickup (Appendix B.5), it can be seen that the current to the emitter has to be limited to 12 mA. To ensure this limit is never exceeded, a  $220\text{ }\Omega$  resistor had to be connected. To enable the entire sensor to only be activated when the user is ready, an extra button was also added. For more information regarding the Arduino Mega, refer to Appendix B.3.

## 4.2 Software Integration

In this section, the software used for the setup, as can be seen in Figure 4.1, is discussed. This includes the use of Robot Operating System (Section 4.2.1)



in MATLAB (Section 4.2.2), Python (Section 4.2.3), C++ and Arduino (Section 4.2.4). In Section 4.2.5 Hardware in the Loop (HITL) software is also investigated to see how it can be used to simulate multicopter flight.

## 4.2.1 Robot Operating System

Robot Operating System (ROS) is a flexible framework for writing robot software. It is becoming a widely used development medium as truly robust robot software is not easy to program from scratch. Currently, ROS only runs on Unix-based systems and that is why Ubilinux (an embedded Linux distribution based on Debian) was used on the Intel Edison (Section 4.1.2).

ROS is an open-source, meta-operating system that provides the same services expected from an operating system. These services include the freedom to add hardware, device control on a lower-level, package management as well as message distribution between different processes. It is also possible to write, build and run code across multiple platforms (Arduino, Intel Edison and Desktop) while using one or a combination of different programming languages (Arduino, Python, C++ and MATLAB).

ROS has several different ways of communicating between different programs and hardware. This includes a synchronous Remote Procedure Call (RPC) communication method used to request services. Therefore, the RPC can be used to enable a procedure or subroutine to execute, commonly on another computer in a shared network. Another way of communicating is by using topics where data streaming is done asynchronously, which decouples the production of the information from its consumption. In general, the nodes generating the information do not know where the information is going or who they are communicating with. Instead, they publish the data to the relevant topic and all the nodes that need this information can subscribe to the same topic. Therefore, a node can be described as a process that performs calculations and communicates by means of topics. A robot control system would usually consist out of multiple nodes where, for example, one node would control the laser range-finder, one node would perform localization while another node performs path planning etc.

By using nodes, code complexity is reduced while fault finding can easily be done as crashes are isolated to individual nodes. The implementation details are also hidden from the rest of the nodes/topics as they only have access to the data that is published. Each topic has a message type that is used to publish or subscribe to it. Therefore, if the subscriber message type does not correspond with the publisher message type, a communication bridge between the two topics will not be established.

## 4.2.2 MATLAB

After the necessary ROS messages (Section 4.2.1) were added to MATLAB, it was possible to connect to the Intel Edison via a ROS communication protocol. A WiFi link was established by MATLAB, enabling it to create an obstacle avoidance node on the Intel Edison. This node was then able to publish and subscribe to the necessary topics created by the Pixhawk, allowing MATLAB to plan a collision free path for the multicopter.

The MATLAB node was subscribed to two topics. The first was the local position of the multicopter. This topic provides the position and orientation of the multicopter at any given moment. The second was the LIDAR topic. This node and topic was generated by the proximity sensor and contained the data necessary to reconstruct the environment. The avoidance algorithm was interrupted every time new data was received, enabling it to immediately be integrated into a map. This in turn enabled the multicopter to react on new obstacles as fast as possible.

A publisher was created in MATLAB to publish new setpoints generated by the avoidance algorithm. The python script (Section 4.2.3) running on the Intel Edison was subscribed to this publisher, enabling it to receive the new data and continuously resend it to the multicopter to keep it in offboard control (Section 4.1.1).

The main reason MATLAB was chosen to program the avoidance algorithm is its debugging capabilities. With the setup shown in Figure 4.1 it is possible to easily make changes to the algorithm as well as see a live feed of the map building process and the planned path. Errors that were not present in the HITL simulations (Section 4.2.5) could therefore easily be isolated and corrected during the outdoor flight tests. If the same avoidance algorithm was programmed on the Intel Edison, it would not have been possible to isolate errors so easily as a visual representation of all the variables, planned path and environment would not have been available. For more information regarding the avoidance algorithm programmed in MATLAB, refer to Chapter 6.

## 4.2.3 Python

As mentioned in Section 4.1.1, setpoints have to be sent at least every 0.5 s to keep the flight controller in offboard control. Therefore, the python script was programmed to send the setpoints at a frequency of 5 Hz ensuring that offboard control is maintained. The main reason the continuous setpoint publisher was chosen to be on the Intel Edison is because WiFi was used as a communication medium. It is therefore possible to create the same publisher using a MATLAB script, but this can cause the flight controller to go into a failsafe mode as soon as the connection between MATLAB and the Intel Edison is broken. However, if the multicopter goes out of WiFi range while using the python script it is

able to stay at the same position until control is retaken by the pilot or the WiFi connection is regained.

The failsafe mode was created by Pixhawk to give control of the multicopter back to the pilot. This is done by switching from offboard control back to the mode the remote is in as soon as the setpoints are not received fast enough. That is why the remote was always set to position control before offboard control was initiated. This served as a second backup system to ensure that the multicopter did not crash if communication was somehow broken between the controller and the companion computer. In this case, position control ensured that the multicopter stayed at its current position until the pilot was able to manually take over or change the position setpoint with the remote.

#### 4.2.4 Proximity Sensor Design

After the necessary ROS packages (Section 4.2.1) were extracted from the ROS system on the Intel Edison, it was incorporated into the Arduino IDE (Integrated Development Environment). This enabled the Arduino Mega (Section 4.1.3) to connect with the Intel Edison (Section 4.1.2) using a ROS communication protocol. These packages also allowed the Arduino to create its own node on a ROS system. The proximity sensor can therefore be connected to any ROS system, allowing that system access to the data generated.

Since a ROS communication protocol was established, a ROS LaserScan message had to be used to transfer the data. From Table 4.1 it can be seen that this message requires an array of distance measurements as well as the angle increment between these measurements. The angle increment is however not a matrix, but only one value. Therefore, the measurements taken would have to be evenly divided around the entire measurement area ( $360^\circ$ ). This, however, posed a problem as an mostly open environment would cause the measurements to be distributed incorrectly. With this in mind, the rotational setup (Figure 5.1 in Chapter 5) was used to break the environment into 18 sections. Each section corresponding to a gear tooth. The measurements taken in each section were then evenly distributed in that section as the exact angle of the measurement was not known. After each section, the new values are stored in the LaserScan message and sent to the LIDAR topic. A flow diagram of the program can be seen in Figure 4.6.

It should be noted that nearby objects will produce more measurements in a given time and therefore give a better representation of the environment. This is due to the measurements distance being time dependant (Section 4.1.3). Therefore, if objects are located far away, fewer measurements will be taken. To compensate for this, the maximum time the sensor was allowed to wait for a measurement was set to 10 ms i.e. a maximum distance of 10 m. Since the sensor was designed to do a full rotation in 1 s, each section takes about 56 ms to measure. With the programmed time restraint, a theoretical minimum of

Table 4.1: ROS LaserScan Message type: Message data and definitions. All the data in this message was continuously collected and published by the proximity sensor setup described in Section 4.1.3 and Section 4.2.4.

Message Data	Data Definition
Header	Time stamp Frame ID
Min Angle [rad]	Start angle of the scan
Max Angle [rad]	End angle of the scan
Angle Increment [rad]	Angle between measurements
Time Increment [s]	Time between measurements
Scan Time [s]	Time between scans
Min Range [m]	Minimum distance sensor can scan
Max Range [m]	Maximum distance sensor can scan
Ranges [m]	Distance Measurements
Intensities	Device specific units, if not provided, leave empty

five measurements can be taken in one section if all the measurements are taken at a maximum distance of 10 m.

To start and stop the sensor at will, a button was added to the system. The sensor, however, always had to be manually readjusted to the correct starting position before it was enabled. This was done to ensure that the correct minimum and maximum angles (Table 4.1) were given as these angles are essential to the map building process. These angles were calculated relative to the starting section and had to be recalculated for each new section. Therefore, the correct starting position for the sensor was imperative.

### 4.2.5 Hardware in the Loop

Hardware in the loop (HITL) simulators replace the vehicle and the environment with a simulation. These simulations have high precision dynamic models of the vehicles while the environment can mimic wind, turbulence and obstacles. Therefore, the sensor data perceived by the flight controller is generated by the simulation and sent to the controller. The control outputs are then sent back to the simulation and displayed, giving an accurate representation of how the vehicle would react in certain scenarios. jMAVSim, a lightweight multirotor simulator (Figure 4.7) was used to extensively test all the connections and avoidance algorithms before actual flights were completed. This was mainly done to prevent unnecessary crashes and to ensure that the multicopter reacted accordingly.

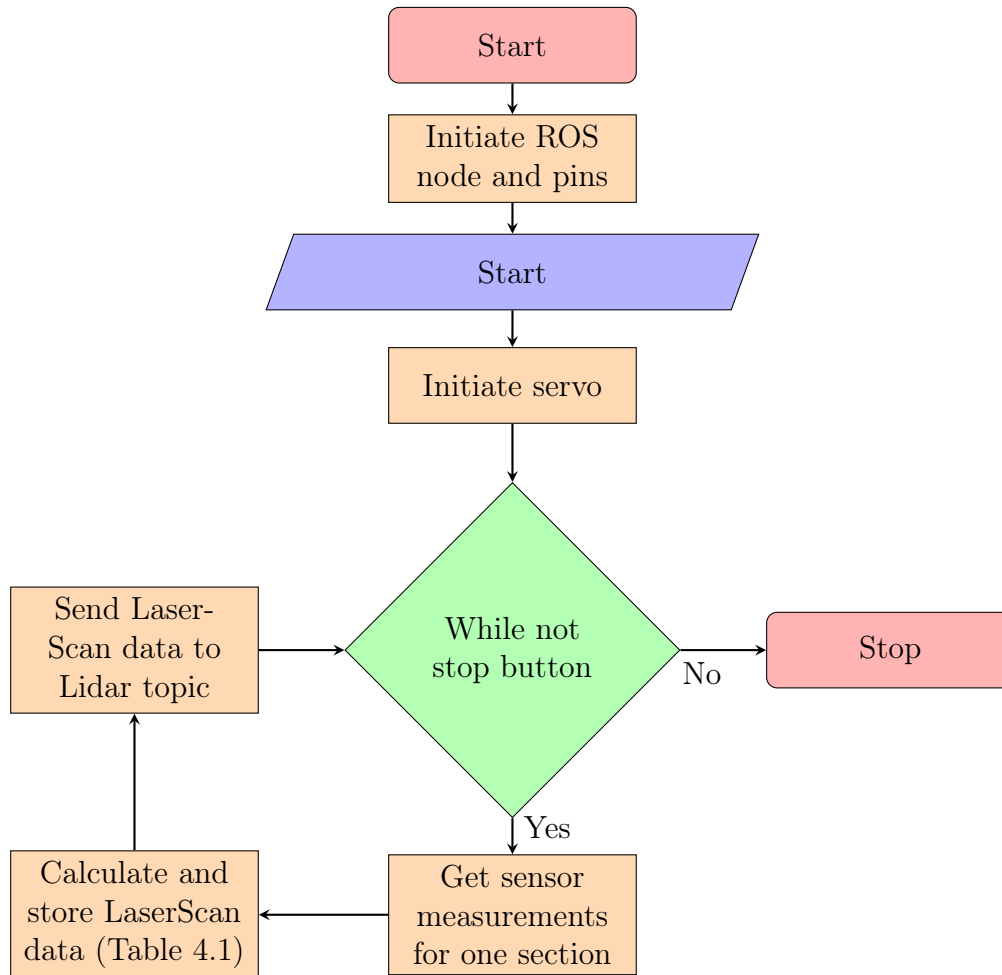


Figure 4.6: Flow diagram of the proximity sensor code.

### 4.3 Conclusion

In this chapter the hardware and software used to obtain a system capable of making a multicopter fly autonomously are investigated. An overview of the entire system can be seen in Figure 4.1. To get a working obstacle avoidance system, the Pixhawk flight controller (Section 4.1.1) had to be connected to a companion computer (Section 4.1.2). The companion computer (Intel Edison) acted as both a companion computer and a data relay between the ground station and all the other components connected to it, including the proximity sensor (Section 4.1.3 and Section 4.2.4). Test results regarding this sensor can be found in Chapter 5.

To communicate between all the different devices (Arduino, Intel Edison, Pixhawk and desktop) and programming languages (MATLAB (Section 4.2.2), Python (Section 4.2.3) and Arduino), Robot Operating System (Section 4.2.1) was used. This open-source, meta-operating system allowed seamless integra-

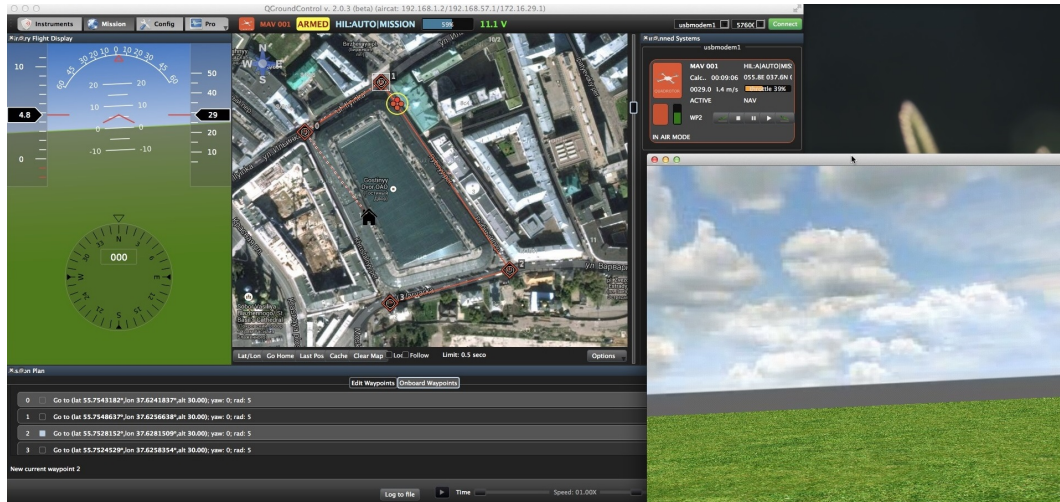


Figure 4.7: jMAVSim HITL interface (jMA, 2015).

tion between all the mentioned components and allowed MATLAB to fully control the multicopter. All the new code and connections were extensively tested with hardware in the loop (HITL) software (Section 4.2.5) before flight test were done to minimize crashes and to ensure the code as well as the multicopter reacted properly.

Since WiFi was used to relay all the data, two extra safety precautions were introduced to ensure the safety of the multicopter (Section 4.2.3). To ensure the multicopter does not switch to failsafe mode as soon as the WiFi connection is broken, a python script was added to the companion computer. This enabled the controller to stay in one position till the pilot took over. To compensate for the failsafe mode (i.e. when the connection between the companion computer and controller was broken), the remote should always be switched to position control before offboard control is enabled. Therefore, when the connection is broken, the controller will switch back to communicating with the remote, enabling position control and ensuring the safety of the multicopter.

# Chapter 5

## Proximity Sensor

In Figure 5.1, the sensor setup used for the flight tests can be seen. This sensor was designed to be able to work as its own unit. Therefore, corresponding with the modular design of the entire system (Chapter 4). When connected to a ROS system, the sensor is able to create its own node and publish data, granting the system access to the information generated (Section 4.2.4). The way the sensor is designed enables it to send measurements 18 times per second. The amount of measurement taken in this time, however, depends on the distance to the objects. Objects that are placed far away will receive less measurements than object in close proximity as the measurements are time dependant (Section 4.1.3). In Section 5.1 test results regarding the sensor is discussed. This is followed by an conclusion regarding the test results in Section 5.2.

### 5.1 Test Results

For the first test, a small environment was chosen as can be seen in Figure 5.2. From the results it can be seen that it is possible to reproduce the environment from the data received by the sensor. In Figure 5.2b a uniform map of the environment is given, illustrating that the small uncertainty in the measurement distance does not significantly affect the map building process.

For the second test, a bigger room was scanned as can be seen in Figure 5.3. With the information received from the proximity sensor, it was again possible to reconstruct the environment fairly accurately. In Figure 5.3b a uniform map of the environment is given, illustrating that the uncertainty in larger measurements still does not significantly affect the map building process. By comparing this test with the previous one, it is possible to see that the data is more spread out as would be expected since the distance measurements are time dependant.

For the last test (Figure 5.4) an even bigger room was scanned to determine what effect it will have on the map building techniques output (Figure 5.4b).

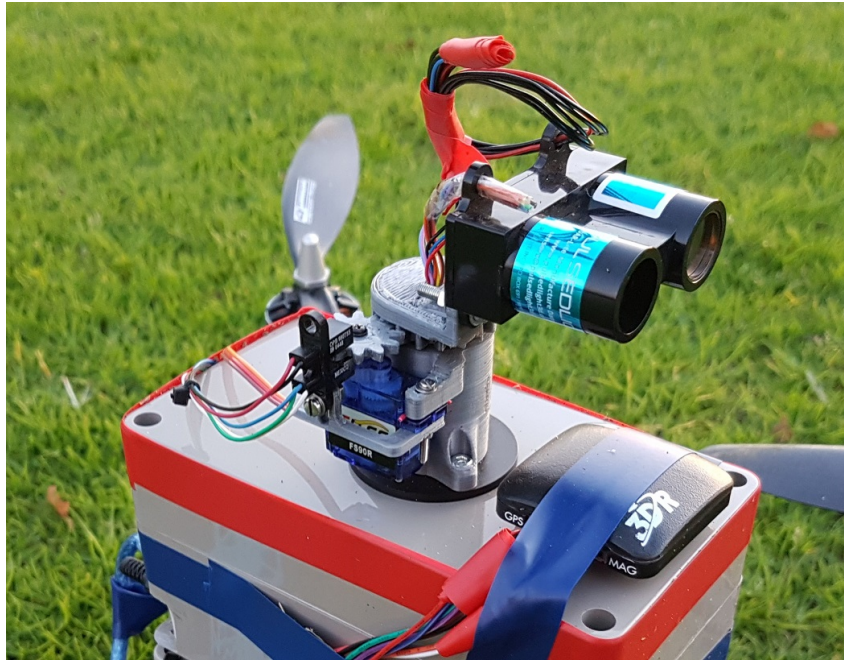
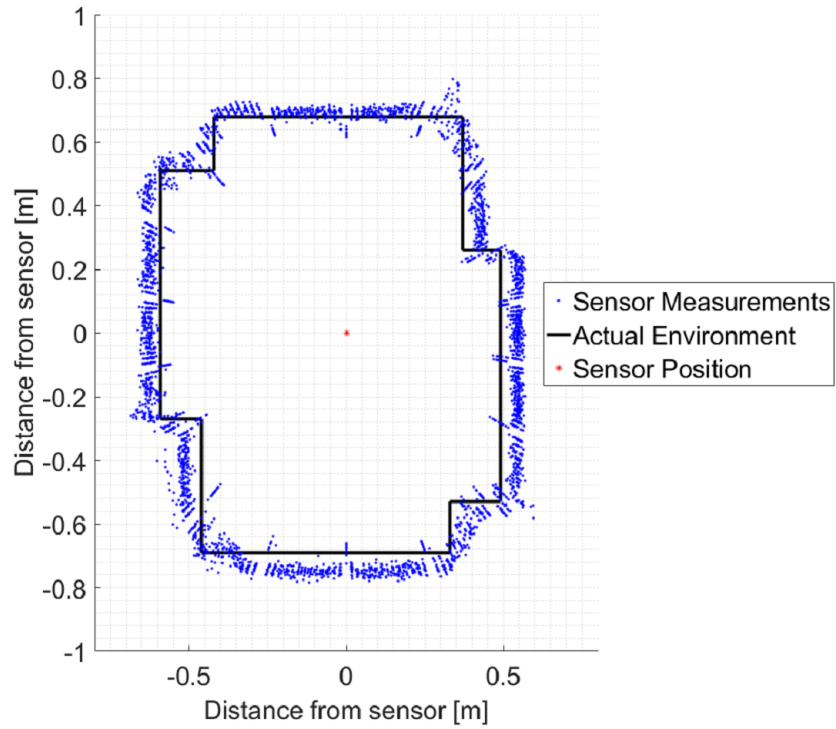


Figure 5.1: Sensor Setup

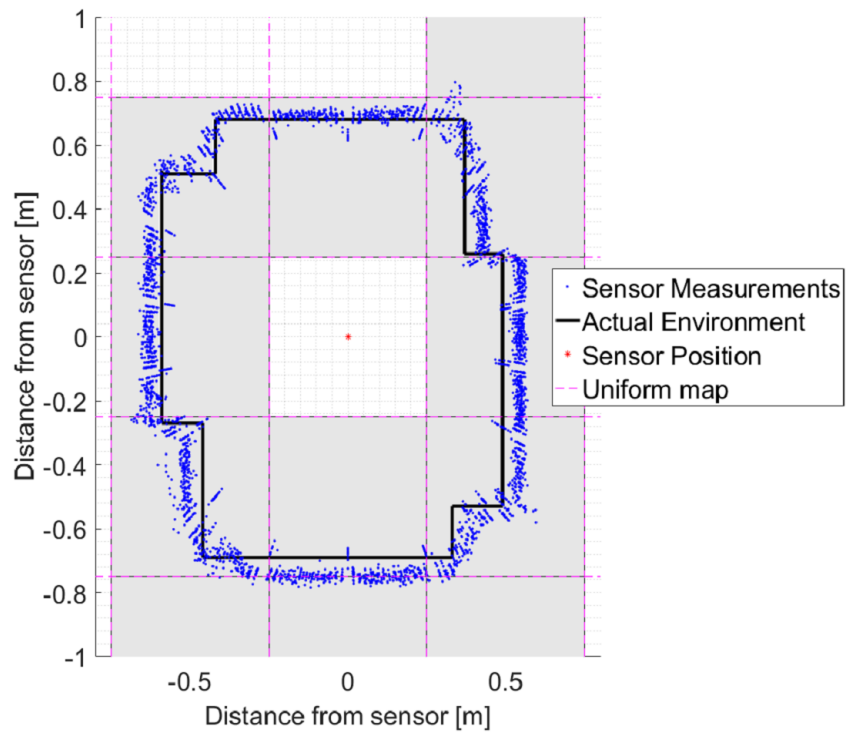
As would be expected, obstacles close to the sensor are represented better compared to obstacles farther away. However, even though the data is more scattered than the previous test results, it still builds a fairly accurate representation of the environment (Figure 5.4b).

One thing that can be seen from all the tests is that there are some measurement points standing out from the rest. These points are generated from an assumption made during the design process of the sensor. If Figure 5.2 is carefully investigated, it can be seen that these points creates 18 sectors. Therefore, because it was assumed that the measurements are equally spaced in each section, some of the measurements are placed on the edges. The reason these measurements are not in line with the others is because the ROS message has to be generated and published at the end of each section. During this time, no measurements are taken, however, the sensor is still rotating. Therefore, when it starts measuring again, it is not at the beginning of the section anymore. When the new message has to be generated, the scanned measurements are spread across the entire section. Because the first measurement would have to rotate back the most from its actual measured angle, it is easily noticeable. To explain why the measurements appear scattered in straight lines, the same reasoning can be used as above. In other words, since the actual measurement angles may vary each time, the angle the data is rotated by is also affected, causing the data to appear scattered in a straight line.



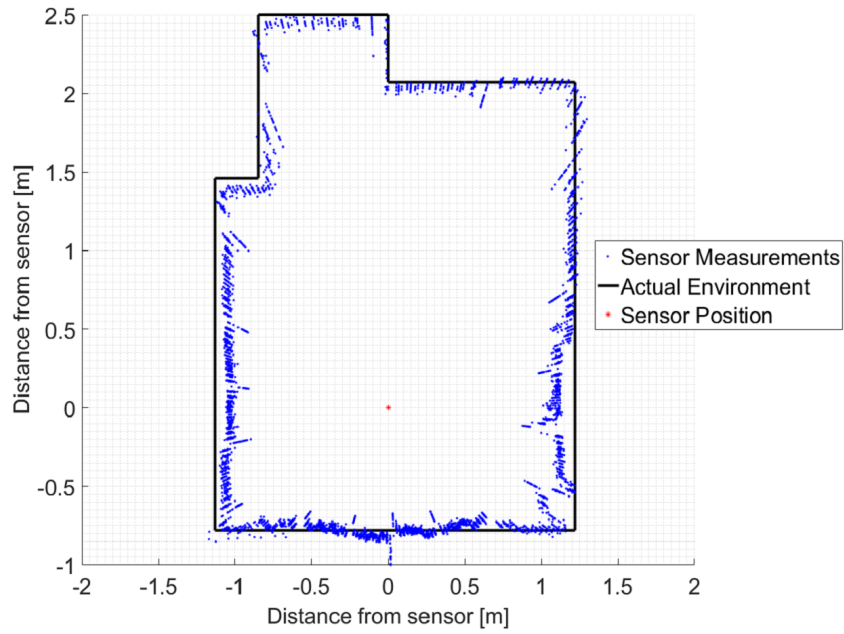


(a)

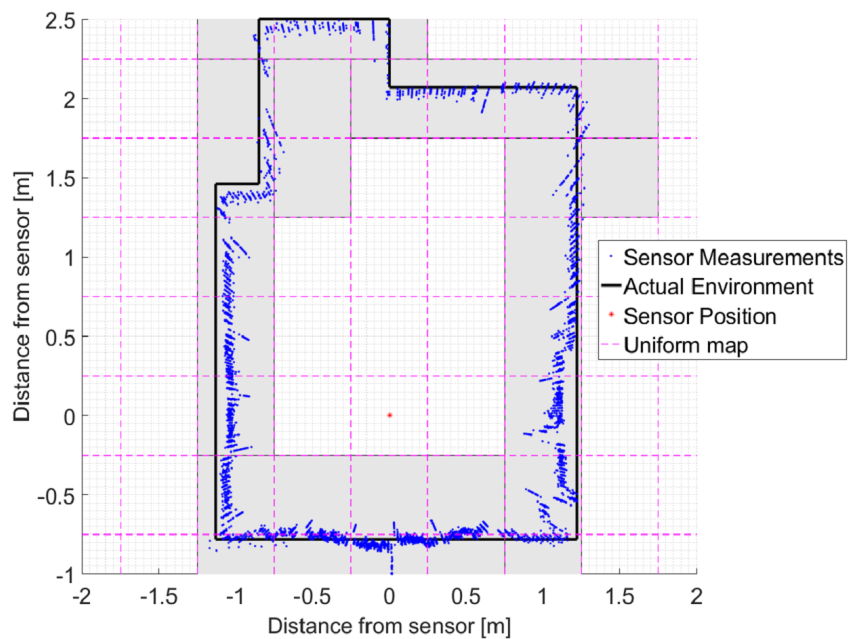


(b)

Figure 5.2: First scanned environment. In (a) the actual environment as well as the scanned environment can be seen. In (b) it is illustrated how the scanned environment would be represented by the uniform map build technique (gray).

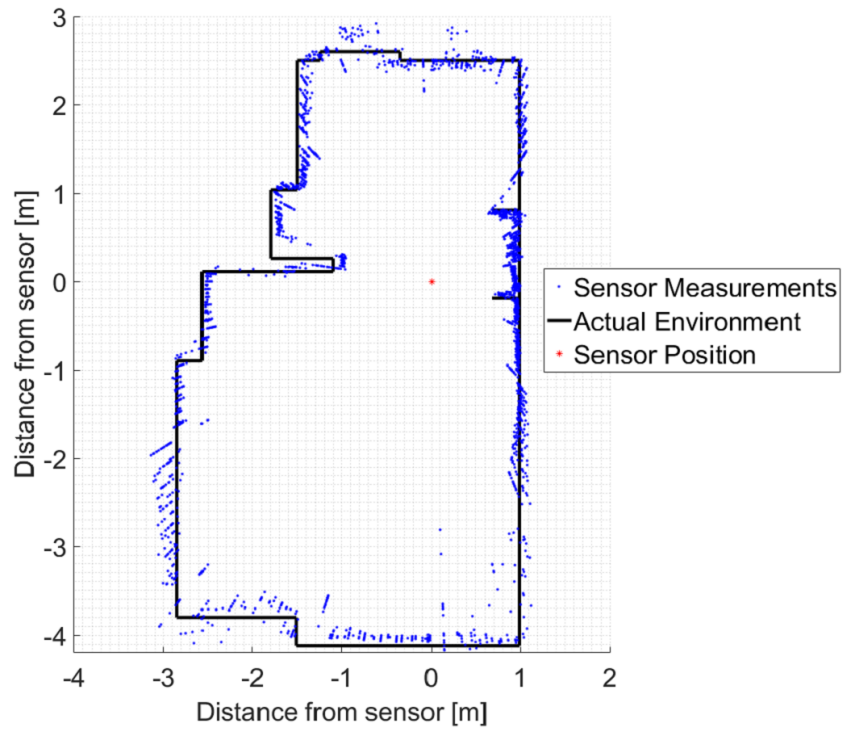


(a)

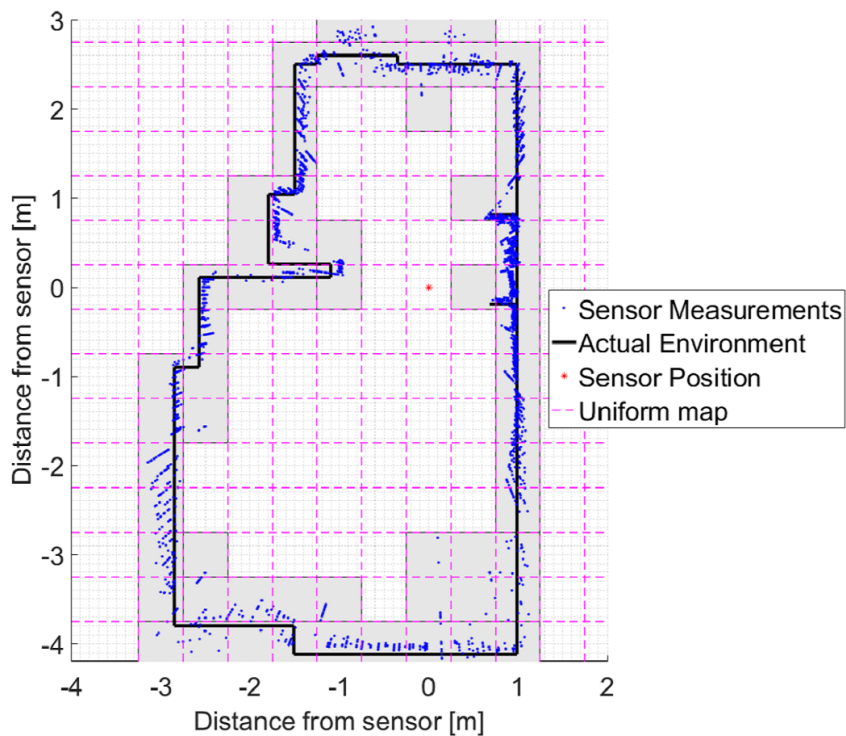


(b)

Figure 5.3: Second scanned environment. In (a) the actual environment as well as the scanned environment can be seen. In (b) it is illustrated how the scanned environment would be represented by the uniform map build technique (gray).



(a)



(b)

Figure 5.4: Third scanned environment. In (a) the actual environment as well as the scanned environment can be seen. In (b) it is illustrated how the scanned environment would be represented by the uniform map build technique (gray).

## 5.2 Conclusion

From the test results in Section 5.1 it is possible to see how the designed proximity sensor would react in different environments. It was found that even though the amount of data points may vary for each scanning section, depending on the distance of the obstacle in that section, it is still possible to reconstruct a fairly accurate map of the environment using uniform grids. Therefore, with regards to the results obtained from these test and others, the designed proximity sensor was deemed acceptable for implementation alongside the designed obstacle avoidance system.

## Chapter 6

# Obstacle Avoidance Implementation

In Chapter 2, the background of different map building techniques and obstacle avoidance algorithms are discussed. From this investigation, a combination of three different techniques were used to obtain a working obstacle avoidance algorithm. These techniques: the uniform occupancy grid, the Virtual Force Field (VFF) and D\* Lite were algorithmically integrated as described in Section 6.1. With this combined obstacle avoidance algorithm in place, extensive simulations were done in MATLAB (Section 6.2). The aim of these simulations was to ensure that the algorithm reacted as intended and to debug any unforeseen errors in the code. The results of these simulations can be found in Section 6.3. After the code was extensively tested on MATLAB, Hardware in the Loop (HITL, Section 4.2.5) simulations were done to ensure the multicopter reacted appropriately to the commands received from MATLAB. Real time flight tests were done with a multicopter only after these simulations were successfully completed. The test setup and the results for these flights can be found in Chapter 7.

### 6.1 Algorithm Integration

To be able to fly autonomously in a partially known (or unknown) environment, it was decided to use a combination of three different techniques discussed in Chapter 2. These techniques are the uniform occupancy grid map building algorithm (Section 2.1.1), D\* Lite (Section 2.2.4) and the VFF (Section 2.2.1) method.

To obtain a map of the environment, the probability map building technique (Equation 2.10) was used alongside the ideal-like inverse sensor model (Equation 3.1) derived in Section 3.3. With a constantly updating map of the environment available, D\* Lite could be used as a global path planner. In a static environment, this algorithm will only have to compute the path once.

However, in a dynamic or unknown environment, this algorithm will have to be executed continuously until the goal position is reached. Even though the path may stay the same, the algorithm must still check whether the addition or removal of obstacles affects the previously planned path before every move. This continuous re-checking and planning can get computationally intensive. One way of limiting this is by only re-checking and planning every few steps. However, if an obstacle is scanned that blocks the planned path to the goal while D\* Lite is not active, the multicopter will fly straight into it. For this reason it was decided to combine D\* Lite with the VFF method.

By combining these two algorithms, D\* Lite only has to be executed every few steps since the VFF ensures the safety of the multicopter while D\* Lite is not active. In other words, D\* Lite plans a path from the multicopters' current position to the goal position. The multicopter will then move along the calculated path and update the map as usual. If an unexpected obstacle appears on this path, the map will be updated accordingly while the VFF ensures the safety of the multicopter by maintaining the necessary safety distance. The multicopter is then kept at that distance until D\* Lite is initiated again to plan a new path.

When combining the VFF and D\* Lite as described above, the VFF method can still get stuck in local minima. This is due to D\* Lite being able to plan through denser packed obstacles than the VFF can move through. To compensate for this, a safety distance was added to the D\* Lite algorithm. This enabled the algorithm to ignore vertices that are too close to known obstacles. To ensure that local minima is not a problem anymore, the safety sphere for both the VFF and D\* Lite have to be the same size at all times.

## 6.2 MATLAB Simulation Setup

Before the combined algorithm discussed in Section 6.1 is simulated (Section 6.2), the heuristic and cost functions used in D\* Lite are investigated (Section 6.2.1). This is done to ensure they adhere to the conditions set out in Chapter 2. The algorithmic formulation and implementation of the VFF alongside D\* Lite is also investigated (Section 6.2.2).

### 6.2.1 D\* Lite

As mentioned in Chapter 2, the heuristic function for D\* Lite must adhere to the forward-backward consistent condition. That is, it has to adhere to  $h(s, s'') \leq h(s, s') + h(s', s'')$  for all  $s, s', s'' \in S$ . The octile distance heuristic

function was therefore chosen, and is given by:

$$\begin{aligned}
 &\text{function } heuristic(vertex_1, vertex_2) = \\
 &\quad d_x = \text{abs}(vertex_1.x - vertex_2.x) \\
 &\quad d_y = \text{abs}(vertex_1.y - vertex_2.y) \\
 &\text{return } D_1 * (d_x + d_y) + (D_2 - 2 * D) * \min(d_x, d_y)
 \end{aligned} \tag{6.1}$$

where  $D_1 = 1$  and  $D_2 = \sqrt{2}$ . To test if the function adheres to the condition set out above, a small 5 by 5 matrix is tested in Table 6.1. The second term in the inequality,  $h(s, s')$ , is represented in the left hand table. This represents the heuristic value from the start vertex ( $s$ ), indicated in green, to any of the given vertices ( $s'$ ). The third term in the inequality,  $h(s', s'')$ , represented in the middle table, gives the heuristic value from any of the vertices ( $s'$ ) in the grid to the end vertex ( $s''$ ), indicated in red. The first term in the inequality,  $h(s, s'')$ , is the heuristic value from the start vertex ( $s$ ) to the end vertex ( $s''$ ). This value is 5.7, as seen in the left hand table. The sum of the first two tables can be found in the right hand table. The values in this table represent the heuristic value if the robot went from the start vertex (green) to any of the other vertices and then to the end vertex (red). From this it can be seen that the octile distance heuristic function always adheres to  $h(s, s'') \leq h(s, s') + h(s', s'')$ .

Table 6.1: Octile distance heuristic function inequality test. In the left had table, the heuristic value is given from the start vertex (green) to any of the other vertices. In the middle table, the heuristic value is given from any of the vertices in to the goal vertex (red). In the right hand side table, the heuristic value is given if the vehicle moves from the start vertex (green) to any of the other vertices and then to the end vertex (red). From these tables it can be seen that the octile distance heuristic function adheres to the given inequality.

$h(s, s')$					$h(s', s'')$					$h(s, s') + h(s', s'')$				
0	1	2	3	4	5.7	5.2	4.8	4.4	4	5.7	6.2	6.8	7.4	8
1	1.4	2.4	3.4	4.4	5.2	4.2	3.8	3.4	3	6.2	5.7	6.2	6.8	7.4
2	2.4	2.8	3.8	4.8	4.8	3.8	2.8	2.4	2	6.8	6.2	5.7	6.2	6.8
3	3.4	3.8	4.2	5.2	4.4	3.4	2.4	1.4	1	7.4	6.8	6.2	5.7	6.2
4	4.4	4.8	5.2	5.7	4	3	2	1	0	8	7.4	6.8	6.2	5.7

As mentioned in Chapter 2, the vertices also need to be acceptable no matter where the goal vertex is. In other words, they have to obey  $h(s, s') \leq c^*(s, s')$  for all  $s, s' \in S$ ; where  $c^*(s, s')$  is the cost of the shortest path from  $s$  to  $s'$ . If, however, the cost is calculated in the same manner as the heuristic function, it will always adhere to  $h(s, s') \leq c * (s, s')$ . Therefore, the cost and the heuristic function can easily be demonstrated with Figure 6.1. This illustrates the cost that will be added, depending on the chosen route. In this case, the added cost equals the distance the robot would have travelled if all

the grid cells had a size of 1. To move forward, backward or sideways would therefore have an added cost of 1. However, to move diagonally, the added cost will be  $\sqrt{2}$ .


$\sqrt{2}$	1	$\sqrt{2}$
1		1
$\sqrt{2}$	1	$\sqrt{2}$

Figure 6.1: Cost and heuristic function illustration. When the multicopter moves forwards, backwards or sideways the added cost will be 1. Whereas if it moves diagonally, the added cost will be  $\sqrt{2}$ . In other words, this diagram is used to calculate the  $rhs(s)$  values of D\* Lite. However, the octile distance heuristic function also uses a similar method of calculating the distance between vertices.

## 6.2.2 Virtual Force Field

By using the gradient method alongside the VFF (as described in Chapter 2), a path can be generated by checking the neighbouring values at each cell and continuously moving to the cell with the lowest  $F_{\text{total}}$  value. Where  $F_{\text{total}}$  is given by:

$$F_{\text{total}}[i, j] = F_{\text{att}}[i, j] + F_{\text{rep}}[i, j] \quad (6.2)$$

where

$$F_{\text{att}}[i, j] = \sqrt{(x - x_{\text{goal}})^2 + (y - y_{\text{goal}})^2} \quad (6.3)$$

and

$$F_{\text{rep}}[i, j] = \begin{cases} 0 & \text{if } r \geq r_{\text{safe}} \\ 2^{(r_{\text{safe}} - r)} & \text{if } r < r_{\text{safe}} \end{cases} \quad (6.4)$$

In Equation 6.4, the distance of a node to the nearest obstacle is given by  $r$ , where  $r_{\text{safe}}$  is the size of the safety boundary generated by the VFF. In Equation 6.3, both  $x$  and  $y$  represent the position of the node being investigated whereas  $x_{\text{goal}}$  and  $y_{\text{goal}}$  is the temporary attraction point generated by D\* Lite. The matrix indices  $i$  and  $j$ , given in Equation 6.2, represent the map position being investigated to determine its resultant force value.

This algorithm was implemented by only calculating the resultant force of the multicopters' current node along with its immediate surrounding nodes (3 by 3 area). Since the proximity sensor is able to detect objects at a distance of up to 10 m (Section 4.2.4), the distance to the nearest obstacle ( $r$ ) can easily be determined for all these nodes. This value is then compared to the safety



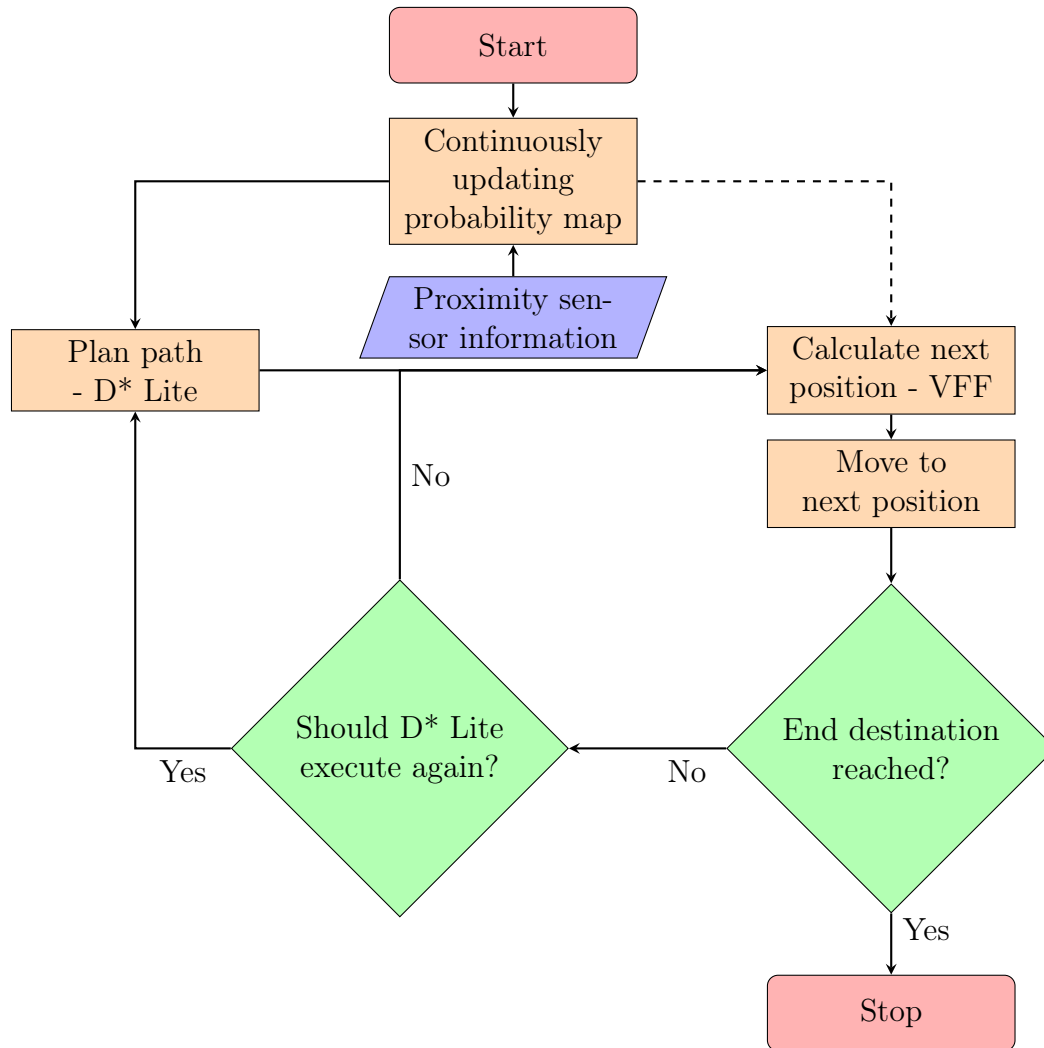


Figure 6.2: Combined avoidance algorithm flow chart.

boundary ( $r_{\text{safe}}$ ) generated by the VFF to calculate a corresponding repulsive force (Equation 6.4) for each node. As the position  $(x, y)$  of each node is also known, along with the temporary attraction points, the attractive force (Equation 6.3) for each node can be calculated. With both the attractive and repulsive forces known, the resultant force (Equation 6.2) can be calculated. From the resultant force, the next position the multicopter should fly to can be determined by finding the node with the smallest  $F_{\text{total}}$  value. This process is continuously executed alongside D\* Lite, as indicated in the flow chart seen in Figure 6.2, until the end destination is reached.

### 6.3 MATLAB Simulation Results

With the three algorithms combined as illustrated in Figure 6.2, it was possible to simulate the avoidance algorithm in different conditions. Both static and dynamic environments were simulated to see whether the avoidance algorithm reacted accordingly. Different map sizes were also simulated to determine what effect the size of a map will have on the computational time of both D\* Lite and the VFF.

Figure 6.3a illustrates a dynamic simulated environment. From this figure it can be seen how obstacles were both added and removed from the environment. Each time a new path was calculated, temporary attraction points were made along the path enabling the VFF to pull the multicopter to the goal position. The nodes evaluated to find the path are given in gray. Each time a new path was calculated, only some of the nodes had to be recalculated as D\* Lite reuses all the nodes' information that was not affected by the new obstacles. Different static environments (Figure 6.3b and Figure 6.4) were also investigated to determine how the path planning algorithm (D\* Lite) reacts to different environments.

After conducting a thorough testing process simulating a wide variety of test conditions for both D\* Lite and the combined algorithm (of which Figure 6.3 and Figure 6.4 show only a few), it was found that the programmed algorithms reacted as intended. D\* Lite was therefore always able to calculate or re-calculate a path in both a static and dynamic environment, unless none existed. With the known path, the multicopter was pulled to the goal position by the VFF while it ensured the safety of the vehicle until the path could be re-evaluated by D\* Lite. As expected, the safety boundary added to D\* Lite ensured that the VFF could not get stuck in local minima anymore.

To determine what effect the different map sizes have on the time it takes to calculate a path, D\* Lite was simulated under different conditions. It was decided to calculate a path diagonally across a map. Secondly, each map size and random arrangement of obstacles were tested 200 times. Therefore, one map size was simulated 200 times with a given obstacle density, for example 1 %. However, for each run, a different random arrangement of obstacles was used. For an example of a 750 by 750 map with an obstacle density of 5 % (28125 obstacles), refer to Figure 6.4.

The simulations were done in MATLAB R2016a on a computer with a 2.4GHz Intel Core i7 processor and 16GB of memory and the results can be seen in Figure 6.5. This figure illustrates the average time it took to successfully calculate a path from the starting position to the goal position. As would be expected, the average time increased exponentially with the size of the map. The time also increases with the obstacle density, as it is usually more difficult to plan a path with more obstacles present, depending on how they are placed. Figure 6.6 illustrates how scattered the timing data was for the 200 D\* Lite simulations done on the 750 by 750 map with an obstacle density of 5 %. This

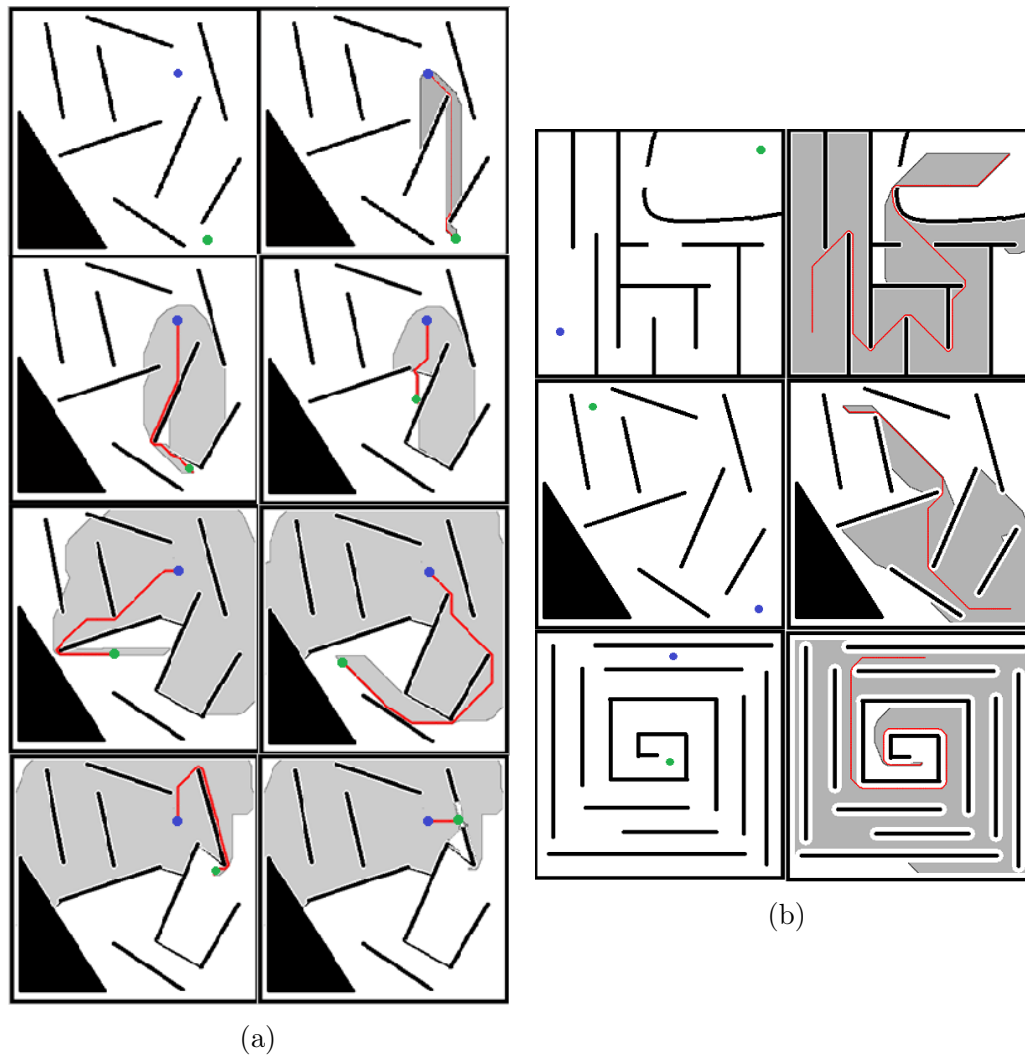


Figure 6.3: MATLAB simulations of static (b) and dynamic (a) environments. Red indicates the calculated path whereas gray indicates the nodes evaluated to obtain the path. The green and blue dots indicate the position of the robot and the goal position respectively. Figure (a) illustrates how a combined algorithm would move and plan in a dynamic environment (left to right, top to bottom). Whereas Figure (b) illustrates the path planned by D\* Lite for three different static environments.

shows that the obstacle positioning also has an effect on the time taken. In short, these results indicate that the time it takes to plan a path depends on the map size, obstacle placement and the obstacle density if the start and goal positions stay unchanged.

After a path is planned, the VFF always takes less than 4 ms to execute i.e. plan the next position for the multicopter. This time stays relatively constant as only 9 resultant forces have to be calculated each time the algorithm is initiated. The obstacle density and placement also doesn't have a significant

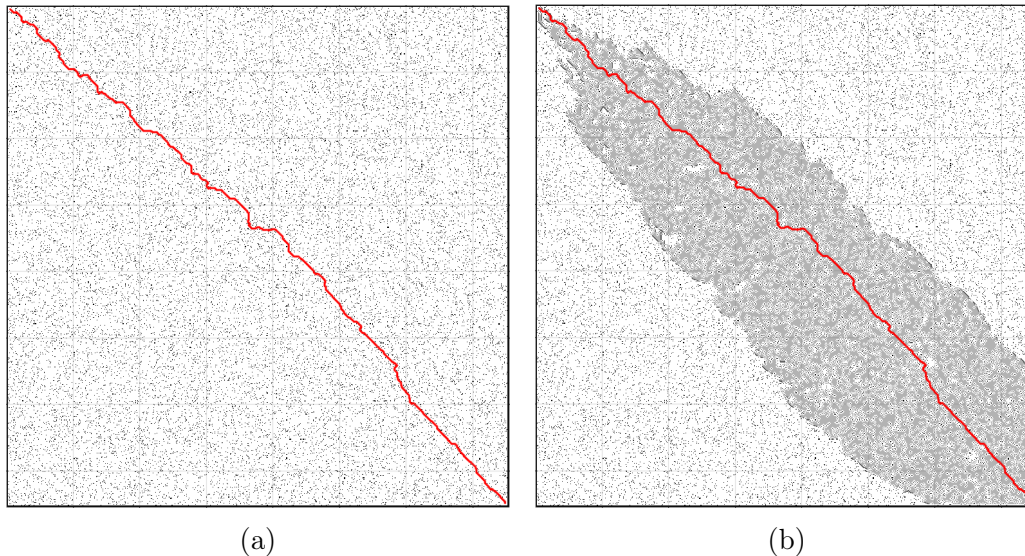


Figure 6.4: D\* Lite path planning simulation of a 750 by 750 map (static environment) with a 5 % obstacle density. All the obstacles are given in black and D\* Lite was executed with a safety distance of 2. In Figure (a) the planned path can be seen. In Figure (b) the same planned path is shown along with the vertices evaluated to plan the path given in gray.

effect as the distance to the nearest obstacle ( $r$ ) can easily be obtained by only investigating the nodes in range of the proximity sensor. Therefore, a constant window size around the multicopter is investigated for obstacles each time the algorithm is initiated (Default cell size: 0.5 m, Proximity sensor range: 10 m).

## 6.4 Conclusion

In this chapter, the algorithmic combination of the occupancy grid map building technique, D\* Lite and the Virtual Force Field (VFF) method is explained (Section 6.1). This is followed by an investigation of the heuristic and cost function used for D\* Lite (Section 6.2.1), as well as the formulation and algorithmic implementation of the VFF (Section 6.2.2). Simulation results of both static and dynamic environments are then illustrated and discussed along with the computational time of D\* Lite and the VFF method (Section 6.3).

By combining the three algorithms mentioned above, it was possible to obtain an algorithm capable of planning a global path while still being able to react to local disturbances (Figure 6.2). To achieve this, however, a safety boundary had to be added to the D\* Lite algorithm. This boundary enables the algorithm to ignore vertices that are too close to known obstacles. Therefore, to ensure local minima are not a problem, the two safety boundaries should always be the same. By combining the VFF and D\* Lite, temporary goal positions are set along the planned path while the multicopter moves, pulling

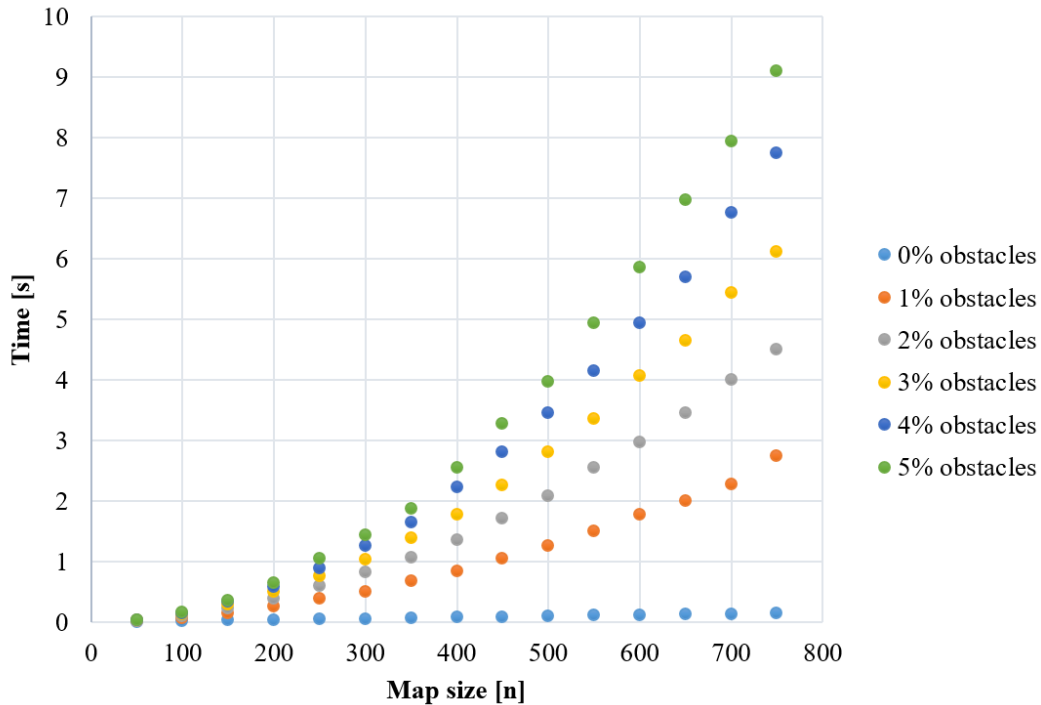


Figure 6.5: D\* Lite execution time.

it to the end destination and allowing the VFF to navigate the environment without getting stuck in local minima. This also enables D\* Lite to be less computationally intensive as it only has to re-evaluate the path every few steps.

To ensure D\* Lite adheres to both conditions set out in Chapter 2, the heuristic and cost function used in the algorithm was investigated. For the heuristic function it was found that the octile distance heuristic function (Equation 6.1) adheres to the forward-backward consistent condition as proven in Section 6.2.1. To ensure the cost function adhered to its condition, it has to be calculated in the same manner as the heuristic function to calculate the added cost of moving from one node to the next. The concept of the added cost is also illustrated in Figure 6.1.

By combining the VFF formulas given in Equations 6.2, 6.3 and 6.4 with the gradient method described in Chapter 2, it is possible to calculate the multicopters' next position by only evaluating its current node along with the immediate surrounding nodes i.e. a 3 by 3 area. The next position will then be the node with the smallest resultant force given by Equations 6.2.

By simulating the D\* Lite and the combined algorithm for both static and dynamic environments, it was found that the algorithms executed as intended. D\* Lite was therefore always able to calculate and re-calculate a path in both a static and dynamic environment, unless none existed. While moving along the path, the VFF was able to ensure the safety of the multicopter when D\* Lite was not active. Different map sizes, obstacle placement and obstacle

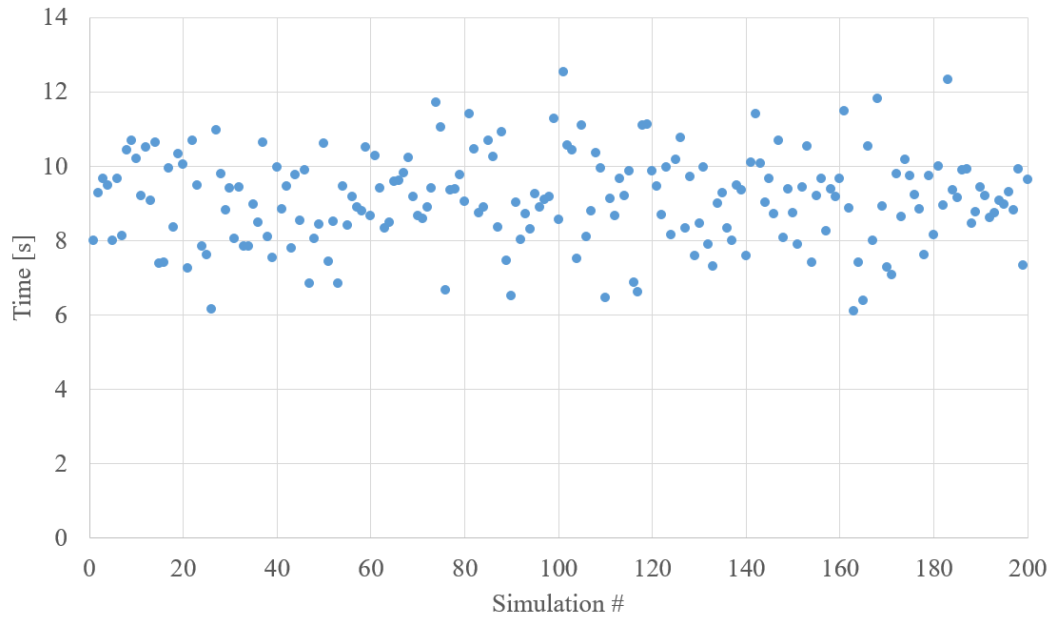


Figure 6.6: D\* Lite execution time in 750 by 750 map with different obstacle configurations each time. Obstacle density stayed constant at 5%.

densities were then tested to see the effect they have on the simulation time. It was found that the average execution time of the D\* Lite algorithm grew exponentially with the map size (Figure 6.5). Added obstacles also increased the execution time, as would be expected. From Figure 6.6 it was however found that the execution time mainly depended on the map size, obstacle placement as well as the obstacle density if the start and goal positions stay unchanged. The time to execute the VFF was found to be less than 4 ms each time. This stayed relatively constant as the same amount of nodes had to be investigated each time to determine the multicopters' next position.

# Chapter 7

## Flight Tests

In Chapter 3, pose and measurement uncertainties are investigated. This was incorporated into simulations and it was found that the measurement uncertainty does not affect the map building procedure as the uncertainty in the measurement is  $\pm 2.5$  cm where the default grid size that has to be updated is 0.5 m. Therefore, an ideal-like sensor model was incorporated alongside the probability map building formula as indicated in Chapter 6.

It was also found that the pose uncertainty could only be taken into account if it is readily available in PDF (Probability Density Function) form. Unfortunately, this data is not available while flying the multicopter, but can be generated afterwards if a Real Time Kinematic (RTK) GPS is added to the flight setup. The data generated by this GPS can then be used as a ground truth measurement to determine the error made by the flight controllers position estimation. However, since the data is not available while flying, the effect of the pose uncertainty on the map building process was investigated through simulation. It was found that the map being built will be in constant flux as the error made by the position estimation of the flight controller is constantly changing. But since the map is being built relative to the multicopters' current position estimate, it can be assumed safe to travel.

To enable the multicopter to fly autonomously with the help of MATLAB, a combination of different hardware and software were integrated as discussed in Chapter 4. This included the proximity sensor (Chapter 5) used to obtain the data needed to build a map of the environment. To plan a collision free path through different environments, however, three algorithms were combined (Chapter 6). These include the uniform occupancy grid map building technique, the VFF (Virtual Force Field) method and D\* Lite. Before flight tests were done with this algorithm, extensive simulation tests were done in both MATLAB (Section 6.3) as well as jMAVSim (Section 4.2.5).

The relevant assumptions made for the flight tests are evaluated in Section 7.1. This is followed by the flight test setup in Section 7.2 as well as the results and a discussion in Section 7.3. Section 7.4 then concludes this chapter with a summary of the results and a conclusion.

## 7.1 Assumptions

As obstacle avoidance will only be implemented in a two dimensional environment, there are certain assumptions that need to be made. Firstly, the orientation of the multicopter will stay constant for the entire flight period. Therefore, the yaw angle of the multicopter can be assumed to be constant the entire flight and will not affect the measurements received from the proximity sensor. Secondly, it is assumed that the sensor is mounted correctly on the flight setup, resulting in no additional angle uncertainty. Thirdly, when flying, the multicopter stays horizontal. Therefore, the pitch angle does not affect the sensor measurements. Fourthly, it is assumed that the multicopter will fly to a specified height and remain at that height until the goal position is reached. Included in this assumption, is the fact that there are no obstacles at either the liftoff or the landing site to obstruct the multicopters' movements. Lastly, since the orientation and the height of the multicopter is assumed to be constant throughout the entire flight, only the horizontal position of the multicopter will be measured. The data from the RTK GPS is therefore assumed to be the actual position of the multicopter and will be compared to the position estimation of the multicopters' controller to determine the position error.

## 7.2 Outdoor Test Setup

In Figure 7.1, the multicopter used for the outdoor flight tests is shown. To prove the multicopter is able to fly autonomously, two scenarios had to be proven:

1. That the multicopter is capable of planning a path and then navigating it around an obstacle (Figure 7.2b).
2. That the multicopter is capable of planning a path and then navigating it in between two obstacles (Figure 7.2a).

Therefore, if these scenarios can be proven, it will show that the multicopter is capable of safely navigating on its own in an unknown two dimensional environment. By extension, it will also be able to traverse more complicated environments as all two dimensional environments will require the multicopter to navigate either in between or around obstacles.

At the start of each test, the yaw angle of the multicopter was locked. This enabled the avoidance algorithm to align its axis system with that of the multicopter. By doing this, it was possible to calculate the coordinates the multicopter should move to in the ENU (East, North, Up) coordinate system. It also forced the flight controller to try and keep the orientation of the multicopter constant.

The flight test had to be done during windless days as the wind significantly affected the height estimation of the multicopter. This is because the height





Figure 7.1: Multicopter test setup used to obtain the flight test results seen in Section 7.3.



(a) Obstacles with a gap in between them

(b) Solid obstacles

Figure 7.2: Flight test obstacle placements used to verify the multicopters' obstacle avoidance capabilities.

estimation is done with a barometer. To counter the wind, the multicopter also had to tilt itself to fly against it, affecting the map building process. In other words, since the tests were done at a height of 1.5 m, the ground was scanned and added as an obstacle in the map if the multicopter tilted to much. During windless days the multicopter, however, stayed horizontal for the proximity sensor measurements, but still struggled to maintain the set

height at all times. Like with the xy-setpoints (Section 7.3), the height was also continuously checked to see if it is within 0.3 m of the given setpoint. However, due to the nature of the obstacle course (no obstacles in range except the given obstacles) and the sensor measurement implementation (Chapter 5), the map was not influenced if, for a few seconds, the proximity sensor scanned above the obstacles.

Before flight tests were started, several safety features were included in the design to ensure that the multicopter would not fly off and endanger anyone in the vicinity of the test area. These safety features were that:

1. The pilot was always able to regain manual control with the remote when needed.
2. A python script on the Intel Edison continuously sent setpoints at a speed of 5 Hz to ensure the controller stayed in offboard mode, even if WiFi communication was lost (Section 4.2.3).
3. The multicopter was always armed and disarmed by the pilot and never with the use of the avoidance algorithm. Before it could be armed by the pilot, however, a safety switch on the multicopter had to be engaged.
4. After the multicopter was armed, it was switched to position control and then to offboard control. This enabled the multicopter to stay at its position if communication between the companion computer and the flight controller was broken during the flight (Section 4.1.1).
5. If an obstacle was detected on the goal position i.e. there is no valid path available, the multicopter was programmed to land.
6. When the path planning algorithm was initialised, a discretized map of the environment was generated. An additional safety boundary was then added around this map, representing obstacles. This ensured that no path can be planned outside a given area. This safety boundary could also not be affected by sensor measurement i.e. it cannot disappear over time.

### 7.3 Flight Test Results

In this section, multiple flight tests results are discussed. For each test, 4 data images are shown. The first image of each test represents the flight trajectory of the multicopter. This data is oriented relative to the axis system of the multicopter, where forward is in the positive x-direction and left is in the positive y-direction. Therefore, all the obstacles (black) indicated on the maps are given relative to the controllers' position estimation (blue). Also displayed on the test result are the setpoints (yellow) generated by MATLAB. These

setpoints are surrounded with an setpoint area (red). When the multicopter got within this area while flying, it was assumed that the setpoint was reached. The next setpoint was then generated by the avoidance algorithm and sent to the companion computer. The position estimation from the RTK GPS (orange), however, indicates the actual position of the multicopter during the flight time. This data is overlaid across the Pixhawk position estimation to see the error the controller was making. Lastly, the start and end point of the algorithm is indicated with a green circle and a red cross respectively.

The second and third image indicates the position data in the Northern and Eastern direction respectively. This data was generated by both the Pixhawk flight controller and the RTK GPS. From this data it is possible to determine the error the controller was making as seen in the last image. Therefore, it can be seen how the error in both the Northern and Eastern direction changed over time.

It should be noted that the first image of each test represent the position and obstacle data collected by MATLAB during the flight. However, the last three images for each test were generated afterwards as the Pixhawk saves all its flight data on a SD card at a higher rate than MATLAB received it. In other words, more data points were available for the position error analyses.

At the starting point of these tests, it can sometimes be seen that the multicopter "struggled" before it started moving towards the goal positions. This was due to the multicopter sometimes struggling to take off and reach its set height. For each test a different safety sphere size was used as indicated. D\* Lite, however, was always initiated to check the global path after six setpoints were generated by the VFF. Therefore, with a default grid size of 0.5 m, the maximum distance the multicopter could have moved before D\* Lite was initiated again is 3 m. For a summary of the test results as well as the position error data for each flight, refer to Table 7.1 and Table 7.2 respectively.

For the first test (Figure 7.3), no obstacles were added to the test environment. The avoidance algorithm, however, was still able to successful navigate the multicopter from the start position to the goal position as seen in Figure 7.3a. The entire test took 338 s (5 min and 38 s) to fly 14.5 m in the forward direction. Therefore, the average speed of the multicopter in the forward direction for this flight test was 0.043 m/s. From the position error data it can be seen that the Northern and Eastern error changed with time. However, after about one minutes, the error started to stabilize as the remainder of the data is more cluttered together.

In the second test (Figure 7.4), one obstacle was placed in front of the multicopter. The safety distance used for this test was 2 m. From Figure 7.4a it can be seen that the avoidance algorithm successfully planned and navigated the multicopter around the obstacle. The entire test took 274 s (4 min and 34 s) to fly 13.5 m in the forward direction. Therefore, the average speed of the multicopter in the forward direction for this flight test was 0.05 m/s. From the position error data it can be seen that the Northern and Eastern error

Table 7.1: Summary of flight test results.

Test #	Test Status	Safety distance [m]	Forward distance travelled [m]	Time [s]	Average forward speed [m/s]
1	Successful	-	14.5	338	0.043
2	Successful	2	13.5	274	0.05
3	Successful	1.5	16.5	339	0.05
4	Successful	2	14	151	0.09
5	Successful	1	18.5	119	0.16

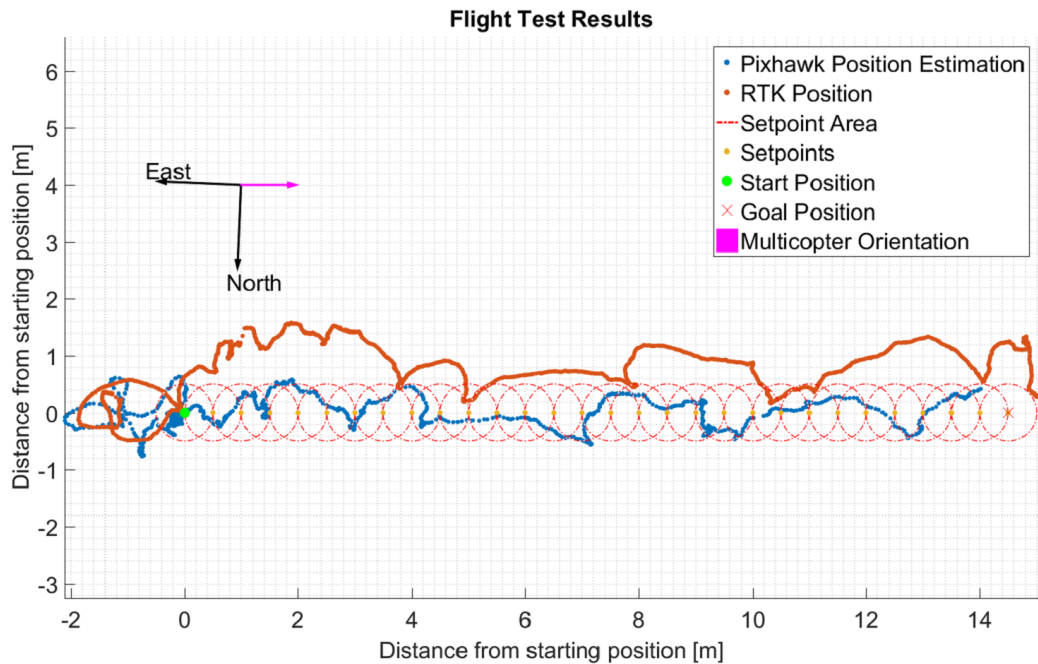
Table 7.2: PDF distribution values obtained through testing for both the Northern and Eastern error (Given in North, East and Down (NED) coordinate system). These values represent the average error,  $\mu$ , as well as the standard deviation,  $\sigma$ , at the end of each individual test.

Test #	North		East	
	$\mu$ [m]	$\sigma$ [m]	$\mu$ [m]	$\sigma$ [m]
1	1.1687	0.4841	1.0238	0.4891
2	1.3718	0.5264	0.2925	0.2603
3	-0.9442	0.3036	0.2953	0.3010
4	0.5507	0.2592	0.3424	0.1434
5	0.0308	0.2832	-0.2193	0.1532

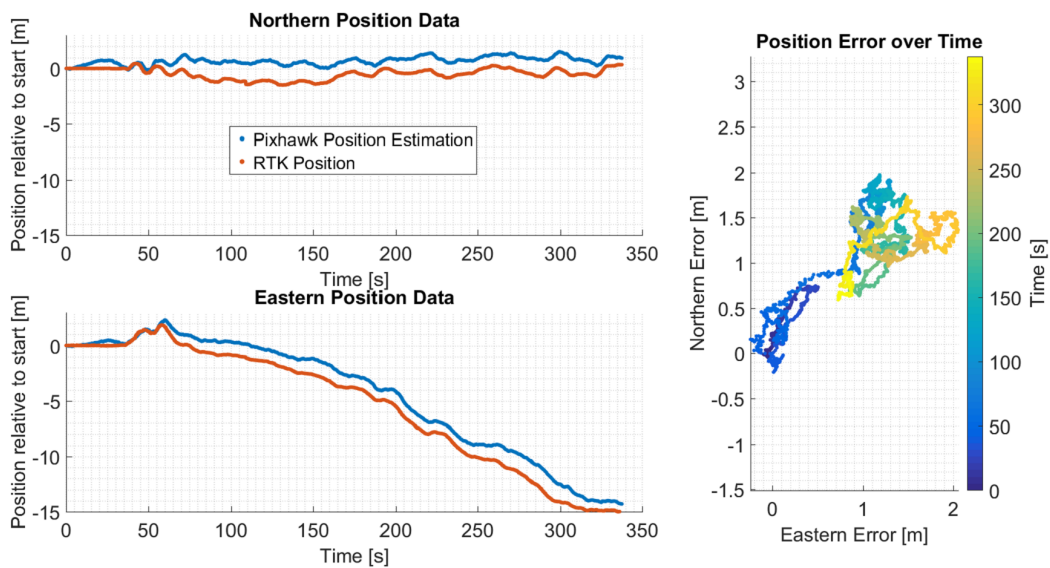
changed with time. However, again after about one minutes, the error started to stabilize as the remainder of the data is more cluttered together.

For both the third and the fourth flight test (Figure 7.5 and Figure 7.6 respectively) two obstacles were placed at a distance of 5 m from each other. For the third flight test (Figure 7.5a), a safety distance of 1.5 m was used. Whereas a safety distance of 2 m was used for the fourth flight test (Figure 7.6a). In both cases, the avoidance algorithm was able to successfully plan and navigate the multicopter from the start position, in between two obstacles to the goal position.

The third test took 339 s (5 min and 39 s) to fly 16.5 m in the forward direction. Therefore, the average speed of the multicopter in the forward direction for this flight test was 0.05 m/s. From the position error data it can be seen that the Northern and Eastern error changed with time. However, after about 40 s, the error started to stabilize as the rest of the data is more cluttered together.

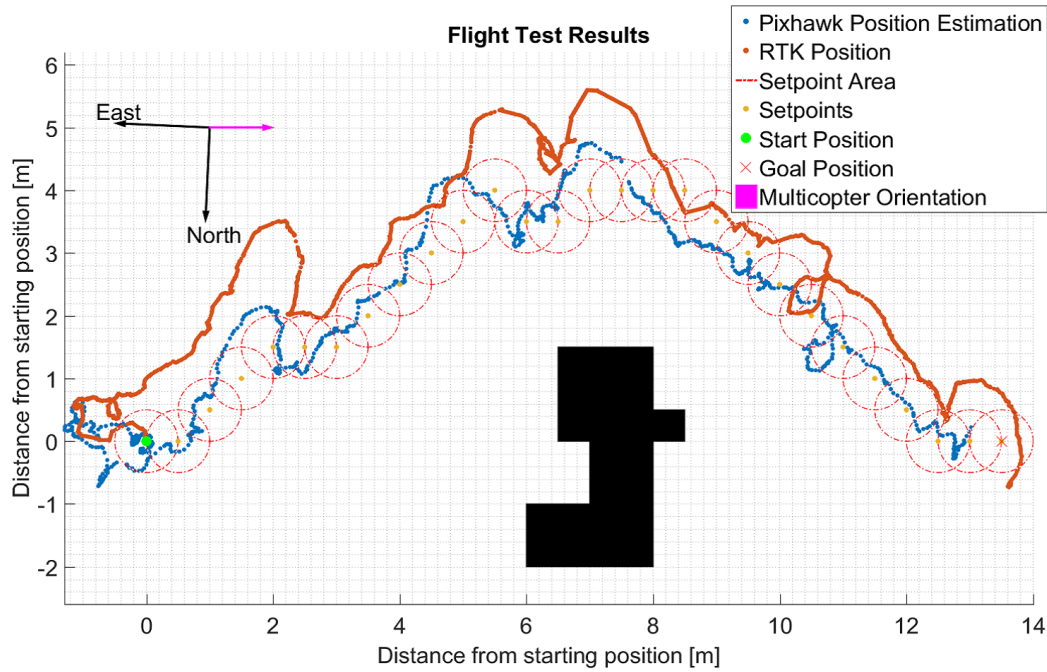


(a) Multicopter trajectory

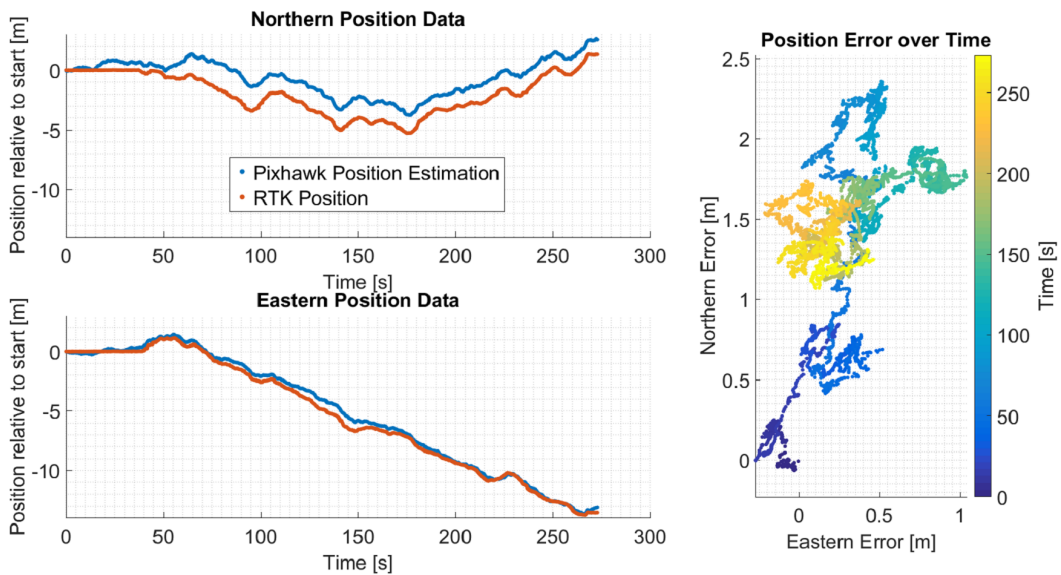


(b) Position and error data

Figure 7.3: First flights' test results. No obstacles were introduced to the test environment. a) Represents the trajectory followed by the multicopter during the flight test. It also contains the setpoints generated by MATLAB and the RTK (actual) position of the multicopter during the flight. b) Represents the position and error data of the multicopter in the Northern and Eastern direction for the duration of the flight. From the data it can be seen that the avoidance algorithm was able to successfully navigate the multicopter from the start position to the goal position in 338 s (5 min and 38 s).

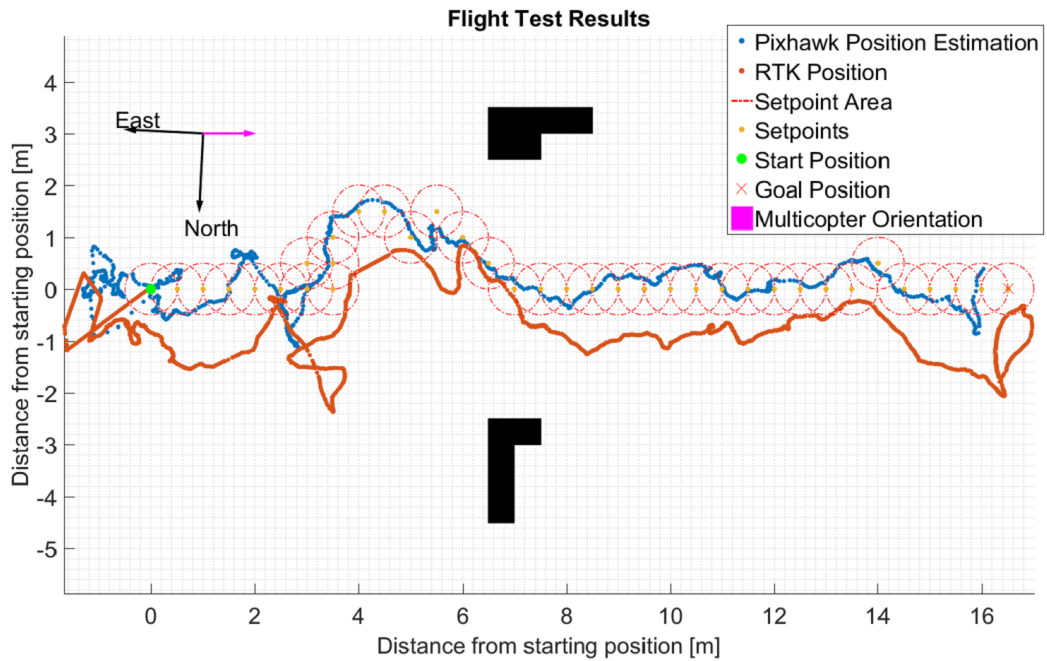


(a) Multicopter trajectory

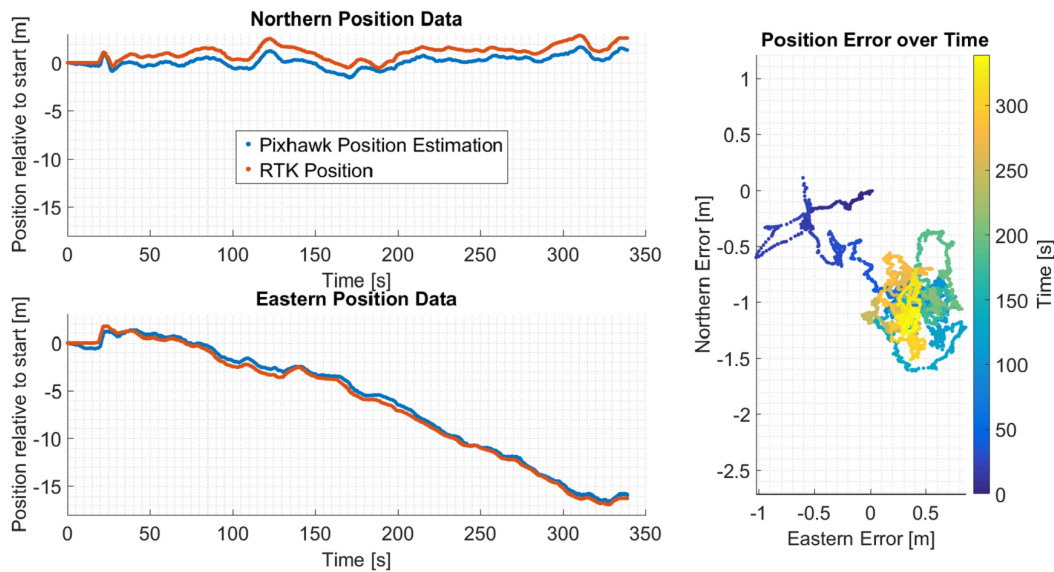


(b) Position and error data

Figure 7.4: Second flights' test results. One obstacles was added to the test environment. a) Represents the trajectory followed by the multicopter during the flight test. It also contains the setpoints generated by MATLAB and the RTK (actual) position of the multicopter during the flight. b) Represents the position and error data of the multicopter in the Northern and Eastern direction for the duration of the flight. From the data it can be seen that the avoidance algorithm was able to safely navigate the multicopter from the start position, around the obstacle to the goal position. The test was completed in 274 s (4 min and 34 s) while using a safety sphere with a radius of 2 m.

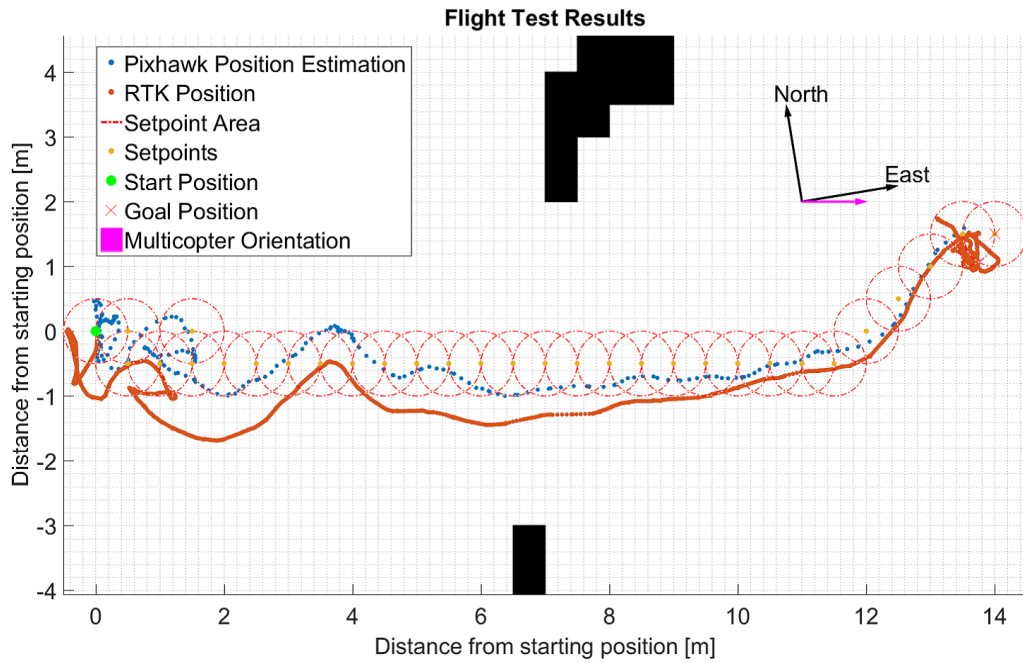


(a) Multicopter trajectory

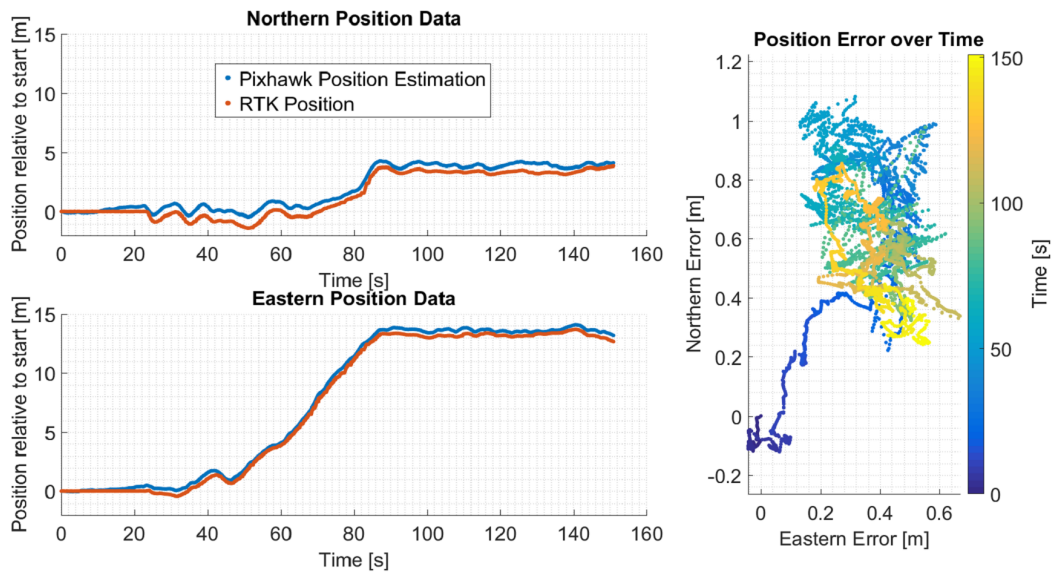


(b) Position and error data

Figure 7.5: Third flights' test results. Two obstacles were added to the test environment, 5 m apart from each other. a) Represents the trajectory followed by the multicopter during the flight test. It also contains the setpoints generated by MATLAB and the RTK (actual) position of the multicopter during the flight. b) Represents the position and error data of the multicopter in the Northern and Eastern direction for the duration of the flight. From the data it can be seen that the avoidance algorithm was able to safely navigate the multicopter from the start position, in between the two obstacles to the goal position. The test was completed in 339 s (5 min and 39 s) while using a safety sphere with a radius of 1.5 m.



(a) Multicopter trajectory



(b) Position and error data

Figure 7.6: Fourth flights' test results. Two obstacles were added to the test environment, 5 m apart from each other. a) Represents the trajectory followed by the multicopter during the flight test. It also contains the setpoints generated by MATLAB and the RTK (actual) position of the multicopter during the flight. b) Represents the position and error data of the multicopter in the Northern and Eastern direction for the duration of the flight. From the data it can be seen that the avoidance algorithm was able to safely navigate the multicopter from the start position, in between the two obstacles to the goal position. The test was completed in 151 s (2 min and 31 s) while using a safety sphere with a radius of 2 m.



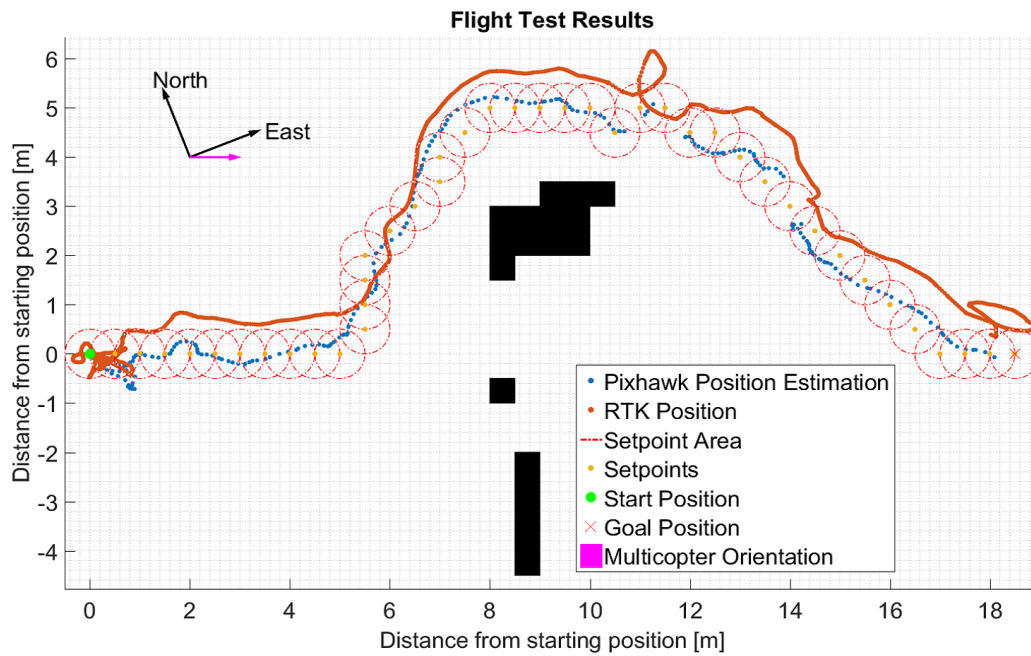
The fourth flight test on the other hand, took 151 s (2 min and 31 s) to fly a distance of 14 m in the forward direction. Therefore, the average speed of the multicopter in the forward direction for this flight test was 0.09 m/s. From the position error data it can be seen that the Northern and Eastern error changed with time. However, after about 20 s, the error started to stabilize as the rest of the data is more cluttered together.

For the fifth test (Figure 7.7), two obstacles were added to the test environment with a distance of 4 m between them. The safety distance used for this test setup was 1 m. From Figure 7.7a, it can be seen that the multicopter should have gone through the opening between the two obstacles, however, a false positive was received from the proximity sensor causing the path to be re-planned. As can be seen from the figure, the multicopter moved approximately 6 m and then changed direction i.e. D\* Lite was initiated to plan a new path. However, because a new path was generated, the multicopter was pulled along that path while it was correcting the map. When it was time to recalculate the path, the shortest path was to continue in the current direction and not turn around to try and move through the obstacles again. Even though the multicopter did not go in between the two obstacles, the avoidance algorithm was still able to safely navigate the multicopter from the start position to the goal position.

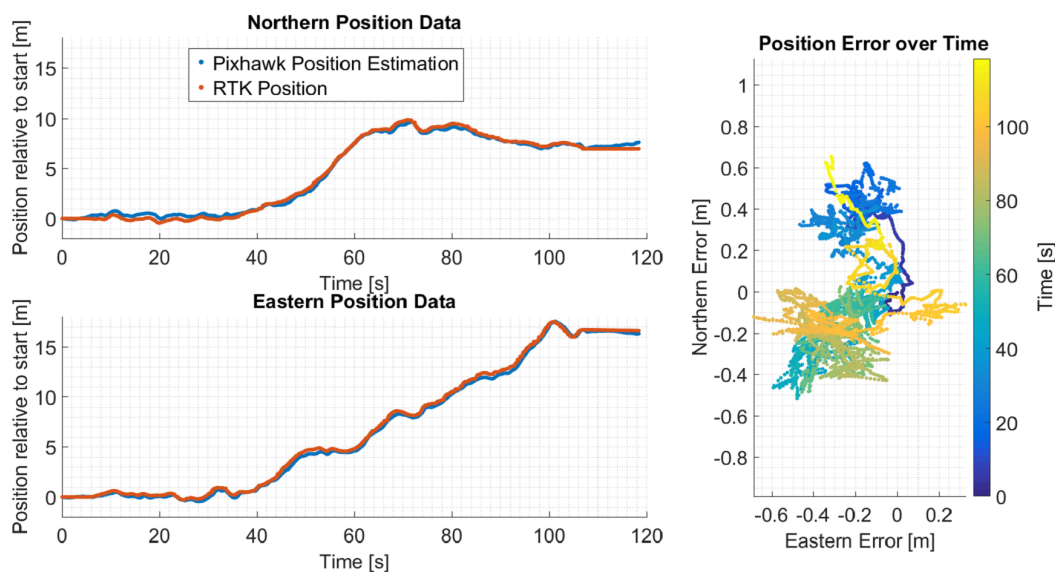
The fifth test took 119 (1 min and 59 s) to fly 18.5 m in the forward direction. Therefore, the average speed of the multicopter in the forward direction for this flight test was 0.16 m/s. From the position error data it can be seen that the Northern and Eastern error changed with time. Unlike all the other flight tests, the error stayed relatively constant from the start.

In the first three tests the forward speed of the multicopter was well below 0.1 m/s (Table 7.1). Only after the controller was tuned, the multicopter was able to fly faster as can be seen in the last two tests. For the fourth and fifth test the average speed in the forward direction increased by a factor of 2 and 3 respectively compared to the first three tests. As seen in Section 6.3, both D\* Lite as well as the VFF method execute at high speeds for small maps, therefore, most of the time flying was spent on getting to the given setpoint. This can also be seen in the multicopters' trajectory plots. For the first three tests, the pixhawk's position estimation is more densely packed than that of the last two tests. Therefore, from the plots it can also be seen that the multicopter moved slowly to the goal position.

Along with the speed increase, the tuned controller also had a significant decrease in both the average error and the standard deviation in the Northern direction. The size of the average error made in the Eastern direction, however, stayed relatively unchanged, but the standard deviation also decreased (Table 7.2). Therefore the error data for the last two flight tests are more compact than that of the first three tests.



(a) Multicopter trajectory



(b) Position and error data

Figure 7.7: Fifth flights' test results. Two obstacles were added to the test environment, 4 m apart from each other. a) Represents the trajectory followed by the multicopter during the flight test. It also contains the setpoints generated by MATLAB and the RTK (actual) position of the multicopter during the flight. b) Represents the position and error data of the multicopter in the Northern and Eastern direction for the duration of the flight. From the data it can be seen that the avoidance algorithm was able to safely navigate the multicopter from the start position, around the obstacles to the goal position. A safety sphere with a radius of 1 m was used for this test and it was completed in 119 s (1 min and 59 s). The algorithm was suppose to navigate the multicopter in between the two obstacles. However, a false positive proximity measurement regarding the environment caused the algorithm to rather navigate around the obstacles.

If it was possible to have access to the PDF while flying, the error could have been directly incorporated into the map building process. However, since it is not known while flying, the map is in constant flux due to the error in the position estimation made by the controller. For this reason, most of the obstacles in the occupancy grid appear deformed. However, the map is still traversable as it always adds the given obstacles correctly relative to its position estimation at any given time. From the error data shown in the test results, it could be seen that all the errors started to stabilize after a while. However, if the error is not 0 or entirely constant while the multicopter is flying, the effects will always be seen in the map building process.

## 7.4 Conclusion

In Section 7.1, the assumptions made regarding the test setup are given. The flight test setup used is then explained in Section 7.2, along with all the safety precautions taken to ensure that the risk of accidents were minimised. Final flight test results and a discussion regarding these results are then given in Section 7.3.

By using the combined algorithm as described in Chapter 6 in conjunction with the hardware and software setup described in Chapter 4, a working prototype of the obstacle avoidance system was obtained. This setup enabled the multicopter to safely navigate multiple unknown environments as illustrated in the test results (Section 7.3). For a summary of the test results as well as the position error data for each flight, refer to Table 7.1 and Table 7.2 respectively.

The flight test had to be done during windless days as the wind significantly affected the height estimation of the multicopter. This is because the height estimation is done with a barometer. To counter the wind, the multicopter also had to tilt itself to fly against it, affecting the map building process. In other words, since the tests were done at a height of 1.5 m, the ground was scanned and added as an obstacle in the map if the multicopter tilted too much. During windless days the multicopter, however, stayed horizontal for the proximity sensor measurements, but still struggled to maintain the set height at all times. Like with the xy-setpoints (Section 7.3), the height was also continuously checked to see if it is within 0.3 m of the given setpoint.

Since the yaw angle was algorithmically locked before each flight, the assumption regarding the multicopter orientation staying constant was correct. Along with only doing flight tests on windless days, a maximum pitch angle was also given to the multicopter, constricting the tilt angle when moving. The obstacle placement was also done in such a way that the obstacle course looked identical 0.3 m above and below the given flight height. However, due to the nature of the obstacle course (no obstacles in range except the given obstacles) and the sensor measurement implementation (Chapter 5), the map was not influenced if, for a few seconds, the proximity sensor scanned above

the obstacles. Therefore, in the setup used for the flight tests, the assumptions regarding the tilt angle and the height of the multicopter can be assumed correct.

# Chapter 8

## Conclusion and Recommendations

This study was aimed at designing, building and testing a system that enables a multicopter to fly autonomously in a dynamic 2D environment. This included the proximity sensor design, algorithm development, simulations, component integration and testing. In the end, a modular system was developed and successfully integrated, enabling the multicopter to fly autonomously in a dynamic 2D environment. In Section 8.1, a brief overview of the work is given alongside concluding remarks. This is followed by recommendations regarding future work in Section 8.2.

### 8.1 Conclusion

In Chapter 2 a literature review regarding two different map building techniques (uniform and adaptive occupancy grids) is given. This is followed by a review of several obstacle avoidance algorithms, including: The Virtual Force Field (VFF) method, A\*, LPA\* and D\* Lite. From the investigated algorithms a combination of three (Uniform occupancy grid map building technique, the VFF method and D\* Lite) were chosen for the path planning algorithm. To be able to use the uniform occupancy grid map building technique, a probability update formula was derived.

One of the requirements for the probability update formula is an inverse sensor model. Therefore, an analytical formula for the proximity sensor, corrupted by Gaussian noise, is derived in Chapter 3. An analytical formula for the pose uncertainty of the multicopter is also derived. This enabled the probability update formula to incorporate both measurement and pose uncertainty straight into the map building process. Both of these formulas were then simulated and the results evaluated to determine what effect they have on the map building process and whether or not simplified methods can be used to generate similar results.

In Chapter 4, an overview is given of the different hardware and software that had to be integrated to allow MATLAB to fully control the Pixhawk flight

controller. The modular design used in the system integration also allows individual components to be changed, rather than re-designing the entire system every time. In other words, problems are isolated. If there is a problem with the avoidance algorithm, changes can easily be made without affecting other parts of the system like the flight controller. Also included in this chapter is the proximity sensor design. Test results regarding the designed proximity sensors' performance can be found in Chapter 5.

In Chapter 6, the algorithmic combination of the occupancy grid map building technique, D\* Lite and the VFF is explained. This is followed by an investigation regarding the heuristic and cost function used for D\* Lite, as well as the formulation and algorithmic implementation of the VFF. Simulation results regarding both static and dynamic environments are then illustrated and discussed along with the computational time of D\* Lite and the VFF.

The final system was then implemented on the multicopter and tested in an outdoor test environment as shown in Chapter 7. From the results, it can be seen that the obstacle avoidance system was able to successfully navigate the multicopter through different obstacle arrangements. Therefore, the aims of the project were reached as a system enabling the multicopter to fly autonomously in a dynamic 2D environment was successfully implemented.

## 8.2 Recommendations

The problem of autonomous flight still remains an active research area and, with the proposed system taken into account, offers many directions for further development and improvement. Here follows some proposed areas of improvement.

To further increase the speed of the multicopter, the Pixhawk flight controller can be better tuned. As seen from Section 7.3, the speed of the multicopter can be greatly increased if the controller is properly tuned. Another method would be to send velocity commands to the controller (Section 4.1.1) and not setpoints as has been done for this project. However, if the speed of the multicopter is dramatically increased, the pitch of the multicopter will have to be taken into account as it will have an effect on the proximity sensor measurements. To counter this, a gimbal can be added to keep the proximity sensor horizontal at all times. This was however not necessary for this project as the low speeds essentially kept the multicopter horizontal.

For the controller to better estimate its position, the sensor fusion between the GPS and the onboard IMU (Inertial Measurement Unit) can be improved. Another method would be to integrate the RTK GPS directly alongside the Pixhawk flight controller. However, if this is used, an additional ground station unit would have to be present at all times. The RTK GPS also only has an accuracy of 2 cm horizontally in clear skies if it has an RTK lock which can take up to 20 min to obtain each time. With a RTK GPS available, the height

estimation of the multicopter can also be greatly improved. This can be done by either using the height given by the RTK GPS or doing sensor fusion between both the RTK and the barometer. Another way to increase the height stability of the multicopter would be to have a range finder measuring the height. However, the pitch angle of the multicopter can affect these measurements. Therefore, a gimbal would also be advised.

All of the above mentioned solutions can help improve the stability and accuracy of the multicopter while flying, however, the weight the multicopter can carry should always be taken into account. If this gets too heavy, the time the multicopter can fly will be significantly reduced as the power usage of the rotors will increase.

To improve the path planning algorithm, the quadtree map building algorithm can be added alongside D\* Lite and the VFF method. For this project it was not a viable solution as the map was constantly changing, during flight tests, due to the error in the position estimation. However, if the position estimation is significantly improved, quadtrees would be a better two-dimensional map building technique. If the code is extended to a three-dimensional environment, uniform grids would be advised at first, however, an octree (three-dimensional version of a quadtree) map building algorithm can also be used. ROS recently developed its own octree mapping algorithm that can seamlessly be integrated with the current system. If the quadtree or octree map building technique is, however, chosen and implemented, the current avoidance algorithm would also have to be adapted accordingly.

To decrease the size of the system, the Arduino sensor setup can be eliminated, connecting the distance sensor directly to the Intel Edison as it has the similar pin setup to the Arduino Mega. It is also possible to eliminate the ground station computer by re-writing all the avoidance algorithm in C, C++ or Python and then executing it on the Intel Edison. However, if this is done, the modular design of the system is lost. The way it is set up at the moment (Figure 4.1), any avoidance algorithm programmed in MATLAB can be tested alongside the Pixhawk flight controller, if programmed correctly. The distance sensor can also easily be swapped out for a better one.

To ensure better safety of the vehicle and the people in the surrounding areas, more safety features can be added to the design. For instance, enabling the multicopter to automatically land if the battery gets too low or if communication is lost between the flight controller with the remote.

# List of References

- (2014). Arduino Mega 2560 Pinout.  
Available at: <http://www.electroschematics.com/7963/arduino-mega-2560-pinout/>
- (2015). jMAVSim - Pixhawk Flight Controller Hardware Project.  
Available at: <https://pixhawk.org/dev/hil/jmavsim>
- (2015). Pixhawk Flight Controller.  
Available at: [http://www.sishobby.com/358-thickbox\\_default/3dr-pixhawk-flight-controller-ublox-gps-module.jpg](http://www.sishobby.com/358-thickbox_default/3dr-pixhawk-flight-controller-ublox-gps-module.jpg)
- (2016). Drones Cut Cost of Thermographic PV Panel Inspections - UAS VISION.  
Available at: <http://www.uasvision.com/2016/09/15/drones-cut-cost-of-thermographic-pv-panel-inspections/>
- (2016). Intel Edison.  
Available at: [http://www.mouser.com/images/microsites/Intel\\_EDI1ARDUINALK.jpg](http://www.mouser.com/images/microsites/Intel_EDI1ARDUINALK.jpg)
- Choset, H. (2005). *Principles of robot motion : theory, algorithms, and implementation*. MIT Press, Cambridge, Mass.
- Coetzee, J. and Smit, W. (2016). Simulation of an Obstacle Avoidance Algorithm in a Dynamic 2D Environment. In: PENG, P.Z. and LIN, D.F. (eds.), *International Micro Air Vehicle Competition and Conference 2016*, pp. 256–263. Beijing, PR of China.
- De Filippis, L., Guglieri, G. and Quagliotti, F. (2012). Path Planning Strategies for UAVS in 3D Environments. *Journal of Intelligent & Robotic Systems*, vol. 65, no. 1-4, pp. 247–264.
- Droeschel, D., Nieuwenhuisen, M., Beul, M., Holz, D., Stückler, J. and Behnke, S. (2016). Multilayered Mapping and Navigation for Autonomous Micro Aerial Vehicles. *Journal of Field Robotics*, vol. 33, no. 4, pp. 451–475.
- Goerzen, C., Kong, Z. and Mettler, B. (2010). A Survey of Motion Planning Algorithms from the Perspective of Autonomous UAV Guidance. *Journal of Intelligent and Robotic Systems*, vol. 57, no. 1-4, pp. 65–100.



- Hoy, M., Matveev, A.S. and Savkin, A.V. (2015). Algorithms for collision-free navigation of mobile robots in complex cluttered environments: a survey. *Robotica*, vol. 33, no. 03, pp. 463–497.
- Joubert, D. (2012). Adaptive occupancy grid mapping with measurement and pose uncertainty. Master of Science in Applied Maths. University of Stellenbosch. Stellenbosch, South Africa.
- Joubert, D., Brink, W. and Herbst, B. (2015). Pose Uncertainty in Occupancy Grids through Monte Carlo Integration. *J Intell Robot Syst Journal of Intelligent & Robotic Systems : with a special section on Unmanned Systems*, vol. 77, no. 1, pp. 5–16.
- Koenig, S. and Likhachev, M. (2002a). D\* Lite. *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pp. 476–483.
- Koenig, S. and Likhachev, M. (2002b). Improved fast replanning for robot navigation in unknown terrain. In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, vol. 1, pp. 968–975. IEEE.
- Moravec, H.P. (1988). Sensor Fusion in Certainty Grids for Mobile Robots. *AI Magazine*, vol. 9, no. 2, pp. 61–74.
- Mujumdar, A. and Padhi, R. (2011). Evolving Philosophies on Autonomous Obstacle/Collision Avoidance of Unmanned Aerial Vehicles. *Journal of Aerospace Computing, Information, and Communication*, vol. 8, no. 2, pp. 17–41.
- Paul, T., Krogstad, T.R. and Gravdahl, J.T. (2008). Modelling of UAV formation flight using 3D potential field. *Simulation Modelling Practice and Theory*, vol. 16, no. 9, pp. 1453–1462.
- Rubinstein, R. (2008). *Simulation and the monte carlo method*. John Wiley & Sons, Hoboken, N.J.

# Appendices

# Appendix A

## Pseudo Code For Obstacle Avoidance Algorithms

### A.1 A\*

**Data:** Start vertex, Goal vertex, Graph

**Result:** Shortest path between given vertices (if one exists)

**Procedure** *Main()*:

```

while priority queue is not empty do
  Pick  $n_{best}$  from  $O$  such that  $f(n_{best}) \leq f(n), \forall n \in O$ ;
  Remove  $n_{best}$  from  $O$  and add to  $C$ ;
  if  $n_{best} = q_{goal}$  then
    | Exit
  end
  for all  $x \in Star(n_{best})$  not in  $C$  do
    | Expand  $n_{best}$ 
  end
  if  $x \ni O$  then
    | add  $x$  to  $O$ 
  else if  $g(n_{best}) + c(n_{best}, x) < g(x)$  then
    | update  $x$ 's backpointer to point to  $n_{best}$ 
  end
end
end

```

**Algorithm 1:** A\* algorithmic progression adapted from (Choset, 2005)

## A.2 LPA\*

**Data:** Start vertex, Goal vertex, Graph**Result:** Shortest path between given vertices (if one exists)**Procedure** *Main()*:

```

| Initialize();
| while 1 do
|   ComputeShortestPath();
|   Wait for changes in edge cost;
|   for all directed edges  $(u, v)$  with changed edge costs do
|     Update the edge cost  $c(u, v)$ ;
|     UpdateVertex( $v$ );
|   end
| end
end

```

**Procedure** *Initialize()*:

```

|  $U = \emptyset$ ;
| for all  $s \in S$  do
|    $rhs(s) = g(s) = \infty$ ;
| end
|  $rhs(s_{start}) = 0$ ;
|  $U.Insert(s_{start}, CalculateKey(s_{start}))$ ;
end

```

**Procedure** *CalculateKey( $s$ )*:

```

| return  $[\min(g(s), rhs(s)) + h(s, s_{goal}); \min(g(s), rhs(s))]$ 
end

```

**Procedure** *UpdateVertex( $u$ )*:

```

| if  $u \neq s_{start}$  then
|    $rhs(u) = \min_{s' \in Pred(u)} (g(s') + c(s', u))$ 
| end
| if  $u \in U$  then
|    $U.Remove(u)$ 
| end
| if  $g(u) \neq rhs(u)$  then
|    $U.Insert(u, CalculateKey(u))$ 
| end
end

```

**Algorithm 2:** LPA\* algorithmic progression adapted from (Koenig and Likhachev, 2002a) part 1

```

Procedure ComputeShortestPath():
  while ( $U.TopKey() < CalculateKey(s_{goal}) \parallel rhs(s_{goal} \neq g(s_{goal}))$ ) do
     $u = U.Pop()$ ;
    if ( $g(u) > rhs(u)$ ) then
       $g(u) = rhs(u)$ ;
      for all  $s \in Succ(u)$  do
         $UpdateVertex(s)$ 
      end
    else
       $g(u) = \infty$ ;
      for all  $s \in Succ(u) \cup \{u\}$  do
         $UpdateVertex(s)$ 
      end
    end
  end

```

**Algorithm 3:** LPA\* algorithmic progression adapted from (Koenig and Likhachev, 2002a) part 2

## A.3 D\* Lite

**Data:** Start vertex, Goal vertex, Graph**Result:** Shortest path between given vertices (if one exists)**Procedure** *Main()*:

```

     $s_{\text{last}} = s_{\text{start}};$ 
    Initialize();
    ComputeShortestPath();
    while  $s_{\text{start}} \neq s_{\text{goal}}$  do
         $s_{\text{start}} = \arg \min_{s' \in \text{Succ}(s_{\text{start}})} (c(s_{\text{start}}, s') + g(s'));$ 
        Move to  $s_{\text{start}};$ 
        Scan graph for changes in edge cost;
        if any costs changed then
             $k_m = k_m + h(s_{\text{last}}, s_{\text{start}});$ 
             $s_{\text{last}} = s_{\text{start}};$ 
            for all directed edges  $(u, v)$  with changed edge costs do
                Update the edge cost  $c(u, v);$ 
                UpdateVertex( $u$ );
            end
            ComputeShortestPath();
        end
    end

```

**end****Procedure** *Initialize()*:

```

     $U = \emptyset;$ 
     $k_m = 0;$ 
    for all  $s \in S$  do
         $rhs(s) = g(s) = \infty;$ 
    end
     $rhs(s_{\text{goal}} = 0);$ 
     $U.\text{Insert}(s_{\text{goal}}, \text{CalculateKey}(s_{\text{goal}}));$ 

```

**end****Procedure** *CalculateKey*( $s$ ):

```

    | return  $[\min(g(s), rhs(s)) + h(s_{\text{start}}, s) + k_m; \min(g(s), rhs(s))]$ 

```

**end****Algorithm 4:** D\* Lite algorithmic progression adapted from (Koenig and Likhachev, 2002a) part 1

```

Procedure UpdateVertex(u):
  if  $u \neq s_{goal}$  then
    |  $rhs(u) = \min_{s' \in Succ(u)} (c(u, s') + g(s'))$ 
  end
  if  $u \in U$  then
    |  $U.Remove(u)$ 
  end
  if  $g(u) \neq rhs(u)$  then
    |  $U.Insert(u, CalculateKey(u))$ 
  end
end

Procedure ComputeShortestPath():
  while ( $U.TopKey() < CalculateKey(s_{start}) \parallel rhs(s_{start}) \neq g(s_{start})$ )
  do
    |  $k_{old} = U.TopKey()$ ;
    |  $u = U.Pop()$ ;
    | if  $k_{old} < CalculateKey(u)$  then
    | |  $U.Insert(u, CalculateKey(u))$ 
    | else if ( $g(u) > rhs(u)$ ) then
    | |  $g(u) = rhs(u)$ ;
    | | for all  $s \in Pred(u)$  do
    | | |  $UpdateVertex(s)$ 
    | | end
    | else
    | |  $g(u) = \infty$ ;
    | | for all  $s \in Pred(u) \cup \{u\}$  do
    | | |  $UpdateVertex(s)$ 
    | | end
    | end
  end
end

```

**Algorithm 5:** D\* Lite algorithmic progression adapted from (Koenig and Likhachev, 2002a) part 2

# Appendix B

## Datasheets

### B.1 Pixhawk Flight Controller





# PX4FMU – Flight Management Unit

## QUICK START – HARDWARE VERSION 1.7

### Description

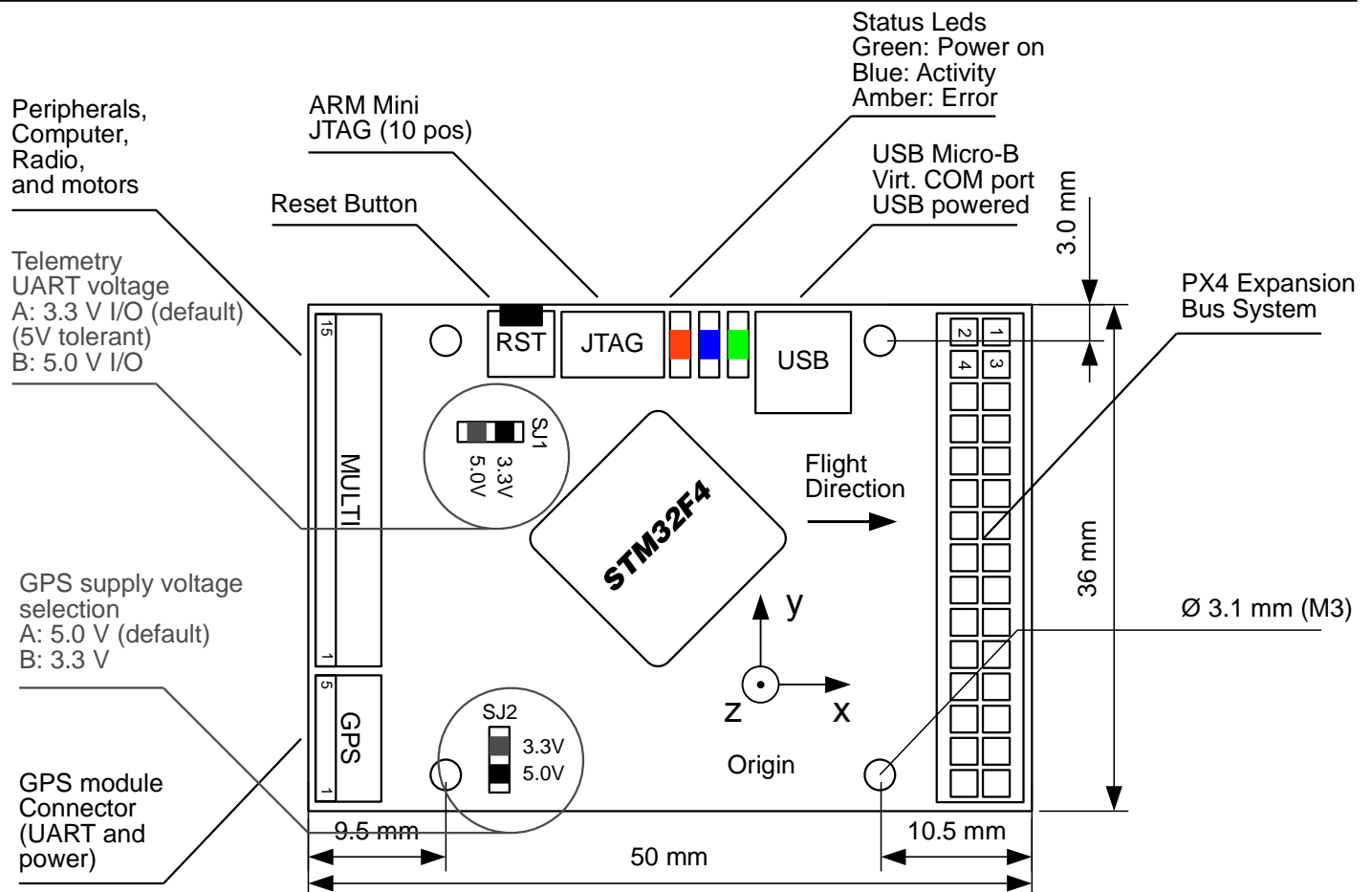
PX4FMU is an onboard management unit for micro air vehicles. It combines an autopilot and inertial measurement unit and enables the control of an aircraft using a single-board solution. Additional I/O can be easily connected via the 30-pin expansion bus.

<http://pixhawk.ethz.ch/px4/>

### Features

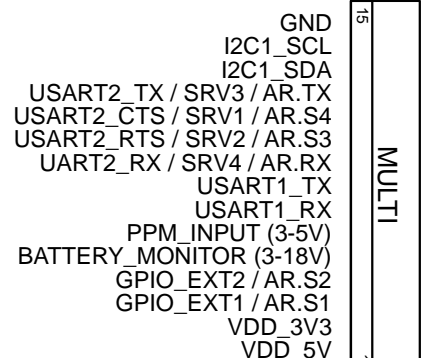
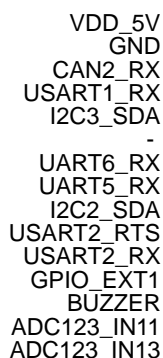
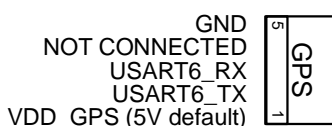
- 168 Mhz Cortex-M4 CPU (196 KB RAM, 1 MB Flash)
- 250 mW typical power consumption
- Reverse polarity protection on all power inputs
- 3D gyro, accelerometer and magnetometer, pressure sensors
- I2C, 3x UART, PPM, analog, GPS, 2x 5V GPIO, 4x PWM / Servo
- MicroSD card slot
- Expansion bus: CAN, 2x I2C, SPI, 4x analog, 2x UART, GPIOs
- USB Serial Port (Virtual COM Port / VCP) and bootloader
- 50 x 36 x 6 mm (1.38x1.97x0.24"), 8g, 30x30 mm mounting holes
- 4.5-6 V wide supply input range (incl. USB power)
- Selectable 3.3 V or 5 V IO for UART2 and GPS ports

### Connectors, Jumpers and Dimensions



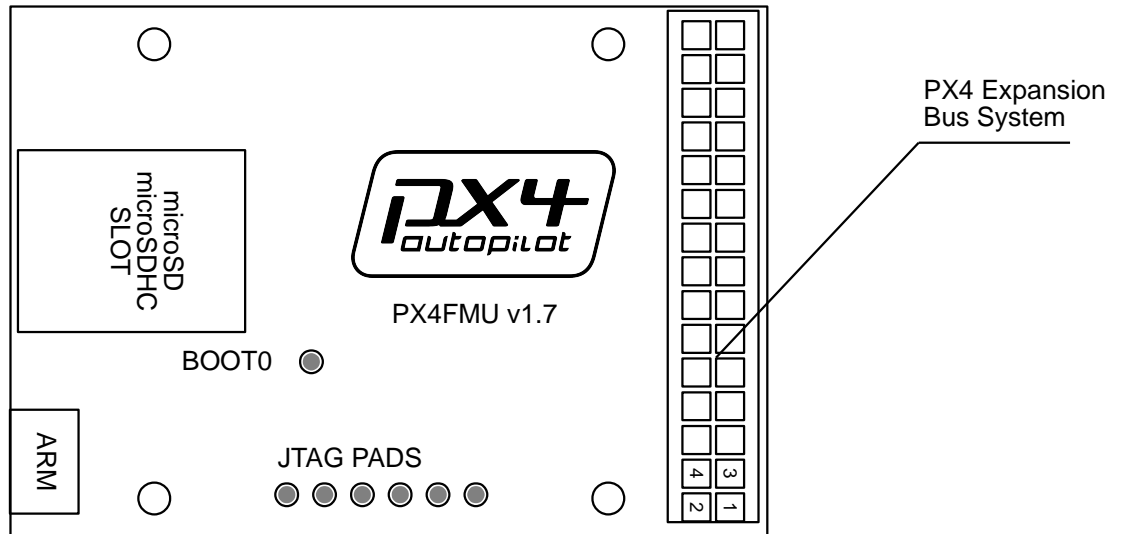
### Pinout and absolute maximum Ratings

- Input: 4.3-6 V (VDD\_5V), 20 mA onboard use, max. 800 mA for max peripheral load. Reverse-polarity protected.
- Output: 3.3 V (VDD\_3V3), fuse-limited 500 mA EXT, 3.3 V, fuse-limited 200 mA GPS



## Additional connectors (bottom side)

The footprints on the bottom side of the connector can be used by advanced users to interface additional boards or sensors.



## Software Tools / Getting Started

Please check the most recent user manual at <https://pixhawk.ethz.ch/px4/users/>

## Upgrading Firmware / Developing Custom Code

Please check the most recent developer instructions at <https://pixhawk.ethz.ch/px4/dev/>

## Open Hardware License

PX4FMU is an open hardware design, following the OSHW 1.1 definition licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) license. PX4FMU uses the BSD-licensed NuttX operating system as base for the PX4 software stack (<http://nuttx.sourceforge.net>).

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

<http://creativecommons.org/licenses/by-sa/3.0/>

## **B.2 Intel Edison**



# Intel® Edison Development Platform

## Introduction

The Intel® Edison development platform is designed to lower the barriers to entry for a range of inventors, entrepreneurs, and consumer product designers to rapidly prototype and produce “Internet of Things” (IoT) and wearable computing products.

## Intel® Edison Board for Arduino\*

Supports Arduino Sketch, Linux, Wi-Fi, and Bluetooth.

Board I/O: Compatible with Arduino Uno (except 4 PWM instead of 6 PWM):

- 20 digital input/output pins, including 4 pins as PWM outputs.
- 6 analog inputs.
- 1 UART (Rx/Tx).
- 1 I<sup>2</sup>C.
- 1 ICSP 6-pin header (SPI).
- Micro USB device connector OR (via mechanical switch) dedicated standard size USB host Type-A connector.
- Micro USB device (connected to UART).
- SD card connector.
- DC power jack (7 to 15 VDC input).

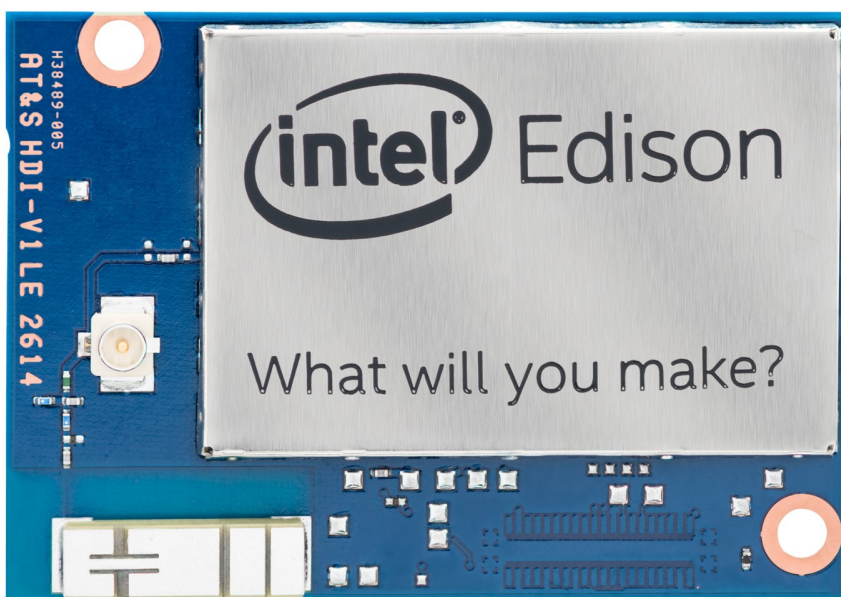
## Intel® Edison Breakout Board

Slightly larger than the Intel® Edison module, the Intel® Edison Breakout Board has a minimal set of features:

- Exposes native 1.8 V I/O of the Edison module.
- 0.1 inch grid I/O array of through-hole solder points.
- USB OTG with USB Micro Type-AB connector.
- USB OTG power switch.
- Battery charger.
- USB to device UART bridge with USB micro Type-B connector.
- DC power supply jack (7 to 15 VDC input).

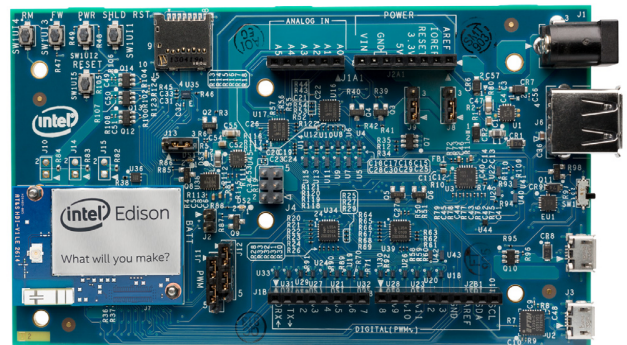
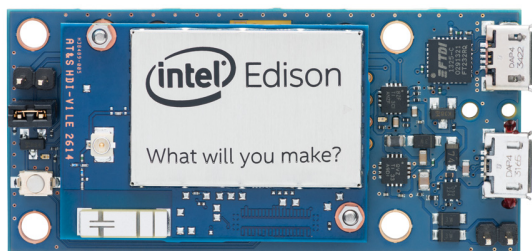
## Intel® IoT Analytics Platform

- Provides seamless Device-to-Device and Device-to-Cloud communication.
- Ability to run rules on your data stream that trigger alerts based on advanced analytics.
- Foundational tools for collecting, storing, and processing data in the cloud.
- Free for limited and noncommercial use.



## Intel® Edison Development Platform

PHYSICAL	
Form factor	Board with 70-pin connector
Dimensions	35.5 × 25.0 × 3.9 mm (1.4 × 1.0 × 0.15 inches) max
C/M/F	Blue PCB with shields / No enclosure
Connector	Hirose DF40 Series (1.5, 2.0, or 3.0 mm stack height)
Operating temperature	32 to 104°F (0 to 40°C)
EXTERNAL INTERFACES	
Total of 40 GPIOs, which can be configured as:	
SD card	1 interface
UART	2 controllers (1 full flow control, 1 Rx/Tx)
I2C	2 controllers
SPI	1 controller with 2 chip selects
I2S	1 controller
GPIO	Additional 12 (with 4 capable of PWM)
USB 2.0	1 OTG controller
Clock output	32 kHz, 19.2 MHz
MAJOR EDISON COMPONENTS	
SoC	22 nm Intel® SoC that includes a dual-core, dual-threaded Intel® Atom™ CPU at 500 MHz and a 32-bit Intel® Quark™ microcontroller at 100 MHz
RAM	1 GB LPDDR3 POP memory (2 channel 32bits @ 800MT/sec)
Flash storage	4 GB eMMC (v4.51 spec)
WiFi	Broadcom* 43340 802.11 a/b/g/n; Dual-band (2.4 and 5 GHz) Onboard antenna or external antenna (SKU configurations)
Bluetooth	Bluetooth 4.0
POWER	
Input	3.3 to 4.5 V
Output	100 ma @3.3 V and 100 ma @ 1.8 V
Power	Standby (No radios): 13 mW Standby (Bluetooth 4.0): 21.5 mW (BTLE in Q4-14) Standby (Wi-Fi): 35 mW
FIRMWARE + SOFTWARE	
CPU OS	Yocto Linux* v1.6
Development environments	Arduino* IDE Eclipse supporting: C, C++, and Python Intel XDK supporting: Node.JS and HTML5
MCU OS	RTOS
Development environments	MCU SDK and IDE



Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725 or by visiting Intel's website at <http://www.intel.com/design/literature.htm>.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See [http://www.intel.com/products/processor\\_number](http://www.intel.com/products/processor_number) for details.

Intel, the Intel logo, Atom, Pentium, Quark, and Xeon are trademarks of Intel Corporation in the United States and other countries.

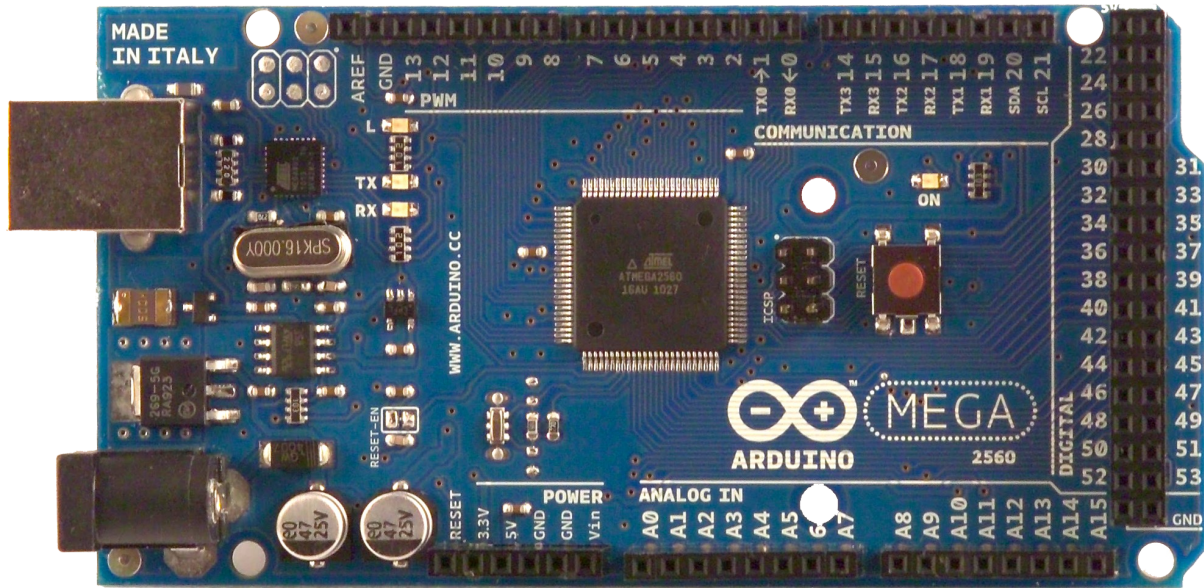
\*Other names and brands may be claimed as the property of others.

Copyright © 2014 Intel Corporation. All rights reserved.



## **B.3 Arduino Mega**

# Arduino MEGA 2560



## Product Overview

The Arduino Mega 2560 is a microcontroller board based on the ATmega2560 ([datasheet](#)). It has 54 digital input/output pins (of which 14 can be used as PWM outputs), 16 analog inputs, 4 UARTs (hardware serial ports), a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started. The Mega is compatible with most shields designed for the Arduino Duemilanove or Diecimila.

## Index

Technical Specifications

Page 2

How to use Arduino  
Programming Environment, Basic Tutorials

Page 6

Terms & Conditions

Page 7

Environmental Policies  
half sqm of green via Impatto Zero®

Page 7



**radiospares**

**RADIONICS**



# Technical Specification

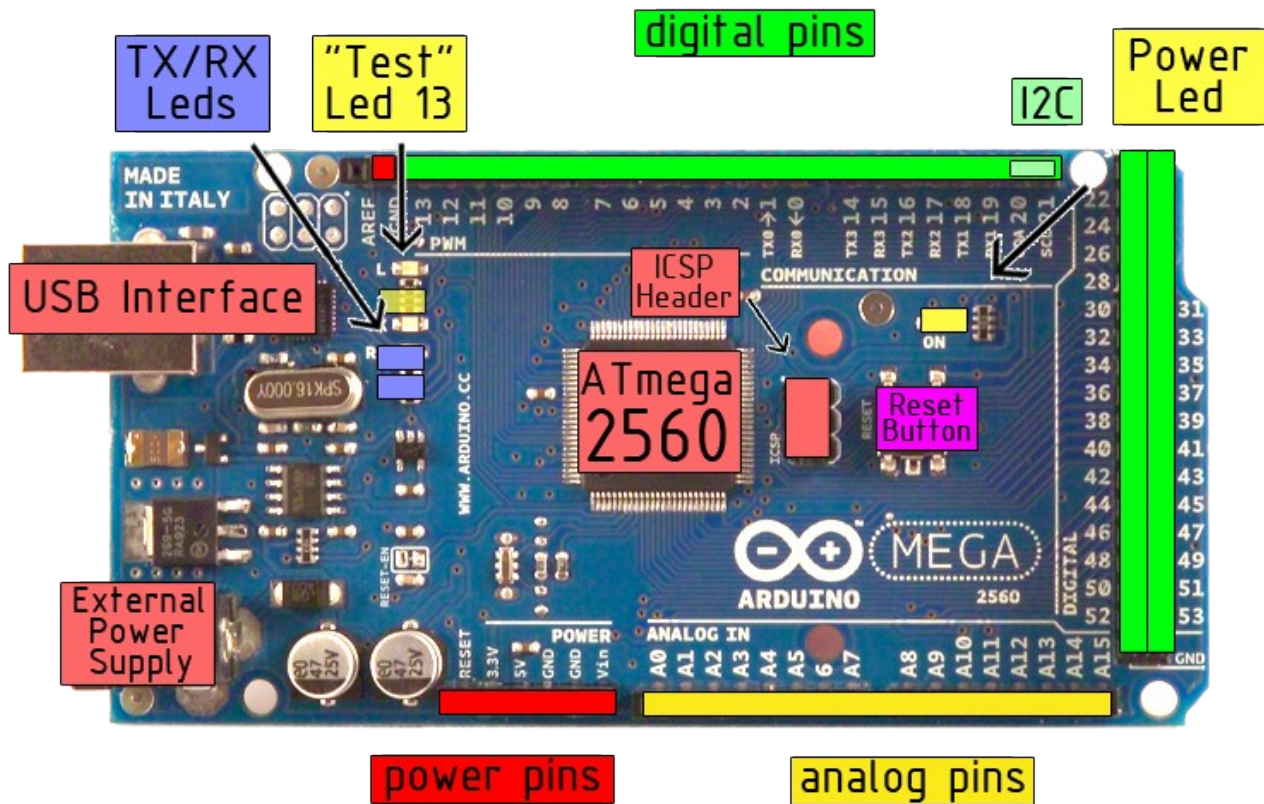


EAGLE files: [arduino-mega2560-reference-design.zip](#) Schematic: [arduino-mega2560-schematic.pdf](#)

## Summary

Microcontroller	ATmega2560
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	54 (of which 14 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	256 KB of which 8 KB used by bootloader
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz

## the board



*radiospares*

**RADIONICS**





## Power

The Arduino Mega2560 can be powered via the USB connection or with an external power supply. The power source is selected automatically. External (non-USB) power can come either from an AC-to-DC adapter (wall-wart) or battery. The adapter can be connected by plugging a 2.1mm center-positive plug into the board's power jack. Leads from a battery can be inserted in the Gnd and Vin pin headers of the POWER connector.

The board can operate on an external supply of 6 to 20 volts. If supplied with less than 7V, however, the 5V pin may supply less than five volts and the board may be unstable. If using more than 12V, the voltage regulator may overheat and damage the board. The recommended range is 7 to 12 volts.

The Mega2560 differs from all preceding boards in that it does not use the FTDI USB-to-serial driver chip. Instead, it features the Atmega8U2 programmed as a USB-to-serial converter.

The power pins are as follows:

- **VIN.** The input voltage to the Arduino board when it's using an external power source (as opposed to 5 volts from the USB connection or other regulated power source). You can supply voltage through this pin, or, if supplying voltage via the power jack, access it through this pin.
- **5V.** The regulated power supply used to power the microcontroller and other components on the board. This can come either from VIN via an on-board regulator, or be supplied by USB or another regulated 5V supply.
- **3V3.** A 3.3 volt supply generated by the on-board regulator. Maximum current draw is 50 mA.
- **GND.** Ground pins.

## Memory

The ATmega2560 has 256 KB of flash memory for storing code (of which 8 KB is used for the bootloader), 8 KB of SRAM and 4 KB of EEPROM (which can be read and written with the [EEPROM library](#)).

## Input and Output

Each of the 54 digital pins on the Mega can be used as an input or output, using [pinMode\(\)](#), [digitalWrite\(\)](#), and [digitalRead\(\)](#) functions. They operate at 5 volts. Each pin can provide or receive a maximum of 40 mA and has an internal pull-up resistor (disconnected by default) of 20-50 kOhms. In addition, some pins have specialized functions:

- **Serial: 0 (RX) and 1 (TX); Serial 1: 19 (RX) and 18 (TX); Serial 2: 17 (RX) and 16 (TX); Serial 3: 15 (RX) and 14 (TX).** Used to receive (RX) and transmit (TX) TTL serial data. Pins 0 and 1 are also connected to the corresponding pins of the ATmega8U2 USB-to-TTL Serial chip .
- **External Interrupts: 2 (interrupt 0), 3 (interrupt 1), 18 (interrupt 5), 19 (interrupt 4), 20 (interrupt 3), and 21 (interrupt 2).** These pins can be configured to trigger an interrupt on a low value, a rising or falling edge, or a change in value. See the [attachInterrupt\(\)](#) function for details.
- **PWM: 0 to 13.** Provide 8-bit PWM output with the [analogWrite\(\)](#) function.
- **SPI: 50 (MISO), 51 (MOSI), 52 (SCK), 53 (SS).** These pins support SPI communication, which, although provided by the underlying hardware, is not currently included in the Arduino language. The SPI pins are also broken out on the ICSP header, which is physically compatible with the Duemilanove and Diecimila.
- **LED: 13.** There is a built-in LED connected to digital pin 13. When the pin is HIGH value, the LED is on, when the pin is LOW, it's off.
- **I<sup>2</sup>C: 20 (SDA) and 21 (SCL).** Support I<sup>2</sup>C (TWI) communication using the [Wire library](#) (documentation on the Wiring website). Note that these pins are not in the same location as the I<sup>2</sup>C pins on the Duemilanove.

The Mega2560 has 16 analog inputs, each of which provide 10 bits of resolution (i.e. 1024 different values). By default they measure from ground to 5 volts, though is it possible to change the upper end of their range using the AREF pin and [analogReference\(\)](#) function.

There are a couple of other pins on the board:

- **AREF.** Reference voltage for the analog inputs. Used with [analogReference\(\)](#).
- **Reset.** Bring this line LOW to reset the microcontroller. Typically used to add a reset button to shields which block the one on the board.



**radiospares**

**RADIONICS**



## Communication

The Arduino Mega2560 has a number of facilities for communicating with a computer, another Arduino, or other microcontrollers. The ATmega2560 provides four hardware UARTs for TTL (5V) serial communication. An ATmega8U2 on the board channels one of these over USB and provides a virtual com port to software on the computer (Windows machines will need a .inf file, but OSX and Linux machines will recognize the board as a COM port automatically). The Arduino software includes a serial monitor which allows simple textual data to be sent to and from the board. The RX and TX LEDs on the board will flash when data is being transmitted via the ATmega8U2 chip and USB connection to the computer (but not for serial communication on pins 0 and 1).

A [SoftwareSerial library](#) allows for serial communication on any of the Mega's digital pins.

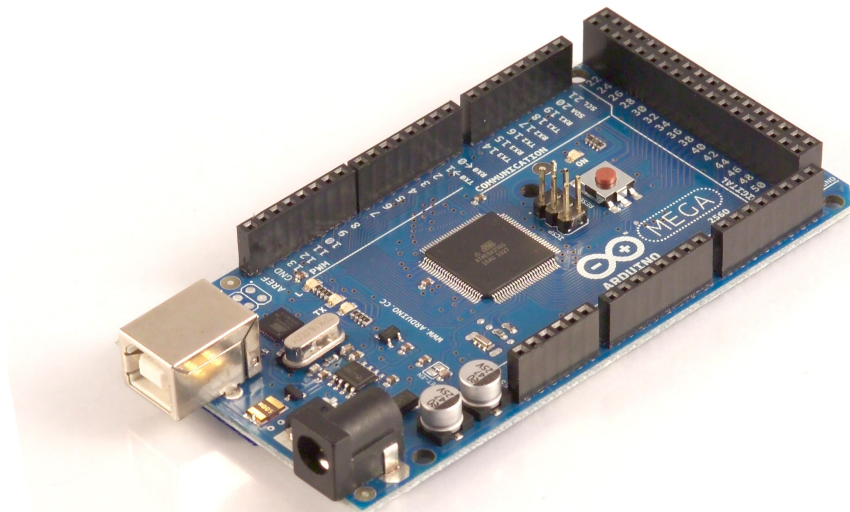
The ATmega2560 also supports I2C (TWI) and SPI communication. The Arduino software includes a Wire library to simplify use of the I2C bus; see the [documentation on the Wiring website](#) for details. To use the SPI communication, please see the ATmega2560 datasheet.

## Programming

The Arduino Mega2560 can be programmed with the Arduino software ([download](#)). For details, see the [reference](#) and [tutorials](#).

The ATmega2560 on the Arduino Mega comes preburned with a [bootloader](#) that allows you to upload new code to it without the use of an external hardware programmer. It communicates using the original STK500 protocol ([reference](#), [C header files](#)).

You can also bypass the bootloader and program the microcontroller through the ICSP (In-Circuit Serial Programming) header; see [these instructions](#) for details.



**radiospares**

**RADIONICS**



## Automatic (Software) Reset

Rather than requiring a physical press of the reset button before an upload, the Arduino Mega2560 is designed in a way that allows it to be reset by software running on a connected computer. One of the hardware flow control lines (DTR) of the ATmega8U2 is connected to the reset line of the ATmega2560 via a 100 nanofarad capacitor. When this line is asserted (taken low), the reset line drops long enough to reset the chip. The Arduino software uses this capability to allow you to upload code by simply pressing the upload button in the Arduino environment. This means that the bootloader can have a shorter timeout, as the lowering of DTR can be well-coordinated with the start of the upload.

This setup has other implications. When the Mega2560 is connected to either a computer running Mac OS X or Linux, it resets each time a connection is made to it from software (via USB). For the following half-second or so, the bootloader is running on the Mega2560. While it is programmed to ignore malformed data (i.e. anything besides an upload of new code), it will intercept the first few bytes of data sent to the board after a connection is opened. If a sketch running on the board receives one-time configuration or other data when it first starts, make sure that the software with which it communicates waits a second after opening the connection and before sending this data.

The Mega contains a trace that can be cut to disable the auto-reset. The pads on either side of the trace can be soldered together to re-enable it. It's labeled "RESET-EN". You may also be able to disable the auto-reset by connecting a 110 ohm resistor from 5V to the reset line; see [this forum thread](#) for details.

## USB Overcurrent Protection

The Arduino Mega has a resettable polyfuse that protects your computer's USB ports from shorts and overcurrent. Although most computers provide their own internal protection, the fuse provides an extra layer of protection. If more than 500 mA is applied to the USB port, the fuse will automatically break the connection until the short or overload is removed.

## Physical Characteristics and Shield Compatibility

The maximum length and width of the Mega PCB are 4 and 2.1 inches respectively, with the USB connector and power jack extending beyond the former dimension. Three screw holes allow the board to be attached to a surface or case. Note that the distance between digital pins 7 and 8 is 160 mil (0.16"), not an even multiple of the 100 mil spacing of the other pins.

The Mega is designed to be compatible with most shields designed for the Diecimila or Duemilanove. Digital pins 0 to 13 (and the adjacent AREF and GND pins), analog inputs 0 to 5, the power header, and ICSP header are all in equivalent locations. Further the main UART (serial port) is located on the same pins (0 and 1), as are external interrupts 0 and 1 (pins 2 and 3 respectively). SPI is available through the ICSP header on both the Mega and Duemilanove / Diecimila. **Please note that I<sup>2</sup>C is not located on the same pins on the Mega (20 and 21) as the Duemilanove / Diecimila (analog inputs 4 and 5).**



*radiospares*

**RADIONICS**



## **B.4 PulsedLight Lidar**



## Lidar Lite v3 Operation Manual and Technical Specifications

### Laser Safety

#### **⚠ WARNING**

This device requires no regular maintenance. In the event that the device becomes damaged or is inoperable, repair or service must be handled by authorized, factory-trained technicians only. Attempting to repair or service the unit on your own can result in direct exposure to laser radiation and the risk of permanent eye damage. For repair or service, contact your dealer or Garmin® for more information. This device should not be modified or operated without its housing or optics. Operating this device without a housing and optics, or operating this device with modified housing or optics that expose the laser source, may result in direct exposure to laser radiation and the risk of permanent eye damage. Removal or modification of the diffuser in front of the laser optic may result in the risk of permanent eye damage.

Use of controls or adjustments or performance of procedures other than those specified in this documentation may result in hazardous radiation exposure. Garmin is not responsible for injuries caused through the improper use or operation of this product.

#### **⚠ CAUTION**

This device emits laser radiation. This Laser Product is designated Class 1 during all procedures of operation. This designation means that the laser is safe to look at with the unaided eye, however it is advisable to avoid looking into the beam when operating the device and to turn off the module when not in use.

### Documentation Revision Information

Rev	Date	Changes
0A	09/2016	Initial release

## Table of Contents

<b>Lidar Lite v3 Operation Manual and Technical Specifications .....</b>	<b>1</b>
Laser Safety .....	1
Documentation Revision Information.....	1
<b>Specifications .....</b>	<b>2</b>
Physical .....	2
Electrical .....	2
Performance .....	2
Interface.....	2
Laser.....	2
<b>Connections.....</b>	<b>2</b>
Wiring Harness .....	2
Connector .....	2
Connector Port Identification .....	2
I2C Connection Diagrams .....	3
Standard I2C Wiring .....	3
Standard Arduino I2C Wiring .....	3
PWM Wiring.....	3
PWM Arduino Wiring.....	3
<b>Operational Information.....</b>	<b>4</b>
Technology .....	4
Theory of Operation.....	4
Interface.....	4
Initialization .....	4
Power Enable Pin .....	4
I2C Interface .....	4
Mode Control Pin.....	4
Settings.....	4
<b>I2C Protocol Information.....</b>	<b>6</b>
I2C Protocol Operation .....	7
Register Definitions .....	7
Control Register List .....	7
Detailed Control Register Definitions.....	8
<b>Frequently Asked Questions .....</b>	<b>12</b>
Must the device run on 5 Vdc? Can it run on 3.3 Vdc instead?.....	12
What is the spread of the laser beam?.....	12
How do distance, target size, aspect, and reflectivity effect returned signal strength?.....	12
How does the device work with reflective surfaces? .....	12
Diffuse Reflective Surfaces.....	12
Specular Surfaces .....	12
How does liquid affect the signal? .....	13

## Specifications

### Physical

Specification	Measurement
Size (LxWxH)	20 × 48 × 40 mm (0.8 × 1.9 × 1.6 in.)
Weight	22 g (0.78 oz.)
Operating temperature	-20 to 60°C (-4 to 140°F)

### Electrical

Specification	Measurement
Power	5 Vdc nominal 4.5 Vdc min., 5.5 Vdc max.
Current consumption	105 mA idle 135 mA continuous operation

### Performance

Specification	Measurement
Range (70% reflective target)	40 m (131 ft)
Resolution	+/- 1 cm (0.4 in.)
Accuracy < 5 m	±2.5 cm (1 in.) typical*
Accuracy ≥ 5 m	±10 cm (3.9 in.) typical Mean ±1% of distance maximum Ripple ±1% of distance maximum
Update rate (70% Reflective Target)	270 Hz typical 650 Hz fast mode** >1000 Hz short range only
Repetition rate	~50 Hz default 500 Hz max

\*Nonlinearity present below 1 m (39.4 in.)

\*\*Reduced sensitivity

### Interface

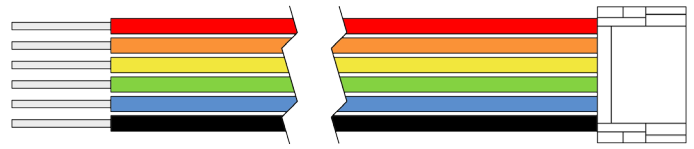
Specification	Measurement
User interface	I2C PWM External trigger
I2C interface	Fast-mode (400 kbit/s) Default 7-bit address 0x62 Internal register access & control
PWM interface	External trigger input PWM output proportional to distance at 10 µs/cm

### Laser

Specification	Measurement
Wavelength	905 nm (nominal)
Total laser power (peak)	1.3 W
Mode of operation	Pulsed (256 pulse max. pulse train)
Pulse width	0.5 µs (50% duty Cycle)
Pulse train repetition frequency	10-20 KHz nominal
Energy per pulse	<280 nJ
Beam diameter at laser aperture	12 × 2 mm (0.47 × 0.08 in.)
Divergence	8 mRadian

## Connections

### Wiring Harness



Wire Color	Function
Red	5 Vdc (+)
Orange	Power enable (internal pull-up)
Yellow	Mode control
Green	I2C SCL
Blue	I2C SDA
Black	Ground (-)

There are two basic configurations for this device:

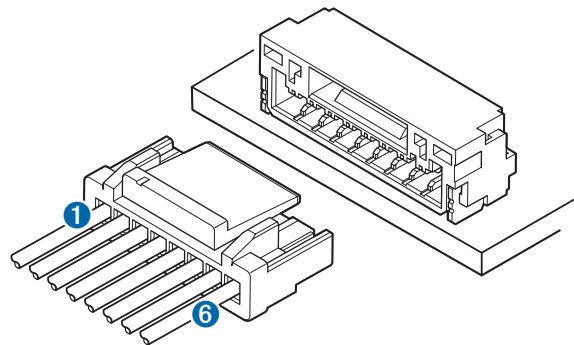
- **I2C (Inter-Integrated Circuit)**—a serial computer bus used to communicate between this device and a microcontroller, such as an Arduino board (“I2C Interface”, page 4).
- **PWM (Pulse Width Modulation)**—a bi-directional signal transfer method that triggers acquisitions and returns distance measurements using the mode-control pin (“Mode Control Pin”, page 4).

### Connector

You can create your own wiring harness if needed for your project or application. The needed components are readily available from many suppliers.

Part	Description	Manufacturer	Part Number
Connector housing	6-position, rectangular housing, latch-lock connector receptacle with a 1.25 mm (0.049 in.) pitch.	JST	GHR-06V-S
Connector terminal	26-30 AWG crimp socket connector terminal (up to 6)	JST	SSHL-002T-P0.2
Wire	UL 1061 26 AWG stranded copper	N/A	N/A

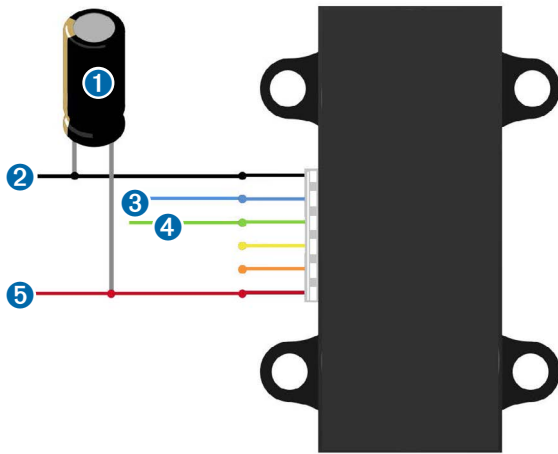
### Connector Port Identification



Item	Pin	Function
1	1	5 Vdc (+)
	2	Power enable (internal pull-up)
	3	Mode control
	4	I2C SCL
	5	I2C SDA
6	6	Ground (-)

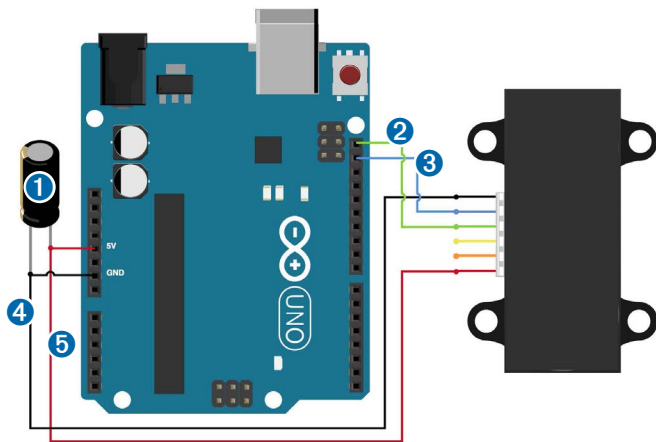
## I2C Connection Diagrams

### Standard I2C Wiring



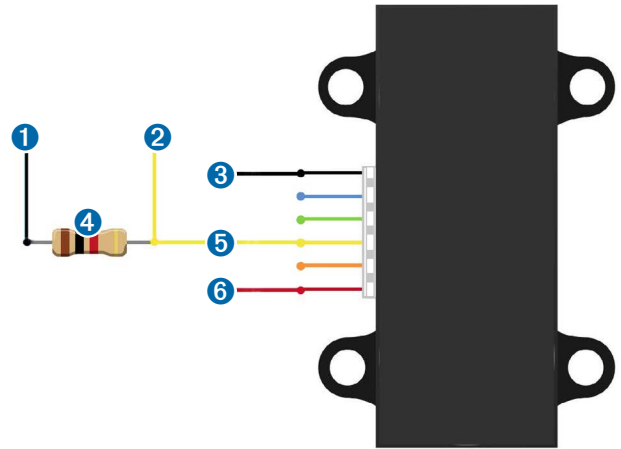
Item	Description	Notes
1	680µF electrolytic capacitor	You must observe the correct polarity when installing the capacitor.
2	Power ground (-) connection	Black wire
3	I2C SDA connection	Blue wire
4	I2C SCA connection	Green wire
5	5 Vdc power (+) connection	Red wire The sensor operates at 4.75 through 5.5 Vdc, with a max. of 6 Vdc.

### Standard Arduino I2C Wiring



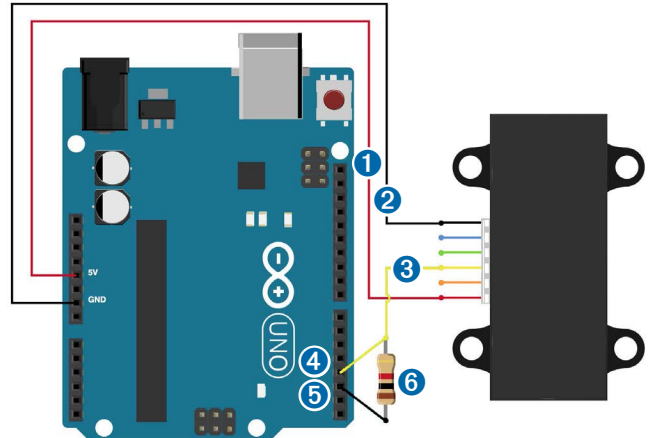
Item	Description	Notes
1	680µF electrolytic capacitor	You must observe the correct polarity when installing the capacitor.
2	I2C SCA connection	Green wire
3	I2C SDA connection	Blue wire
4	Power ground (-) connection	Black wire
5	5 Vdc power (+) connection	Red wire The sensor operates at 4.75 through 5.5 Vdc, with a max. of 6 Vdc.

### PWM Wiring



Item	Description	Notes
1	Trigger pin on microcontroller	Connect the other side of the resistor to the trigger pin on your microcontroller.
2	Monitor pin on microcontroller	Connect one side of the resistor to the mode-control connection on the device, and to a monitoring pin on your microcontroller.
3	Power ground (-) connection	Black Wire
4	1kΩ resistor	
5	Mode-control connection	Yellow wire
6	5 Vdc power (+) connection	Red wire The sensor operates at 4.75 through 5.5 Vdc, with a max. of 6 Vdc.

### PWM Arduino Wiring



Item	Description	Notes
1	5 Vdc power (+) connection	Red wire The sensor operates at 4.75 through 5.5 Vdc, with a max. of 6 Vdc.
2	Power ground (-) connection	Black Wire
3	Mode-control connection	Yellow wire
4	Monitor pin on microcontroller	Connect one side of the resistor to the mode-control connection on the device, and to a monitoring pin on your microcontroller.
5	Trigger pin on microcontroller	Connect the other side of the resistor to the trigger pin on your microcontroller.
6	1kΩ resistor	

## Operational Information

### Technology

This device measures distance by calculating the time delay between the transmission of a Near-Infrared laser signal and its reception after reflecting off of a target. This translates into distance using the known speed of light. Our unique signal processing approach transmits a coded signature and looks for that signature in the return, which allows for highly effective detection with eye-safe laser power levels. Proprietary signal processing techniques are used to achieve high sensitivity, speed, and accuracy in a small, low-power, and low-cost system

### Theory of Operation

To take a measurement, this device first performs a receiver bias correction routine, correcting for changing ambient light levels and allowing maximum sensitivity.

Then the device sends a reference signal directly from the transmitter to the receiver. It stores the transmit signature, sets the time delay for “zero” distance, and recalculates this delay periodically after several measurements.

Next, the device initiates a measurement by performing a series of acquisitions. Each acquisition is a transmission of the main laser signal while recording the return signal at the receiver. If there is a signal match, the result is stored in memory as a correlation record. The next acquisition is summed with the previous result. When an object at a certain distance reflects the laser signal back to the device, these repeated acquisitions cause a peak to emerge, out of the noise, at the corresponding distance location in the correlation record.

The device integrates acquisitions until the signal peak in the correlation record reaches a maximum value. If the returned signal is not strong enough for this to occur, the device stops at a predetermined maximum acquisition count.

Signal strength is calculated from the magnitude of the signal record peak and a valid signal threshold is calculated from the noise floor. If the peak is above this threshold the measurement is considered valid and the device will calculate the distance, otherwise it will report 1 cm. When beginning the next measurement, the device clears the signal record and starts the sequence again.

### Interface

#### Initialization

On power-up or reset, the device performs a self-test sequence and initializes all registers with default values. After roughly 22 ms distance measurements can be taken with the I2C interface or the Mode Control Pin.

#### Power Enable Pin

The enable pin uses an internal pullup resistor, and can be driven low to shut off power to the device.

#### I2C Interface

This device has a 2-wire, I2C-compatible serial interface (refer to I2C-Bus Specification, Version 2.1, January 2000, available from Philips Semiconductor). It can be connected to an I2C bus as a slave device, under the control of an I2C master device. It supports 400 kHz Fast Mode data transfer.

The I2C bus operates internally at 3.3 Vdc. An internal level shifter allows the bus to run at a maximum of 5 Vdc. Internal 3k ohm pullup resistors ensure this functionality and allow for a simple connection to the I2C host.

The device has a 7-bit slave address with a default value of 0x62. The effective 8-bit I2C address is 0xC4 write and 0xC5 read. The device will not respond to a general call. Support is not provided for 10-bit addressing.

Setting the most significant bit of the I2C address byte to one triggers automatic incrementing of the register address with successive reads or writes within an I2C block transfer. This is commonly used to read the two bytes of a 16-bit value within one transfer and is used in the following example.

The simplest method of obtaining measurement results from the I2C interface is as follows:

- 1 Write 0x04 to register 0x00.
- 2 Read register 0x01. Repeat until bit 0 (LSB) goes low.
- 3 Read two bytes from 0x8f (High byte 0x0f then low byte 0x10) to obtain the 16-bit measured distance in centimeters.

A list of all available control registers is available on [page 7](#).

For more information about the I2C protocol, see [I2C Protocol Operation \(page 7\)](#).

#### Mode Control Pin

The mode control pin provides a means to trigger acquisitions and return the measured distance via Pulse Width Modulation (PWM) without having to use the I2C interface.

The idle state of the mode control pin is high impedance (High-Z). Pulling the mode control pin low will trigger a single measurement, and the device will respond by driving the line high with a pulse width proportional to the measured distance at 10  $\mu$ s/cm. A 1k ohm termination resistance is required to prevent bus contention.

The device drives the mode control pin high at 3.3 Vdc. Diode isolation allows the pin to tolerate a maximum of 5 Vdc.

As shown in the diagram [PWM Arduino Wiring \(page 3\)](#), a simple triggering method uses a 1k ohm resistor in series with a host output pin to pull the mode control pin low to initiate a measurement, and a host input pin connected directly to monitor the low-to-high output pulse width.

If the mode control pin is held low, the acquisition process will repeat indefinitely, producing a variable frequency output proportional to distance.

The mode control pin behavior can be modified with the ACQ\_CONFIG\_REG (0x04) I2C register as detailed in [0x04 \(page 8\)](#).

#### Settings

The device can be configured with alternate parameters for the distance measurement algorithm. This can be used to customize performance by enabling configurations that allow choosing between speed, range and sensitivity. Other useful features are also detailed in this section. See the full register map ([Control Register List \(page 7\)](#)) for additional settings not mentioned here.

#### Receiver Bias Correction

Address	Name	Description	Initial Value
0x00	ACQ_COMMAND	Device command	--

- Write 0x00: Reset device, all registers return to default values
- Write 0x03: Take distance measurement without receiver bias correction
- Write 0x04: Take distance measurement with receiver bias correction

Faster distance measurements can be performed by omitting the receiver bias correction routine. Measurement accuracy and sensitivity are adversely affected if conditions change (e.g. target distance, device temperature, and optical noise). To achieve good performance at high measurement rates receiver bias correction must be performed periodically. The recommended method is to do so at the beginning of every 100 sequential measurement commands.

#### Maximum Acquisition Count

Address	Name	Description	Initial Value
0x02	SIG_COUNT_VAL	Maximum acquisition count	0x80

The maximum acquisition count limits the number of times the device will integrate acquisitions to find a correlation record peak (from a returned signal), which occurs at long range or with low target reflectivity. This controls the minimum measurement rate and maximum range. The unit-less relationship is roughly as follows: rate = 1/n and range = n<sup>(1/4)</sup>, where n is the number of acquisitions.



### Measurement Quick Termination Detection

Address	Name	Description	Initial Value
0x04	ACQ_CONFIG_REG	Acquisition mode control	0x08

You can enable quick-termination detection by clearing bit 3 in this register. The device will terminate a distance measurement early if it anticipates that the signal peak in the correlation record will reach maximum value. This allows for faster and slightly less accurate operation at strong signal strengths without sacrificing long range performance.

### Detection Sensitivity

Address	Name	Description	Initial Value
0x1c	THRESHOLD_BYPASS	Peak detection threshold bypass	0x00

The default valid measurement detection algorithm is based on the peak value, signal strength, and noise in the correlation record. This can be overridden to become a simple threshold criterion by setting a non-zero value. Recommended non-default values are 0x20 for higher sensitivity with more frequent erroneous measurements, and 0x60 for reduced sensitivity and fewer erroneous measurements.

### Burst Measurements and Free Running Mode

Address	Name	Description	Initial Value
0x04	ACQ_CONFIG_REG	Acquisition mode control	0x08
0x11	OUTER_LOOP_COUNT	Burst measurement count control	0x00
0x45	MEASURE_DELAY	Delay between automatic measurements	0x14

The device can be configured to take multiple measurements for each measurement command or repeat indefinitely for free running mode.

OUTER\_LOOP\_COUNT (0x11) controls the number of times the device will retrigger itself. Values 0x00 or 0x01 result in the default one measurement per command. Values 0x02 to 0xfe directly set the repetition count. Value 0xff will enable free running mode after the host device sends an initial measurement command.

The default delay between automatic measurements corresponds to a 10 Hz repetition rate. Set bit 5 in ACQ\_CONFIG\_REG (0x04) to use the delay value in MEASURE\_DELAY (0x45) instead. A delay value of 0x14 roughly corresponds to 100Hz.

The delay is timed from the completion of each measurement. The means that measurement duration, which varies with returned signal strength, will affect the repetition rate. At low repetition rates (high delay) this effect is small, but for lower delay values it is recommended to limit the maximum acquisition count if consistent frequency is desired.

### Velocity

Address	Name	Description	Initial Value
0x09	VELOCITY	Velocity measurement output	--

The velocity measurement is the difference between the current measurement and the previous one, resulting in a signed (2's complement) 8-bit number in cm. Positive velocity is away from the device. This can be combined with free running mode for a constant measurement frequency. The default free running frequency of 10 Hz therefore results in a velocity measurement in .1 m/s.

### Configurable I2C Address

Address	Name	Description	Initial Value
0x16	UNIT_ID_HIGH	Serial number high byte	Unique
0x17	UNIT_ID_LOW	Serial number low byte	Unique
0x18	I2C_ID_HIGH	Write serial number high byte for I2C address unlock	--
0x19	I2C_ID_LOW	Write serial number low byte for I2C address unlock	--
0x1a	I2C_SEC_ADDR	Write new I2C address after unlock	--
0x1e	I2C_CONFIG	Default address response control	0x00

The I2C address can be changed from its default value. Available addresses are 7-bit values with a '0' in the least significant bit (even hexadecimal numbers).

To change the I2C address, the unique serial number of the unit must be read then written back to the device before setting the new address. The process is as follows:

- 1 Read the two byte serial number from 0x96 (High byte 0x16 and low byte 0x17).
- 2 Write the serial number high byte to 0x18.
- 3 Write the serial number low byte to 0x19.
- 4 Write the desired new I2C address to 0x1a.
- 5 Write 0x08 to 0x1e to disable the default address.

This can be used to run multiple devices on a single bus, by enabling one, changing its address, then enabling the next device and repeating the process.

The I2C address will be restored to default after a power cycle.

### Power Control

Address	Name	Description	Initial Value
0x65	POWER_CONTROL	Power state control	0x80

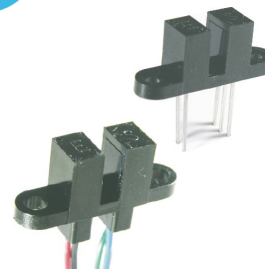
**NOTE:** The most effective way to control power usage is to utilize the enable pin to deactivate the device when not in use.

Another option is to set bit 0 in this register which disables the receiver circuit, saving roughly 40mA. After being re-enabled, the receiver circuit stabilizes by the time a measurement can be performed. Setting bit 2 puts the device in sleep mode until the next I2C transaction, saving 20mA. **Since the wake-up time is only around 2 m/s shorter than the full power-on time, and both will reset all registers, it is recommended to use the enable pin instead.**

## **B.5 Pickup Sensor**

# Photologic® Slotted Optical Switch

## OPB960, OPB970, OPB980, OPB990 Series



### Features:

- Choice of logic and output driver circuits
- Choice of aperture size, covered or open
- Wire or PCB leads
- Choice of mounting features
- Direct TTL, LSTTL, CMOS Interface

### Description:

The OPB960/ 970/ 980/ 990 series of non-contact Photologic® slotted optical switches provides flexibility in meeting application specific requirements for the design engineer.

Building from a standard housing with a 0.125" (3.18mm) wide slot, the user can specify output logic state, output driver circuit, aperture width, aperture surface and mounting tab locations. Furthermore, an option of wire or PCB leads allows electrical interface flexibility.

The device body is an opaque plastic which minimizes sensitivity to both visible and near-infrared external light sources which may impact operation. Aperture width choices provide different optical resolution for motion sensing. A covered aperture provides dust protection, while an open aperture provides maximum protection against external light sources.

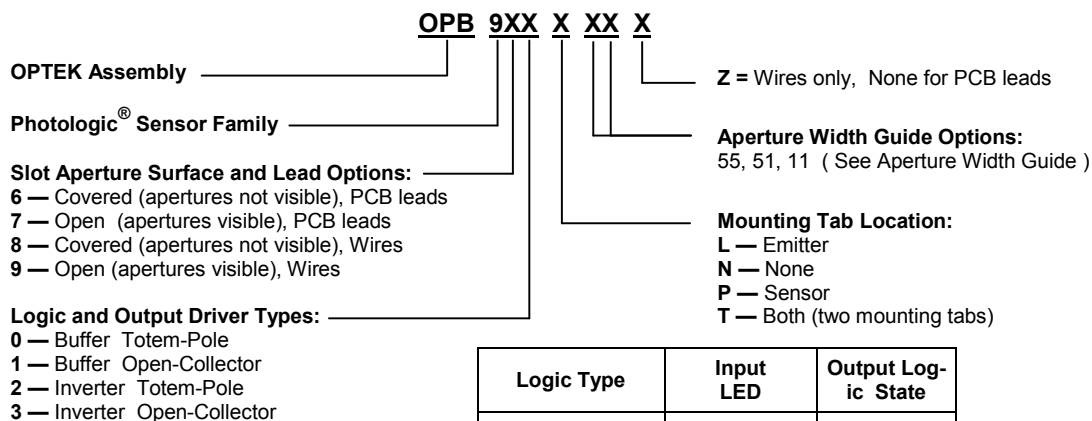
Electrical operation is over a wide supply voltage range. LED emissions are near-infrared (850—940nm). Detector digital output logic choices of buffer or inverter with totem-pole or open-collector driver circuit simplify interface for various electrical requirements.

Custom electrical, wire and cabling services are available. Contact your local representative or OPTEK for more information. Compliant to EU RoHS Directive 2002/95/EC

### Applications:

- Speed and direction indication
- Mechanical switch replacement
- Printers - Top of form, End of travel, Home position.
- Rotary encoders
- Mechanical limit indication
- Sliding Door Automotive and Lift gate applications

### Part Number Guide



Logic Type	Input LED	Output Logic State
Buffer	OFF	LOW = 0
Inverter	OFF	HIGH = 1



General Note  
TT Electronics reserves the right to make changes in product specification without notice or liability. All information is subject to TT Electronics' own data and is considered accurate at time of going to print.

OPTEK Technology, Inc.  
1645 Wallace Drive, Carrollton, TX 75006 | Ph: +1 972 323 2200  
[www.optekinc.com](http://www.optekinc.com) | [www.ttelectronics.com](http://www.ttelectronics.com)

# Photologic® Slotted Optical Switch

## OPB960, OPB970, OPB980, OPB990 Series

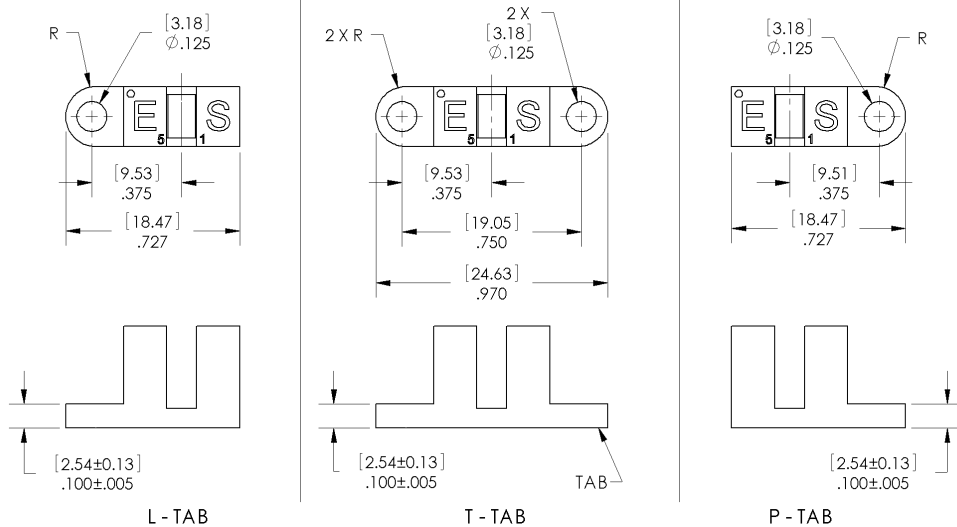
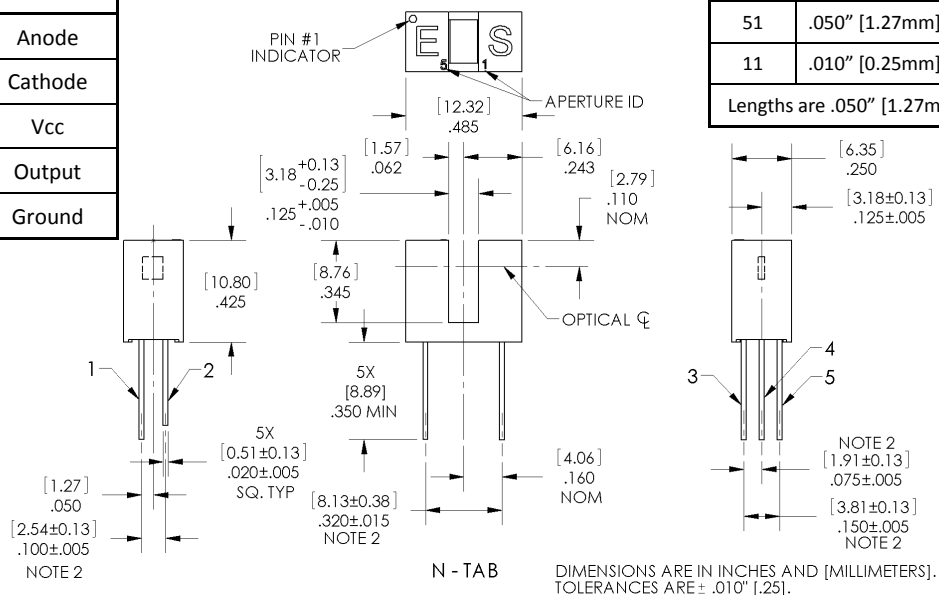


### PACKAGE OUTLINE for OPB960 and OPB970 Series

TABLE 1

Lead No.	Function
1	Anode
2	Cathode
3	Vcc
4	Output
5	Ground

APERTURE WIDTH GUIDE		
CODE	LED	SENSOR
55	.050" [1.27mm]	.050" [1.27mm]
51	.050" [1.27mm]	.010" [0.25mm]
11	.010" [0.25mm]	.010" [0.25mm]
Lengths are .050" [1.27mm]		



Notes:

- (1) RMA flux recommended. Duration can be extended to 10 seconds max.
- (2) Feature controlled at body.
- (3) Highly activated water soluble fluxes may attack plastic. Recommend trial to verify application.
- (4) Maximum lead soldering temperature [1.6mm from case for 5 seconds with soldering iron] 260° C.
- (5) Cathode lead may be shorter.
- (6) Part number marking may be on any side.

General Note  
TT Electronics reserves the right to make changes in product specification without notice or liability. All information is subject to TT Electronics' own data and is considered accurate at time of going to print.

OPTEK Technology, Inc.  
1645 Wallace Drive, Carrollton, TX 75006 | Ph: +1 972 323 2200  
[www.optekinc.com](http://www.optekinc.com) | [www.ttelectronics.com](http://www.ttelectronics.com)

# Photologic® Slotted Optical Switch



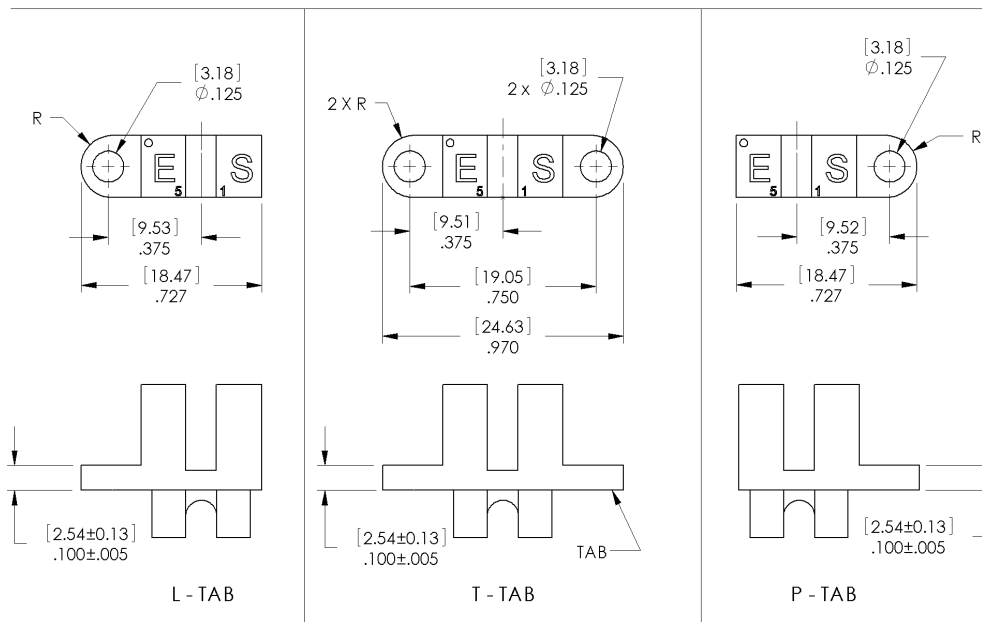
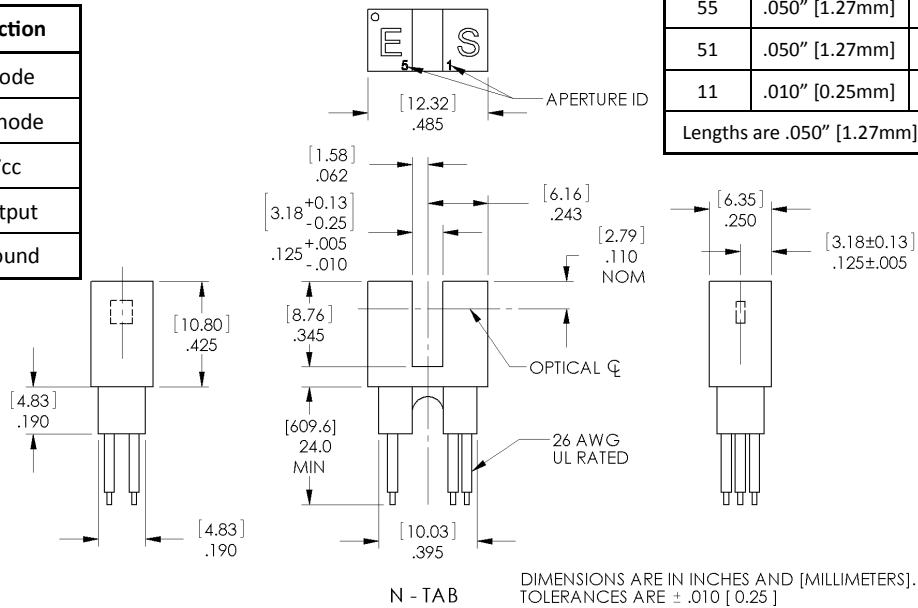
## OPB960, OPB970, OPB980, OPB990 Series

### PACKAGE OUTLINE for OPB980 and OPB990 Series

TABLE 2

Wire Color	Function
Red	Anode
Black	Cathode
White	Vcc
Blue	Output
Green	Ground

APERTURE WIDTH GUIDE		
CODE	LED	SENSOR
55	.050" [1.27mm]	.050" [1.27mm]
51	.050" [1.27mm]	.010" [0.25mm]
11	.010" [0.25mm]	.010" [0.25mm]
Lengths are .050" [1.27mm]		



**Notes:**

- (7) Wire is 26AWG, UL Rated PVC insulation.
- (8) Ideal torque for bolt or screw 0,45 to 0,68 Nm ( 4 to 6 Lb-in ).
- (9) When using a thread lock compound, ND Industries "ND Vibra-Tite" Formula 3" will avoid stress cracking plastic.
- (10) Plastic is soluble in chlorinated hydrocarbons and ketones. Methanol or isopropanol are recommended as cleaning agents.

General Note  
TT Electronics reserves the right to make changes in product specification without notice or liability. All information is subject to TT Electronics' own data and is considered accurate at time of going to print.

OPTEK Technology, Inc.  
1645 Wallace Drive, Carrollton, TX 75006 | Ph: +1 972 323 2200  
[www.optekinc.com](http://www.optekinc.com) | [www.ttelectronics.com](http://www.ttelectronics.com)

# Photologic® Slotted Optical Switch

## OPB960, OPB970, OPB980, OPB990 Series



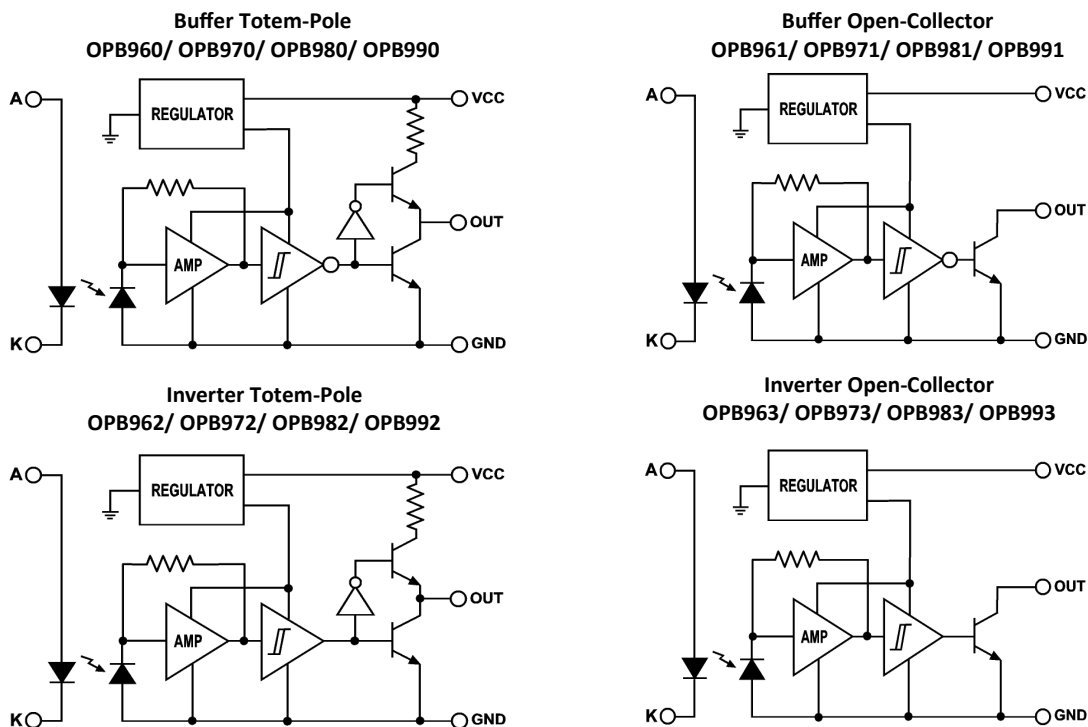
### Absolute Maximum Ratings ( $T_A = 25^\circ\text{C}$ unless otherwise noted)

Storage Temperature Range	-40°C to +85°C
Operating Temperature Range	-40°C to +70°C
<b>Input Diode (E)</b>	
Input Diode Power Dissipation	100 mW <sup>(11)</sup>
Input Diode Forward D.C. Current, $T_A = 25^\circ\text{C}$	40 mA <sup>(14)</sup>
Input Diode Reverse D.C. Voltage, $T_A = 25^\circ\text{C}$	2 V
<b>Sensor (S)</b>	
Supply Voltage ( $V_{CC}$ to Ground)	18 V <sup>(13)</sup>
Output Photologic® Power Dissipation	200 mW <sup>(12)</sup>
Voltage at Output Lead (Open-Collector Output), $T_A = 25^\circ\text{C}$	35V
Short Circuit Output Current to Ground ( $I_{OS}$ ) 1 sec Max.	30 mA

#### Notes:

- (11) Derate linearly 2.22 mW / °C above 25° C.
- (12) Derate linearly 4.44 mW / °C above 25° C.
- (13) Prior to 2004  $V_{CC}$  was limited to 5.5V maximum.
- (14) Do not connect input diode directly to a voltage source without an external current limiting resistor.

### Block Diagram



#### General Note

TT Electronics reserves the right to make changes in product specification without notice or liability. All information is subject to TT Electronics' own data and is considered accurate at time of going to print.

OPTEK Technology, Inc.  
1645 Wallace Drive, Carrollton, TX 75006 | Ph: +1 972 323 2200  
[www.optekinc.com](http://www.optekinc.com) | [www.ttelectronics.com](http://www.ttelectronics.com)

# Photologic® Slotted Optical Switch

## OPB960, OPB970, OPB980, OPB990 Series



Electrical Characteristics (T <sub>A</sub> = 25° C unless otherwise noted)						
SYMBOL	PARAMETER	MIN	TYP	MAX	UNITS	TEST CONDITIONS
<b>Input Diode</b> (See OP140 / OP240 LED for additional information)						
V <sub>F</sub>	Forward Voltage	-	-	1.70	V	I <sub>F</sub> = 20 mA, T <sub>A</sub> = 25° C
I <sub>R</sub>	Reverse Current	-	-	100	μA	V <sub>R</sub> = 2.0 V, T <sub>A</sub> = 25° C
<b>Coupled</b> (See OPL560 Detector for additional information)						
V <sub>CC</sub>	Operating D.C. Supply Voltage	4.5	-	16	V	
I <sub>CC</sub>	Supply Current	-	-	12	mA	V <sub>CC</sub> = 4.5V to 16V
V <sub>OL</sub>	Low Level Output Voltage: Buffer Totem-Pole OPB960,OPB970 OPB980,OPB990 Buffer Open-Collector OPB961,OPB971 OPB981,OPB991	-	-	0.4	V	V <sub>CC</sub> = 4.5V, I <sub>OL</sub> = 12.8mA I <sub>F</sub> = 0 mA <sup>(14)</sup>
	Inverter Totem-Pole OPB962,OPB972 OPB982,OPB992 Inverter Open-Collector OPB963,OPB973 OPB983,OPB993					V <sub>CC</sub> = 4.5V, I <sub>OL</sub> = 12.8mA I <sub>F</sub> = 15 mA
V <sub>OH</sub>	High Level Output Voltage: Buffer Totem-Pole OPB960,OPB970 OPB980,OPB990	V <sub>CC</sub> -2.1	-	-	V	V <sub>CC</sub> = 4.5V to 16V, I <sub>OH</sub> = 800μA I <sub>F</sub> = 15 mA
	Inverter Totem-Pole OPB962,OPB972 OPB982,OPB992					V <sub>CC</sub> = 4.5V to 16V, I <sub>OH</sub> = 800μA I <sub>F</sub> = 0 mA <sup>(14)</sup>
I <sub>OH</sub>	High Level Output Current: Buffer Open-Collector OPB961,OPB971 OPB981,OPB991	-	-	100	μA	V <sub>CC</sub> = 4.5V to 16V, V <sub>OH</sub> = 30V I <sub>F</sub> = 15 mA
	Inverter Open-Collector OPB963,OPB973 OPB981,OPB991					V <sub>CC</sub> = 4.5V to 16V, V <sub>OH</sub> = 30V I <sub>F</sub> = 0 mA <sup>(14)</sup>
I <sub>F</sub> (+)	LED Positive-Going Threshold Current <sup>(16)</sup>	-	-	15	mA	V <sub>CC</sub> = 5.0V, T <sub>A</sub> = 25° C
I <sub>F</sub> (+) / I <sub>F</sub> (-)	Hysteresis Ratio	-	1.5	-	-	V <sub>CC</sub> = 5.0V
t <sub>R</sub> , t <sub>F</sub>	Output Rise Time, Output Fall Time	-	70	-	ns	V <sub>CC</sub> = 5.0V, I <sub>F peak</sub> = 15 mA, T <sub>A</sub> = 25° C 100 kHz square wave, C = 10pF max.
t <sub>PLH</sub> , t <sub>PHL</sub>	Propagation Delay Time Low to High, High to Low	-	5.0	-	μs	R <sub>L</sub> = 360 Ω to GND (Totem-Pole) R <sub>L</sub> = 1KΩ pull-up (Open-Collector)

### Notes:

- 14) Normal application would be with light source blocked, simulated by I<sub>F</sub> = 0 mA.
- 15) All parameters are tested using pulse techniques.
- 16) An increasing current applied to the LED which causes the output logic state to change.  
For proper application I<sub>F</sub>(+), LED current, should be more than the stated maximum.

### General Note

TT Electronics reserves the right to make changes in product specification without notice or liability. All information is subject to TT Electronics' own data and is considered accurate at time of going to print.

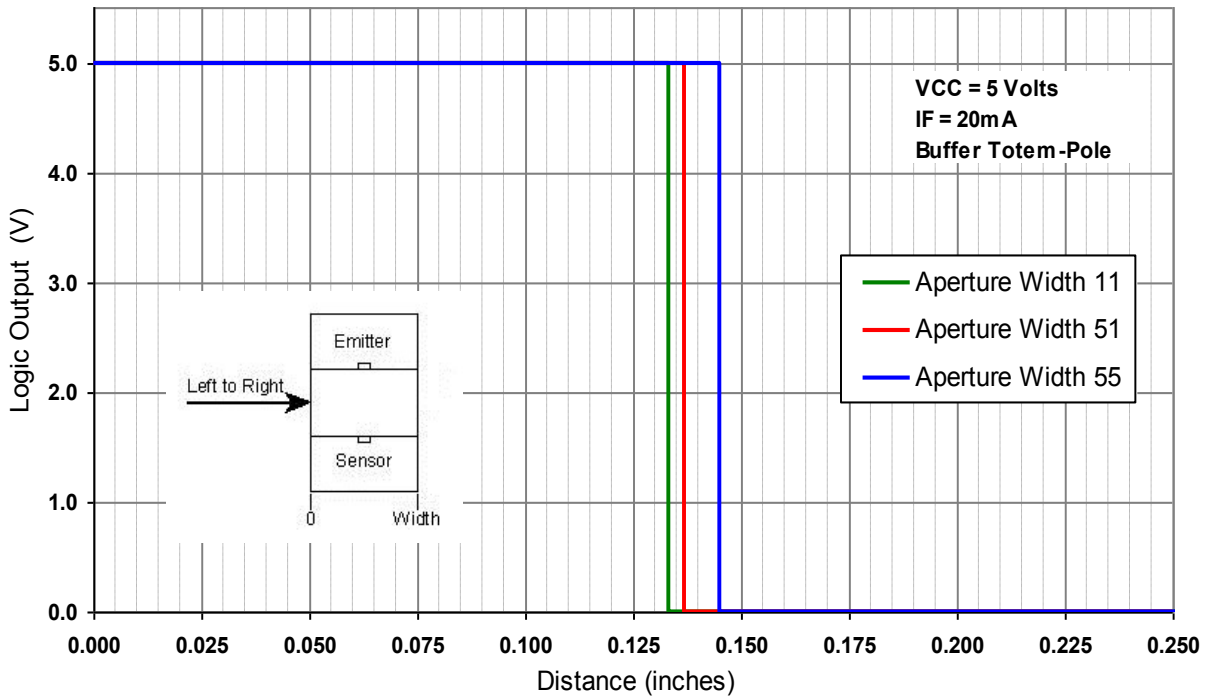
OPTEK Technology, Inc.  
1645 Wallace Drive, Carrollton, TX 75006 | Ph: +1 972 323 2200  
[www.optekinc.com](http://www.optekinc.com) | [www.ttelectronics.com](http://www.ttelectronics.com)

# Photologic® Slotted Optical Switch

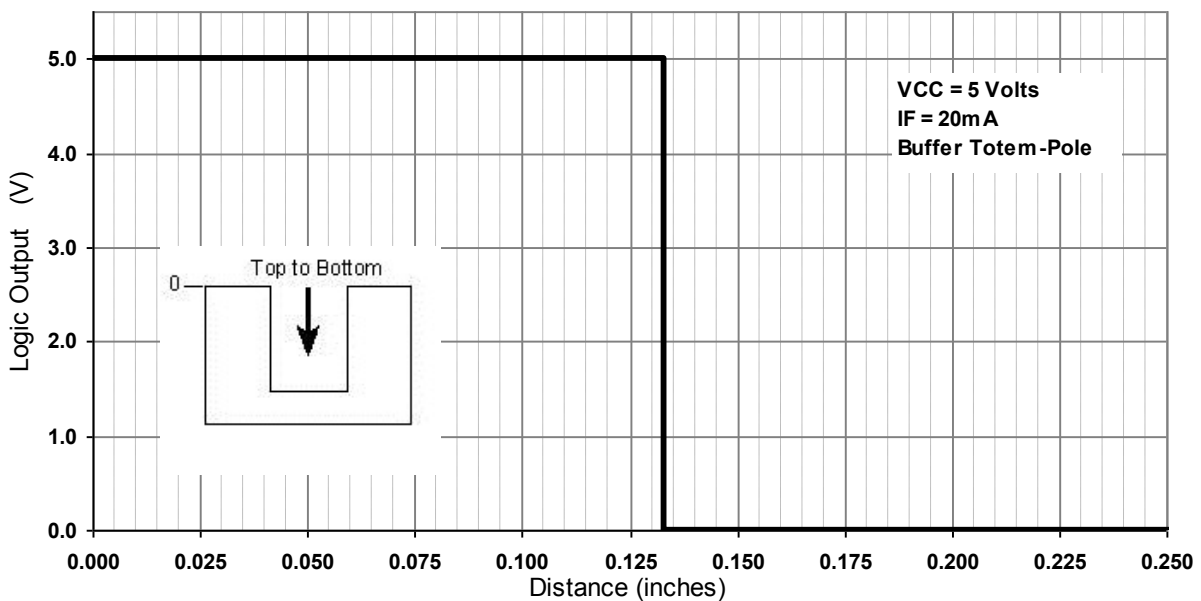
OPB960, OPB970, OPB980, OPB990 Series



Logic Output vs Left to Right Bocking Distance (X-Axis Blocked)



Logic Output vs Top to Bottom Bocking Distance (Y-Axis Blocked)



General Note  
 TT Electronics reserves the right to make changes in product specification without notice or liability. All information is subject to TT Electronics' own data and is considered accurate at time of going to print.

OPTEK Technology, Inc.  
 1645 Wallace Drive, Carrollton, TX 75006 | Ph: +1 972 323 2200  
[www.optekinc.com](http://www.optekinc.com) | [www.ttelectronics.com](http://www.ttelectronics.com)



## **B.6 Piksi RTK**



# Piksi Datasheet

Flexible, high-performance GPS receiver platform running open-source software

## Features

- Centimeter-accurate relative positioning (Carrier phase RTK)
- 10 Hz position/velocity/time solutions
- Open-source software and board design
- Low power consumption - 500mW typical
- Small form factor - 53x53mm
- USB and dual UART connectivity
- External antenna input
- Full-rate raw sample pass-through over USB

## Applications

- Autonomous Vehicle Guidance
- GPS/GNSS Research
- Surveying Systems
- Precision Agriculture
- Unmanned Aerial Vehicles
- Robotics
- Space Applications

## Overview

Piksi™ is a low-cost, high-performance GPS receiver with Real Time Kinematics (RTK) functionality for centimeter-level relative positioning accuracy.

Its small form factor, fast position solution update rate and low power consumption make Piksi ideal for integration into autonomous vehicles and portable surveying equipment.

Piksi's open source firmware allows it to be easily customized to the particular demands of end users' applications, easing system integration and reducing host system overhead.

In addition, Piksi's use of the same open source GNSS libraries as Peregrine, Swift Navigation's GNSS post-processing software, make the combination of the two a powerful toolset for GNSS research, experimentation and prototyping at every level from raw samples to position solutions.

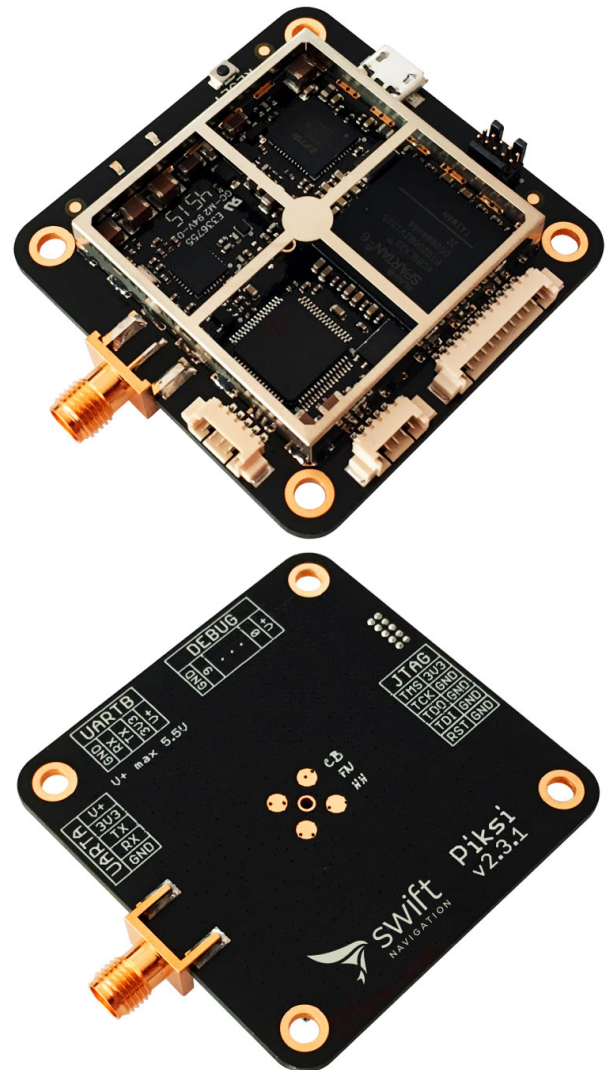


Figure 1: Piksi front and back view

With these tools, developers can quickly move from prototyping software on a desktop to running it standalone on the Piksi hardware.

A high-performance DSP on-board and our flexible Swift-NAP correlation accelerator provide Piksi with ample computing resources with which advanced receiver techniques, such as multipath mitigation, spoofing detection and carrier phase tracking can be implemented.

## System Architecture

The Piksi receiver architecture consists of three main components. The RF front-end downconverts and digitizes the radio frequency signal from the antenna. The digitized signal is passed into the SwiftNAP which performs basic filtering and correlation operations on the signal stream. The SwiftNAP is controlled by a microcontroller which programs the correlation operations, collects the results and processes them all the way to position/velocity/time (PVT) solutions.

### Front-end

The RF front-end consists of a Maxim MAX2769 integrated down-converter and 3-bit analog-to-digital converter operating at 16.368 MS/s. This front-end is capable of covering the L1 GPS signal bands.

### SwiftNAP

The SwiftNAP consists of a Xilinx Spartan-6 FPGA that comes pre-programmed with Swift Navigation's SwiftNAP

firmware. The SwiftNAP contains correlators specialized for satellite signal tracking and acquisition. The correlators are flexible and fully programmable via a high-speed SPI register interface and are used as simple building blocks for implementing tracking loops and acquisition algorithms on the microcontroller.

While the SwiftNAP HDL is not open-source at this time, the Piksi has no restrictions against loading one's own firmware onto the on-board Spartan-6 FPGA.

### Microcontroller

The on-board microcontroller is a STM32F4 with an ARM Cortex-M4 DSP core running at up to 168 MHz. This powerful processor performs all functions above the correlator level including tracking loop filters, acquisition management and navigation processing and is able to calculate PVT solutions at over 10 Hz in our default software configuration. All software running on the microcontroller is supplied open-source.

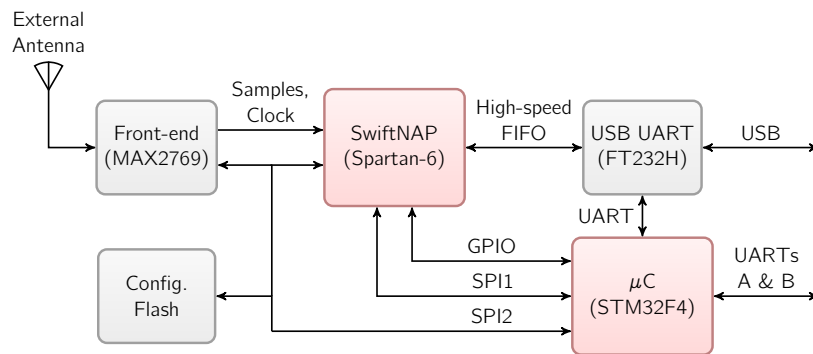


Figure 2: Piksi Block Diagram

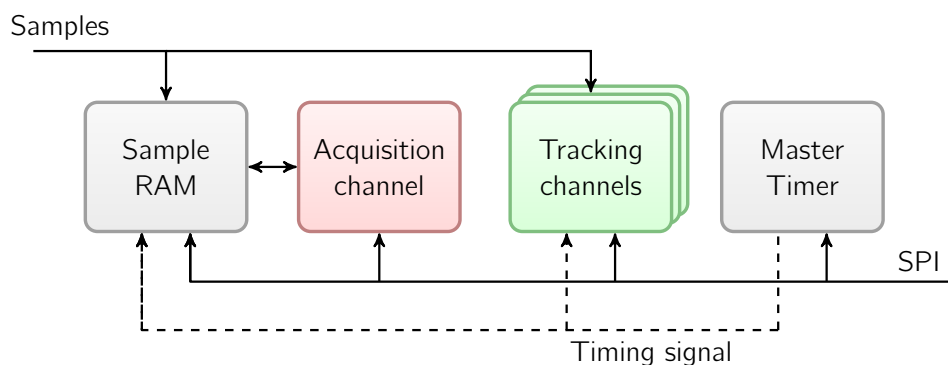


Figure 3: SwiftNAP Block Diagram