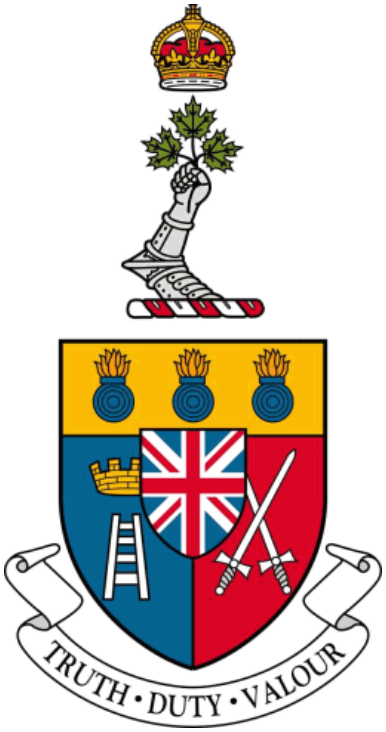
Royal Military College of Canada
Department of Electrical and Computer Engineering



Designing Coatimunde

Computer Optics Analyzing Trajectories In Mostly Unknown, Navigation Denied, Environments

DID-07 - Detailed Design Document

Presented by:
Amos Navarre Hebb & Kara Stephan

Presented to:
Dr. Sidney Givigi & Capt Anthony Marasco

March 21, 2019

Acknowledgement

We would like to thank our supervisor, Dr. Sidney Givigi, for first introducing us to the problem of real time obstacle avoidance and providing us access to the tools and robots we used to complete our work.

Abstract

The COATIMUNDE project attempts to extract target and depth information from a single camera feed source using the Robot Operating System as both an underlying architecture and a design philosophy. Four custom 'nodes' are created which interact with existing nodes necessary to execute Robot Operating System code. These nodes identify targets, identify depth information, record movement of the robot, and compile it into a single set of points which allow the program to send appropriate commands to the robot to navigate toward targets while avoiding obstacles.

Contents

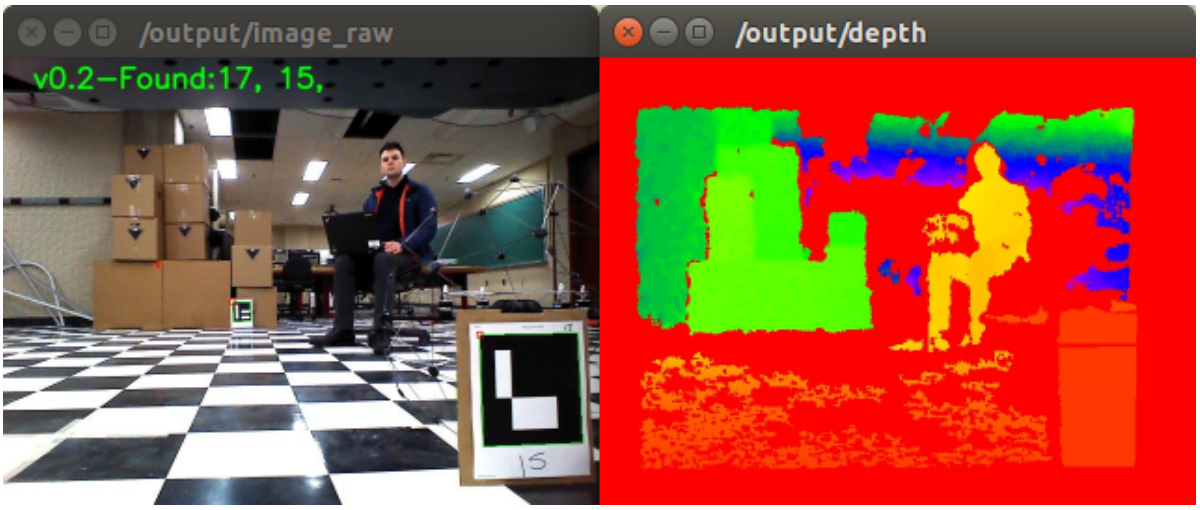


Figure 1: Visual Output of Running COATIMUNDE Software

1 Introduction

1.1 Document Purpose

Using Computer Optics for Analyzing Trajectories in Mostly Unknown, Navigation Denied, Environments (COATIMUNDE) is the goal of this project. The purpose of this document is to outline the detailed design for COATIMUNDE. That is, any deviations from the original design/plan, what the final design was, how it was built, and how this met the requirements of the project. The requirements for the project have been outlined in the Statement of Requirements. This design of the project is shown, through design artifacts, and discussed in the Design Section. The final results of the project are presented in the Results section, and the tests done to obtain these results are presented in the Verification and Validation section, some of which produced images like those shown in Figure 1. This document will then provide a summary of the degree of success of the project and provide feedback on the course experience as a whole.

1.2 Background

Both in the consumer and professional sectors the use of autonomous aerial vehicles is growing quickly. Currently these vehicles rely on skilled pilots to accomplish a very limited set of tasks. Adding obstacle avoidance capabilities to these vehicles and simplifying the task of following targets could allow for these systems to be used in many more situations. This section will give a quick background on obstacle avoidance, unmanned aircraft systems, computer vision, and the platforms we intend to use in this project.

1.2.1 Aim

The aim of this project was to design a high level control system that will allow an air robot to identify a target and move toward it, avoiding any obstacles that are in the way. Tracking targets of interest in complex environments with a flying robot was the ultimate goal of this project. To accomplish this goal we used a TurtleBot and then a flying robot through only the use of a camera to identify targets and obstacles.

1.2.2 Benefits

Having a flying robot capable of accomplishing the project's tasks while totally autonomous will allow for the use of flying robots in environments closer to the ground, and will assist pilots in complex environments. These general requirements can be used in many situations. These benefits for society are the motivation behind this project.

- **Surveillance:** The robot could follow an interesting object, especially in an urban environment, without colliding with obstacles.
- **Search and Rescue:** The robot could move toward visual way-points set by pilots while avoiding obstacles in complex environments assisting in search efforts.
- **Inspections:** The robot could inspect objects in hard to reach environments and complex environments like rooftops or bridges.
- **Disaster Relief:** The robot could check inside buildings that may have compromised structural integrity, rubble on ground, *&c.*
- **Agriculture:** The robot could inspect tree-fruit or crops that can not be observed from overhead or check assets in remote locations such as irrigation equipment.

1.2.3 Scope

The project started through the use of a ground robot then progressing to a flying robot. The project used computer vision to find targets and for avoiding multiple obstacles. Both robots are highly autonomous, requiring only user input to commence.

The scope of the project was limited due to only being able to test indoors. Ideally the UAV would be operating at a high speed and identifying an arbitrary target in an unpredictable environment. In this case though the testing was completed within the confines of a relatively small robotics laboratory. Smaller obstacles were used indoors simply due to space constraints. Lower speeds were used as well, again due to the lack of space.

1.2.4 Requirements

The requirements for the project are described in the Statement of Requirements (DID-03) [1]. The main functional requirements for this project were identification and movement towards a target within 15m radius, and identification and avoidance of multiple obstacles.

The initial requirements show certain assumptions that were made which can be attributed to ignorance at the time. The most notable of these, the use of a single video stream and exclusively OpenCV to parse depth data. It proved very difficult to extract reliable depth information from a single camera. Although we initially had identified a method using stereo cameras, the push-broom approach [2] we had initially explored, we had abandon this due to a belief that depth from a single camera would be similarly visible, and constrained ourselves in the requirements phase to this.

1.2.5 Preliminary Design

The preliminary design document outlines the initial design for the project [3]. The initial design was fairly robust, but certain aspects had intentionally not yet been selected. The main block diagram Figure 3 remains totally unchanged, the only additions are the specific topics being subscribed to by each of the custom modules.

The most substantial changes come to the actual actions to be performed by each node, given that the nodes were intentionally left fairly undefined and only their specific input and output were described this allowed for fairly substantial changes to the underlying logic that allows these nodes to complete their tasks.

1.2.6 Changes to Schedule

While ROS is a very flexible framework for writing robot software, there is a significant understanding of the underlying plumbing needed before some of the higher level capabilities can be used. Most of the changes to the schedule are a result of over-estimating how difficult it would be to implement some of the higher level aspects of this project, and significantly under-estimating how much time it would take to do simpler tasks.

1.2.7 Changes to Custom Nodes

There are many existing nodes that are similar to the nodes we had intended to create, but for the sake of simplicity these were initially ignored to create a very simple set of nodes according to our preliminary design. The most notable changes are only in what specific information was passed between different nodes, in particular, the way that depth information was parsed from a registered depth image. By parsing all information from a single depth image, including information about ArUco markers locations instead of attempting to use other mechanisms, the hope was that on the flying platform this information could be extracted in exactly the same way once a mechanism to extract depth information from a single camera image was developed.

1.2.8 Porting to UAV

Presumably delayed forever, the inertia to simply make this execute on a platform that allows very reliable feedback and predictable reactions to inputs means that, given our lack of exposure to the flying platform, it will not be implemented in its final form on a flying platform.

1.3 Definitions

1.3.1 Obstacle Avoidance

Obstacle avoidance is the task of satisfying a control objective, in this case moving toward a visual target, while subject to non-intersection or non-collision position constraints. The latter constraints are, in this case, to be dynamically created while moving in a reactive manner, instead of being pre-computed.

1.3.2 Unmanned Aircraft Systems

Very generally any powered vehicle that uses aerodynamic forces to provide lift, without a human operator being carried, can be considered an unmanned aerial vehicle. Currently most of these vehicles make up a single component of a larger unmanned aircraft system. An Unmanned aircraft system (UAS), or remotely piloted aircraft system (RPAS), is an aircraft without a human pilot on-board, instead controlled from an operator on the ground. Such a system can have varying levels of autonomy, something as simple as a model aircraft could be considered a UAS without any automation capabilities.[4]

1.3.3 Computer Vision

Currently there are many different ways that computers can make high-level decisions based on digital image information. There are many methods to acquire, process, and analyze data from the real world using a camera. While this is a very broad field, we intend to focus on motion estimation and object recognition. Both will be working with a video stream taken from a camera.

Motion estimation can be accomplished using direct methods which compare whole fields of pixels to each other over successive frames, compared to indirect methods which look for specific features. The information resulting from motion estimation streams can be used to both compensate for motion while analyzing other aspects of an image, and update a state machine.

Object recognition in our project will be accomplishing two tasks: identifying a marker or target which will require more involved object recognition calculations, and very simple techniques, such as edge detection, to identify obstacles that exist in the path of the robot.

1.3.4 OpenCV

The Open Source Computer Vision Library (OpenCV) of programming functions is a cross-platform and free for use collection of functions primarily aimed at real-time computer vision[5]. Most well documented techniques to accomplish all of the computer vision goals of our project have already been created and refined in OpenCV.[6] For this reason this project utilized many preexisting OpenCV functions.

1.3.5 Gazebo

Gazebo is a robot simulator that allows for creation of a realistic environment which includes both obstacles and markers similar to those being used in the lab. It was then used to rapidly test algorithms.

1.3.6 Robot Operating System

The Robot Operating System (ROS) is a distributed message system that allows for various sensors and processors to work together to control a robot. It is open source and has been integrated already with OpenCV and Gazebo. There are many additional tools for detecting obstacles, mapping the environment, planning paths, and much more. It is also a robust messaging system that has been proven to be capable of real-time processes.

1.3.7 gmapping

OpenSLAM has developed gmapping to create grid maps from laser range data. It uses a Rao-Blackwellized particle filter whe are each particle carries an individual map of the environment, and then the number of particles needed are reduced to create a more accurate map with significantly reduced uncertainty.[7]

While this process does rely on laser information, and it was only ever used in its unmodified form for this project, the intent was to port it to work with a slice from a depth registered image to allow it to work on a platform with only a single camera, a process that has been shown to turn a three-dimensional field into a two-dimensional map accurately.[8]

1.3.8 TurtleBot

The TurtleBot is a robot kit with open-source design and software. A TurtleBot is a robot built to the specification for TurtleBot Compatible Platforms[9]. In this case the TurtleBot has a Kobuki Base, an Acer Netbook running Ubuntu with ROS packages added, an X-Box Kinect, and some mounting plates.

The resulting robot looks like a disk supporting some circular shelves with a small laptop and a camera on the shelves. The base disk is 35.4cm in diameter, the topmost shelf is 42cm from the ground. The robot has a maximum speed of 0.65m/s.

1.3.9 AscTec Pelican

The Ascending Technologies Pelican is a 65.1cm by 65.1cm quad-copter designed for research purposes[10]. It includes a camera, a high level and low level processor set up for computer vision, and simultaneous localization and mapping (SLAM) research. It is also capable of interfacing easily with other controllers and can carry up to a kilogram of additional gear. This particular flying robot comes installed with a single camera, that was used to collect stereo data which was later used for depth information.

2 Design

2.1 Computer Vision

Computer optics was used as the robot employed a camera to identify objects in the surrounding environment, specifically targets and obstacles. There are three unique manners that computer vision was used as an input source for this project; these are target finding, obstacle finding, and optical flow. The implementation of all three created a navigation kit that exclusively uses a camera as input, resulting in a very transferable and lightweight package for flying robots.

2.1.1 Target Finding

While finding arbitrary targets would be preferred, this project is going to be limited to finding special targets designed to be easily identified in a busy environment called ArUco shapes. There are existing libraries in OpenCV that are useful for identifying ArUco shapes with very little overhead.

2.1.2 Obstacle Finding

Finding obstacles will be a considerable aspect of this project. There are various algorithms which leverage the robot's motion through the environment to extract key features of the environment, indicating the potential presence of obstacles. Parallax shift is the tendency for items that are closer to a camera to appear to move more or change in size more than a background that is further away and can be used to indicate where items are and how close they are. Occlusion is where one item moves in front of another item and covers it up, this indicates that the item still in view is closer to the camera than the item that has been hidden. OpenCV contains libraries for both of these tasks, as well as others that may end up being employed to identify obstacles in the environment.

2.1.3 Optical Flow

Optical Flow is a possible source of motion information for the robot. As a camera is moved through an environment the entire image will appear to scale larger as the camera moves forward, rotate left and right as a robot rolls, and shift left, right, up, and down as the robot pitches and yaws. Calculating shifts from one frame to another for global shifts caused by movement of the camera is a mathematically intensive process beyond the scope of this project. OpenCV contains libraries to try to parse this information, but it requires a considerable amount of effort to combine this information with a model of a robot to get useful position estimation. Given the extreme difficulty of extracting meaningful information from a camera, especially given the very constrained geometry of a single camera, means that this approach was not used in this project.

2.1.4 Pinhole Camera

The pinhole camera, in this context, is not an existing camera, but a mathematical construct which models the relationship between objects in three-dimensional space, and a projection onto an image plane. Images are grids of pixels, and ideally a straight line from any point on this grid, through a single point in front of the plane, would terminate at the real location in three-dimensional space of the object in the photo. Real cameras use lenses and imperfect capture devices which introduce distortion. This is ignored for the most part, and only objects in the center of the camera's field of vision are used in this project, this is computationally inexpensive and allows software to be ported between different cameras quickly.

2.1.5 Depth Registered Image

An image where pixels have an extra value: a distance in real space at a known scale, allowing for distances to targets and interesting points to be easily extracted from the image. These can be created in hardware using advanced sensors like the Astra Pro on the TurtleBot 2, or using other post-processing techniques.

2.1.6 Stereo Depth Estimation

By using two vantage points, depth information can be extracted by examining interesting points in the two vantage panels. Several pre-processing steps are required for this technique to be most effective, all of which are built in to OpenCV. Undoing distortion caused by cameras allows comparisons to be made between images and a three dimensional image to be rectified. This rectified image can be corrected using a known scale, then mapped back to one of the input images, creating a depth-registered image.

2.1.7 amcl

amcl is a probabilistic localization system for a robot moving in two dimensional space. Using adaptive Monte Carlo localization [14] it is able to quickly estimate the pose of a robot against a known two dimensional map. It is implemented as an already existing node in the standard ROS navigation stack, a stack containing code related to finding where the robot is and how it can get somewhere else.

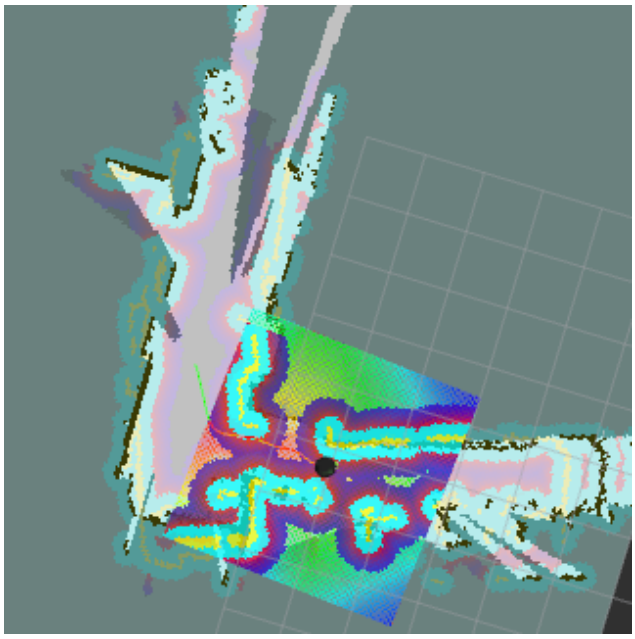


Figure 2: RViz Visualization of amcl Localization in Angled Hallway

2.1.8 RViz

RViz is a visualization tool for ROS, it contains many tools to observe output from various ROS topics. Most notably, there are mechanisms that allow for the visualization of the amcl mapping and goal finding processes showing the robot is a point navigating a two dimensional birds-eye-view of what it perceives its environment looks like. An example of typical RViz output while running amcl is seen in Figure 2.

2.2 Analyzing Trajectories

Analyzing trajectories is a term that refers to the robot completing movements to go towards the target. To do path planning the project will implement potential fields. Potential fields function through creating vectors pointing to objects in the environment and assigning them positive or negative values. The negative vectors attracts the robot towards the target and the positive vectors make obstacles repulsive to the robot [11]. This algorithm was selected to do path planning as the robot will not have to build a map of the environment to create path to the target [12]. Potential fields allow for the robot to change its path along the way if it discovers new obstacles. This approach to movement does not require complete knowledge of the environment it is navigating. In its simplest implementations the environment must be sparsely populated with obstacles, or simple things like straight lines can become 'wall traps' and a robot can become impossibly lost.

2.3 Obstacle Avoidance

There are multiple approaches to obstacle avoidance. The simplest require no prior knowledge of the environment they are navigating and simply react to obstacles as they appear. This approach can be useful for high-speed applications with sparse obstacles, [13] but become more limited in smaller spaces.

Using more complex algorithms that require more memory of the environment end up being easier when a complete map is saved. For this project the amcl localization system was used, which relies on a known map and fairly simple path planning algorithms which are able to solve more complex routes, at the expense of needing to spend more time estimating the robots position in the environment.[14]

2.4 Unknown and Navigation Denied Environments

The robot must always remember where identified target and obstacles are in relation to itself even when its own position changes. Navigation denied means that the robot will not have access to a GPS to locate itself, so it must search its unknown environment to identify targets and obstacles. An unknown environment can be defined as an environment in which the robot has no prior knowledge of its surroundings and must identify objects in this environment itself through computer vision. Potential fields will be used to track the location of the target and obstacles through updating each objects vector. These vectors will be used in the state machine to properly create a path to the target, as they are either negative for the target or positive for obstacles. This once again is the reason that the project will be using potential fields to plan the path to the target.

2.5 Block Diagram

The block diagram that is shown in Figure 3 contains the outline of the COATIMUNDE project. It is a very high level overview and does not contain references to most of the hardware in use for two reasons: the actual robot and laptop being used are relatively arbitrary as long as the software included is present on both. ROS also includes many nodes which are not represented on the block diagram. Only the nodes that are most important to our project, have been specifically added to ROS, or have been custom created are included on our system block diagram.

The laptop contains the initialization, target selection, and RViz. The laptop also takes the user input when necessary and processes it appropriately. The robot does all the target identification, obstacle avoidance, and movement decisions autonomously. This means it takes the input from the camera into a video processing node and then creates a corresponding ROS message to be sent to the other systems on the robot. These messages are created and used throughout the different systems on the robot. The model of the robot's environment is found through subscribing to messages from the obstacle finder node, target finder node, and the state estimate node. This model then creates a message to pass to the route planning node which tells the movement node how to update its position and speed.

Most of the nodes used in this project are either being used stock, or are very minimally modified from provided libraries. The four nodes, highlighted in grey, which have been custom built for this project are the State Estimate, Target Finder, and Obstacle Finder. Along with each of these functional names assigned, the actual nodal name for each of the project's custom nodes are listed.

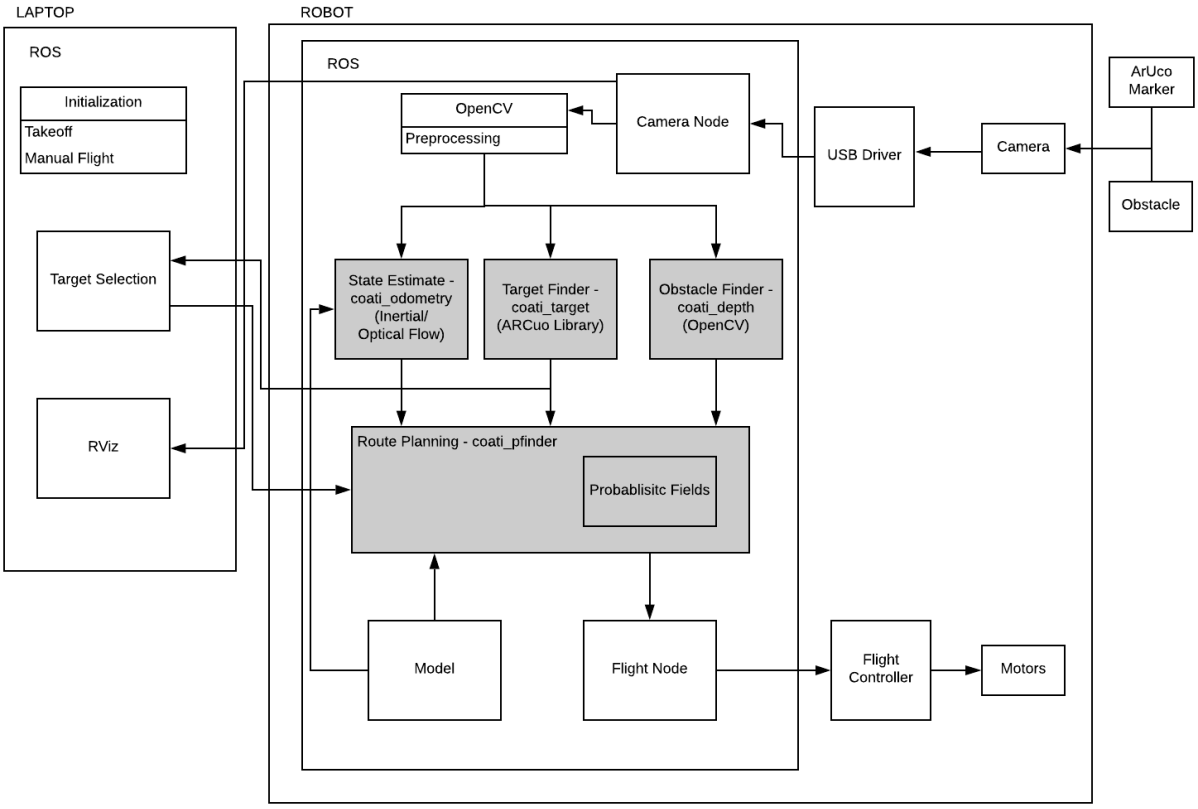


Figure 3: Project Block Diagram

2.6 Custom Nodes

All of the nodes that were written for this project were written in Python and then built in ROS to be implemented on the robot. Python was used for this project because C++ and Python are the most supported languages for both ROS Nodes and OpenCV. While C++ could possibly be faster in execution, we believe that most of our bottlenecks occur from the actual processing of images. The library calls to OpenCV were still executing in the lower level C that OpenCV was written in, so there would be minimal gains had we written it in C++.

The Camera node has been referenced in many locations. The camera node is built into ROS and provides image data in various formats. Processes can subscribe to these different image formats. In general the raw image information was used by some of the custom nodes and many other built in nodes. Common transformations that must be applied to all photographs will be added to the camera node. These transformations are to correct for the distortion that any camera has due to its own geometry.

2.6.1 Target Finding

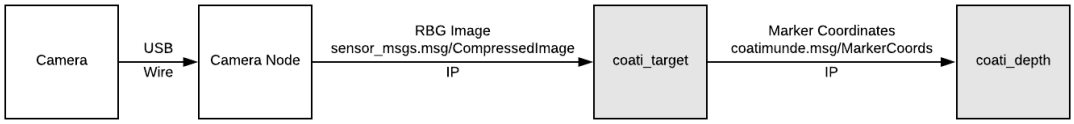


Figure 4: Target Finding Node

Camera data, after being processed by the camera node, goes into the target finder node. The target finder is mostly implementing the ArUco symbol finding libraries in OpenCV passing messages as shown in Figure 4. ArUco symbols are intended for augmented reality purposes and provide very accurate position, orientation, and distance information. These values are inserted into a marker coordinate message that the other systems on the robot can understand and be published at least ten times per second.

Given the limited field of view of the robot, it cannot always be able to see the target. The robot, in the pfinder node, saves a dictionary of points that are associated with targets that have been resolved. Only the most recent 'citing' of a target is recorded, and as soon as odometry updates are provided about the location of a target the last known location will be updated with these transformations.

2.6.2 Obstacle Finding

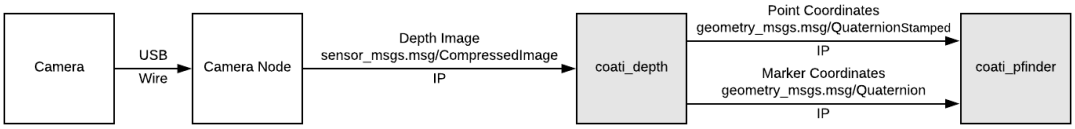


Figure 5: Obstacle Finding Node

Finding obstacles with depth information is difficult using monocular vision. The approach currently is to take advantage of the efficiency of the quaternion math allowing the finder node to update many points at once, and to sample many points along a plane at all times. When an image is processed, the depths along a row of pixels through the center of the image are converted into depth vectors which are published constantly according to the block diagram shown in Figure 5.

Getting depth information from a single camera is still a topic of ongoing research to which there are no simple existing solutions. The TurtleBot also has a suite of depth sensors built in with the camera so these may be used initially instead of trying to use parallax information from a camera to allow for testing other aspects of the system. Currently the approach, which will not work on the flying platform with simpler cameras, is to combine the depth information from another sensor with the image information, and then to sample depths from this.

A node should be created which uses the results from a convolutional neural network trained to extract depth information from images to create a similarly depth registered image file with the camera on the flying platform from a single camera source.

Targets, extracted from the images as well, report the depth to a point exactly in the centre of the target, and the same geometry used to calculate angles away on a camera is used to calculate where the vector to the target should be recorded. All targets are assumed to be in an even plane to the robot, so any x or y rotations are ignored, and it is assumed that targets only have z rotations and absolute distances. The current program, lacking a non-distorted image, only registers new targets that are in the centre of the camera range, and assumes that they are straight ahead.

The depth node subscribes to the target node to get the target locations. The depth node then uses the given location to find the depth value for the target. This target depth value is transformed in a quaternion and passed off to the path finder node.

2.6.3 Limitations of Obstacle Finding Approach

This approach proved sufficient in the lab environment, but is not very robust in larger environments and leaves the robot in a very reactive state. By not attempting to group together vectors which all point to a single obstacle, the robot was unable to navigate through many simple configurations of multiple obstacles, and never had any memory of an obstacle once it had passed it.

Error accumulation was also significantly faster for this reason, points would all diverge resulting in obstacles appearing larger and larger the longer the robot drove toward them. If instead an approach where the geometry of the obstacle was worked out and fixed-sized boxes were used internally to model obstacles, they would presumably not diverge in the same manner.

2.6.4 State Estimator

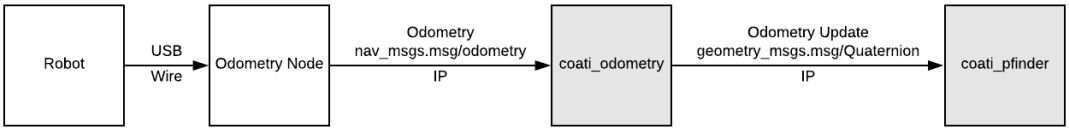


Figure 6: State Estimating Node

State estimating from camera information is very difficult and computationally expensive. The TurtleBot has stepper motors that can measure distances fairly reliably, and the Quadcopter has accelerometers that provide changes in position more reliably than image information could.

Even given these fairly reliable sources of odometry, error in the robots knowledge of its location in the environment were shockingly error prone. The need to apply some sort of particle filter became very apparent once serious testing was being performed.

Creation of a reliable system to extract pose information from camera information is well beyond the scope of this project.

These simpler forms of measuring the robot moving through the environment were used and the odometry node subscribed to the robot's odometry updates, according to Figure 6. This node turns all state updates into quaternions and publishes them for the route planner. The route planning node then adjusts the vectors to nearby obstacles and targets according to how much the robot has moved.

New position estimates should be provided at least 10 times every second in the form of a quaternion from the last position that the robot was in passed over a ROS message.

2.6.5 Route Planning

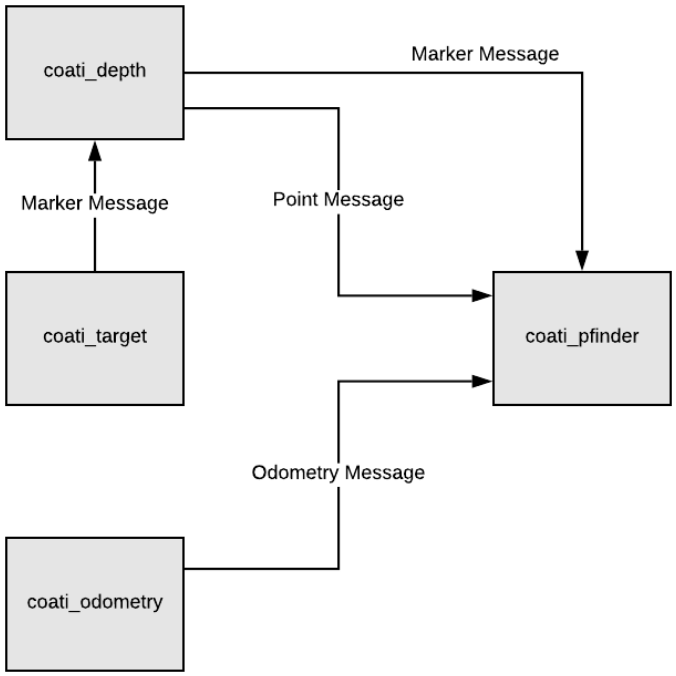


Figure 7: Route Planning Node

Route planning takes input from the other three custom nodes and determines a route that should carry it toward the target as shown in Figure 7. The implemented solution was to create a potential field where positive vectors that repel the robot are attached to all obstacles while a large negative vector is attached to the target location.

As the route node receives more information about the robot moving through the environment it applies transformations to all of the contained vectors to keep them up to date with where they are as the robot changes position in the environment.

Target locations are all stored in a separate array to remember the location of every target the robot finds. Differentiating targets is simple as each ArUco marker has an associated identification value. These values allow the node to know when a target has been re-spotted and over-write it or allow for finding and remembering the positions of multiple targets. If it discovers a new target it simply adds it to the array with its positional information.

2.6.6 PFinder v1

The first path finding method was a very simplistic and totally linear attempt at using a potential field system. Vectors toward all points in space, including obstacles, goals, and walls, were constantly updated and the robot would attempt to center itself along the vector pointing toward the goal, while all other vectors would only be avoided when dangerously close.

Error accumulation resulted in two things, the robot failing to reach the goals as soon as it was too close to capture the entire ArUco marker, meaning that effectively it still required line-of-sight, and all obstacles would appear dangerously close often after very little movement in their general direction.

The method to remove unnecessary or repetitive vectors for obstacles is to have these vectors simply expire after a certain amount of time. This amount of time, to be platform independent, is not a set value, but instead is based on the requirement to provide a new update 10 times every second. If, when applying

transformations to all stored vectors, the node requires more than 0.1 seconds to execute, it will then delete the oldest half of the nodes it is aware of. This continuous addition and replacement system will ensure that large numbers of points can be remembered, while ensuring that performance requirements are met.

2.6.7 PFinder v2

The navigation suite for ROS comes with many tools for more accurate state estimation. The biggest problem with PFinder v1 was the rate at which errors would accumulate, making any attempts to navigate with the ground robot not successful, and surely on the flying robot estimates would be totally useless.

The most interesting tool for our purposes is amcl. amcl is a localization system for robots moving in two dimensional planes. According to our scope both the TurtleBot and the flying robot will be able to do this using a very simple particle filter to track the pose of a robot. While this is the simplest, and therefore easiest to understand, approach to probabilistic localization, it also relies heavily on a known map, something our project definition does not allow.

In a small lab environment with a fairly high density of objects, slower but more accurate movement is desirable. So the decision to quickly develop a map and then try to navigate this was made. GMapping was used to create a map on startup, and then amcl would be used to navigate this newly created map. New maps could be created any time the robot became lost, allowing for the use of very short-ranged sensors and to enable navigation of a mostly unknown environment.

The PFinder v2 first rotates in place while running the gmapping node and records the locations in real space of all of the ArUco markers that are discovered using the existing output. These points are then stored in an array relative to the absolute position of a fixed marker, defined arbitrarily by the gmapping node. gmapping at the same time is creating a map using laser depth information from the Astra Pro, although the intention is to modify this behaviour to allow it to record information from the depth-registered image the depth node is still generating from the camera output.

Once a full rotation has been performed, the amcl node is launched, and PFinder provides a user with targets to select from. Once a user has selected a target for the robot to navigate toward its absolute location is passed into the `/move_base_simple/goal` topic as a goal to which ROS has existing logic to navigate toward. Solving the navigation to this goal is achieved using both the map created initially, overlaid with any currently occupied areas discovered by amcl. amcl naturally updates these locations based on current laser data, and will remove them if an obstacle is removed. This allows for the robot to make very accurate decisions while avoiding obstacles in real time.

2.7 Node Diagram

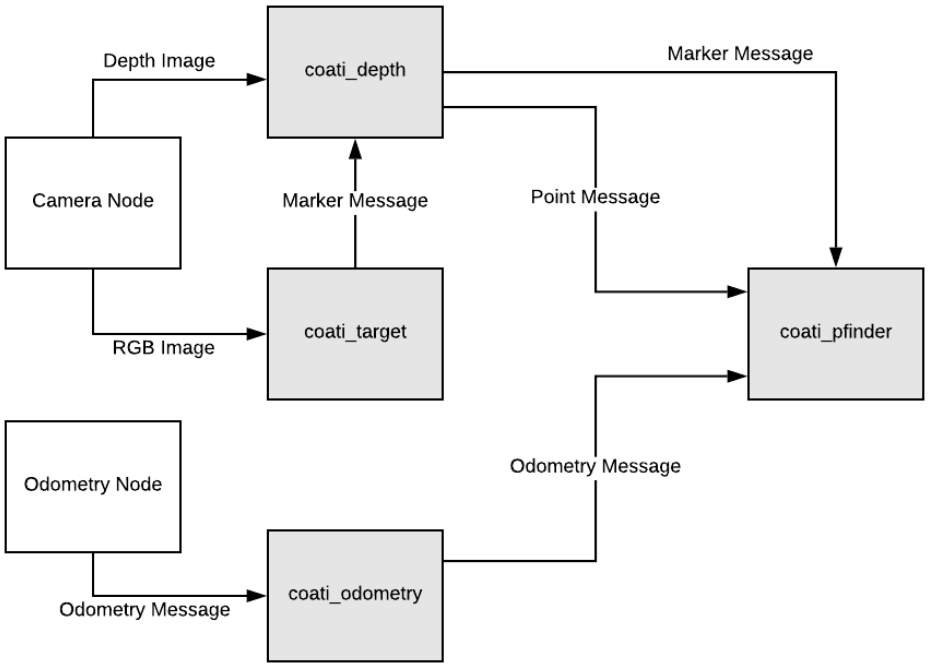


Figure 8: Project Node Diagram

Through Figure 8 it can been shown what nodes were created or used for this project. The diagram displays what node subscribed to the other nodes, such as the target finder node subscribed to the camera node to get messages about the target's potential location.

It is important to recognize that the directions indicated in Figure 8 are the flow of messages, not the direction of the subscriptions.

2.8 Mathematical Modeling

2.8.1 Flying Robot

Given that a quad-copter has six degrees of freedom, modeling them is a very difficult task. The AscTec Pelican already has many existing models intended for use with ROS. There is also a large collection of hardware agnostic tools for operating a quad-copter.

2.8.2 Initial Approach to Robot in 3D Space

The robot itself will need to remember where it is located in its environment. Remembering where targets are located relative to the robot is necessary, and presumably remembering the locations of obstacles will be used as well. Using quaternion vectors relative to the robot will allow for simple transformations on these vectors while using very little computational power [15]. ROS contains methods and messages to easily communicate, convert, and apply transformations to quaternions.

The limitations to this approach are the limited accuracy of the very small quaternions that result from real time odometry information from the robot, and the rate at which error accumulates when blindly trusting all robot odometry.

To combine two vectors, they are multiplied together. The result is a very accurate representation when both vectors are quite accurate and are similar in size, but when repeatedly multiplying a very small vector

with a larger vector the errors compound very quickly and points diverge and become useless far faster than we had anticipated.

2.8.3 Pose Estimation

Initially feedback from inertial sensors will be used for pose estimation, but ideally a version of Optical Flow would be used to estimate the position of the robot in 3D space. There are some libraries in OpenCV that have addressed generalized pose estimation in 6 degrees of freedom, but it requires a significant amount of computational power, a higher frame rate than our goal of 10Hz to be used for real time performance, and hundreds of visual odometry measurements must be made in every frame requiring a visually busy background. None of these libraries were explored seriously after seeing how fast error accumulated with the presumably far simpler to access and more reliable inertial sensors on board the platforms being used. Any optical flow based solution to pose estimation would need to be combined with a very robust probabilistic model of robot motion to be of any use at all.

2.8.4 Dieter Fox's Monte Carlo Localization

This approach to real-time localization attempts to find a match between a robot's current laser scan, and a known laser map generated before-hand. This is done using a particle filter, which is updated based on information from the robot's odometry information.

The approach is to assume the robot starts in the center of the map that it has just created, and then to create many potential locations the robot could be on the map and save these all as different possible locations. Every time a scan is performed, locations that have the smallest difference between the observed laser scan distances and the distances on the map are kept, while those with great variance are left behind. From the remaining particles, new ones are randomly scattered around based on a very simple probabilistic distribution and the process is constantly repeated.

Quickly the particles will converge and the robot will know where it is located in the environment.

2.9 Interfacing

The interfacing of almost all sensors with the on-board computer will be done over USB. Communication between different nodes on the same computer, and any computers off-board the robot, will be done exclusively with ROS messages sent over either internal loops or Wi-Fi.

3 Equipment

3.1 Equipment Table

A large number of items are being used in this project, most are arbitrary and may be changed in future iterations. Part way through our project we intend to re-implement the entire solution onto a different platform. A naming convention has been developed for all of our components using a three character code.

The first letter, ascending from A for hardware and descending from Z for software, represents the subsystem or general classification the equipment falls under. The second character represents a generic piece of equipment, and the third is for a particular example that we are using for our project. Any part with the same two first letters should be able to be substituted freely, an example may be two cameras. One camera (CAA) may be used on the TurtleBot as it is built in to the TurtleBot frame, while another (CAB) may be used on the flying robot as it is much lighter. The generic camera (CA-) can refer to any camera which provides the same information to the same nodes. The equipment table is shown in Table 1.

4 Verification and Validation

4.1 Unit Tests

When using ROS it is typical to create individual nodes that all execute independent of one another. Using this modular design, it is easy to write an entire node and verify it individually before executing it with the

Purpose		Description		Equipment	
A	Ground Based Robot	A	Robot	A	Clearpath Robotics TurtleBot 2
A	Ground Based Robot	B	Robot Controller	A	Acer Aspire E-11
A	Ground Based Robot	C	Robot Base	A	Kobuki Robot Base
A	Ground Based Robot	D	Sensors	A	Stepper Motor Feedback
A	Ground Based Robot	D	Sensors	B	Orbbec Astra Pro Sensors
A	Ground Based Robot	D	Sensors	C	Gyroscope
B	Flying Robot	A	Robot	A	AscTec Pelican Quadcopter
B	Flying Robot	B	Robot Controller	A	AscTec Low Level Processor
B	Flying Robot	B	Robot Controller	B	AscTec High Level Processor
B	Flying Robot	B	Robot Controller	C	ODROID-XU4
B	Flying Robot	C	Robot Base	A	AscTec Pelican Frame
C	Camera	A	USB Camera	A	Orbbec Astra Pro Camera
C	Camera	A	USB Camera	B	oCam-1MGN-U Plus
C	Camera	B	Digital Camera	A	AscTec Option 4
D	Offboard Computer	A	Computer	A	Lenovo T520
D	Offboard Computer	A	Computer	B	Mathworks VM - ROS Gazebo v3
E	Testing Hardware	A	ArUco Symbol	0-9	Printed Single Digit ArUco Symbol
E	Testing Hardware	B	Physical Obstacle	A	Human
E	Testing Hardware	B	Physical Obstacle	B	Box
E	Testing Hardware	B	Physical Obstacle	C	Sheet
E	Testing Hardware	C	Motion Tracking	A	Reflective Dot
E	Testing Hardware	C	Motion Tracking	B	Motion Tracking Cameras
E	Testing Hardware	D	Camera Calibration	A	Printed 7cm Checker Pattern
F	Interfacing	A	Wireless Router	A	D-Link 2.4 GHz Router
S	Linux Operating System	A	Operating System	B	Ubuntu 18.04 Bionic
S	Linux Operating System	A	Operating System	K	Kubuntu (Mathworks) 14.04
S	Linux Operating System	A	Operating System	T	Ubuntu 14.04 Trusty
S	Linux Operating System	B	Bundled Program	A	Generic Loopback Device
S	Linux Operating System	C	Version Control	A	git
T	ROS Service - Custom	A	Action Services	A	Set Goal
U	ROS Service - Standard	A	Action Services	A	Trigger
V	ROS Message - Custom	A	Action Messages	A	Goal
V	ROS Message - Custom	A	Action Messages	B	Obstacle
W	ROS Message - Standard	A	Sensor Messages	A	Image
W	ROS Message - Standard	A	Sensor Messages	B	PointCloud
W	ROS Message - Standard	B	Geometry Messages	A	Twist
W	ROS Message - Standard	B	Geometry Messages	B	Quaternion
W	ROS Message - Standard	B	Geometry Messages	C	Transform
X	ROS Node - Custom	A	Depth	A	coati_depth
X	ROS Node - Custom	B	Path Finder	A	coati_pfinder
X	ROS Node - Custom	B	Odometry	B	coati_odom
X	ROS Node - Custom	B	Vision	D	coati_target
Y	ROS Node - Standard	A	Vision	A	vision_opencv
Y	ROS Node - Standard	A	Vision	B	image_pipeline
Y	ROS Node - Standard	B	Coordinates	A	tf
Y	ROS Node - Standard	C	Vision	B	image_pipeline
Z	Robot Operating System	A	Operating System	A	ROS Indigo Igloo
Z	Robot Operating System	A	Operating System	B	ROS Melodic Morenia
Z	Robot Operating System	B	Bundled Program	A	Gazebo2
Z	Robot Operating System	B	Bundled Program	B	Gazebo7
Z	Robot Operating System	B	Bundled Program	C	catkin
Z	Robot Operating System	B	Bundled Program	D	RViz
Z	Robot Operating System	B	Bundled Program	E	rosbag
Z	Robot Operating System	B	Bundled Program	F	rqt
Z	Robot Operating System	C	Client	A	roscpp
Z	Robot Operating System	C	Client	B	rospy

Table 1: Equipment

entire system. Verification and validation of the project's nodes will be done prior to implementation in the Gazebo simulation environment.

4.1.1 coati_target tests

The target node was the easiest to verify, as it was only looking for the markers in their positions within an image. The ArUco marker library has tools built in to draw an outline around the images, and the id of the marker was normally overlaid on the image as well. In the final version of the target node, the id number is placed in the upper left corner of the image to allow other information to be placed on the image.

4.1.2 coati_depth tests

The depth node required more extensive verification. It exists in two forms, but the version that was verified is using the depth registered images which combine lidar and camera feeds into a unified image. The images were then normalized and a colour gradient was applied based on the distance of each pixel. This allowed for visual verification of the depth values being read by the robot at a glance, seeing something change colour as it is moved closer or further.

For more accurate testing, objects of known distances away were checked against the value generated in the program. After verifying that the depth registered versions of images were accurate, the depth registered images from the combined sensor source will be used as a reference to verify other methods of parsing depth information.

4.1.3 coati_odom tests

The incoming odometry data is parsed and passed along, due to the high frequency which this data arrives at, the transformations are all very small. It is easy to print a stream of values as they are moving along to verify that they make sense, and because the robot will return values even when not being driven it is possible to turn both the TurtleBot and the AscTec Pelican by hand and observe what values the odometry reports back.

If a version of odometry was created that did try to use optical flow from visual information, it would be possible to compare the values it generated against those collected by the sensors on the robot.

4.1.4 coati_pfinder tests

PFinder is the highest level of logic on the robot, and therefore relied on all of the lower level nodes co-operating.

Given that all of the other nodes were verified independently, it was obvious that the unexpected and undesirable behaviour, most notably just driving directly into targets, was a result of over-reliance on odometry information and errors in the models being used.

4.2 Gazebo

Gazebo is a simulation environment that runs nodes that give similar input and output to nodes being executed on a real robot moving around in a real environment. This includes camera feeds, and allows for the insertion of obstacles and ArUco shapes. Testing of custom written nodes were done in Gazebo congruently with unit-tests to ensure that nodes which rely on subscribing to other nodes are behaving as expected before executing code on an actual robot.

Gazebo ended up not being employed extensively due to the simplicity of actually executing code on the TurtleBot, combined with the difficulty of creating environments in Gazebo with all of the aspects we would like compared with the simplicity of moving boxes around in the lab and placing arbitrary ArUco markers wherever they were wanted.

4.3 Camera Tests

Given that most of the project's custom nodes revolve around transforming camera data to extract information, the ArUco shapes for targets, parallax shift for obstacles, and optical flow for odometry, tests performed

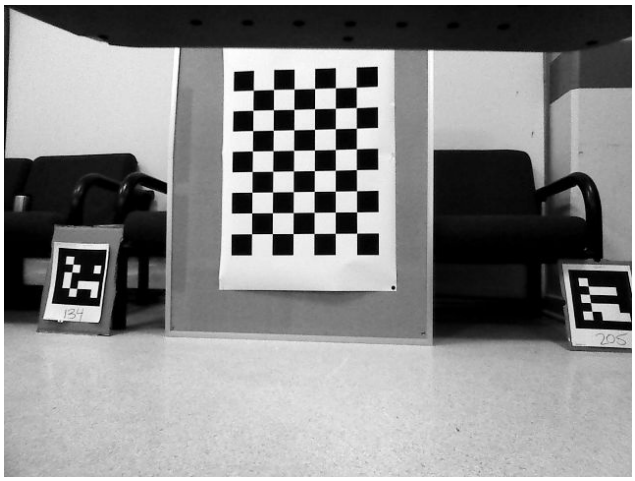


Figure 9: Example of Calibration Image

with only the camera and OpenCV outside of a node can be created and and executed without being contained within a ROS node. This was useful as one can use OpenCV's feature highlighting capabilities to ensure that the targets and obstacles were being identified while moving a camera through an environment.

Large collections of images taken with calibration grids, like those shown in Figure 9, were used when running different tests of the camera. These image sets allowed for significantly quicker verification of all aspects of the camera.

4.4 Ground Based Robot

Testing the code on a ground based robot was necessary to ensure that the logic in the movement nodes is sound. Initial testing was done through simply having the robot follow moving targets or turn to face targets that are not head on. Initially simply watching the robot move away from obstacles was considered sufficient testing. When the project was sufficiently developed the testing was having the robot avoid obstacles and move towards a target. As the project moved closer to porting to the flying robot, the ground robot was tested to ensure it met the requirement outlined in the SOR [1].

4.5 Flying Robot

Due to time constraints along with difficulty in simply starting flying robots, this platform was not extensively tested for.

The cameras used in developing the monocular to depth vision were those mounted on the flying robots and the video collected was using these platforms.

5 Results - NOT DONE

- presentation of results - prove we met the requirements

- we need to get some data points to prove the target and obstacles are found - depth readings - able to move towards the target? - talk about what happened with the obstacle avoidance and why it doesn't work

5.1 Finding Targets

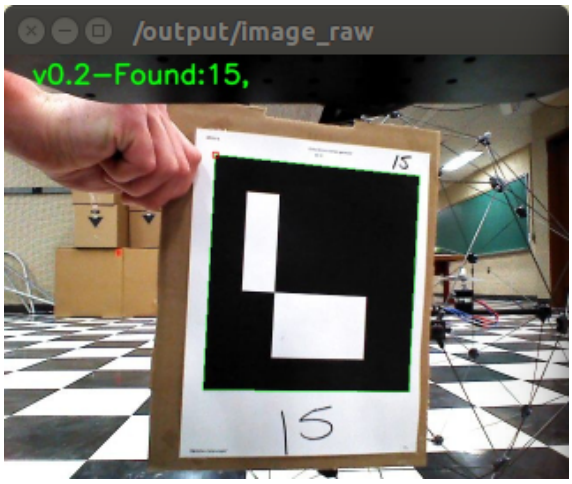


Figure 10: Positive Target Identification

Finding targets, especially how they were defined, was fairly simple to perform by leveraging existing libraries. Every image that is published to the camera topic on by the robot is taken in and saved both in colour and in grayscale. The grayscale image is then passed into an ArUco searching library, which returns an array containing arrays which contain the id and corners of every marker discovered in the image.

Overlaying these corners on the colour image, as well as outputing the id number and any other necessary information was the easiest way to ensure that this program was discovering targets reliably. This frame is then published as its own image topic, an example of this output is shown in Figure 10.

5.2 Extracting Depth Information

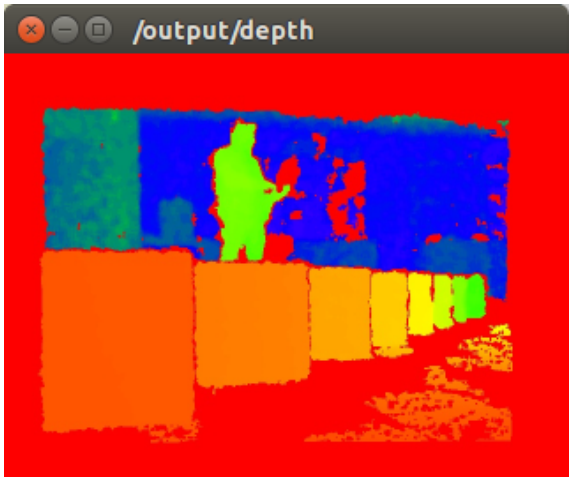


Figure 11: Registered Depth Image

Extracting depth information, while using the TurtleBot, was done by using the depth registered images which use Astra Pro hardware to combine image information with depth information from a Lidar. This composite image can be coloured based on the depth values alone, and targets can be plotted on it for easy verification that samples are being taken from appropriate locations and interesting points. An example of

the depth information is shown in Figure 11, which shows multiple boxes at different depths displaying an obvious gradient in colours.

5.3 Compiling Odometry Information

5.4 Path Finding

6 Discussion

6.1 Overview

The design chosen was very capable of moving toward targets under control, and could even recognize, in the most abstract sense of the term, obstacles in its environment. Using a totally reactive approach felt overly simplistic, and was not interesting especially given the extremely small working space the lab provided.

This caused the change in scope that resulted in this project not being ported to the flying robot. Instead of focusing on implementing an overly simplistic avoidance algorithm on an increasingly difficult platform to operate, the choice to build a greater memory into the program was made to allow for more complex obstacle avoidance.

6.2 Flying Robot Requirements

The majority of the requirements outlined were related to implementing code on a flying robot, so although there is existing solutions that have been verified to most of the requirements, the requirement that they be done on a flying robot leaves them not met.

Given that all of our solutions have been left highly portable, there is no software bound reason that our solution would not execute the same on board the flying robot, but the difficulty of operating a flying robot has prevented us from having opportunities to verify that everything performs as expected.

6.3 Major Issues

The biggest issue that plagued us for the entire duration of the project was the challenge of extracting depth information from a monocular camera. The flying robots only came with a monocular camera, and although initially we had intended to simply attach a second camera the decision to use this single camera was taken.

After a significant amount of time trying to parse depth information using OpenCV modules, it was discovered that it may be easier to use some sort of approach rooted in more modern machine learning algorithms.

The machine learning approach did provide interesting results, but it was never implemented into a depth module. The depth module simply used the hardware registered depth information that was available from the TurtleBot's onboard sensors. These sensors did provide very accurate information, but were implemented later than useful simply to allow for the development of other modules, and would not be able to be ported over to the flying robot.

Not having a verified manner to parse depth information that could be implemented on the flying robot left us wanting to push the driving platform to be able to perform more interesting obstacle avoidance.

6.4 Recommendations for Future Expansion

To expand further on this project, it would be necessary to find a testing environment that allows for more complex tests to take place. Even on the TurtleBot there were space limitations inside the lab to lay out obstacles in ways that could be navigated while still leaving far walls observable some of the time.

Obtaining permission to test in a larger enclosed space would enable far more testing on both the ground and flying platforms at speeds where the models used may actually be able to produce useful results.

The reactive and totally potential field based navigation was somewhat overkill when useful, and then when used with many obstacles and lower speeds became inadequate due to the way errors would accumulate.

The amcl approach was able to navigate quite complex environments, but would be useless in truly unknown environments or while moving at higher speeds.

Using either the pushbroom algorithm or some other mechanism of sensing depth in the environment would result in a significantly more reliable system, and modeling more things as vectors against a central 'odom' or 'map' point would also prevent accumulating errors from causing as many problems as our earlier pfinder modules ran in to.

6.5 Functional and Performance Requirements

Index	Description of Requirement	Result	Comment
FR-01	The TurtleBot shall move towards a target under control	Met	
FR-02	The TurtleBot shall recognize optically an OpenCV's ArUco shape and be able to give information on the location of the target relative to the robot	Met	
FR-03	The TurtleBot shall recognize obstacles in its environment and identify where they are relative to itself	Met	Simply as areas that are occupied
FR-04	The TurtleBot shall be able to make a deviation from its current movement pattern to avoid an obstacle in its path and then return to its pattern	Met	
FR-05	The TurtleBot shall be able to identify multiple obstacles and avoid them accordingly	Partially Met	No concept of a particular obstacle, only points that are close
PR-01	The TurtleBot shall be able to identify and locate OpenCV's ArUco shapes within a 15m radius	Met	

Table 2: Summary of Results for the TurtleBot

The requirements for the TurtleBot can be seen in Table 2. The reason FR-04 was not satisfied on the TurtleBot is the same reason that was discussed in the results section. FR-05 also was not fully satisfied, the TurtleBot was able to recognize multiple obstacles at once and register each as a potential obstacle. The ability to avoid multiple obstacles, or to remember obstacles as objects based on having seen them multiple times from different angles, does not exist as the memory is too short term.

Index	Description of Requirement	Result	Comment
FR-01	The Flying Robot shall move towards a target under control	Not Met	Flying Robot was not Flown
FR-02	The Flying Robot shall recognize optically an OpenCV's ArUco shape and be able to give information on the location of the target relative to the robot	Met	By subscribing to Flying Robots Camera Node
FR-03	The Flying Robot shall recognize obstacles in its environment and identify where they are relative to itself	Not Met	Depth from single camera not verified
FR-04	The Flying Robot shall be able to make a deviation from its current movement pattern to avoid an obstacle in its path and then return to its pattern	Not Met	
FR-05	The Flying Robot shall be able to identify multiple obstacles and avoid them accordingly	Not Met	
PR-01	The Flying Robot shall be able to identify and locate OpenCV's ArUco shapes within a 15m radius	Met	When processed off-board
PR-02	The Flying Robot should be able to process an image and plot obstacles at a rate of 10 Hz	Met	When processed off-board
PR-03	The Flying Robot shall be able to navigate an environment and detect up to two obstacles simultaneously	Not Met	

Table 3: Summary of Results for the Flying Robot

Given that the flying robot was not implemented, most of these requirements are not met simply for the reason that it was not possible to verify if our solutions did work as expected on the flying robot.

6.6 Interface Requirements

Index	Description of Requirement	Result	Comment
IT-01	The Gazebo simulator shall be run on a laptop and robots within the simulation will be interfaced through ROS	Met	
IT-02	Communication to the TurtleBot will be done through ROS over USB	Met	
IT-03	The Flying Robot will be communicating through ROS over WiFi wireless network	Met	
IT-04	A simple interface, perhaps command line, will allow a user to select a marker as a target to start the robot's movement towards the target and this signal should be received over wireless transmission	Met	

Table 4: Summary of Results for the Interface Requirements

6.7 Simulation Requirements

Index	Description of Requirement	Result	Comment
SimR-01	A Gazebo simulation environment which roughly approximates a lab environment with marker placed around will be created	Partially Met	Environments were created but not used
SimR-02	Stationary obstacles will be added to the lab simulation environment and a TurtleBot shall navigate toward markers in the lab while avoiding obstacles	Not Met	
SimR-03	An environment with obstacles will be created a Flying Robot shall navigate toward markers in the simulation while avoiding obstacles	Not Met	Flying Robot was never used in Gazebo

Table 5: Summary of Results for the Simulation Requirements

The use of simulation was limited due to the ease of operating the TurtleBot in the lab, combined with the ease of modifying the lab environment by simply moving boxes instead of having to make changes to the simulation environment.

6.8 Implementation Requirements

Index	Description of Requirement	Result	Comment
ImpR-01	OpenCV programs shall be created that can use a single video stream and identify both markers and obstacles	Met	
ImpR-02	Prior to testing on the TurtleBot, the program shall be implemented on the gazebo simulation	Met	
ImpR-03	Using the robot in the simulation environment the appropriate components, tools, and libraries to interpret and OpenCV stream, make decisions based on the environment, and execute instructions will be developed	Partially Met	Initial confirmation performed in Gazebo
ImpR-04	The simplest obstacle avoidance algorithm must be implemented on a TurtleBot using ROS	Met	
ImpR-05	The obstacle avoidance algorithm used for a Flying Robot will be implemented in a simulation	Not Met	

Table 6: Summary of Results for the Implementation Requirements

6.9 Schedule Requirements

Index	Description of Requirement	Result	Comment
SchR-01	The first TurtleBot simulation shall be able to operate in a Gazebo environment no later than November 5^{th}	Met	
SchR-02	The first functional prototype shall be a TurtleBot robot capable of positively identifying a marker, moving towards the marker, and avoiding an obstacle placed in its environment no later than December 18^{th}	Met	
SchR-03	The first functional prototype shall be capable of identifying a marker, moving towards the marker, and avoiding an obstacle placed in its environment no later than February 18^{th}	Met	
SchR-04	Data Item Descriptions (DID) to be presented are DID-04: Preliminary Design Specification due November 22^{nd}, DID-05: Preliminary Design Review Presentations due November 29^{th}, and the DID-06: Schedule Update is due January 17^{th}. The DID-07: Final Detailed Design Document is due March 21^{st}, the Final Project Presentation, DID-08, is March 28^{th} and the Final Project Demonstration, DID-09, is on April 9^{th}	Met	

Table 7: Summary of Results for the Schedule Requirements

7 Conclusion

The detailed design has been shown for Computer Optics for Analyzing Trajectories in Mostly Unknown, Navigation Denied, Environments (COATIMUNDE). The background outlined how the project has progressed over the year and the motivation behind it. It gave a summary of the previous Data Item Deliverables, such as the Statement of Requirements and the Preliminary Design Document. The design section laid out the final design of the project and how it was implemented. The verification and validation section explained how different aspects of the project were tested throughout the year. The results section showed the final results and how they proved we met the requirements previously outlined. A discussion section gave a detailed analysis of the design in comparison with the requirements and gave recommendations for future expansions of the project. The final design laid out in this document will also be used in the the Final Project Presentation and the Final Project Demonstration.

References

[1] K. Stephan and A. N. Hebb, "Data item deliverable 03- statement of requirements", 2018.

[2] A. J. Barry and R. Tedrake, "Pushbroom stereo for high-speed navigation in cluttered environments", in *Robotics and automation (icra), 2015 ieee international conference on*, IEEE, 2015, pp. 3046–3052.

[3] K. Stephan and A. N. Hebb, "Data item deliverable 04- preliminary design specification", 2018.

[4] *Royal canadian air force — news article — update and new name for the joint unmanned surveillance target acquisition system (justas) project*, Jul. 2018. [Online]. Available: `http://www.rcaf-arc.forces.gc.ca/en/article-template-standard.page?doc=update-and-new-name-for-the-joint-unmanned-surveillance-target-acquisition-system-justas-project/j9u7rzyf`.

[5] *About - opencv library*. [Online]. Available: `https://opencv.org/about.html`.

[6] A. C. Woods and H. M. La, "Dynamic target tracking and obstacle avoidance using a drone", in *International Symposium on Visual Computing*, Springer, 2015, pp. 857–866.

[7] G. Grisetti, C. Stachniss, and W. Burgard, *Gmapping*. [Online]. Available: `https://openslam-org.github.io/gmapping.html`.

[8] P. R. Florence, J. Carter, J. Ware, and R. Tedrake, "Nanomap: Fast, uncertainty-aware proximity queries with lazy search over local 3d data", *arXiv preprint arXiv:1802.09076*, 2018.

[9] M. Wise and T. Foote, *Rep: 119 - specification for turtlebot compatible platforms*, Dec. 2011. [Online]. Available: `http://www.ros.org/reps/rep-0119.html`.

[10] *Asctec pelican - uas for computer vision and slam*. [Online]. Available: `http://www.asctec.de/en/uav-uas-drones-rpas-roav/asctec-pelican/`.

[11] Y. K. Hwang and N. Ahuja, "A potential field approach to path planning", *IEEE Transactions on Robotics and Automation*, vol. 8, no. 1, pp. 23–32, 1992.

[12] S. A. Bortoff, "Path planning for uavs", in *American Control Conference, 2000. Proceedings of the 2000*, IEEE, vol. 1, 2000, pp. 364–368.

[13] A. J. Barry, P. R. Florence, and R. Tedrake, "High-speed autonomous obstacle avoidance with pushbroom stereo", *Journal of Field Robotics*, vol. 35, no. 1, pp. 52–68, 2018.

[14] A. Doucet, N. D. Freitas, and N. Gordon, *Sequential Monte Carlo methods in practice*. Springer, 2001.

[15] B. Williams and I. Reid, "On combining visual slam and visual odometry", in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, IEEE, 2010, pp. 3494–3500.