Monocular Visual SLAM and Object Pose Estimation

Generating 3D Maps and Identifying Objects
using a single video feed

Domhnall ó Póil
15520787

Final Year Project – 2019
B.Sc. Single Honours in
Computer Science and Software Engineering

Department of Computer Science
Maynooth University
Maynooth, Co. Kildare
Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc.
Single Honours Computer Science and Software Engineering
Supervisor: **Dr. John McDonald**

Table of Contents:

Abstract i

Declaration ii

Acknowledgements iii

List of Figures iv

Chapter one: Introduction 1

1.1 Topic addressed in this project 1

1.2 Motivation 1

1.3 Problem Statement 2

1.4 Methodology 2

1.5 Project Milestones 2

Chapter two: Technical Background 4

2.1 Technologies 4

2.2 Technical material 5

Chapter three: The Problem 7

3.1 Challenges 7

3.1.1 Calibrating the Camera 7

3.1.2 Estimating the geometry of the environment 7

3.1.3 Estimating the position of objects in the world 7

Chapter four: The Solution 8

4.1 Analytical Work 8

4.1.1 Camera Calibration 8

4.1.2 Calculating the position of the objects from the camera: 9

4.1.3 Calculating the global position of detected objects 12

4.2 Architectural Level 13

4.2.1 Configuration of ROS Architecture 13

4.3 Implementation 14

4.3.1 Developing the Calibration Script 14

4.3.2 Creating the SLAM Module 14

4.3.3 Creating the Object Detection Module 15

4.3.4 Creating the Map Visualiser 16

Chapter five: Evaluation 17

5.1 Testing of PNP Solutions 17

5.2 Validation of SLAM System 18

5.2.1 Testing methodology 18

5.2.2 Results 18

5.2.3 Explanation of Results 19
5.4 Software Validation 19
5.4.1 Testing method 19
5.4.2 Test Results: 19
5.4.3 An interpretation of the results 20
Chapter six: Conclusion 21
6.1 Project Conclusion 21
6.2 Future Work 21
References 22

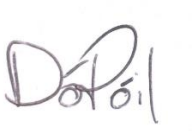
Abstract

This paper presents a novel method for generating sparsely populated point-cloud based interactive 3D maps, with the estimated location of discreet objects identified within the map, from a monocular video feed. This system is built upon state-of-the-art Simultaneous Localisation and Mapping (SLAM) systems and uses the Robot Operating System (ROS) as a communication channel by which individual modules can broadcast messages to each other. This structure increases performance by facilitating parallel computation of the SLAM and object detection algorithms before recombining into the final map. It also allows for a modular design of mapping systems that can be expanded upon to map multiple object types by utilising ROS topics to broadcast information to many modules at once. This software successfully handles both large scale outdoor and indoor video sequences and is robust to moderate motion blur caused my rapid rotations, making suitable for use in a multitude of environments from open roads to narrow corridors. For the benefit of the community the source code for this project has been made public.

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of my B.Sc. Single Honours, Computer Science and Software Engineering qualification, is *entirely* my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

I hereby acknowledge and accept that this thesis may be distributed to future final year students, as an example of the standard expected of final year projects.

Signed:  Date: 26/03/2019

Acknowledgements

I would like to take this opportunity to thank my project supervisor, Dr John McDonald, for his guidance throughout this project. John's expertise in the field of computer vision was instrumental in the completion of this project, and he never turned me away whenever I had any questions or problems.

I'd also like to thank Dr Charles Markham, who never shied away from a tricky mathematical problem or an at length discussion about coordinate systems, and whose infectious passion left a lasting impression on me. A special thanks also to Kealan Maas who graciously gave up his free time to help me set up scenes and record video sequences for testing.

Finally, I would like to thank my girlfriend, Kelly Geraghty, who put up with my endless ramblings and encouraged me every step of the way through this project and supported me through all of the long nights and early mornings.

List of Figures

Figure 1: Potential applications for this project.....1
Figure 2: Image Representing the relationship between camera and objects.....9
Figure 3: convex lens ray diagram.....10
Figure 4: diagram showing the viewing angle issue.....11
Figure 5: Testing the distance calculator.....12
Figure 6: Data Flow Diagram showing the communication between nodes. 13
Figure 7: Screenshot of the map Visualiser module.....16
Figure 8: Showing the loss in accuracy at greater distance.....18
Figure 9: Left: Paths of camera in sequence one. Right: Final positions relative to (0,0),.....19
Figure 10: Visualizer display for sequence 1.....20
Figure 11: Output of test on sequence 1, showing object location estimations in green.....20

Chapter one: Introduction

1.1 Topic addressed in this project

This project will attempt to combine existing technologies in the field of computer vision to develop a novel method of generating maps from a monocular video feed. In particular, this project aims to combine state of the art Simultaneous Localization and Mapping (SLAM) [1] techniques with Object detection algorithms - in this case Automated Licence Plate Recognition (ALPR) [2] to generate a map of the location of vehicles identified by their licence plates in 3-dimensional space.

1.2 Motivation

Figure 1: Potential applications for this project includes vehicular autonomy, mobile robotics platforms and traffic analysis

From traffic analysis and law enforcement to private car park companies and city planning, there are a multitude of situations which would benefit from having a real time, affordable way of identifying and locating vehicles in the real world. This technology would allow city planning officials to passively gather data about where and for how long car are parking, data which could be used as an indicator to locations where additional car parking might be required, or it might allow traffic wardens to instantaneously determine if cars have been parked in a parking space longer than the maximum allowed time.
The above applications could be used to reduce both monetary and temporal costs associated with city planning and traffic management and are just some of the applications that this technology may be used for. The goal of this project is not to create one specific program but rather a technology platform that many programs can be built upon.

1.3 Problem Statement

In recent years, modern visual SLAM techniques have become increasingly effective at reconstructing environments digitally and estimating the position of the robot in the environment to the extent that they are sufficiently accurate for use in real world and large scale applications. However, one major disadvantage of visual SLAM is that while it generates reconstructions of the geometry of an environment, the resultant map does not explicitly provide semantics i.e. where specific objects are within the map. This project aims to combine state of the art visual SLAM techniques with object detection algorithms to reconstruct a 3-dimensional map of an environment and identify where the location of specific objects in the map are.

1.4 Methodology

The approach taken is to first break down the problem into individual modules each describing and completing a specific task within the scope of the whole problem. Each of these modules were then approached as individual tasks, allowing them to be researched and solved individually before being combined where appropriate and finally coupled to all of the other modules to create the finalised package. Fig 6 in section 4.2 shows to finalised structure of all the modules and how they communicate with each other.

This approach had multiple advantages in this case. Firstly, upon commencement if this project I was largely unfamiliar with a lot of the techniques and technologies used and splitting the problem up into smaller parts allowed them to be researched and studied as individual concepts, rather than trying to comprehend multiple different complex concepts at once. Additionally, this methodology proved useful from a practical stand point as each module was entirely decoupled, it reduced the number of bugs that occurred during development, as each module only needed to perform a relatively simple task, and the outputs we combined after the tasks had been completed.

Throughout the project, prototypes were developed using models, simulations and dummy data to determine correct functionality before being integrated to the project proper. This meant experimental code never introduced bugs or incorrect functionality to the project. It also meant that there were well-defined programming tasks to complete throughout the entire development process.

1.5 Project Milestones

Detailed below are the milestones that were achieved throughout the project. These milestones were used as a way to gauge the success of various stages of this project and as an evaluator of the level of work done on the project.

Achievement 1 – Getting ORB_SLAM2 working

Coming into this project I had zero experience in the fields of computer vision or robotics and limited experience in UNIX/UNIX-like operating systems. These facts combined with the unmaintained nature of the ORB_SLAM2 library meant that it was actually quite the endeavour to get the software up and running.

I began by reading various articles online about visual-SLAM and the research papers on ORB_SLAM, and once I believed I had a basic understanding of the concepts, I began the installation process. This process ended up taking over a week as there were many out-dated dependencies, and support was limited to a small

number of active users on the GitHub repositories issues page. Eventually I managed to set the correct environmental variables, organise the file system, update all the dependencies and re-write many of the build scripts and successfully build the libraries, and run the sample code. More details of this process can be found in section 2.2

Achievement 2 – Reconfiguring ORB_SLAM to work with ROS and map the camera path in 3 Dimensions

The next major milestone in this project was to build the ORB_SLAM_ROS module and output the estimated pose of the camera to a ROS topic. This section of the project presented many roadblocks and issues of varying complexity, the first of which was figuring out how to get the camera pose from the ORB_SLAM module to publish. as well as figuring out how the SLAM module and the Map Display Module would deal with loop closure. The Solution to this was to modify an existing fork of ORB_SLAM by Abhineet Singh [3] which has been designed to output the keyframes and detected points to a ROS topic that could be interpreted by the RVIZ 3D Visualiser in ROS-Kinetic. The exact process and modifications made to this code are detailed in section 4.3.2.

Achievement 3 – Successfully compute the 3D camera coordinates of an object relative to the camera:

The next goal was to successfully return an estimated pose of the object relative to the camera. This was done using PnP solving algorithms to estimate the pose of the object using known world coordinates of the object (in this case the corners of a license plate or checkerboard) and the pixel coordinates of those points in the image. This step is discussed further in section 4.3.3.

Achievement 4 – Combine the Camera and Licence plate poses to get final 3D coordinates of the plate relative to the origin, plot in real-time/3d

The final milestone achieved of this project was to successfully plot the position of the detected objects in a map of the environment. This was done by combining the position of the camera and the position of the object relative to the camera, taking into account the orientation of the camera at the time of detection and rotation the translation vector of the object accordingly. Full details of this process can be found in section 4.1.3.

Chapter two: Technical Background

2.1 Technologies

Simultaneous Localisation and Mapping (SLAM) is an important field in robotics and navigation. It is the process by which a robot traverses an unknown environment and uses its sensor data and visual odometry to generate a map of the environment while simultaneously estimating its location within the map. SLAM technologies have been indispensable in the development of robotic systems that can operate autonomously. SLAM using 3D sensors such as LIDAR or depth sensing cameras is generally considered a solved problem [1]. However, there are fundamental limitations associated with these techniques, such as the high cost associated with LIDAR sensors or the limited range of depth sensing cameras that render them unsuitable for low-cost, outdoor applications. As a result, visual SLAM – the process of performing SLAM using only 2-Dimensional cameras as sensors – is still a widely researched area. Visual SLAM is also quite difficult as the absence of information in the 3rd dimension means that the 3D structure imaged by a 2 dimensional camera must reconstructed by matching features in multiple images of the scene taken from different perspectives.

ORB-SLAM [4] is a state of the art visual SLAM system that supports both monocular and stereo video input and is capable of creating sparsely populated point cloud based maps of challenging outdoor and large scale environments without the need for additional sensors. Though the resultant point cloud can be used to deduce the general geometry of the environment, it cannot identify individual entities in the scene. This is the problem that this project aims to solve by combining the ORB_SLAM2 system with entity detection techniques – in this instance Automatic Licence Plate Recognition (ALPR) techniques – to create a map of an environment with the position of specific objects in the environment estimated within the scene.

ALPR is a technology that uses optical character recognition on images to retrieve information about an unidentified vehicle. It works by detecting potential plate candidates in an image, projecting that candidate to a frontal view based upon the ideal licence plate size, running various filters and convolutions to clean up the image and then attempting to recognise the characters present in that projection [5]. ALPR can be used to find owner and registration information about a car by identifying the plate number and running it through a known database of registered cars. It is also frequently used by traffic enforcement in average speed camera systems. ALPR is considered a solved problem [2], open source libraries exist that can quickly and accurately recognize licence plates in images and licence plate scanners have been deployed by both public and private agencies worldwide to great effect.

OpenALPR is one such open source library that uses a Local Binary Pattern (LBP) algorithm to detect regions of an image that contain a licence plate [5]. This is particularly useful in the case of this project as evaluating the characters of the licence plate is not the goal, but rather to compute the position of the plate in 3D space relative to the camera. The first step in this is to locate the pixel region of the plate . This is achieved through the OpenALPR API which exposes a function which returns the image coordinates of the edges of any detected licence plates. This information along with the known area of standardised licence plate is what will be used to calculate the distance from the camera of the plate, using algorithms and techniques which will be discussed in later sections.

Robotics Operating System (ROS) is a collection of software frameworks for developing robotic systems. It provides services including package management, device control, implementations of commonly

used functionality and inter-process communication. ROS is designed to work as a loosely coupled system where each process is known as a node and each node performs a specific task. Nodes can communicate with each other by passing messages though channels called topics. Topics are described in the ROS Documentation as *"Named buses over which nodes exchange messages"* [6]. Topics have anonymous publish/subscribe semantics, which separates the production of information from its consumption. This means that unless a node broadcasts its own information, nodes communicating with it will not be aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic.

This feature of ROS is why it was chosen as a platform for this project as it allows the video feed to be broadcast to various nodes and processed in parallel rather than sequentially running each operation. This parallel process dramatically reduces total processing time and memory usage and facilitates the ability to process footage in real time. For full details on the configuration of the ROS workspace used in this project see section 4.2.1.

2.2 Technical material

The following section details tutorials and/or instructions followed in configuring the various software packages and builds needed in order to work on this project.

Installing ORB_SLAM - Installing ORB-SLAM2 using the instructions provided on the project GitHub page [3].

The most recent version of ORB-SLAM2 has a number of conflicts due to outdated dependencies, notably Boost, Eigen 3.3, Pangolin, ROS-Kinetic and OpenCV3 all had some sort of issues. As a result, Eigen 3.2.10 was installed, an older stable branch of pangolin was installed and built through git, and the Boost issues were solved by editing some of the build scripts to point to a separate Boost library. Additionally, OpenCV 3 was not used, rather OpenCV 2.4, which is installed alongside the ROS-Kinetic package, was used.

Due to the open source and experimental nature of many of the packages used, documentation and support was sparse, and in some cases, non-existent. As a result, installing and configuring all of the required packages added significant delays to this project, they did however allow for a much deeper understanding of why exactly each dependency was needed and how the software all worked.

Installing the ORB_SLAM2 Python Binding Library –

For this project the plan was for it to be written in its entirety in Python3.5.2, And as such the python wrapper library was installed and built to expose the ORB_SLAM package to Python.

The installation process for ORB_SLAM_Python was very simple in comparison to the installation of the C++ library. All that is needed is to install ORB_SLAM2 and all of its dependencies, git clone the repository to your home directory, apply a patch file to the ORB_SLAM2 package (which adds some additional installation instructions to the CMakeLists.txt files), Build ORB_SLAM2 and then building the wrapper library. This all worked without the need to modify anything other than some environment variables and the sample python code successfully compiled and ran.

Unfortunately, this process ended up being redundant as ROS was not supported in the Python binding library, as such the SLAM module of this project was written using the Original C++ library. The details of this module will be discussed in a further section of this report.

Installing OpenALPR –

The OpenALPR package is available for download either via the ubuntu package manager or via git and the installation process is detailed clearly in the read-me. As this project required Real-Time Processing capabilities, there were a number of extra steps to the installation process in order to scale the OpenALPR Agent to be capable of handling video footage rather than still images.

The Process was still relatively simple, simply install any listed dependencies using *APT*, then clone the repository to your local drive and build the package. Once the package is built it can be tested to ensure that it is working in the terminal by using the command '*alpr'* followed by the path to an image.

To compile the python binding package, navigate to the folder where the package is cloned and go to *'{OpenALPR installation folder}/src/bindings/python/'* and run the setup.py script. Once this has finished running OpenALPR can be imported into any python script using the following command: *'from openalpr import Alpr'.*

Using PyQTGraph to visualise the data –

In order to visualise the output data of the software, a visualisation module will be written in python using the PyQTGraph package, data visualisation and user interface framework built on the QT graphics framework and NumPy. It provides the ability to plot data in real time and can be combined with OpenGL's python binding library to plot data in three dimensions. This means that a simple python script can be written that subscribes to the output of the SLAM and ALPR modules and displays the received data in an interactive map in real time. This data can also be stored in a simple text file and recalled to the visualiser module at any time.

Chapter three: The Problem

3.1 Challenges

This project contains a number of problems that need to be individually solved before being combined into the final product. This section details the individual challenges faced in this project

3.1.1 Calibrating the Camera

In order to perform calculations on an image accurately, the first step is to undistort the image so as to give a true representation of the scene being captured. This is done by calibrating the camera used in order to obtain the Extrinsic and Intrinsic Parameter specific to that camera. This process is detailed in section 4.1.1

3.1.2 Estimating the geometry of the environment

To generate a map of an environment it is necessary to first know the geometry of the environment. This can be done a number of ways including manually measuring the environment, however for the purposes of this project this project is being handled by ORB_SLAM [7].

3.1.3 Estimating the position of objects in the world

Possibly the most complex problem in this project is to be able to detect objects within the environment and estimate their pose. This problem can be further broken down into three separate parts:

- Estimating the position of the camera in the world
- Estimating the pose of the object relative to the camera
- Combining these poses to get the position of the object in the world

The first problem is solved by Using ORB_SLAM, in the array of points output by ORB_SLAM, the first entry is always the position of the camera.

The second is slightly more complicated and will depend on the specifics of the object being detected (does is have a symmetrical shape, are its dimensions known etc.). Details on the specifics of how this was done in this project can be found in section 4.1.2

The final problem requires some consideration. In order to combine the poses to get the global pose, both poses will need to be in the same coordinate system, and as the object position relative to the camera will always be along the same axis, it will be necessary to rotate that vector to align with the orientation of the camera at the time of detection. For full details see section 4.1.3

Chapter four: The Solution

4.1 Analytical Work

4.1.1 Camera Calibration

Modern low-cost cameras have a tendency to introduce distortion into images due to the lenses used and manufacturing tolerances. The two main types of distortion are radial distortion and tangential distortion. Radial distortion is caused by the shape of the lens and causes straight lines to appear curved. This distortion is also more pronounced the further from the centre of the image the line appears. Radial distortion is corrected using the following set of equations:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Tangential distortion occurs when the lens is not perfectly aligned with the image sensor, causing one section of the image to appear closer than the other, and can be corrected using the following equations:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

Utilising these equations to remove the effects of these types of distortion requires calibration of 5 parameters, called the distortion coefficients, given in the form :

$$Distortion\ coefficients = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

Additionally, we require intrinsic and extrinsic camera parameters.

Intrinsic parameters are specific to a camera and include the focal length (f_x, f_y), principal point(c_x, c_y) and image sensor format of the camera. These parameters are typically collected in the camera matrix as they can be expressed as a 3x3 matrix of the form

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Extrinsic parameters parameterise transformation between the 3D camera coordinates and 3D world coordinates. They are denoted by *R* and *T* where *R* is a 3x3 Rotational Matrix and a 3x1 Translation Matrix.

These values are required to accurately estimate the pose of the camera in the environment and in visual SLAM applications. If the image is distorted, then the readings of the points in the video frames will be inaccurate. The geometry of the scene will be incorrectly calculated as the calculations are not done with respect to a distorted image but rather the platonic ideal of an image, a true 2D representation of a 3D scene. As a result, the images need to be undistorted using the aforementioned parameters. These parameters are passed into the ORB_SLAM module and the undistorting process is done within ORB_SLAM, but first we need to acquire those parameters. To do this a python script was written which takes in a number of reference images – taken using the same camera that will be used for the project – and outputs the necessary parameters.

The OpenCV2 Python Library exposes functions which perform calibration operations and return the necessary parameters. In order to get these parameters, we need provide some sample images taken with the camera which contain some well-defined pattern (like a chess-board) to the program. Generally, ten or more

images are required to get accurate measurements. We then pass these images to the cv2.getChessboardCorners() function to get the pixel positions of the corners of the chess board, and we can pass the resultant image points and an array of object points – a set of 3D coordinates which denote the position of each square relative to the camera – to the OpenCV cv2.calibrateCamera() function to get the distortion coefficients, rotation vectors and translation vectors. In the script written for this project those values are then saved to a YAML file for use by the other modules. An explanation of how the script was written can be found in section 4.3.1

There were a number of issues that occurred throughout this process that can mostly be attributed to the camera used. Upon initial calibration the reference images were undistorted using the cv2.undistort() function but it there was still obvious barrel distortion at the edges. It was determined that there were 3 possible causes of this issue:

1) The lens used was cheap and plastic, and as such may not have had a consistent radius
2) The camera used had a particularly wide viewing angle (170+ degrees), making the calibration process less effective at the edges. The cv2.calibrateFisheye() function was also tested but yielded similar issues.
3) It is possible that some sort of post processing was done in-camera before the images were offloaded to the computer for processing.

Unfortunately, no definitive answer was found within a suitable time-frame, so the solution was to use a different camera, as it was deemed too expensive in terms of person-hours to solve the problem on the software end.

4.1.2 Calculating the position of the objects from the camera:

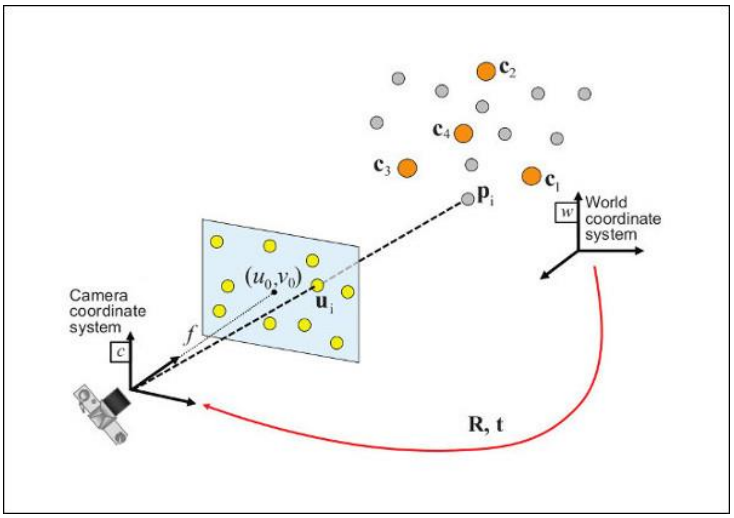


Figure 2: Image Representing the relationship between camera and objects

In order to back-project the image of the object into the 3D world space and acquire its world coordinates, we first need to find the perpendicular distance between the Image plane and the plane of the object. This can be done a number of ways depending on the information available. A number of options were explored to achieve this, detailed below:

Method 1: Calculating the Distance using the focal length and known real width of the object:

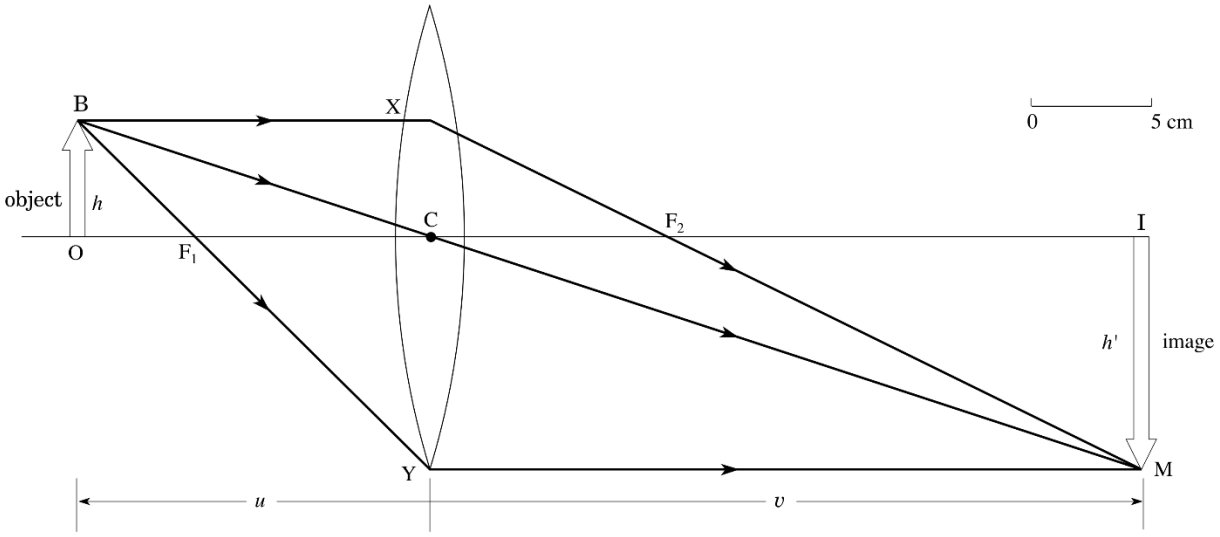


Figure 3: convex lens ray diagram

The first method I tried was to calculate use the fact that the ratio between the image width and real width of an object is the same as the ratio between the focal length and the object distance, i.e.:

$$\frac{image\ width}{object\ width} = \frac{focal\ distance}{object\ distance}$$

Which can be re-arranged and written in terms of the pixel size of the sensor to express the distance of the object plane to the camera plane as:

$$Distance(mm) = \frac{focal\ length\ in\ mm \times object\ width\ in\ mm \times camera\ frame\ width\ in\ pixels}{image\ width\ in\ pixels \times sensor\ width\ in\ mm}$$

Using the known dimensions of licence plates in Ireland (as defined in the Vehicle Registration and Taxation Regulations 1992 (Statutory Instrument (S.I.) 318/1992 as amended, and Statutory Instrument (S.I.) 542/2012) [8]), it is theoretically possible to compute the distance between the image and the camera, and in practice it works in some situations, however it had two major drawbacks. The first being that you need to have the specifications of the camera sensor size and resolution and the focal length of the camera and given the affordable nature of the cameras used in testing, that information was not available in either the manuals or the online documentation. The patent applications were also not publicly available.

The second major, and the main reason this option was eliminated as a viable process, is that it was affected by the viewing angle of the plate. For example, if the plate was detected at a 60 degree angle to the camera, it be perceived as having half the image width as a plate at the same distance but viewed straight on, causing the system to calculate its distance as twice the actual distance.

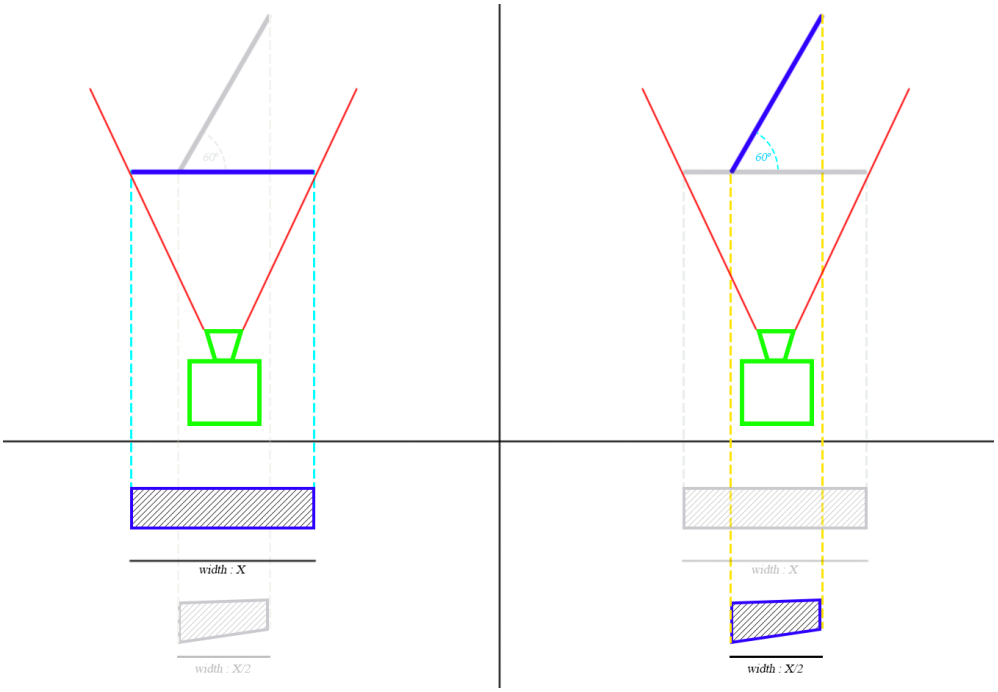


Figure 4: diagram showing the viewing angle issue

Method 2: Calculating Distance using the pixel area of the plate in the image:

The second method explored for calculating the distance of the plate from the image was to use the known real area and known image area of an object. This option was presented in a summer internship paper written by Saleem Ahmad for the Indian Institute of technology [9] and was suggested as a method for robots to calculate their distance from a charging point in their field of vision.

The paper posits that since the dimensions of the licence plate is invariant, the non-linear relationship between the pixel area of the plate in the image and the distance between the camera and the plate can be calculated by taking reference images at known distances from the camera and plotting the resulting plate pixel area against the distance. Then a sum of squared difference curve fitting technique is applied to calculate the relationship.

The resultant relationship is approximately the inverse of the square root of the area, however this varies slightly from camera to camera due to a number of variables including focal length, field of view and the sensor size. It also suffers from the same problem as the first method, where plates viewed at sharp angles would be measured as having substantially smaller areas than plates viewed at the same distance but straight on, causing inaccuracies. As a result, this method was not used in the final version of this project.

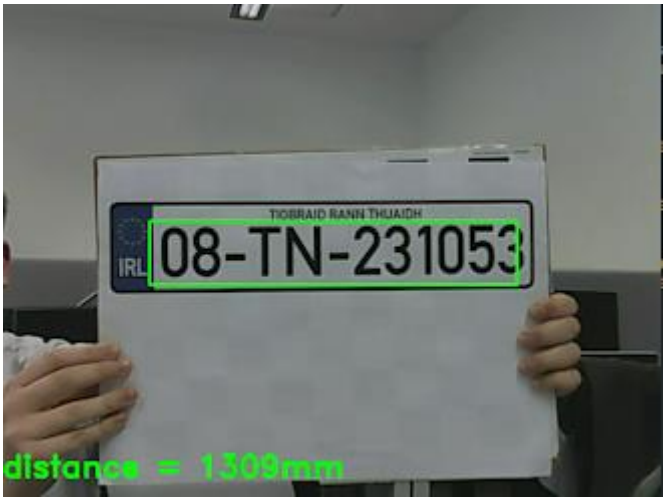


Figure 5: Testing the distance calculator

Method 3: Pose Estimation Using PNP Pose Estimation Methods:

The Third method tested was to use an existing solution to the Perspective-N-Point (PnP) Problem to estimate the 3-Dimensional pose of the plate. The PnP problem is as follows:

"Given n (n ≥ 3) 3D reference points in the object framework and their corresponding 2D projections, to determine the orientation and position of a fully calibrated perspective camera"[10]

This problem has been solved using multiple methods. One common solution for the problem where *n=3* is known as P3P, and there are many solutions that solve the general case where n ≥ *3.* OpenCV contains a *solvePnP()* function that allows an argument that enables the use of different well known methods, and additionally a *solvePnPRansac*() *function* that uses Random Sample Consensus (RANSAC) in conjunction with the chosen method in order to make the final solution more robust to the presence of outliers [11]. The available methods were all tested both with and without the addition of RANSAC to determine the most effective method for this particular application. Details of this testing process can be found in section 5.1.

This method had the distinct advantage over the other methods mentioned previously of being unaffected by the viewing angle of the license plate as it computes the pose of the plate relative to the camera, and as such takes the viewing angle into account. It also returns a set of 3D camera coordinates rather than just a Z distance between the camera and object planes. This saves on performing additional calculations later.

4.1.3 Calculating the global position of detected objects

Once the position the object relative to the camera has been obtained, the next step is to translate that to the global coordinate system output by ORB_SLAM.

In order to this, the pose of the object is scaled down to be roughly in line with the scale of the map, then the translation vector of the object is rotated by the quaternion orientation of the camera at the time of detection (this is done because the camera is assumed to be facing down the Z-axis in the object detection module, and must be corrected when the values are combined).

The next step is to translate the coordinate system of the object vector to that of the SLAM module. This is necessary because in the SLAM module the vertical axis is Z whereas in the object detection module the vertical axis is Y. to this the Y and Z components of the camera pose are swapped.

The rotation is then done using the *pyQuaternion.rotate()* class which when given the point vector P and rotation R performs the following operation:

$$P = [0, p1, p2, p2], R = [w, x, y, z],\ R' = [w, -x, -y, -z], P' = RPR',\ P' = H(H(R, P), R')$$

Where *H( )* represents a Hamiltonian product.

The function then returns the rotated vector, *P'*, and that the dot product of P' and the position of the camera relative to 0,0. The resultant 3x1 vector represents the position of the object relative to 0,0.

4.2 Architectural Level

4.2.1 Configuration of ROS Architecture

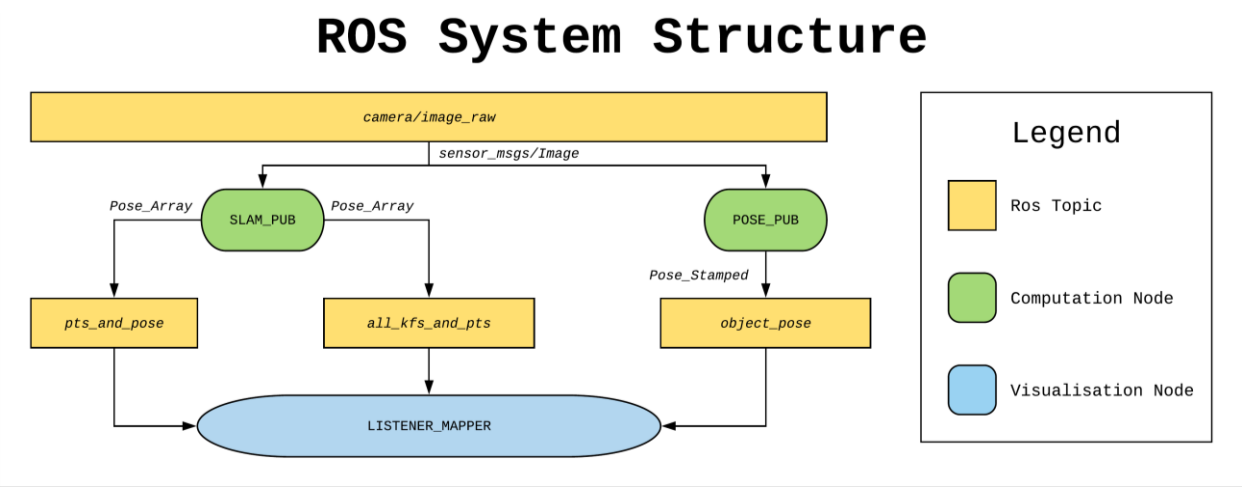


Figure 6: Data Flow Diagram showing the communication between nodes.

The above figure shows the flow of information through the software, as well as the individual ROS Nodes. As previously mentioned in section 2.1, the image topic, labelled '*IMAGE_RAW'* broadcasts the video feed frame by frame to subscribed nodes *'SLAM_PUB'* and *'POSE_PUB',* which process the current frame and calculate the camera pose relative to the origin, and the pose of the plate relative to the camera respectively. These two nodes run in parallel on separate threads in order to reduce processing time by spreading the workload.

SLAM_PUB outputs data to two topics: *'pts_and_pose'*, which outputs the pose of the camera for the current frame being processed, and *'all_kf_and_pts'* which outputs at a user-defined keyframe interval, or when a loop closure is detected, and broadcasts the loop corrected array of poses up to that point.

POSE_PUB processes the current frame and attempts to detect license plates, when a licence plate is detected, it estimates the positions of that plate relative to the camera at that point in time, and outputs the pose of the plate to the topic '*plate_pose_stamped'* along with the frame number that the plate was detected.

A third Subscriber node, **LISTENER_MAPPER**, subscribes to all three topics, *'pts_and_pose'*, *'all_kf_and_pts'* and *'plate_pose_stamped',* and plots the data to an interactive 3D graph. This node has a call-back for each topic. The first call-back is for 'pts_and_pose' and is called on each frame, updating a list with the current camera pose. The second call-back, executed when a new message is posted to '*all_kf_and_pts',* clears the current list of poses and replaces it with the scale-corrected list. And the third call-back is triggers whenever a new licence plate is detected and takes the estimated pose of the plate and adds it to the pose of the current camera frame, yielding an estimated pose for the plate relative to the origin. This is then added to a separate list. These lists are then accessed by the MapWindow object and the data is graphed in real time.

4.3 Implementation

4.3.1 Developing the Calibration Script

As nearly every aspect of this project depends on having the camera matrix, intrinsic parameters and distortion coefficients, the first requirement of the project was to write a script that takes some media input, calibrates the camera and saves the parameters to a file. The save feature allows the camera to be calibrated once and then used indefinitely, rather than needing to calibrate every time.

This script is based on the OpenCV python calibration tutorial [12], however with several features added specific to the needs of this project. It was written in python using the OpenCV calibrateCamera() function, and also the fileWriter function to save the parameters as a YAML file. YAML was chosen as is allows the variables to be directly read into another file rather than needing some sort of parsing function as would be needed with, for example, a set of comma-separated values. The script was written to be as user friendly as possible and allows the user to input either a set of preselected images, or a video containing candidate frames. If the latter is selected, then the script will automatically scrub through the video selecting the candidate frames, and then will take a selection of *n* images for calibration, where n is a flag defined by the user (note: this adds a considerable amount of additional processing). Additionally, it also provides the ability to calibrate fisheye lenses using the cv2.calibrateFisheye() function. The function used to calibrate the images is set by the user using a flag.

This script utilises the chessboard method of calibration, that is to say that several images of a chessboard of known dimensions in several different orientations relative to the camera are passed into the program, whereby the corners of each square are detected. This allows the calibration function to detect radial distortion (where the points of the corners do not fall along a line but rather a curve) and tangential distortion (where one side of the chessboard does not the match the dimensions of the other). As the calibration function not only knows if there is radial distortion present or not, but also the degree to which it is present (how sharply the lines curve), it is able to calculate a set of vectors which correct the distortion.

4.3.2 Creating the SLAM Module

As mentioned in section 2.2 initially the SLAM module was to be written in python 3.6, however the python binding library was written prior to ROS support being added to the original repository. As such, it is not

possible to integrate ORB_SLAM and ROS in python at this time. As a result, this module was written in C++ and is based on some of the original sample code from the ORB_SLAM2 repository.

This module functions by subscribing to the ROS topic broadcasting the video feed as a sequence of frames and passing that sequence frame by frame into ORB_SLAM. ORB_SLAM then returns the computed camera pose using the *GetPoseInverse()* function. ORB_SLAM returns an array of points where the first element of the array is the camera position and every subsequent point is a point of interest in that frame. These are all published however they are not used by the Map Viewer Module by default. A new PoseArray is published for every frame and these are published to *'pub_pts_and_pose'*.

In addition to this, whenever a loop closure is detected, the array of points up to that point is recalculated to match the loop and the corrected array is published to *'all_kf_and_pts'*. This can also be set to update at a defined interval by using a command line argument.

4.3.3 Creating the Object Detection Module

For the purposes of this project two types of object will be identified, checkerboard patterns, as they are easy to identify exact dimensions of and can be used as a control to measure the effectiveness of various aspects of the project, and license plates, as an applied example of how the technologies can be combined. The Object detection node was written in python and utilises OpenCV and OpenALPR.

This script defines an objectDetector Class which subscribes to the input topic and converts the broadcast images from ROS sensor_msgs.msg.Image type to OpenCV compatible images using the CV_Bridge library. It has the following functions:

- Constructors for the ROS nodes, the ALPR node and the CV_Bridge node.
- A Converter function which converts Ros Images to OpenCV images. This is triggered when a new image is published to '*camera/image_raw'*
- An experimental function which calculates the distance between the camera plane and the target plane by comparing the pixel area to the known real area of the object (this function was only used in testing).
- Another function which detects checkerboards and estimates their poses using the PNP problem solutions provided by OpenCV.
- A final pose estimation function that works like the above function but uses OpenALPR to detect the corners of the license plate.

This module was written in such a way that it can be easily adapted to identify many different objects, providing that there exists a solution to estimate the pose of the object. In this case the script includes methods for estimating the pose of either a license plate or a checkerboard pattern. In the case of the checkerboard pattern that was used for testing the PNP algorithms, the built-in OpenCV checkerboard detector has quite bad performance when searching for the checkerboard. In order to try to alleviate this, the flag *'CALIB_CB_FAST_CHECK'* was passed to the *findCheckerboardCorners()* function. This flag runs a fast check that looks for chessboard corners, and shortcuts the call if none are found [12]. This improved performance dramatically, allowing the module to keep up with 30fps video, and although this was not tested, it appears to have improved the overall robustness of the pose. Testing details of the algorithms can be found in section 5.1.

4.3.4 Creating the Map Visualiser

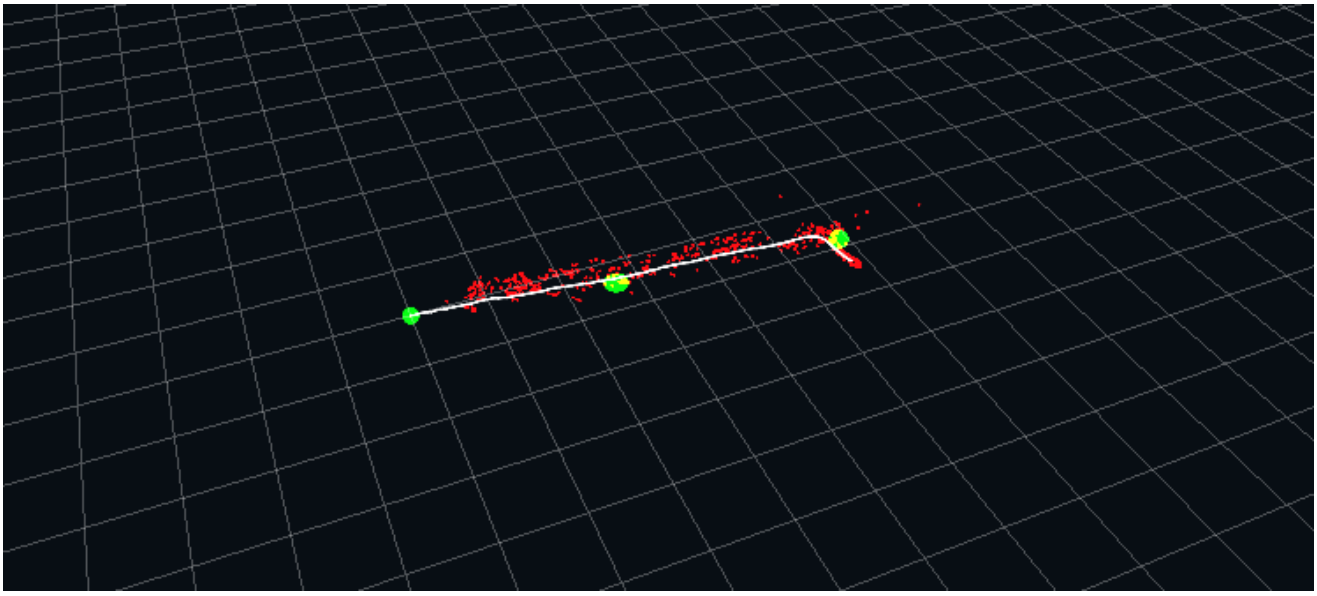


Figure 7: Screenshot of the map Visualiser module

In order to evaluate the output of the mapping and object detection modules, a data visualisation module is required. This module was written in python using the PYQTGraph library and its OpenGL integration to draw the path of the camera and points where plates are detected.

The script first defines a Map class which constructs a QtGUI application window and initialises it with a blank 3D environment. It also defines the following functions:

- Constructor functions to define the ROS nodes as well as to initialise an instance of the Map class.
- writer functions for both the camera pose and the plate positions, which allows them to be saved to text files so that they can be recalled at a later date. It also allows the messages to be converted from ROS PoseArray objects into X,Y,Z floating point numbers which are natively supported in NumPy, avoiding the need for any parsing functions. These are executed as callbacks whenever a message is published to any of the subscribed ROS topics and are spun up on parallel threads to increase performance.
- A Map update function which reads from the text files and updates the array of poses to include new values. The script uses NumPy to store the information
- An animator function which runs the map updater at a predefined interval, this facilitates real-time data visualisation.

Chapter five: Evaluation

5.1 Testing of PNP Solutions

In OpenCV there are a number of built in algorithms for solving the PNP problem. As the aim of this project is not to propose a new approach to solving the PNP problem, it was determined that the available algorithms should be tested instead in order to evaluate their effectiveness.

In order to determine which method worked best for this project, each algorithm was tested multiple times on 3 separate video sequences, as well as a live video feed. Each algorithm was evaluated on a number of aspects: Computation time, Accuracy and Consistency. The results of this experiment are summarized below

.

Initially, these tests were performed on license plates, but OpenALPR struggled to consistently determine the corners with any degree of accuracy, so for testing purposes these tests were performed on checkerboard patterns, where only the outer corners of the checkerboard were used to estimate the pose, which served as an analogue for licence plates.

The first method tested is known as the PnP iterative method and is based on Direct Linear Transform (DLT) with Levenberg-Marquardt Optimization [11]. It attempts to estimate the relative rotation and translation of an object from an initial pose to a new pose by minimizing the reprojection error [13].

The results of testing this algorithm determined that while it performed well both in video playback and live video scenario, being able to estimate the pose in real time, there were insufficient known points to correctly estimate the pose, and the pose estimation varied frame by frame. This problem can be slightly alleviated by reducing the resolution of the video feed, however for the application of this project it was deemed to be unsuitable as the pose could not be relied upon.

The second method tested was the P3P Solution Presented by Gao et. al. in their 2003 IEEE Transactions on Pattern Analysis and Machine Intelligence paper [14]. The Solution presents an algebraic approach which uses the Wu-Ritt zero decomposition algorithm [15] to give a complete triangular decomposition for the P3P equation system [14]. In practice this method, with only 3 point correspondences, produces multiple solution, so typically a 4^{th} reference point is used to remove ambiguity. This method is suitable for this project as we have 4 point correspondences, the corners of the licence plate.

The test results show that this method appears to produce quite a robust pose when using a live camera feed to test. However, when using a recorded video, the pose becomes less accurate at distance. This may be caused by a number of factors, including video resolution, the OpenCV object detection algorithms, or the ambient light of the environment. Results were unable to determine the cause of these issues. This algorithm performed very well in terms of computation time, able to keep up with 30fps video.

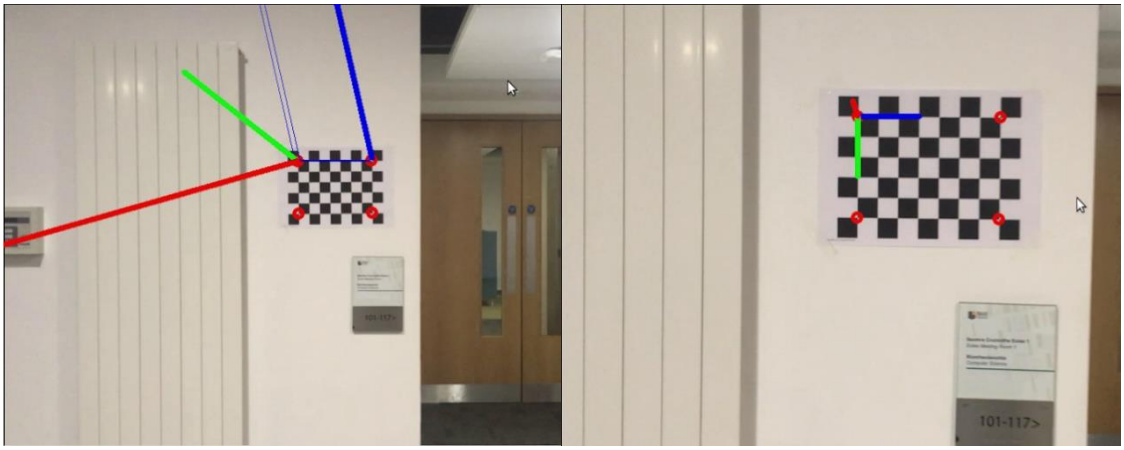


Figure 8: Showing the loss in accuracy at greater distance

The final method tested was the Efficient Perspective-N-Point method as proposed By Lepetit, Moreno-Noguer and Fua in their paper "EPnP: An Accurate O(n) Solution to the PnP problem" [16]. The foundation of this method is the notion that each of the n points can be expressed as a weighted sum of four control points. Hence, the coordinates of the 4 control points become the unknowns of the problem. The final pose of the camera is then solved for from the 4 control points.

This method performed very similarly to the P3P Solution tested above, being well able to detect objects at reasonably acute angles and similar distances - approximately 3-5 meters with a frontal view, < 2 meters when viewed at an acute angle. As with the other methods, robustness of the pose improved the closer the object was to the camera.

These results show that either the EPNP or P3P methods work for the purposes of this project and EPNP was chosen in this instance.

5.2 Validation of SLAM System

5.2.1 Testing methodology

In order to determine the consistency of the SLAM module, the same sequences were measured repeatedly under the same parameters in order to see how often it lost track, what caused it to lose track, and when it managed to not lose track, how consistently it was able to track the camera path.

5.2.2 Results

Out of 20 runs on the test sequence, the following information was obtained:

Sequence 1 – Indoor environment, Sharp corners:

Did not lose track	Lost Track on Straight	Lost Track on Corner #1	Lost Track on Corner #2
19	0	1	0

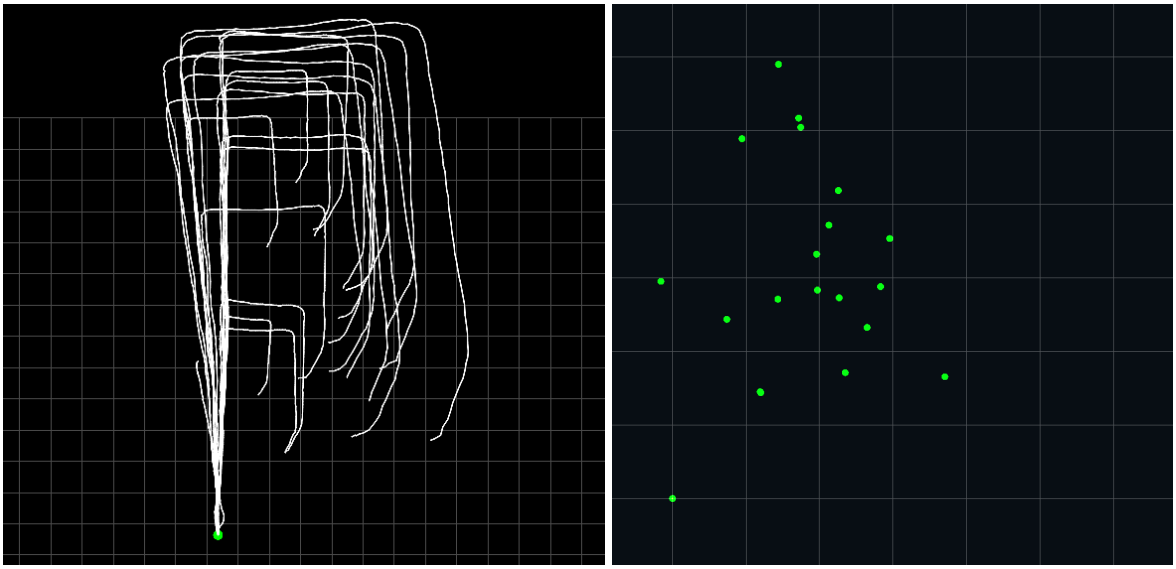


Figure 9: Left: Paths of camera in sequence one. Right: Final positions relative to (0,0),

Min Distance	Max Value	Mean Value	Variance	Standard Deviation
1.817662	6.516745	3.851221	1.602639	1.265954

5.2.3 Explanation of Results

From the above results we can see that while the general shape of the corridor is being detected, and rotations are reasonably stable, having a 95% success rate when turning corners, scale drift is still quite a large issue. This is somewhat mitigated by loop closure algorithms however this requires that the camera pass through the same point twice, so it is not always possible to perform loop closure. Additionally, while the ORB_SLAM algorithms are able to maintain a track fairly reliably, the scale factor varies wildly between runs, making it very difficult to for example stitch multiple maps together. The table above shows that if we assume the mean value to be unit scale, then the scale variance is approximately 1.6 and the standard deviation is ~1.25. This is not a problem for applications where only a rough shape is needed, this is not a problem, however without being able to ensure loop closure occurs it is not suitable for applications that require a great deal of precision.

5.4 Software Validation

5.4.1 Testing method

In order to determine whether or not the software correctly mapped the position of the camera and the objects detected, a test sequence was run through the software and the results were overlaid over a floor plan of the environment, with the actual location of the objects marked on the plan. As ground truth measurements are not available for these sequences, the program was tested qualitatively rather than quantitively.

5.4.2 Test Results:

Fig. 10 below shows a screenshot of the visualiser following computer, and fi. 11 shows the output of the test on sequence 1, which shows the path of the camera as tracked by the SLAM module in Red with orientation indicated by the arrows (Corrected for loop closure after computation), the real position of the objects marked with an orange X, and the estimated location of the objects marked in green as output by the software.

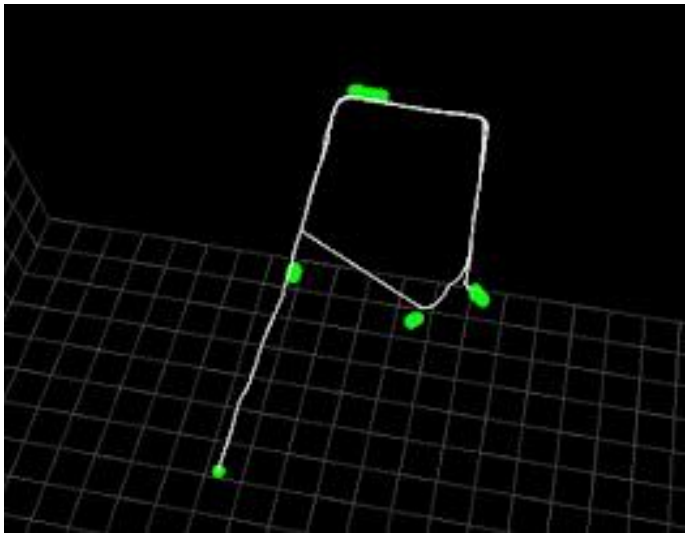


Figure 10: Visualizer display for sequence 1

Figure 11: Output of test on sequence 1, showing object location estimations in green.

5.4.3 An interpretation of the results

As fig 11 shows, the estimated location of the objects is shown as an area rather than a point, this is because the scale factors are not perfectly aligned and as such the position deviates slightly between frames. It is clear from the above figure that the software is correctly detecting the approximate location of the plates within a margin of error that can be attributed to the scale factor, so from that respect the software functions correctly and as expected.

Chapter six: Conclusion

6.1 Project Conclusion

The ultimate goal of this project was to be able to generate a sparsely populated 3-dimensional map of an environment with an estimated location of discreet objects identified within the map. To that effect, this project succeeded in its goal. This was achieved in this project by combining state-of-the-art SLAM with systems with object detection algorithms by having them communicate with each other through ROS topics. The system built is modular and allows object detection algorithms to be build into new modules and combined with the software without having to rewrite any existing code. Theoretically, it would be possible to generate a map which identifies any number of objects, not just one particular type of object.

Initially, the goal was to be able to map car parks and identify cars within the carpark, however due to the unstable output from OpenALPR Rendering the pose estimation unreliable, this was not possible within the given timeframe. Through testing with checkerboards of a similar size to license plates, it has been shown that given an object that can be detected with sufficient stability, it is possible to estimate its position within the environment. OpenALPR provides functions to train the ALPR System on a dataset to improve its accuracy, and there exists other ALPR systems that may offer better performance. These may result in a usable pose. However, this is a potential future improvement that may be added later and is not currently implemented.

6.2 Future Work

This project only demonstrates a proof of concept, and there are a number of issues that may be addressed to further refine this technology. These include:

- Addressing the issue of scale drift: Presently, while the loop closure feature of ORB_SLAM is functional and incorporated into the map, the positions of the objects are not corrected when a loop is closed. This could be built into the visualiser module.
- Aligning the scales both modules: Monocular SLAM initializes with an arbitrary scale every time it is run, whereas the object detection uses a fixed scale. Currently the scale relationship is only approximated, and a better solution for this may be implemented later.
- Usability: In terms of the usability, it is a very involved process to get this project working and there is scope to simplify processes and consolidate libraries to provide a more useful technology that can be used in a variety of applications. On specific feature that could be added is the ability to Save maps as SVG images so they can be overlaid over topographical maps.
- Improving Accuracy: There exists a number of ways to improve accuracy of object detection algorithms, including training recurrent neural networks on known datasets, these may dramatically improve the estimation accuracy.

References

[1] C. Cadena *et al.*, “Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age,” *IEEE Trans. Robot.*, vol. 32, no. 6, pp. 1309–1332, 2016.

[2] S. Du, M. Ibrahim, M. Shehata, S. Member, and W. Badawy, “Automatic License Plate Recognition (ALPR): A State-of-the-Art Review,” *IEEE Trans. CIRCUITS Syst. VIDEO Technol.*, vol. 23, no. 2, p. 311, 2013.

[3] A. Singh, “ORB_SLAM2 Fork,” 2017. [Online]. Available: https://github.com/abhineet123/ORB_SLAM2. [Accessed: 13-Oct-2018].

[4] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, “ORB-SLAM: a Versatile and Accurate Monocular SLAM System,” 2015.

[5] M. Hill, “OpenALPR Design,” 2014. .

[6] “Documentation - ROS Wiki.” [Online]. Available: http://wiki.ros.org/Documentation. [Accessed: 26-Mar-2019].

[7] R. Mur-Artal and J. D. Tardós, “ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras,” 2017.

[8] *S.I. No. 318/1992 - Vehicle Registration and Taxation Regulations, 1992.* Irish Stature E-book, 1992.

[9] R. P. K. Saleem Ahmed, “Vision Based Navigation and Odometry for Autonomous Mobile Robots,” 2015.

[10] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision, Second Edition*. 2000.

[11] Y. Zheng, Y. Kuang, S. Sugimoto, K. Åström, and M. Okutomi, “Revisiting the PnP Problem: A Fast, General and Optimal Solution,” 2013.

[12] “Camera Calibration — OpenCV-Python Tutorials 1 documentation.” [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html. [Accessed: 16-Oct-2018].

[13] D. Grest, T. Petersen, and V. Krüger, “A Comparison of Iterative 2D-3D Pose Estimation Methods for Real-Time Applications,” 2009, pp. 706–715.

[14] X.-S. Gao, X.-R. Hou, J. Tang, and H.-F. Cheng, “Complete Solution Classification for the Perspective-Three-Point Problem.”

[15] W. Wen-Tsun, “Basic principles of mechanical theorem proving in elementary geometries,” *J. Autom. Reason.*, vol. 2, no. 3, pp. 221–252, Sep. 1986.

[16] V. Lepetit, F. Moreno-Noguer, P. Fua, V. Lepetit, F. Moreno-Noguer, and P. Fua, “EPnP: An Accurate O(n) Solution to the PnP Problem.”

Appendix:

The code for this project can be found in the support material or can be accessed via git by cloning the repository: https://github.com/DomhnallP/SLAM_MAP.git . There exists a branch called Thesis-submission which contains the code exactly as it existed at the time of submission of this paper.