Håkon Elfving

Vehicle Detection and Landing as an Addition to Package Delivery Drones

Hovedoppgave i MTTK
Veileder: Ole Morten Aamo
Juni 2019

Hovedoppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for ingeniørvitenskap
Institutt for teknisk kybernetikk

Håkon Elfving

Vehicle Detection and Landing as an Addition to Package Delivery Drones

Hovedoppgave i MTTK
Veileder: Ole Morten Aamo
Juni 2019

Norges teknisk-naturvitenskapelige universitet
Fakultet for ingeniørvitenskap
Institutt for teknisk kybernetikk

Jeg vil gjerne takke veillederen min Ole Morten Aamo, for veileding og hjelp.

Abstract

The recent developments in autonomous package delivery drones show promising results, but potentially face limitations when it comes to their maximum distance. In this thesis, the possibility of using public communication to aid in this investigated. This investigation is realized using a simplified example in the simulation environment Gazebo. It is tested whether a drone can be made to autonomously detect and land on a bus equipped with a landing platform on its roof, using computer vision.

Abstrakt

De nylige utviklingene i autonome pakkeleveransedroner gir lovende resultater, men har potensiellt begrensninger når det gjelder maksimal avstand de kan dra. I denne oppgaven skal jeg ta for meg muligheten til å bruke offentlig kommunikasjon for hjelpe med denne begresningen. Denne forskningen er gjennomført i formen av et forenklet eksempel i simuleringsprogrammet Gazebo. Det er testet om en drone kan detektere og lande på en buss utstyrt med en landingsplatform, ved hjelp av datasyn.

Abbreviations

Pose	=	The combined vector of position and orientation
SLAM	=	Simultaneous Localization and Mapping
VSLAM	=	Visual Simultaneous Localization and Mapping
KF	=	Keyframe
FOV	=	Field of view, angle of visibility spanned by a camera
LSD	=	Large-Scale Direct Monocular SLAM
PTAM	=	Parrallel TRacking and Mapping
SVO	=	Semi-direct visual odometry
DSO	=	Direct Sparse Odometry
DL	=	Deep Learning
ML	=	Machine Learning
AR	=	Augmented Reality
SD	=	Standard Definition: 360p, 4:3 aspect ratio
HD	=	High Definition: 1080p, 16:9 aspect ratio
FPS	=	frames per second (framerate)
POV	=	Point of view
GPS	=	Global Positioning System
GNSS	=	Global Navigation Sattellite System
UGV	=	Unmanned Ground Vehicle
UAV	=	Unmanned Aerial Vehicle
AUV	=	Autonomous Underwater Vehicle

Chapter 1

Introduction

There has been great strides in the the creation of autonomous drones. As the cost of both drones and cameras has greatly reduced, the resulting potential for new applications are vast. Drones made to be autonomous (capable of operating without direct human control) is one of these. Without direct human control, the use of them for autonomous package delivery is a popular modern use.

1.1 Recent development of autonomous drones

Simran Brar (2015) describes the recent development of drones for package deliveries, and the firms that take this approach. Amazon, as the largest online retailer (Simran Brar (2015)), are among the most notable in this subject with its recent "Amazon prime air" which was launched in (cite år). "Amazon Prime Air" has been the most public with its research into this technology. As a majority of the packages ordered through Amazon are fairly small, a large majority of them can be delivered using drones **?**.

Additionally Google has been working on its package delivery project "Project Wing" since 2011. FedEx and DHL among other firms have considered or worked on package delivery drones as well (Simran Brar (2015) and Vincent and Gartenberg (2019)). "Amazon Prime Air" however seem to have come the farthest, claiming on a conference this June 5th **?** that it will launch this new delivery service "within the coming months" (Vincent and Gartenberg (2019)).

To summarize, the field of autonomous drones is a state of the art field with possibilities several businesses are trying to take advantage of. Amazon state that the drone package service that will be releaseed within the year (2019) will be using visual techniques to stay robust and avoid obstacles, and will have a range of 25 km (Wilke (2019)).

1.2 Range limitation and proposal to mend it

This range limitation on the drone is seemingly vast, but will be in this paper considered as an obstacle. Traveling vast distances while performing tasks and calculations can render a huge drain on a battery. As the exact technical challenges related to this are known, some assumptions are made:

- The limitation in the range of the drone is a technical one. It could be the case that this limitation is mainly caused by legal or safety related issues. This is however assumed not to be the case.
- The limitation is due to the battery-life of the drone. It could be the case that the limitation was mainly due to the reach of a signal necessary to communicate with the drones. This is assumed not to be the case.

With these assumptions made, it is proposed to mend this issue using collective communication and visual techniques. Could the drone identify vehicles on the way to its destination and use them for transportation? The drone could identify public communication, namely busses, and then potentially land on them. Busses often travel for farther distances than the mentioned limit for drones. This could serve the purpose of using them for transportation, but additionally the bus could be equipped with technology to charge the battery on the drone while the drone is landed on it. The technical challenge of this can be split into two problems:

- Identifying the bus: This does not just entail correctly identifying the correct type of public communication, but also identifying the correct right route. What will be investigated in this thesis is if this is possible using computer vision, how it can be done and what the modern ways of doing this entail.
- Landing on the bus: How much of a challenge this is would depend to a large degree on the type of tracking/detecting implemented for it. What will be investigated in this thesis is if this can be performed accurately using solely computer vision, and what that would involve.

These are both points that make use of computer vision techniques, and this is the direction they will be attemptedly solved in this thesis. In this work the feasibility of these two points will be discussed, using a simplified example. Additionally the field computer vision and image recognition and the state of the art use of these technologies are discussed. As the field of VSLAM and computer vision for object avoidance is also heavily related to the field of autonomous drones, it is explained in the appendix **?**.

1.3 Application

The example is investigated using the following tools and software.

1.3.1 ROS

The the major programming will be done using ROS: the Robot Operating System (for more information about this, see Quigley et al. (2009)). ROS is to be precise not specifically an operating system, but a baseline program that helps coding in multi thread system, as well as a resource for easy usability and easy implementations of common algorithms. With ROS defining the bottom layer of the software, it also serves as a way to keep track of the different additional programs dependencies (called ROS packages), as well as how the can be used with each other using what they refer to as topics and services. In this project I mainly used 'ROS Indigo', which is an older release of the software, however the amount of packages available for the 'Indigo' release justified this decision.

1.3.2 Gazebo

Gazebo is a standard simulator that is often used back to back with ROS, for several use cases (Koenig and Howard (2004)). It provides a 3D world with a physics engine, where one can spawn 3d models with actuators, generate sensor signals etc.

The Parrot AR-drone is a currently discontinued consumer quad-copter drone. This drone was a popular product, mounted with a camera, and controllable using a cellphone for more information about this product, see **?**. Gazebo is issued with as a 3D model of this drone, that can be controlled in Gazebo. This model of a drone is what will be mainly used for the tasks presented in this thesis. The drone is equipped with two cameras (one front-facing, and one facing downwards), an IMU (intertial measurement unit) and GPS (global positioning system). The technicalities of these products will be explained further in the sections below.

More about the specifications of the Parrot AR-drone can be studied in .1. For more information about ROS and Gazebo, see Quigley et al. (2009) and Koenig and Howard (2004).

1.4 Simplified problem that will be considered

For this thesis, the problem will be simplified into the map that can be seen on figure 1.1 and more descriptively on figure 1.2. The drone starts at one (x=0,y=0), and needs to make its way to the goal (x=0,y=50). It does this by traveling along the road. The presence of the road along the y-axis at x=20 is assumed known, however the presence and the position of the bus is not. Otherwise, recognizing the road, a well as the obstruction, would be a great addition but is beyond the scope of this paper (see section **??** for further work). The bus is also assumed stationary, however the challenge relating to the bus moving would potentially not make a large difference in the usability of what is presented. More about this is discussed in section 5. More information about the bus and the drone models and what they are equipped with will be explained in section 4.

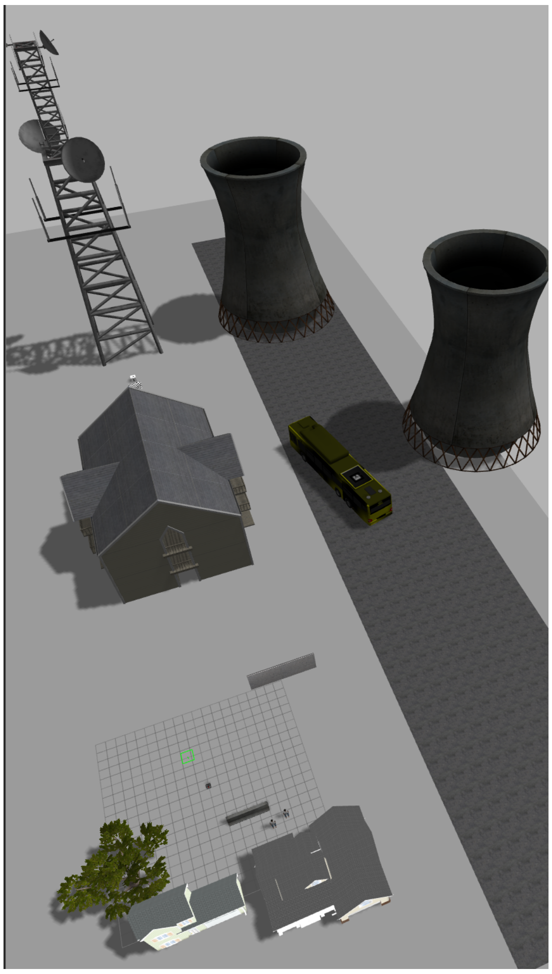


Figure 1.1: A different view of the environment createdf for the simulation.

On its way, the drone needs to detect the presence of the bus, interrupting its current path plan (ie. towards the goal at (x=0,y=50). Using some sort of visual technique it needs to guide its way to the position of the bus, and then land accurately on top of its landing platform.

This essentially splits the problem into three parts.

This problem and the test landscape shown in 1.2 was designed the way it is due to a couple of requirements that was needed to be fulfilled:

- The test can not solely consist of a drone and a vehicle starting a certain distance apart with a couple of changed initial parameters, because a movement of the drone needs to be taken into account as well. A slow detection algorithm would be detrimental in real life situation, and only real movement and maneuvering by either the drone and/or the bus would recreate this.

- The goal and starting position should be a certain distance away from each other, creating a distance that is obstructed an object. This obstruction is not necessary, but creates the need for some maneuvering from the drone as well as a reasoning behind it. This maneuvering, essentially an initial reorientation and two left turns, make the drone potentially detect the bus from several angles and positions. Both along the road and not.

- To make the environment more lifelike and show the location of the road clearly, the minimalistic road was added.

- In order for as many objects as possible to be trackable or discernable with a camera, as much detail as possible should be spread out. This especially concerns the area close to $y = 60$, as this is the direction the drone will be facing.

- Too many objects can not be placed within a field of view, as a certain amount of 3D detail can make the Gazebo simulated footage (as well as the entire simulation) lag or stutter. The amount of objects placed in the area along $y = 60$ is on the verge creating lag (that is given the performance of my system which the simulation was running on 6). This is why the rather large tower and two silos were added.

- For the sake of showing its relation to a real world setting, placing more 3D object in order to make it look more like a real life environment as possible is desirable. However this severely limited by potential for lag in the simulation. As not much detail is placed along the negative y-axis, some is detail is placed there.

- Items can not be placed more than 50 meters away, as this is seemingly the maximum rendering distance for the simulated Gazebo footage. Scenery suddenly appearing as one moves closer might not big problem but it can be an issue with tracking applications.

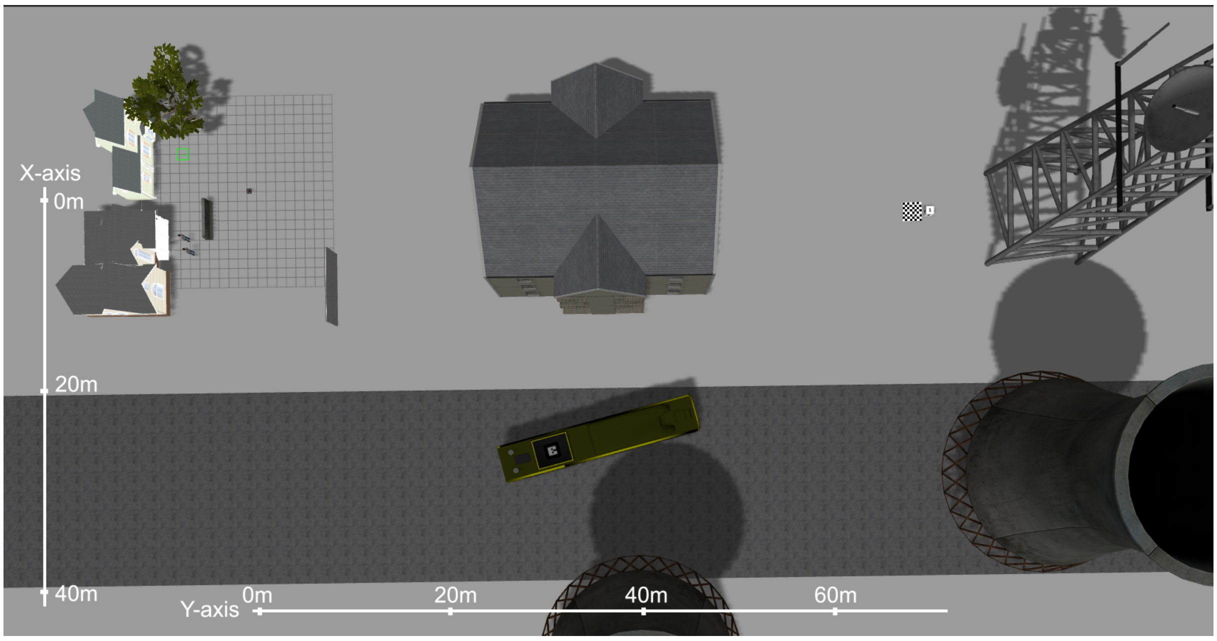


Figure 1.2: Map of the Gazebo environment that will be used as a simplified problem, rotated for a better view of the environment.

1.4.1 Scope of this thesis

In order to deal with this problem from all angles, several things could have been worked on. Some notable points are:

- In the simplified example the bus is assumed stationary. A moving bus could raise the question of whether or not the drone could remain stationary on a landing platform while the bus is moving, and under what conditions. This however will not be discussed in this thesis.
- More accuracy and better performance could potentially be achieved combining several techniques from different fields, but this will not be discussed.
- There are severel rules and regulations about the use of drones. Newer rules and regulations about the use of drones in Norway require them not to be used within an arguably large distance away from people and vehicles. Legalities and issues similar to these is also a problem for other firms attempting to do perform a drone package delivery (Simran Brar

(2015). Proving the feasibility in the terms of safety or lack of liability will not be discussed, being beyond the scope of this paper.

What this thesis will discuss is limited to the problem described in the subsection above, using the Gazebo environment. To what degree this relates to the real world potential issue of drone package delivery over long distances will discussed more in section 5, however this will not be the main focus of this work.

1.4.2 Structure of this thesis

This thesis will in section 2 explain the field of computer vision with the main focus on object detection. The newer field of visual SLAM (simultaneous localization and mapping), will additionally be explained in the appendix .2.1 due to its inherent relationship to the path planning and object avoidance used by autonomous package delivery drones.

The results and reasoning behind the techniques used to solve the problems will be explained in section 4. The results will then be discussed in section 5. In section **??** a conclusion based on the work in this thesis will be presented.

Chapter 2

Computer vision for recognition

2.1 Camera Anatomy

In order to gather information about the real world using a camera it can be helpful to first understand how a camera works.

The projective space

Images are basically created everywhere as light bounce off from everything in all wavelengths creating a uniform light exposure - a unsharp, blurry image. Focusing the image through a lens would however make it sharper. That is what a pinhole camera would do, which is essentially just a hole. The pinhole camera wouldn't gather much light, nor be very sharp, but it would create an image.

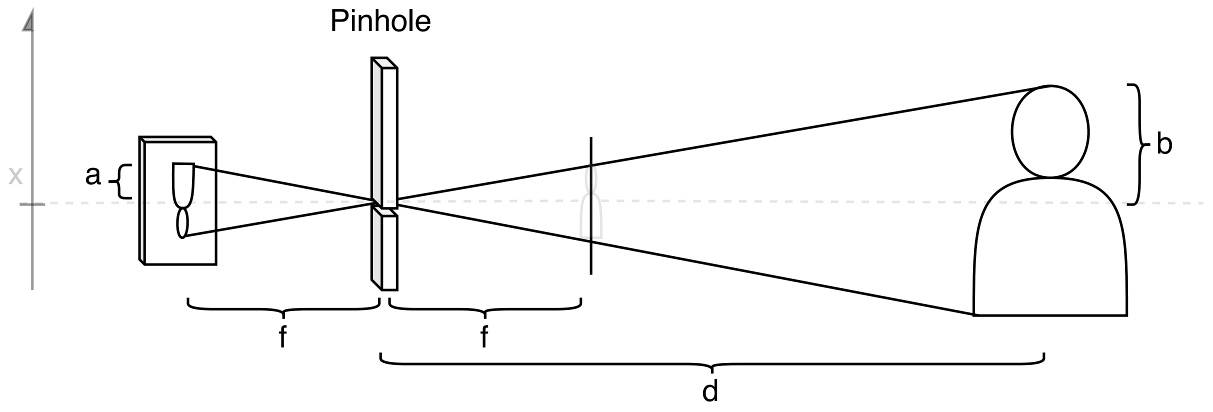


Figure 2.1: Pinhole camera model

The image will be created a distance f (known as the focal length) away from the hole, upside-down. Instead of dealing with the upside down projection, lets assume that the image is being created on an image plane a distance f infront of the hole (rendering their size the same), and lets call the hole the 'camera center'. This assumption will only simplify, if you dont care about how it is infact an upside down image. This is in effect a projective transformation from $\mathbb{P}^3$ to $\mathbb{P}^2$. For a an explaination of the projective space see 2.1.1.

Taking into account the triangles of the geometry we get that $x_1 = f\frac{X_2}{Z}$. Extending this into 3D we get the equality below. The coordinates in capital letters denote the position in 3D space, and the ones in lowercase represent the ones on the image. f is the focal length, and z is introduced to keep track of the dept (Scelizki (2011) chapter 1).

$$\begin{bmatrix} \mathrm{x} \\ \mathrm{y} \\ \mathrm{z} \end{bmatrix} = \begin{bmatrix} \mathrm{f} & 0 & 0 & 0 \\ 0 & \mathrm{f} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathrm{X} \\ \mathrm{Y} \\ \mathrm{Z} \\ 1 \end{bmatrix}$$

Definition of the Projective space 2.1.1. Unlike the Euclidian space $\mathbb{R}^2$ of cartesian coordinates, the projective space $\mathbb{P}^2$ describe a point on the 2D plane using homogenous coordinates, projected from a 3D point. This entails that any point $\widetilde{\boldsymbol{x}}$ can be thought of as a vector, from the image center to this point, and all vectors in this direction maps to the same 2D point. Or in other words:

$$\widetilde{\boldsymbol{x}} = (\widetilde{x}, \widetilde{y}, \widetilde{w}) \in \mathbb{P}^2 \mapsto \widetilde{\boldsymbol{x}} = [\widetilde{x}, \widetilde{y}, \widetilde{w}]$$

$$s.t.$$

$$(\widetilde{x}, \widetilde{y}, \widetilde{w}) = \lambda(\widetilde{x}, \widetilde{y}, \widetilde{w}) \forall \lambda \in \mathbb{R} \setminus 0$$

One can easily transfer back and forth between the protective frame and euclidean frame using that $\widetilde{\boldsymbol{x}}$=$(\widetilde{x},\widetilde{y},\widetilde{z},\widetilde{w})$$\mapsto$$\boldsymbol{x}$=$(\widetilde{x}/\widetilde{w}, \widetilde{y}/\widetilde{w}, \widetilde{z}/\widetilde{w})$ $\in$R^3

A lens, bending all light from an object into a smaller space, gathering more light, would render a brighter picture. And give us more trigonometry to keep in mind, but basically give the same effect.

Perspective camera model

The model called the perspective camera model uses the projective framework by introducing that the camera image center is a frame like any other, C, with the wide spanning image plane with its center at z=1 away from C. Placed on this image plane is the frame that is playing the part of the image/sensor itself. In order to simplify the image planes coordinate system to something more easily scalable, the origin I is moved to the top right with its axis u and v spanning the image.

In other words we get the two transformations below, one for transforming from 3D to 2D, the other transformation from the normalized image plane to I.

$$\widetilde{\boldsymbol{u}} = \begin{bmatrix} f_u & s & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \widetilde{\boldsymbol{x}}$$

Lets scale from meters to pixels. Using the new coordinates u and v with its origin to the top left of the image plane, as well as working with the resolution in both directions ρ_u and ρ_v, we get the following coordinate transformation.

$$\begin{bmatrix} \mathrm{u} \\ \mathrm{v} \\ \mathrm{w} \end{bmatrix} = \begin{bmatrix} \frac{1}{\rho_u} & 0 & 0 \\ 0 & \frac{1}{\rho_v} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \\ \mathrm{z} \end{bmatrix}$$

$$\begin{bmatrix} \mathrm{u} \\ \mathrm{v} \\ \mathrm{w} \end{bmatrix} = \underbrace{\underbrace{\begin{bmatrix} \frac{1}{\rho_u} & 0 & \mathrm{u}_0 \\ 0 & \frac{1}{\rho_v} & \mathrm{v}_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{f} & 0 & 0 & 0 \\ 0 & \mathrm{f} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{K}} \begin{bmatrix} \mathbf{R} & t \\ \mathbf{0}_{1x3} & 1 \end{bmatrix}^{-1}}_{\mathbf{C}} \begin{bmatrix} \mathrm{X} \\ \mathrm{Y} \\ \mathrm{Z} \\ 1 \end{bmatrix}$$

The intrinsic parameters of a camera is given and/or needs to be found beforehand, before anything more can get done. This will give you the camera calibration matrix K. This describe things that are innate and unchangeable about the camera itself. The new f_u and f_v denotes the vertical and horizontal scale. c_v and c_u are the optical counterpoints of the image. S is the skew parameter; if the image is not orthogonal to the z-axis, resulting in somewhat of a distortion. This is not an ideal parameter to have a great value of. There are also radial and barrel distortions that can arise from non-optimal cameras, but in effect will appear to some degree in all cameras. The effect of these other distortions can sadly not be mended in the matrix transformations like the rest, but is instead usually undistorted beforehand using a seperate procedure for that.

$$K = \begin{bmatrix} \text{f} & 0 & c_u \\ 0 & \text{f} & c_p \\ 0 & 0 & 1 \end{bmatrix}$$

In order to find out how much distortion is present, as well as all other intrinsic camera parameters, we calibrate the camera using software such as matlab or opencv. This is typically done using pictures of a chessboard, with its size and dimensions already known.

Further explanations about how this relates to the calculation of 3D point placements is described in the appendix section .2.1.

Using the calibration matrix K, one can among other things calculate the field of view of the camera, and the angle associated to a certain pixel position. Using the calculations above one can also use this to place the points found on an image in 3D space. This is the technique used by the drones to avoid obstacles, and is explained in the appendix .2.1.

2.2 Classical Computer Vision

Of all the tasks a computer might perform, object recognition and analyzing a scene remains one of the most challenging. Even if the recreation of a scene using the techniques described in .2.1 in the appendix are successful, it does not provide any understanding of what the objects in scene are. There are several reasons why object recognition is difficult. One points is that there is a large degree of variability within what could be determined as the same object. Several seemingly different things can be considered a table for instance, even the same table can be oriented and placed in such a way that it can be hard to recognize. Additionally, objects can occlude each other, making eg. having a certain body part visible not a prerequisite for something to be recognised as a human (Scelizki (2011) p.577).

"Object detection" is the problem of locating an already specified object, wanting the potential position of this object within an image. If there is a specific physical instance of an object we need to locate, the problem is referred to as "instance recognition". Instance recognition can then be thought of as the next step, as having detected the object is often necessary before attempting to recognize a certain instance of it. There are other types of recognition problems as well, such as "category recognition" (for more on this, see Scelizki (2011) chapter 14). This section will mainly describe the potential frameworks for performing object detection.

Performing object detection can be broken up into two major categories: using classical computer vision methods and neural networks. In the following section object detection using classical methods for object detection will be explained, followed by how this is done using machine learning.

Classical computer vision utilize several arguably simple methods for interpreting an image. In this section a couple of them will be explained, followed by how they can be used for object detection.

2.2.1 Filtering

Filtering has several uses in computer vision. It can be used to remove noise and high frequency detail. This can be realised on an image by calculating new image values for a pixel by letting it be a function of the average of those surrounding it. This is how it works in a general sense, but more complexities allowing for maintaining sharp edges etc. exist (Scelizki (2011) chapter 3.). These remaining sharp edges can be further used in a an edge detector.

When a certain object needs to be tracked, knowledge about its distinct color can often be present. Using this, one can also use filtering to separate this color from the background. Leaving one only with the pixels that are important. This can be distinguished using a range in RGB values, but additionally the color space can be converted into HSV (hue, saturation, lightness) or HSV (hue, saturation, value). These are alternate representations of the color space and can be filtered as well. These other representations can among other things make it easier to filter out grayscale color ranges (Scelizki (2011) chapter 3).

2.2.2 Edge detection

Edges can be described as a rapid intensity variation on an image (Scelizki (2011) chapter 4). In other words, this can be described as the rate of change in the intensity of the image at different points, or the gradients. We can describe this using a gradient vector $\boldsymbol{J}(\boldsymbol{x}) = \nabla I(\boldsymbol{x}) = (\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y})(\boldsymbol{x})$ with I being the image intensities (image brightness, in the grayscale case this would simplify into being the numerical pixel value), and $\boldsymbol{x}$ being the pixel location. (Scelizki (2011) Chapter 4).

Using a calculation such as this, edge detectors can look at neighbouring pixels and see compare their intensities in order to detect edges. Additionally, usig this one can seperate the countour of a shape wanted to distinguish.

If one wanted to find the outline of a blue table, one could start with using a color filter, in order to then use the edge detector. A framework much like this could be used to provide eg. the corner points of a table, if its color is distinguishing enough.

2.2.3 Key-points and features

Feature detection and matching are an important section in the field of computer vision, and is used for several applications, such as image stabilisation and image stitching. A certain point in the image that stand out, fullfilling some criteria doing so, is called a key-point. This is essentially a point of interest, and can be used to for tracking. A 'feature' of an image is essentially a keypoint being noted and saved, but additionally a detailed description of this part of the image (known as a desctiptor) is accurately created and stored.

In order to collect these features, one do what is called feature detections. When performing feature detection, points of interest is identified. These are often chosen due to distinctive part of the image in the surrounding area of the feature. An example of this is a point feature, using which one usually describe the surrounding couple of pixels. However this surrounding grid of pixels (or pixel "patch") need to be different enough from the surrounding pixels, in order to not be misplaced. These point features can be used to relocate themselves on a scene from another POV, and find a sparse set of corresponding locations in different images. There are several different types of features, mainly different due to their different descriptors. The features are often described by their appearance using the patch of pixels surrounding them. Edges are another, and can be described by the entire shape against another. For instance could the shape of a mountain be used as a feature against the background that is the sky. These and more can be used to analyze pictures and compare them. This would involve finding one feature from one image in another, and require an accurate feature description.

Patches with large contrast are the easiest to localize. Straight lines suffer from what is called the aperture problem, in that the gradient in this patch only change in one direction, so a placement along this line is hard to do. They can infact only be placed along this line in the orthogonal directon (ie. given that the coordinate of the orthogonal dimension is known). A feaure that is easier to localize needs changing gradients in two directions significajntly differnet. How much the gradient changes in directions would depend on the size of the patch, and introduce the topic of scale invariance. In order to take into account the different potentional scale of features, one could choose features at different scales. However, it is found that it is more efficient to extract features that are stable in both localization and scale instead. Another problem that arise in addition to scale invariance of a feature is rotation invariance, this can be solved using feature descrpitors that are rotationally invariant, but a better way is found to estimate a dominant orientation, and use that in the feature desctiptors. So once a feature is detected, the feature is stored by first finding a re-scaled and re-oriented version with a patch around the point in question. Additionally, in order to be invariant to changes in lighting (ie. shadows and the like, can make the feature patch look different) a feature needs to be made invariant to these changes. This is done in different ways in the different popular feature descriptors. The popular feature descriptor SIFT(Scale invariant feature transform) for instance simply calculate the image intensity gradient for the pixels on its patch. This can have the effect of it being invariant to light changes, as eg. a shadow would not necessarily effect the change in gradient between the pixels. SIFT computes the intensity gradient in 16x16 patch around the point. It additionally stores the info of the associatated four quadrants of the patch. All of these values are saved and a part of the descriptor.

As there are several feature detectors and descriptors, choosing which one to use is an important task. This is usually solved checking the "repeatability" of the detector, ie. checking how many of the same features are found in an image transformed in a random way. Even though diffrerent choices are made in regards to this, depending on the application, Schmidt Bohr Baukhage (2000) claim that certain operators work best for the feature detection.

2.3 Tracking

2.3.1 Feature tracking

There are two ways of doing feature tracking: independently find features in all images, and then match these using their descriptions and the other which is tracking the features. The other is more used with video, when the next frame is fairly similar to the previous. Here the features proximity to the location of the one one in the next frame is used to make this process easier. A search area around the features previous position is used for this. As noise and motion can have changed the feature, it wont have the exact same description when found again in comparison to what is stored in the database, but when the likeness between the two is above a cedrtain threshold it is considered a match. For more ifnormation about this see Scelizki (2011).

2.3.2 Optical flow and tracking

To estimate motion between images an accurate error metric must be chosen. Depending on the complexity of the motion and precision wanted, several methods can be used. The most general and challenging version of all of these is the problem of optical flow, which entails estimating the motion at every pixel. As opposed to the tracking of features, optical flow usually do not save accurate feature descripter of the keypoints it use. Instead it usually involves minimizing the brightness or color difference between corresponding pixels summed over the image. This can be refered to as "brightness constancy assumption", assuming no shade or anything similar changing the keypoint being tracked. The image intensity (ie. lightness) is tracked assuming only a change in its position. This is however not enough constraints to solve the problem (Scelizki (2011) chapter 8.4). There are several algorithms in this subfield, that all take different assumptions about the problem, making them the appropriate choice in different situations (Scelizki (2011) chapter 8.4).

Optical flow is a method to discern the movement in images such as edges, and is a technique within the field of image processing. There are several good algorithms in this field, however one that is wanted to mention is the Lukas Kanade method. The Lukas Kanade method looks at image patches of certain features, assuming that motion is constant within this area. This results in a more sparse set of tracked 2D points (as opposed to other methods for calculating optical flow, such the Gunnar Farneback Algorithm or the Simple Flow algorithm Bradski (2000)) that potentially track for longer, giving a set that is arguably easier to work with (Scelizki (2011) chapter 8.4).

2.3.3 Object detection and instance recognition using classical methods

Combining the techniques in the above section one can track objects. As opposed to machine learning object detectors (that will be explained in the next section) there are different ways of creating detectors using classical methods. As the object and use case case in question call for different methods, different methods are used.

In the context of object detection several methods are usually stringed together in a "pipeline". An example of a popular topic in pobject detection is face detection. In this category Viola and Jones (2001) stand out as the first object detection framework to deliver real-time frame rates, in addition to great robustness. This is achieved searching for simple features that are usually found on a face, namely the difference in lightness between the cheeks and the eyes, and afterwards two other parts of the face. These are called HAAR features (Viola and Jones (2001)). In Viola Jones face detection they adds up to a small pipeline of feature comparisons. These features are not like the ones using ie. SIFT, but is still in essence an example of using features for detection. These cascading Haar features are a popular choice object detection, finding a image intensity pattern in the object.

Typically object detection use several techniques to work together (or in 'cascade' or 'pipeline'), in order to create a result that is reliant. Another example of which is the following:

- Filter the color of the item you wish to track.
- Use the area of this filtered color to produce points that can be tracked, using a good feature descriptor. Combine these to get an estimate of the objects center and change in position.
- Taking this further, the crossectional area produced by these points can be used to estimate a change in orientation of the object. Potentially even in 3D.

For object recognition, this type of pipeline is common. If more of the object is known, several features and their feature descriptors can also be saved in advance. When an area with enough inliers is found (much like loop closing in orb, see .2.7), it can be considered a match. There are programming libraries such a ’Object Recognition Kitchen’ (Willow Garage (????)), that provide ready-made pipelines databases of features associated with different objects.

A pipeline of techniques used together to perform one precise task arguably provide a less generalized result than an object detection neural network. However, the use of special purpose detectors generally produce a more effective and rapid result (Scelizki (2011)), and is for that among other reasons, often opted for.

2.3.4 Fiducial markers

As opposed to short term tracking using a patch of pixels as a feature, a fiducial marker provides a very distinctive and potentially already known contrast histogram. This makes it easy to track. Knowing its size and dimensions as well, one can additionally solve for its entire 3D pose. This can be solved for in real time using plugins from the opencv library. There are several types of fiducial markers, and the type tracked using opencv applications is called the Aruco marker.

2.3.5 Opencv applications

’Open computer vision’ is a popular programming library (Bradski (2000)) for real time computer vision. It offers easy an easy to use framework for several applications, including several of the mentioned above. Opencv and its applications is often opted for when creating real-time computer vision software, as they are both fast, reliant and robust (the methods of classical computer visions reliance and robustness will be discussed more in 5.2).

2.4 Machine Learning

The basic overview of machine learning in the section below is from Elfving (2018).

There are several techniques that can be considered an AI algorithm or method. They’re all mainly characterized by attempting to mimick a human way of solving a new problem. Machine Learning are the subset of the AI methods that would use previous data to help them, and would typically get better and better the more old data they have to work with. As the data being gathered could be influenced by stochastic properties, the resulting AI would also be, and thus will not necessarily arrive at neither a perfect answer, nor any answer at all.

2.4.1 Perceptron

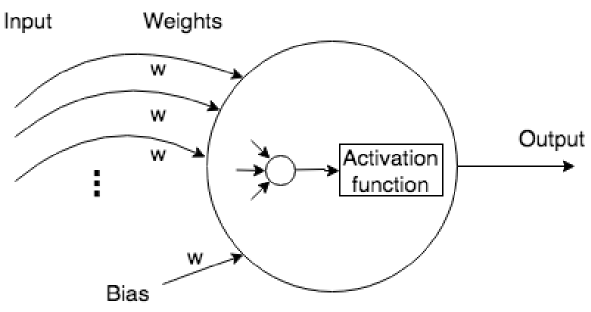


Figure 2.2: Block diagram for a perceptron.

The simplified model of a human brain cell can be made as simple as the model above. The cell gets a certain electrical signal, and if this signal is above a certain threshold the cell sends another signal further. Where this limit is however is dependent on the cell. Considering this model, and adding the possibility of multiple inputs, we find that we can arrive at many binary classifications as long a set of values were chosen well enough: how much we should weigh each input signal (often referred to as the input weights), where along the y-axis the threshold should be (bias) and what threshold the total sum would need to rise above for the cell to give an input (the activation function). This results in what we call a perceptron. With all of these chosen correctly one could arrive at any binary linear classification of these input values.

These can also often be refered to as neurons and nodes, all depending on the context.

2.4.2 Artificial Neural Networks

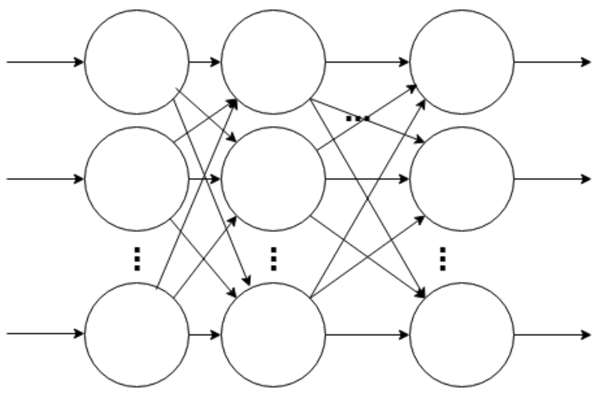


Figure 2.3: Block diagram for a neural network.

If you create a network of perceptrons, ie. letting the output of one or more perceptrons be the input of another, then you have an artificial neural network. With a high enough amount of perceptrons with well enough tuned parameters, any function can be approximated using a neural net**?**. The implications of this fact are wast. That means that neural nets could be an option no matter what the problem is, as long as you have large amounts of data and a way to express how well the neural net performed.

If you now connect this grid of perceptrons to the grid of pixels that is a picture, one can get in the same way learn the network to extract something meaningful out of it. A simple network of about 6 layers of perceptrons, also known as dense layers is bottom tier of ML aided computer vision. Using this dense 6 layers one can solve the famous MNIST dataset, a dataset of small grayscale handwritten digits from 0-9. This is considered the "Hello World" of ML image recognition.

2.4.3 Convolutional layers

There are several ways to store the data of a color image, but a common one is the use the different channels RGB, for Red Green and Blue color, as all colors can be described as mix between these. If a neural network where to make use of this, it would have to be connected to all of these three layers. This means that the input image of a neural net will have 3 layers, or a depth of 3. Another way to think about this depth of an image is that is another additional way to describe different features of the one image. Creating more of this depth to signal can effectively work as a series of ways to express the information stored in the image. There are several ways of dealing with depth in neural nets. We can get more of it by using convolutional layers.

Convolutional layers are layers with filters such as the ones below. The filters iterate through the size of the image, calculating their elementwise matrix multiplication. Using this we can get more substantial information out of an image. Consider for instance the matrix in 2.4a, performing an elementwise matrix multiplication with this and a part of an image would output something close to zero if the part of the image in question was the same shade, no matter the color. However with a vertical change in shade, it would create an output, being essentially an edge detector. 2.4b is essentially the same, just being at an angle. There are more of these of different sizes and shapes. Normal convolutional layers will have random filters, and many of them. An edge is an example of a feature. With more convultional layers in succession they will effectively manage to abstract or generalize more and more information, going from edges

1	0	-1
1	0	-1
1	0	-1

(a) Vertical edge detector

-1	-1	0
-1	0	1
0	1	1

(b) Angled edge detector

1	1	-1
2	1	-1
0	-2	2

(c) Random filter

Figure 2.4: Example off small 3x3 convolutional filters

to basic shapes, to eg. body parts. This additionally has the added effect of not requiring what is to be detected to be centered in the image, as the convolutions filters are smaller and goes through the entire picture with the same filter.

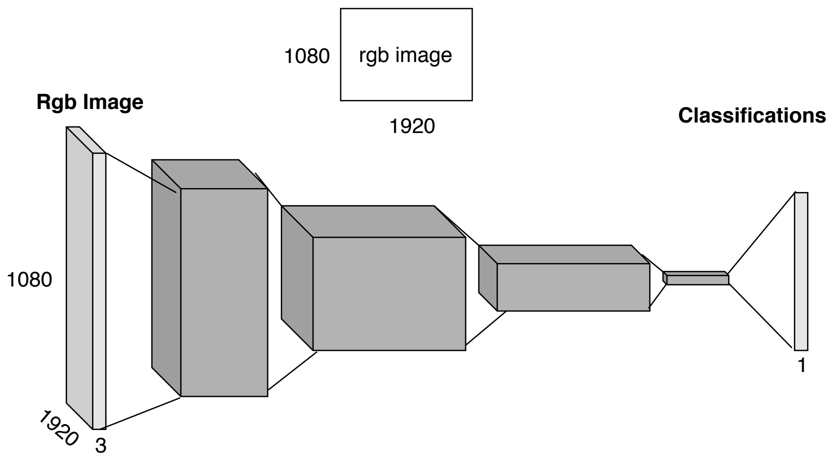


Figure 2.5: Typical illustration of a convolutional neural network.

Figure 2.5 show a typical convolutional neural network architecture, albeit simplified. Note the steadily decreasing length and width and increasing depth of the convolutional layers, which is typical as more features can be extracted and their position matters less. The last layer(s) are often dense, and the final layer is ofcourse flat, being a series of potential classifications.

2.4.4 Different layers and concepts

There are several other additions to neural networks that are integral parts of the entire architecture. There are also variatians of all of these, eg. for a 3D element as well as scalar value.

Maxpooling

Instead of the convolution where the elementwise matrixmultiplication is calculated, there are layers where the element with the most value is outputted. This has the added effect of creating a layer that can output what has made the biggest impact. Minpooling and average pooling are also used.

Normalization vs regulatication?

With extreme values on several images, the features in the image are argueably the same with a lighter image (and especially not for a neural network, as its the correlation with the original iamge that matters). With very large input values into a perceptron, it needs to adapt its bias value into something extremely large. With larger differences between elements such as this, the perceptron generalize worse. Normalization can be used as a alayer, but also as something that is generally used with all data before both training and forward-passes. It offsets the mean value of the input and noralize the differences within a certain mean. This makes it easier to work with and generalize.

Activation functions

There are several activation functions, and the choice of which can play a large role due to their respective nonlinearities and gradients.

Dropout

Certain neurons are often trained have a large weight and make a big difference, creating the situation that some neurons are makeing it so that the others are not being trained. In order for a more robust network, a certain percentage of random nodes are simply turned off during a batch of training, training all neurons more.

2.4.5 Object detection

Using the techniques described above one can now classify an image to be a member of one class or another. Another feat however is knowing where in an image an object of a certain class is. This is typically shown using a bounding box, a box marking a square around an object. This could can be theoretically achieved using what is called a sliding window. The sliding window technique marks a small area in our image and feeds it through the neural network to get a prediction, and does this iterating through the entire image for several different sized images. This will eventually provide a large amount of predictions with their corresponding bounding boxes, where one can disregard the lowest probability predictions, leaving the true object detections.

Iterating through several different sized windows for the entire image is a very long and computationally ineffective process however. Instead so called region proposals are calculated before feeding the resulting region through the neural network. R-CNN (Region - Convolutional Neural Network) was the first neural network architecture to use region proposals. Instead of iterating through all potential region sizes, a number of randomly sized frames were generated and fed through the CNN. After a resulting prediction outputs a large probability, the region is run through a linear regression to refine the position of the bounding box.

2.4.6 Comparing object detection algorithms

When training a neural network for object detection, as well as comparing the performance of one method against another, several techniques are used (for more about this see Scelizki (2011)).

With several algorithms trained on the same dataset, they have a similar starting ground and can be compared properly. Given a dataset with a correct set of labeled bounding boxes, one can calculate how much of an overlap this is. This is referred to as the intersection over union (IoU). In IoU > 0.5 is often considered a correct prediction.

$$\boldsymbol{IoU} = \frac{AreaOfOverlap}{AreaOfUnion} \tag{2.1}$$

Given that a prediction was considered correct, we can define an algorithms number of true positives (TP), false positives (FP), in order to get the mean average precision (mAP).

$$\boldsymbol{Precision} = \frac{TP}{TP + FP} \qquad \boldsymbol{Recall} = \frac{TP}{TP + FN} \tag{2.2}$$

$$\boldsymbol{mAp} = \frac{1}{\#classes} \sum_{c \in classes} \frac{\#TP}{\#TP(c) + \#FP(c)} \tag{2.3}$$

Another factor is offcourse the speed of the algorithm, and what range in size of objects the network can see (the size of the first convolutional layers can make this difference). Even though the mAP as functions of size of objects etc., are intersting statistics, the mAP and speed of the algorithm on a dataset are the most common ways of comparison.

2.4.7 Popular ML object detection algorithms

There are several ML object detection algorithms that perform well, two of the more popular among these will be presented in this section. These are are YOLO(You Only Look Once) and SSD(Single Shot Detector). Their performance are arguably among the greater as of this paper, however no exact claims what algorithm perform the best will be made in this paper, as these numbers are continously changing.

YOLO: You only look once

Yolo is an example of a image recognition neural network. It gets its name from that it only do the massive forward pass in the neural network only once per prediction. This is what makes it among the fastest, as a forward pass through several convolutional layers is the most computationally demanding.

In order to do this, the frame is divided into a grid of S by S sections. Each section from then predict N potential bounding boxes. Each of these estimations return a confidence in the prediction of the bounding box in question contains an object, what the object is, and the precision of the bounding box.

Considering all of these bounding boxes of predictions, they are all taken into account making a rough resolution of an estimated bounding box of an object. The bounding boxes with predictions below a certain threshold are discarded. (LITT FEIL: REWRITE)

YOLO is being updated with newer versions continuously, with the newest version being its third. However, its main selling point remains being the fastest/computationally lightest. This section is based on the papers presenting Yolo, for more about it read Joseph Redmon (2018).

SSD: Single Shot Detector

SSD is also popular choice in in object detection. SSD feeds the input through a convolutional filter, making a feature map. After this a 3x3 kernal is used to find bounding boxes and category probability. Much like R-CNN, SSD also uses bounding boxes of multiple sizes. It then learns the correct offset?

So, total SxSxN boxes are forecasted. On the other hand, most of these boxes have lower confidence scores and if we set a doorstep say 30Single Shot Detector (SSD)

SSD attains a better balance between swiftness and precision. SSD runs a convolutional network on input image only one time and computes a feature map. Now, we run a small 3×3 sized convolutional kernel on this feature map to foresee the bounding boxes and categorization probability.

SSD also uses anchor boxes at a variety of aspect ratio comparable to Faster-RCNN and learns the off-set to a certain extent than learning the box. In order to hold the scale, SSD predicts bounding boxes after multiple convolutional layers. Since every convolutional layer functions at a diverse scale, it is able to detect objects of a mixture of scales.

SSD is a healthier recommendation. However, if exactness is not too much of disquiet but you want to go super quick, YOLO will be the best way to move forward. First of all, a visual thoughtfulness of swiftness vs precision trade-off would differentiate them well.

SSD is a better option as we are able to run it on a video and the exactness trade-off is very modest. While dealing with large sizes, SSD seems to perform well, but when we look at the accurateness numbers when the object size is small, the performance dips a bit. This section is based on the papers presenting Yolo, for more about it read **?**.

2.5 Image recognition

Image recognition could help out with autonomous navigation, being able to identify what is around the drone, such as poeple and vehicles, so that it can for instance know to avoid them. As was mentioned in the introduction .2.16 it was suggested to add that the drone could land on a buss in order to get charged and get further in the correct direction. In order for this to work the drone would also need to recognize the busses that can provide this service. Lets assume this service is provided by Trondheims characteristic green public communication AtB, with its distinct green busses. A neural net could be trained on the identifying these green busses. Options here would potentially be general color tracking, however this wont be very robust. Even though this is also the case with ML and that reliance on ML is not optimal (.2.16), this seems to be the only way to effectively recognize a bus.

From there it was considered what neural network architecture could be used for this. YOLO and SSD was the main considerations. An argument could be made about the size recognition abilities of the two, considering the wanted height of the drones flight. As SSD has a somewhat lesser ability to recognize smaller objects, as its first convolutional layers are a size of 38x38px, should it be ruled out? This is noted as just a small dip in performance, and all image recognition software would naturally perform worse with smaller objects, but the hard limit of 38 by 38 pixels is an interesting point. In other words, the smallest potential feature it can distinguish is 38x38 pixels. This can be mended with a larger image output, but consdering that the standard ardrone outputs SD, 640×360, we can calculate its maximum bus-observable height considering the intrinsic camera parameters of the ardrone .2.16 and the size of a bus.

SSD potentially provides a larger accuracy in its recognition as well. However, a more important point is that YOLO requires less computations, and can then run with a faster framerate (if real-time performance is wanted). Even if real-time performance is not necessary, the computational should be considered important when running alongside something as computationally heavy as VSLAM. It was decided to use YOLO.

YOLO could be trained to identify whatever. There are no dataset full of images of AtB's standard Trondheim busses, and creating one is hard, and beyond the scope of this paper. So instead of training a network on the AtBs distinctive busses, a set of weights for yolo trained on the VOC12 was used, as the VOC12 dataset has busses (as well as 19 other classes).

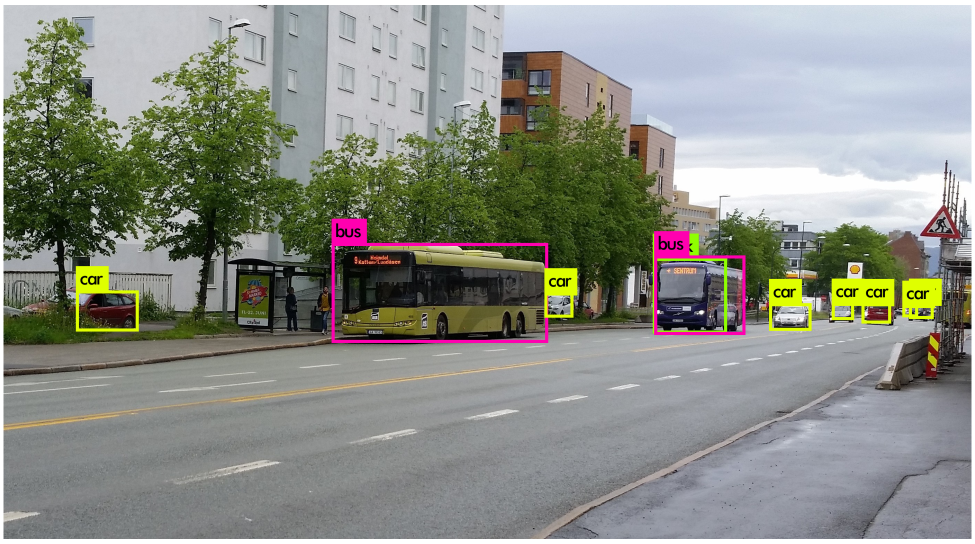


Figure 2.6: Yolo recognizing an AtB bus, as well as other vehicles. Note the other bus that would offcourse also be recognized.

Chapter 3

Literature Review

As the field of computer vision and autonomous navigation have made extreme progress the past years and is continuously evolving. This is tied to the growing indoor applications for mobile robotics, that feature several obstacles and does not allow GPS. Reducing the computational complexity is among the state of the art problems that have been worked on the past years. This achievement is mainly reached through spatial augmentation, taking advantage of the sparsity of the SLAM problems high dimensional space (T. Bailey (2006b) and Cadena et al. (2016)).

3.1 State of the art ML Object detection

The field of object detectors using ML is a field with constant improvements, that can be a challengen to keep track of. The comparisons considered below, as well in the rest of this work is the current state of the art as of late 2018, according to Joseph Redmon (2018). The current better performing algorithms for this can be split into the two categories, considering the accuracy of the speed of its performance. When comparing results, both accuracy and speed is also important to keep in mind as a high performance in one is can be compensated with low performance in the other.

3.1.1 Accuracy

Tsung-Yi Lin (2018) introduce the "Retina Net", which provides a close to 40% AP on the popular "Common Objects in Context" dataset, also known as the COCO (et al (????)) with 5 FPS. It does this by using a combination of previous techniques and introducing a new cost function. This is state of the art accuracy, with an also impressive speed. It is troublesome to achieve both high speed and high accuracy.

3.1.2 Speed

SSD and Yolo are continously facing improvements and updates. The third and newest update to Yolo (see Joseph Redmon (2018)), seem very fast and accurate. It reports the same accuracy as SSD, while 3 times the speed. Its accuracy is also comparable to that of the "Retina Net" by Tsung-Yi Lin (2018), but lower. According to the paper it runs faster than any other current ML object detector. Joseph Redmon (2018) additionally argues about the accuracy metrics that COCO uses, and that humans cant even see the difference between an IoU (see 2.4.6) of 0.3 and 0.5. As explained in section 2.4.6, an IoU of more than 0.5 is typically considered a correct prediction.

3.2 Image recognition aided through VSLAM

The field of visual SLAM (Simultaneous Localization And Mapping) use visual techniques to map an area. For drones to be able to perform object avoidance visual SLAM is usually the technology that is being used, making it presumably a technology that is already present in package delivery drones. As an area is mapped, it is found that image recognition can be used in cooperation with this technology to provide an even more accurate image recognition, taking advantage of its ability to place an object in 3D space and then assume a view of it from several frames. Some knowledge about the technology terms related to VSLAM may be needed to understand the subsections below. The explanation of the inner workings of this technology, and more about its state of the art implementations can be found in the appendix .2.16.

3.2.1 Semantic SLAM

Making use of the semantic data gathered through creating a map could be interpretable, and used. However the problem of semantic information is still in its infancy (Cadena et al. (2016)) even if it is an important problem, as it lacks a cohesive formulation. A basic part of it is categorization of what it is seeing, however it should involve high level understanding and related actions as well. Making use of semantic information in a situation is a very basic idea for a human, as the semantic information affect all of someones other reasoning.

When it comes to categorization, which is arguably an important part of semantic information, several strides have been made. This objective is often also refered to as semantic mapping, or semantic SLAM. Daniele De Gregorio (2018) and others use object recognition algorithm alongside SLAM to place objects in 3D space. K. Himri (2018) takes this a step further by using object recognition in cooperation with SLAM, by using objects as landmarks. Sudeep Pillai (2015) use SLAM to aid in the object recognition, as on abject in a known 3D space can get a combined resulting prediction using multiple vantage points. This is essentially showing that SLAM and object recognition perform better when working together. Trung T. Pham (2018) make use of DL to create a recognition algorithm that segments and classifies the individual pixels as members of a class, successfully classifying walls etc. This also has the added ability of finding the pixels that are not classified: unknown objects. Finding unknown is something state of the art 2D classification algorithms such as YOLO can not do. Vibhav Vineet (2018) performs semantic SLAM by identifying a dense on a large scale semantic SLAM outdoors, in real time. Using the fact that all of these techniques require prior knowledge about the objects being identified, Fei and Soatto (2018) and Renato F. Salas-Moreno (2013) use a database of 3D models of the objects it is identifying. Using a 3D model of what is to be identified, it manages to identify the objects pose as well. This allows it extrapolate information about the rest of the object, even if parts of it is not within view. In a similar fashion Niko Sünderhauf (2019) use predictions by YOLOv3 over a couple of frames (see 2.4.3) to create 'Quadrics' (3D surfaces such as ellipsoids) to overlap the object. This essentially estimating the 3D shape of an unknown object.

The semantic mapping with place recognition from Niko Sünderhauf (2016) provide more semantic information by using a convolutional neural network 2.4.3 for place recognition to label places in a gradually built map. In Staffan Ekvall (2006) the semantic information about labeling rooms was done by identifying objects associated with certain room labels.

3.3 Related work

As explained in 2.3.1, the classical methods of computer vision and recognition is usually a specialized identifier. This makes it hard to arrive the state of the art within object detection usijng classical computer vision in general.

In the context of landing quadrotors and with large similarities to the scope of this thesis, et al. (2017) creates a framework for a quadrotor landing on moving platform. It claims that it is the first time demonstration of a quadrotor landing on a moving target without relying on external infrastructure (ie. GPS and IMU). In order to autonomously move it relies on visual odometry (see .2.1 in the appendix), and no IMU or GPS. The landing was performed using a fiducial marker on the landing platform, and using which it managed to get great results. Some other notable points in this work is the use of Kalman filtering to aid with loss of tracking of the landing platform, and minimum jerk (the third derivative) reference modelling. This work has similarities to this thesis beyond the actual scope, in that the both the search and movement of the platform are interesting topics for the future of this project but not within the scope of this thesis. **?** describes other attempts at autonomous landing on statically placed targets, and the different landing platform markers that can be used. The problem with other methods, such as identifying an H, or a symbol or a moire pattern is, that it may not be observable for too large heights. A series of black and white circles was presented as a potential better option (**?** p. 482-491).

Hoang (2017) also attempts a landing, but on a ground vehicle that is additionally detected. The quadrotor used in Hoang (2017) has a downwards-facing camera, allowing for continous following of the vehicle from above, while keeping it in track, and land. In order to perform this detection in real time, SIFT feature matching is used. Using a set of features stored off-line for several vehicles, they can then be identfied on-line using a visual-bag-of-words model, as created by et al. (2004). Bag-of-words is mainly a text categorization tool, that uses a series of words placed in one bag, to help categorize an entire text into the same bag. If the certain amount of words found in the text belong in this bag, the entire text is categorized there as well. The Bag-of-visual-words from et al. (2004) do the same thing stored visual features. Hoang (2017) further goes into the controller scheme necessary to ensure the track of the vehicle below.

3.4 Drone Package delivery

The state of the art within drone package delivery in general is briefly touches upon in the introduction of this thesis. As this is a field with many new commerical developments, it can be assumed that much of the related state of the art technology is not presented exactly as its developed. Instead one can get an idea about the current state of the art by reading about the current benchmarks presented. For description of that beyond what is presented in the introduction is beyond the scope of this thesis.

Chapter 4

Results

The choices made in section above was implemented in ROS and its simulation environment Gazebo. ROS has several packages made by its community that are made to be easily implemented, several of these were used for the implementations below.

4.1 Simulation and thirdparty packages

As previously mentioned, VSLAM and VO produce several services, such as positioning and mapping. Due to the promising results of DSO showed in section .2.17, and the potentially less computational power needed, it was the first choice for a visual odemetry method in the Gazebo environment. However it was shown that inaccurate pointclouds were very quickly made. Due to this, it was made clear that maintaining an accurate point cloud for object avoidance is the most challenging and important, as its added benefits in position comes second to the possibility of object avoidance in this project. Instead of DSO, a VSLAM or VO that produce a dense pointcloud, and performs point refinements was needed. As LSD-SLAM fulfilles both of these requirements, it was chosen. It has the ability to refine the position of points, work on large scale as well as being selectively dense.

This is not necessarily proving that DSO performs any objectively worse than previously thought however; the placement of these pointclouds could have potentially rendered a superior localization of the drone. This is arguably also a symptom of a poor field of view of the drone, its fast movement in comparison, the lack of detail in the gazebo environment, and its monocular cameras. Laxit Gavshinde (2013) describes how monocular VSLAM systems often suffer from this, as when camera motion is subject to steep changes in orientation, tracked features over the previous instances are lost. This can make monocular VSLAM estimates highly unreliable, erroneous and unrecoverable. This might not be a problem in other situations, where loss and regaining of track rarely happens.

In the subsections below, some of the other additional third party packages used for this will be explained.

4.1.1 Autonomy Lab: Ardrone Autonomy package

Making a drone hover staticly, and avoid tipping over, is an unstable system. Controlling this requires ie. a PID controller. Developing this however is beyond the scope of this work, and will instead by solved by using a piece of thirdparty software, in the form of what is called a ROS package.

The 'Ardrone Autonomy package' offers an easy to use framework for control, using the standard ROS tools for communication: topics and services. It has stabilized the drone, and provides a framework for sending commands to the drone in the form of angular and linear velocities, as well as taking off and landing the drone.

PID controller parameters			
Coordinate	K_p	K_i	K_d
Yaw	2	0	1
Roll/pitch	10	0	0.5
XY	5	0	1
Z	5	0	1

Figure 4.1: The PID values used in the Ardrone Autonomy package. Note that it additionally use threshold values.

This is done using a PID controller. For a basic explanation of PID control, see .3.4 in the appendix. The PID parameters it used for this can be studied in 4.1. For more information about this software, see of Simon Fraser University by Mani Monajjemi (2012).

4.1.2 TUM: Ardrone Control package

The 'computer vision group' of TUM (Technical University of Munich) has also created a package for the autonomous control of the drone. The 'Tum ardrone package' take advantage of "ardrone autonomy", and has created the basic navigational software needed to control it to move to a certain coordinate (ie. it has calculated the inverse kinematics related to controlling

PID controller parameters			
Coordinate	K_p	K_i	K_d
Gaz	0.6	0.1	0.001
Roll/pitch	0.5	0.35	0
Yaw	0.05	0	0

Figure 4.2: The PD-controller parameters used in the Tum ardrone package

the drone to stear to a certain coordinate). The standard PID controller provided by it will be used for the basic navigational tasks in this work. The PID parameters used in it can be seen in the figure 4.2.

Additionally, the package offers an easy way to use PTAM, in order to position oneself and navigate solely using the camera, in addition with the IMU. The package provides an easy way of operating the two, using a GUI. The GUI features an interactive way of sending commands, a list of the sent commands, and state information about the drone. This software package is the implementation of the three papers Jakob Engel (2014), Jakob Engel (2012a) and Jakob Engel (2012b).

4.2 Aided transportation using busses

As explained in the introduction .2.16, it was wanted to make the drone try to use public communication to get to its destination given that it is too far away for the drone to make it on its own, potentially getting recharged on its way as well. For this to work it needs to both be able to recognize the bus, know how to fly well towards it, as well as be able to land on top of it.

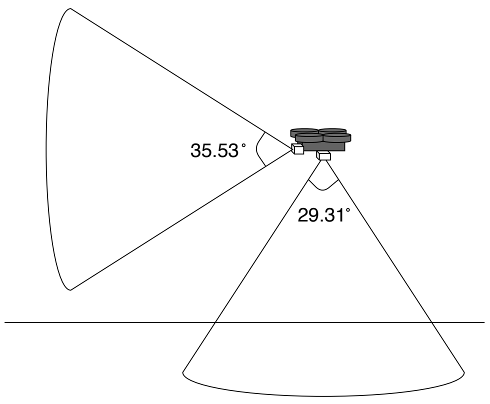


Figure 4.3: The field of view of the ar parrot drones cameras, illustrated on the YZ-plane.

The ardrone continuously knows the remaining capacity of its battery. Using this and how far away the goal is it can make a judgement about whether or not it can travel it on its own or if it needs to make use of a bus. Given that it knows the location of a road, it could travel towards it and use it for a certain distance of the journey. The drones ability to recognize roads would be a good addition to this framework, but is beyond the scope of this paper to attempt to implement. An assumption will be made that the road is located along the y-axis at x=20. Having arrived alongside the road the drone will enter a state where it can look for a bus.

4.3 Identifying a bus

In order to simulate to the interaction between the AtB bus and the drone, a 3d modell of the 'Man bus model Lion City LE'(**?**) was modified to look like an AtB bus, and imported into the Gazebo simulation (see figure 4.4).

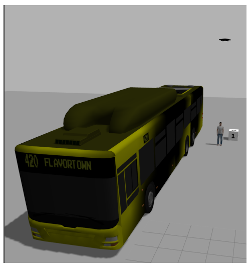
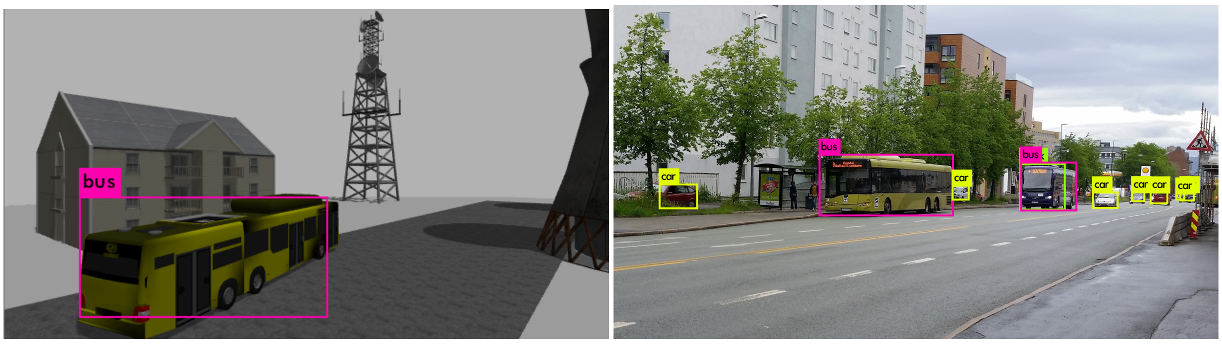


Figure 4.4: 3D model of the classic AtB bus, in a Gazebo simulation.

As one can observe on .2.16, Yolo can identify both the bus in the 3D model and the real world AtB bus. The 3D model has also been modified to have a landing pad on top of it. Further modifications regarding the landing platform will be discussed further down in this section.



(a) The 3D model in the gazebo environment, detected with an 80% certainty.

(b) A real AtB bus, detected with a 100% certainty

Figure 4.5: Busses being detected by the Yolo object detector.

Travelling alongside the road a safe height should be maintained to avoid as vehicles etc. An optimal height could also depend on the field of view (angle spanned by a camera) of the cameras. The field of view can be calculated using:

$$FOV_x = 2\arctan(\frac{w}{2f_x}) \quad and \quad FOV_y = 2\arctan(\frac{h}{2f_x}), \tag{4.1}$$

with h and w being respectively the height and width of the image(Bradski (2000)). Using this with the calibration matrices of the parrot ardrone cameras (.1) we get that:

$$FOV_{front} : \qquad Horizontal =\sim 59.31°, \; Vertical =\sim 35.53° \tag{4.2}$$

$$FOV_{bottom} : \qquad Horizontal =\sim 49.95°, \; Vertical =\sim 29.31° \tag{4.3}$$

These estimations are also illustrated on **??**. Considering that the width of the road in the example gazebo environment is 22m, the drone could fly at a height of about 9.3m (or more) above the center of the road and keep the entire width of the road within its field of view, making the front camera potentially unnecessary for the bus localization. However, as the drone would potentially in a real world application additionally have need for a bus localization from a far, this solution was not opted for. In order to test the applicability of bus localization from a far, the bottom camera will not play a role in the localization of the bus itself. Instead it was decided that the drone travels alongside the road at a "safe height" of 6 meters, and continues until it is the closest to the goal while still following the road. For the example in question, that would be the point ($x = 20, y = 50$). As the drone travels along the road, the detection of a bus should make the drone change its path.

The identification of the bus and its position could be done using classical computer vision methods. This being object detection, a pipeline similar to that described in **??** could be used. As the object that is wanted to track is additionally known, a series of features could be created from images of the bus. Then much like the loop closing scheme used by ORB-SLAM described in .2.7, if enough inliers is found, the bus is detected. This could be used in cooperation with color filtering, ie. only starting the potential matching thread when the color is present.

As this method of object recognition is concerned with the features found on an object, the point mentioned in section .2.3 needs to be considered again. While CNN recognize an object using all of the pixels in the area in question, the positive and negative sides of using the classical framework as opposed to a CNN is similar to the discussion of feature based versus photometric error. If the object in question is textureless and it has few or no features, it can not be tracked or identified using classical feauture based methods (Fei and Soatto (2018)). A CNN could however be trained to identify it regardless. As the classic AtB bus is a distinctive green color with a lot of contrast, in this case it will not be problem.

Using a pipeline of classical computer vision methods would be a more reliant modern approach. However, as it was desired to involve a CNN as well due to its relation to the field of computer vision, it was implemented instead. For further work with this, an object recognition pipeline similar to what described above can be implemented. More about the positive sides of using classical methods can be read at 5.2. More about recommendations about the techniques that should have been opted for in further work with this can be read in section 5.

Running on the gazebo simulation along side other pieces of software, YOLOv3 ran at a speed of only 0.3 FPS. According to Joseph Redmon (2018), it can potentially give much higher framerates on the right system, however with the system specification described in 6, only 0.3 FPS was possible. Reliance on Yolo alone for navigation was then impossible, as the drone could have travelled far past the bus before it gets the output from Yolo using this framerate. Given the assumption that the identification of the AtB bus from afar is only possible using a ML object detector like Yolo, it was decided to find a way to know to wait for the slow output of Yolo.

Color filtering

Seperating the color of the distinctive green AtB busses can be done using the opencv application for fitlering rgb color. After inspection of the color scheme of the busses, as well as the effects of different lighting, the following ranges of rgb color was arrived at:

- Red: 0-120
- Green: 0-255
- Blue: 0-0

Using these ranges of color, the entire bus can be separated from an image in the gazebo environment. These ranges can be considered large, and could be narrowed in to get a result that is more robust to other objects in a similar shade of green. Given that green is a color one otherwise find a lot of a in nature, and along the streets of Trondheim, simply separating out the color green from an image and identifying it is not sufficient to recognize the bus. However, taken into account testing environment in Gazebo described in .2.16 this could provide a sufficient result.

Finding the amount of green on the frame and comparing it to a certain threshold. Upon this being the case, the drone can stop and Yolo start.

Detecting

Given that the recognition of the bus using Yolo is trusted, one can then wait for Yolo and its output after first recognizing the right shade of green being present. Some additions to this method was considered:

- In order to be more certain of the bus being of the correct type, instead of simply another bus next to a patch of grass in the correct shade, it was suggested to check the IoU between the bounding box outputted by Yolo and the green area outputted by the filter.

- The IoU of the filtered image and the bounding box outputted by Yolo could be used to refine the RGB filtering ranges. This could make the tracking easier. Considering however the often imprecise bounding box outputted by Yolo, this would be difficult to implement well.

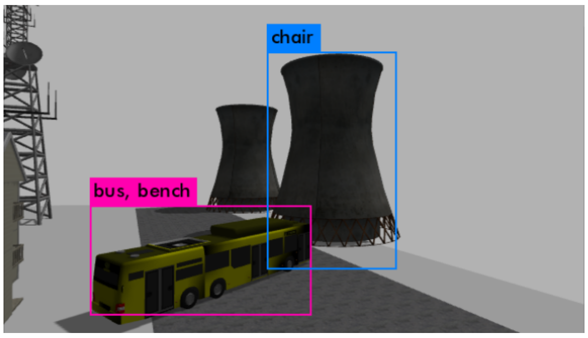


Figure 4.6: The bounding boxes outputted by Yolo. Note the several bad predictions, namely the bus potentially being a bench, and the silo potentially being a chair.

Given that the resulting gains from these suggestions would also mainly show themselves in the real world application of this (with other green elements besides the bus), and not in the gazebo simulation, it was not added. If Yolo predicts the position of a bus, with a probability beyond a certain threshold the bus is considered detected. Otherwise, if Yolo predicts nothing within a certain timeframe the drone moves on.

4.4 Converging on the position of the bus

As the potential detection of the bus happens with the slow frequency of 0.3 FPS, it is difficult to use it for the guidance of the drone towards the bus. Naturally a reference signal has to be updated with a high frequency, but even creating such as a signal using the slowly updated Yolo detections has its problems. In addition to some inaccurate bounding boxes and slow detections, the factor of the associated delay can be considered a separate issue. As the jaw orientation and position of the drones drifts, the delay in the signal result in the detecton being aditonally faulty. Additionally some drift in the position and orientation would make the detection especially faulty. Considering Yolo as the object detector used, two options to provide a continuous reference signal were considered:

- Using the bounding box outputted by Yolo, the center of mass can be estimated to be in its center. Using this the error between the middle of the frame and the center of mass of the bus can be calculated. This results in a discrepency in pixels, that can be converted into its corresponding discrepancy in degrees using equation 4.1. Using the control of the jaw angle of the drone, taking advantage of the accurate information the drone has of its orientation using an IMU, the estimated orientation could be converged on.

- Using a short term tracker, or a method of optical flow several points will be created that can keep a continous track of an object.

The latter option was deemed a more robust option and was therefore chosen. This renders the use of Yolo mainly an aid for the detection step, an nothing relating to the position or convergence on the bus.

Lukas-Kanade Method for optical flow

Generating a sparse pointcloud of 2D tracked points using the Lukas Kanade method, the points outside and inside the filtered green area of the image can be distinguished. Finding the average velocity and direction of these points, one can then estimate the 2D movement of the previously detected bus, by taking the average of all the pointwise estimations.

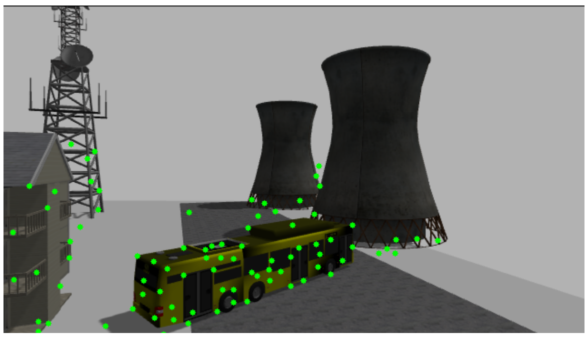


Figure 4.7: LK method and its 2D tracked points. Due to the contrast on the bus, several points are placed there. The points within the bounding box described above is used.

Using the position and velocity of the point-cloud, the drone is regulated to keep the point-cloud in the middle of its FOV, and additionally move forward. However, the LK method does not necessarily placed its tracked points evenly on the bus, as there are more features on some places than others. This result in a centroid esimation that will have a certain bias or error. How detrimental this is function of the robustness of any landing platform tracking (see the next subsection 4.5 for more information).

Regulating drone position with a PID controller can give smooth convergence to a certain position, but a drawback is weakness to noisy signals. Noisy feedback signals will result in noisy inputs, meaning instantaneous large inputs for example, a huge strain on drone motors and potentially making flight unstable. This can also be the case with sudden large inputs in general. While the resulting jerk in motion can still create a feasible controller for an unmanned drone, it proved to be especially detrimental in this case as the quality of the track is dependent on the camera moving smoothly. A jerk in the position of the drone result in the drone losing track. One way to mend this is by using a low-pass filter to create a reference model (Fossen (2014), chapter 10). The reference model will construct a smooth reference input signal for the drone to follow while tracking the bus position, smoothing the reference and the path and potentially reducing wear as well.

A common reference model is made with a simple low pass filter. Giving it an order to realistically to provide a model close to the system dynamics is also common **??**. Using the python SciPy library (Jones et al. (2001–)), a discrete FIR (Infinite Impulse Response) low-pass filter was made. Through trial and error a FIR filter of order 80 with a cutoff frequency of 0.004 provided the gentle filtering required to provide a smooth signal to the drone, that result in minimal jerk of its motion. This control scheme is illustrated on figure **??**. The resulting smoothed response of this filtering can be seen on figure 4.8.

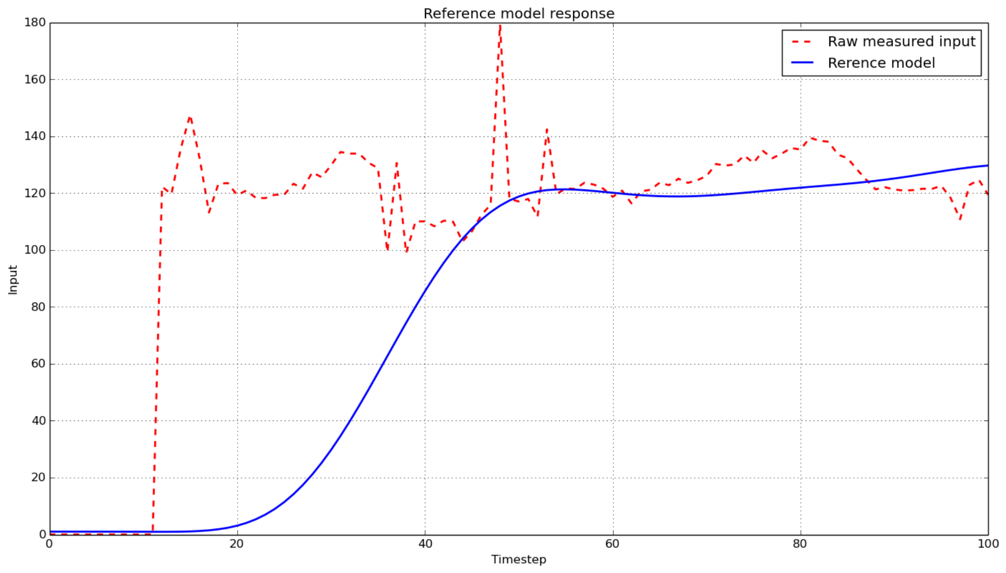


Figure 4.9: The response of a smoothing filter creating a reference model.

The controller scheme illustrated in figure **??**, makes the drone head straight for the horizontal position of the bus, no matter its distance away. This also refered to as a Line Of Sight Guidance (Fossen (2014), chapter 10). There are other methods of target tracking, however they would need more information such as the distance between the drone and the target. As the angle of the bus itself is not known, estimating the distance to the bus using the size or the shape of the bounding box is arguably difficult. If the distance was estimated, the drone could travel along the road for longer. Considering the small scale example taken into account in this work this will not make any difference either way here. Otherwise this could be discussed in depth.

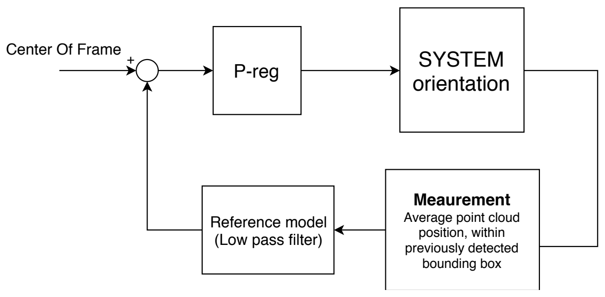

Center Of Frame
+
P-reg
SYSTEM orientation
Meaurement
Average point cloud position, within previously detected bounding box
Reference model (Low pass filter)

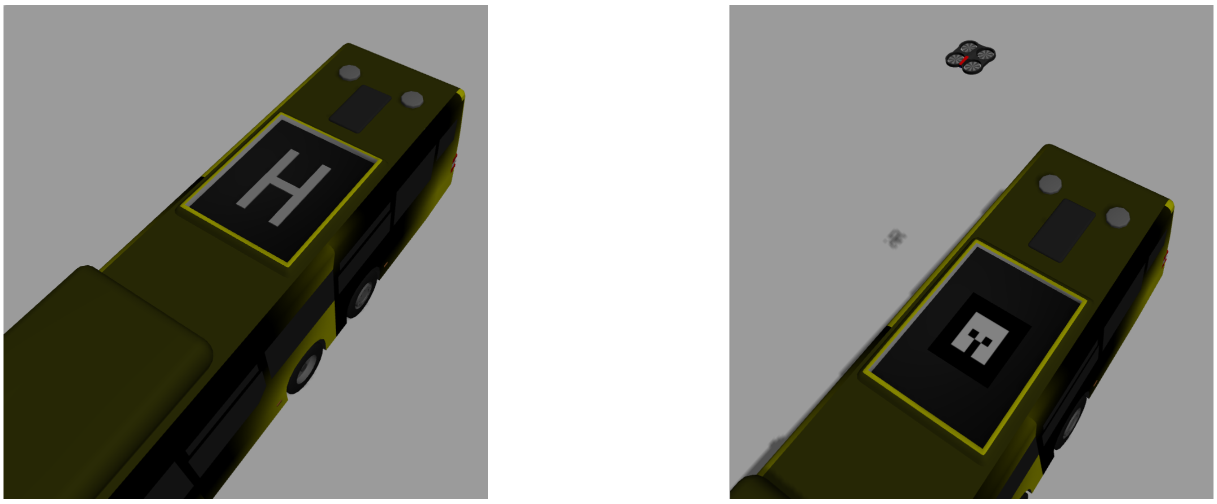
Figure 4.8: Control loop for controlling the yaw and x,y position for the

Results of LK-flow aided LOS guidance towards bus		
Angle of bus	Time until detection	Error along y-axis
-10	>3min	F
-10	5 sec	1 m
-5	33 sec	1 m
0	1m 15sec	2.15 m
10	>3min	F
20	1m 9sec	5 m
45	1m 20sec	6 m
45	20sec	3 m
60	6 sec	19.5
90	>3min	F
90	18sec	3 m

Testing this method to detect and track the bus from several angles give the results that is shown on 4.4. For a given angle of the bus in relation to the direction of the road, the time until Yolo detects the bus and error in the x-axis position of the drone in comparison to the bus center of mass is noted. At the given slow framerate of 0.3 seconds, 3 minutes provide 54 predictions. With no correct predictions within these 3 minutes and 54 predictions, the score for this angle is written down as an F.

It was assumed that a certain correlation would emerge, showing less error with a larger angle. This could be because a a pose of the bus showing more of its features could also make for an easier prediction. When it comes to the object detection, the results does not show much correlation between the angle of the bus and these results at all however. Note the case then -10 degrees on the bus was tested twice, with very different results. Through trial and error, no other techniques seemed to influence when Yolo gave a correct prediction. However, when a detection was made, the tracking aided by the Lukas-Kanade point-cloud allways came sufficiently close to the bus. More about what would be considered sufficient is described in the next section.

4.5 Landing on the roof



(a) Platform with the letter H

(b) Platform with 'Aruco marker'

Figure 4.10: Landing platforms considered

The drone would approach the bus from a large height and need to be able to recognize the bus from there, and know how to land on it safely. Considering this, the bus could use a distinctive landing platform. The first version of the bus had a large distinctive H on the top of it, seeming like an obvious choice. A neural network could be trained to recognize letters and symbols, and identify the letter H on top of the bus model and output its general position. This solution is however neither robust, reliant(5.2) nor fast enough for the use case of landing on a detected platform, as the bounding boxes of image recognition neural networks are often jittery. Instead it was decided to place an fiducial marker on top of the platform.

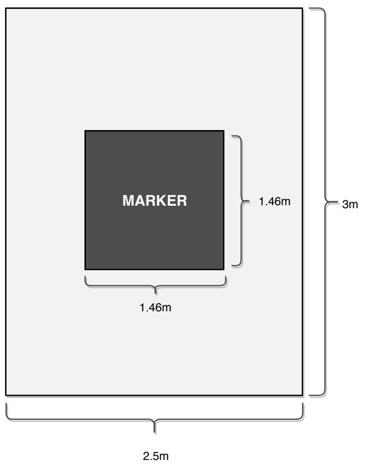


Figure 4.11: Bus landing platform and its size.

'Aruco markers' are examples of fiducial markers, that also OpenCv (Bradski (2000)) has an easy to use framework for detecting (for more about this framework, see 2.3.4). They can can additionally embed information. Placing this on the bus roof allows the drone to easily get the markers position and orientation, as well extra data (which can be used for information such as the bus route). With the drone retrieving the information by flying above the bus and using its bottom camera, it does not have to fly around the bus in order to find the route number on the front or the back of it which would otherwise be a hard and cumbersome task. For this example, only a standard Aruco marker was chosen.

The landing platform was created to be very large, for it to be an easier time landing, but also so that smaller errors in its landing could be observed and noted, without the drone tumbling off the side. The scale of the landing platform can be seen in figure 4.11. The drone would aim to land on the center of the marker, which it is large enough for (the dimensions of the ardrone, as well as more information about it can be studied in .1). Considering the real world applications, such as that the bus could need room for more than one drone, and drone could need to be recharged, this accuracy in landing (landing in the center of the marker) could arguably considered the least required. In addition to landing within the surface area of the marker, the drone should also be orientated towards the front.

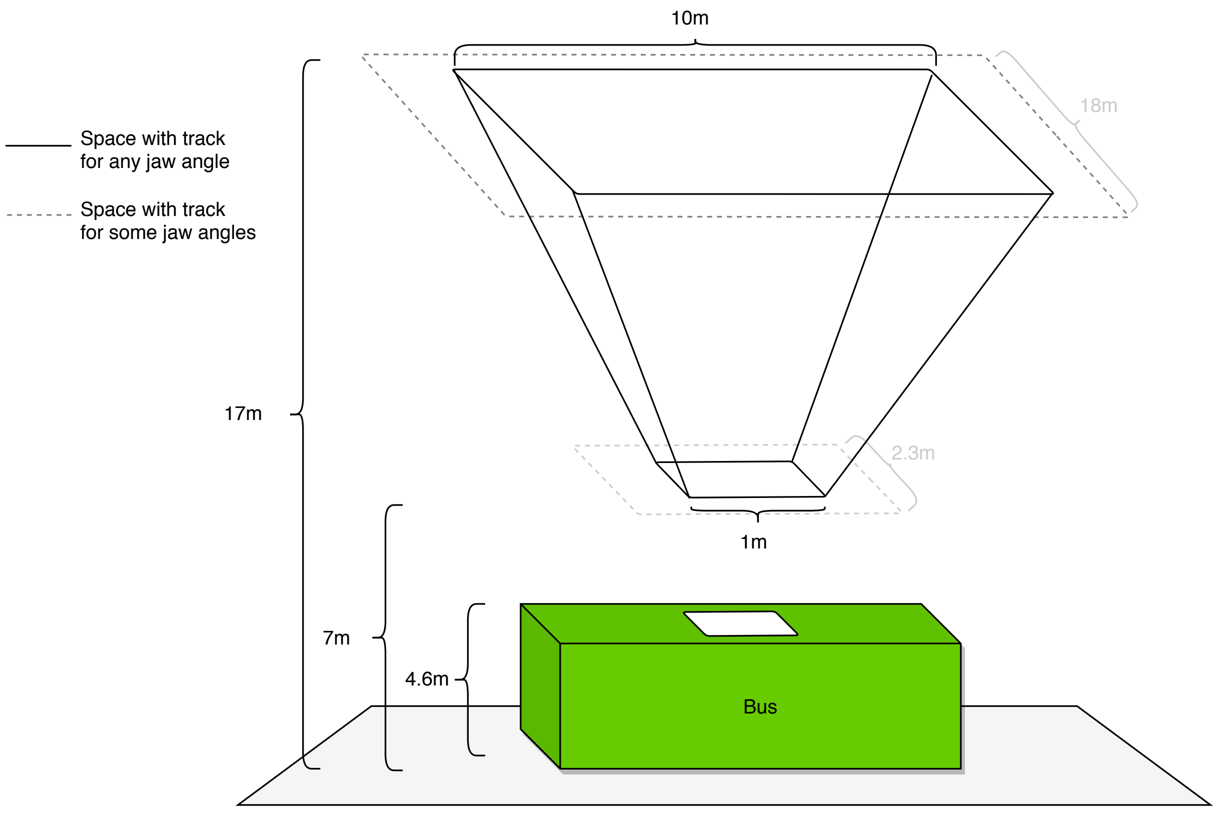

10m
18m
Space with track for any jaw angle
Space with track for some jaw angles
17m
2.3m
1m
7m
4.6m
Bus

Figure 4.12: The prism in which the bottom camera of the drone can track the Aruco marker.

The FOV of the bottom camera and intricacies of the Aruco tracker limits the space from which the drone can observe the marker, and that the tracker is being recognized. Through trial and error, the 3D prism describing this area on figure 4.12 was arrived at. Above a height of 10.2m, the marker can not be tracked anymore, and it also can not maintain a track when it is closer to the bus than 0.5m, as can be observed on the illustration. The rectangular aspect ratio of the image allows the drone to keep a track in a slightly larger area when oriented perpendicular to the platform, which is indicated on the figure with a dotted line.

Initially, a simple script using the wanted orientation and position to the bus was fed into the TUM ardrone controller. As described in section 4.2, this software package takes an inputted waypoint, as opposed to a continuously changing reference. In order to refine this estimation on the way down, the waypoint was reset often towards the platform. Using this it was found that one essentially get a sporadic drift in the estimation of the position of the landing platform, as the position is not continusly being updated. Testing the accuracy in landing that could be achieved using this framework, several random positions within the prism of good tracking was tested. In order to test the landing a point system for grading the landings where used. If drone lands on the marker itself, with the right orientation, it scores 100%. If it lands outside the entire platform it scores a 0%, and for rest of the platform the score is given by $S = |100 - 50|$(Distance away from center).

This understandably did not give robust results. The tum ardrone framework allows only to set a staticly placed waypoint, making it very voulnarable to lack of precision in the track of the marker. For randomly generated positions and jaw-angles within the prism the landings scored an average of about 57%, with only 60% landing perfectly. These positions were also from anywhere within the prism. When the tracking and landing procedure starts its more likely on the rim of the prism, which would alas make the landing even harder. On randomly generated positions and jaw angles on the rim of the prism, the average landing score falls to about 32%, with 28% landing perfectly.

Considering a the shortcomings of this framework a new one was attempted. The shortcomings were mainly three-fold: tracking lost with difficult orientations, precision in the descent and the final landing maneuvering. First of all, in order to make the landing more precise, a simple P-controller(.3.4) was created (the parameters can be seen at 4.13).

P controller parameters		
XY-motion	Z-motion	Spin
0.2	0.3	0.05/J(x)

Figure 4.13: Parameters for the simple P-controller used in the landing. Note the costfunction $J(x)$ in the parameter for the spin input

Attempting to land when the marker is tracked on the far edges of the frame (indicated by the dotted line on 4.12) can be a problem. With a simple control loop simply making the drone converge on the position below, it can make the drone reorient itself to a jaw-angle with which it has no track of the landing platform. This situation is also shown in 4.14. To mend this, one could use the estimated transformation between the drone and the platform to compare it with the prism in which it would have a track with every jaw angle. Instead of this however, an easier solution was chosen, where height and orientation error was taken into account. First of all, with height discrepancies of more than 7, no changes in jaw angle is performed. Secondly, a costfunction is made to lessen the input the jaw input for certain discrepancies in the drones relative x-direction. The function ended up with was $J(\Delta x) = \frac{1}{10|\Delta x|}$ for $\Delta x > 1.3$, otherwise $J(\Delta x) = 1$. This costfunction manifested itself as a factor in the relative orientation setpoint. This had the wanted effect no diminishing all jaw motion before the bus landing platform is far enough within the frame for the track to maintained. Finally, the final maneuvering made within 0-1m above the platform is made without a track. Any real maneuvering within this range cause inaccuracies. Instead of this, a land command using 4.1.1 was sent with a height discrepancy of 3 meters, which provided a more presice landing. As the automated landing command from 4.1.1 on the ardrone provide an arguably slow landing, the horizontal position of the drone was maintained during it.

Using the additions above several randomly generated points within and on the edge of the prism was tested. In every single case, as long as a track was initially found, the drone lands with a 100% accuracy.

It was additionally considered to add a reference model much like in section 4.4, considering potential gains in the quality of tracking with a smoother path. However it was decided that this was unnecessary. For a faster convergence to the platform a K_d and K_i factor could have been added to the regulator. As the simple use of a P-regulator described above was sufficient for a successful landing, further additions where not worked on.

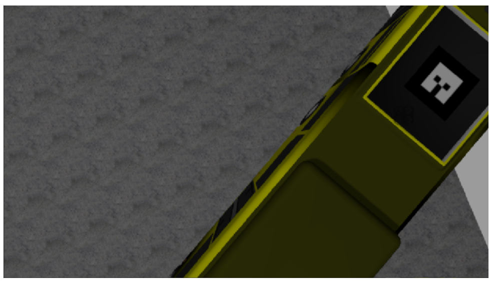


Figure 4.14: A POV of the landing platform from the drone above. The landing platform can be seen at the top right corner, oriented at about 135 degrees. Note that any change in orientation on this position would make the drone lose the tracking.

4.6 Total algorithm

In figure 4.15 the entire framework and algorithm described in the two subsections above is illustrated. The two frameworks work well together, but the the detection part is lacking robustness and accuracy in tracking and detection. However, due to the close proximity and simplicity of the example used and the very forgiving size of the prism in which accurate landing can be achieved, the total algorithm show decent results in the following ways:

- All the different epplications and plugins work well together, and can be started from a signel launch file.

- It is robust in that the drone can start from any position, and end at the goal unless it spots something green above a certain threshold.

- Given that Yolo provides a detection it is pretty robust in landing, no matter the angle of the bus.

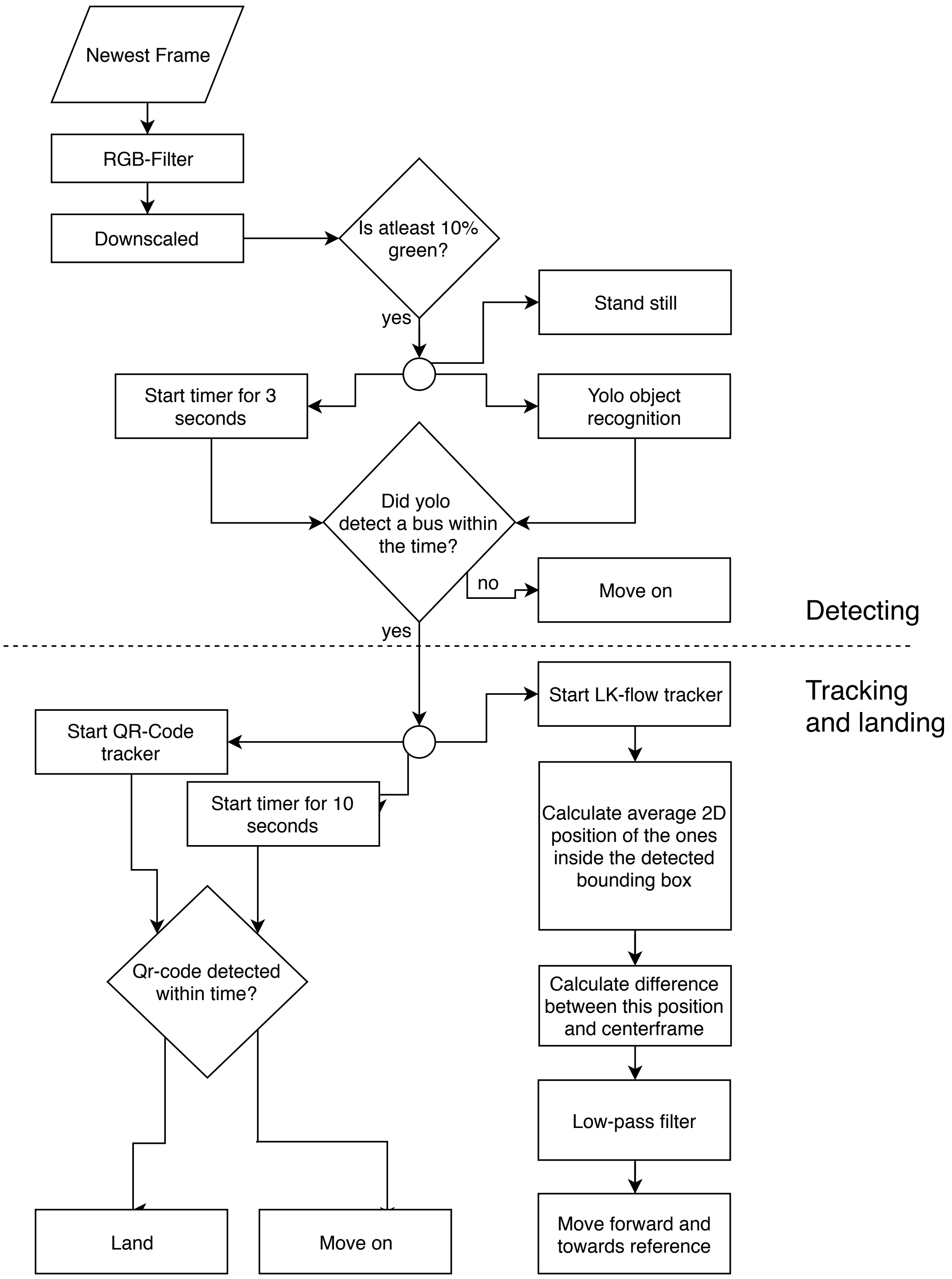

Newest Frame
RGB-Filter
Downscaled
Is atleast 10% green?
yes
Stand still
Start timer for 3 seconds
Yolo object recognition
Did yolo detect a bus within the time?
no
Move on
Detecting
yes
Tracking and landing
Start LK-flow tracker
Start QR-Code tracker
Start timer for 10 seconds
Calculate average 2D position of the ones inside the detected bounding box
Qr-code detected within time?
Calculate difference between this position and centerframe
Low-pass filter
Land
Move on
Move forward and towards reference

Figure 4.15: Flowchart for the tracking and landing of the drone

However, due to extremely slow speed of Yolo, no limit to its detection time could/should be implemented. Because of this, the additional "Move on" situations in the flowchart describing the algorithm were not implemented either.

It was wanted to start the different applications when the algorithm found that they were needed, however it turned out that the version of ROS used (Indigo) had no support for this. This makes little difference, but should be changed if implemented in another ROS version due to the less computational load.

4.6.1 The code

In the appendix .3.4 the entire code used is added. A certain knowledge about the ROS API can be necessary to understand it all. The .launch file is what can be considered a main file, and starts all the necessary processes.

Plugins.py checks all the plugins and computer vision applications, and whether or not detections and thresholds is passed. It keeps control over this and its place in the algorithm using the array "algoOrder".

mainInteractor.py use the information provided by plugins.py and the measurement devices to move the drone.

Chapter 5

Discussion

In the section below, what can be determined using the results is discussed. Additionally, the additions that can be made to the autonomous vehicle control using these techniques is considered.

5.1 Computational expenses

As mentioned in 6, the simulations where done on an i7 CPU, without a GPU. Running all the previously mentioned applications at the same time made the simulation run in a slower speed, and potentially led to worse result than could be achieved with eg. a GPU.

A drone however would presumably not be equipped with this much processing power on board, neither as much as described in 6 nor more. Creating something that was computatioanlly lightweight in order to run on a along side other applications were not among the goals in this work, but is real concern. The computational power required is not considered or discussed in this work, but its compatability with a machine with the specs mentioned in 6, is arguably concerning.

Without getting deep into this problem, some directions one could take in mending this problem could be:

- Fitting the drone with a stereo camera, with high FOV (eg. what is known as a fish eye lens). This could help the drone in maintaining track for longer. With an easier tracking, the system could potentially suffice with a VO system alone, such as the DSO, which should surprisingly good results. The use of VSLAM is arguably the highest source of computational load in this framework.

- The use of neural networks for computer vision is more computationally heavy than other classical methods in computer vision. Using a different algorithm for the recognition of the bus could potentially aid in the computational load as well.
- Due to limitations of ROS indigo, several pieces of software have to run simultaneously. Using a newer version of ROS would allow the software to launch different software such as eg. the VO only when needed.
- If the computational load is still a problem, the work could potentially done offsite instead, on a server. However, this would require the drone to get its navigational information in another way, such as a internet connection (which is another problem).

5.2 Reliance on AI in navigation

As explained in **??** and several other texts, neural networks are universal approximators. However, their performance is completely reliant on the dataset and how well it is trained on it. If the real world problem is not represented sufficiently in the dataset, the neural network wont give a good performance. As mentioned in .2.16, the typical way to compare neural networks is using their mAP on a given dataset. This doesn't necessitate a real world performance. The neural network is limited to a number of factors, such as:

- The dataset, and how well it shows the real world application the neural network needs to mimic.
- How well the real world application of the drone will be similar to the dataset. A real world application is continuously changing and can result in situations the network is not trained for.
- How well the network is trained. It could for instance be overfitted, unable to generalize to new real worls situations.

As opposed to classical methods in computer vision, you can not prove the performance of the neural network, which is why it is often not opted for in the field of vehicle navigation. When it comes to vehicle recongiiton, thera are options in the field of classical computer vision.

In this work, it was mainly chosen due to its ease of use, stable results in an environment with limited detail, and the fun of it (?). However, for further work into this concept, its reliance on neural networks should be avoided. However, it can be argued that as the ML is just a step in the algorithm, before the drone afterwards uses a seperate classical method in CV for recognizing the aruco marker. In other words the system has arguably no reliance on ML as the lack of a aruco marker on top of the vehicle could allow the drone to move on, rendering no error.

5.3 The result

In this section, the results arrived at in the previous chapter will be summarized and generalized.

5.3.1 Object avoidance

Using visual techniques for object avoidance works, in that it is possible. Point clouds and occupancy maps can be made live as the drone moves around, however the process of combining these and path planning in order to create a route that avoids crashing into objects can be computationally heavy.

The framework in Hao-Chih (2012) provided a working result, using both the IMU, PTAM and LSD fused together with a least square algorithm to create a good estimate of the scale of the scene. This arguably proved to be too much of a computational load. However, other more simple frameworks could give a computationally lighter result. An example would be the use of a stereo camera along with DSO.

5.3.2 Bus recognition, tracking and convergence

With the current framework of detecting the bus, this is a slow process. The detection of the bus is reliant on a very slow neural network, that is not trained on AtB busses specifically, but the VOC2012 dataset that features busses in general. Additionally, the use of neural networks for vehicle control is not considered good practice. However, it can be argued that this use of neural network does not make it reliant on it. In the case of a bad positive, the drone would continue along the road when it fails to see the Aruco marker of a roof a bus within a given time. In any case, there are other approaches could ahve been used for the recognition of the bus

5.3.3 Bus platform landing

The landing performance of the drone using a fiducial marker to provide track, show a perfect success rate of a 100%. The bus landing platform was made large so that all discrepancies in landing was observable easily, and that some results can be made even if the drone misses the exact mark. However, as the drone managed to land accurately, further work showing the exact accuracy obtainable should be made (for more about this see **??**).

5.4 VSLAM and GPS

As mentioned in .2.16, modern uses of VSLAM combine it with the measurements from IMU and GPS in factor graphs, creating a sensor fusion. This combined estimate should provide a more accurate estimate of the pose over time. As mentioned in .2.16, Rones2019 the disturbances in the position estimation at .2.16 was due to buildings creating an urban canyon. The results shown in .2.16 display that the pose estimation using a VSLAM algorithm perform well in the situations where the GPS fails - situations affected by urban canyons. As the package delivery drone would have to navigate in urban areas, urban canyons is a problem it will often be affected by. Additionally, with the dense distribution of personal addresses in urban areas (and potentially then points the drone must reach), accuracy in positioning is arguably vital.

The modern framework of using VSLAM in cooperation with GPS and IMU with factor graphs can then potentially be a good option for the setting of a drone navigating in urban areas.

5.4.1 Expanding the elevation masking

The modern framework of using factor graphs to combine the estimations of VSLAM and GPS create a modern combined estimation a chage in pose, but running VSLAM can be a computatonally heavy feat. However, considering that the problem of urban canyons relate to the unknown distance of the multipath signal, the computationally heavy bundle adjustments may not by needed. Given a simple pointcloud created by a stereo camera, one could use this directly to aid the GPS by in tree potential ways:

- Expanding the elevation mask to include the elevation angles covered by these points. However, as only the required minimum of four satellites are guarenteed visible, simply removing more satellite signals from your estimation would have to depend on how many you have available. Letting this factor into the Kalman filter could be a safer option

- Calculating the full length of the multipath signal can also be possible, as the position of the vehicle in question and the position of what it has potentially bounced off of is known.
- The amount of tracked points can also be used to identify the presence of an urban canyon. This would not necessarily helping with the precision of the position estimation, but help with the knowledge of its potential lac of precision.

The standard elevation masking cuts off all satellites situated at a 15 degree elevation angle, as the estimates using their signals are especially untrustworthy. Signals arriving from angles where there are obstructions would most likely be multipath signals and also untrustworthy. Given that we already have an occupancy grid, with information about where some potential obstructions are, one can potentially use this to expand the elevation masking to where one now know there is no clear line of sight.

5.5 Further work

The detection and tracking of the bus which was very dependent on ML object detection was shown to be non-reliant and not robust. Feature tracking could potentially give more robust and accurate result in both tracking and detection of the bus. The process of creating a bus detector and tracker using this would require several features from all sides of the bus being created and stored in a database. Using this AtB bus detector along with others, potentially the ones provided by Willow Garage (????), could potentially be used to create a navigation framework that use more semantic information (such as avoiding vehicles etc.).

The tracking performance of the ardrone was severely limited by its field of view, arguably forcing the choice of LSD instead of DSO which performed very well (see section .2.17). With a drone with a stereo camera with a large field of view, imported into Gazebo this would potentially not be a problem. For further work, stereo DSO (or one of its many extentions and variations) with large field of view cameras, should be tested. This should additionally be computationally lighter.

In order for a computationally light object avoidance, path planning frameworks besides the

RRT (rapidly exploring random tree, **??**) algorithm provided by move-It (section .3.2) should also be tested.

The landing platform was shown to be more than large enough. If this project is pursued any further, how small one can make the landing platforms as well how many drones one can safely fit on top of the bus should be tested. This would involve both the problem of the proximitty of multiple drones potentially landing and taking-off close to each other, as well as the problem of wanting the drones to remain stationary as the bus is accelerating etc. Additionally the drone landing platform should be moved, as it is in its current position replacing a vent that was originally there.

As described earlier, in order for object avoidance to perform well, accurate localization is key. This is especially concerning in the context of urban drones, as the problem of urban canyons (see .2.15) are especially detrimental to GPS data, and can be aided using VSLAM or VO (this is shown in section .2.17 and figure 3). State of the art uses of VSLAM use factor graphs (described in **?**) that can efficiently use both GPS and VSLAM data for accurate localization and mapping.

The need for a road tracking algorithm was realized early on during this work. As the accurate placement of roads in relation to a drones position is not necessarily easily known using eg. a map, a tracker would be very helpful. Using classical methods 2.3.1 such as line trackers, discerning the road should be feasable.

This project was done using ROS Indigo, which was mainly chosen due to LSD-SLAM not being supported by other ROS distributions. The newer distributions of ROS support spontaneous launching of ROS packages and threads depending on other factors. This is not supported by Indigo, and allowing for this would make ROS easier to work with. Indigo is additionally reaching is end of life the summer of 2019.

Rosbags is the common filetype to store messages and data that is incoming and sendt between the threads in ROS, using the ROS topics and ROS services. These files can easily also created and used to simulate a drones performance in the outside world without it being there. Using rosbag files in order to test the drones ability to create a map of the city on a large scale, how well it could track and place real world vehicles and how it would navigate would be interesting to test.

For further work, a test could be made to include the bus moving back and fourth, and with the drone taking off from the bus roof when the bus is close to the goal. However, this test ends with the drone having landed on the bus roof.

Chapter 6

Conclusion

Checking the feasibility of autonomous drone package delivery using busses for long distances was tested using a simple example in the ROS simulation environment Gazebo. This test can be broken up into tree parts: object avoidance and navigation, bus detection for long term tracking, and bus roof landing. The landing can be considered a success, as a simple P-regulator was found to provide a very accurate landing, as long as the control of its descent is limited a certain way described in 4.5. Like the litterature describes, the tracking and detection using ML was shown to be unreliable and inaccurate. However, due to the simplicity of the example taken into account it produced a working result. Detection and tracking using a classical feature based methods seem to be a better option, and should be tested in further work.

What proved to be the most troublesome is the autonomous object avoidance, which proved to be both complicated and potentially computationally heavy. Without knowing exactly what system requirements is needed to make the framework tested to run smoothly, it would arguably be something beyond the specs on the system used for the Gazebo simulation in question, that can be studied on 6, as the software made the simulation run slow. Presumably, this would imply that the hardware needed for this computation is beyond what the package delivery drone could carry. Either simplifications to the object avoidance framework or off-site computations computing could fix this, but the off-site computations introduce the problem off the need for a constant eg. internet connection to the drone.

Bibliography

Bradski, G., 2000. The OpenCV Library. Dr. Dobb's Journal of Software Tools.

Cadena, C., Carlone, L., Carrillo, H., Latif, Y., Scaramuzza, D., Neira, J., Reid, I., Leonard, J., 2016. Past, present, and future of simultaneous localization and mapping: Towards the robust-perception age. IEEE Transactions on Robotics 32 (6), 1309–1332.

Chieh-Chih Wang, Charles Thorpe, M. H. S. T. H. D.-W., 2007. Simultaneous localization, mapping and moving object tracking.
URL https://www.ri.cmu.edu/pub_files/pub4/wang_chieh_chih_2007_1/wang_chieh_chih_2007_1.pdf

Crowe, S., 2019. Researchers back tesla's non-lidar approach to self-driving cars.
URL https://www.therobotreport.com/researchers-back-teslas-non-lidar-approach-to-

Daniele De Gregorio, Tommaso Cavallari, L. D. S., 2018. Skimap++: Real-time mapping and object recognition for robotics. IEEE.
URL http://openaccess.thecvf.com/content_ICCV_2017_workshops/papers/w13/De_Gregorio_SkiMap_Real-Time_Mapping_ICCV_2017_paper.pdf

Elfving, H., September 2018. Heuristics and methods of machine learning control. NTNU.

Engel, J., Koltun, V., Cremers, D., July 2016. Direct sparse odometry. In: arXiv:1607.02565.

et al., D. F., 2017. Vision-based autonomous quadrotor landing on a moving platform. IEEE.
URL https://ieeexplore.ieee.org/document/8088164

et al., G. C., 2004. Visual categorization with bags of keypoints.
URL https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/csurka-eccv-04.pdf

et al, T.-Y. L., ???? Common objects in conctext.
URL http://cocodataset.org/#home

Fei, X., Soatto, S., 2018. Visual-inertial object detection and mapping. ECCV.
URL http://openaccess.thecvf.com/content_ECCV_2018/papers/Xiaohan_Fei_Visual-Inertial_Object_Detection_ECCV_2018_paper.pdf

Fossen, T. I., 2014. Handbook of Marine Craft Hydrodynamics and Motion Control.
URL https://onlinelibrary.wiley.com/doi/book/10.1002/9781119994138

Ha1, J., Sattigeri, R., 2012. Vision-based obstacle avoidance based on monocular slam and image segmentation for uavs.
URL https://pdfs.semanticscholar.org/fc72/cffb4d477864b942a6d8832342f0d2f7d50c.pdf

Hao-Chih, L. . H., 2012. Ardrone indoor navigation.
URL https://github.com/Hypha-ROS/hypharos_ardrone_navigation

Hartley, R., 2000. Multiple view geometry in computer vision.
URL https://www.cambridge.org/core/books/multiple-view-geometry-in-computer-visio
0B6F289C78B2B23F596CAA76D3D43F7A

Hoang, T., 2017. Vision-based target tracking and autonomous landing of a quadrotor on a ground vehicle.
URL http://www-personal.umich.edu/~dpanagou/assets/documents/THoang_ACC17.pdf

Ivan Maurović, Marija Seder, K. L. I. P., 2016. Yudai hasegawa, yasutaka fujimoto. IEEJ.
URL https://www.jstage.jst.go.jp/article/ieejjia/5/3/5_253/_article

Ivan Maurović, Marija Seder, K. L. I. P., 2017. Path planning for active slam based on the d* algorithm with negative edge weights. IEEE.
URL https://ieeexplore.ieee.org/document/7878681

Ivan Maurović, Marija Seder, K. L. I. P., 2018. Demim fethi, abdelkrim nemra, kahina louadj. SAGE.
URL `https://journals.sagepub.com/doi/full/10.1177/1687814017736653`

Jakob Engel, Jurgen Sturm, D. C., 2012a. Accurate figure flying with a quadrocopter using onboard visual and inertial sensing. TUM.
URL `https://github.com/tum-vision/tum_ardrone`

Jakob Engel, Jurgen Sturm, D. C., 2012b. Camera-based navigation of a low-cost quadrocopter. TUM.
URL `https://github.com/tum-vision/tum_ardrone`

Jakob Engel, Jurgen Sturm, D. C., 2014. Scale-aware navigation of a low-cost quadrocopter with a monocular camera. TUM.
URL `https://github.com/tum-vision/tum_ardrone`

Jones, E., Oliphant, T., Peterson, P., et al., 2001–. SciPy: Open source scientific tools for Python. [Online; accessed <today>].
URL `http://www.scipy.org/`

Joseph Redmon, A. F., 2018. Yolov3: An incremental improvement.
URL `https://pjreddie.com/media/files/papers/YOLOv3.pdf`

K. Himri, P. Ridao, N. G. A. P. N. P. R. P., 2018. Semantic slam for an auv using object recognition from point clouds. Elsevier.
URL `https://www.sciencedirect.com/science/article/pii/S2405896318321864`

Koenig, N., Howard, A., 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: ICRA Workshop on Open Source Software.

Laxit Gavshinde, Arun K Singh, K. M. K., 2013. Trajectory planning for monocular slam systems. IEEE.
URL `https://ieeexplore.ieee.org/document/6662820`

Niko Sünderhauf, Feras Dayoub, B. T. R. S. P. C. G. W. B. U. M. M., 2016. Place categorization and semantic mapping on a mobile robot. IEEE International Conference on Robotics and

Automation (ICRA).
URL `https://ieeexplore.ieee.org/abstract/document/7487796`

Niko Sünderhauf, Lachlan Nicholson, M. M., 2019. Quadricslam: Dual quadrics from object detections as landmarks in object-oriented slam. IEEE Robotics and Automation Letters.
URL `https://ieeexplore.ieee.org/document/8440105`

of Simon Fraser University by Mani Monajjemi, A. L., 2012. Autonomy lab - ardrone autonomy.
URL `https://github.com/AutonomyLab/ardrone_autonomy`

Omid Esrafilian, H. D. T., 2017. Autonomous flight and obstacle avoidance of a quadrotor by monocular slam. IEEE.
URL `https://ieeexplore.ieee.org/document/7886853`

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A. Y., 2009. Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software.

Renato F. Salas-Moreno, Richard A. Newcombe, H. S. P. H. J. K. A. J. D., 2013. Slam++: Simultaneous localisation and mapping at the level of objects. IEEE.
URL `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=EF1F5CF26335944E5626B255DBFCBF33?doi=10.1.1.677.2893&rep=rep1&type=pdf`

Rones, , April 2019. Tight, online integration of ins/gnss based on rtklib. NTNU.

Scelizki, R., 2011. Differential Neural Networks for Robust Nonlinear Control.
URL `https://www.springer.com/gp/book/9781848829343`

Shoudong Huang, N.M. Kwok, G. D., 2006. Multi-step look-ahead trajectory planning in slam: Possibility and necessity. IEEE.
URL `https://ieeexplore.ieee.org/abstract/document/1570261`

Simran Brar, Ralph Rabbat, V. R. G. R. A. Y., 2015. Drones for deliveries.
URL `https://scet.berkeley.edu/wp-content/uploads/ConnCarProjectReport-1.pdf`

Somkiat Wangsiripitak, D. W. M., 2009. Avoiding moving outliers in visual slam by tracking moving objects. IEEE.
URL http://www.robots.ox.ac.uk/~lav/Papers/wangsiripitak_murray_icra2009/wangsiripitak_murray_icra2009.pdf

Staffan Ekvall, Patric Jensfelt, D. K., 2006. Integrating active mobile robot object recognition and slam in natural environments.
URL http://www.csc.kth.se/~danik/Papers/ekvallIROS06.pdf

Sudeep Pillai, J. J. L., 2015. Monocular slam supported object recognition.
URL http://people.csail.mit.edu/spillai/projects/vslam-object-recognition/pillai_rss15.pdf

T. Bailey, H. F. D.-W., 2006a. Simultaneous localisation andmapping (slam): Part i. Robotics and Autonomous Systems, 108–117.
URL https://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/readings/Durrant-Whyte_Bailey_SLAM-tutorial-I.pdf

T. Bailey, H. F. D.-W., 2006b. Simultaneous localisation andmapping (slam): Part ii. Robotics and Autonomous Systems, 108–117.
URL https://pdfs.semanticscholar.org/27d4/6db7ed4e96944080052b761c62102f26b23f.pdf

Trung T. Pham, Thanh-Toan Do, N. S. I. R., 2018. Scenecut:joint geometric and object segmentation for indoor scenes. In Proc. of IEEE International Conference on Robotics and Automation (ICRA).
URL https://arxiv.org/pdf/1709.07158.pdf

Tsung-Yi Lin, Priya Goyal, R. G. K. H., 2018. Focal loss for dense object detection. arXiv.
URL https://arxiv.org/pdf/1708.02002.pdf

V Le, Q., Saxena, A., Y Ng, A., 06 2019. Active perception: Interactive manipulation for improving object detection.

Vibhav Vineet, Ondrej Miksik, M. L. M. N. S. G. V. A. P. O. K. D. W. M. S. I. P. P. P. H. S. T., 2018. Incremental dense semantic stereo fusion for large-scale semantic scene reconstruction.

ECCV.
URL http://openaccess.thecvf.com/content_ECCV_2018/papers/Xiaohan_Fei_Visual-Inertial_Object_Detection_ECCV_2018_paper.pdf

Vincent, J., Gartenberg, C., 2019. Here's amazon's new transforming prime air delivery drone.
URL https://www.theverge.com/2019/6/5/18654044/amazon-prime-air-delivery-drone-new-design-safety-transforming-flight-video

Viola, P., Jones, M., 2001. Robust real-time object detection. In: Int. Journal of Computer Vision.

Wilke, J., 2019. A drone program taking flight.
URL https://blog.aboutamazon.com/transportation/a-drone-program-taking-flight

Willow Garage, R. c., ???? Ork: Object recognition kitchen. https://github.com/wg-perception/object_recognition_core.

Yang, N., Wang, R., Stueckler, J., Cremers, D., September 2018. Deep virtual stereo odometry: Leveraging deep depth prediction for monocular direct sparse odometry. In: eccv. <a href="https://arxiv.org/abs/1807.02570" target="$_b lank$" > [$arxiv$] < /a > , < $ahref$ = "/$_m edia/spezial/bib/yang$2018$dvso - supp.pdf$"$target$ = "$_b lank$" > [$supplementary$] < /

Appendix A - Hardware Used

The simulations where done using Ubuntu 14.04 LTS, on a Optiplex 9020 Dell Computer. It the following specifications:

- Memory: 15.6 GB
- Processor: Intel Core i7 @ 3.6 GHz CPU

- Integrated intel graphics

.1 Information about the Ar parrot drone

The version used in this was the ar drone parrot 2. It is a currently discontinued real quadrotor product produced by the company Parrot. Without claiming for certain that these specifications are the same on the real product, the Gazebo realizations have these specifications:
Dimensions: 58.42 x 1.27 x 58.42 cm
Weight: 1.8 kg.

.1.1 Sensors

The drone has an inertail measurement unit (IMU), and a GPS. It has two cameras, on in front and one placed on the bottom pointing down. Both have the resolution 360x640. They have the following camera calibration matrices:

$$K_{front} = \begin{bmatrix} 561.999146 & 0 & 307.433982 \\ 0 & 561.782697 & 190.144373 \\ 0 & 0 & 1 \end{bmatrix}$$

$$K_{bottom} = \begin{bmatrix} 686.994766 & 0 & 329.323208 \\ 0 & 688.195055 & 159.323007 \\ 0 & 0 & 1 \end{bmatrix}$$

For more information, see Jakob Engel (2014).

.2 Explanation of VSLAM

When talking about computer vision, visual odometry and visual SLAM has to be mentioned, as it is not only a state of the art series of techniques, but also a field with continues breakthroughs (T. Bailey (2006a)). It is also a fairly new field, with the first true introduction of the SLAM problem was at the 1999 International Symposium on Robotics Research (ISSR'99) . The continuous breakthroughs can even make it hard to create a detailed overview over the basic innerworkings of it, as well as what is the most recent breakthoughs. With the reader taking that into account, this section will give an overview over how the technique can work, as well as how it does in its 3 most well known and defining algorithms. Essentially all of explanation in this subsection is taken from the book Multi view geometry (Hartley (2000)).

.2.1 Simultaneous Localization and Mapping

Simultaneous Localization and Mapping in general is a method of using a distance measuring sensor in order to both create a map of an area, as well as placing oneself in the area in question. This process would also potentially involve the movement of the subject with the sensor, rendering this process not completely not straight forward, as a measured change in a measurement could both potentially mean a change in the position of the measurer, as well as the points in the surrounding area that is being mapped. SLAM takes into account both of these possibles and solves this seemingly impossible problem using optimization and error minimization.

A SLAM algorithm often consists of 3 independent threads: a short term tracking thread, a long term tracking thread which both feed into a mapping thread. All the different threads deal with their own responsibilities etc. The short term tracking threads notices the immediate change in a measurement, and uses this to generate pose estimations. The long term tracking thread creates what are called loop closures, that is comparing poses and positions of points to figure out whether it can combine previous estimations update its current. The mapping thread receives the data of changes of poses and points, and stores this all in a map.

This is an algorithm that can work with any distance measurement sensor, like a laser sensor or a sonar. However, the sensors that are usually used are RGB-cameras, RGB-D cameras (ones that see depth using an additional sensor), stereo cameras (two cameras about 10cm apart, so that the depth of an image can be deduced) and LIDAR ('Light Detecting and Ranging', a popular distance sensor using laser pulses). The use of cameras for this instead of depth measurement sensors is however significantly more difficult. SLAM is often used on autonomous vehicles, in which LIDAR is often used. Using a LIDAR for navigation however has some downsides besides it being expensive and large (Fei and Soatto (2018) and Vibhav Vineet (2018)). Tesla decided against using LIDAR on their autonomous vehicles and instead use cameras, for this reason (Crowe (2019)). This work will continue its focus on Visual SLAM using RGB cameras.

Geometric camera model

In section 2 the calculations related to a cameramodel was briefly explained. Scelizki (2011) chapter 1, takes these further ending with the following equations:

$$\boldsymbol{\pi}_{perspective}(x) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \boldsymbol{K}\frac{1}{z}\boldsymbol{x} = \begin{bmatrix} f_u \frac{x}{z} + c_u \\ f_v \frac{y}{z} + c_v \end{bmatrix}$$

$$\boldsymbol{\pi}^{-1}_{perspective}(\boldsymbol{u}, d) = \frac{1}{d}\boldsymbol{K}^{-1} \begin{bmatrix} \boldsymbol{u} \\ 1 \end{bmatrix} = \frac{1}{d} \begin{bmatrix} \frac{u-c_u}{f_u} \\ \frac{v-c_v}{f_v} \\ 1 \end{bmatrix}$$

Using this, one can express the re-projection of a 3D point into its corresponding one on a 2D image. This is done using the matrix K (expressing the innate camera information) and the transformation $T_c w$ of its pose.

$$\boldsymbol{x^c} = \boldsymbol{R_{cw}} \boldsymbol{x^w} + \boldsymbol{t^c_{cw}} \tag{1}$$

.2.2 Short term tracking

Even though we have the ability to now know where a point can be found in the 3D space, it takes a lot in order to map and track ones own position at the same time. Short term tracking is the first step to doing visual slam.

Two-view geometry

Considering the geometry, if a change in pose of the camera was correctly estimated, one can calculate a 3D pose transformation between these two. If the frame from the first pose provided

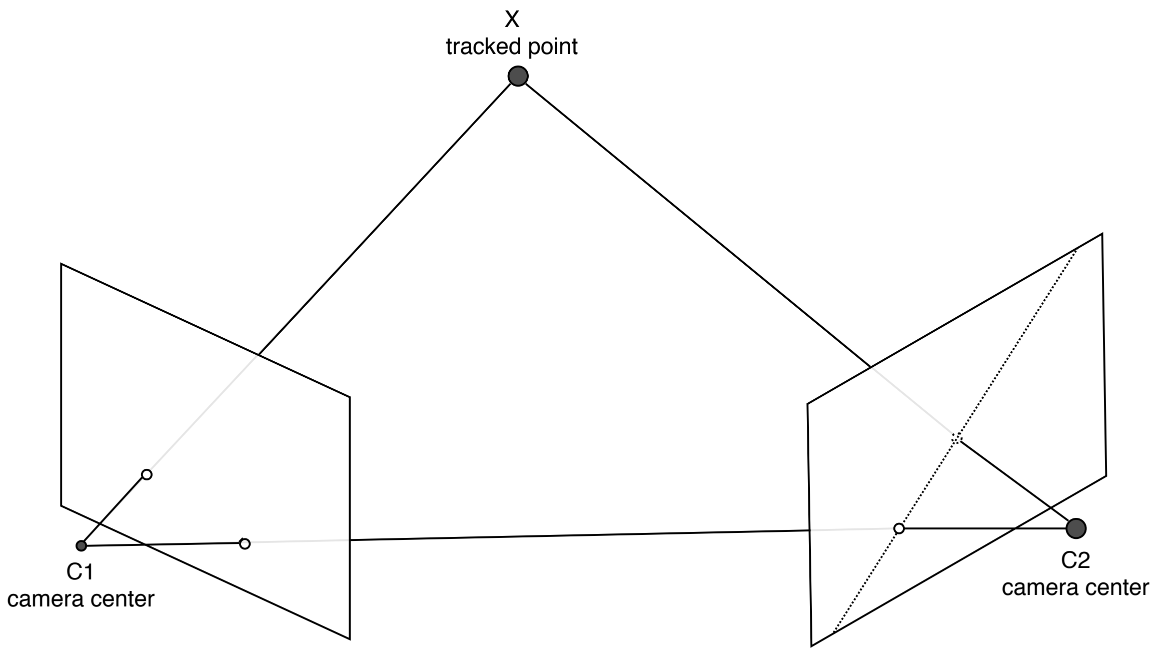


Figure 1: The epipolar geometry

a 2D points that stood out, then an 'epipolar plane' is created. The vector from camera 1 to camera 2 is estimated (known as the baseline), and so is the vector from camera 1 to the tracked 3D point. These two vectors span the epipolar plane. This plane now intersects the image plane of camera 2, with a line on which the point in question can be found (known as the epipolar line). This is called the epipolar constraint. This constraint is expressed using the essential matrix, and the fundamental matrix, describing the change in pose and the epipolar line, respectively (Scelizki (2011) p. 471-505).

If the point in question was tracked, and found on the second point of view, then one can triangulate the position of this 3D point. This can also be used to estimate the position of the camera, given that the points are stationary. As the pose of a camera is given 6 coordinates, not many points are needed to track the movement of the camera (only 5 points are in fact needed as scale is assumed in the monocular case). Using these the movement of the camera itself can be expressed as a change in pose over time, where estimates are continuously added.

.2.3 Tracking methods

Using visual methods we can look for certain characteristics in the image that has moved, and use that to get an estimate of a movement or a position on a map. This is also called visual odometry. Much like the odometry in finding the position of a vehicle by integrating its velocity or acceleration, this short term tracking is a short term estimate, but will likely result in large drifts over time.

There are different ways of doing short term tracking. The main distinction between these usually lies in ways of defining an discrepancy between frames as feature based approaches and photometric methods. The photometric method considers the change in the overall image pixel intensity in every frame. The feature based methods consider certain points that are easier to spot, also known as a feature. These points are then easy to consider and track one by one. Additionally, the number of points an algorithm uses or tracks is a normal distinguishing factor between algorithms. Either for the sake of the tracking or in order to make an accurately recreation of a 3D space, some VSLAM methods gather a very large amount of points. These are called dense methods. The ones that collect fewer are called sparse.

Feature based and photometric methods of tracking have their positive and negative sides. As feature based methods require distinguishing sets of pixels for it keep a track, objects without discerning small sets of contrast can not be tracked. An example of this would be white wall. Photometric methods however do not ahve this problem.

Initialize points

An example of a feature based short term tracking method can be described as the following. First several features are chosen from an image. There are several types of features. As some need to be found in a completely new image, some features also have a very detailed feature descriptor eg. what are called ORB and Fast. In either case, a feature is often chosen due to the local contrasts, and that the gradient in contrasts is changing in every direction, making it easier to distinguish (ie. a corner detector). After saving the points and the normalized characteristic gradient in a patch around the chosen point, one could match this point with others by calculating the difference a certain feature descriptor and the other, and if this is below a certain threshold it is considered a match.

Considering we have tracked a number of these features through a couple of frames j, Assuming a known estimation of a tranformation $Telno$, and a measured u_j, one can descrive the error in estimation of the tranformation as:

$$\boldsymbol{E} = \pi(\boldsymbol{T}_{CW}\widetilde{\boldsymbol{x}}_j^W) - \boldsymbol{u}_j$$

Then the new estimation of a transformation would be found by minimizing the reprojection error.

$$\boldsymbol{T}_{CW}^* = \underset{\boldsymbol{T}_{CW}}{\operatorname{argmin}} \sum_j \left\| \pi(\boldsymbol{T}_{CW}\boldsymbol{x}_j^W) - \boldsymbol{u}_j \right\|^2$$

This is a highly nonlinear process, and instead solved in different ways such as calculating linearizations etc.

.2.4 Bundle Adjustment

Bundle adjustments are large optimization calculations VSLAM algorithms take when estimates its change in pose or placements of 3D points. This is computationally heavy. As the error can be expressed as:

$$\boldsymbol{E} = \pi(g(\boldsymbol{T}_{WC}, \boldsymbol{x}_j^W) - \boldsymbol{x}_{n,j}$$

The state variable θ can then be found as:

$$\boldsymbol{T}_{CW}^* = \underset{\boldsymbol{T}_{CW}}{\operatorname{argmin}} \sum_j \left\| \pi(\boldsymbol{T}_{CW}\boldsymbol{x}_j^W) - \boldsymbol{u}_j \right\|^2$$

Bundle adjustments are usually categorized into what the type of state variables they are optimizing and whether it is done on a localized smaller version of the map, or the entire map.

- Motion only BA: It assumes that the map and the movement of the points are static and correct, and tries to optimize the movement of the pose instead. Usually done for a few iterations until convergence.
- Structure only BA: Given that the change in pose is correct, it optimizes the placement of the 3D points. Usually done for a few iterations until convergence.

- Full BA: It assumes nothing and tries to optimize the total error in the map and pose. This is a computationally heavy process.

Even though different algorithms use these differently, the problem of deciding whether an observation is due to a change in pose or an inaccuracy or change in an observation, is often solved simply using a series of bundle adjustments. First one thing is assumed, then the other (given a certain estimate of pose and structure already available).

.2.5 Mapping

All the points that it manages to place in 3D space, it can use to estimate its own speed and trajectory, but it is also used to create a map of the area. All the points are put into a 3D map, along with a represenation of the trajectory that was followed. In order to store this information well, at a certain point what is called a keyframe is chosen. At a keyframe the pose as well as the position of the points it can see is saved. Tracking frames (the ones that are not keyframes), are points in time where its pose and the points are not specifically saved. These are only used to estimate a change, until a new KF is saved. Having more KF results in a more dense and complex map, as well as a larger computational load when performing a BA.

When a KF needs to saved is a problem that the mapping thread deals with. this is usually dependent on a number of different things, such as the number of well tracked points, and how long it has been since the last. In order to simplify the maps, it is normal to try to mimic a larger map by adding up smaller consistent ones. THis could essentially make a larger topologically correct map.

Only taking into account the steps between these keyframes when making a map is what is called visual odometry. It will most likely create global drift over time. A VSLAM algorithm however can perform bundle adjustments over old keyframes as well(the entire map), which are many degrees of freedom, and a reason why pure VSLAM algorithms requires more than VO.

.2.6 Long term tracking

Short term tracking techniques alone result in inaccuracies and global drift. Visual SLAM techniques however can mend these problems with the additional long term tracking thread. There are several ways this is achieved, but the general way is to keep track of previous points that are added to the map, and optimize over the position over these previous points as well in order to get a more precise map. Additionally, the long term tracking thread look for what are called loop closures. Due to the computationally light weight of using short term tracking alone, several popular methods uses simple visual odometry alone.

Loop closing

It is hard to get a refined map over an area when your estimate continously get worse and worse. Imagine going for a walk around a building - with a continously worse estimate of where you are, you wouldn't even be sure when you have arrived back at where you started if it weren't for your ability to recognize where you have been. It is essentially closing an loop on the map and getting a more refined estimate of previous observations doing it. Loop closures can happen faily often, when eg. spotting previously mapped points from another point of view, and is an important addition to framework. Cadena et al. (2016) describes it as an essential part of SLAM, as the metric information and place recognition goes hand in hand, making the other job easier and more robust.

The Visual SLAM algorithm do this using several techniques. A potentially easy to understand way is how it is done in the algorithm ORB-SLAM. In ORB-SLAM all points are described using their respective features (contrast etc. in the surrounding pixels), along with the features of all the other points that have been spotted around this point. A bag of words model, is an AI technique where you take into account a number of information, and try to match this with another. If there are enough inliers(amount of information in common), the bag-of-words model determine that they are a match. ORB-SLAM does this with the points it looks at, and use this to determine whether or not it is looking at a closed loop.

.2.7 Popular VSLAM algorithms

This field is an ongoing one with often new additions and methods being continuously released. However, a couple of main ones are allready stand out as major contenders. As there are different ways to do VSLAM, the best way to understand the technicalities is to know how a couple of the more popular algorithms work.

SVO and PTAM

SVO and PTAM are two methods that are infact visual odometry, as they do not perform any loop closing or anything similar. They will then produce a lot of global drift, but locally what they provide is accurate. SVO is well known enough that is usually mentioned in the context of VSLAM either way. SVO

SVO is a hybrid tracking method, using both the photometric and feature error. It has two threads, tracking and mapping.

It initializes a tracking using all the points it can see from the first image to the next and that it has an estimate for the depth of. It then calculates the photometric error minimization on the pixels in these features, getting the first estimate of the pose. It then improves this initial guess using feature alignment. It finds out what key-frame has the best view of a feature, then optimize using the error in a patch around this feature using the different key-framed poses. Finally it performs a motion only BA, structure only BA then full BA in local map.

The mapping thread is also in charge of estimating the depth of 2D features for which the corresponding 3D point is not known. The depth estimate of a feature is modeled as a probability distribution. Starting off, the variance in this distribution is very high. With every new observation the distribution is refined more, much like a Kalman filter. The new KF only make new observations of this feature searching the epipolar line, making the process a bit faster. When the variance is small enough, the point is converted into a 3D point.

Parallel tracking and mapping (PTAM) is a tracking method for small AR usecases. In other words, it was created for simulating a 3D object placed on a tracked map. Much like other visual odometry, it tracks the change of pose in the camera as well as the map, however it works best locally. PTAM has a tracking and a mapping thread, with the tracking thread also responsible for rendering graphics when PTAM is used for AR.

The tracking thread use a motion model to estimate positions of previously tracked points, and minimize the error of re-projecting them onto the image from another pose. The mapping thread initializes well by maintaining a smooth 2D track of a couple of features, as the camera is moving sideways then triangulating the base map. In addition to assuming the camera moving sideways, it creates a good initial map estimation at the cost of assuming the base map is flat (being meant for small scale AR, this can be a good assumption).

ORB-SLAM

ORB-SLAM gets its name from using what are called ORB features (Oriented FAST and rotated BRIEF). This is a feature descriptor made for computer vision situations where the feature needs to be distinguishable enough to be recognized well.

ORB-SLAM runs 3 threads: tracking, local mapping and loop closure.

The tracking thread localize the camera and decide whether or not to insert a keyframe. Features from the current KF to the new image are matched and then the transformation is estimated by motion-only BA. When there is an estimation of the pose and feature matches, the covisibility graph of keyframes are used to get a local visible map. This local map consists of keyframes that share map points with the current frame. Through reprojection, matches of the local map is searched for on the frame and the camera pose is optimized using these matches.

If the tracking is lost, the place recognition module kicks in and tries to relocalize itself. New KF are taken frequently to ensure robust tracking.

The covisibility graph is a spanning tree linking a KF to the KF with the most points in common(like looking at the same 3d points). So when a new KF is inserted it needs to be put in in this covisibility graph, additionally a Bag of Words representation is made for the KF with the same information. Using the covisibility graph, a point is only placed on the map if it is found 25% of the KF where it is expected. Using epipolar geometry, new map points are created by triangulating ORB from connected keyframes in the covisibility graph. Additionally, unmatched ORBs in one keyframe are checked against other unmatched ORBs in the new keyframe to see if they fit there.

ORB-SLAM then use a local-BA on the current KF and all in its covisibility graph.

To check for loop-closure, the Bag of Words vectors are checked between the new KF and the neighbors in the covisibility graph. If three loop candidates are detected consecutive it is regarded as a serious candidate. For these loops a SIM(3) is estimated, and RANSAC is performed and if there is enough inliers the loop is accepted.

The current KF pose is then adjusted and this is propagated to its neighbors and the corresponding map-points are fused. Finally a pose graph optimization is performed over the essential grapth to take out the loop closure created errors along the graph, included scale drift.

LSD

Large Scale Monocular SLAM (LSD-Slam), is direct semi-sparse method. Not relying on features separates it a bit from the previously mentioned methods. Its keyframes are created using a couple of frames, rendering an inverse depth map. The keyframe also stores the depth variance in this depth map. The map is sparse since the depth and the corresponding variance is only saved for pixels in neighbors with high gradients. Using this it can incorporate depth uncertainty into keyframe based tracking, much like SVO.

The tracking thread estimates the pose change by minimizing photo-metric error.

New images that are not used as keyframes are used for depth map estimation of the current KF, using many small stereo comparisons.

Map optimization Whether or not a KF should be added to the map depends on its distance and difference in orientation. If a KF is made then points are projected from the old KF into the new one, and then spatial regularization is performed, along with outlier removal.

After the KF is added to the map, the keyframes expected to be close based on their tracking checks are used to detect loop-closure. Loop-closure also helps with scale drifts since all keyframes have a scale and they are set up together.

.2.8 Stereo vs Monocular Cameras

All of these techniques struggle with placing elements in 3D, as it needs to estimate its own movement in order to triangulate a points 3D position. Additionally, the scale of map is unknown, as all that is known is simply the relative position between points and camera. However, if we instead had two cameras connected with each-other side-by-side, then the transformation between the two can be considered known, making the placement of these points easier. Infact, the entire initialization step can be skipped using a so called "stereo camera" as both the depth of the points are known from the start, and that there is no problems initializing the scale of the scene, as the transformation between the two cameras is always known.

Appendix B - State of the art techniques and additions to VSLAM

The field of VO and VSLAM see new additions and potential techniques added to it continously, being such a new field. In this section I will go through what is the basic benchmarks for modern uses of these algorithms. This section, is based heavily on the book Multi View Geometry (Hartley (2000)).

Factor graphs

When using an algorithm like a SLAM to estimate a change in pose in eg. a vehicle, there are usually more than just a camera helping you. Instead of getting a live reading of eg. the position from the VSLAM algorithm, in order to then feed this into a sensor fusion along with eg. the GPS, this is done in the VSLAM algorithm itself. In order to perform the sensor fusion one could have used Kalman filters, but in the context of VSLAM there are so many points of data to take into account that one instead use what are called factor graphs for forward passes. This is standard for modern SLAM.

Image processing

In order to mend some of the visual problems that can occur, there are some image processing techniques that can be used on an image before the VSLAM algorithm.

- Remove things that can visually change a lot over time. For instance remove color (using standard techniques, physics based)
- Perform a night-to-day transform if it is night (using DL, learning based).

In he context of place recognition, the use of map in addition to image processing gets us what is typically called the belief state. In order to mend the effects of shade and darkness etc., the modern uses of slam would also directly use DL and computer vision techniques to remove these or do a night-to-day transfer in order for the algorithms to able to recognize points and perform loop closures at different times of day.

Appendix C - Litterature Review in modern Visual SLAM

Several problems considered in SLAM have essentially created their own fields and variations on SLAM. Below some of these are presented. Cadena et al. (2016) present these subfields as the future of SLAM, and where the focus of research should be.

.2.9 Semantic SLAM

Making use of the semantic data gathered through creating a map could be interpretable, and used. However the problem of semantic information is still in its infancy (Cadena et al. (2016)) even if it is an important problem, as it lacks a cohesive formulation. A basic part of it is categorization of what it is seeing, however it should involve high level understanding and related actions as well. Making use of semantic information in a situation is a very basic idea for a human, as the semantic information affect all of someones other reasoning.

When it comes to categorization, which is arguably an important part of semantic information, several strides have been made. This objective is often also refered to as semantic mapping, or semantic SLAM. Daniele De Gregorio (2018) and others use object recognition algorithm alongside SLAM to place objects in 3D space. K. Himri (2018) takes this a step further by using object recognition in cooperation with SLAM, by using objects as landmarks. Sudeep Pillai (2015) use SLAM to aid in the object recognition, as on abject in a known 3D space can get a combined resulting prediction using multiple vantage points. This is essentially showing that SLAM and object recognition perform better when working together. Trung T. Pham (2018) make use of DL to create a recognition algorithm that segments and classifies the individual pixels as members of a class, successfully classifying walls etc. This also has the added ability of finding the pixels that are not classified: unknown objects. Finding unknown is something state of the art 2D classification algorithms such as YOLO can not do. Vibhav Vineet (2018) performs semantic SLAM by identifying a dense on a large scale semantic SLAM outdoors, in real time. Using the fact that all of these techniques require prior knowledge about the objects being identified, Fei and Soatto (2018) and Renato F. Salas-Moreno (2013) use a database of 3D models of the objects it is identifying. Using a 3D model of what is to be identified, it manages to identify the objects pose as well. This allows it extrapolate information about the rest of the object, even if parts of it is not within view. In a similar fashion Niko Sünderhauf (2019) use predictions by YOLOv3 over a couple of frames (see 2.4.3) to create 'Quadrics' (3D surfaces such as ellipsoids) to overlap the object. This essentially estimating the 3D shape of an unknown object.

The semantic mapping with place recognition from Niko Sünderhauf (2016) provide more semantic information by using a convolutional neural network 2.4.3 for place recognition to label places in a gradually built map. In Staffan Ekvall (2006) the semantic information about labeling rooms was done by identifying objects associated with certain room labels.

.2.10 Active SLAM

As opposed to the passive task of creating a map using a position, the dynamic task of changing the position to improve the map or the pose uncertainty is what is called active SLAM (Cadena et al. (2016)). Much like the use of loop-closing, active SLAM could help getting consistent map and tracking, and is therefore an important subject. Active SLAM is complex however, as ties into the problem of 'exploration vs exploitation' dilemma. V Le et al. (2019) use what it calls 'active perception', changing the position to aid in object detection, allways choosing greedy actions maximizing the potential information was shown to increase detection performance. Cadena et al. (2016) explains the problems relating active SLAM. There are several frameworks that that can be used to aid it, but a more popular one follow these basic steps: A couple of potential vantage points are identified, the action with the largest associated utility is chosen, the action is done and finally the robot decides whether or not to continue. This tie into the forecasting of future states, which is a very costly procedure. Solving of this would involve ML.

.2.11 Moving objects

SLAM would take into account all information gathered, even if some was subject to change. The estimations would improve if the algorithm could manage moving objects, by either ignoring them or tracking them. The algorithm must not add a moving object to the map assuming it is stationary (Cadena et al. (2016)). In addition to this rendering a erroneous map, the often sparse maps created by VSLAM are sensitive to errors like these, potentially resulting in bad estimations of pose as well. With this in mind Somkiat Wangsiripitak (2009) track 3D objects and manage to make the SLAM disregard these. Much like Sudeep Pillai (2015) this shows that SLAM and object recognition goes hand-in-hand, they were however mainly done on a small scale. T. Bailey (2006b) consider the applications of SLAM on a large scale, vehicles is an obvious example of objects that can pose a problem. Vibhav Vineet (2018) blabla. Chieh-Chih Wang (2007) introduce SLAM-MOT (SLAM and Moving Object Tracking), which allow for motion modeling of detected objects. It was found that allowing for motion modeling in the general SLAM framework rendered a too large computational demand, making it infeasable at the time. Doing in two separate problems however provided great results.

Appendix D - State of the art techniques and additions to VSLAM

The techniques mentioned in .2.1 are usually the 3 most defining algorithms within the field. However there are notable additions.

.2.12 DSO

Among the newer algorithms created is DSO - Direct Sparse Odometry (Engel et al. (2016)). Among the people working on DSO is Prof. Daniel Cremers of the The Technical University of Munich (TUM). Cremers is on known to be in the forefront of new developments in SLAM, with also the creation of LSD. As opposed to LSD, DSO is visual odometry but can still provide great accuracy.

As the name implies, DSO is sparse and uses the photometric error. As opposed to many other methods, it uses a joint optimization of all model parameters, ie. both motion and geometry. It achieves real-time performance while doing this by omitting smoothness, which is otherwise also accounted for. According to Cramer, DSO has been tested to outperform state-of-the-art both direct and indirect methods in several real world settings. It has several extentions, much like the other algorithms; it can be used with stereo, using an IMU, aided with DL and added extra loop closure detection. As these newer extentions sound promising, I will not take a stance on these in comparison with others, as this much of an in-depth comparison is beyond the scope of this paper.

.2.13 Deep learning

SLAM and Visual SLAM is computationally complex problem with an objective ground truth, which is a sign that DL can make a difference. It has already been shown that it is possible for a neural network to learn change in pose between between frames, as well as estimating the depth of a scene Cadena et al. (2016). Yang et al. (2018) show promising results using deep learning to estimate the change in depth over a couple of frames with a monocular camera, removing the need for depth estimation and epipolar geometry in VSLAM. This field is however very new. In June 2018 the first international workshop on deep learning for Visual SLAM was held.

Appendix E - State of the art Path planning using SLAM

Path planning and object avoidance is a complicated problem. This is especially the case in SLAM, as object avoidance requires very presize localization (Ivan Maurović (2018)). That is why modern research on this it often goes hand in hand with the problem of active SLAM, as a path planning with limited information would improve with the initial or ongoing gathering of more. In control theory, object avoidance manifests itself in constraints, and a problem that needs to be solved using techniques such as model predictive control (MPC) (for more about MPC see Elfving (2018)). Ivan Maurović (2018) use MPC for object avoidance on slow moving UGV (unmanned ground vehicle), showing MPC working for this usecase. Shoudong Huang (2006) use the adaptability of a nonlinear MPC further by looking a larger variable amount of timesteps ahead, and take into account the amount of information and the potential gathering of more. As information gathering is a trade-off between the amount gained, and the accumulation of process noise, it is a lot to take into account, and then also computationally heavy. Ivan Maurović (2016) and Ivan Maurović (2017) approaches the problem using Djikstra's shortest path graph algorithms (A* and D* respectively). Faster obstacle avoidance can be achieved using eg. potential field method and rapidly exploring random tree (RRT) (for more about this see Omid Esrafilian (2017). Using object recognition for path planning Ha1 and Sattigeri (2012) identifies obstacles, and places waypoints on the side of their estimated bounding box.

Appendix F - GNSS

When global positioning is required, GNSS (Global Navigation Satelite System) is usually what is opted for. In order to compare the results found using a VSLAM algorithm with a realistic option, the technology behind GNSS will be explained in this section.

GNSS (Global Navigation Satellite System) is a common way to provide global geolocation to eg. a vehicle, using satellites. GPS (Global Positioning System) is the United States 'Global Positioning System', which consists of 32 satellites. There are other systems like it, for instance the Russian GLONASS. When making a GNSS receiver one can take advantage of several of these, as the more satellites one can take advantage of the better.

As GPS is what is usually opted for, GPS will be used interchangeably with GNSS in this paper.

.2.14 A basic understanding of GNSS

This section is heavily based on Rones (2019). A GNSS receiver uses a low frequency signal recieved from a series of satellites which consists of among other things their position, along with the timestamp from their very accurate atom clocks. Using this timestamp one can measure the time it has taken the series of signals to reach you and from that calculate their distance to you. Using a minimum of four different satellite signals like these that have a clear line of sight to the vehicle in question, one can then solve for its 3D position on the globe. However this calculation has many complications, among other things the atmospheric effects. It is assumed that the signal travel in the speed of light in vacuum the entire journey, which is not the case. Due to all of the complications like these, all GNSS use several satellite signals along with

Kalman filters and optimization algorithms to remove noise and bias. One of these problems that are harder to simply correct are what is called urban canyons.

.2.15 Urban canyons

GPS can not be used indoors as the low frequency signals do not penetrate walls, and objects like it. When you get a GPS signal when indoors, it is more likely due to the signal bouncing around until you recieve it. This would off course take the signal longer time than for a signal taking the straight route. The straight route is what is taken into account when calculating its distance, making a bouncing GPS signal not one to be trusted. The sattelite signal bouncing off of objects before being received is refered to as a multipath signal, and can occur even with a straight line of sight to a the satelite, but as the signal that come from the straight path can easily be distinguished in this situation by choosing the signal (with the timestamp in question) that was recieved first. So GNSS wont be as accurate indoors. However this effect is also met when simply being around other objects that can disrupt the measurers field of view from the satelite - which is common in urban environments. Buildings in cities create what are called urban canyons, that create a hard time for a GPS to position itself.

.2.16 Elevation angle and elevation masking

Given that a larger elevation angle (angle between the horizon and the vertical axis) to the satellite generally has a bad effect on the signal, the estimates arrived at from satelites located at a higher elevation angle are considered more trustworthy. They are then weighted more in the Kalman filter using all the different satellites. Additionally, GNSS recievers completely disregard satelites that have an angle below 15 degrees to you, because these would have a very long time travelling through the atmosphere, and have a much larger probability of being a multipath signal. This is called elevation masking. Additionally the elevation angle factor into the Kalman filter used in the estimations (as the lower elevation angle signals travel through the atmosphere for longer). The train of thought here is that there are often more satelites to choose from, given that we know their position we can use the to choose the satellites that give us the best result. Infact in the case of GPS, one is only guaranteed 4 satelites within the elevation mask, but there are often 5 or 6, and using GLONASS as well it would be even more.

Appendix G - Testing of VSLAM methods

There are several different camera setups that can be used for this. A monocular (single) camera, stereo (two) cameras, with varying intrinsic settings, depth sensors etc. A stereo camera would yield better results, with a denser and more presize pointcloud produced. However, as the standard ardrone in gazebo setup is with only one camera and it requires some work to set it up with two cameras, the standard single camera setup was chosen. Additionally, the stereo setup would not be testable in the real world, as a real life stereo camera was not accessible at the time.

Additionally one could also use cameras with a wider field of view, which would have the effect of allowing the camera to keep track of more points as it moves around. A too narrow field of view can result in sudden loss of track as the camera moves around.

Better or more interesting results can also be achieved using depth sensors, with which a point cloud can be gotten directly from the sensor. This can for instance be gotten using the Xbox kinect sensor, and is often used for this due too its availability.

.2.17 Testing of DSO, LSD and ORB

In order to test the algorithms on real world data, a small eight minute walk around the campus park was recorded on a cellphone. Using pictures of a 2D chess-like grid from the same camera, the Matlab camera calibrator found the intrinsic camera matrix to be:

$$K = \begin{bmatrix} 12575.53 & 0 & 958.46 \\ 0 & 1588.14 & 536.06 \\ 0 & 0 & 1 \end{bmatrix}$$

Much like the camera used on the ardrone, this camera has a low field of view which makes it harder to maintain tracking. The camera was handheld while solely concerned with walking, and on the day it was taken it was starting to rain. All of this can have contributed to a dataset of a worse quality than it could be. As such, one needs to keep in mind that the results from this testing does not show its optimal performance, but one under some sub optimal conditions. As a package delivery drone could be under similar conditions, this is not necessarily a bad thing for the test.

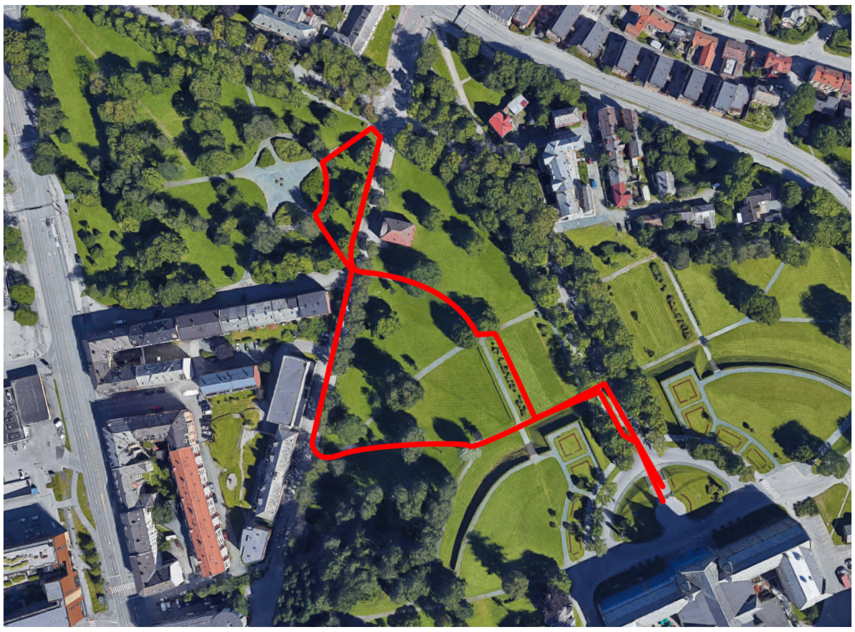


Figure 2: A short walk taken while filming with a cellphone. The small trip features two potential loop closures, and ends where it started

This path was also used in Rones (2019), who used it to test the accuracy of a well tuned GPS. The results of Rones (2019) can be seen in figure 3. The performance of a GPS on the same path can be a good comparison, as GPS as one can see what the standard is when it comes to positioning.

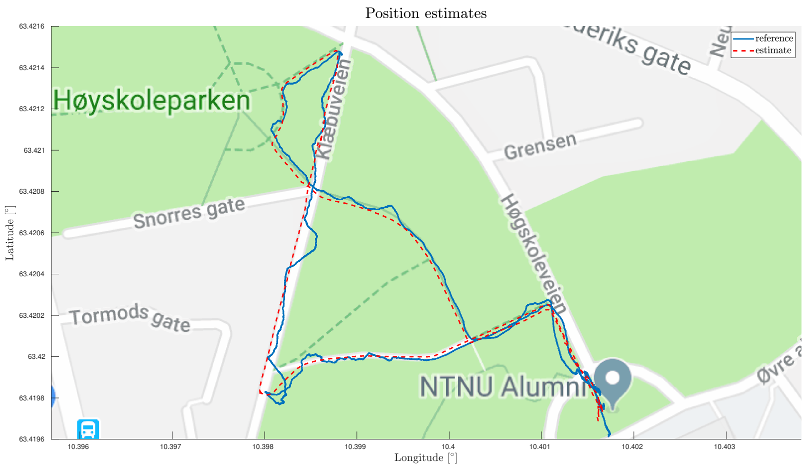


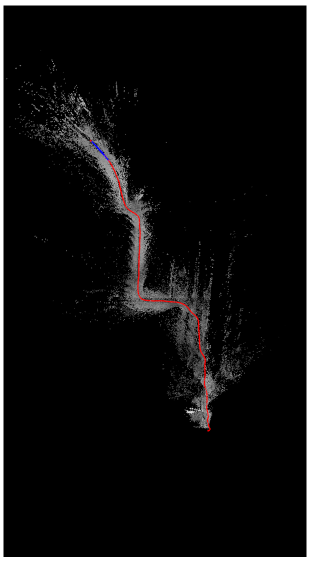
Figure 3: Results from **?** from taking the same path with a GPS. Note the dips in performance in the lower left corner.

In Rones (2019) it was described that the dips in performance at about ($x = 10.398, y =$

62.42) were likely due to the nearby buildings blocking the signal and creating what are called 'urban canyons' (this is described further at .2.15). As the GPS can output the exact coordinates of a position, the VSLAM algorithm simply extends the relationsship between points from one place to another. Its maps are scaleless. However, in order to compare the drift in scale of the different algorithms tested on this dataset in the sections below, the resulting maps are proportionally scaled to match the initial left turn. In other words, in order to compare the drift in scale, the part following the initial left turn is the part that needs to be evaluated.

Testing LSD-SLAM

As one can observe below, LSD-SLAM quickly lost its tracking. This is a poor performance. When using something like this for positioning, one would attempt to initialize the tracking again after losing it, worst case producing several piece-wise good estimations of the map. For the sake of this test however, this was not attempted.


(a) LSD's performance, showing how it very quickly lose track

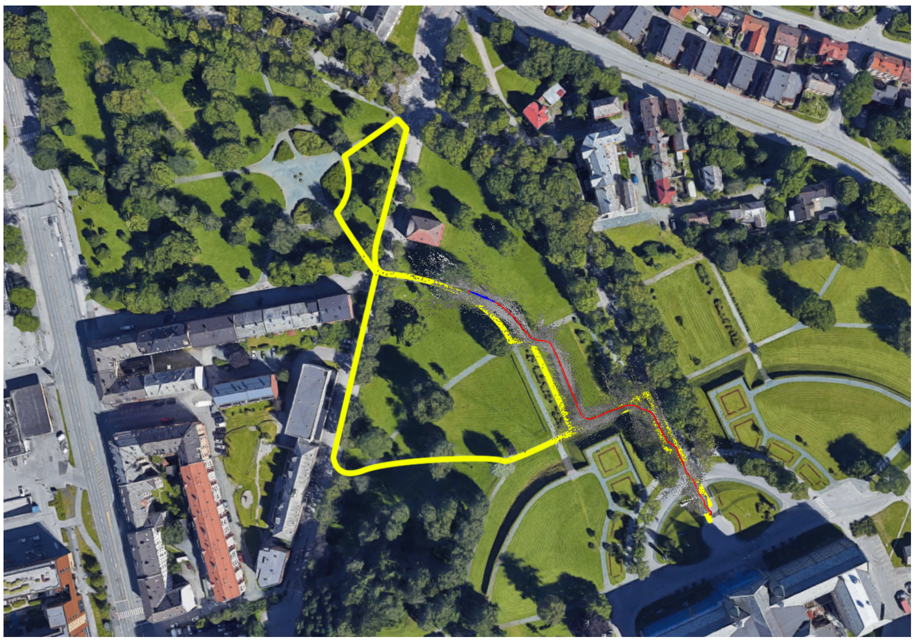

(b) The map produced by LSD slam overlapped with the real map

Figure 4: LSD's performance

A small sign of drift in scale can be observed, in that the right turn was estimated to happen before it did.

Testing DSO

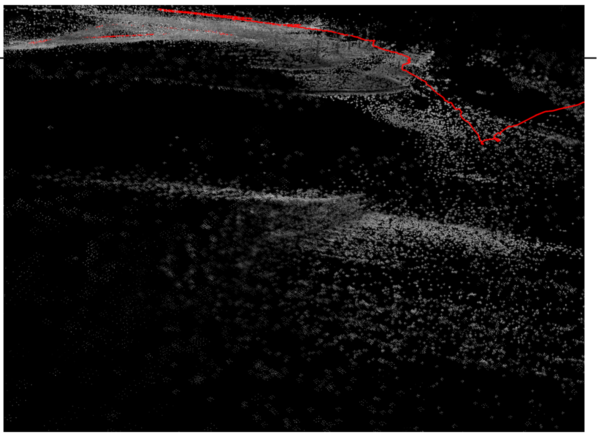
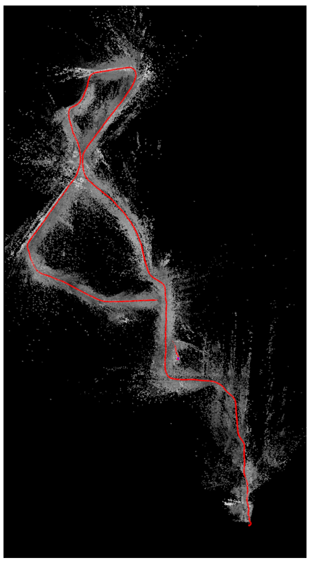


Figure 5: DSO showing an initially accurate pointcloud. One can distinguish the steps in front of the university in the foreground, as well as the upcoming sidewalk in the background

DSO was suprisingly accurate being an odometry based method. On the figure .2.16 one can observe the map produced by DSO, and additionally the map produced by DSO overlapping the real world map of the walk. Note that the algorithm produced an accurate enough result for the entire walk to be distinguishable. The areas affected by noise in the results from Rones2019 due to urban canyons, one can naturally observe here being completely free from this error.

Upon comparison with the ground truth, the scale drift is clearly visible however. Additionally it did not maintain its track all the way back to the beginning. The area immediately after the first left turn, in the middle of the map has several trees, and is apparently an area that makes these algorithms struggle.


(a) The performance by DSO, showing that the shape looks correct, and that it lost track at one point.

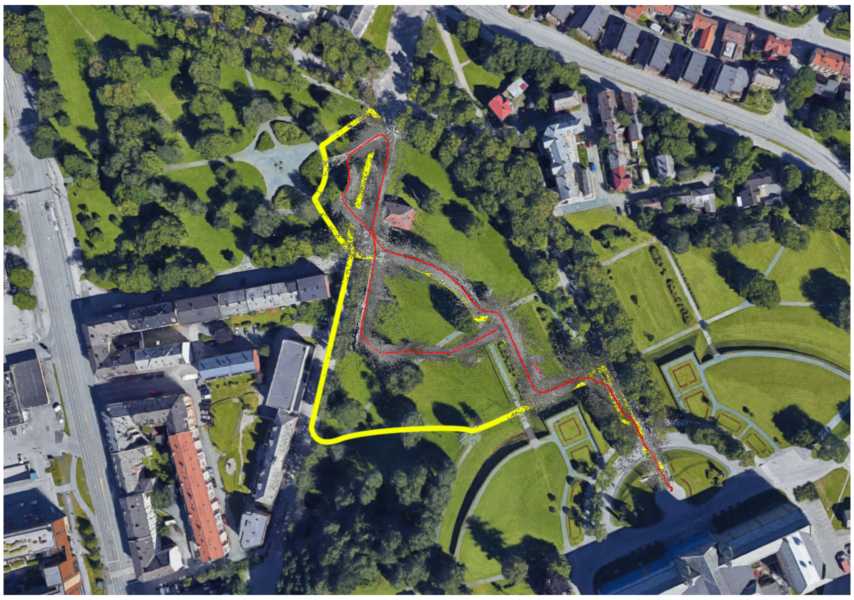

(b) DSO's performance on top of the map of the path taken, showing its large drift in scale

Figure 6: DSO's performance, showing that it performed very well for just being odometry. The scale drift however is very noticable

Testing ORB-SLAM

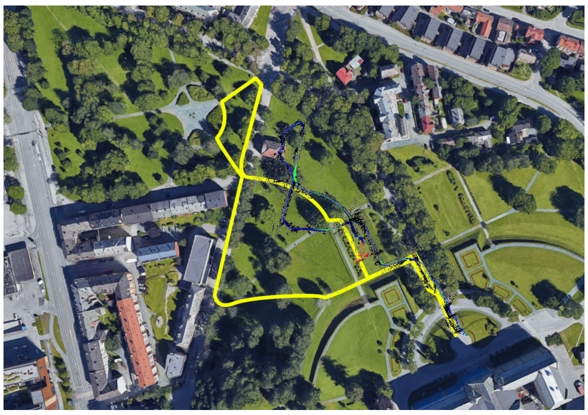
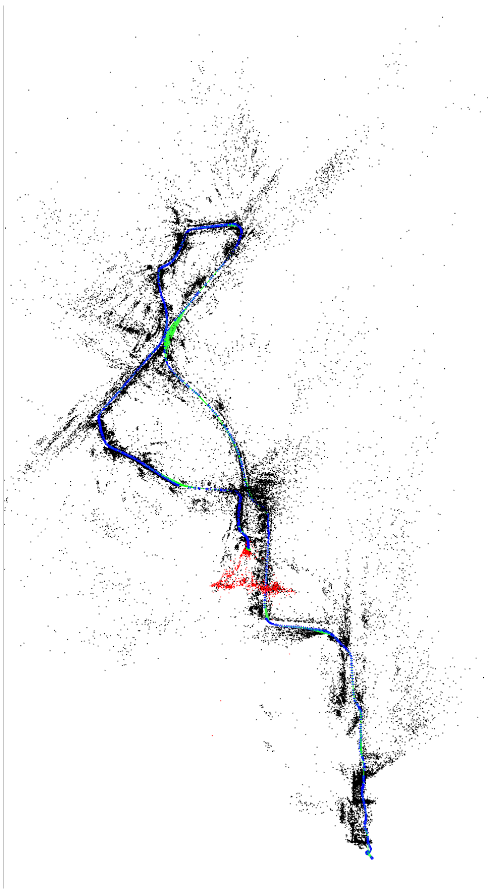


Figure 7: ORB's performances on top of the map

ORB SLAM struggled with its tracking in the same area as as DSO and LSD, but using its robust loop closure thread it managed to arrive at an estimation of the movement in the end. One can see that the estimation of the loop closure has some errors. If the dataset video clip had progressed further, it could have continously created a better estimation of this. In the figure of its movement estimation in relation to the map one can see that it is the most affected by scale drift however.


(a) ORB-SLAMs performance, showing the area where all the algorithms seemed to struggle tracking

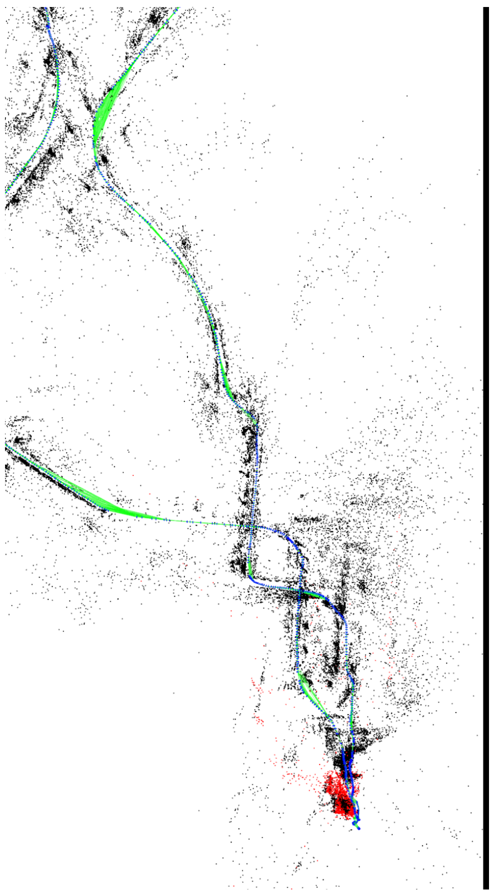

(b) A moment after, having found a loop closure at the beginning

Figure 8: ORB-SLAM's performance

.3 Comparison between SVO, ORB and LSD

As all of these algorithms usually have 3 similar parts, in their own threads, a grid can seem like an obvious way to compare their inner workings properly.

Innerworkings of the most popular VSLAM/VO Algorithms				
	Short term tracking	Long term tracking	Mapping	Extra
Explaination of term	Localize the camera and decide when to put in a new keyframe. Recognize points and estimate a certain transformation	Loop closures or other, to perform a larger refinement or minimize drift	When keyframes and points should be added	Features or limitations
ORB Indirect method	Tracking Thread 1. Initial pose estimation or re-localization. ORB/ FAST features are matched (guided match search with constant velocity), transform estimated through motion-BA. 2. Track local map.The local map(points seen by covisibility graph) is projected onto the frame. Unmatched orbs are tried matched, and then local map motion-BA.. 3. The decision of adding a keyframe. Done if more than 20 frames since last, and more than 50 points tracked, where max 90 percent old points (done generously, making it robust).	Loop Closure Thread Candidates detection: Bag of words vectors are checked between new KF and neighbours in co-visibility graph. 1. If 3 loop candidates in a row detected: serious candidate. 2. Compute sim(3). Results in SIM(3) estimated, RANSAC performed. If enough inliers are found the loop is accepted. 3. Loop fusion. This results in current pose adjusted and propagated to neighbours and map points in entire loop, and optimization over the entire graph pose graph.	Local Mapping Thread 1. Keyframe insertion checking the covisibility graph and Bag of Words. 2. Track local map.The local map(points seen by covisibility graph) is projected onto the frame. Unmatched orbs are tried matched, and then local map motion-BA.. 3. The decision of adding a keyframe. Done if more than 20 frames since last, and more than 50 points tracked, where max 90 percent old points (done generously, making it robust).	Has a place recognition module, if tracking is lost. Uses covisibility graph and BoW model All threads use the same features
SVO Sparse, hybrid-method	Motion Estimation thread 1. Sparse Modelbased Image Alignment: Minimize photometric error on pixels seen in both new image and the last image with known depth(direct) $\rightarrow$ initial guess of transformation of pose. 2. Feature Alignment: Align the initial guess with the rest of the map. Improve initial guess with feature alignment (find out what other KF sees the 3d point), then minimize photometric error in patch around to refine the pose.. 3. Pose & Structure refinement: Optimize pose with motion BA, structure BA, then full BA.	None: Global drift	Mapping Thread 12% rule: if distance from previous KF is more than 12% average scene depth, new KF created. 1. If keyframe: Feature extraction and initialize depth filters. Every depth filter is associated to a keyframe, initialized with high uncertainty. Its depth has a corresponding probability distribution. It estimates the depth of 2D features using probability distributions, which is updated with later observations. Starting off the mean is the average scene depth, high variance. 2. Else: Update depth filters. When variance small enough, the point is made 3D using π^{-1}, and inserted map. Look for features on the epipolar line here the desnity is highest.	Performs no loop closures, being VO. Creates feature points using "FAST". Computationally lightweight.

Innerworkings of the most popular VSLAM/VO Algorithms				
	Short term tracking	Long term tracking	Mapping	Extra
LSD Semi-dense, direct	Tracking Thread Track on current KF: Estimate pose transformation based on photometric error. Minimize photometric error to find pose . A KF is an image, inverse depth map and its variance: $K(I,D,V)$. It is sparse, as its only saved for spots of high gradients. 1. Calculate epipolar line 2. Search for correct disparity 3. Inverse depth from disparity	Map Optimization Thread Candidates detection: Bag of words vectors are checked between new KF and neighbours in co-visibility graph. 1. If 3 loop candidates in a row detected: serious candidate. 2. Compute sim(3). Results in SIM(3) estimated, RANSAC performed. If enough inliers are found the loop is accepted. 3. Loop fusion. This results in current pose adjusted and propagated to neighbours and map points in entire loop, and optimization over the entire graph pose graph.	Depth Map Estimation thread The map is continusly optimized in the background using posegraph optimization. Based on proximitty and then pose we try to detect loop closures. The 10 closest KF are used for this when KF is added 1. Find closest KF 2. Estimate SIM(3) edges (ligning) 3. Map optimization	Incoorporates depth uncertainty Large scale

Appendix H - Testing LSD SLAM for object avoidance in Gazebo

.3.1 Octomap

The use of a VSLAM algorithm would result in what is called a pointcloud. A 3D pointcloud is an amount of singular points placed in 3D space, but these points are not like pixels, in that they take up a certain amount of space. They only have a position. A pointcloud does not have any volume, as its singular points does not. In order to calculate the space occupied in a pointcloud, one transforms it into an octree (Hao-Chih (2012)).

When an object or point is found, one can show its 3D position with a volume using a bounding box - a box indicating the bounds of the object or points in question. This bounding box can be made in a certain size, resulting in fewer bounding boxes. In 2D this bounding box can be thought of a pixel. When more resolution is needed, bounding boxes could be made smaller. In this case, in 2D the bounding box would be split into 4 - making it a quad-tree. In 3 dimensions this potential change in resolution makes it an oct-tree. It is from this selectively dense resolution that octrees gets its name (Hao-Chih (2012)).

The ROS octomap package is a standard package that comes with the ROS installation. It is used to create 3D bounding boxes, also known as occupancy grids, using pointclouds. In order to use this to create 3D obstructions on map, the octomap package is normally used. Octomap can be modified to take full advantage of the octree datatype, by creating its characteristic selectively dense resolution, but this will not be used here. (Hao-Chih (2012)).

.3.2 Move it

Move it a motion planning framework made for ROS, that is made for motion planning in cooperation with octomap and its occupancy grid mapping. The move it package for ROS is a program mainly made for stationary robot arms, that solves for a 3D movement avoiding obstructions. It uses an RRT algorithm to do this (Hao-Chih (2012)).

.3.3 Ardrone indoor navigation package Hao-Chih (2012)

As previously mentioned, the ardrone has a monocular camera. VSLAM on a monocular camera come with the side effect issues estimating scale. The scale of the scene would have to be initialized or estimated. This estimation is performed by LSD. In order to get a better estimation of the scale, a package from a Hao-Chih (2012) estimates this by getting both an estimation from PTAM, LSD-SLAM and the IMU and using a recursive least-square algorithm it continously improves until when a user has observed a convergence. It then feeds this easily into the previously mentioned Move-it package, that then make short term waypoints using the TUM ardrone package. For more information about this framework see Hao-Chih (2012).

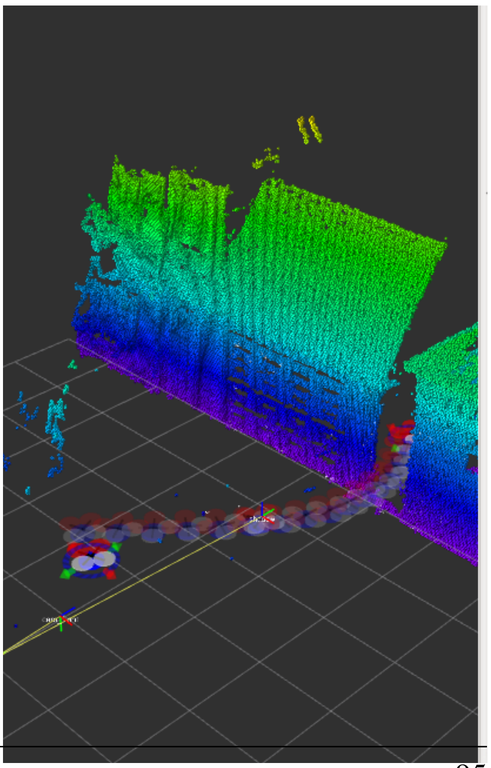


Figure 9: Using the VSLAM framework from , two buildings are detected, and moveit creates the path of going in-

.3.4 Results

The framwork created in Hao-Chih (2012) was shown to work, as detections can be made, and then avoided. This could be shown to allow the drone to move accross the test environment made in Gazebo. However, running this in real time on a system with the specifications described in 6 went slow.

Appendix I - PID explained

The explaination of this basic controller is taken from Elfving (2018).

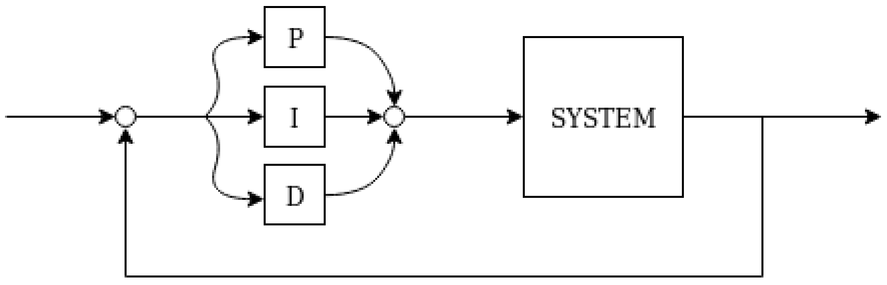


Figure 10: Block diagram for a PID controller

Linear control is often great to take advantage of when possible because of its simplicity, robustness and that it can possibly deal with some strongly nonlinear processes.

One of the earliest controllers, and one that has got a lot of good faith, is the PID controller. Because of its robustness, tune-ability as well as its simplicity to implement, it has remained one of the most popular as well the most widely used controllers out there. The PID works like this: Using a sensor to get a measurement of the process you are trying to control, you use that measurement in a feedback-loop and compare it to the reference value - the value that you're trying to get your system to converge to. This error in measurement, you feed back. The input to the system can be something proportional to this error. This is what is called a Proportional controller, as the input is proportional to the error. A PID controller also adds an element that integrates the total error over time, and adds that as well as a factor proportional to the errors rate of change. Tuning the PID ie. changing the constants K_p,K_i and K_d, will often be a trade-off between wanting a robust or an optimally performing controller (Fossen (2014)).

Modern uses of the PID also has high pass filter on the derivative term. This makes the faster oscillations not being taken into account in the derivative term, as these are in practice noise. In the context of state of the art vehicle control one might also add a factor that is inversely proportional to the acceleration, and connected through a lowpass-filter, so that careful and slow movements of eg. a ship, are made even more so (Fossen (2014)).

Appendix J - Code

.3.5 Main launch file

```
<?xml version="1.0" encoding="utf-8"?>


<launch>
  <!-- GENERAL -->
  <arg name="launch_prefix" default=""/>

  <!-- Bus aiding tools and republish-->
  <node pkg="my_package" type="plugins.py" name="bus_tools" output="screen" launch-prefix="">
  </node>


  <!-- Resize image to look for green size (1/32 crop for simplicity) -->
  <include file="$(find ros_imresize)/launch/imresize.launch" />
  <param name="/ros_imresize/resize_width" value="20" />
  <param name="/ros_imresize/resize_height" value="15" />
  <param name="/ros_imresize/topic_crop" value="/businteract/image" />
  <param name="/ros_imresize/camera_info" value="/ardrone/camera_info" />

  <!-- Look for green, output as white and black -->
  <include file="$(find opencv_apps)/launch/rgb_color_filter.launch" />

  <!-- Start yolo -->
  <include file="$(find darknet_ros)/launch/darknet_ros.launch" />

  <!-- Start LK opencv -->
  <include file="$(find opencv_apps)/launch/lk_flow.launch" />

  <!-- Start aruco marker -->
  <include file="$(find ar_track_alvar)/launch/pr2_indiv_no_kinect.launch" />


  <!-- Drone navigation -->
  <node pkg="my_package" type="interactorMain.py" name="drone_node" output="screen" launch-prefix=""
  </node>

</launch>
```

.3.6 Main interactor and state manager

```
#!/usr/bin/env python
from __future__ import division
import rospy
from std_msgs.msg import String
import std_msgs.msg
from dynamic_reconfigure.msg import ConfigDescription
import re
from sensor_msgs.msg import Imu
from geometry_msgs.msg import Vector3
from std_msgs.msg import Empty
import math
from geometry_msgs.msg import Vector3Stamped
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from visualization_msgs.msg import Marker
from octomap_msgs.msg import Octomap
from nav_msgs.msg import OccupancyGrid
import time
from geometry_msgs.msg import PoseStamped
from sensor_msgs.msg import Image
import os
from scipy import zeros, signal, random
import matplotlib.pyplot as plt


class DroneState:
    def __init__(self):

        # HARDWARE
        self.position = [0, 0, 0]
        self.orientation = [0, 0, 0]

        # NAVIGATION
        self.goalProximitty = 2
        self.goal = [3, 50]
        self.safeHeight = 10
        self.relativeBusPos = [0, 0, 0]

        startState = "asleep"
        self.state = startState
        self.stateMemory = [startState]
        self.semaphore = False
        self.test = True
        self.cameraRes = 640  # as the horizontal is the only used
        self.batteryLevel = 0.2  # percentage
        self.thresholdBus = 0.4

        # Drones and additions
        self.nickname = random.randint(0, 9)

        # road
        self.carRoad = [20, None]  # road moves along y axis at x = -3
        self.alongRoad = False
        self.twistMemory = [0] * 10
        self.linearMemoryX = [0] * 10
        self.linearMemoryY = [0] * 10
        self.oriDiscrepancy = 0  # mainly for LOS bus tracking
        self.gpsEnabled = True
        self.waypointQ = []

        # debug
        self.iterator = 0
        self.inputs = []
        self.debugconst = 0
        self.testlist = []
        self.talkative = False
        self.updateCounter = 0

        # landing
        self.loopFreq = 10
        self.busLoopIter = self.loopFreq - 3

        # standards
        self.landmessage = {"ori": 0, "pos": [0, 0, 0], "mode": "", "delay": 1}
        print("initialized")

    def continouslyCommence(self, data, pubCom):
        self.statemachine(data, pubCom)
        self.lowlevelcontroller(data, pubCom)

    def filter_sbs(self, data):
        """
        Filter generate

        Using sci py a filtered version is returned, using 50 and 0.04 as of now

        Args:
            data: list of measured discrepancy to regulate

        Returns:
            filtered response list

        Raises:
            None as of now
        """

        b = signal.firwin(50, 0.04)
        z = signal.lfilter_zi(b, 1)
        result = zeros(len(data))
        for i, x in enumerate(data):
            result[i], z = signal.lfilter(b, 1, [x], zi=z)
        return result  # could have had just the final input here: [-1]

    def getRefModel(self, measuredLx, measuredLy, measuredT):
        """
        Get the refmodel signal

        Get the refmodel for the current timstep

        Args:
            measuredx: discrep x
            measuredy: discrep y
            measuredz: discrep z


        Returns:
            reference response lists x,y,z

        Raises:
            None as of now
        """
        self.twistMemory.append(measuredT)
        self.linearMemoryX.append(measuredLx)
        self.linearMemoryY.append(measuredLy)
        refLx = self.filter_sbs(self.linearMemoryX)
        refLy = self.filter_sbs(self.linearMemoryY)
        refT = self.filter_sbs(self.twistMemory)

        return refLx, refLy, refT

    def callbackBattery(self, data):  # should take com and vallback live battery signal

        self.batteryLevel = data

    def callbackBusevent(self, data, pubCom):
        """
        Changes state bc busevent

        When plugins have measured thresholds broken they send messages dealt with here

        Args:
            data: message sendt
            pubCom: communicatio channel

        Returns:
            nothing
            changes only state variables

        Raises:
            None as of now
        """
        msg = interpret(data)
        if self.talkative:
            print("4. Callback busevent {}. msg is {}".format(self.state, msg))

        if msg[0] == "greenSpotted" and self.state == "onTheRoad":
            if self.state != "onhold":
                self.stateMemory.append("onhold")
                self.state = "onhold"
                print("Amount of green above threshold. Hold up")

        if (
            (msg[0] == "LOShere:")
            and (self.state == "onhold" or self.state == "onLOS")
            and not (self.semaphore)
        ):
            pcHorizontal = float(msg[1])
            if pcHorizontal:
                self.oriDiscrepancy = self.cameraRes / 2 - pcHorizontal

            self.semaphore = True
            self.setTumautoCommand("c clearCommands", pubCom)
            self.setTumautoCommand("u l Autopilot: Cleared Command Queue", pubCom)

            if self.state != "onLOS":
                self.stateMemory.append("onLOS")
                self.state = "onLOS"
            self.debugconst = float(msg[1])

            self.semaphore = False

    def safeClearQueue(self, pubCom):
        self.setTumautoCommand("u l Autopilot: Cleared Command Queue", pubCom)
        self.setTumautoCommand("c clearCommands", pubCom)

    def goToRoad(self):
        """
        Go to road

        Add the closest road intersection points to waypoint queoue

        Args:
```

```

        Returns:
            nothing
            changes only state variables

        Raises:
            None as of now
        """
        roadHeight = 8
        roadIntercept = [None, None]
        if self.carRoad[0]:
            roadIntercept = [self.carRoad[0], self.position[1], roadHeight]
            roadOutercept = [self.carRoad[0], self.goal[1], roadHeight]
        elif self.carRoad[1]:
            roadIntercept = [self.position[1], self.carRoad[1], roadHeight]
            roadOutercept = [self.goal[1], self.carRoad[1], roadHeight]

        forward = 0.5 * self.forwardAngle(roadIntercept) * 180 / math.pi
        roadAngle = forward + 90

        self.waypointQ.append(roadIntercept)
        self.waypointQ.append(roadOutercept)
        self.waypointQ.append(self.goal)

    def convergeOnCoordinate2(self, goal, wantedOri=0, mode="default", delay=0):
        """
        Converge on coordinate

        Create an input converging on goal

        Args:
            goal: the goal position
            wanted orientation: the orientation wanted when having converged
            mode: different aggressiveness and tuning of P parameters
            delay: delay in twist convergence

        Returns:
            nothing
            changes only state variables

        Raises:
            None as of now
        """
        pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)

        if talkative:
            print("Objective aquired: {}{}".format(data[0], data[1]))
        posDiscrepancy = [
            goal[0] - self.position[0],
            goal[1] - self.position[1],
            goal[2] - self.position[2],
        ]
        oriDiscrepancy = self.orientation[2] - wantedOri

        distance = math.sqrt(posDiscrepancy[0] ** 2 + posDiscrepancy[1] ** 2)
        distanceError = math.sqrt(posDiscrepancy[0] ** 2 + posDiscrepancy[1] ** 2)
        heightError = posDiscrepancy[2]

        ##REGULATION. P-reg
        if mode == "default":
            kpT = 0.6  # 0.5 #angular
            kpL = 0.8  # 0.2 #linear
            kpH = 0.7  # vertical
        if mode == "easy":
            kpT = 0.001  # 0.5 #angular
            kpL = 0.001  # 0.2 #linear
            kpH = 0.001  # vertical
        if mode == "delay":
            kpT = 0.05 / delay
            kpL = 0.3
            kpH = 0.2
        if mode == "forward":
            kpT = 0.3
            kpL = 0.7
            kpH = 0.2

            rotationMatrix = [
                math.cos(self.orientation[2] * math.pi / 180),
                math.sin(self.orientation[2] * math.pi / 180),
                -math.sin(self.orientation[2] * math.pi / 180),
                math.cos(self.orientation[2] * math.pi / 180),
            ]
            convertedCord2 = [
                posDiscrepancy[0] * rotationMatrix[0]
                + posDiscrepancy[1] * rotationMatrix[1],
                posDiscrepancy[0] * rotationMatrix[2]
                + posDiscrepancy[1] * rotationMatrix[3],
            ]
            convertedCorderu = [convertedCord2[0], convertedCord2[1]]
            delay = 1
            if distanceError > 3:
                delay = 10
            wantedOri = (
                math.atan2(convertedCorderu[1], convertedCorderu[0]) * 180 / math.pi
            )

        inputT = kpT * wantedOri
        inputLx = kpL * convertedCorderu[0]
        inputLy = kpL * convertedCorderu[1]
        inputH = kpH * heightError

        ##CREATE MESSAGE
        twist = Twist()
        twist.angular.x = 0
        twist.angular.y = 0
        twist.angular.z = inputT
        twist.linear.x = inputLx
        twist.linear.y = inputLy
        twist.linear.z = inputH
        pub.publish(twist)

    def forwardAngle(self, wantedPos):
        """
        Forward Angle

        Get the angle that would correspond to forward, given a change in position from current to w

        Args:
            goal: the goal position
            wanted orientation: the orientation wanted when having converged
            mode: different aggressiveness and tuning of P parameters
            delay: delay in twist convergence

        Returns:
            nothing
            changes only state variables

        Raises:
            None as of now
        """

        posDiscrep = [self.position[0] - wantedPos[0], self.position[1] - wantedPos[1]]
        return math.atan2(posDiscrep[1], posDiscrep[0])

    def initAutoTum(self, pubCom):
        """
        initTum

        Sends all the initial messages to TUM AR-Ardrone controller, in order for it to start succes

        Args:
            pubCom: the channel to TUM-ardrone communication

        Returns:
            nothing
            changes only state variables

        Raises:
            None as of now
        """
        self.setTumautoCommand("c autoInit 500 800", pubCom)
        self.setTumautoCommand("c setMaxControl 1", pubCom)
        self.setTumautoCommand("c setInitialReachDist 0.5", pubCom)
        self.setTumautoCommand("c StayWithDis 0.5", pubCom)
        self.setTumautoCommand("c setMaxControl 1", pubCom)

    # started continously with framwthread
    def lowlevelcontroller(self, data, pubCom):
        """
        lowlevelcontroller

        Finite State Machine, in charge of delegating lowlevel controller schemes, depending on curr

        Args:
            pubCom: the channel to TUM-ardrone communication

        Returns:
            nothing

        Raises:
            None as of now
        """

        if self.state == "onhold":
            # control instead of the current 'hover' by 'autonomy' could be implemented

            # The waiting functionalities, forcing a switch of states if no bus is found should be a
            # For testing purposes, and due to extreme waiting times by yolo it is not implemented.
            return
        elif self.state == "onLOS":
            self.convergeOnRefmodel(
                signalDiscrep=self.oriDiscrepancy, wantedOri=0, mode="easy", delay=1
            )
        elif self.state == "onLanding" and not (self.semaphore):
            self.iterator += 1
            dist2d = math.sqrt(
                self.landmessage["pos"][0] ** 2 + self.landmessage["pos"][1] ** 2
            )
            dist3d = math.sqrt(dist2d ** 2 + self.landmessage["pos"][2] ** 2)
            if dist3d < 3:
                pub3 = rospy.Publisher("ardrone/land", Empty, queue_size=1)
                pub3.publish()
            self.convergeOnPlatform(
                self.landmessage["pos"],
                oriDiscrepancy=self.landmessage["ori"],
                mode="delay",
                delay=self.landmessage["delay"],
            )
        elif self.waypointQ:
            totalPosError = d3Distance(posDiscrep(self.waypointQ[0], self.position))
            if (totalPosError) < 3:
                self.waypointQ.pop(0)
            self.convergeOnCoordinate2(
                self.waypointQ[0], wantedOri=0, mode="forward", delay=0
            )

    def statemachine(self, data, pubCom):
        """
        lowlevelcontroller

        Finite State Machine, in charge of delegating lowlevel controller schemes, depending on curr

        Args:
            pubCom: the channel to TUM-ardrone communication

        Returns:
            nothing

        Raises:
            None as of now
        """
        if talkative:
            print(
                "I am drone{}. Story: Im {}\n\n".format(self.nickname, self.stateMemory)
            )
        if self.state == "asleep" and not (self.semaphore):
            self.semaphore = True
            rospy.sleep(3)
            print("decided to sleep")
            self.semaphore = False
            if self.state != "onGround":
                self.stateMemory.append("onGround")
            print(self.stateMemory)
            self.state = "onGround"

        if (
            (self.batteryLevel <= self.thresholdBus)
            and not (self.alongRoad)
            and (self.goal)
            and (self.state == "online")
            and not (self.semaphore)
        ):

            self.semaphore = True
            thresholdStop = 0.2
            thresholdBus = 0.4
            self.goToRoad()
            self.alongRoad = True
            if self.talkative:
                print("STATE ORDERS 3: go to road")
            if self.state != "onTheRoad":
                self.stateMemory.append("onTheRoad")

            self.state = "onTheRoad"
            rospy.sleep(3)
            self.semaphore = False

        elif ((self.state == "onGround") and (self.goal)) and not (self.semaphore):
            self.semaphore = True
            self.initAutoTum(pubCom)
            print("sent init sequence")
            self.setTumautoCommand("u l Autopilot: Cleared Command Queue", pubCom)
            self.setTumautoCommand("u l Autopilot: Start Controlling", pubCom)
            self.setTumautoCommand("c clearCommands", pubCom)
            self.setTumautoCommand("c takeoff", pubCom)
            self.setTumautoCommand("c start", pubCom)

            if self.state != "online":
                self.stateMemory.append("online")

            self.state = "online"
            self.semaphore = False

        elif self.state == "online" and self.goal and not (self.semaphore):
            self.semaphore = True
            command = "c goto {} {} {} 0".format(self.goal[0], self.goal[1], height)
            self.setTumautoCommand(command, pubCom)
            if self.state != "onmove":
                self.stateMemory.append("onmove")
            self.state = "onmove"
            self.semaphore = True

        elif self.state == "undefined":  # testing purposes
            self.iterator += 1
            if self.iterator > 300:
                print("change state")
                self.state = "onTheRoad"

        elif (self.state == "onLOS") and not (self.semaphore):
            self.semaphore = True
            print("onLOS state through testnav")

            self.semaphore = False

    def gps(self, data):
        if self.gpsEnabled:
            self.position = [
                data.pose.pose.position.x,
                data.pose.pose.position.y,
                data.pose.pose.position.z,
            ]

    ##ESTIMATES
    def callbackTumpose(self, data):  # if estimation paradigm etc is used, without gps
        if not (self.gpsEnabled):
            self.position = [
                data.pose.position.x,
                data.pose.position.y,
                data.pose.position.z,
            ]
        quatOri = data.pose.orientation
        eul = quat_to_euler(quatOri.x, quatOri.y, quatOri.z, quatOri.w)
        self.orientation = eul

    def setPosEstimate(self, data):
        self.updateCounter += along

        updateFreq = 4  # usikker paa denne. Rostopic hz
        h = along / updateFreq
        if self.gpsEnabled:
            return

        self.position[0] += data.vector.x * h
        self.position[1] += data.vector.y * h
        self.position[2] += data.vector.z * h

        if self.updateCounter > 5:
            self.printStats()
            self.updateCounter = 0

    def setPosGPS(self, data):  # doesnt work.
        self.updateCounter += 1
        self.position[0] = data.pose.pose.position.x
        self.position[1] = data.pose.pose.position.y
        self.position[2] = data.pose.pose.position.z

        if self.updateCounter > 5:
            self.updateCounter = 0
            self.gotoGoal()

    ##TROUBLESHOOTING
    def printStats(self):
        print(
            "hello, Im {}! Pos:{}, {}, {}".format(
                self.nickname,
                round(self.position[0], 2),
                round(self.position[1], 2),
                round(self.position[2], 2),
            )
        )
        print(
            "\t\t\tOri:{}, {}, {}".format(
                round(self.orientation[0], 2),
                round(self.orientation[1], 2),
                round(self.orientation[2], 2),
            )
        )
```

```python

    ##CALLBACK
    def callbackImu(self, data):
        header = data.header
        ori = data.orientation
        self.orientation = quat_to_euler(ori.x, ori.y, ori.z, ori.w)

    def callbackVelo(self, data):
        # if talkative: print("EstimatePOS: {}".format(data))
        self.setPosEstimate(data)

    def callbackMarker(self, data, pubCom):
        self.state = "onLanding"
        self.iterator = 0
        quatOri = data.pose.orientation
        eulOri = quat_to_euler(quatOri.x, quatOri.y, quatOri.z, quatOri.w)

        self.busLoopIter = 0
        self.relativeBusPos = [
            data.pose.position.x,
            data.pose.position.y,
            data.pose.position.z,
        ]  # realtive down doesnt matter?
        self.landOnBusInput(self.relativeBusPos, eulOri[2], pubCom)

    def landOnBusInput(self, relativePosition, angle, pubCom):
        if self.relativeBusPos > 3:
            self.setTumautoCommand("c clearCommands", pubCom)
            self.setTumautoCommand("u l Autopilot: Cleared Command Queue", pubCom)
        dist = max(self.position[2] - 4.6, 1)
        if dist > 7:
            angle = 0
        delay = 1
        if abs(self.relativeBusPos[0]) > 1.3:
            # fast turning will lose track
            delay = 10 * abs(self.relativeBusPos[0])
        posDiscrepancy = relativePosition
        self.landmessage.clear()
        self.landmessage["pos"] = posDiscrepancy
        self.landmessage["ori"] = angle
        self.landmessage["delay"] = delay
        self.state = "onLanding"

    ##CONTROL AND DRONE STATE VARIABLE FUNCTIONS

    def setTumautoCommand(self, command, pubCom):
        rate = rospy.Rate(3)
        msg = String()
        msg.data = command
        pubCom.publish(msg)

    def straightForward(angle):
        """This returns this thing

        This is a function that descirptiuon

        Args:
            angle: a float to convert

        Returns:
            converted angle from radians to degrees

        Raises:
            FileNotFoundError: Wrong location on file system
        """
        rad = angle * (math.pi / 180)
        x = math.cos(rad)
        y = math.sin(rad)
        return x, y

    def setGoal(self, pos, place="end"):
        if talkative:
            print("Objective aquired: {},{}".format(data[0], data[1]))
        if not (place) or place == "end":
            self.waypointQ.append(pos)
            return
        if place == "start":  # neccessary?
            print("unimplemented")
            return

    def convergeOnPlatform(
        self, posDiscrepancy, oriDiscrepancy=0, mode="default", delay=0
    ):
        """Converge on platform

        Using some addons, it converges to a continously updated position

        Args:
            posDiscrepancy: list 3d
            posDiscrepancy: list 3d (not used as of now)
            mode: default main

        Returns:
            None as of now

        Raises:
            None as of now
        """
        pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)
        if talkative:
            print("Objective aquired: {}{}".format(data[0], data[1]))
        distance = math.sqrt(posDiscrepancy[0] ** 2 + posDiscrepancy[1] ** 2)
        distanceError = math.sqrt(posDiscrepancy[0] ** 2 + posDiscrepancy[1] ** 2)
        heightError = posDiscrepancy[2]
        ##CONTROL.                        P-reg
        if mode == "default":
            kpT = 0.8  # angular
            kpL = 0.8  # linear
            kpH = 0.8  # vertical
        if mode == "easy":
            kpT = 0.001  # angular
            kpL = 0.001  # linear
            kpH = 0.001  # vertical
        if mode == "delay":
            kpT = 0.05 / delay
            kpL = 0.3
            kpH = 0.2
        inputT = kpT * oriDiscrepancy * (-1)
        inputLx = kpL * posDiscrepancy[0] * (-1)
        inputLy = kpL * posDiscrepancy[1] * (-1)
        inputH = kpH * heightError * (-1)
        ##CREATE MESSAGE
        twist = Twist()
        twist.angular.x = 0
        twist.angular.y = 0
        twist.angular.z = inputT
        twist.linear.x = inputLy
        twist.linear.y = inputLx
        twist.linear.z = inputH
        pub.publish(twist)
        if self.talkative:
            print("Wanted values: {},{}".format(self.goal[0], self.goal[1]))
            print("Current values:{},{}".format(self.position[0], self.position[1]))
            print(
                "Distance error: {}, angle error: {}".format(
                    distanceError, oriDiscrepancy
                )
            )
            print("Resulting in inputs:{}and{},{}".format(inputT, inputLx, inputLy))

    # creating a guidance control using refmodel, in this case only used for bus track with LK
    def convergeOnRefmodel(
        self, signalDiscrep=None, wantedOri=None, mode="default", delay=1
    ):
        """Converge on ref model

        Using some addons, it converges to the filtered response of signal given

        Args:
            posDiscrepancy: list 3d
            posDiscrepancy: list 3d (not used as of now)
            mode: default main

        Returns:
            None as of now

        Raises:
            None as of now
        """
        pub = rospy.Publisher("cmd_vel", Twist, queue_size=1)
        if self.talkative:
            print("Objective aquired: {}{}".format(data[0], data[1]))
        landDetectionHeight = 13
        heightError = landDetectionHeight - self.position[2]

        refLx, refLy, refT = self.getRefModel(
            0, 0, signalDiscrep
        )  # only used for jaw as of now

        ##CONTROL.                      #P-reg as of now
        if mode == "default":
            kpT = 0.05  # angular
            kpL = 0.2  # linear
            kpH = 0.6  # vertical
        if mode == "easy":
            kpT = 0.0003  # angular
            kpL = 0.2  # linear
            kpH = 1.3  # vertical
        if mode == "forward":
            kpT = 0.5  # angular
            kpL = 0.5  # linear
            kpH = 0.2  # vertical
        inputT = kpT * refT[-1]
        inputLx = (
            10
        )  # kpL*refLx[-1]     #Refmodel unused as constant 10 forward works well
        inputLy = kpL * refLy[-1]
        inputH = kpH * heightError
        ##PLOT REFMODEL PERFORMANCE
        if self.test:
            if len(self.twistMemory) > 300:
                l1 = plt.plot(self.twistMemory, label="Raw measured input")
                l2 = plt.plot(refT, label="Rerence model")
                plt.title("Reference model response")
                plt.xlabel("Timestep")
                plt.ylabel("Input")
                plt.grid(True)
                plt.setp(l1, "color", "r", "linewidth", 2.0, "linestyle", "--")
                plt.setp(l2, "color", "b", "linewidth", 2.0)
                plt.legend()
                plt.show()
        ##CREATE MESSAGE
        twist = Twist()
        twist.angular.x = 0
        twist.angular.y = 0
        twist.angular.z = inputT
        twist.linear.x = inputLx
        twist.linear.y = inputLy
        twist.linear.z = inputH
        pub.publish(twist)
        if self.talkative:
            print("Current values:{},{}".format(self.position[0], self.position[1]))
            print(
                "Distance error: {}, angle error: {}".format(
                    distanceError, oriDiscrepancy
                )
            )
            print("\nResulting in inputs:{},{},{}\n\n".format(inputT, inputLx, inputLy))


# SMALL AIDING FUNCTIONS


def interpret(data):
    """
    inpret message

    Put message from plugins in an easier to use list format

    Args:
        data string

    Returns:
        list

    Raises:
        None as of now
    """
    msg = str(data.data)
    msgList = msg.split()
    return msgList


def interpretCom(msg):
    return msg.split("\n")


def bin2Rad(a):
    return (a + 1) * pi


def orientNormal(ori):  # normalized and in radians
    """
    orientnormal

    Angle grom the gazebo simulation is oriented -180 from what is common, and is in radians

    Args:
        gazebo generated jaw angle in radians

    Returns:
        normalized jaw angle i degrees

    Raises:
        None as of now
    """
    return (ori + 180) * math.pi / 180


def getWantedAngle(posx, posy):  # normalized and in radians
    """
    Get jaw angle

    Get the jaw angle from a 2d positional angle discre

    Args:
        two lists a,b

    Returns:
        difference list c

    Raises:
        None as of now
    """
    wanted = math.atan2(posy, posx)
    return (
        wanted
    )  # +math.pi #range correction, making it between 0 and 2pi (moved to main)


def getBusPass():
    """
    get bus pass

    Can be implemented /expanded to involve whether the bus is correct considering route and directi

    Args:
        unimplemented

    Returns:
        True

    Raises:
        None as of now
    """
    return True
```

```python


def posDiscrep(a, b):
    """
    vector from 2 positions

    This returns discrepancy/difference between positions

    Args:
        two lists a,b

    Returns:
        difference list c

    Raises:
        None as of now
    """
    if len(a) == 2:
        a.append(0)
    if len(b) == 2:
        b.append(0)
    return [b[0] - a[0], b[1] - a[1], b[2] - a[2]]


def d3Distance(discrep):
    """
    Norm 2

    position to the corresponding 3D distance

    Args:
        position discrepancy in a list

    Returns:
        scalar distance

    Raises:
        None as of now
    """
    plane = math.sqrt(discrep[0] ** 2 + discrep[1] ** 2)
    return math.sqrt(plane ** 2 + discrep[2] ** 2)


def quat_to_euler(x, y, z, w):
    """
    This returns euler angles

    Quaternion to euler transformation

    Args:
        x,y,z,w quat in list

    Returns:
        euler list of angles

    Raises:
        None as of now
    """
    t0 = +2.0 * (w * x + y * z)
    t1 = +1.0 - 2.0 * (x * x + y * y)
    X = math.degrees(math.atan2(t0, t1))

    t2 = +2.0 * (w * y - z * x)
    t2 = +1.0 if t2 > +1.0 else t2
    t2 = -1.0 if t2 < -1.0 else t2
    Y = math.degrees(math.asin(t2))

    t3 = +2.0 * (w * z + x * y)
    t4 = +1.0 - 2.0 * (y * y + z * z)
    Z = math.degrees(math.atan2(t3, t4))
    return [X, Y, Z]


# MAIN THREAD CONTROL


def listener():
    rospy.init_node("droneRun", anonymous=True)
    pubCom = rospy.Publisher("tum_ardrone/com", String, queue_size=1)
    ##
    rospy.Subscriber(
        "/businteract/cmd", String, my_drone.callbackBusevent, callback_args=pubCom
    )
    rospy.Subscriber("/ground_truth/state", Odometry, my_drone.gps)
    rospy.Subscriber(
        "/visualization_marker", Marker, my_drone.callbackMarker, callback_args=pubCom
    )
    rospy.Subscriber("/ardrone/imu", Imu, my_drone.callbackImu)
    rospy.Subscriber("/tum_ardrone/pose", PoseStamped, my_drone.callbackTumpose)
    rospy.Subscriber(
        "/ardrone/image_raw", Image, my_drone.continouslyCommence, callback_args=pubCom
    )
    ##
    rospy.spin()


my_drone = DroneState()

if __name__ == "__main__":
    talkative = False
    try:
        listener()
        print("Cycle")
    except rospy.ROSInterruptException:
        pass
```

.3.7 Plugin control

```
#!/usr/bin/env python
from __future__ import division
import rospy
from std_msgs.msg import String
from string import split
from dynamic_reconfigure.msg import ConfigDescription
import re
from sensor_msgs.msg import Imu
from geometry_msgs.msg import Vector3
from std_msgs.msg import Empty
import math
from geometry_msgs.msg import Vector3Stamped
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from visualization_msgs.msg import Marker
from octomap_msgs.msg import Octomap
from nav_msgs.msg import OccupancyGrid
from darknet_ros_msgs.msg import BoundingBoxes
from opencv_apps.msg import FlowArrayStamped
from sensor_msgs.msg import Image
import opencv_apps
import cv_bridge
from cv_bridge import CvBridge
import time
from std_srvs.srv import Empty


class BusEncounter:
    def __init__(self):
        # STATE EVEN VARIABLES
        self.greenConfirmed = False
        self.busConfirmed = None
        self.flowState = 0
        self.cloudcenter = [0, 0]

        # ALGORITHM ORDER CONTROL
        self.algoOrder = [0, 0, 0]

        # DEBUG
        self.talkative = False

    def encounterControl(self, pubBusCmd):  # started every frame
        """
        State machine

        Started every frame, it changes state  and parts of bus encounter using the other plugins

        Args:
            pubBusCmd: bus even channel

        Returns:
            None, only changes state

        Raises:
            None as of now
        """
        if self.talkative:
            print(
                "3. encountercontrol! {}, sending it to callbusevent".format(
                    self.cloudcenter[0]
                )
            )

        if not (self.algoOrder[0] * self.algoOrder[1]):
            if self.talkative:
                print("-conditions not met to continue 3")
            return
        elif not (self.algoOrder[2]):
            if self.talkative:
                print("-conditions met to continue 3")
            spawnPoints = rospy.ServiceProxy("lk_flow/initialize_points", Empty)
            spawnPoints()

        self.algoOrder[2] = 1

        msg = "LOShere: {} {}".format(self.cloudcenter[0], self.cloudcenter[1])
        pubBusCmd.publish(msg)

    def callbackBusCmd(self, data, pubBusCmd):
        """
        Bus Cmd message Postbox

        Event messages are handled

        Args:
            data: the message
            pubBusCmd: Channel for event communication

        Returns:
            None, only changes state

        Raises:
            None as of now
        """
        msg = interpret(data)
        if self.talkative:
            print("2. cmdCallback sees that {} msg gives {}".format(msg[0], msg[1]))

        if msg[0] == "-greenSpotted":
            if self.algoOrder[0] and self.algoOrder[1]:
                return
            self.algoOrder[0] = 1

            if self.algoOrder[0] and self.algoOrder[1]:
                return

        elif msg[0] == "-busSpotted:":

            self.busConfirmed = [msg[1], msg[2], msg[3], msg[4]]
        elif msg[0] == "-Flowcloud:":

            self.cloudcenter = [msg[1], msg[2]]
            if self.talkative:
                print("CLOUDCENTER: {}".format(self.cloudcenter))
            if self.talkative:
                print("Update on cloudcenter is well {}".format(self.cloudcenter))

    def callbackFlow(self, data, pubBusCmd):
        """
        flowCloudCallback

        LK flow produces a list of point with velocities and positions

        Args:
            data: flow list
            pubBusCmd: bus even channel

        Returns:
            None, only changes state

        Raises:
            None as of now
        """
        if self.talkative:
            print("1, but will I continue?: {}".format(self.busConfirmed))
        if not self.busConfirmed:
            return  # As LK is much faster than Yolo, this point is often arrived at.

        trackedPoints = len(data.flow)
        direc = 0

        flowpointInside = []
        for i in data.flow:
            if isInBB(i.point, self.busConfirmed):
                flowpointInside.append(i)
        if not (flowpointInside):
            return
        totalVel, direction, avg = getPointCloudVel(flowpointInside)  # the one
        if self.talkative:
            print(
                "1. {}/{}, says: cloudvel at {} is: {} in direction {}".format(
                    len(flowpointInside),
                    trackedPoints,
                    avg,
                    round(totalVel, 2),
                    direction,
                )
            )
        msg = "Flowcloud: {} {}".format(avg[0], avg[1])
        pubBusCmd.publish(msg)


def getSimplePointVel(flowpoint):
    """
    Get point velocity

    Get the velocity of one points 2d movement

    Args:
        flowpoint LK object. Has a velocity member variable

    Returns:
        totalVel: the total velocity scalar value
        ratio: ratio of totalVel in a sideways direction vs up/down.

    Raises:
        None as of now
    """

    vel = flowpoint.velocity
    totalVel = math.sqrt(vel.x ** 2 + vel.y ** 2)
    if vel.y < 0.0001:
        vel.y = 0.1
    ratio = abs(vel.x) / abs(vel.y)

    direction = "still"
    if totalVel > 5:
        if ratio > 80:
            direction = "sideways"
        elif ratio < 0.01:
            direction = "updown"

    return totalVel, ratio


def getPointVel(flowpoint):
    """
    Gets all velocities (can be mixed with getSimplePoint)

    Args:
        flowpoint LK object. Object with velocity member variable

    Returns:
        2D vel object with x and y
        totalVel: scalar
        ratio: ratio of totalVel in a sideways direction vs up/down.

    Raises:
        None as of now
    """
    vel = flowpoint.velocity
    totalVel = math.sqrt(vel.x ** 2 + vel.y ** 2)
    if vel.y < 0.0001:
        vel.y = 0.1  # float zero division
    ratio = abs(vel.x) / abs(vel.y)
    return vel, totalVel, ratio


def getPointCloudVel(flowpoints):
    """
    GetPointCloudVeloxcity

    Gets the average velocity of the entire point cloud using all points

    Args:
        flowpoints LK object list. Has velocity member variables

    Returns:
        cloudTotalVel: the total velocity scalar value
        direction: String with updown vs sideways, as in the motion of PC
        avg: the average point position, used to estimate center of mass of PC

    Raises:
        None as of now
    """
    vels = [0, 0]
    avgPoints = [0, 0]
    totalVels = 0
    ratiosum = 0
    for i in flowpoints:
        pointVel, totalVel, pointRatio = getPointVel(i)
        vels[0] += pointVel.x
        vels[1] += pointVel.y
        totalVels += totalVel
        ratiosum += pointRatio
        avgPoints[0] += i.point.x
        avgPoints[1] += i.point.y
    length = len(flowpoints)
    if not length:
        return
    cloudVel = [vels[0] / length, vels[1] / length]
    cloudTotalVel = totalVels / length
    cloudRatio = ratiosum / length
    avg = [avgPoints[0] / length, avgPoints[1] / length]
    direction = "still"
    if cloudTotalVel > 5:
        if cloudRatio > 80:
            direction = "sideways"
        elif cloudRatio < 0.01:
            direction = "updown"
    return cloudTotalVel, direction, avg


def callbackBB(data, pubBusCmd):
    """
        Callback BB

        Save the bounding box outputted by Yolo if its certainty of a bus is beyond a certain thresh

        Args:
            Data: bounding box object on the form [xmin ymin xmax ymax]
            pubBusCmd: channel for bus event communication

        Returns:

        Raises:
            None as of now
    """
    if (not (newEncounter.algoOrder[0])) or newEncounter.algoOrder[1]:
        return

    numberOfObjects = len(data.bounding_boxes)
    for obj in data.bounding_boxes:
        if obj.Class == "bus" and obj.probability > 0.55:
            msg = (
                "busSpotted: "
                + str(obj.xmin)
                + " "
                + str(obj.ymin)
                + " "
                + str(obj.xmax)
                + " "
                + str(obj.ymax)
            )
            pubBusCmd.publish(msg)
            newEncounter.algoOrder[1] = 1


def isInBB(point, bb):
    """
    isin BB

    Given a 2 position, it checks if it is within the given bounding box

    Args:
        point: a list of 2 points
        bb: a list with the 4 corner points of the bounding box

    Returns:

    Raises:
        None as of now
    """
    expansion = 10  # pixels of area expanded larger
    bb0 = float(bb[0])
    bb1 = float(bb[1])
    bb2 = float(bb[2])
    bb3 = float(bb[3])
    if (
        (point.x > (bb0 - expansion))
        and ((point.x - expansion) < bb2)
        and (point.y > (bb1 - expansion))
        and ((point.y - expansion) < bb3)
    ):
        return True
    return False


def callbackRepublish(data, args):
    """
    Callback republish

    Republishes the image, starts encounter control in the framerate of the camera

    Args:
        Data: the image
        args[0]: image publishing channel
        args[1]: bus event channel

    Returns:
        None
    Raises:
        None as of now
    """
    if False:
        print("algorder in callbackim, {} ".format(newEncounter.algoOrder))

    pubImage = args[0]
    pubBusCmd = args[1]

    pubImage.publish(data)

    newEncounter.encounterControl(pubBusCmd)


def callbackImage(image, pubBusCmd):  # checks how green it is
    """
    Callback image

    Republishes the image, starts encounter control in the framerate of the camera

    Args:
        Data: the image
        args[0]: image publishing channel
        args[1]: bus event channel
    """
    bridge = CvBridge()
    img = bridge.imgmsg_to_cv2(image, desired_encoding="passthrough")
    totalElements = len(img) * len(img[0])
    elemsum = 0
    avgPos = [0, 0]
    for i in range(len(img)):
        for j in range(len(img[i])):
            if img[i][j]:
                avgPos[0] += i
                avgPos[1] += j
                elemsum += 1
    avgPos[0] = avgPos[0] / image.height
    avgPos[1] = avgPos[1] / image.width
    ratio = 100 * elemsum / totalElements

    if elemsum > 7:
        msg = "greenSpotted {} {}".format(avgPos[0], avgPos[1])
        pubBusCmd.publish(msg)


def interpret(data):
    msg = str(data.data)
    msgList = msg.split()
    return msgList


def listener():
    rospy.init_node("greenCheck", anonymous=True)
    pubBusCmd = rospy.Publisher("/businteract/cmd", String, queue_size=1)
    pubImage = rospy.Publisher("/businteract/image", Image, queue_size=1)

    ##
    imgCallArgs = (pubImage, pubBusCmd)
    rospy.Subscriber(
        "/ardrone/image_raw", Image, callbackRepublish, callback_args=(imgCallArgs)
    )
    rospy.Subscriber(
        "/rgb_color_filter/image", Image, callbackImage, callback_args=(pubBusCmd)
    )
    rospy.Subscriber(
        "/darknet_ros/bounding_boxes",
        BoundingBoxes,
        callbackBB,
        callback_args=(pubBusCmd),
    )
    rospy.Subscriber(
        "/lk_flow/flows",
        FlowArrayStamped,
        newEncounter.callbackFlow,
        callback_args=(pubBusCmd),
    )
    rospy.Subscriber(
        "/businteract/cmd",
        String,
        newEncounter.callbackBusCmd,
        callback_args=(pubBusCmd),
    )

    print("No subscription errors")
    rospy.spin()


newEncounter = BusEncounter()

if __name__ == "__main__":
    try:

        listener()
    except rospy.ROSInterruptException:
        pass
```

Håkon Elfving

NTNU
Kunnskap for en bedre verden