Abstract

Drones are quickly rising in popularity for carrying out missions in environments where humans have difficulties performing tasks manually. They excel in both safety and efficiency. However, accuracy in the landing phase for autonomous drones is a difficult challenge. This thesis presents the current progress in the continued development of a system where an unmanned aerial vehicle (UAV) is to autonomously land on and pick up a micro underwater glider (MUG) using computer vision and DUNE: unified navigation environment.

A system implementation in DUNE with guidance methods and computer vision models which the system rely on are developed and explained. The software and hardware necessary to reproduce the methods used in this thesis for simulation and field testing are presented.

Developed algorithms and methods are carried out by a miniature drone in a small scale environment and ran through simulations with DUNE. The simulation provides a full implementation test in a simulated environment governed by DUNE. These tests are used as validation of the system before executing a field test. Results from the miniature drone and simulations are presented and discussed in the thesis.

Field tests were conducted on land with a quadcopter and a fiducial marker on the ground representing the MUG. The implemented system was able to detect the marker at an altitude of 5m and descend to an altitude of 40cm while maintaining the marker within the camera view. The attempted landing maneuvers after closing in on the marker was not able to keep the drone on the ground, but horizontal position error was kept within 20cm before attempting to land. The results from the field test are presented, discussed and compared with the results of the miniature drone and simulations.

Due to the national corona virus lockdown initiated 12.03.2020, field testing with the drone intended to be used for this project was not performed.

Sammendrag

Autonome droner er i økende grad brukt til å utføre oppdrag som er vanskelig for mennesker å utføre selv. Droner tilbyr høy sikkerhet og effektivitet, men nøyaktighet i landingsfasen er en vanskelig utfordring. Denne masteroppgaven presenterer statusen av et kontinuerlig arbeid hvor målet er å lage programvare for en unmanned aerial vehicle (UAV) slik at den kan autonomt lande på en micro underwater glider (MUG) og plukke den opp ved hjelp av datasyn og DUNE: unified navigation environment.

Programvaren og maskinvaren som er nødvendig for å reprodusere metodene i masteroppgaven for simulering og felttesting blir presentert. Navigasjonsmetoder og datasynalgoritmer som programvaren er basert på blir utviklet og forklart.

Utviklet algoritmer og metoder blir utført på en miniatyr drone, og gjennom simulering med DUNE. I simuleringen blir den fulle implementasjonen av DUNE testet. Testene blir brukt som gradvis validering før en felttest med en større drone blir utført. Resultatene fra testene med miniatyrdronen og simuleringene blir presentert og diskutert i masteropeggaven.

Felttester ble utført på land med et quadcopter og med en lapp som inneholder posisjonsreferanser og som representerer en MUG. Systemet som ble laget var i stand til å gjenkjenne lappen i en høyde på 5m og holde lappen innenfor bildet samtidig som den senket høyden ned til 40cm. Forsøkte landinger klarte ikke å holde dronen på bakken, men horisontale feil ble holdt innenfor 20cm før landing ble forsøkt. Resultatene blir drøftet og sammenlinget med resultatene fra miniatyrdronen og simuleringene.

Grunnet nedstegninger forårsaket av koronautbruddet ble felttestene utført av en annen drone enn den som masteroppgaven legger opp til å bruke.

Preface

The work presented in this thesis concludes my master's degree in Cybernetics and Robotics at Norwegian University of Science and Technology (NTNU). The thesis represents the continued work on the specialization project, written by the author and Aleksander Asp, focused on establishing a quick and iterative testing platform for this thesis using a miniature drone.

I would like to thank my supervisors; main supervisor Tor Arne Johansen and co-supervisor Martin L. Sollie. Meetings and discussions with Johansen helped define my goals for the thesis, and his feedback throughout the semester has helped keep the course steady. I would like to thank Artur Piotr Zolich who has provided invaluable input on the software stack and performed field testing on my behalf. I would also like to thank Alexander Asp for his contributions to the specialization project and for the constructive discussions along the way. Lastly, I would like to thank Ove Eldøy for providing the inspiration which lead me to pursue this degree.

Table of Contents

List of Tables

List of Figures

Abbreviations

API	=	Application programming interface
AUV	=	Autonomous underwater vehicle
AVC	=	Advanced video coding
BBB	=	BeagleBone Black
CNN	=	Convolutional neural network
CV	=	Computer vision
DOF	=	Degrees of freedom
DUNE	=	Unified navigation environment
GPS	=	Global positioning system
HSV	=	Hue-Saturaion-Value
IMC	=	Intermodule communication
LOS	=	Line-of-sight
LSTS	=	Underwater Systems and Technology Laboratory, (Laboratório de Sistemas e Tecnologia Subaquática)
MCU	=	Micro-controller unit
MUG	=	Micro underwater glider
NED	=	North-East-Down
NTNU	=	Norwegian University of Science and Technology
RGB	=	Red-Green-Blue
HSV	=	Hue-Saturation-Value
ROV	=	Remotely operated vehicle
R-CNN	=	Region based convolutional neural network
UAV	=	Unmanned Aerial Vehicle
UDP	=	User datagram protocol
USARSim	=	Unified System for Automation and Robot Simulation
USV	=	Unmanned surface vehicle
VTOL	=	Vertical take-off and landing
HITL	=	Hardware in the loop
HIL	=	Hardware in the loop
SITL	=	Software in the loop
SIL	=	Software in the loop
EPM	=	Electropermanent magnet

Chapter 1

Introduction

The scope of this thesis is to create a robust system for a multi-rotor UAV to identify, approach and pickup a micro underwater glider (MUG) at the ocean surface. This thesis is part of a larger, international project called Oasys. Oasys aims at creating fully autonomous systems for reducing cost of observing and monitoring oceans. The motivation behind the Oasys project is described in full on their website [9] and is summarized as follows:

”One of the barriers towards a better understanding and sustainable development of marine related economic activity is the high cost associated with ocean observing systems. Autonomous robotic systems are steadily revolutionizing the way we obtain data and interact with the ocean. However most of existing autonomous systems still require the involvement of manned missions in the deployment/recovery phases which represents a high percentage of the total operational costs”

1.1 Problem Overview

The UAV takes off from an unmanned surface vehicle (USV) at sea where the UAV and the MUG docks. The MUG will drift from the USV and the objective is to locate, pickup the MUG and return it to the USV. There is no direct communication between the MUG and the UAV which means that the drone has to initially search for the MUG in the ocean. To help locate the MUG at the ocean surface, a fiducial marker of type ArUco is attached to the fin of the MUG. The specific marker used is designed to be highly visible at the ocean surface making it easily detectable. Additionally, this marker aids the UAV in approaching the MUG for pickup as necessary accurate pose estimates can be gathered through computer vision tracking methods. The task can be divided into four phases.

1. Initial MUG localization
2. High altitude approach.

3. Low altitude approach.
4. Pickup and return.

This thesis main focus is the approach phases of the project. Robust algorithms for tracking the MUG is needed, especially high accuracy pose estimations at lower altitudes before locking the MUG to the UAV with an electropermanent magnet. A system will need to be developed in DUNE for controlling all the phases of the flight. An explanation is given for each phase below.

1.1.1 Initial MUG Localization

The MUG will share its GPS location over radio network, but this measurement is expected to be imprecise. There is however no guarantee that a common communication link between the MUG and the drone exists. Therefor some localization method must be performed by the UAV to find the initial location of the MUG. The localization is finished when the MUG is identified in the camera view. Computer vision algorithms must be created for identifying the MUG based on generic visual properties available at high altitudes such as shape and color.

1.1.2 High Altitude Approach

The marker attached to MUG provides pose estimates, but requires a certain resolution of the marker not available at higher altitudes. The main objective is to descend while maintaining the the marker within view until the fiducial details of the marker is visible. Navigation methods based on the camera view location of the MUG will have to be developed. The altitude the drone has to descend to heavily depends on the visual noise and camera resolution. Higher camera resolutions can be achieved by hardware, but this comes at a cost of extra processing power.

1.1.3 Low Altitude Approach

At the stage where the marker details are sufficiently visible, computer vision algorithms for performing pose estimation of the ArUco markers must be developed. Also guidance methods based on these pose estimates is needed.

The low altitude approach will be stabler with higher pose estimate frequency. Therefor the robustness of the noise reduction, filtering and segmentation must be optimized to provide optimal conditions for a estimate to happen at every camera frame.

1.1.4 Pickup and return

For the drone to be able to pick up the MUG precisely, data about the ocean state has to be estimated and compensated for. Ocean currents will induce a constant drift to the MUG and waves will induce temporal movements. Additionally, wind will disturb the motion of the drone. All these factors have to be regarded in order to achieve a robust and reliable method for picking up the MUG in the expected environment.

The drone has an electropermanent magnet for securing the MUG as a payload. Additional difficulties arise when the camera used by the drone is not able to keep the MUG visible while at close enough contact to lock the MUG to the magnet. Different methods with hardware setup will need to be tested to ensure measurements exists when closing in on the MUG, or a good estimation process. Ultimately, this means there will be guesses at when to activate the magnets and a method to identify a successful tethering must be used before returning to the USV.

1.2 Outline

Chapter 2 - Literature Review: This chapter discusses previous work in the field related to this project.
Chapter 3 - Basic Theory: The chapter presents most of the relevant theory for the main ideas and concepts used later in the project. This includes segmentation, Kalman filter, computer vision, and also the dynamics of the system model.
Chapter 4 - System Overview: Chapter 4 describes the hardware and the main software components used in the project. This includes DUNE, ArduPilot, drones, cameras, the ArUco markers and more.
Chapter 5 - Implementation: The chapter describes the implementation details for each of the modules needed to complete the full system. This covers both the DUNE and the miniature drone implementation.
Chapter 6 - Experiments and Results: The penultimate chapter presents the experiments performed and the data gathered. The experiments includes miniature drone flights, simulations and field tests.
Chapter 7 - Discussion and Conclusion The final chapter contains a small discussion, the conclusions drawn and future work.

Chapter 2

Literature Review

Generation of fiducial marker dictionaries using Mixed Integer Linear Programming[26] and *Speeded Up Detection of Squared Fiducial Markers*[31] is the origin behind the ArUco markers that will be used as fiducial marker canditates in this thesis. The paper goes into the implementation details of the ArUco markers. The markers are designed to provide fast and accurate fiducial data with occlusion tolerance for AR applications. These markers prove potential for the fast and high accuracy estimates of the MUG position required for successfully executing a pick up mission.

Vision-Based Landing of a Simulated Unmanned Aerial Vehicle with Fast Reinforcement Learning [32] looks into one of the critical steps before testing computer vision systems and control systems in the field, that is, simulation. In the paper, a vision-based landing approach for autonomous UAV's is proposed, using fast reinforcement learning. This approach is tested in an extended version of the USARSim (Unified System for Automation and Robot Simulation) environment with a simulated quadrocopter [32]. In the simulation, the quadrocopter has a camera fixed at the center of its belly, and the target landing site is a fully black circle surrounded by circles in a range of gray. The approach makes use of the OpenCV framework to detect the target and the Least-Square Policy Iteration as the reinforcement learning method. In the event that ArUco markers are found to be sub-optimal, applying methods and concepts from this paper is an alternative.

Vision-Based Autonomous Landing of a Quadrotor on the Perturbed Deck of an Unmanned Surface Vehicle [29] also uses a fiducial marker, placed on the platform of the unmanned surface vehicle (USV) to accommodate the task of finding the USV's relative position and pose. An extended Kalman (EKF) filter is used to estimate the current position of the USV, to compensate for potential temporary loss of marker detection. The EKF provides accurate enough estimations, that in combination with odometry, this method is found to be sufficient and applicable in poor weather conditions and in the absence of a global positioning system. A relatively small USV is used in this paper, however, it is still larger than the quadrocopter itself, demanding less accuracy than what is required in this project.

Autonomous Landing of a UAV on a Moving Platform Using Model Predictive Control [23] presents an autonomous landing method for micro UAV's to land on moving targets in the presence of uncertainties and disturbances. For optimal localization of the moving platform, a Kalman filter is implemented, and model predictive control is developed as part of the system architecture. The computer vision system makes use of an AprilTag, similar to the before mentioned ArUco codes, to estimate the relative position and pose of the platform with respect to the camera. The simulation results in this paper demonstrate an autonomous landing on a platform travelling up to $12m/s$ with an error of less than $37cm$ from the center of the platform. The use of markers in computer vision systems to estimate camera pose, as well as landing on a moving target are highly relevant tasks for this project.

In *Autonomous Maritime Landings for Low-cost VTOL Aerial Vehicles* [28] an architecture that negates sensor limitations and allows for precise pose estimation, even in the presence of wind disturbances, is proposed. The final landing method performs landing maneuvers in the body-fixed reference frame to nullify poor estimation accuracy caused by noisy measurements from GPS and magnetometers. The total system consists of three different stages. The initial stage calculates an intersection point based on current positions of both vehicles and the estimated velocity of the marine vessel. The UAV flies to this point autonomously using constant heading control. Note that the intersection point can be updated repeatedly to compensate for drift and varying vessel velocity. Stage two begins when the UAV is in vicinity of the marine vessel, and starts its search for an AprilTag. Once the AprilTag has been located, the control system switches from inertial control to relative control, using the body frame as reference and initiates the final landing sequence. Several of these stages relate to the challenges in this project, especially landing on a marine vessel. However, a sizeable boat was used as the marine vessel in this paper [28], with a significantly larger landing area than that of the micro underwater glider.

Multirotor pickup of object in the sea [27] presents a system setup for using onboard computing, computer vision and radar as basis for picking up objects a sea. The detected position with computer vision and the radar measurements are triangulated to estimate the position of the object. The object is modelled with an constant velocity Kalman filter to compensate for ocean currents. The implementation is integrated with DUNE, and uses NED navigation to fulfill the objective. This paper share similar objectives as this one, but this thesis will focus on body fixed frame navigation with fiducial markers as source for high accuracy estimates. The DUNE implementation and design is very relevant to this thesis.

Chapter 3

Basic Theory

3.1 Reference frames and Transformations

3.1.1 Reference frames

A coordinate system intended to express an object's position is generally made up of two or three axes, depending on the number of dimensions the object can move in and which of these are considered of interest. The position of the object can then be uniquely expressed using the same number of coordinates as the number of axes.

Coordinate systems can have their origin fixed in different locations. A reference frame defines the location and orientation of a coordinate system. For small aerial vehicles it is a common convention to adhere to the North-East-Down (NED) reference frame as an inertial frame. An inertial frame is defined as reference frame in which the object does not accelerate when there is zero net-force acting upon said object [24]. In the NED-frame, the x-axis points towards true north, the y-axis east and the z-axis down towards earth to complete a right-hand coordinate system. The origin of the NED frame is a starting point of the earth's surface.

For vehicles operating in a local area, that is when longitude and latitude can be approximated as constant, flat Earth navigation is used. This assumes the North-East axes forms a constant tangential surface plane at the origin. In this case the NED reference frame can be considered inertial.

In the case of controllable moving bodies, e.g. UAVs, it is also common practice to define a body reference frame. In the body frame the origin of the coordinate system is typically defined as either the center of mass or the geometric center of the object. For aerial vehicles it is common that the x-axis points out of the nose of the vehicle, along the longitudinal axis of the plane, the y-axis points out of the right wing/side, lateral axis, and the z-axis out of the belly/bottom of the aircraft.

A final common frame for aerial vehicles is the vehicle frame. The vehicle frame is simply a NED coordinate system with the origin fixed in the geometric center or mass center of the vehicle. With several different reference frames, a transformation from a point in one frame to another is needed.

3.1.2 Rotations and Transformations

The relationship between coordinate frames can be expressed using a composite series of rotational matrices and translation. The following methods are described using Euler angles notation of type roll (ϕ), pitch (θ), yaw (ψ), which are commonly used to describe motion rigid body vehicles moving freely. There are 3 fixed axis rotations defined as simple rotations which is a rotation about a single fixed axis. The simple rotation matrix for each axis is defined in equation 3.1.

$$\mathbf{R}_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \tag{3.1a}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{3.1b}$$

$$\mathbf{R}_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1c}$$

If two frames share an origin, e.g. frame a and frame b, but the axis are at different orientations, a composite series of simple rotations, $\mathbf{R}_b^a$ can be used to express a point with reference to frame B, $\mathbf{p}^b$, in frame A, $\mathbf{p}^a$. The resulting rotation matrix is defined as

$$\mathbf{R}_b^a = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi)$$

The angle parameters are defined as the relative angle of the axis in A with regards to the axis in B. The point $\mathbf{p}^a$ can then be expressed as

$$\mathbf{p}^a = \mathbf{R}_b^a\mathbf{p}^b$$

In the case where the origin of the frames are not shared, a translation has to be included in addition to the rotation. The translation can be done in sequence after a rotation, but alternatively the translation and rotation matrices can be combined into what is defined as a homogenous transformation matrix. A homogenous transformation matrix fully describes the position and orientation of a coordinate frame with respect to a reference frame.

A transformation matrix $\mathbf{T}_b^a$ used to express a point in frame A with respect to a point expressed in reference frame B.

$$\mathbf{T}_b^a = \begin{bmatrix} \mathbf{R}_b^a & \mathbf{r}_{ab}^a \\ \mathbf{0}_{1x3} & 1 \end{bmatrix} \tag{3.2}$$

where $\mathbf{r}_{ab}^a$ is defined as the position of frame B relative to frame A, expressed in frame A coordinates.

As the transformation matrix is homogenous and 4x4, the position vector has to be augmented to an homogenous vector to compute the transformation. This is done by augmenting a value of 1.

$$\mathbf{p}^b = \begin{bmatrix} x_b & y_b & z_b \end{bmatrix}^\top \Rightarrow \begin{bmatrix} x_b & y_b & z_b & 1 \end{bmatrix}^\top \tag{3.3}$$

With this, any point expressed in B can be expressed in A by the following

$$\mathbf{p}^a = \mathbf{T}_b^a \mathbf{p}^b \tag{3.4}$$

A point can also be expressed with regards to another frame by a series of intermediate relative transformations similar to the composite simple rotations.

$$\mathbf{p}^a = \mathbf{T}_b^a \mathbf{T}_c^b \mathbf{p}^c \tag{3.5}$$

3.2 Computer Vision

3.2.1 Camera Model

When a picture is taken with a camera, the real world 3D scene is projected into a 2D image. The rays of light that enters the camera lens, followed by an aperture, then hits the surface of a light sensor which excites an area (pixel). The specific pixel that gets excited on the surface of the light sensor is determined by the direction of the light ray. Certain methods in computer vision relies on estimating the mathematical relationship between the 3D scene and its 2D image projection. Thus, a camera model is needed to approximate this process. The most commonly used camera model is the pinhole model.

The pinhole model represents an single aperture that lets light through a barrier. The projected object is then flipped onto the film on the other side of the barrier. To find the mapping of a scene to the image plane, the aperture is defined to be the size of a single point. This causes there to be no light saturation and each point in the scene will map to an unique point on the film. If the aperture were to be larger, more light rays would hit the same point on the film, increasing brightness at the cost of making the projection blurrier.

Modern camera uses lenses to compensate for the brightness versus sharpness factor. The lenses also causes all the light rays traveling parallel to the optical axis to be focused to a single point defined as the focal point. The distance from the center of the lens to the focal point is defined as the focal length f. The augmentation of the ideal pinhole camera with a lens is commonly done and thus the effect of the lens has to be included in the model. The location of the pinhole aperture projection in into the film is defined as the principal point c.

In projective geometry, this information can be combined to make up the intrinsic parameters, K, of the camera model and is expressed as the following matrix

$$K = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

Where f_x and f_y are the focal lengths, s is the skew coefficient, which is non-zero when the image axes are not perpendicular, and x_0 and y_0 is the principal point coordinates. All values are given in pixel units.

As well as the intrinsic parameters, there are extrinsic camera parameters which describes the pose of the camera with respect to a world frame. The extrinsic camera parameters consists of a rotation R and a translation t.

Together the intrinsic and the extrinsic camera parameters make up the camera model and can be used to get the projection from a 3D world point $\begin{bmatrix} X & Y & Z \end{bmatrix}$ onto the image plane $\begin{bmatrix} x & y \end{bmatrix}$ by using the following relationship

$$W \begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} X & Y & Z \end{bmatrix} P \tag{3.7}$$

Where W is defined as a scalar value used for scaling, and the camera matrix P defines the camera model and is given by

$$P = \begin{bmatrix} R \\ t \end{bmatrix} K \tag{3.8}$$

With a known camera matrix, K, and known object models in a scene, different methods can be utilized to determine the pose of the camera with respect to the objects.

3.2.2 Calibration

As mentioned in the previous section, the pinhole model is a mathematical representation of a camera. The process of determining the cameras intrinsic and extrinsic parameters such that the pinhole model best approximates the camera is called camera calibration. A camera model is not perfect and the error between the estimated projection by using (3.7) and the actual projection is defined as the reprojection error. To find the best camera model parameters, calibration techniques tries to minimize this reprojection error across multiple calibration images.

The general procedure for identifying the parameters involve capturing images of a world scene where an object with known 3D keypoints exists. If there exists an object with known 3D keypoints in a scene captured by the camera, and with the corresponding projection into the 2D image, the camera matrix can be determined by using (3.7). Frameworks which provides computer vision algorithms, such as Matlab and OpenCV, has methods for estimating the best camera matrix across multiple calibration images. A commonly used method for estimating the intrinsic camera parameters without knowing the extrinsic parameters beforehand is the checkerboard calibration method.

3.2.3 Pose Estimation

In section 3.2.1 extrinsic camera parameters were explained as the orientation and translation of a camera relative to a scene, expressed in a global reference frame. With known intrinsic camera parameters and known 3D keypoints such as fiducial objects in the scene, the relationship of (3.7) can be used to calculate the extrinsic parameters of the camera and thus give an estimate of the pose of the camera relative to an object in the scene. The accuracy of the pose estimation is heavily dependant on the reprojection error of the camera model, identified by the calibration process. Several pose estimation methods exists, but the most relevant method for this thesis will be the ArUco square marker pose estimation [31]. Alternatively, April Tags is another popular choice for fiducial markers that provides pose estimation.

With a marker present and visible in the image plane, the corners of the ArUco marker offers planar pose estimators. The binary codification of the ArUco markers is used to determine the orientation of the marker as each corner in the marker is uniquely identified by the coding. With the planar pose estimator and the orientation of the ArUco marker, the relative pose of the camera can be found.

3.2.4 Object Detection

Object detection is a computer vision task where the goal is to classify and locate all instances of known object classes in an image. With an image classifying network as a starting point there are several ways to achieve object detection.

Early approaches mainly consisted of the regression method and the sliding window method. The common trait for these methods, and later ones, is that the network outputs bounding boxes for each detected object, in addition to the classification. A bounding box is described by four values. The first two define the top left corner of the box (x- and y-value) and the last two define the height and width of the box. The regression method computes the difference between the network's predicted bounding box and the true bounding box, known as the ground truth box and uses this error in a regression analysis. This method works fine with a fixed number of objects in the image, but falls short in the need of variable sized output caused by variable number of objects in the input images.

The sliding window method entails the application of a CNN to many different crops of the image. The CNN classifies each crop as either *background* or one of the object classes, with a level of certainty. The biggest problem with this method is its computational cost, as there is a large amount of different positions, scales and aspect ratios the CNN must consider. The solution to this issue, which gave birth to several modern approaches, is the concept of region proposals.

Today's modern object detectors are either based on two-stage methods, which incorporate region proposals, or single shot methods. An example of a two-stage method is the *Faster R-CNN* method, where R-CNN means it is a region based convolutional neural network. The Faster R-CNN method uses a region proposal network, which takes the post convolution feature map as input, to predict regions of interest. The main improvement in this method, compared to its predecessor *Fast R-CNN*, is that it uses a CNN in the region proposal network, greatly increasing the speed of region proposals and thereby eliminating what was the current bottleneck for performance.

3.2.5 Segmentation

The idea of image segmentation is to partition the input image in a manner that makes it more meaningful from an analytical standpoint. A common use of segmentation is to locate objects or contours by detecting edges and corners. In a segmented image, areas that have the same label or colour, share at least one similar characteristic.

There are many different methods available to achieve segmentation, one of the simplest being thresholding. The most basic threshold method takes a gray-scale image as an input and outputs a black and white binary image by transforming each pixel to either black or white, depending on the threshold value.

3.3 Modeling and Control

The following sections in Modeling and Control are reused from the specialization project [21].

3.3.1 Model Dynamics

When modeling the dynamics of the system, the multirotor is assumed to be a rigid body, with center of mass in the geometric center and with six degrees-of-freedom (DOF). The translational equations of motion of the multirotor are then well established [30] and can be expressed in the body frame as:

$$\boldsymbol{F}^b = m(\dot{\boldsymbol{V}}^b + \boldsymbol{\omega}^b \times \boldsymbol{V}^b), \tag{3.9}$$

where $\boldsymbol{V}^b = [u, v, w]$ represents the drone's translational velocity in the body frame, $\boldsymbol{\omega}^b = [p, q, r]$ is the drone's rotational velocity expressed in body frame, m is the mass of the multirotor and the vector F^b represents all applied external forces.

The external forces involved in the system are the forces of gravity, thrust and drag which can be expressed as:

$$\sum \boldsymbol{F} = \boldsymbol{F}_g + \boldsymbol{F}_t + \boldsymbol{F}_d \tag{3.10}$$

The force of gravity is simply $mg\overrightarrow{k}$ in the inertial NED frame, and can be expressed in the body frame using the rotation matrix $\boldsymbol{R}_i^b = (\boldsymbol{R}_b^i)^T = (\boldsymbol{R}_{z,\psi}\boldsymbol{R}_{y,\theta}\boldsymbol{R}_{x,\phi})^T$. The direction of the thrust force of the multirotor is entirely in the negative z-direction in the previously defined body frame. The drag force of a body moving at a relatively high speed relative to the air around it can be modeled as

$$\boldsymbol{F}_d = \frac{1}{2}\rho \boldsymbol{V}_a^2 C_d A, \tag{3.11}$$

where V_a is the airspeed of the drone, ρ is the air density, C_d is the drag coefficient and A is the surface area of the drone perpendicular to the airspeed direction.

Expressing the drag force in the body frame yields:

$$\sum \boldsymbol{F}^b = \boldsymbol{R}_i^b \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -\sum_{i=1}^{N} F_{t\,i}^b \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\rho V_{ax}^2 C_d A_x \\ \frac{1}{2}\rho V_{ay}^2 C_d A_y \\ \frac{1}{2}\rho V_{az}^2 C_d A_z \end{bmatrix} \tag{3.12}$$

Multiplying in $\boldsymbol{R}_i^b$ gives

$$\sum \boldsymbol{F}^b = \begin{bmatrix} -sin(\theta)mg \\ sin(\phi)cos(\theta)mg \\ cos(\theta)cos(\phi)mg \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -\sum_{i=1}^{N} F_{t\,i}^b \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\rho V_{ax}^2 C_d A_x \\ \frac{1}{2}\rho V_{ay}^2 C_d A_y \\ \frac{1}{2}\rho V_{az}^2 C_d A_z \end{bmatrix} \tag{3.13}$$

Note the negative sign for the thrust force as the z-axis is defined to be positive out of the belly/bottom of the drone. Plotting this into equation (3.9) yields:

$$m(\dot{\boldsymbol{V}}^b + \boldsymbol{\omega}^b \times \boldsymbol{V}^b) = \begin{bmatrix} -sin(\theta)mg \\ sin(\phi)cos(\theta)mg \\ cos(\theta)cos(\phi)mg \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -\sum_{i=1}^{N} F_{t\,i}^b \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\rho V_{ax}^2 C_d A_x \\ \frac{1}{2}\rho V_{ay}^2 C_d A_y \\ \frac{1}{2}\rho V_{az}^2 C_d A_z \end{bmatrix}, \tag{3.14}$$

which finally can be rewritten as:

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} -sin(\theta)g \\ sin(\phi)cos(\theta)g \\ cos(\theta)cos(\phi)g \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{m}\sum_{i=1}^{N} F_{t\,i}^b \end{bmatrix} + \begin{bmatrix} \frac{1}{2m}\rho V_{ax}^2 C_d A_x \\ \frac{1}{2m}\rho V_{ay}^2 C_d A_y \\ \frac{1}{2m}\rho V_{az}^2 C_d A_z \end{bmatrix} - \begin{bmatrix} qw - rv \\ ru - pw \\ pv - qu \end{bmatrix} \tag{3.15}$$

For the rotational equations of motion expressed in the body frame, the following equation for the sum of moments is well established: [30]

$$\boldsymbol{J}\dot{\boldsymbol{\omega}}^b + \boldsymbol{\omega}^b \times \boldsymbol{J}\boldsymbol{\omega}^b = \boldsymbol{M}^b, \tag{3.16}$$

where $\boldsymbol{J}$ is the moment of inertia tensor and $\boldsymbol{M}^b$ is the sum of external moments. Due to the drone's geometric symmetry in both the $i^b k^b$-plane and the $j^b k^b$-plane the inertia products $J_{xz} = J_{zx}$, $J_{xy} = J_{yx}$ and $J_{yz} = J_{zy}$ all equal zero. This gives the following expression for the sum of external moments:

$$\boldsymbol{M}^b = \begin{bmatrix} J_{xx}\dot{p} \\ J_{yy}\dot{q} \\ J_{zz}\dot{r} \end{bmatrix} + \begin{bmatrix} (J_{zz} - J_{yy})qr \\ (J_{xx} - J_{zz})pr \\ (J_{yy} - J_{xx})pq \end{bmatrix} \tag{3.17}$$

3.3.2 Guidance Systems

A guidance system repeatedly, or continuously, computes the desired position, velocity and attitude of a controllable vehicle, which are to be used in a given control system. These desired parameters will vary, depending on the guidance method in use. In the case of target tracking using velocity control, line-of-sight guidance, pure pursuit guidance and constant bearing guidance are commonly used methods.

Line-of-sight (LOS) guidance is a 3-point guidance scheme in which the interceptor, the controllable vehicle/object, must limit its motion along the reference-target line of sight vector. This guidance method is typically used in surface-to-air missiles.

The pure pursuit guidance method is similar to that of the LOS method, but is instead a 2-point guidance scheme, where no reference point is in use. With a pure pursuit approach the interceptor aligns its linear velocity with the interceptor-target line of sight vector. This is a common strategy in nature as well, where most predators chasing prey will adopt this method. In modern technology however, it is commonly employed in air-to-surface missiles. The desired velocity can be calculated as follows

$$\boldsymbol{v}_d^n = -k\frac{\tilde{\boldsymbol{p}}^n}{||\tilde{\boldsymbol{p}}^n||}, \tag{3.18}$$

where

$$\tilde{\boldsymbol{p}}^n := \boldsymbol{p}^n - \boldsymbol{p}_t^n \tag{3.19}$$

is the line-of-sight vector between the interceptor and the target and

$$k = U_{a,max}\frac{||\tilde{\boldsymbol{p}}^n||}{\sqrt{(\tilde{\boldsymbol{p}}^n)^T\tilde{\boldsymbol{p}}^n + \Delta_{\tilde{p}}^2}} \tag{3.20}$$

where $U_{a,max}$ defines the max approach speed toward the target and $\Delta_{\tilde{p}} > 0$ impacts the transient interceptor-target rendezvous behaviour [25]

Constant bearing guidance differs from the previously described methods as it is a predictive approach. Instead of following a target, this method predicts an intersection point at controls the interceptor towards this point. It is a 2-point guidance scheme and is often referred to as proportional navigation. It is considered ideal for scenarios that involve non-maneuvering targets. The desired velocity can be calculated as follows

$$\boldsymbol{v}_d^n = \boldsymbol{v}_t^n + \boldsymbol{v}_a^n \tag{3.21}$$

where $\boldsymbol{v}_a^n = -k\frac{\tilde{\boldsymbol{p}}^n}{||\tilde{\boldsymbol{p}}^n||}$ and $\boldsymbol{v}_t^n$ is the target velocity.

3.4 Kalman filter

Modern control systems are usually equipped with a state estimator used in the processing of sensor and navigation data. This raw data is typically sent to a signal processing unit for quality control and wild point removal before being transmitted to a control system. The state estimator is capable of noise-filtering, making state predictions and reconstructing unmeasured states. One of the more famous algorithms for this purpose is the Kalman filter, first introduced in the 1960's [25].

The Kalman filter is an efficient recursive algorith that uses a series of noisy measurements from a system's sensors in order to estimate the states of a dynamic system. The Kalman filter works for both linear and nonlinear systems. The noise-filtering capabilities of the Kalman filter allow it to remove both white noise and colored noise from the state estimates and even wild point removal can be implemented. If a temporary loss of measurements should occur, the filter equations behave as a predictor, ensuring the controlled vehicle does not immediately deviate far from its desired pose. At the moment when new measurements are available, the predictions are corrected and updated to give the minimum variance estimate.

A necessary assumption when designing a Kalman filter is that the model of the system is observable. This assumption must hold in order for the estimated states, $\tilde{\boldsymbol{x}}$ to converge to the actual states $\boldsymbol{x}$. Additionally, with an observable system, the state vector can be reconstructed recursively using the measurement vector $\boldsymbol{y}$ and the control input vector $\boldsymbol{u}$.

A discrete-time Kalman filter is often the state estimator applied in electromechanical systems and is defined in terms of the system model as follows [25]:

$$\boldsymbol{x}(k+1) = \boldsymbol{\Phi}\boldsymbol{x}(k) + \boldsymbol{\Delta}\boldsymbol{u}(k) + \boldsymbol{\Gamma}\boldsymbol{w}(k), \tag{3.22a}$$

$$\boldsymbol{y}(k) = \boldsymbol{H}\boldsymbol{x}(k) + \boldsymbol{v}(k) \tag{3.22b}$$

where

$$\boldsymbol{\Phi} = \exp(\boldsymbol{A}h) \approx \boldsymbol{I} + \boldsymbol{A}h + \frac{1}{2}(\boldsymbol{A}h)^2 + ... + \frac{1}{N!}(\boldsymbol{A}h)^N \tag{3.23}$$

$$\boldsymbol{\Delta} = \boldsymbol{A}^{-1}(\boldsymbol{\Phi} - \boldsymbol{I})\boldsymbol{B}, \tag{3.24}$$

$$\boldsymbol{\Gamma} = \boldsymbol{A}^{-1}(\boldsymbol{\Phi} - \boldsymbol{I})\boldsymbol{E} \tag{3.25}$$

and h is the sampling time.

The algorithm for the linear discrete-time Kalman filter is depicted in Table 3.1, from *Handbook of Marine Craft Hydrodynamics and Motion Control* [25].

Table 3.1: Discrete-time Kalman filter

Description	*Equation*
Design Matrices (usually constant)	$\boldsymbol{Q}(k) = \boldsymbol{Q}^T(k) > 0, \quad \boldsymbol{R}(k) = \boldsymbol{R}^T(k) > 0$
Initial Conditions	$\bar{\boldsymbol{x}}(0) = \boldsymbol{x}_0$ $\bar{\boldsymbol{P}}(0) = E[(\boldsymbol{x}(0) - \hat{\boldsymbol{x}}(0))(\boldsymbol{x}(0) - \hat{\boldsymbol{x}}(0))^T] = \boldsymbol{P}_0$
Kalman gain Matrix	$\boldsymbol{K}(k) = \bar{\boldsymbol{P}}(k)\boldsymbol{H}^T(k)[\boldsymbol{H}(k)\bar{\boldsymbol{P}}(k)\boldsymbol{H}^T(k) + \boldsymbol{R}(k)]^{-1}$
State estimate update	$\hat{\boldsymbol{x}}(k) = \bar{\boldsymbol{x}}(k) + \boldsymbol{K}(k)[\boldsymbol{y}(k) - \boldsymbol{H}(k)\bar{\boldsymbol{x}}(k)]$
Error covariance update	$\hat{\boldsymbol{P}}(k) = [\boldsymbol{I} - \boldsymbol{K}(k)\boldsymbol{H}(k)]\bar{\boldsymbol{P}}(k)[\boldsymbol{I} - \boldsymbol{K}(k)\boldsymbol{H}(k)]^T$ $+\boldsymbol{K}(k)\boldsymbol{R}(k)\boldsymbol{K}^T(k), \quad \hat{\boldsymbol{P}}(k) = \hat{\boldsymbol{P}}^T(k) > 0$
State estimate propagation	$\bar{\boldsymbol{x}}(k+1) = \boldsymbol{\Phi}(k)\hat{\boldsymbol{x}}(k) + \boldsymbol{\Delta}(k)\boldsymbol{u}(k)$
Error covariance propagation	$\bar{\boldsymbol{P}}(k+1) = \boldsymbol{\Phi}(k)\hat{\boldsymbol{P}}(k)\boldsymbol{\Phi}^T(k) + \boldsymbol{\Gamma}(k)\boldsymbol{Q}(k)\boldsymbol{\Gamma}^T(k)$

Chapter 4

System Overview

4.1 Software

4.1.1 DUNE - Unified Navigation Environment

DUNE is a runtime environment for unmanned systems' onboard software, developed by the Laboratório de Sistemas e Tecnologia Subaquática (LSTS) [22]. DUNE interacts with sensors, payloads and actuators, but also provides systems for communications, maneuvering, navigation, plan execution and vehicle supervision [6]. In essence, DUNE acts as a task manager and as a message bus manager. The runtime environment is written in `C++` and is both cpu architecture and operating system independent. DUNE has been designed for systems/vehicles with a wide range of performance capabilities, including very limited systems.

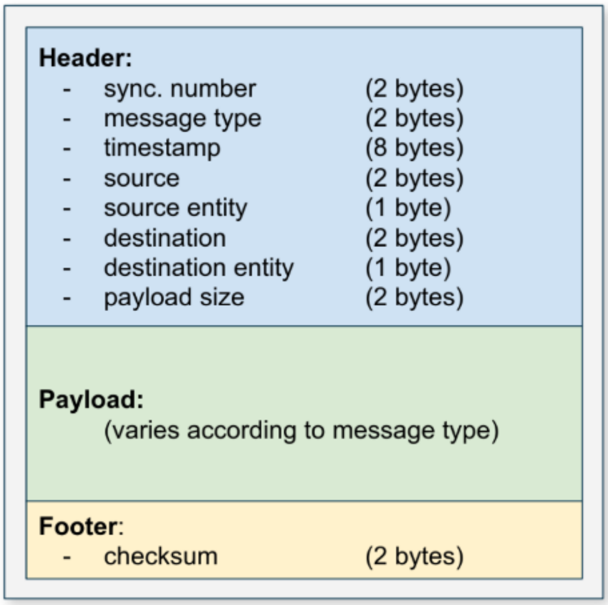
A task in DUNE is a `C++` class that has specific life time cycles and generally has responsibility of fulfilling a clear objective during runtime. This can for example be to read camera frames and collect information from the frame by using computer vision methods. The task manager coordinates the multiple tasks that are running. The scheduling and communication between tasks is then orchestrated by the task manager providing a programmatic consistency in the runtime of the tasks.

Another focus of DUNE is the message passing concept. The idea is that different tasks written for arbitrary parts of the system run on different threads, but can share data using a message bus. The communication, sharing of data, between the different tasks is governed by a publisher/subscriber design pattern.

4.1.2 IMC protocol

The communication protocol used in DUNE is the IMC protocol (InterModule Communication) [16], which is also created by LSTS. The protocol is used for sharing data between different tasks, but is also used for communication between different vehicles. The protocol is aimed to serve as a transport agnostic, delay and interrupt tolerant protocol to coordinate networked vehicles. A problem arises when communication is to be done between systems built upon different operating systems and with different hardware for communication. Thus the IMC protocol has been designed such that the data to be exchanged is self contained and can be recognized and interpreted by all participating modules regardless of platform dependencies. The IMC protocol can easily be expanded to include new message definitions in a local version of the protocol.

The whole protocol definition is contained in an XML file. Language bindings for different languages and documentation is automatically generated from this file. Thus DUNE can easily communicate through the IMC protocol with external system built on other languages than `C++`. A message following the IMC protocol has the following top-level structure:



Header:

- sync. number (2 bytes)
- message type (2 bytes)
- timestamp (8 bytes)
- source (2 bytes)
- source entity (1 byte)
- destination (2 bytes)
- destination entity (1 byte)
- payload size (2 bytes)

Payload:
(varies according to message type)

Footer:

- checksum (2 bytes)

Figure 4.1: IMC top level structure

The payload contains the message to be sent and contains several fields depending on the definition of the message in the XML file. An example of a message definition that will be used later is given below.

```xml
<message id="4610" name="Camera Tracking" abbrev="CameraTracking">
  <description>
  Message to track object
  </description>
  <field name="Target position x" abbrev="x" type="fp32_t" unit="m">
  </field>
  <field name="Target position y" abbrev="y" type="fp32_t" unit="m">
  </field>
  <field name="Target position z" abbrev="z" type="fp32_t" unit="m">
  </field>
</message>
```

4.1.3 MAVLink Protocol

Another protocol that is widely used for communication with drones is the MAVLink protocol [7]. MAVLink is a lightweight messaging protocol sharing similarities with the IMC protocol. Because of limited transmission rates that drones often experience, the protocol has been designed to have a small overhead. Although MAVLink is very efficient, it is limited to a max payload size of 256 bytes. IMC on the other hand has no boundaries for this and thus can nest many IMC messages in one message. The protocol is also commonly used for communication between onboard hardware modules as well. For example communication between DUNE and ArduPilot running on the flight management unit onboard a drone is done through MAVLink.

4.1.4 The DUNE Task

DUNE tasks follows the unix philosophy of doing one thing and doing it well. In general, DUNE tasks can be split into two categories: producer tasks and consumer tasks. The producer task is one that typically creates an IMC message variable, for example by reading a sensor value. This value can then be dispatched (published) to the message bus using the `dispatch(msg)` method, which sends the variable as an IMC message. The other tasks which then wants to be subscribed to a certain type of IMC message can do this by calling the `bind<IMC::[msg_type]>(this)` in the task constructor. The IMC message will now be received by the task whenever a IMC message of this type is dispatched from another source. Each tasks then has to have an implementation of what action do perform when receiving a message of the type they are subscribed to. The specific function handle for this event is `consume(const IMC::[msg_type] *msg){})`. The function defines the action of the task on said event, and can for example be used to update the internal reference of a position of an object of interest, which is gathered by another task. There is currently no method for subscribing to messages from specific publishers sources other than filtering the messages by source post reception.

Some of the tasks implemented in DUNE only need to react to new message events, but other ones needs to execute at a certain frequency. DUNE has options for defining a task as periodic which augments the task with a main loop that is executed at a configurable frequency. This can for example be used to read from a sensor at 10Hz. The task manager in DUNE handles the scheduling of the periodic tasks.

Configuration

There are many different tasks in the source code for DUNE for distinct purposes and many of these these will not be relevant for the vehicle using DUNE. Therefor configuration files are used to tell DUNE which tasks to include in the runtime.

The selection of tasks that are to be run during a session of DUNE is given as a command line argument with the name of the configuration file to be used. These files are characterized by their `.ini` extensions. As well as telling DUNE which tasks to include, configuration files also contain parameters that are passed on to the tasks. Configuration files can include other configurations, and thus enables building upon existing configurations.

For the OASYS project NTNU has a configuration file developed for the base features of the UAV that will be used, specifically the `ntnu-hexa-003.ini` configuration file. This configuration handles basic functionality such as tasks that needs information about the mathematical model of the drone and handles the communication systems available.

The configuration file created and used in this thesis for field testing is `3DR.ini` To run DUNE with this configuration the following command is used: `./dune -c 3DR`

4.1.5 OpenCV

OpenCV is one of the worlds most popular open source library for computer vision algorithms [10]. The library provides highly optimized implementations of more than 2500 algorithms and is used by large companies such as Google and Microsoft, as well as researchers. The library supports many languages such as `python` and `Matlab`, but most importantly `C++`. This `C++` support means that OpenCV can easily be included in DUNE implementations. OpenCV has implementations of many of the computer vision concepts discussed in section 3.2.

4.1.6 ArduPilot

ArduPilot is an open-source autopilot suite aimed at controlling UAVs [5]. The software suite supports many different types of crafts, but mainly fixed wing and multi-rotor crafts. With a collection of low level control systems and high level algorithms for navigation, the control of the craft becomes considerable an easier task to execute. ArduPilot is capable for supporting fully autonomous systems as well as aiding a remote operator in controlling a vehicle. ArduPilot offers helpful abstractions, such as setting desired linear velocity without having to manually calculate the rotor angles and velocities which achieves this velocity.

ArduPilot uses the MAVLink protocol for communications with external systems. DUNE supports communication with ArduPilot by abstraction through the ArduPilot Task.

ArduPilot also includes MAVProxy [8] which is a GCS terminal that lets the user interface the running ArduPilot software by translating the commands to the appropriate MAVLink messages. MAVProxy can be used to initiate basic commands such as takeoff, set position and landing. A command reference for basic command messages are supplied here [4].

ArduPilot is often run on flight management units onboard UAVs. A common setup is to run DUNE on a microprocessor unit onboard the UAV while running ArduPilot on the flight management unit. Alternatively, external computing can be done by running DUNE on a GCS which communicates with the UAV running ArduPilot onboard.

4.1.7 FlightGear

FlightGear is an open source software providing flight-simulation. ArduPilot and FlightGear can be synced, and by extension also DUNE. The flight simulator provides an virtual 3D environment for visualizing the flight. The simulation view can easily be redirected as input to a computer vision algorithms and will be a useful tool for providing testing navigation methods based on computer vision.

The camera view is shown in the FlightGear program, but the preview is also available locally over port 8080 by default over http. The default path for the preview is `http://localhost:8080/screenshot`. This address can easily be provided as video input source to capturing methods provided by OpenCV. A notable caveat with this method is that the preview address is not a dynamically providing the updated view, unless the request is refreshed. To achieve a live feed of the FlightGear view, the capturing device provided by OpenCV must be reinitialized before each frame grab attempt to get the current view.

The ArduPilot install location contains a script for launching FlightGear where synchronization with ArduPilot is automatically established. This script is located under `Tools/autotest/fg_quad_view.sh` in the ArduPilot install folder. The script takes care of all the command line arguments that are needed for the ArduPilot to work with FlightGear and requires no editing.

4.2 Hardware

4.2.1 DJI S1000 multirotor

The S1000 octo-rotor UAV is a sturdy and powerful drone produced by DJI [11]. With a takeoff weight of 9.5kg the drone has a hovering time of 15min. The drone can handle extra hardware mounted onboard and still remain capable of carrying the MUG payload during flight. The drone is customized with a microprocessor unit, flight management unit, camera and electropermanent magnet for locking on to the MUG.

4.2.2 3DR solo

An alternative to the DJI s1000 drone is the 3DR Solo quad-copter [19]. The creators of the Solo quad-copter are the same people that are behind the autopilot platform Pixhawk 4.2.7. This drone is smaller than the S1000 weighing about 1.5kg, and has a mount for a GoPro camera. The Pixhawk onboard the 3DR solo runs Open Solo 4 which is developed by the Open Solo team [17] independently from 3DR.
In contrast to the S1000 drone, the 3DR solo does not have the capability of running DUNE onboard on a microprocessor unit.
This drone is not as powerful as the S1000, and is not able to carry out the full mission described by the thesis. On the other hand, the quadcopter is sufficient to complete a fully guided approach of the marker aided by computer vision, but not capable of carrying the added weight of the MUG.

4.2.3 Ryze Tello

Another drone powered by DJI is the miniature quadcopter drone, Ryze Tello [13]. This drone has been used for loose testing of different aspects that is to be implemented in the DUNE. Communication and control is done over UDP which differs from the hardware communication of DJI S1000. Modules such as the camera vision detection of ArUco Markers and control reference values can be tested on the Tello drone without much loss of generality. The Tello drone offers a poor quality stream of data, although at a low latency, and a small API of available commands [12]. The miniature drone in this thesis serves as quick iterative testing platform where detection methods and simple guidance methods which use velocity references can be tested.

4.2.4 ArUco Markers

When the MUG appears at the ocean surface it has to be detected by the UAV using computer vision. The ocean environment makes it difficult to distinctively separate the MUG from its surroundings. To aid the detection of the MUG, an fiducial marker of type ArUco is attached to the top of the wing of the MUG. ArUco (Augmented Reality University of Cordoba) is a fast, lightweight and open source library for camera pose estimation using squared markers [26]. An ArUco marker consists of an external black border and an inner region that encodes a binary pattern.

The ArUco code is designed in black and white for detection algorithms to easily identify them, but this choice of colour coding provides a lower visibility in ocean environments. Instead, the used marker follows the colour scheme of international aerospace color which is designed to easily differentiate the object from its surroundings. The specific marker used has dimensions 200x100mm.

To convert back to the original colour convention such that standard ArUco detection methods can be applied, a simple threshold algorithm is used in post processing of the raw image data. Instead of using advanced object detection methods for tracking the MUG, established fast and accurate detection methods provided by the ArUco library can be used

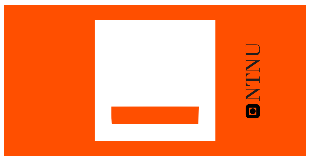


Figure 4.2: ArUco Marker Design

on the highly visible marker. This will provide reliable tracking while descending on top of the MUG for pickup. The pictured ArUco marker in figure 4.2 used in this project belongs to the original ArUco dictionary and has an id of 1.

The ArUco detection algorithm needs a good view of the marker to correctly identify the ArUco code, and estimate pose. This means that the mentioned tracking method will only work at lower altitudes, but with high accuracy.

4.2.5 MUG

The proposed design for the MUG is given in the figure below:

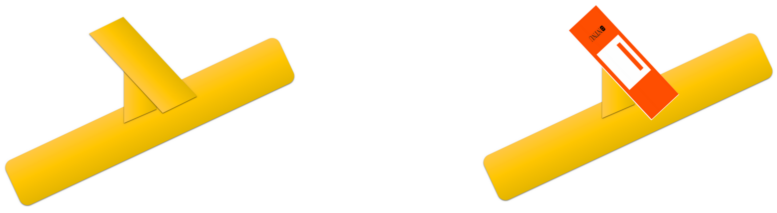


Figure 4.3: MUG model

The top part of the MUG is a wing which is flat and magnetic. This wing will be almost exactly at the water surface when the MUG is to be picked up. The wing will be where the ArUco marker is placed as well, and center of gravity can be expected to be directly below it.

4.2.6 BeagleBone Black

BeagleBone black is a high powered microprocessors capable of running a small linux distribution while being a minimal system [1]. The BeagleBone black is used as an onboard microprocessor for running DUNE on the S1000.

4.2.7 Pixhawk 4

The S1000 utilizes different sensors and actuators with different interfaces. To ease the process of interfacing with all the connected units, Pixhawk hardware allows full control and a single tied point where all data is interfaced [18]. Pixhawk also supports autopilot softwares with their flight management unit (FMU), such as the ArduPilot software suite.

4.2.8 oCam-1CGN-U-T

The camera model used with the S1000 drone is the oCam-1CGN-U-T developed by Withrobot in 2017 [3] The main importance of the camera used by the drone for computer vision is high fidelity for detail and fast framerate for decreasing latency. It is a 1 mega pixels colour shutter camera that supports external triggering. The colour output is Bayer RGB. The oCam producers also supplies source files for interfacing the camera in `C++`.

4.2.9 GoPro Hero4

The camera used by the 3DR solo drone is GoPro Hero4 [15] and is mounted on a gimbal underneath the drone. GoPro Hero4 allows for live streaming over WiFi at a high framerate and resolution. With the 3DR solo, all processing will be done externally, which means that the GoPro footage stream will be interfaced over a network stream, instead of a hardware interface.

4.2.10 Electropermanent magnet

To secure the MUG to the UAV during pickup, an electropermanent magnet, OpenGrab EPM v3 [2], is used to lock onto the top wing of the MUG. The magnet has a typical max lifting force of 300N, more than enough for lifting the MUG.

In previous attempts a string has been used to hook the MUG, but were too imprecise. Using a magnet results in a more accurate locking method at the cost of needing to be closer to the target. This will affect the descent phase when the MUG is too close to be identified by the camera. A benefit with the electropermanent magnet is that it only consumes power during switching states, and thus being locked on to the target doesn't yield a higher power consumption for the drone. That is of course other than that of the increased total weight caused by the payload. The magnet can cause magnetic interference, especially when left in the on state with no payload attached. Depending on the mounting, the interference can be ignored as it is heavily reduced beyond 10cm.

4.3 System State Machine

A system including a state machine was implemented in DUNE to handle the approach phases described in the introduction 1.1. The following states are used in the implementation:

- Initialization
- Manual
- High Altitude Approach
- Low Altitude Approach
- Landing

A supervisor task is added to control the state machine of the UAV and the following state diagram describes the top down behavior:

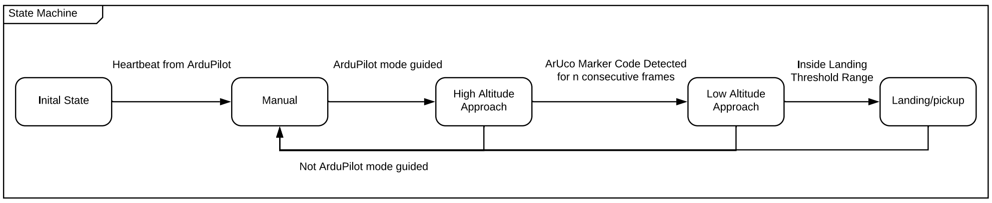


Figure 4.4: State Diagram

4.3.1 Initialization

The DUNE program relies on communication with ArduPilot which sends out heartbeat messages following the MAVLink protocol. These heartbeat messages contains the mode the ArduPilot software currently is in. Upon receiving a heartbeat the state transitions to the manual state.

4.3.2 Manual

In the manual state, the state machine runs idle. The main purpose of the manual mode is to offload the control to a operator. A operator can e.g. be used for takeoff and landing while in this state. The state machine will also always listen for heartbeat messages indicating the operator has taken control of the vehicle and and return to this state.

4.3.3 High Altitude Approach

In this state the MUG position is located by pixel position using detection method based on shape and color of the visible part of the MUG. ArUco marker detection is always performed as different factors such as fog and image noise will affect which height the ArUco marker will be visible from.

4.3.4 Low Latitude Approach

At lower altitudes the marker is visible for the drone and guidance based on the fiducial references in the marker can be used.

4.3.5 Landing

If within the landing threshold, the drone sets a predetermined landing velocity to assert a landing. In this phase the drone only moves in z direction as the visual of the marker is lost when closing in.

Chapter 5

Implementation

To cover the necessary implementations required for executing the desired functionality, implementation is divided into the following:

1. Mutual Modules
2. Miniature Drone - DJI Ryze Tello
3. DUNE - Simulation and 3DR

The DUNE implementation will be the same for simulation and the 3DR solo drone, except for specific task parameters. The mutual section will cover the modules which are implemented as the same for both the miniature drone and DUNE.

5.1 Mutual Modules

5.1.1 Filtering and Segmentation

A key concept in computer vision methods that was brought up in 3.2.5 is filtering out the unwanted information in the image. The ArUco marker detailed in 4.2.4 is the object of interest and filtering will be employed to make the information encoded in this marker as visible and distinct as possible for the detection algorithms. The filtering process consists of sequential methods given in sequence below

1. Convert from BGR to HSV
2. Gaussian Blur Filter
3. Range Thresholding

Color Space Conversion

In OpenCV the default image captured is in BGR color space. The first step in the filtering process is to convert the input image from BGR to HSV. BGR values are correlated with the amount of light hitting the object, making the color channels correlated to each other. On the other hand, HSV has properties hue, saturation and value, which are better properties for discriminating different objects from another based on color. With this is mind, the marker color scheme is specifically chosen because of it's uniqueness in HSV color values compared to natural environments.

The method for converting the color space of an image to another is provided by OpenCV:

```
cv::Mat image_in_BGR, image_out_HSV;
cv::cvtColor(image_in_BGR, image_out_HSV, cv::COLOR_BGR2HSV
   );
```

Noise Removal

Noise filtering is then necessary to remove unwanted noise in the image which can affect the detection quality. The Gaussian blur filter works by smoothing the pixels inside a moving window, and function as a low pass filter. This reduces the noise at the cost of an added blur, which reduces the quality by a factor. The method is implemented by the following function in OpenCV

```
cv::Mat image_in, image_out;
cv::Size window_size(3, 3);
int standard_deviation = 0;
cv::GaussianBlur(image_in, image_out, window_size,
   standard_deviation);
```

A zero value for the standard deviation argument means that function calculates the used value based on the window size.

Binarization

The next filtering method is the binarization of the image. As the HSV color scheme of the marker is known, the marker information can be extracted by binarization performed with threshold range values. The following method performs the binarization in OpenCV:

```
cv::Mat image_in_HSV, image_binarized_out;
cv::Scalar low_HSV(0, 0, 0), high_HSV(180, 255, 255);
cv::inRange(image_in_HSV, low_HSV, high_HSV,
   image_binarized_out);
```

Different software uses different value ranges for HSV values. In OpenCV the HSV value ranges has a lower bound of 0 and 180, 255 and 255 respectively for the upper bound. All the pixels in the input image that lies within the range specified will outputted with a value of 255, and values outside the range will be outputted as 0. The HSV values of the marker is affected by the light conditions at the scene, and good threshold ranges for extracting the ArUco marker from its surroundings have to be used.

5.1.2 ArUco Detection

The creators of the ArUco markers [26] have created a library [14] that provides the general ArUco detection functionality. This library uses OpenCV for image processing. OpenCV also provides a ArUco library for implementing ArUco detection, but this library is not as complete and up to date as the library provided by [26]. All classes and functions from the library uses the `aruco` namespace.

There are 3 pivotal classes in the ArUco library that need some configuration to be correctly used:

- `aruco::Marker`
- `aruco::CameraParameters`
- `aruco::MarkerDetector`

They are declared and initialized in the following way.

```
cv::Mat intrinsic, distortion;
aruco::CameraParameters cameraParameters;
aruco::MarkerDetector markerDetector;
cameraParameters.CameraMatrix = intrinsic;
cameraParameters.distortion = distortion;
markerDetector.setDictionary(aruco::Dictionary::ARUCO, 0.2)
    ;
```

Firstly, the intrinsic camera parameter matrices has to be created as `cv::Mat` objects. An object of class `CameraParameters` is then created with these values. This camera parameters object is used by the marker detector later on. The next step is to configure the marker detector with the desired ArUco dictionary and the error correction rate.

The process of identifying markers in a frame provided by the camera is then done by:

```
cv::Mat frame;
std::vector<aruco::Marker> detectedMarkers;
float markerSize = 0.1; // meters
detectedMarkers = MDetector.detect(frame, cameraParameters,
     markerSize);
```

This provides a vector of identified ArUco markers belonging to the dictionary defined earlier. The vector of ArUco markers can then be iterated through and information about id and pose can be gathered in the following way:

```
int id;
cv::Vec3d position;
cv::Mat orientation;
for (auto &marker : detectedMarkers) {
  id = marker.id;
  position = marker.Tvec;
  orientation = marker.Rvec;
  aruco::CvDrawingUtils::draw3dAxis(frame, marker,
     cameraParameters);
}
```

The ArUco library also includes utilities for drawing marker objects in an image providing a visualization of the detected pose of the marker.

5.1.3 Rectangle Marker Detection

At higher altitudes the marker will not be detailed enough to decode the marker information with ArUco detection methods. Thus a simpler detection method must be used that relies on the shape and color of the marker, which is much easier to identify with lower resolution.
The algorithm is based off the initial filtered image. Firstly all the contours are found in the image.

```
cv::Mat image_filtered;
std::vector<cv::Mat> contours;
cv::findContours(image_filtered, contours, cv::
   RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
```

The `cv::RETR_EXTERNAL` implies that only the outer contours should found, ignore nested contours. Additionally, `cv::CHAIN_APPROX_SIMPLE` means that redundant information of the contour points will be removed, e.g. a line would only need two points to be represented.

Then with a list of all the contours found in the filtered image, each of the contour is seen as a candidate. For each contour candidate, the rectangle with the smallest possible pixel area that envelops all the contour points of a candidate is generated. Then the rectangle fit is calculated as the area of the contour divided by the area of the generated enveloping rectangle. Finally, the rectangle fit is weighted with the area of the contour candidate. The latter part is included to compensate for pixel noise and artifacts as they are likely to score high on rectangle fit. The best candidate is then determined as likely being the ArUco marker. Thresholds are also included as a minimum standard that a candidate has to meet.

```
std::vector<cv::Mat> contours;
float min_size = 100 // min pixel area
float min_shape_fit = 0.2; // => min 20% fit
float best_fit = -1;
cv::RotatedRect best_candidate;
for (auto &contour : contours) {
  float contour_area = cv::contourArea(contour);
  if (contour_area < min_size) {
    continue;
  }
  cv::RotatedRect min_area_rect = cv::minAreaRect(contour);
  float rect_area = cv::contourArea(min_area_rect.points);
  float shape_fit = contour_area/rect_area;
  if (shape_fit < min_shape_fit) {
    continue;
  }
  float weighted_fit = shape_fit * contour_area;
  if (weighted_fit > best_fit) {
    best_fit = weighted_fit;
    best_candidate = min_area_rect;
  }
}
```

A negative `best_fit` value after the procedure implies that no candidates were found that met the criteria.

The performance of the filtering/segmentation greatly increases the detection rate as this method is more prone to false positives than the ArUco marker. Good filtering also decreases the amount information the `cv::findContours` has to search through and the amount of contour candidates, improving processing speed.

5.1.4 Transformations

The computer vision detection locations by the miniature drone, simulation and the S1000 are all expressed in the same frame. This frame is centered in the center of the camera view where a detected ArUco marker is given x,y position according to positive x horizontally to the right and positive y vertical downwards. This means the z position is given with positive direction coinciding with the direction the camera is facing. The camera coordinate frame is illustrated in figure 5.1.

The position estimates the navigation method uses is gathered from the camera which is mounted on the body of the vehicle. All navigation will be done with reference to body frame, and the pose estimates from the camera must be transformed from camera to body frame. The camera on Tello, FlightGear, 3DR and S1000 has a fixed mounting where suitable static transformation matrices from camera to body must be found. In the case

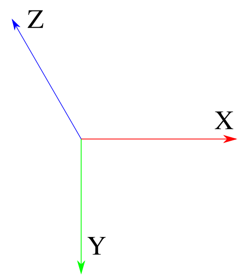


Figure 5.1: Camera Reference Frame

where a variable/gimbal mounting is used for the camera, a composite time-varying roll-pitch-yaw rotation must be used instead to express position in body frame as discussed in 3.1.2.

5.1.5 Camera Calibration

The quality of the camera calibration is necessary to correctly perform pose estimation as this directly affects the accuracy of the detection methods.
The chosen method for calibration was a standard checkerboard calibration performed with OpenCV. As the size and number of squares are known in the printed checkerboard, the information about the camera parameters and distortion coefficients can be retrieved by analyzing the image projections. This is done by analyzing multiple images of the chessboard at different angles and comparing the identified chessboard grid to known values, such as the number of corners in rows and columns of the board.

Algorithm

Given a chessboard, for example printed on a piece of A4 paper, the key points will be the corners of each square that is not on the edge of the chessboard. The size of the printed squares in meters have to be known before running the calibration. To create the vector containing the object points for each of the corners of the checkerboard, the following procedure is used.

```
cv::Size boardSize(7, 9);
float squareSize = 0.02; // 2cm
std::vector<cv::Point3f> corners;
for( int i = 0; i < boardSize.height; i++ ) {
    for( int j = 0; j < boardSize.width; j++ )
        corners.push_back(Point3f(float(j*squareSize),
                                  float(i*squareSize), 0));
    }
}
```

The resulting vector of object points contains the 3D keypoints with origin in the upper left square. This list of object points is then used as basis for what the corners in the 2D

image should map to.
The pixel position of the chessboard corners in a calibration image can be found by in the following way:

```
cv::Size boardSize(7, 9); //example
std::vector<std::vector<<cv::Point2f>> totalImagePoints
std::vector<cv::Mat> calibrationImages;
for (auto &calibrationImage : calibrationImages) {
  std::vector<cv::Point2f> imagePoints;
  cv::findChessboardCorners(calibrationImage, boardSize,
     imagePoints);
  totalImagePoints.push_back(imagePoints)
}
```

Where `imagePoints` is an output argument that contains the pixel points of each corner of the checkerboard found in the calibration image. This process is done for each calibration image and the corner positions are stored for each image in `totalImagePoints`.
With a list for the 3D object points and the 2D image pixel points of the corners, the camera matrix and distortion coefficients can be found by the OpenCV function `cv::calibrateCamera`.

```
cv::Size calibrationImageSize;
cv::Mat cameraMatrix, distCoeffs;
std::vector<cv::Mat> rvecs, tvecs;
std::vector<std::vector<cv::Point3f>> totalObjectPoints;
std::vector<std::vector<cv::Point2f>> totalImagePoints;
double rms;
rms = cv::calibrateCamera(objectPoints, imagePoints,
   calibrationImagesSize, cameraMatrix, distCoeffs, rvecs,
   tvecs, 0);
```

The `totalObjectPoints` is a vector that includes the previous list `objectPoints` replicated to the amount of calibration images. The rms value of the reprojection error are returned from the function indicating the fit of the camera matrix based on all the calibration images. The function start off by estimating the initial camera pose as if the intrinsic parameters was already known using a SolvePnP function. Then the function minimizes the reprojection error, the sum of squared distances between the observed image points and the projected object points using the current intrinsic estimates and poses.

5.1.6 ArUco Detection Tuner

To further improve the filtering methods for detecting the ArUco markers in the video stream, a program was made to analyze the percentage of detected ArUco markers in all frames contained in a directory or from a video file. This module uses the captured video data from flights where the filter parameters easily can be adjusted and the result analyzed for increased effectiveness with the same input. This is used to optimize the parameters based on the current flight environment and detect the highest as possible number of markers.

5.2 Tello Modules

All the modules created for the Tello drone has been implemented in `C++` independently of DUNE. The Tello drone does not have the capability of running custom software onboard, but has an API for interfacing the onboard firmware. This API is accessed by sending messages over UDP and the list of API calls available is listed here [12].

The GCS software created for the Tello was created with aim to build a modular setup for controlling and interfacing the drone. In difference to DUNE, all layers had to be developed from scratch. Multiple modules were created to abstract the lower level interactions built upon the UDP communication such that the control algorithms and data manipulation could be implement at a higher level without knowledge of underlying interfaces. DUNE already offer this level of modularity by design, but translating the primitive API that the Tello offers to a DUNE implementation would remove the benefit of using DUNE at all. The main modules of the Tello implementation is

- State Machine
- Control
- Video Decoding

5.2.1 Control

In the API list for the Tello drone, there is a command for setting the desired velocity with respect to body frame of the vehicle by sending a command of the following form: `rc ud lr bf yaw` Where `rc` is the command identifier and the proceeding arguments are replaced with a value of -100 100 respectively. The arguments signifies up-down, left-right, back-forward, yaw rate. The constant bearing guidance method described in 3.3.2 is implemented with reference to body frame navigation and the generate velocities are used as arguments for this command.

5.2.2 Video Decoding - h.264 decoder

The camera input from the Tello drone is sent to port 11111 to the IP of the machine that connected to the drone. The video is sent over UDP with H.264 encoding. OpenCV normally has options for capturing input video streams of any type, but for H.264 encoded streams sent over UDP, it causes a high delay of about 5 seconds. OpenCV uses `ffmpeg` to decode the stream internally, but no workaround for the internal buffering was found. Thus an attempt at creating an custom H.264 byte stream decoder was made to amend the latency issue.

When installing `ffmpeg` the additional source code library, `avlib`, is also installed which can be used to create specialized video processing tools in `C++`. Using the source code a module for decoding a H.264 stream was created. The module stitches the fragmented H.264 encoded UDP stream into a sequence of `AVframe` type frames. Finally the module converts the frames into `cv::Mat` objects such that the functionality provided by OpenCV can be used. Using this H.264 decoder implementation, internal buffering is avoided as the processing method is fully customized and thus latency is reduced to a minimum.

5.2.3 State Machine

The main module of the Tello drone is the state machine. Three states were defined to handle the flight of the drone.

- Takeoff
- Approach
- Landing

Takeoff

When the drone is manually turned on, it creates a network which users can connect to. It establishes a session with an external machine when it receives an message over UDP containing the the string `ok`. To enable video streaming, a API call on the form `streamon` is sent. The drone responds with an `ok` string on the same port if the command was received and successful. After this the takeoff command is sent and when `ok` is received back the state machine transitions to approach.

Approach

After takeoff the image stream data is analyzed with the ArUco detection method described in 5.1.2. While continuously processing the image dat and detecting markers, a timer for dead reckoning is set. The timer is reset whenever a marker is detected, but in the event the timer elapses a control signal for setting the velocity reference to zero is sent and the drone hovers in place. At the event of an object detection, the located object position is transformed from camera frame to body frame and a feed forward loop is executed, instructing the Tello to approach the marker following constant bearing guidance as mentioned in 3.3.2. At the event of a detection loss, the dead reckoning timer will elapse based on a set time, causing the drone to enter the dead reckoning state again. A predefined altitude and horizontal radius is used to determine when the drone should transition to the landing state.

Landing

Given a certain amount of consecutive frames inside these thresholds, the drone will given signal to perform a landing maneuver, which terminates the flight.

5.2.4 Transformations

The Tello drone has a built in camera which its mounted at the end of the body. To capture what is underneath the drone, a mirror is mounted to angle the view. This mirror acts a simple rotation of -90° degrees about the camera x axis and in addition causes the image to become flipped about the x axis. The flipped image is corrected in OpenCV by using the function `cv::flip(image, image_out, 0` where the zero argument is the flip code which signifies a horizontal flip. An illustration of the body and camera frame is given below:

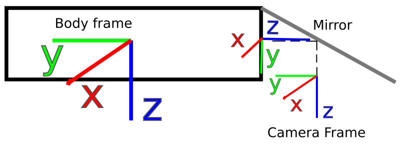


Figure 5.2: Tello Side view with Reference Frames

This causes the body frame to be translated with respect to the camera frame, but with no orientation change. A suitable transformation matrix from camera to body frame that compensates for the translation distance is:

$$\mathbf{T}_c^b = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -0.035 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

5.2.5 Video Logs

Flight data logging is important for analysis in post processing, and especially the video stream data recorded from the Tello drone. The video stream is recorded as a sequence of images and `ffplay frame%1d.png` is used to playback the images as video. `ffplay` is provided by the developers of ffmpeg [20].

5.3 DUNE Modules

5.3.1 IMC messages

Reusing IMC messages should be preferred, but for some cases new IMC messages was necessary for creating a explicit information flow between tasks and to handle data logging.

Reused Messages

- IMC::DesiredControl - Contains float values for desired control loop velocity references.
- IMC::Autonomy - Contains information about ArduPilot state, e.g. autonomous mode or manual.
- IMC::EstimatedState - Contains values for NED position, velocity and orientation using the takeoff position as origin.

Newly Defined IMC messages

The following messages were created to complete the information flow in the implementation.

- IMC::ArUcoTracking - Contains fields for position and orientation and the ID of the ArUco marker.
- IMC::CameraTracking - Contains fields for position of general objects.
- IMC::ConstantBearingTarget - Contains target positions and velocities used to update constant bearing velocity.
- IMC::SeaSurfacePickupState - Contains enumerated fields for sharing the current state of the system with other tasks.

IMC::CameraTracking

This messages is used to dispatch location of objects tracked within the camera view. This message is used for sharing general position of a tracked object where the method for finding the object is not of importance. This message is used by the task that captures the video stream during flight to update the tracked position of the desired object.

IMC::ArUcoTracking

This is the specific message used by the camera task to dispatch information about detected ArUco markers in the camera view. In addition to the position fields in the generic `IMC::CameraTracking` messages, this message also includes ArUco specific information such as the ArUco marker ID and orientation of object. This means that the subscriber, can filter the detected markers by ID and apply custom behavior based on ID.

IMC::ConstantBearingTarget

This message was created to update information about the target position and velocity needed by the constant bearing guidance task. A custom message also provides a method for doing transformation and processing of the tracked camera object before dispatching a target reference. In this implementation, it is necessary to transform the detected position of the marker to body frame before dispatching a message of this type.

IMC::SeaSurfacePickupState

This message includes the state of the supervisor state machine. It is used to share the current system state to other tasks that execute differently depending on the state of the system.

5.3.2 Tasks

As mentioned earlier in section 4.1.1, DUNE is built upon tasks that can subscribe to and publish messages following IMC protocol. The different tasks necessary for building up a functional state machine that covers the different problems discussed in this thesis are:

- SeaSurfacePickup Supervisor Task
- Camera Task
- ConstantBearing Task
- Logger Task
- ArduPilot Task

Altered Tasks

ArduPilot Task

The ArduPilot task is responsible for the communication between DUNE and the external ArduPilot program. The task therefor includes a lot of the functionality that ArduPilot supports, providing the interface to DUNE. Missing from the original task was sending velocity references to the ArduPilot software by the MAVLink protocol. As the constant bearing guidance method outputs a reference velocity, the task had to be altered to support this for the guidance method to work. This was solved by making the ArduPilot task subscribe to IMC::DesiredControl messages and adding an action to send the reference to ArduPilot with the apropriate MAVLink message that supported this. In the same MAVLink message, a flag can be used to indicate which frame the velocity reference is expressed in. The appropriate flag for body fixed navigation was used in this case instead of the default NED flag.

Newly Added Tasks

Camera Task

The camera task interfaces the video stream coming from the mounted vehicle camera, filtering the frame, performing detection methods and displaying the video stream with annotated data displayed on top of the received frame. For post processing purposes, the video stream is also saved to file.

The task is defined as a periodic task executing at the defined frequency specified in the configuration file. The flow of the periodic loop of the camera task is given by figure 1 in appendix B. At each execution cycle, the task tries to grab a frame of the incoming video stream. If no such frame is available at the time, the current cycle is ended. If a frame is available, the next action is to filter the input by the same method as discussed in 5.1.1. First the task tries to identify an ArUco marker in the filtered image by using the ArUco detection algorithm from 5.1.2. If an ArUco marker is detected, the task dispatches the pose data, else the task tries to identify the pixel location of the rectangle shaped marker as described in 5.1.3. Both the ArUco pose and the pixel location is expressed in the same frame. The pixel position is then normalized based on the frame size and dispatched. The SeasSurfacePickup supervisor task later handles the translation from pixel location to meter units.

Lastly, the original frame with annotated information is displayed to the operator alongside the filtered frame. In the case where the marker is visible in the original frame, but not the filtered frame, improved methods for filtering can be developed in post analysis of the saved video data. The annotated data shown on top of the frame received is marker position in either pixel or centimeters depending on the successful detection method, the current state of the supervisor task, and the fps.

The position data of the ArUco markers is not altered in the camera task, all transformation from camera frame to body frame is done by the supervisor task.

A configuration file example for this task can be found in appendix A under Sensors.Camera. Parameters such as ArUco marker square size, camera matrix, distortion coefficients, and filter values can be set in the configuration file.

Constant Bearing Task

The constant bearing task calculates the desired velocity of the vehicle for performing constant bearing guidance. The algorithm is implemented as described in section 3.3.2. The task receives the vehicle position from IMC::EstimatedState messages, and the target position and velocity from IMC::ConstantBearingTarget. One caveat with this implementation is that the theory for constant bearing guidance relies on using a common inertial reference frame for both the interceptor and the target. The pose estimates of the target given by the other task are expressed in vehicle fixed frames. The constant bearing algorithm then has to be changed to handle body frame navigation. To achieve this, the NED vehicle position received from IMC::EstimatedState is ignored and set to zero for all axes. The target position is then seen as the direct error in position. The IMC::EstimatedState functionality is included for potential future navigation methods based on NED reference frames. The constant bearing task flow is shown in figure 2 in appendix B

The task has parameters for setting max velocity of the vehicle, and the rendezvous factor. Setting the max velocity prevents the vehicle from performing sudden maneuvers while doing testing as the operator can easily detect and react to unwanted behavior before harmful damage is done.

SeaSurfacePickup Supervisor Task

The SeaSurfacePickup supervisor task is the main system task that governs the state machine detailed in 4.3. The supervisor subscribes to the following messages

- IMC::CameraTracking
- IMC::ArUcoTracking

- IMC::AutopilotMode

which it uses for basis for state transitions and actions. An in detail flow diagram of the task is given in appendix B by figure 3.

When maneuvering based on IMC::CameraTracking data, the supervisor has two criteria that it uses to decide when to descend. These criteria are given as percentage radii centered at the origin of the camera frame. The first criteria is an initial radius, e.g. 5%, that the pixel location must be within before descending is added to the constant bearing target message. After meeting this criteria, a looser criteria, e.g. 10%, is used for allowing some displacement when descending. If the looser criteria is breached, the pixel location of the marker must be centered within the initial radius again before descending.

The supervisor task can also be parametrized with a hovering altitude as shown in appendix A under Supervisors.ssp. When maneuvering based on IMC::ArUcoTracking data in Low Altitude Approach, a hovering altitude of 0 implies that the drone should attempt to land on the target, and a value above 0 will cause the state machine not to transition to the landing state and instead hover at this altitude while keeping the horizontal position error at 0. As the marker is eventually lost within the camera view when approaching closely, a parameter, `Landing Mode Altitude`, is used for determining which altitude above the marker is sufficient before transitioning to the landing state(if landing is intended). When attempting to land, the dispatched constant bearing target z value is chosen as a constant value to enforce downwards movement without ArUco pose estimate updates.

Logger Task

DUNE has built in logging of IMC messages dispatched during a session, but a custom logger task was created to log data to a simpler and human readable format. The logger task subscribes to all the IMC messages detailed in 5.3.1 and does not dispatch anything. Instead the task reads all the relevant imc data dispatched and logs a copy of the data to a file corresponding to the IMC message. All data entries are appended with the time elapsed since the program execution began. Each file name is also appended the start time of the flight for future reference.

5.3.3 Transformations

Because the camera has a fixed mounting in simulation, on the s1000 and 3DR, the transformation matrix can be calculated beforehand and added to the implementation. Should the camera be mounted on a controllable gimbal in the future, the camera task should be expanded to consume a message with information about the gimbal pose and calculate the transformation matrix when updated.

FlightGear Simulation

The camera view in flight gear is a virtual camera mounted centered 10cm underneath the copter. The body frame in Flight Gear has a positive x out of the head of the copter and positive y direction out the right wing. To get from camera to body frame only a simple rotation of 90 degrees about the z-axis is needed.

$$\mathbf{T}_c^b = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

3DR Solo

The camera on the 3DR solo is mounted on a gimbal which was set to replicate the view in the FlightGear simulations for consistency between simulation and field testing. Therefor the same simple rotation of 90 degrees about the z-axis is needed to transform the ArUco position in camera frame to body frame.

$$\mathbf{T}_c^b = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

5.3.4 Vendor Libraries in DUNE

External libraries are included in DUNE by placing the relevant header and source files for a library into ”vendor/librares/*library name*” under the DUNE source folder. For inclusion of the ArUco library source files provided by [26], the files would be placed into ”vendor/libraries/ArUco”. To link these source files to a desired task, a CMake configuration file has to be created and placed under the ”vendor/libraries/ArUco” path. Each vendor library has this CMake configuration with the required name ”Library.cmake”. The CMake rules needed for the ArUco library were added such that no external dependencies except OpenCV are needed when building the source code.

Chapter 6

Experiments and Results

6.1 Small Scale Implementation - Tello

6.1.1 Description

The aim of the experiment is to look at the viability of processing the ArUco detection algorithm while navigating based on the position estimates in real time. The vehicle used for this experiment will be the miniature drone Tello 4.2.3. For guidance the constant bearing method detailed in 3.3.2 is used to create the desired body frame velocities as input to the miniature drone. If this composite experiment is successful, the system implemented for the miniature drone will serve as the foundation of the DUNE implementation.

The miniature drone is used to track an ArUco marker lying on the ground. The camera parameters are found by the same calibration method as discussed in 5.1.5 Then by using the detection algorithm discussed in 5.1.2, the ArUco marker is identified when inside the camera view and the Tello moves towards the marker by the velocity inputs generated by the constant bearing method discussed in 3.3.2.

Two test will be performed where one is hovering at a desired altitude over the marker, and the second one being landing on the marker. There is a firmware enforced limit on the miniature Tello drone of not being able to descend to a lower altitude than about 40cm if not executing a landing maneuver. Thus a certain amount of consecutive frames inside a predefined threshold will server as the criteria for a landing maneuver is automatically sent. The distinction between the hovering and landing test is the loss of marker data as the drone moves closely to the marker.

6.1.2 Setup

The hardware required for this experiment is

- DJI Ryze Tello with mirror

- ArUco Marker
- Computer.

The drone and the marker is pictured below. The embedded ArUco marker has a side length of 8cm for reference.

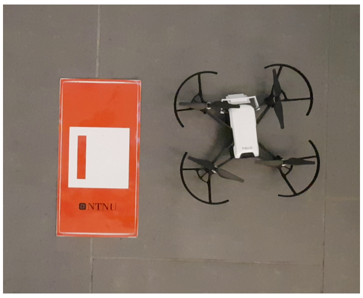


Figure 6.1: Miniature Drone and ArUco Marker

The flight data is logged to file with reference to time at when the takeoff maneuver is finished. This is also when the main loop of the program is started.

6.1.3 Calibration

Calibration is done according to the method described in 5.1.5 A total amount of 21 pictures were gathered from the miniature drone video stream at different heights and angles. From these pictures the following camera matrix and distortion coefficients were identified:

$$P = \begin{bmatrix} 930.074 & 0 & 462.764 \\ 0 & 964.503 & 391.505 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.1}$$

$$dc = \begin{bmatrix} 0.09337 & -2.17267 & 0.017468 & -0.00093258 & 11.4034 \end{bmatrix} \tag{6.2}$$

with a overall mean reprojection error of 0.526 pixels for the calibration set.

6.1.4 Results

Image Corruption and Artifacts

The image quality of the video stream from the Tello drone is periodically heavily corrupted. A showcase of the best and corrupted image data is given in the figure below.
These pictures are in sequence and shows that the corruption of the image can spontaneously cause a jump of estimated position of the marker. To compensate for this, a low pass filter was introduced to remove sudden changes in marker position estimate based on the estimated velocity of position changes.

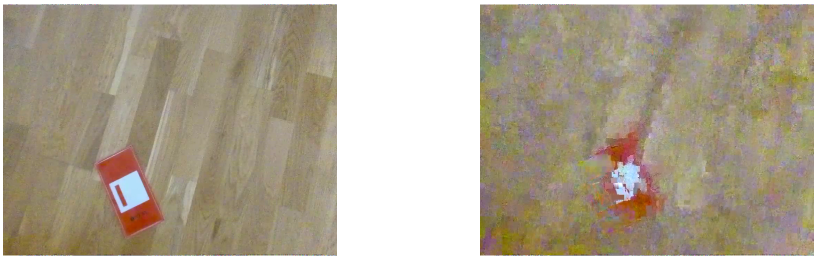


Figure 6.2: Tello Video Feed Corrupted vs Normal Quality

Dead Reckoning

The image corruption also means that the drone will be dead reckoning spontaneously. To handle the loss of estimate data, a timer was added which elapses if no new detections are made within a predefined time of 1 second. Each time a new detection is done, the timer is reset. If the timer elapses the velocity of the drone is set to zero. In almost every case of dead reckoning the marker was still inside the camera view, but not detectable due to the image corruption. Thus not moving until the image quality improved was often enough to recover and continue execution.

Stable Hovering

The aim of this test was to center itself directly above the ArUco marker lying on the ground while maintaining a predefined altitude of 1m above the marker. The following data presents the position of the ArUco marker expressed in body frame.

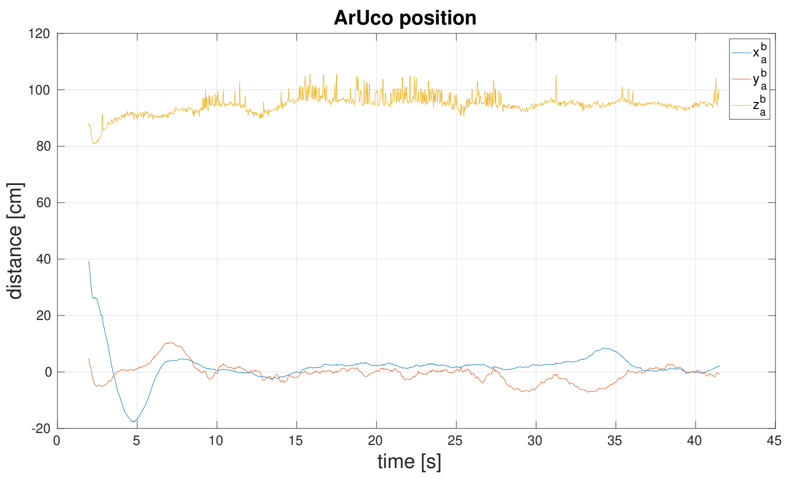


Figure 6.3: Relative ArUco position

The sharp value changes in the chart occurs when there is a loss of marker tracking. As mentioned earlier, due to image corruption this happens fairly often. Instead of hovering at 1m above the ArUco marker, the drone hovered at about 95cm. The build in firmware of the Tello drone tries to maintain the drone at whatever height and position it is currently at if no input is given. This is not perfect at the drone deviates a lot when there is no input as well, meaning when there is a loss of the ArUco marker, the drone may start to deviate as seen in the graph.

Landing

The aim of this test was to approach the ArUco marker while maintaining the marker within the camera view. The threshold ranges for landing was a horizontal radius of 2cm and an altitude below 40cm. After 5 consecutive frames within this threshold, a landing maneuver was sent and the flight terminated. The flight position data are presented in the figure below.

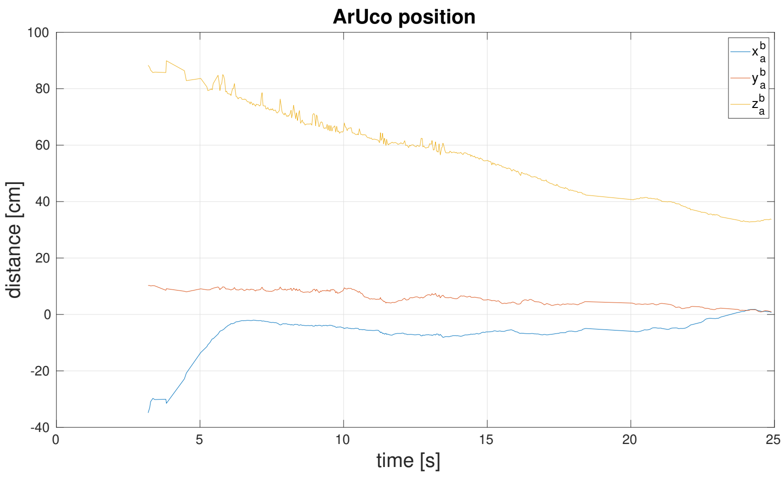


Figure 6.4: ArUco Position in body frame

The drone kept a steady approach, but often limiting movement in 3 axis simultaneously. This occurred because even though the API for the drone supports small velocity set points, there was some thresholds before the drone would start moving. As the max velocity of the velocity referenced generated by the constant bearing method is capped at a desired level, the lesser error axes were given a lesser velocity reference. This caused the movement to be prioritized in axes with larger error until they became about leveled.

6.1.5 Discussion

After handling the image corruption and dead reckoning, the drone made steady approaches to its target in both the landing and hovering experiment resulting in successful execution. The successful detection of the ArUco marker in the high noise environment serves as a proof of concept for guidance based on ArUco markers. This can be seen as an worst case test, as the image quality from the camera on a full size test drone will be substantially better. Therefor the implementation was found worthwhile to also implement in DUNE for simulation trials and full size testing.

6.2 DUNE Implementation - Simulation

6.2.1 Description

To test the full DUNE implementation, a simulated run with ArduPilot in the loop and FlightGear as the camera source was done. An virtual ArUco marker was added closely to the initial drone position such that marker were visible to the drone after takeoff.

Similar to the previous test with the miniature drone, this experiment aims at using both detected rectangle and ArUco marker data to navigate the simulated drone into a controlled hovering altitude and landing on the marker.

6.2.2 Setup

Before starting the DUNE program, setup for ArduPilot and FlightGear is required.

FlightGear

FlightGear is launched by running the script `Tools/autotest/fg_quad_view.sh` which is located under the ArduPilot build folder. After executing the script the virtual camera is shown. To feed the camera view to the DUNE task, a parameter for video source of the Camera Task is set in the config file to be `http://localhost:8080/screenshot`.

ArduPilot

To launch ArduPilot, the current working directory has to be under the `ArduCopter` folder. Then by entering `sim_vehicle.py -w` ArduPilot is launched with the configuration in the `ArduCopter` folder. The trailing command `-w` is to wipe persistent data that may be stored between sessions. Via the MAVProxy terminal prompt that is created when running ArduPilot, the drone is firstly set in mode guided with the command `mode guided`. Then the drone is armed with the command `arm throttle` before finally sending the takeoff command `takeoff 15` which initiates the liftoff of the drone to an altitude of 15 meters above the takeoff position.

DUNE

The DUNE configuration file for vehicle needs to be adapted to the differences between the simulation and a real drone. All these differences can be set in the configuration files. This means parameters such as thresholding ranges, video source path, landing criteria etc. have to be customized for the simulation environment.

The configuration file for the simulation tests is provided in appendix A with name `flightgear.ini`. This file details all the specific parameters used in the experiment. The final step to start the experiment is to launch the DUNE program with this config file with ArduPilot as SITL. Specifically `./dunce -c flightgear -p AP-SIL`. The DUNE program starts to execute detection and maneuvering as soon as it's launched.

6.2.3 Results

To illustrate the results, the constant bearing target data dispatched in the DUNE system is displayed for altitude z and for horizontal positions x and y. The dashed vertical line indicates the transition from navigation based on detected marker rectangles to ArUco marker data.

Stable Hovering

Altitude data is shown below:

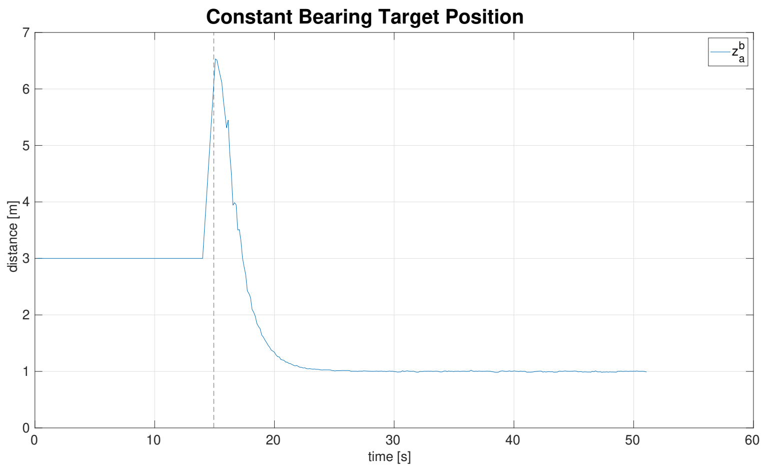


Figure 6.5: Simulation Hovering Altitude

Since there is no height information in the rectangle detection method, the drone is assumed to be 3m above the target, making the drone continuously descend until the fiducial detail of the ArUco marker is visible. As discussed in 5.3.2, the transition happens after 5 consecutive frames with ArUco marker information detected. An altitude of 6.5m over the ArUco marker seems to be sufficient to achieve a stable detection of the ArUco marker. The following graph shows the horizontal position:

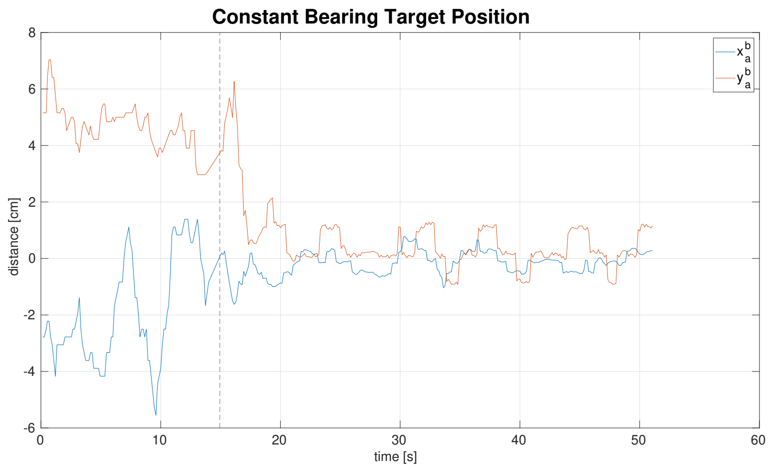


Figure 6.6: Simulation Hovering Horizontal Position

From the view provided by FlightGear, there is some quantization that is noticeable at sub centimeter positions. This causes the horizontal position of the drone to jump when approaching 0cm.

Landing

In addition to the dashed line indicating the switch from rectangle detection to ArUco detection, the second dashed line indicates the beginning of the landing state. The altitude data of the landing is given in the graph below.

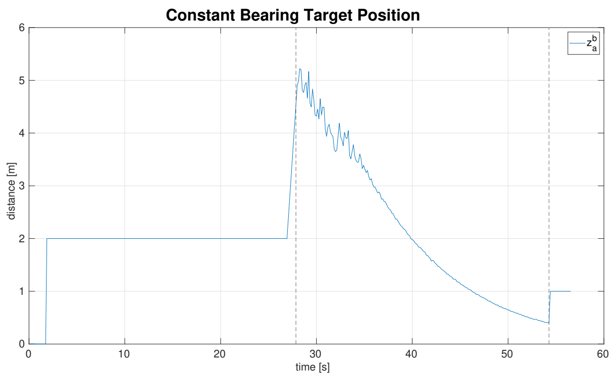


Figure 6.7: Simulation Landing Altitude

When switching to the landing state, the constant bearing target altitude is given constantly as 1m below the drone, causing the drone to move assertive to the ground.

The horizontal data is given in the graph below.

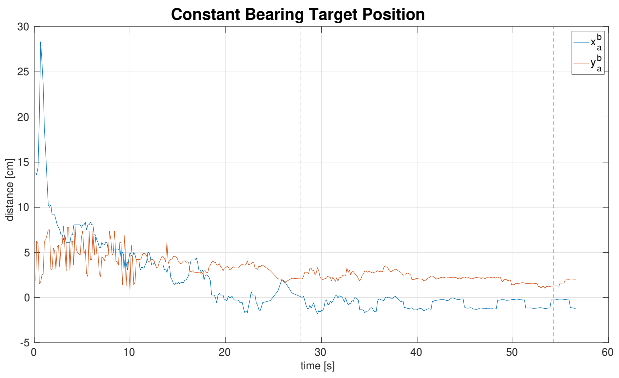


Figure 6.8: Simulation Landing Horizontal Position

Before entering the low altitude state, the ArUco marker position is fairly precise and when ArUco marker data is available the horizontal error is not much different from the pixel position. The landing takes place at an altitude of about 40cm where the horizontal error is within 2cm in both x and y.

6.2.4 Discussion

Some unexpected resolution boundaries in the simulated camera view were found to cause an inaccuracy in the final part of the approach. Being affected by this, the resulting position data rarely exceeded 2cm when stabilizing on hovering and when landing on the target. As the MUG fin is of size 20x10cm, an error of 2cm would be sufficient accuracy for testing the implementation with a full size drone such as the S1000. The discrete camera view positions will also not be a factor when doing field testing and thus these discrete value jumps will be absent in the a non-simulated implementation.

The simulation had a stable detection of the ArUco marker at 6.5m, but from runs with the Tello drone, the height for stable detection in the field test will probably be at much lower altitudes.

The experiment shows that the implementation provides a stable method for navigating the drone closely to the marker while not affected by disturbances. This simulation does not include wind disturbances, ocean currents or wave induced motion, which will be factors to consider when doing a full sized field test.

6.3 DUNE Implementation - 3DR

As the S1000 full size drone was not available for testing due to the circumstances, the DUNE implementation will be tested on the 3DR solo instead. A successful result with the 3DR testing will not mean that the same implementation would suffice for the S1000. The S1000 differs in both weight and amount of rotors and hardware setup which will alter the viability of the implementation. The results are instead taken as strong indicators of the behavior of the implementation.

6.3.1 Description

With successful experiments with the DUNE implementation in simulation, the same implementation is to be tested with the 3DR solo drone affected by a real world environment. A notable difference from the OASYS project ocean environment, is that the 3DR solo does not have capabilities for landing on water and thus will be tested in a field instead. This means that wave induced motion and ocean currents will be absent in this test.

The DUNE program is ran on a GCS in this experiment and communication with the ArduPilot program running onboard the drone is proxied through MAVProxy. The aim of the experiment is to evaluate the performance of the implementation running on a GCS while communicating with the ArduPilot software running on the drone. Delays are expected as all communications happen through a WiFi access point. Compared to the simulation there will be wind disturbances and more elements in the video stream to differentiate the ArUco marker from. Three tests were done were the goal was to land the 3DR drone on the marker.

6.3.2 Setup

Hardware

- 3DR Solo
- GoPro Hero4
- Laptop as GCS
- ArUco Marker

The remote controller for the 3DR solo works as a WiFi access point. The GCS and the drone are both connected to this AP.

Software

Some software setup is required for enabling external computing and the data communication between the GCS and the 3DR solo drone. The software setup nodes are illustrated in appendix C.

The video feed from the GoPro is duplicated on the GCS to avoid buffering and DUNE grabs frame from the duplicated stream. The MAVLink messages DUNE intends to send to the drone is proxied through a local MAVProxy session and then to the ArduPilot.

The specific flight parameters used for this experiment are detailed in the `3DR.ini` file included in appendix A. The altitude threshold for landing was set to 40cm in for the these flights.

6.3.3 Results

This section will only present the data from the second flight. The data plots from of other flights are included in appendix D.

The filter implementation worked well in the lighting conditions in the 3 flights, and the marker shape was consistently tracked from the start of the flights. The first and second flight entered the landing phase, but the last flight did not. The positional data of the ArUco marker is presented below. The first dashed line represents the transition from high to low altitude approach, while the second dashed line represents the start of the landing phase.

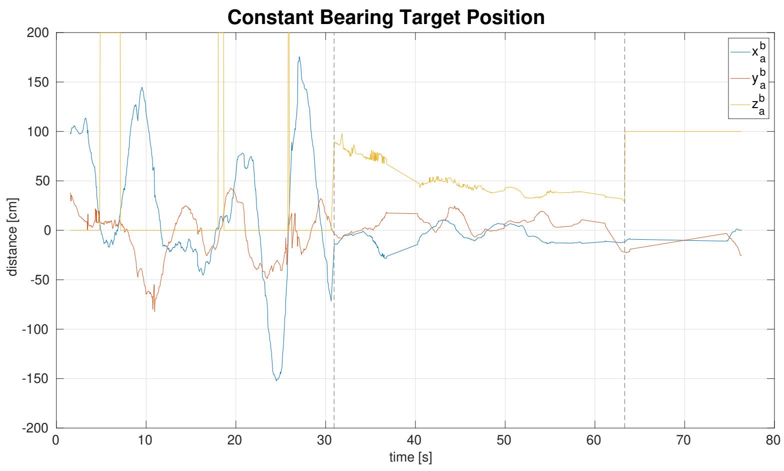


Figure 6.9: 3DR Constant Bearing Target Position

The horizontal position is increasingly oscillating around 0 while descending until the ArUco marker details are visible. After switching to low altitude approach at an altitude of 84cm, the horizontal position error is maintained within 20cm while the drone is slowly descending to the landing altitude threshold of 40cm. Shortly after meeting the landing criteria and entering the landing state, the marker is lost within the camera view. Although the drone was instructed to land, the drone did not come to a still at the ground. From the ArUco position data given in the graph below, the drone begins to ascend after failing to land for a while, and thus new ArUco measurements are seen after landing. The horizontal error when reappearing is initially very low, implying a likelihood of a potential successful landing had the drone stayed at the ground.

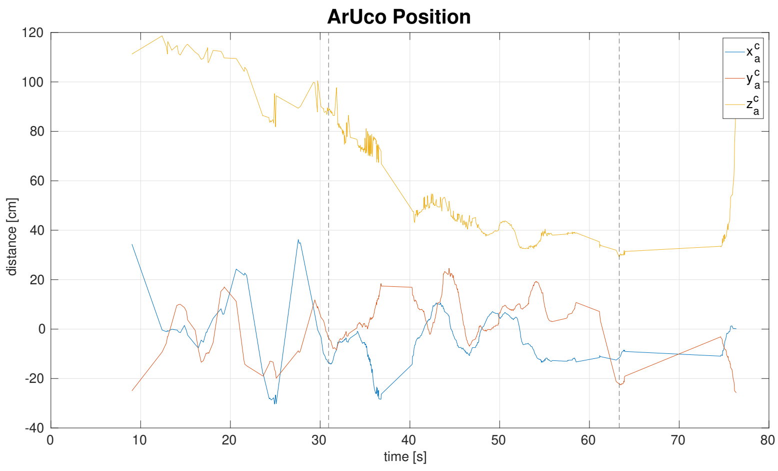


Figure 6.10: 3DR ArUco Position

From figure 6.10 the first ArUco detection is at an altitude of about 110cm above the marker. 20 seconds elapsed from the first detection until the ArUco detection updates were consistent enough to switch to low altitude approach.

Reviewing the data points after the flight, the video stream was consistently 25Hz as configured in the `3DR.ini` file in appendix A. Pose estimation data was detected in 29.6% of all the frames received.

6.3.4 Discussion

Comparing the high altitude with the low altitude navigation, the high altitude navigation method based on pixel values seemed to cause a consistent overshoot that increased as the drone descended. After switching to the low altitude navigation, the motion of the drone was much stabler. This would imply that the navigation method based on the pixel location of the marker should be able to perform better, even if there was an influence of wind disturbance. This differs from the simulation experiment where the pixel method was reliable. In the simulation test the drone entered low altitude approach at 6.5m, as seen in figure 6.5, while in this flight the switch occurred at 84cm. This might have caused the navigation method to perform better in simulation.

In two of the flights the drone switched to the landing state, but the landing velocity was tuned too low. The z velocities were correctly dispatched after reviewing the logged data, but from the flight it seems this wasn't enough to actually move the drone downwards. This caused the drone to hover when it should have asserted a landing. This seems to be similar to the Tello drone where too low velocity references didn't cause any movement. This should be a quick fix of increasing the constant bearing guidance `U_a_max` in the landing stage.

Although the drone reached altitudes were the ArUco pose could be used in each of these 3 flights, a better high altitude navigation method would reduce the flight time. The instability could also have caused the drone to lose the marker within the camera view, which would be critical.

Post Flight Filter Analysis

In post review of the filter values used for the flights and the recorded video stream, new filter values were found which increased the percentage amount of ArUco markers detected from 29.6% to 58%. This would've increased the altitude the drone switched to low altitude approach from 84cm to 148cm. As the navigation performed much better based on the ArUco marker pose information, an earlier detection of the ArUco marker would greatly improve the marker approach.

Chapter 7

Discussion and Conclusion

To carry out part of the mission described by the OASYS project, a system with an appropriate state machine was formulated and implemented. The state machine was designed to handle the states from a MUG detection based on shape and color available at a high altitude to the landing on the MUG for pickup. The different modules for computer vision and guidance methods were implemented to realize the state machine. Supporting modules such as data logging were also implemented to inspect the behavior of the system in post analysis.

A fast and iterative platform for testing independent algorithms were made for a miniature drone. Including programmatic video decoding, computer vision and guidance methods. This platform was used as a proof of concept before transitioning the algorithms to the DUNE implementation.

A setup for a simulated environment was created to enable testing of the full DUNE implementation. Most importantly, the simulation environment included an artificial camera view cohering to the simulated position of the drone. ArUco markers could be placed in this artificial scene and thus provided simulation basis for a full implementation test.
Field tests with the intended DJI S1000 drone were not done because of the national corona lockdown initiated March 12, 2020. Instead, three flight tests with an alternative drone were done. The experiment was performed on ground and not in a ocean environment, and instead of onboard computing the implemented system was ran externally on a GCS.

The implemented system in DUNE was able to identify the ArUco marker shape at altitudes of 5m, and the ArUco fiducial details at 1m. The navigation method was able to keep the ArUco marker within view from first shape detection until an altitude of 40cm where landing was attempted. The drone was not able to complete the landing and stay on the ground. In the flights were landing was attempted, the horizontal position error was small and did not exceed 20cm. This result was considered reasonable, as no wind disturbance model was included in the test. The ArUco markers served as reliable pose estimators

while the constant bearing guidance method could use some tuning.
Because of the external computing, performance capability increased at the cost of minor latency in the communication with the drone. Higher processing of the image data was achieved which enabled faster frame rate and higher resolution than what an onboard system could achieve. With 5G support at offshore distances, fast data rates at low latency for mobile units will be available and makes a case for doing external computing with the S1000 drone as well.

7.1 Further Work

The field test in this thesis were conducted on land, and thus testing in a sea environment which is the intended environment remains. These conditions will add factors as ocean currents and wave induced motions for which this thesis has not covered, but will be essential for a higher success percentage. As testing with the DJI S1000 drone was not possible, this also remains as further work. The next step of this thesis would be to tune the constant bearing guidance velocity outputs by doing more field tests, and implementing models for wind and ocean disturbances.

With the transition to external computing, detection methods using trained neural networks to detect the ArUco marker shape might be worth exploring. Furthermore, the system implemented in DUNE has to be expanded to handle the initial search of the MUG after takeoff and attaching the MUG to the UAV with the electropermanent magnet.

Bibliography

[1] Beaglebone black - wiki.
https://elinux.org/Beagleboard:BeagleBoneBlack. Accessed February, 2020.

[2] Open grab epm v3 - wiki.
https://kb.zubax.com/display/MAINKB/OpenGrab+EPM+v3. Accessed January, 2020.

[3] Withrobot - ocam-1cgn-u-t.
http://withrobot.com/en/camera/ocam-1cgn-u-t/. Accessed January, 2020.

[4] Ardupilot - mavlink interface.
https://ardupilot.org/dev/docs/mavlink-basics.html, Accessed January, 2020.

[5] Ardupilot - software.
https://ardupilot.org/index.php/about, Accessed January, 2020.

[6] Dune - unified navigation environment.
https://lsts.fe.up.pt/toolchain/dune, Accessed January, 2020.

[7] Mavlink - about.
https://mavlink.io/en, Accessed January, 2020.

[8] Mavproxy - about.
https://ardupilot.org/mavproxy/index.html, Accessed January, 2020.

[9] Oasys - home.
https://blogg.hioa.no/oasys/, Accessed January, 2020.

[10] Opencv - about.
https://opencv.org/about/, Accessed January, 2020.

[11] S1000 - user manual.
http://dl.djicdn.com/downloads/s1000/en/S1000_User_Manual_v1.10_en.pdf, Accessed January, 2020.

[12] Tello - sdk 2.0 user guide.
https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf, Accessed January, 2020.

[13] Tello - specs.
https://www.ryzerobotics.com/tello/specs, Accessed January, 2020.

[14] Aruco library source.
https://sourceforge.net/projects/aruco/files/, Accessed June, 2020.

[15] Gopro hero4 - docs.
https://gopro.com/en/us/update/hero4, Accessed June, 2020.

[16] Imc protocl docs.
https://www.lsts.pt/docs/imc/master/, Accessed June, 2020.

[17] Open solo 4 - wiki.
https://github.com/OpenSolo/OpenSolo/wiki, Accessed June, 2020.

[18] Pixhawk 4 - docs.
https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk4.html, Accessed June, 2020.

[19] 3dr - about.
https://www.3dr.com/company/about-3dr/, Accessed March, 2020.

[20] Ffmpeg - software.
https://ffmpeg.org/about.html, Accessed March, 2020.

[21] Aleksander Asp and Marius Eskedal. Ocean surface pickup with multirotor drone. 2020.

[22] Paulo Dias. Dune - home.
https://github.com/LSTS/dune/wiki, Accessed February, 2020.

[23] Yi Feng, Cong Zhang, Stanley Baek, Samir Rawashdeh, and Alireza Mohammadi. Autonomous landing of a uav on a moving platform using model predictive control. *MDPI*, 2018.

[24] Douglas Fields. Galilean relativity. University of New Mexico, 2015.

[25] Thor I. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control*. Wiley, 2011.

[26] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco Madrid-Cuevas, and Rafael Medina-Carnicer. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition*, 51, 10 2015.

[27] Jens Ludvik Grytnes Joberg. Multirotor pickup of object in the sea. 2019.

[28] Kevin Ling, Derek Chow, Arun Das, , and Steven L. Waslander. Autonomous maritime landings for low-cost vtol aerial vehicles. In *2014 Canadian Conference on Computer and Robot Vision*, 2014.

[29] Riccardo Polvara, Sanjay Sharma, Jian Wan, Andrew Manning, and Robert Sutton. Vision-based autonomous landing of a quadrotor on the perturbed deck of an unmanned surface vehicle. *MDPI*, 2018.

[30] Timothy W. McLain Randal W. Beard. *Small Unmanned Aircraft, Theory and Practice*. Princeton University Press, 2012.

[31] Francisco Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. Speeded up detection of squared fiducial markers. *Image and Vision Computing*, 76, 06 2018.

[32] Marwan Shaker, Mark N.R. Smith, Shigang Yue, and Tom Duckett. Vision-based landing of a simulated unmanned aerial vehicle with fast reinforcement learning. In *2010 International Conference on Emerging Security Technologies*, 2010.

Appendices

A Configuration Files

flightgear.ini

```
# Main ardupilot file
[Require uav/arducopter.ini]

# Disable original supervisor to avoid interference =============
[Supervisors.Vehicle]
Enabled                                    = Never
[Plan.Engine]
Enabled                                    = Never
[Maneuver.Multiplexer]
Enabled                                    = Never
[Plan.DB]
Enabled                                    = Never
[Plan.Generator]
Enabled                                    = Never
[Maneuver.CommsRelay]
Enabled                                    = Never
[Maneuver.FollowReference.UAV]
Enabled                                    = Never
[Simulators.GPSRTK]
Enabled                                    = Never


# Marius Eskedal Specific
# ==================================================================

[Supervisors.ssp]
Enabled                                    = Always
Entity Label                               = State Machine
Hover Altitude                             = 0 ; 0 => land on marker
Landing Mode Altitude                      = 0.4
Debug Level                                = Debug

[General]
Vehicle                                    = ntnu-hexa-003
Time Of Arrival Factor                     = 3

[Control.UAV.Ardupilot/AP-SIL]
Enabled                                    = AP-SIL
; Ardupilot Tracker                          = False
Ardupilot Tracker                          = False
Debug Level                                = Debug
Convert MSL to WGS84 height                = True
TCP - Address                              = 127.0.0.1
TCP - Port                                 = 5762

[Sensors.Camera]
Enabled                                    = Always
```

```
Entity Label                       = Camera Tracking
Camera Source                      = http://localhost:8080/screenshot ;
↪  <- Flight Gear Src
Flight Gear                        = True ; This recaptures the feed on
↪  every loop
Window Output Scale                = 0.6
; Force Input Size                  = 1280 720
Filter HSV low                     = 0, 100, 100
Filter HSV high                    = 180, 255, 255
ArUco Correction Rate              = 0.2
ArUco Marker Size                  = 0.08
Save Video                         = True
Camera Matrix                      = 1.8842924457197389e+03 0 640,
                                     0 1.8128540206355172e+03 360,
                                     0 0 1
Distortion Coefficients            = 0 0 0 0 0
Debug Level                        = Debug
Execution Frequency                = 25

[Supervisors.Logger]
Enabled                            = Always
Entity Label                       = Data Logger
Log CBT                            = True
Log CT                             = True
Log LP                             = True
Log LV                             = True
Log SM                             = True
Log DV                             = True
Log DC                             = True
Log AT                             = True
Debug Level                        = Debug

[Control.UAV.Ardupilot/Hardware]
Enabled                            = Never
Ardupilot Tracker                  = False
Debug Level                        = Debug
#TCP - Address                      = 127.0.0.1
#TCP - Port                         = 5762

[Control.Path.ConstantBearing]
Enabled                            = Always
Execution Frequency                = 10
Entity Label                       = Constant Bearing

[Navigation.UAV.Navigation]
Use RTK If Available               = False

[Simulators.Simple]
Initial Position                   = 10,5,0

[Transports.Ardupilot/AP-SIL]
Enabled                            = AP-SIL
Entity Label                       = Sitl Layer
SITL - Port Out                    = 5763
Debug level                        = Spew
```

3DR.ini

```
# Main ardupilot file
[Require uav/arducopter.ini]

# Disable original supervisor to avoid interference =============
[Supervisors.Vehicle]
Enabled                             = Never
[Plan.Engine]
Enabled                             = Never
[Maneuver.Multiplexer]
Enabled                             = Never
[Plan.DB]
Enabled                             = Never
[Plan.Generator]
Enabled                             = Never
[Maneuver.CommsRelay]
Enabled                             = Never
[Maneuver.FollowReference.UAV]
Enabled                             = Never
[Simulators.GPSRTK]
Enabled                             = Never


# Marius Eskedal Specific
# ==================================================================

[Supervisors.ssp]
Enabled                             = Always
Entity Label                        = State Machine
Hover Altitude                      = 0 ; 0 => land on marker
Landing Mode Altitude               = 0.4
Debug Level                         = Debug

[General]
Vehicle                             = ntnu-hexa-003
Time Of Arrival Factor              = 3

[Control.UAV.Ardupilot/AP-SIL]
Enabled                             = AP-SIL
; Ardupilot Tracker                   = False
Ardupilot Tracker                   = False
Debug Level                         = Debug
Convert MSL to WGS84 height         = True
TCP - Address                       = 127.0.0.1
TCP - Port                          = 5762

[Sensors.Camera]
Enabled                             = Always
Entity Label                        = Camera Tracking
Camera Source                       = http://127.0.0.1:8080
Flight Gear                         = False ; This recaptures the feed on
↪   every loop, Assumed to only be needed for FG
Window Output Scale                 = 0.6
Filter HSV low                      = 0, 90, 100
Filter HSV high                     = 180, 255, 255
ArUco Correction Rate               = 0.2
ArUco Marker Size                   = 0.08
```

```
Save Video                          = True
Camera Matrix                       = 762.2507610259398 0
↪   651.586731790949,
                                      0 767.0199707973958
↪   387.8374003362774,
                                      0 0 1
Distortion Coefficients             = -0.3304015894835765
↪   0.2005034994112137 -0.007653633455265741 0.003153903391754909
↪   -0.095984363467503250
Debug Level                         = Debug
Execution Frequency                 = 25

[Supervisors.Logger]
Enabled                             = Always
Entity Label                        = Data Logger
Log CBT                             = True
Log CT                              = True
Log LP                              = True
Log LV                              = True
Log SM                              = True
Log DV                              = True
Log DC                              = True
Log AT                              = True
Debug Level                         = Debug

[Control.UAV.Ardupilot/Hardware]
Enabled                             = Hardware
Entity Label                        = Autopilot
UDP - Address                       = 127.0.0.1
UDP - Port                          = 14550
Use External Nav Data               = False
Debug Level                         = Debug
Ardupilot Tracker                   = False

[Control.Path.ConstantBearing]
Enabled                             = Always
Execution Frequency                 = 10
Entity Label                        = Constant Bearing

[Navigation.UAV.Navigation]
Use RTK If Available                = False

[Simulators.Simple]
Initial Position                    = 10,5,0

[Transports.Ardupilot/AP-SIL]
Enabled                             = AP-SIL
Entity Label                        = Sitl Layer
SITL - Port Out                     = 5763
Debug level                         = Spew
```

B Flow Diagrams

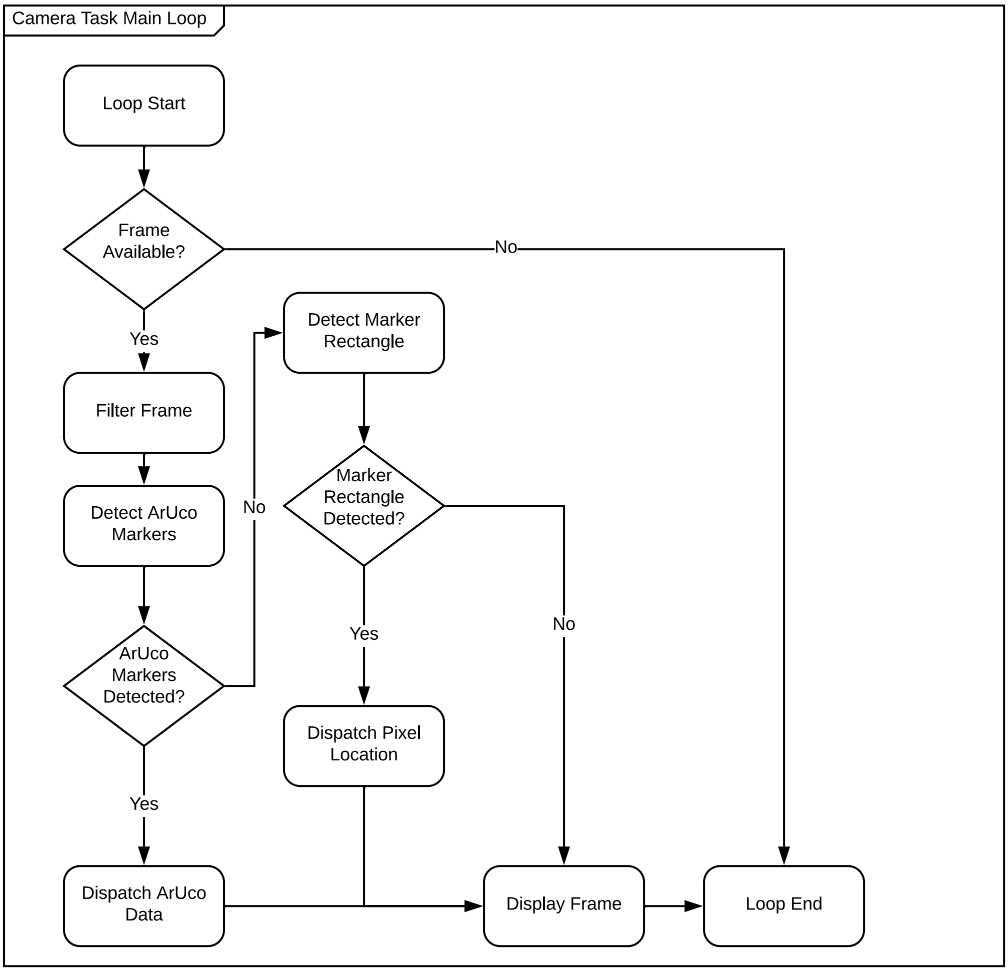


Figure 1: Camera Task Main Loop

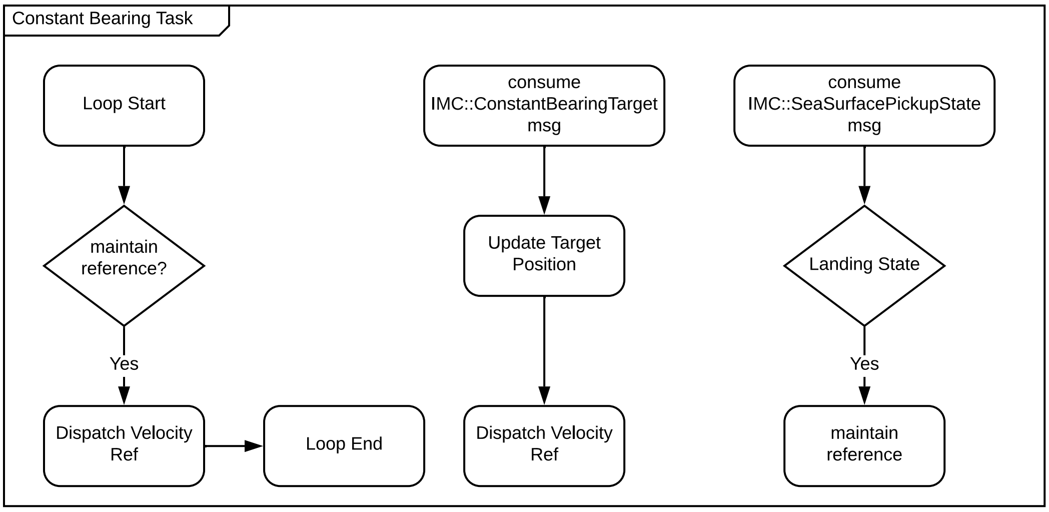


Figure 2: Constant Bearing Task

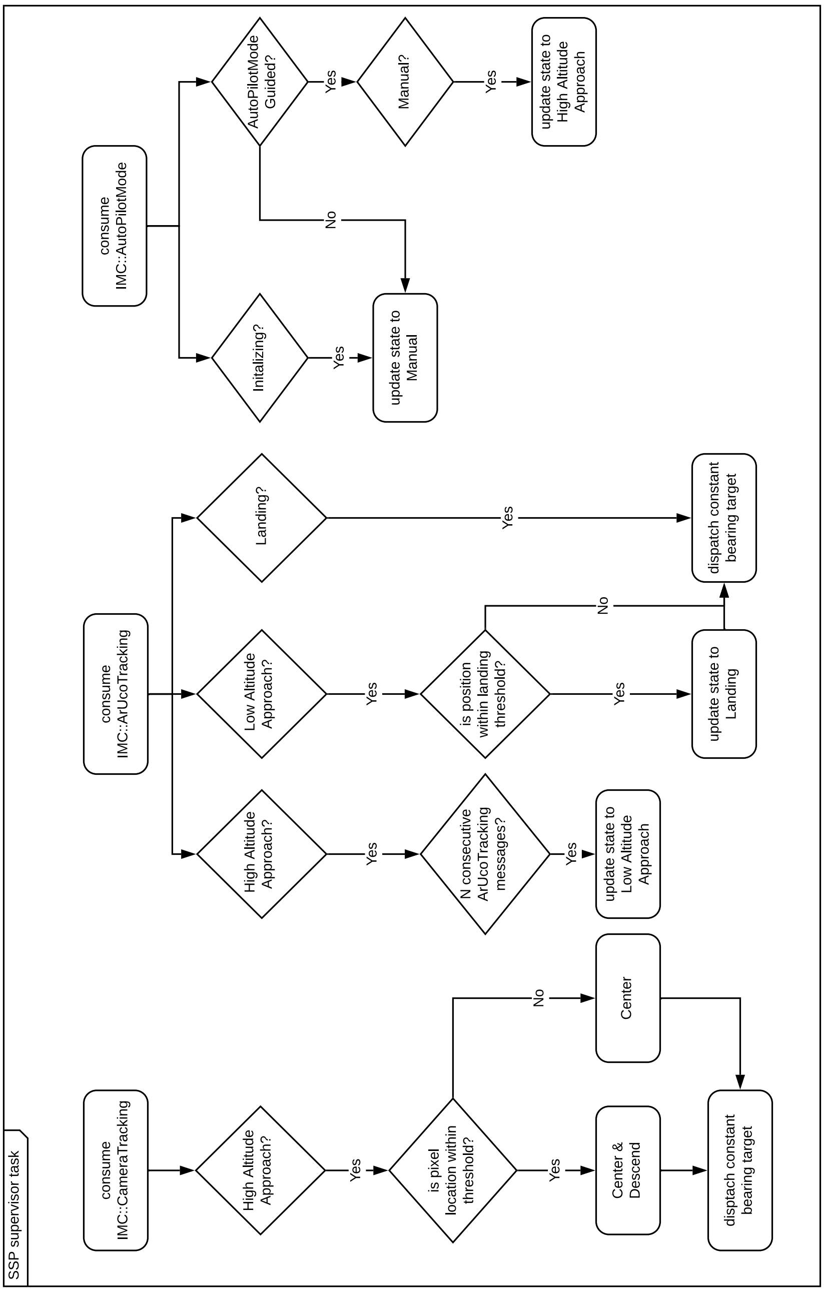

SSP supervisor task
consume IMC::CameraTracking
High Altitude Approach?
Yes
is pixel location within threshold?
Yes
No
Center & Descend
Center
disptach constant bearing target
consume IMC::ArUcoTracking
High Altitude Approach?
Yes
N consecutive ArUcoTracking messages?
Yes
update state to Low Altitude Approach
Low Altitude Approach?
Yes
is position within landing threshold?
Yes
No
update state to Landing
Landing?
Yes
dispatch constant bearing target
consume IMC::AutoPilotMode
Initalizing?
Yes
update state to Manual
AutoPilotMode Guided?
No
Yes
Manual?
Yes
update state to High Altitude Approach

Figure 3: SeaSurfacePickup Supervisor Task

C 3DR Software Setup

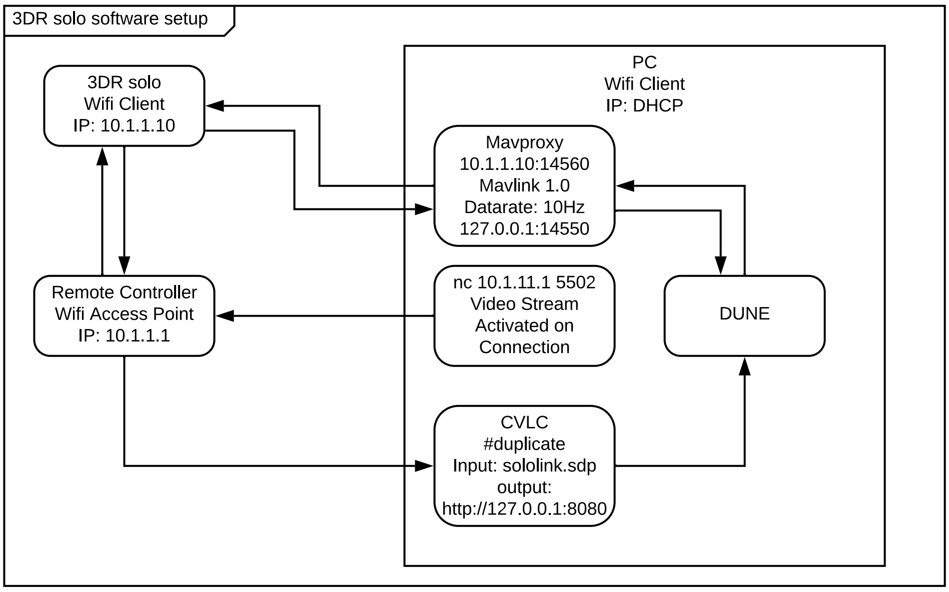


Figure 4: 3DR Solo Software Setup

D Field Test Figures

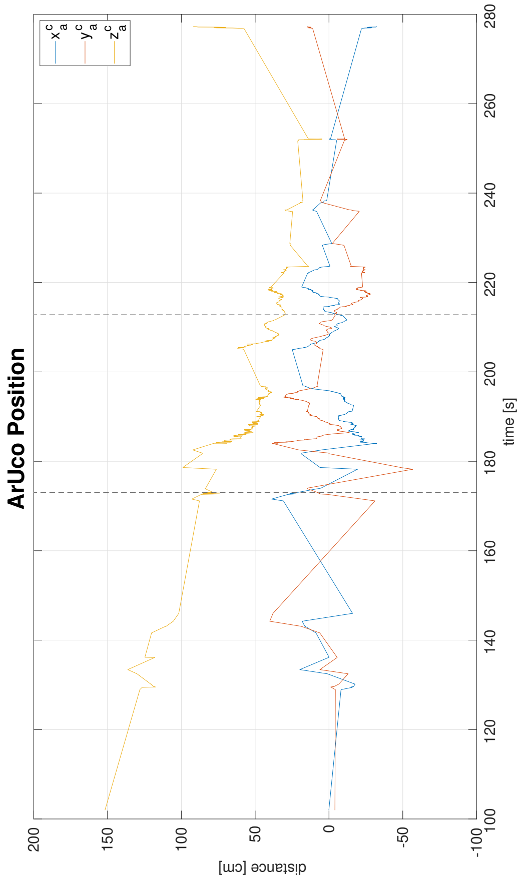


Figure 5: 3DR ArUco Position Flight 1

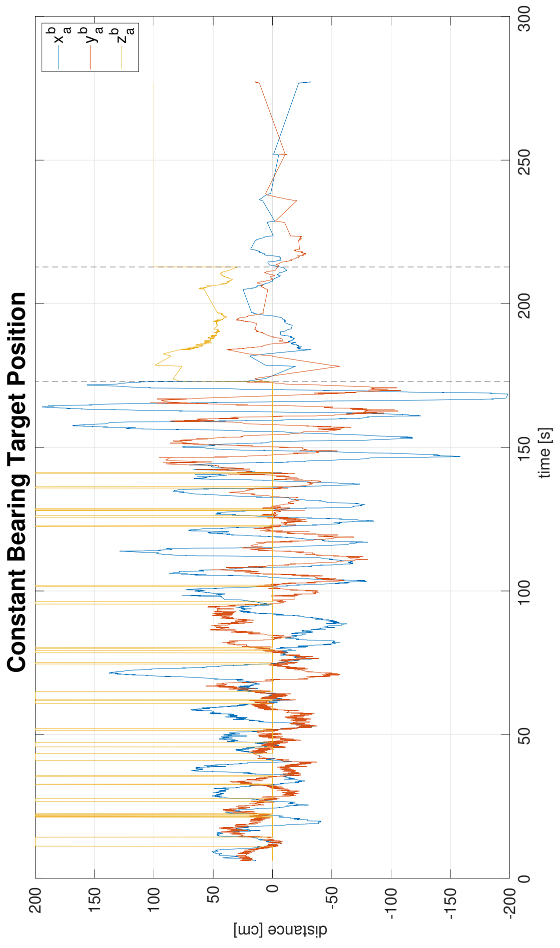


Figure 6: 3DR Constant Bearing Target Position Flight 1

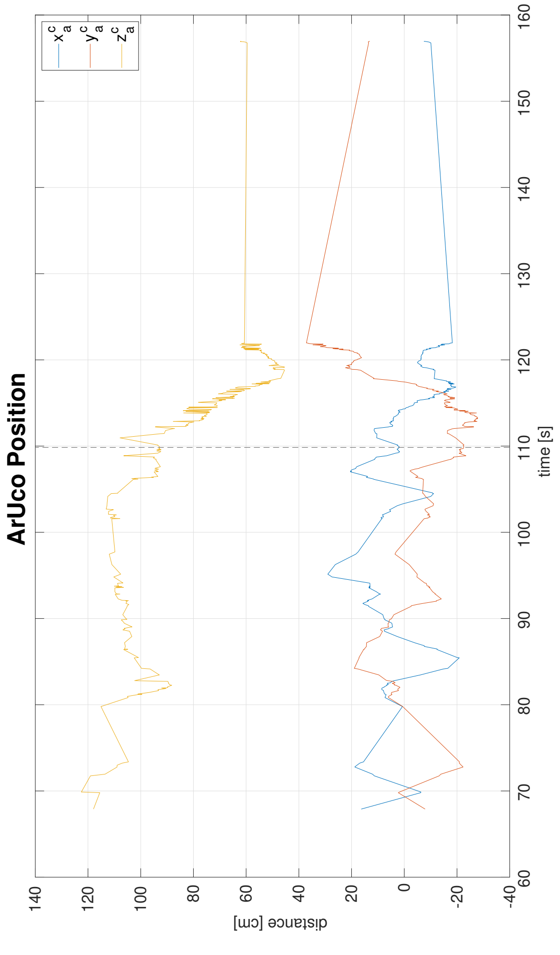


Figure 7: 3DR ArUco Position Flight 3

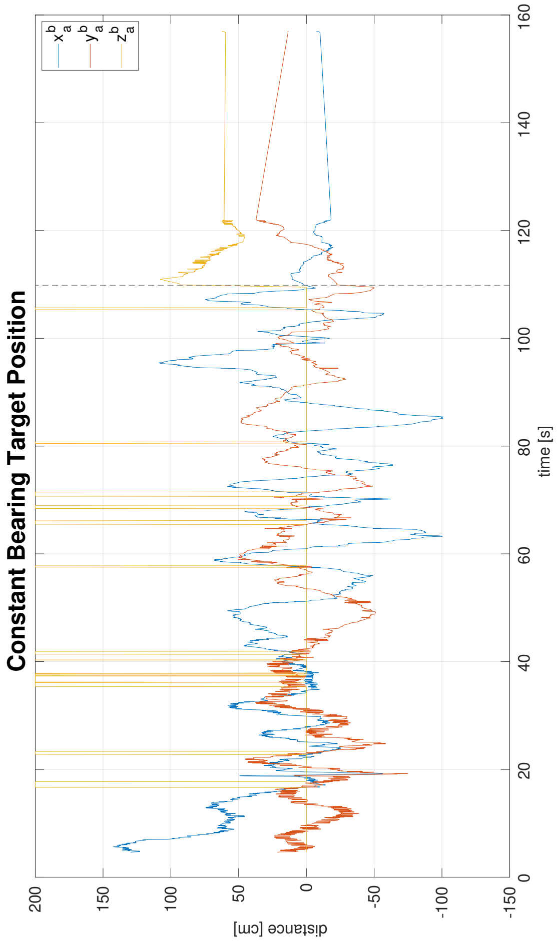


Figure 8: 3DR Constant Bearing Target Position Flight 3