Jens Ludvik Grytnes Joberg

Multirotor pickup of object in the sea

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Arne Johansen
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Jens Ludvik Grytnes Joberg

Multirotor pickup of object in the sea

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Arne Johansen
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Preface

This thesis concludes my work to become a Master of Technology in Cybernetics and Robotics at the Norwegian University of Science and Technology (NTNU) in Trondheim. The work was performed in the spring of 2019.

I would like to thank my supervisor, Tor Arne Johansen, for providing help with the direction and focus areas necessary to realize the project. My co-supervisor, Artur Piotr Zolich, has been very helpful with providing assistance in setting up the hardware used for experiments. He has also provided me with insight into the challenges that needed to be dealt with in order to perform tests. Furthermore, I would like to thank Pål Kvaløy for piloting the multirotor during testing. Finally, I would like to thank my girlfriend, Tonje, for providing me with support and care when I've been to busy to care for my self.

Abstract

This project aims to to land a multirotor unmanned aerial vehicle (UAV) on a small object floating at sea. The object is detected using a camera, while the elevation of the sea surface is measured by a radar. By triangulating the altitude estimate and the inverse projection of the camera detection, a position estimate is made.

The object is perturbed by the sea state, and the position estimates are to be used as input to a model based prediction filter. A constant velocity Kalman filter has been implemented to estimate motion induced by a uniform sea current .

A state machine governing the maneuvers necessary to perform the mission is implemented. An architecture utilizing DUNE and ArduPilot software systems is proposed to realize the state machine functionality. The architecture has been implemented in DUNE.

Libraries for interfacing an X4M03 radar and a Withrobot oCam camera has been configured to be integrated with DUNE.

Tests of a constant bearing guidance controller has been tested in both simulations and flights. While the accuracy in the simulation was within a few centimeters, the flight tests showed error in the scale of tens of centimeters.

Sammendrag

Dette prosjektet sikter på å lande en multirotor (UAV) på et lite objekt som flyter i sjøen. Objektet detekteres ved hjelp av et kamera, mens UAV'ens høyde over vannet måles med radar. Ved å triangulere høydeestimatet og den inverse projeksjonen av kameradeteksjonen kan det gjøres et posisjonsestimat.

Objektet forstyrres av sjøens tilsand, og kameraets posisjonsestimater skal brukes som input til et modellbasert filter. En konstant hastighet Kalman filter har blitt implementert for å estimere en uniform strømningsdel av havtilstandsbevegelsen. Et konstant hastighets Kalman filter er implementert for å estimere bevegelsen foråsaket av en unifor vannstrøm.

En tilstandsmaskin er implementert for å styre manøvrene som er nødvendig for å gjennomføre det satte målet. En arkitektur som benytter DUNE og ArduPilot sofware-systemer er forslått for å realisere tilstandsmaskinens funksjonalitet. Arkitekturen har blitt implementert i DUNE.

Biblioteker for samordning av en X4M03 radar og et Withrobot oCam kamera har blitt konfigurert slik at det er integrert i DUNE.

Tester for en konstant peiling veiledingskontroller har vært utført i både simuleringer of flyvninger. Til tross for at presisjonen i simuleringen var innenfor noen få centimeter, viste flyvningstestene feil i en skala av titalls centimeter.

Contents

List of Tables

List of Figures

Abbreviations

Symbol	=	definition
ECEF	=	Earth Centerd Earth Fixed (navigation frame)
GNSS	=	Global Navigation Satellite System
GPS	=	Global Positioning System
INS	=	Inertial Navigation System
MAV	=	Miniature Aerial Vehicle
MUG	=	Miniature Underwater Glider
NED	=	North East Down (navigation frame)
UAV	=	Unmanned Aerial Vehicle

Chapter 1

Introduction

1.1 Problem overview

The purpose of this project is to land a multi-rotor UAV on an object floating at sea, and pick it up. In the scope of this project, the object is a Miniature Underwater Glider (MUG). The MUG is significantly smaller than the UAV, and an accurate landing is necessary. To do so, the multi-rotor is to be equipped with a camera and a radar altimeter, as well as an autopilot system with an integrated INS.

It is sensible to divide the problem into three tasks:

1. Measure the position of the MUG
2. Filter the state of the MUG and predict its motion
3. Control the UAV to approach and land on the MUG

1.1.1 Measurements of MUG position

As there will not be a communication link between the MUG and the UAV, position data contained in the MUG will not be available to the UAV. Computer vision is then the only way for the UAV to obtain position measurements of the MUG. Visual detection and position measurements can be simplified by applying a distinctive marker with known dimensions. Frameworks for such markers are provided by the ArUco module in OpenCV, and the AprilTag library by the APRIL Robotics lab at University of Michigan.

1.1.2 MUG motion state filter

The position measurements provided by a computer vision system will contain noise and inaccuracies. In addition, the MUG is subject to motion induced by the sea state. By modeling the motion of MUG, and applying a filter/observer, a more accurate position estimate can be obtained. Additionally, predictions of future states can be made by the observer. The predictions will add useful information to the planning of the landing maneuver. Furthermore, the camera vision system is likely to loose track of the target during the last few decimeters of the landing. When tracking is lost, the observer can be used to update the estimated state, although the uncertainty will increase with time.

1.1.3 UAV control system

In order to perform the maneuvers required in this project, an autonomous UAV control system has to be used. This control system must be able to communicate with the camera system. It must also either perform the MUG state filtering, or be able to communicate with the software performing the filtering. The control system also needs to perform inertial navigation, or communicate with a navigation module.

The control system should contain the logic necessary to search for, approach, and land on the MUG.

1.2 Literature review

There has been made several studies into autonomous landing of UAVs on both stationary and moving targets. Most of the studies are concerned with autonomous landing on land based applications, but some investigate landing on marine vessels as well.

1.2.1 Landing on land based targets

(Falanga et al., 2017) performs autonomous landing of a multirotor UAV on a mobile platform. Their UAV uses a camera and onboard computing to detect and measure the position of the landing target. In addition, the camera is used to perform visual odometry. Visual odometry is combined with inertial measurement units (IMU) to perform local navigation. Their landing target is a mobile land robot moving at constant speed, with a platform marked with a tag of known dimensions. They use an Extended Kalman Filter to estimate the position, velocity and orientation of the landing platform. The landing maneuver is planned using an optimization problem.

(Nguyen et al., 2018) performs a landing of a multirotor UAV on a static landing platform on the ground. The platform is marked with several AprilTag markers, and the GPS coordinate of the platform is known. The initial approach to the platform is performed based on GPS, while the landing uses computer vision.

(Line, 2018) performs autonomous landing on both ground based and maritime vehicles. The landing platform is marked with ArUco markers, and camera vision is used to perform pose estimation. The estimated pose is processed in a Kalman filter with a constant velocity assumption. The control logic is implemented in the programming framework Robotics Operating System (ROS).

Similar studies into multirotor UAV landing on visual targets on ground platforms are made by: (Lange et al., 2008), (Lange et al., 2009), (Borowczyk et al., 2017) and (Araar et al., 2017)

1.2.2 Landing on marine targets

There has been some studies into performing landing on targets at sea. Both (Ling et al., 2014) and (Frølich, 2015) investigates UAV landing maneuvers on marine vessels. These vessels are significantly larger than the UAV used. (Ling et al., 2014) performs a landing of a multirotor on a moving vessel, and approximates the velocity velocity to be constant. The initial low accuracy approach to the vessel is done using GPS measurements of the ship position. However, the final landing maneuver uses computer vision with an AprilTag marker on the platform.

On the other hand, (Frølich, 2015) aims to perform an autonomous landing of a fixed wing UAV in a catching net placed on a marine vessel. The UAV uses a neural network for time series forecasting of the ship position. The programming framework DUNE was used to write the logic of the autonomous control system. The DUNE system communicates with an autopilot suite, Ardupilot, which performs the control maneuvers commanded by DUNE.

The master thesis presented in (Lasson, 2018) is an initial study aimed at autonomous landing on a floating platform. However, no landing experiments were made, and the focus is mainly on detection and pose estimation of a marker in varying light conditions.

1.2.3 Relevance to this project

All the studies mentioned above considers landing on platforms larger than the UAV. A higher accuracy is likely to be required in this project, as the smaller target yields lower tolerance for error. Most of the studies also used some form of GPS information about the target location, which is not available in this project. The landing platforms were either assumed to be stationary, or moving at constant velocity. In this project, due to the motion induced by waves, a constant velocity approximation is likely to not be sufficient.

Although this projects differ somewhat from the studies above, the approach of using known visual markers to provide measurements seems sensible. The method of using a Kalman filter to improve on the camera vision measurements is applicable to this project as well, but the motion model should contain more information than a constant velocity assumption.

Chapter 2

Basic Theory

2.1 DUNE - Unified Navigation Environment

The DUNE section is reused from (Joberg, 2018)

DUNE is runtime framework for embedded software in C++, created by Underwater Systems and Technology Laboratory (LSTS) at the University of Porto. It acts as a combined task manager and message bus manager. DUNE offers a modular abstraction of tasks, where a task has a closed scope and contains standardized functions to be called by the task manager at specific events. The messages passed within DUNE are defined by the inter-module communication protocol (IMC). Upon building DUNE, a single executable is made, containing all tasks within it's source directory. Different tasks of DUNE are activated from the executable by using configuration files specifying its behaviour.

2.1.1 The DUNE task

A DUNE task is a struct, inheriting from either `DUNE::Tasks::Task` or `DUNE::Tasks::Periodic`, the latter for periodic tasks. Note that the c++ struct is similar to a public class and can have methods, as opposed to the struct in the C language. Persistent variables in the outermost scope of the task, accessible by all methods, are declared in the first lines of the struct. In the struct constructor, the persistent variables are initialized. In addition, the IMC message subscriptions are declared in the constructor by the line: `bind<IMC::[msg_type]>(this);`. According to the LSTS convention, the variables in persistent memory should contain the prefix `m_` , e.g. `m_var`.

Parameters in a task can be set using arguments from the configuration file. These parameters are typically desired to be in persistent memory, and should be assigned to a persistent variable in the task constructor. The configuration values are extracted to an argument struct by the function: `param(config_var_name, arg_struct)`

Methods called by DUNE

At specified events, the task manager will call specified methods of the task. The methods are listed below

- `consume(IMC::[msg_type])`
- `onUpdateParameters()`
- `onEntityReservation()`
- `onEntityResolution()`
- `onResourceAcquisition()`
- `onResourceInitialization()`
- `onResourceRelease()`
- `onMain()` for regular task
- `task()` for periodic tasks

For each IMC subscription there should be a method:
`consume(const IMC::[msg_type])` containing the actions to be performed at the reception of a message. When a tasks task publishes a message using the `dispatch(msg)` function, the task manager will call the consume method on activated tasks subscribed to said message type.

2.2 ArduPilot

The ArduPilot section is reused from (Joberg, 2018)

ArduPilot (ArduPilot, 2018) is an autopilot suite. It is a collection control systems aimed at several classes of unmanned vehicle control, including multi-rotor crafts. It contains low level control systems, as well as high level guidance algorithms. In addition, ArduPilot performs navigation. Some of the systems are intended to assist a remote operator, while others are intended to be used in full autonomy. To separate the control systems allowed to operate at a given time, a mode state is used.

Some relevant multi-rotor modes are stabilize, guided, and rtl (return to launch). The modes are described on the ArduPilot web page:
`http://ardupilot.org/copter/docs/flight-modes.html`

The controllers used in this project are accessible when ArduPilot is in guided mode. The commands availible in guided mode are documented on the ArduPilot web page:
`http://ardupilot.org/dev/docs/copter-commands-in-guided-mode.html`

2.2.1 MAVLink

ArduPilot employs the MAVLink protocol (MAVLink, 2018), which can be used to access ArduPilot functionality from external software systems. LSTS maintains a DUNE task, Ardupilot, which provides an abstraction of the MAVLink interface to the ArduPilot software. The Ardupilot task sets up a TCP socket to communicate with ArduPilot, and sends and receives messages using the MAVLink protocol. This task subscribes to and publishes various IMC messages, and handles their intended MAVLink use cases.

As of December 2018, on the LSTS DUNE master branch, the Ardupilot DUNE task contains the following multi-rotor functionalities:

- Publishing of navigation data (on IMC::EstimatedState)
- Publishing of ArduPilot mode (on IMC::AutopilotMode)
- Automatic takeoff and landing
- Way-point tracking
- Altitude control
- Vertical rate control
- Roll control
- Speed control
- Idle maneuver
- Arm/disarm craft

In addition, NTNU UAVLab has implemented some additional functionalities to the Ardupilot task in a private UAVLab repository. Of particular interest is the inclusion of control of desired acceleration in three dimensions. However, the ArduPilot documentation of commands in guided mode states that the acceleration command is not supported.

On the other hand, the documentation also states that velocity commands are accepted. Despite this, the Ardupilot DUNE task doesn't contain functionality for sending the relevant MAVLink message. An effort should then be made to implement the velocity command in the Ardupilot task.

The MAVLink message containing velocity and acceleration commands are:

- SET_POSITION_TARGET_LOCAL_NED

- SET_POSITION_TARGET_GLOBAL_INT

Figure 2.1 illustrates an example setup of the communication between DUNE and ArduPilot. In the example, DUNE is running on a Beagle Bone Black (BBB) and ArduPilot is running on a Pixhawk flight computer.

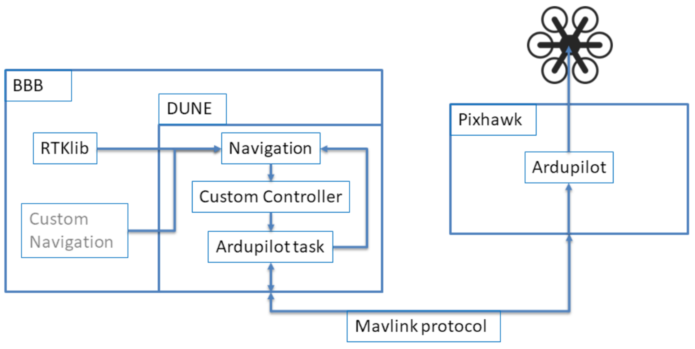


Figure 2.1: Communication diagram of DUNE running on a BBB and ArduPilot running on Pixhawk (Mads Bornebusch, 2018)

2.2.2 ArduPilot simulation

ArduPilot contains environments for simulation of several classes of unmanned systems, including multi-rotor systems. The simulations can be executed on a PC running Linux, Windows or OS X. Setup of the simulation environment on Linux is described on the ArduPilot web page `http://ardupilot.org/dev/docs/setting-up-sitl-on-linux.html`

2.3 Camera vision

The Camera vision section is reused from (Joberg, 2018)

2.3.1 Notation in camera vision

In this section several formats are used to express points, as both three dimensional (3D) and two dimensional (2D) coordinate systems are used. In addition, homogeneous coordinates are used to facilitate homogeneous transformations. Various forms of the letter P

are used to express points, while the letter d is used to express vectors. The notation used is described in table 2.1.

Case	Tilde	Point/vector	Example	Description
Lower	no	point	$p = [u,v]^t$	Image coordinate
Lower	yes	point	$\tilde{p} = [u,v,1]^t$	Homogeneous image coordinate
Upper	no	point	$P = [x,y,z]^t$	World coordinate
Upper	yes	point	$\tilde{P} = [x,y,z,1]^t$	Homogeneous world coordinate
-	no	vector	$d = [x,y,z]^t$	World vector
-	yes	vector	$\tilde{d} = [x,y,z,1]^t$	Homogeneous world vector

Table 2.1: Camera notation

Furthermore, subscripts are used to indicate instances, while superscripts are used to indicate which frame the instance is expressed in. For example, $\tilde{P}_o^C$ is the homogeneous coordinates of the world point P_o expressed in frame $\{C\}$

2.3.2 Camera projection

A camera image is the projection of a world scene, where the rays of light from objects passes through a lens and onto a surface grid. Each grid on the surface, a pixel, is exited by rays of light coming from a specific direction. Unless the ray experiences refraction, the ray direction is the same as the direction from the camera to the object it originated in. The relation between a given pixel in the image and the ray direction is described in the camera model. There are several camera models. The type of lens and model most suited to the application in this project is a pinhole camera model with geometric lens distortion.

The image projection of some point in a world frame depends on the intrinsic and extrinsic parameters of the camera. The intrinsic parameters define the camera model, and describe how a point, expressed in the camera frame, is projected onto the image. The extrinsic parameters define the pose of the camera in some other frame. Together, the parameters are used to form the projection of points expressed in the relevant frame.

Pinhole camera model

A pinhole camera model, also called perspective model, assumes light from the scene passes through a small hole and onto a captive euclidean surface. The pinhole serves the purpose of filtering out directions of light. In this process, the image will be inverted in its axes. However, this is easily fixed by flipping the image arrays, and can be done in post processing or by mapping the image surface grid with inverse axes. To model the image without inverted axes, similarity of triangles can be used to set the captive surface in front of the pinhole. The distance from the pinhole to the surface is called the focal distance.

Figure 2.2 shows the projection of a 3D point $P = [X,Y,Z]^T$ onto an image surface. The following properties are used (all points are expressed in $\{C\}$):

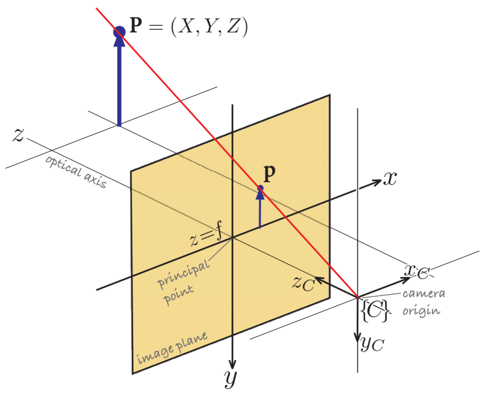


Figure 2.2: Pinhole projection, (Corke, 2017, p. 320)

- The origin, o, of the camera frame $\{C\}$, is in the pinhole.
- The z-axis goes through some point in the image at $[0,0,f]^T$, where f is the focal distance.
- The image surface is parallel to the XY-plane.
- The line from P to the origin of the camera frame, $\overline{OP}$ passes through the image surface at position $p = [x,y,f]^T$.
- The angle between $\overline{OP}$ and the ZY-plane is θ_x
- The angle between $\overline{OP}$ and the ZX-plane is θ_y.

The location of the point in the image can then be described by eqs. (2.1) and (2.2).

$$x = f\frac{X}{Z} = f\tan(\theta_x) \tag{2.1}$$
$$y = f\frac{Y}{Z} = f\tan(\theta_y) \tag{2.2}$$

The values of x and y will obtain the same unit as the focal distance. As we eventually will map the point to a pixel, it is sensible to use pixel units for the focal distance. There might be a difference in the x-dimension and y-dimension of a pixel, and this has to be addressed when representing the focal distance in pixels. f_x and f_y will be used to express the focal distance in x-pixels and y-pixels respectively. Thus, the expressions will be changed to

$$x = f_x\frac{X}{Z} = f_x\tan(\theta_x) \tag{2.3}$$
$$y = f_y\frac{Y}{Z} = f_y\tan(\theta_y) \tag{2.4}$$

The intersection between the camera frame z-axis and the image surface is referred to as the principal point, and is usually close to the image center. The xy-coordinates describe the location of a point relative to this principal point. In an image frame however, the origin is desired to be at the top left corner. Thus an offset has to be included, and u and v will be used to describe points relative to the image frame origin.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} x + c_x \\ y + c_y \end{bmatrix} = \begin{bmatrix} f_x \frac{X}{Z} + c_x \\ f_y \frac{Y}{Z} + c_y \end{bmatrix} \tag{2.5}$$

where $[c_x c_y]^t$ is the position of the principal point represented in the image frame. Thus, when $x = y = 0$, the image frame location will be $[u, v]^t = [c_x, y_x]^T$.

This model can be described in matrix form.

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.6}$$

$$\lambda \tilde{p} = K\tilde{P}^c \tag{2.7}$$

Where $\tilde{p}$ is the homogeneous image coordinates corresponding to the projection of world point $\tilde{P}$. K is the projection matrix, and along with the lens distortion coefficients composes the intrinsic camera parameters. Homogeneous coordinates are used to ensure direct compatibility with homogeneous transformations. However, the projection model could also have been defined using cartesian coordinates.

Lens distortion

When a true pinhole camera is used, very limited amounts of light are let through the pinhole and onto the captive surface. This yields a need for either extremely high sensitivity or long exposure time, resulting in high noise or high motion blur. To counter this effect, a lens is used along with a larger aperture. The lens attempts to focus a larger area of light from an object onto the pixel it would have hit through a pinhole. However, lenses are not perfect, and some geometric distortions are introduced. This renders eq. (2.6) inaccurate. In addition, straight lines in the real world might not be projected as straight lines in the image. As this is an assumption made in many image post-processing algorithms, the geometric distortion can have severe consequences.

The lens distortion can be modeled, and an inverse transformation can be applied to the image to restore the pinhole model properties. The geometric distortion is composed of radial and tangential distortion. Radial distortion applies a translation along the radial direction from the principal point, and tangential distortion is a translation along a 90° angle to the radii. The radial distortion can be approximated by a polynomial (Corke, 2017, p. 261)

$$\delta r = k_1 r^3 + k_1 r^5 + k_1 r^7 + \ldots \tag{2.8}$$

The total effect of geometric distortion is a displacement of the pixel location such that the new distorted pixel is $u^d = u + \delta_u$, $v^d = v + \delta_v$. This combined displacement of radial and tangential distortion is approximated by (Corke, 2017, p. 262)

$$\begin{bmatrix}\delta_u \\ \delta_v\end{bmatrix} = \underbrace{\begin{bmatrix}u(k_1 r^2 + k_2 r^4 + k_3 r^6 + ...) \\ v(k_1 r^2 + k_2 r^4 + k_3 r^6 + ...)\end{bmatrix}}_{\text{radial}} + \underbrace{\begin{bmatrix}2p_1 uv + p_2(r^2 + 2u^2) \\ p_1(r^2 + 2v^2) + 2p_2 uv\end{bmatrix}}_{\text{tangential}} \tag{2.9}$$

When the distortion is modeled with sufficient accuracy, the undistorted image I can be restored by remapping the pixels in the distorted image I_d

$$I(\begin{bmatrix}u \\ v\end{bmatrix}) = I_d\left(\begin{bmatrix}u = u - \delta_u(u,v) \\ v - \delta_v(u,v))\end{bmatrix}\right) \tag{2.10}$$

The pixel displacements will typically not be integer, which means an exact remapping is not possible. The easiest solution is to choose the closest pixel, however a better approach is to use bilinear interpolation.

Camera calibration

In order to use the model, the model parameters has to be determined. They vary between cameras, as well as with different settings on the same camera. Therefore, some procedure should be made to estimate the parameters for the relevant camera with the relevant settings.

Camera calibration is performed by capturing images of some known object, with a corresponding 3D-model of keypoints. The camera model is found by minimizing the projection error of the object onto the image. When calibrating the intrisic parameters of a camera, the extrinsics has to be either known or estimated simultaneously (Corke, 2017, p. 262). Both MATLAB (Computer Vision System toolbox, r2018b) and OpenCV contain functionality for pinhole camera calibration without prior knowledge of extrinsic parameters.

2.3.3 Pose estimation

Pose estimation is the task of determining the extrinsics between a camera and an object captured in the image. Depending on the application, one could be interested in obtaining the full transformation from the camera frame to the object frame, or just relative position, euclidean distance or orientation. There are mainly two approaches to perform the position part of pose estimation. Either using a known 3D-model of keypoints, or using two cameras with known relative position. Note that for the orientation of the pose to be defined, an object model must be present.

Pose estimation using known object model

If the object model is known with at least three keypoints[1] identifiable in the camera image, pose estimation can be performed with a single image. When using n points, this is called the PnP (Perspective n Points) problem. By constraining each keypoint to lie along the line where 3D points are projected to the identified keypoint pixel, the object pose can be extracted. If more than three points are used, the robustness and accuracy is increased, but the system will be over-determined. To fit such a system, one could use minimize the square error between the observed pixel keypoints and their estimated projection using estimated pose.

In order to provide a set of keypoints easy to detect and model, a square printable marker could be used. The marker should contain encoded information about orientation and identification. The open source OpenCV library (Bradski, 2000) contains a module (ArUco), which provides such markers.

Pose estimation using stereo camera

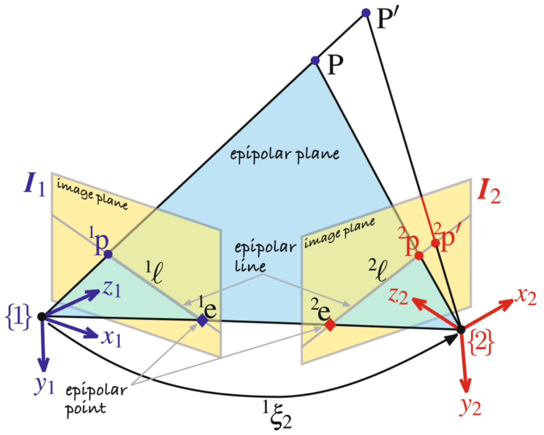


Figure 2.3: Stereo camera projection (Corke, 2017, p. 402)

Using a stereo camera, the position of a keypoint relative to the camera frames can be estimated without a known object model. As the two cameras have their origin at different points, the two lines from the object to the frame origins will intersect at the object. In addition, points along the ray corresponding to the object in one camera will be projected as a line in the other image. This line is called the epipolar line, and can be used to reduce the search area for the relevant keypoint. It is important to note that due to discretization and model errors, the lines are likely to not intersect at all. A good choice of position estimate will then be the center of the two closest points along the lines (Corke, 2017, p. 404).

[1]A keypoint is a feature that is easy to uniquely identify by a computer

The distance between the camera origins is called the baseline $[b]$. For simplicity of calculation, the camera frames are assumed to be equal in orientation and offset along the x-axis. Then the homogeneous transformation between the camera frame 1 $\{1\}$ and camera frame 2 $\{2\}$ will be:

$$T_2^1 = \begin{bmatrix} 1 & 0 & 0 & b \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

Let the homogeneous coordinates of the object in camera i be $\tilde{p}_i$, with projection matrix K_i. The rays from each camera origin, o_1 and o_2, to the object P, are parameterized as $o_i + \alpha_i d_i$. d_i is some vector along $o_i \rightarrow P$.

Let P_i be an output, for any λ, of the inverse camera projection. Then P_i will lie on the $o_i \rightarrow P$ line.

$$\tilde{P}_i^i = K^{-1} \lambda \tilde{p}_i \tag{2.12}$$

The vectors d_i are then constructed

$$d_i = P_i - o_i \tag{2.13}$$

Finally, the object point P is found solving for the intersection point, or the two closest points if errors are present.

$$P = o_1 + \alpha_1 d_1 = o_2 + \alpha_2 d_2 \tag{2.14}$$

Expressed in a single frame, camera 1, this becomes:

$$\tilde{P}^1 = \alpha_1 \tilde{d}_1^1 = T_2^1 (\tilde{o}_2^2 + \alpha_2 \tilde{d}_2^2) \tag{2.15}$$

2.4 Kalman Filter

The Kalman Filter section is reused from (Joberg, 2018)

The Kalman filter uses measurements and a system model to produce estimates of the system states. Under the certain assumptions, the Kalman filter is the optimal observer w.r.t minimum variance. The assumptions are:

- The system is linear
- Process and measurement errors are zero-mean Gaussian and uncorrelated[2]

[2]KF assumes no correlation between process and measurement noise, however, internal coupling of noise is accounted for.

- Process and measurement error variances are known
- The system is observable
- The error dynamics are controllable

that the with known variances, There exist several versions, fitting applications with either linear or nonlinear process model, as well as continuous time or discrete time.

2.4.1 Extended Kalman Filter

The extended Kalman filter (EKF) is used to estimate a nonlinear process, and uses a linearization about the estimated state. The EKF is only locally stable, and can diverge from the correct solution. Given a process

$$\dot{x} = f(x,t,u) + w(t) \tag{2.16}$$

$$y = h(x,u) + v(t) \tag{2.17}$$

the EKF produces a first order linearization by:

$$F = \frac{\partial f(x,t,u)}{\partial x}\Big|_{x=\hat{x}} \tag{2.18a}$$

$$H = \frac{\partial h(x,u)}{\partial x}\Big|_{x=\hat{x}} \tag{2.18b}$$

And performs the regular Kalman filter prediction and update using F as state transition matrix and H as measurement matrix.

2.4.2 eXogenous Kalman Filter

The eXogenous Kalman filter (XKF) is similar to the extended Kalman filter, and is also intended for nonlinear processes. However, instead of applying linearization about the estimated state, it uses a separate nonlinear observer to provide a state to linearize about. The nonlinear observer should be globally convergent, but is expected to have sub optimal performance. This is beneficial, as the XKF inherits the global convergence of the nonlinear observer, while remaining close to optimal for the local solution (Johansen and Fossen, 2017).

2.5 Guidance path control

Guidance control is methods used to construct a desired velocity, such that the vehicle approaches it's target. Line of sight guidance is suitable when the vehicle should approach it's target along the path defined by two waypoints. Pure pursuit is used when the vehicle should align its velocity vector directly at the target. Constant bearing guidance is similar to the pure pursuit method, but uses an estimate of the target velocity to provide a feed forward component. (Fossen, 2011).

In this project, only the Constant bearing guidance is considered.

2.5.1 Constant Bearing Guidance

The constant bearing guidance method consists of two velocity components. v_a is the desired approach velocity, and is the rate at which the distance to the target should be reduced. v_t is the velocity of the target. The desired velocity of the vehicle v_d is then given by eq. (2.19)

$$v_d = v_a + v_t \tag{2.19}$$

v_t is either acquired by a communication link between the vehicle and the target, or by an estimator.

v_a is set as:

$$v_a = -\kappa \frac{\tilde{p}}{\|\tilde{p}\|} \tag{2.20}$$

where $\tilde{p}$ is the position error defined by $p - p_t$. p is the vehicle position and p_t is the target position.

κ is set by eq. (2.21)

$$\kappa = U_{a,max} \frac{\|\tilde{p}\|}{\sqrt{(\tilde{p}^T \tilde{p} + \Delta_{\tilde{p}}^2}} \tag{2.21}$$

$U_{a,max}$ is a configuration parameter determining the approach speed at large distances. $\Delta_{\tilde{p}}$ is a configuration parameter determining the transient behaviour of the speed as the position error gets small.

Chapter 3

Design and concepts

3.1 Overview and purpose

The designs and implementations developed as part of this thesis aims to produce a control system in DUNE, capable of landing a multirotor UAV on a floating target perturbed by the sea state. The system is to be run on a single board computer, and interface a radar, a camera, and a Pixhawk running Ardupilot. The system should be structured in a modular manner, and efforts have been made to design a system architecture where iterations to modules can be made independently.

Firstly, a concept for calculating the position of a detected object is presented. Thereafter a state machine governing the stages of the mission is proposed, A system architecture realizing the state machine is described in fig. 3.3, where the modules are presented. The modules are further explained in section 4.2.

3.2 Camera vision

The Camera vision section is reused from (Joberg, 2018)

In this section, the notation described in section 2.3.1 is used. Here, a method to estimate the position of a detected keypoint floating on the sea surface is proposed. This method relies on knowing the elevation of the camera above mean sea level, and assumes a pinhole camera model is used.

3.2.1 Position estimation using surface constraint

In the absence of waves, a local subset of the sea surface can be approximated as euclidean. As the object of interest in this project is known to float on the sea, a planar surface constraint can be added to the problem. There will be some errors to this assumption, as waves will induce motion above and below the mean sea level. However, it will add useful information, especially at larger distances where other methods of position estimation will suffer. In addition, the horizontal position error induced by waves will be strongly reduced when the camera is positioned at steep angles above the object.

Note that the intersection between a plane and a line will yield one unique solution as long as the plane and line are not parallel. If they are parallel there will either be infinite or no solutions. While the camera is at any significant elevation above sea level, the line from the camera to the object will not be parallel with the sea surface, and a single solution will be produced.

Assuming the object has been detected in an undistorted image, the center of the object can be chosen for position estimation. Using the camera model, a vector can be constructed along the direction from the camera origin to the object. Let the pixel corresponding to the detected object center be noted $\tilde{p_O} = [u_O, v_O, 1]^T$. From eq. (2.7), a line $\frac{1}{\lambda}\tilde{d}_{CO}^{C}$ is constructed, whereupon all points are projected onto $\tilde{p_0}$. The line is realized by varying λ, and $[\tilde{D}_{CO}]$ is some vector pointing along the camera→object vector.

$$\frac{1}{\lambda}\tilde{D}_{CO}^{c} = K^{-1}\tilde{p_o} \tag{3.1}$$

For simplicity of calculation, a new frame is constructed with its origin coincident with the camera frame and the z-axis pointing down towards the center of the earth. Lets call the new frame the camera-1 frame $\{C1\}$. The x_{C1} and y_{C1} axes can be defined in any way satisfying a right hand coordinate system, for example along north and east. The homogeneous transformation from $\{C\}$ to $\{C1\}$ will be denoted T_C^{C1}, then

$$\tilde{P}^{C1} = T_C^{C1}\tilde{P}^{C} \tag{3.2}$$

Let the elevation of the camera above mean sea level be h_C. As the $\{C1\}$ frame points directly down, points at the (planar) sea surface expressed in $\{C1\}$ will have their z-axis coordinate equal to h_C. Let the z-axis value of D_{CO}^{C1} be z_D By choosing $\frac{1}{\lambda} = \frac{h_C}{z_D}$, the intersection point, and the object position estimate P_O is found.

$$\tilde{P}_O^{C1} = \frac{1}{\lambda}\tilde{d}_{CO}^{C1} = \frac{h_C}{z_D}\tilde{d}_{CO}^{C1} \tag{3.3}$$

$$\tilde{P}_O^{C} = \frac{h_C}{z_D}T_{C1}^{C}\tilde{d}_{CO}^{C1} \tag{3.4}$$

Horizontal error

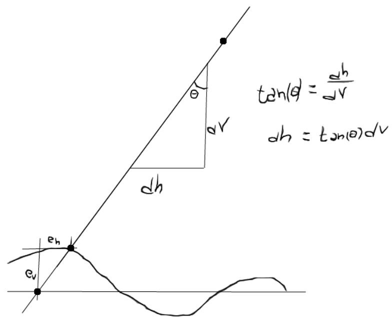


Figure 3.1: Position estimate error induced by wave

The wave motion will induce errors to the position estimate based on the steepness of the camera→object vector $\overline{CO}$. Let θ express the angle between a vertical line and $\overline{CO}$. As shown in fig. 3.1, a unit step in the vertical part of the line will correspond to $\tan(\theta)$ in the horizontal part. If the wave induces an error in height above sea level by e_v, the corresponding error in horizontal position will be $e_h = e_v \tan(\theta)$. Therefore, when the camera is located at a steep angle above the object, θ will be small and the horizontal position error will be small. However, as θ approaches $\pm 90°$, the error will approach $\pm$ infinity (assuming $e_v \neq 0$).

3.3 UAV State machine

3.3.1 UAV state machine

A state machine should be utilized in the UAV control system to switch between control methods relevant to the different phases of the mission. A proposed set of states and transitions are illustrated in fig. 3.2. The diagram is based on the state machine proposed in (Joberg, 2018) It is important to note that an abort state should be included, with transitions from every other state. To avoid clutter, it is excluded from the state diagram.

Initialize navigation

The state machine starts by initializing the navigation system. The navigation system on ArduPilot uses an extended kalman filter for navigation, and uses IMU and GNSS measurements. The IMU should be calibrated, and the GNSS estimates need time to converge after being started. At the ”mission start” event, the machine should move into the takeoff mode. Upon this transition, a MAVLink message should be sent to ArduPilot to enter the ”GUIDED” mode. The DUNE task handling MAVLink messages currently contain no functionality for sending this message. The relevant mes-

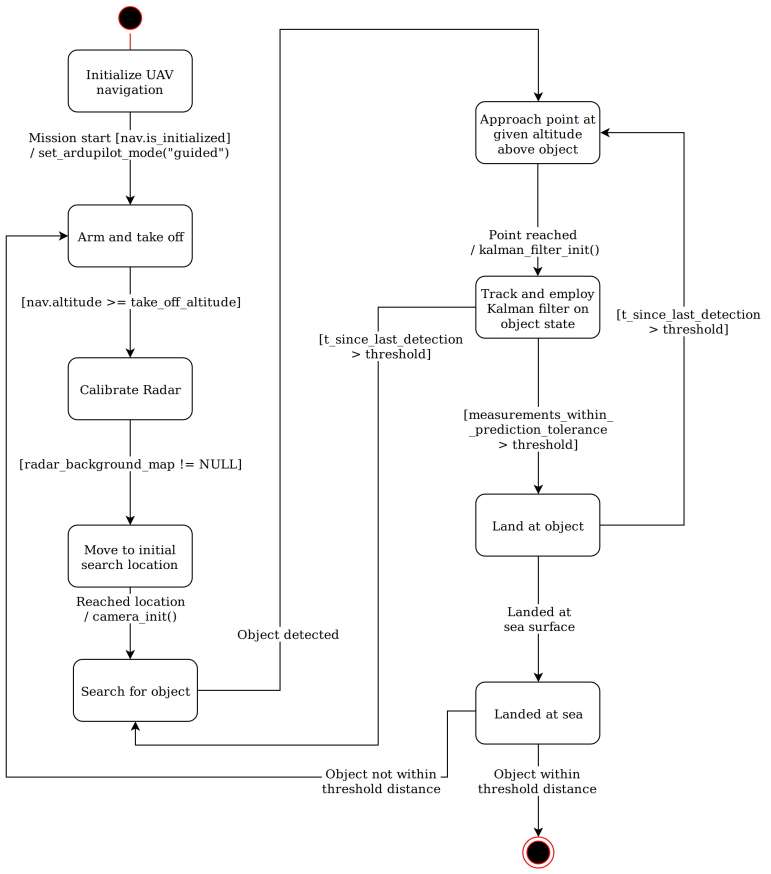

Initialize UAV navigation
Mission start [nav.is_initialized] / set_ardupilot_mode("guided")
Arm and take off
[nav.altitude >= take_off_altitude]
Calibrate Radar
[radar_background_map != NULL]
Move to initial search location
Reached location / camera_init()
Search for object
Object detected
Approach point at given altitude above object
Point reached / kalman_filter_init()
Track and employ Kalman filter on object state
[t_since_last_detection > threshold]
[t_since_last_detection > threshold]
[measurements_within_ _prediction_tolerance > threshold]
Land at object
Landed at sea surface
Landed at sea
Object not within threshold distance
Object within threshold distance

Figure 3.2: State machine of UAV control system

sage is MAV_CMD_DO_SET_MODE and is documented in: `https://mavlink.io/en/messages/common.html#MAV_CMD_DO_SET_MODE`

Arm and takeoff

When the takeoff state is entered, the UAV should be armed and a takeoff should be performed to a suitable altitude. When the desired altitude is reached, the state machine should move to the calibrate radar state.

Calibrate radar

The radar currently used, X4M03, has the transmitting antenna and receiving antenna placed closely together on the same PCB. The poor isolation yields a significant direct path component in the radar signal. In addition, the radar signal might be reflected by components on the multirotor. This is considered as a constant noise component when the radar output is used for altitude estimation. To remove the noise, the radar output is mapped at a high altitude, while the ground is outside of the radar range.

Upon entry to the “Calibrate radar” state, the multirotor should increase it’s altitude, if necessary, such that the sea surface and other objects are outside the maximum radar range. When this altitude is reached, the multirotor should hold it’s position while the radar acquires information about the background noise. When the radar has completed it’s calibration, the state machine should change to the initial search location state.

Move to initial search location

If prior information about the location of the MUG is available, this should be utilized to provide a starting location for the search. A waypoint should be issued to move the UAV to the relevant search location. Upon completion of the waypoint, the state should be changed to search for the object.

Search for object

No work has been done to develop an algorithm for a search pattern in this project. However, when the object has been detected, the state machine should change to approach a point above the MUG

Approach a point above the MUG

As the MUG average velocity will be roughly the same as the sea current, the MUG is assumed to move slowly. A waypoint can then be used to move the UAV to a point above the MUG. If the MUG has moved significantly during the maneuver, a new waypoint can be issued to a position above the new MUG position. When the UAV’s horizontal position is within a satisfactory radius of the MUG, the state should change to track the MUG and employ a Kalman filter on its sate.

Track MUG and employ a Kalman filter

Upon entry to this state, the Kalman filter estimating the MUG state should be initialized. If the UAV has communication with another vehicle using a Kalman filter containing parameters from the sea state, these parameters might be used in the initial state of the MUG Kalman filter. Such a vehicle may be an unmanned surface vehicle (USV), which may be the vessel the UAV was launched from. The initial covariance parameters should be increased when copied from one Kalman filter to another. The covariance increment should be decided based on the distance between the MUG and the vehicle the parameters were based on.

As the UAV continues to observes the MUG, the covariances in the Kalman filter should decrease until convergence is reached. While observing the MUG, the UAV should track either the MUG position or its central wave position. When satisfactory covariances are obtained, the state machine should change to the land state.

If the camera vision looses track of the MUG for a significant time, the state machine should change to the search for MUG state.

Land at the MUG

An algorithm for the landing maneuver has yet to be implemented. However, constant bearing guidance as described in (Fossen, 2011) is a good candidate.

When a landing has been detected, the UAV should be disarmed and the state machine should be changed to the landed state.

Landed at sea

A tool to determine whether the MUG has been landed on with sufficient accuracy has yet to be designed. However, if sufficient accuracy is achieved, the landing mission can be concluded as successful, and the state machine can terminate. If sufficient accuracy is not obtained, the state machine should move to the take off state.

3.4 System architecture

The purpose of the system architecture is to divide the control problem into modules performing a specific manageable functionality, with a concise interface between the modules. The architecture should utilize the exiting Ardupilot task in DUNE, and provide it's own interface to the camera and the radar.

A system architecture was proposed in (Joberg, 2018), and has been expanded upon in this thesis. The outline is shown in fig. 3.3, and utilizes the existing DUNE tasks UAV.Navigation and UAV.Ardupilot, as well as defining five new tasks:

- Supervisor
- Radar altitude
- Camera object detection
- Target model filter
- Path Control

A logical addition to the architecture would be a module handling the search state of the state machine. However, no search functionality is implemented in this thesis, and it is omitted from the architecture diagram to avoid clutter.

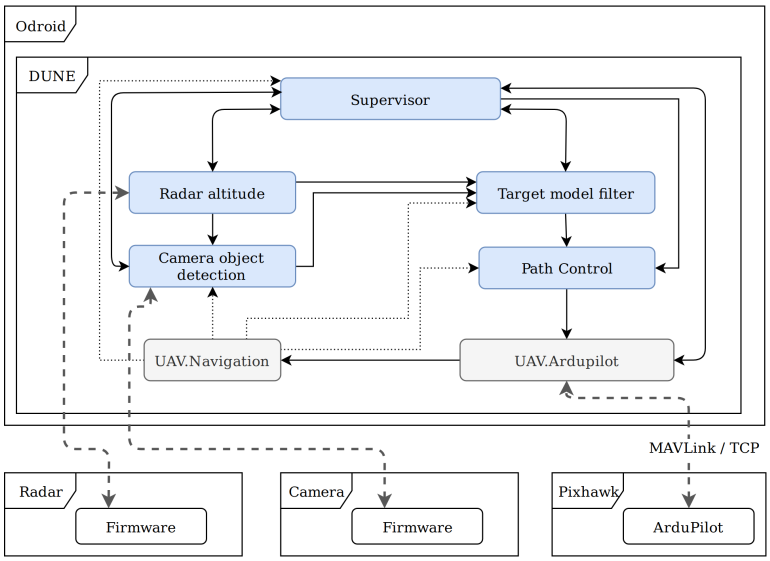


Figure 3.3: Proposed architecture

3.4.1 State machine sub-architectures

For each state of the state machine presented in fig. 3.2, a subset of the system architecture will be utilized. These subsets are presented below.

Throughout the state machine, the UAV.Ardupilot task sends the autopilot status to the Supervisor. This is used to determine if the Supervisor should move to the abort state. The abort state is not illustrated in fig. 3.2 to avoid clutter, as every state can transition to abort.

Arm and take off

The UAV.Ardupilot task maintains a state machine determining the type of control messages it accepts. In addition, the Ardupilot autopilot software must be armed before it will accept any commands that will cause power being applied to the multirotor motors. Upon entry to this state, the Supervisor is responsible for commanding the activation of the velocity control loop in UAV.Ardupilot, as well as arming the autopilot. Additionally, the Path Control module is actuated upon state entry.

Navigation data from UAV.Ardupilot is propagated through UAV.Navigation to the Supervisor and the Path Control modules. Currently, UAV.Navigation adds no new information to the navigation data from the autopilot. However, it can be used to employ RTK, as well as to fuse the radar altitude estimates with the autopilot sensors.

The supervisor dispatches a waypoint to the Path Control directly above it’s current position, and the Path Control assigns the proper velocity set points to UAV.Ardupilot.

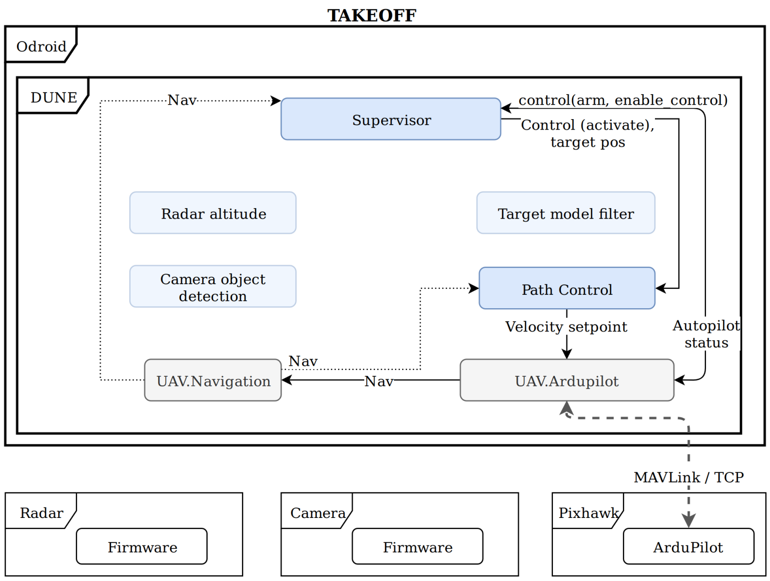


Figure 3.4: Active architecture during state: takeoff

Calibrate radar

When entering the calibrate radar state, the Supervisor should issue a new waypoint to the Path Control module. The waypoint should move the multirotor to a location without any objects in the radar detection range. In the case of an open sea environment, this could be achieved by increasing it's altitude by the range of the radar.

When reaching the waypoint, the Radar altitude module is commanded to calibrate. If the radar module detects abnormalities in the calibration, an error status should be returned to the Supervisor. Otherwise, a successful status is returned, and the radar should begin publishing it's altitude estimates.

If the supervisor receives an error status from the radar, it should move the multirotor to a new waypoint and repeat the process.

Move to initial search location / search

In the current state of the project, the move to search location state and the search state have been combined. When an explicit search state and module is included, it would make sense to delay the activation of the camera module until entering the search state.

Upon entering the "move to search location" state, the Supervisor commands the camera module to start, as well as whether or not auto-exposure calibration should be used. The Supervisor also dispatches the desired initial search location to the Path Control module.

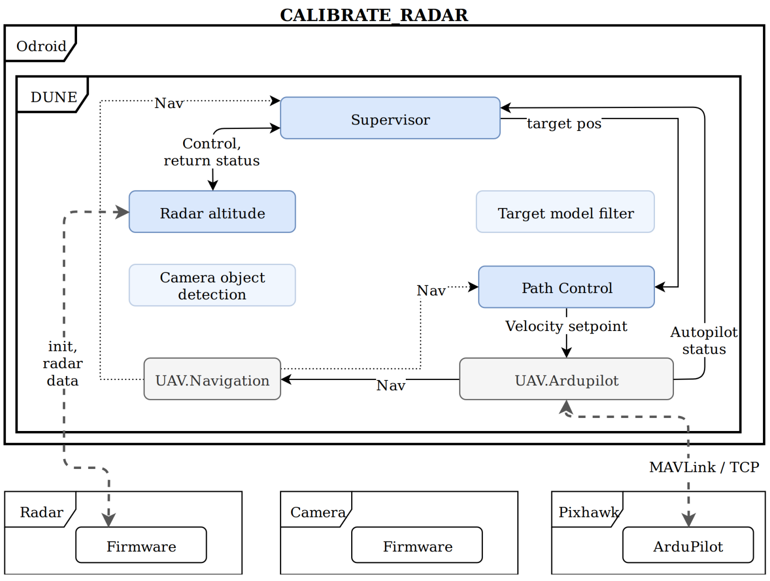


Figure 3.5: Active architecture during state: calibrate radar

When the camera modules acquires a successful detection of the object, an object_found message is sent to the supervisor. The supervisor then transitions to the approach state.

Approach point above object

This state uses a similar architecture utilization as fig. 3.6. The model filter is not initialized yet is due the expectancy of significant target position errors. As presented in section 3.2.1, when the vertical slope between the camera and the target is shallow, errors in the altitude will yield a large position error. If there is a need for filtering while approaching the target, the filter could be initialized here with a significant increase in the covariances expected from the measurements.

Track and employ model filter

Land at object

The landing state uses a similar architecture utilization as fig. 3.7. The difference is that the Supervisor doesn't send an initialize command to the Target model filter. The desired altitude sent from Supervisor to the Target model filter should be zero, while the model filter is used to track the horizontal position of the target. When a landing is detected, the Supervisor moves to the landed state.

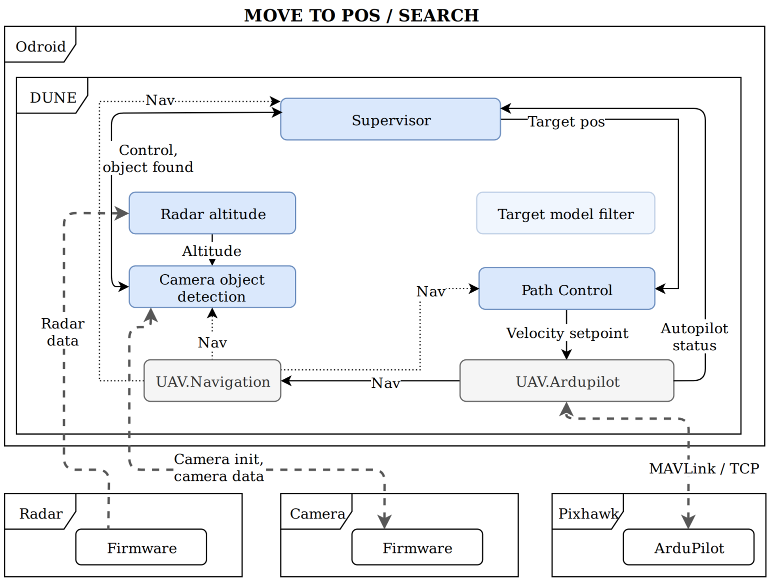


Figure 3.6: Active architecture during state: Move to initial search location

Landed

Upon entering the landed state, the Supervisor should terminate terminate the active modules. A disarm request and a disable velocity control command should be sent to UAV.Ardupilot. The Path Control and radar modules should be terminated. Unless the camera is used to determine if the target is within the threshold distance, the camera module should be terminated as well.

While the mechanism for picking up the target is not yet implemented, it should be able to tell if the target has been properly engaged. If this pickup mechanism returns a success status, the problem of landing on the target can be concluded. If the mechanism returns an error, the Supervisor should move to the Arm and take off state.

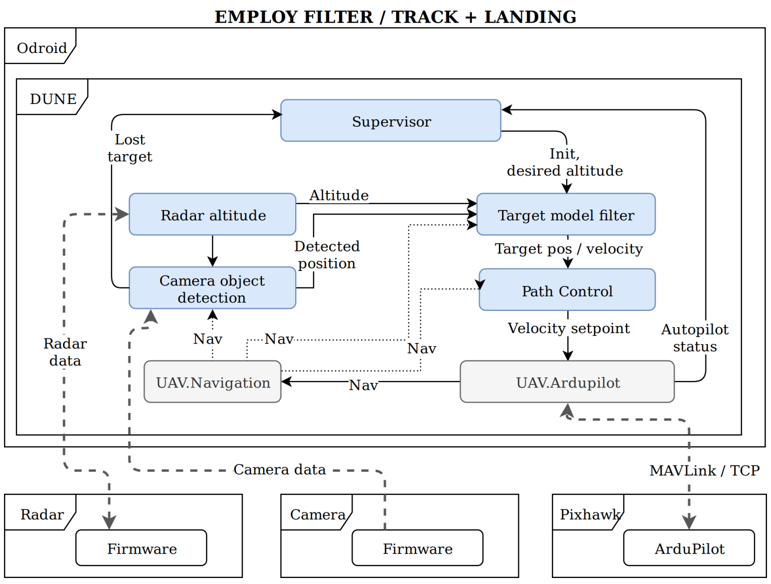


Figure 3.7: Active architecture during state: Track target

Chapter 4

Dune implementation

A slightly modified version of the architecture in fig. 3.3, and state machine in fig. 3.2, is implemented in DUNE. Th implementation is targeted at a test where a pilot is exchanging control of the vehicle with DUNE. In particular, the takeoff and landing will be done by the pilot. This exchange of control causes the need for the changes to the state machine and architecture.

Another modification is that the camera and radar module is being run at all times. This is done to acquire as much log data as possible from the test.

4.1 IMC messages used

4.1.1 Reused IMC messages

It is the recommendation of LSTS to reuse IMC messages where possible.

IMC::EstimatedState

The estimated state message is used by the UAV.Navigation module to publish the estimated spatial states of the vehicle. The estimates includes:

- Vehicle position in ECEF frame (WGS84).
- Vehicle position in a NED frame.
- Vehicle velocity in NED frame.
- Vehicle orientation relative to NED frame.

- Vehicle angular velocity in NED frame.
- Altitude above the Ardupilot HOME location.

IMC::AutopilotMode

The autopilot message is used by the UAV.Ardupilot task to publish information and receive requests related to the Ardupilot state and status. It contains an autonomy field and a mode field. The autonomy field is used to determine determine whether the human operator or the DUNE system is in control of the vehicle. The mode field is a string used for various purposes. In this project the mode field is used for arming and disarming requests.

IMC::VehicleCommand

This message is used for controlling the status of the modules. The command info field is a string used for targeting the module the message is intended for. In future iterations, it is recommended that the targeting of modules is done with an enumerated value.

The command field is a `uint8` type, and is set to an enumerated value assuming EXEC, STOP, START_CALIBRATION or STOP_CALIBRATION

IMC::Distance

The IMC::Distance message is used by the radar to broadcast it's altitude estimates. The 32 bit float field "value" is set to the estimated altitude.

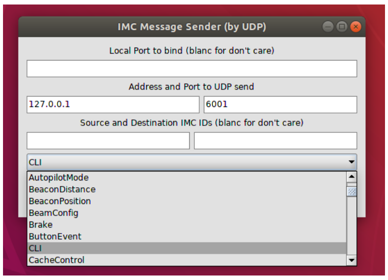
IMC::DesiredControl

The IMC message used for velocity commands to UAV.Ardupilot. The 32 bit float fields x, y, and z is set to the desired velocity in NED frame. The 8 bit uint field "flags" is used as a bit mask for valid velocity components. Invalid components will be set to zero.

4.1.2 Newly defined IMC messages

IMC::CLI

The IMC::CLI message is defined to provide a command line interface (CLI) to be used during experiments. There is a tool provided by LSTS for remotely dispatching an IMC message to DUNE, but it is inconvenient to navigate between different IMC messages. There are several hundred IMC messages, and a relatively tiny scrolling widget is used for selection, fig. 4.1. The message contains a text field, as well as three 32 bit float fields. Currently, it is used for controlling the sensor modules. This is done to enable testing of those modules without running the state machine. In addition, the Supervisor


IMC Message Sender (by UDP)

Local Port to bind (blanc for don't care)

Address and Port to UDP send: 127.0.0.1 6001

Source and Destination IMC IDs (blanc for don't care)

CLI

Figure 4.1: Menu used to select IMC message for manual sending

module requires a IMC::CLI message in order to move to the state machine landing state. Currently, the land message can only be sent manually to ensure that the multirotor will not land unless explicitly told to.

IMC::CameraTracking

The IMC::CameraTracking message is created to transport the world point detected by the camera module to the model filter module. It contains three 32 bit float fields, which should be set to the NED frame coordinates of the detected point.

IMC::ConstantBearingTarget

This message is used to transmit the target state the Path Control module should track. It contains six 32 bit float fields. Three for NED position, and three for velocity in NED.

IMC::DesiredAltitude

This message is constructed to communicate what elevation above the sea surface the vehicle should maintain. While the purpose could be fulfilled by reusing the `IMC::Distance` message, this message is implemented to maintain a clear intention. While it is possible to check which module a message originated from, and thus derive intention, it is prone to programmer errors. As the desired elevation is a safety critical message, clear intention is regarded as more important than restricting the scope of new messages. The 32 bit float field "value" is used to store the desired elevation.

4.2 Module details

4.2.1 Supervisor

The supervisor module has been named Supervisors.H2O_Pickup in DUNE. It handles the state machine illustrated in fig. 3.2 and activates or deactivates the proper modules upon state transitions. It also receives information from the modules, relevant to the transition criteria of the state machine. Simple waypoints are also dispatched from the Supervisor module.

Handling exchange of control

As testing will be conducted with a pilot exchanging control of the multirotor, a `MANUAL` control state is included.

If in `MANUAL`: Upon receiving an `IMC::AutopilotMode` message with the autonomy field set to `AL_AUTO`, the state machine is put in the `TAKEOFF` state.

If not in `MANUAL`: Upon receiving an `IMC::AutopilotMode` message with autonomy field `AL_MANUAL`, the state machine is moved to `MANUAL`. Modules that may send commands to UAV.Ardupilot is deactivated upon the transition to `MANUAL`. In addition, enabled control loops in UAV.Ardupilot is disabled.

Takeoff

Upon entering the `TAKEOFF` state, the IMC message `IMC::ControlLoops` is used to request a change in the set of active control loops utilized by UAV.Ardupilot. The autopilot must also be armed. By dispatching a `IMC::AutopilotMode` message with the `mode` field set to ”ARM”, UAV.Ardupilot will send a MAVLink request to arm the vehicle. The Path Control module is also activated, using the `IMC::VehicleCommand` message.

In addition, a target is generated at a given altitude above the current position. The altitude increment is set by a configuration variable. The target is dispatched to the Path Control module using the message `IMC::ConstantBearingTarget`. Upon reaching a given distance to the target, the state machine is set to `CALIBRATE_RADAR`

Calibrate radar

When entering `CALIBRATE_RADAR`, the a target at a vertical increment is issued to the Path Control module. In addition, the vehicle position at state entry is saved.

When reaching the the target position, a `IMC::CLI` message is used to send a calibration command to the radar module. When the calibration is done, a `IMC::CLI` message will be returned to the supervisor. A successful calibration will cause the Supervisor to move the state machine to `SEARCH`.

Search

As mentioned in section 3.3.1, no search algorithm is implemented. Instead, the vehicle position that was saved at entry to the `CALIBRATE_RADAR` state is used as the initial search location. This position is dispatched as a target to the Path Control module. Upon reaching the target, the position is held until an object is detected by the camera module. The state is then changed to `TRACK_TARGET`.

Track target

In this state, the Target model filter module is activated. While this module is activated, the Supervisor should not send any targets to the Path Control module. While not implemented yet, a suggestion for improvement is to implement a token dictating which module has the right to dispatch targets to the Path Control module.

The altitude which should be maintained while tracking the horizontal position of the target is determined by a configuration variable. The message `IMC::DesiredAltitude` is used to send the desired altitude to the Target model filter module.

For safety reasons, the transition to the landing state is done using a message sent by a human operator.

Landing

Upon entry to this state, the Supervisor dispatches `IMC::DesiredAltitude` with the value field set to zero. The landing is then performed by keeping the horizontal tracking from the previous state while descending.

No detection of a successful landing has been implemented yet. It is noted that the task Monitors.Medium dispatches the message `IMC::VehicleMedium`. It contains information about whether the vehicle is on the ground or in the air. However, it is a simple altitude check based on the altitude in the `IMC::EstimatedState` message. A custom function utilizing the radar values should be better suited to check whether the vehicle is landed.

4.2.2 Radar altitude

The radar altitude module is named Sensors.X4M03 in DUNE. It handles the interface to the X4M03 radar, and publishes it's altitude estimates. A shared-object driver provided by the radar manufacturer, Novelda, is used to provide an abstraction to the radar firmware. It receives receives control commands from the Supervisor, and returns the execution status of the command.

Acquiring radar interface object

The interface to the radar is handled by an `XeThru::XEP` object. It contains methods for configuring the radar, interacting with an output queue, and termination of the radar. A pointer to such an object, named `mp_xep`, is set as a member of the task struct.

The `XeThru::XEP` class needs to be constructed using an `XeThru::ModuleConnector` object. The constructor of the Module Connector takes the path of the device file used by the radar as an argument. This file will typically be `"/dev/ttyACM0"` or `"dev/ttyACM1"`. The purpose of this class is to establish contact with the radar firmware. By calling the `XeThru::ModuleConnector::get_xep()` method, an `XeThru::XEP` object is constructed.

Note that the `XeThru::XEP` object appears to access memory allocated by the Module Connector object. Thus, when the Module Connector is deconstructed, several XEP methods will cause a segmentation fault. This issue is handled by using a static variable to store the Module Connector object.

Start up

On start up, the radar module initializes the firmware with a set of configurable parameters. The parameters, and their default values, are:

• tx_power	= 2
• dac_min	= 949
• dac_max	= 1100
• iteration	= 16
• pps	= 300
• fps	= 10
• offset	= 0.18
• frame_start	= 0.4
• frame_end	= 5.0
• Downconversion	= 1

The parameters are documented in Novelda (2018).

Upon setting the radar fps to a value greater than zero, the radar is started. This is currently done along with the initialization, in order to record and monitor data during the entire flight duration.

Accessing radar data

As the radar captures its frames, the XEP object stores them in a first in first out (FIFO) queue. The oldest frame in the queue is extracted (popped), using the `read_message_data_float(&flo` method. To find the number of frames in the queue, the `peek_message_data_float()` method is used. By calling the read method until the peek method returns 0, the newest frame is acquired.

Getting the amplitude waveform

The amplitude of the correlation between the transmitted and received signal, at a given lag, describes the power of the reflected signal at a given range. As the Downconversion setting is set to 1, the data extracted from the queue is a complex baseband. The complex data is stored in a float array, where the first half contains the real part and the last half contains the complex part. Assuming there are `<n>` complex numbers, the amplitude of the frame is computed by
`amplitude[i] = sqrt(data[i]^ 2 + data[n+i]^2)`
This describes the power of the reflected signal for each bin. (Novelda, 2018).

Bin to range

The radar transceiver operates at a sampling frequency of $f_s = 23.328\text{GHz}$. The time τ from the start of the frame to the bin k, is given by eq. (4.1):

$$\tau = \frac{k}{f_s} \tag{4.1}$$

The delay observed is caused by the signal traversing a given range twice at the speed of light. The range R is then found by eq. (4.2):

$$R = \frac{c * \tau}{2} \tag{4.2}$$

Where $c \approx 3 * 10^8$ is the speed of light.

The first bin is sampled after a specified delay. The range of this delay is acquired using the XEP method `x4driver_get_frame_area`. In addition, the downconversion option causes the bins to be decimated by a factor of 8. The range of a bin `<k>` is then found by:
`range[k] = k*8*c/(fs*2) + frame_area.start`
(Novelda, 2018).

The radar task in DUNE is assigned a member variable to store a vector of the bin ranges, `m_range_vec`. The vector is initialized as empty. If there is a size mismatch between the range vector and the amplitude vector, the range vector is recalculated with the same number of elements (bins) as the amplitude vector.

Calibrating the radar

Upon receiving the command line interface message `IMC::CLI`, with text field `radar calibrate`, a calibration should be performed. The calibration is done by capturing an averaged map of the background noise. The number of frames that should be used in the calibration is determined by the `val_0` field of the message. If the field is set to zero, a default number of samples are used.

After the calibration is done, A `IMC::CLI` message with text field `radar calibrate done` is dispatched. Additionally, a member variable `is_calibrated` is set to true, and altitude estimates will be published in subsequent iterations.

Altitude estimation

The background map set by the calibration routine is subtracted from the measured amplitude vector. A detection candidate is found by taking the max value of this corrected amplitude map. If the amplitude of the candidate is below a configurable threshold, the candidate is rejected.

The range of the candidate is found by looking up the index of the candidate in the range vector. If the task has the variable `m_is_tracking_altitude` set to false, the candidate is accepted, and the tracking variable set to true. If the tracking is true, the candidate is only accepted if it is within a configurable range of the last estimated altitude.

If the time since the last accepted candidate exceeds a threshold, the
`m_is_tracking_altitude` variable is set to false.

Whenever a candidate is accepted, a `IMC::Distance` message is dispatched, containing the range of the candidate.

Library integration

The library provided by Novelda for interfacing the radar consists of a shared object (.so) file, and a set of header files. The library files are placed in the folder “user/vendor/libraries/XeThru”, relative to the DUNE root folder.

The shared object file is named libModuleConnector.so, and there is one for x86 and one for arm architectures. The x86 version is placed in a sub folder named “x86”, and the arm version in an “arm” sub folder.

The library integration is configured using cMake. A cMake file named “Library.cmake” is placed in the XeThru library folder. The architecture is detected by cmake using the if condition:
`IF(${CMAKE_SYSTEM_PROCESSOR` MATCHES ”arm”)}
If an arm architecture is detected, the arm sub folder is appended to the set of DUNE library folders. Otherwise, the x86 folder is included. The appending of a folder `<dir>` to the dune library folder path is done with the command:

```
set(DUNE_VENDOR_LIBS_DIR ...
   ...  ${CMAKE_CURRENT_LIST_DIR}/<dir> ${DUNE_VENDOR_LIBS_DIR}).
```

The superfolder of "XeThru" is added to the include path using the command:

```
set(DUNE_VENDOR_INCS_DIR ${DUNE_VENDOR_INCS_DIR} ...
   ...  ${PROJECT_SOURCE_DIR}/user/vendor/libraries)
```

Prerequisites

The ModuleConnector shared object library requires a specific version of the `boost::filesystem` and `boost::system` libraries. In this project, the ModuleConnector version used was "1.6.2". The x86 compatible ModuleConnector requires boost version "1.58", while the ARM compatible version requires boost version "1.62". This is subject to change in subsequent releases of the ModuleConnector library.

4.2.3 Camera object detection

The camera object detection module is named Sensors.oCam in DUNE. It uses an open-source library provided by the manufacturer, Withrobots, to interface the camera firmware. The library has been modified slightly in order to remove buffering of images. This was necessary as the execution rate of the extraction of images was significantly slower than the frame rate of the camera. The difference in rates lead to the camera module operating on an old frame. By assigning the current body/ned rotation R_B^N to the old image, the detected target position would suffer significant errors.

oCam interface

The manufacturer of the oCam camera provide a library for interfacing the camera firmware. The library defines a class, `Withrobot::Camera`, which handles configuration, start/stop control, and extraction of image data.

A pointer to a `Withrobot::Camera` object is declared as the task member variable `mp_camera`. In the dune standard task method `onResourceInitialization()`, a new camera object constructed:

```
mp_camera = new Withrobot::Camera(m_args.vid_file.c_str());
```

Where `m_args.vid_file` is a string containing the path of the device file used by the camera. The vid_file field og m_args is configurable, and the default path is "/dev/video0".

The camera is set to the correct format with the Camera method `set_format(...)`. The camera is 1280x960 pixels, has monochrome color, and operates at 30 fps. The appropriate call to configure the camera is:

```
mp_camera->set_format(1280, 960,
   Withrobot::fourcc_to_pixformat('G','R','E','Y'), 1, 30);
```

While not necessary, the brightness and exposure is set as well. This is done with the calls:

```
mp_camera->set_control("Brightness", 32);
```

`mp_camera->set_control("Exposure (Absolute)", 48);`
The initial values of the could have been set to configurable variables. Instead, the command line interface message `IMC::CLI` is used to update the parameters. If a CLI message is received with text field "oCam brightness", the brightness is set to the value of the CLI field `val_0`. If the text field is "oCam exposure", the exposure is set to the value field.

The image is extracted into an openCV object, `cv::Mat`, using the Withrobots Camera method `get_frame(...)`. The methods takes the address of the `data` field of the `cv::Mat` as the first argument. The second argument is the size of the image, (1280x960).

After the image is extracted, it is undistortion is applied using the `cv::undistort(..)` function with radial lens distortion parameters determined durging calibration.

Auto exposure

The lighting conditions may vary between flights, or even within a single flight. To deal with different light conditions autonomously, an auto exposure method is implemented. The auto exposure tries to maintain a desired mean pixel brightness in the image. This desired pixel brightness is set as a configurable range. The default range is 100 to 140. The mean pixel value of the image is found using the openCV function `cv::mean`. If the mean value is outside the desired range, an increment to the exposure is made in the desired direction. Currently, the increment is set at a fixed, configurable value. A P controller may be implemented to more quickly adjust for larger errors in the mean pixel brightness.

Auto exposure can be turned on and off using the `IMC::CLI` message, with text field "oCam autoexposure". If the `val_0` field is 1, auto exposure is turned on. If the field is 0, auto exposure is turned off.

ArUco detection

Object detection is not a primary field of investigation for this project. In the absence of an object detection module, the ArUco module of OpenCV has been utilized for detection.

The ArUco module consist of a set of markers, and a set of methods for detecting those markers. There is also methods for performing pose estimation on the detected markers. The pose estimation feature has intentionally been avoided, in order to keep the system compatible with simple object detection methods.

A marker with a specific ID has been printed on a sheet of paper. The openCV function `cv::aruco::detectMarkers(...)` is used to detect the marker, and takes the following arguments:

- `cv::Mat image`
- `cv::Ptr<cv::aruco::Dictionary> dictionary`
- `std::vector<std::vector<cv::Point2f>> markerCorners`
- `std::vector<int> markerIds`

- `cv::Ptr<cv::aruco::DetectorParameters> parameters`
- `std::vector<std::vector<cv::Point2f>> rejectedCandidates`

The function searches the image for markers contained in the dictionary, and gathers a set of candidates. If the candidate detection likelihood is below a threshold defined by the parameters, the candidate is rejected. The candidate's corners is then added to `rejectedCandidates`. If a candidate is accepted, its ID is added to the `markerIds` vector. Additionally, the corners are added to the `markerCorners` vector.

After the `cv::aruco::detectMarkers(...)` function is called, the set of detected markers is evaluated against the ID of the printed marker. If there is found no ID corresponding to the printed marker, the detection is concluded as unsuccessful. Similarly, if multiple markers with said ID is found, an error is assumed, and the detection is unsuccessful. However, if only one correct ID is in `markerIds`, a successful detection has been acquired. The center pixel of the detection is then calculated based on the marker's corners. Given corners A, B, C, D, the center pixel c is calculated as:

$$c_x = \frac{A_x + B_x + C_x + D_x}{4} \tag{4.3a}$$

$$c_y = \frac{A_y + B_y + C_y + D_y}{4} \tag{4.3b}$$

Rotation from camera to NED

The detected position of the target is to be computed in NED frame, and utilizes the method described in section 3.2.1. By applying the inverse camera projection to the detected pixel, a vector is constructed in the camera frame. This vector then needs to be described using the NED frame axes.

A rotation matrix from the camera frame to the NED frame, R_C^B is constructed. It is based on the rotation from camera to BODY R_B^C, and the rotation from BODY to NED R_B^N.

$$R_C^N = R_B^N R_C^B \tag{4.4}$$

R_C^B is constant throughout the flight, and is set as a member variable of the task struct. It is constructed using configurable Euler angles. In the case where the camera has its z-axis aligned with the z-axis of the vehicle, the only nonzero angle is ψ. This occurs when the camera points directly down while the vehicle is level. A typical configuration will have the upwards direction in the image aligned with the x-axis of the vehicle. Then the Euler angles will be $[\phi,\ \theta,\ \psi]^T = [0°,\ 0°,\ 90°]^T$.

R_B^C is determined by the Euler angles from NED to BODY. These angles are acquired directly from the `IMC::EstimatedState` message. When such a message is consumed, the attitude is stored in a task member variable. The computation of the rotation message is delayed until a succesful detection is achieved.

The rotation matrices are calculated according to eq. (4.5)

$$R = R_z(\psi) R_y(\theta) R_z(\phi) \tag{4.5}$$

$$R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.6a}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{4.6b}$$

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \tag{4.6c}$$

The matrices are stored in `cv::Mat` objects, which implements standard matrix operations.

Position estimate

The position estimator employs the method presented in section 3.2. The final equations are repeated here.

In this implementation, the target position is first calculated relative to a NED frame fixed to the camera frame origin. The relative position is then offset to a local earth fixed NED frame. In this module description, the frame symbol $\{N\}$ refers to the NED frame fixed to the camera origin. By using coincident camera and NED frames, no vector translations are performed while calculating the relative position. As only rotations are necessary, the need for homogeneous vectors and homogeneous transformation is eliminated.

When a detection is deemed successful, the position is estimated based on the center pixel, the camera to NED rotation R_C^N, and the altitude. The altitude is received from the `IMC::Distance` message, and is stored in a task member variable. It is assumed that the radar and the camera frame origin are located near each other. If there is a significant distance between them, the difference in elevation should be accounted for. Note that even if they are in the same x-y BODY plane, roll and pitch can induce elevation differences.

The inverse camera projection is applied to the detected point, using the intrinsic camera matrix. The camera matrix is configurable, and determined by a calibration procedure.

$$\lambda P^C = K^{-1}[x(p),\, y(p),\, 1]^T \tag{4.7}$$

$$\lambda P^N = \lambda R_C^N P^C \tag{4.8}$$

λP^N lies somewhere along the line between the vehicle and the target. The direction to the target, assuming the $o_C = o_N$, is described by eq. (4.9).

$$d^N = \lambda P^N = R_C^N \left(K^{-1}[x(p),\, y(p),\, 1]^T \right) \tag{4.9}$$

As the target is at the sea surface, the z-axis of P^N should be the altitude of the camera origin h. This is achieved by scaling d^N by $\frac{h}{z(d^N)}$

$$P^N = \frac{h}{z(d^N)} d^N \tag{4.10}$$

As mentioned above, P^N is relative to a NED frame fixed in the camera origin. Letting $\{N_{EF}\}$ represent the earth fixed NED frame, the target position to be dispatched is given as $P^{N_{EF}}$ The detected position should be dispatched relative to the local NED frame used by ardupilot. This should be done by first offsetting P^N to the BODY frame origin o_B. At this stage, the relation between o_C and o_B is not known, and this first offset is neglected. Then, the relative position is added to BODY frame position expressed in the local earth fixed NED frame. This position, $P^{N_{EF}}_{BODY}$, is aqcuired from the `IMC::EstimatedState` message.

$$P^N_{EF} = P^N + P^{N_{EF}}_{BODY} \tag{4.11}$$

Library integration

The manufacturer of the oCam camera, Withrobots, provide the source code for a library used to interface the radar. The library provides an abstraction to a set of `ioctl` methods that controls and extracts data from the camera.

The source files are placed in the "user/vendor/libraries/oCam" subfolder, relative to the DUNE root path. A cMake configuration file named "Library.cmake" is added to the oCam folder. The cMake file The set of ".cpp" files in the library is stored in the variable `DUNE_OCAM_FILES`, using the command:

```
file(GLOB DUNE_OCAM_FILES user/vendor/libraries/oCam/*.cpp)
```

Thereafter, the `DUNE_OCAM_FILES` set is appended to the `DUNE_VENDOR_FILES` list. This ensures that a static library of oCam is compiled and linked with the final dune program.

The `set_source_file_properties` function is used to set the same flags when compiling oCam as the rest of dune. This is important, as when the program is linked, the libraries and object files must use the same hardware intrinsics.

In addition, OpenCV has to be integrated into the project. There already exist a cMake configuration file for this purpose, "cmake/Libraries/OpenCV.cmake". This file is activated by using the `-DOPENCV=1` flag when generating the cmake build files. In the file, `FIND_PACKAGE(OpenCV REQUIRED)` and `dune_add_lib(<lib>)` commands are used to link in the OpenCV libraries. However, some modifications are necessary to properly set the include headers. This is done with the command

```
set(DUNE_VENDOR_INCS_DIR ${DUNE_VENDOR_INCS_DIR}
   ${PROJECT_SOURCE_DIR}/user/vendor/libraries)
```

In addition, the line `dune_add_lib(opencv_aruco)` has to be included to link the ArUco module.

Prerequisites

In order to successfully compile dune with the cMake configurations described above, the system has to have the following libraries installed:

- OpenCV
- udev
- v4l2

OpenCV has to be installed using cMake, in order to properly configure the `FIND_PACKAGE(OpenCV REQUIRED)` command. Udev and v4l2 can be installed from the command line using apt:

```
sudo apt install libudev-dev v4l-utils
```

4.2.4 Target model filter

The target model filter module has been named Sensors.Target_filter in DUNE, and inherits from the `Periodic` task. In the current state of the project, it employs a constant velocity discrete Kalman filter. In future iterations, it should be updated to employ a model more suited to an object perturbed by the sea state.

Model

The constant velocity kalman filter estimates the velocity and position of the target in the horizontal NED plane. Thus there are four states, x and y position, and x and y velocity.

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} North(p) \\ East(p) \\ North(v) \\ East(v) \end{bmatrix} \tag{4.12}$$

In the case of a constant velocity filter, only first order integration is performed. Thus, the Euler approximation of the discrete model is the same as the exact state model. Given an update period dt, the state matrix is:

$$A = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.13}$$

As there is actuator inputs, the input and feedthrough matrices B, D are empty.

The target position detected by the camera module is used as the measurements in the filter. While the filter operates at a fixed rate, there is no guarantee that the camera produces new measurements within each iteration of the filter. This yields the need for a time variant observation matrix.

Dead reckoning should be performed in iterations where no new measurement is available. During the dead reckoning steps, the model is updated using only predictions. This is achieved by setting all the elements in the observation matrix to zero. As the Kalman gain is given by $K = P_t H_t^T S_t^{-1}$, Thus, a zero observation matrix will remove the updating step of the filter, and only use the prediction.

However, when a new measurement is available, the measurements are are x_1 and x_2.

$$H_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \Bigg| \quad \text{if new measurement} \tag{4.14a}$$

$$H_t = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \Bigg| \text{if no new measurement} \tag{4.14b}$$

Matrix class

The matrices and vectors in the Kalman filter are stored as `DUNE::Math::Matrix` objects. The class supports standard matrix operations. In addition, the class has methods for constructing diagonal matrices, setting sub matrices, and accessing elements.

- `Matrix(float* diag, int n)` constructor for diagonal matrix
- `Matrix::put(int i, int j, Matrix)` sets a sub matrix anchored at element (i,j)
- `Matrix::element(i, j)` access element (i,j)

Initiate filter

The IMC message `IMC::VehicleCommand` is used to activate and deactivate the filter. The `info` field of the message is set to “track” if this module is the destination. If the command field is `VC_EXEC_MANEUVER`, the module enters the active state by setting the member variable `m_is_active`. Another member variable, `m_filter_is_init` is used to determine if a successful initialization has been performed.

Upon receiving a `IMC::CameraTracking` message, the horizontal position of the message is placed in the member variable `m_y`. Simultaneously, the `m_new_measurement` variable is set to true. Additionally in the case of `m_is_active == true && m_filter_is_init == false`, the initialization procedure is to be performed.

The initialization procedure sets the state matrix A and the observation matrix H as member variables of the task. The H matrix is set to the version to be used when new measurements are available. In order to properly set the A matrix, the execution period dt must be known. This is achieved by acquiring the frequency using the `Periodic::get_frequency()` method, and setting `m_dt = 1/get_frequency()`

In addition, the process noise Q, the observation noise R, and the error P covariance matrices are also set as member variables. They are diagonal matrices, with their values determined by a set of configuration parameters. The state vector *x* is set to the horizontal position in `m_y`, with zero velocity.

Finally, the `m_filter_is_init` variable is set to `true`, and the filter is ready for update steps to be made.

Update filter

At each iteration of the `task` method, the `m_is_active` and `m_filter_is_init` variables are checked. If either is `false`, the method is to return without updating the filter. In addition, if the time since the last measurement was received exceeds a configurable threshold, the method returns.

If none of the conditions to return are triggered, the filter is updated. A local variable `H` is created. If a new measurement is available, the matrix is set to `H = m_H`. However, if no new measurement is received, the matrix is set to `H = 0*m_H`. The steps used to update the filter are shown in Listing 4.1

Listing 4.1: Update Kalman filter

```
m_P = m_A*m_P*transpose(m_A) + m_Q;
S = m_R + H*m_P*transpose(H);
m_K = m_P * transpose(H) * inverse(S);
m_P = m_P - m_K * H * m_P;
m_x= m_A*m_x + m_K*(m_y-m_H*m_x)
```

Finally, the `m_new_measurement` is set to false.

Dispatching target

After the Kalman filter has been updated, the target should be dispatched to the Path Control module. The horizontal state of the target is set to the state of the Kalman filter. The vertical position however, is based upon the desired altitude, the measured radar altitude, and the current altitude in the NED frame used by Ardupilot.

The calculation of the target `z` value is made each time the `IMC::Distance` message is received. The message contains the measured radar altitude, which is placed in the `m_current_alt` variable. The `m_desired_alt` variable is set by the message `IMC::DesiredAltitude`. `m_z_current` is acquired from the `z` field in `IMC::EstimatedState`.

The error of the current altitude is calculated as
`z_error = m_current_alt - m_desired_altitude`
In this form `z_error` will be positive if the vehicle is above the desired height. As the z-axis in NED has a downwards direction, the desired `z` value is:
`m_desired_z = m_z_current + z_error`

The `IMC::ConstantBearingTarget` message is used to dispatch the target. Let the message be stored in `cbg_target]` .Using the Kalman filter state vector `m_x` and the desired z value `m_desired_z`, `cbg_target` is set as:

Listing 4.2: Set constant bearing target

```
    cbg_target.x  = m_x.element(0);
    cbg_target.y  = m_x.element(1);
    cbg_target.vx = m_x.element(2);
    cbg_target.vy = m_x.element(3);

    cbg_target.z  = m_desired_z;
    cbg_target.vz = 0;
```

4.2.5 Path Control

The path control module has been named Control.Path.ConstantBearing in DUNE, and inherits from the Periodic task. It implements the constant bearing guidance controller described in section 2.5.1.

Target

The constant bearing target consists of a NED frame position and velocity. Six member variables are used to represent the target, which are set when receiving a `IMC::ConstantBearingTarget` message.

Desired velocity

The desired velocity is calculated based on the target and the current position of the UAV. The UAV position is received from the `IMC::EstimatedState` message. In addition, two configurable variables are used: `Delta_p` and `U_max_a`. `U_max_a` is the approach speed when the target is far away, and `Delta_p` describes the transient behaviour of the approach speed as the vehicle closes in on the target.

Arrays of three 32 bit float values are used for the calculations. Functions for the two norm, dot product and scalar multiplication of such arrays are implemented for convenience.

The desired velocity `v_ref` is
`v_ref = v_a + m_target_v`
The approach velocity is calculated as
`v_a = scalar_x_arr_fp32(-kappa, pos_err_normalized)`
The normalized position error is a unit vector along the direction from the vehicle to the target. It is calculated as the position error divided by the norm of the position error.

`kappa` is the speed at which the vehicle should move. It is calculated based on the configuration parameters and the size of the position error position error. Details are presented in eq. (2.21).

4.2.6 UAV.Ardupilot

The Ardupilot module is named Control.UAV.Ardupilot in DUNE, and is a regular task. The task is primarily responsible for providing an abstraction to the MAVLink interface to the autopilot.

Two modifications are made to the task. One is to enable a velocity control, by consuming `IMC::DesiredControl` messages and sending the appropriate MAVLink message. The other is to handle the local NED frame navigation data received on MAVLink.

Velocity control

The MAVLink command used for velocity control is `SET_POSITION_TARGET_LOCAL_NED`. The neccesary fields in the message are:

- `coordinate_frame`
- `type_mask`
- `vx`
- `vy`
- `vz`

The coordinate frame should be the same as the one used for navigation. However, as only velocity is considered, any NED aligned frame will suffice. The frame is chosen as `MAV_FRAME_LOCAL_NED`.

The message supports control of position, velocity and acceleration. The type of control used is determined by the `type_mask`. In addition, the message supports control upon any subset of axes. However, the ardupilot documentation states that all velocity axes must be enabled for velocity control to function. The appropriate hex value to set velocity control is:
`type_mask = 0x0DC7`

The velocity fields of the message is determined by consuming the `IMC::DesiredControl` message. The IMC message contains a flag field for determining which axes of control is to be set. The flag is checked for the bits determined by `FL_X, FL_Y, FL_Z`. Letting the IMC message be stored in the pointer `d_vel`, the MAVLink message velocity is set by Listing 4.3

Listing 4.3: Set desired velocity

```
vx = (d_vel->flags & DesiredControl::FL_X) ? d_vel->x : 0;
```

```
vy = (d_vel->flags & DesiredControl::FL_Y) ? d_vel->y : 0;
vz = (d_vel->flags & DesiredControl::FL_Z) ? d_vel->z : 0;
```

Local navigation

UAV.Ardupilot contains a function named `handleArdupilotData()`, which is called in the main loop. The functions reads a message from MAVLink, and calls a parsing function on it depending on the message ID. The map between message ID's and appropriate parsing functions is stored in the `m_mlh` variable. There was no callback function for handling the local navigation MAVLink packet, and it has therefore been implemented.

The MAVLink message ID of the local navigation packet is 32, and is stored in the macro
`MAVLINK_MSG_ID_LOCAL_POSITION_NED = 32`
A function `Task::handleLocalPositionPacket(msg)` is created, and added to the callback map:
`m_mlh[32] = &Task::handleLocalPositionPacket;`

The callback function is shown in in Listing 4.4. The type `mavlink_local_position_ned_t` and function `mavlink_msg_local_position_ned_decode` are available as part of the mavlink library used by DUNE.

Listing 4.4: Handle local position MAVLinkpacket

```
void
handleLocalPositionPacket(const mavlink_message_t* msg)
{
  mavlink_local_position_ned_t lp;
  mavlink_msg_local_position_ned_decode(msg, &lp);

  m_estate.x = lp.x;
  m_estate.y = lp.y;
  m_estate.z = lp.z;
}
```

m_estate is an `IMC::EstimatedState` message used by the task to store navigation data. At the reception of an attitude packed, the contents of `m_estate` is copied into a `ExternalNavData` message. This `ExternalNavData` message is then dispatched, and is to be processed by the navigation task.

4.2.7 UAV.Navigation

The navigation task is named Navigation.UAV.Navigation in Dune, and is a non-periodic task. No modifications are made to the task in this project.

The task receives `IMC::ExternalNavData` messages from UAV.Ardupilot. This message contains the navigation data sent by the autopilot using MAVLink. Depending on the configuration of the navigation task, additional processing is applied to the navigation data. In particular, the task is responsible for handling RTK navigation if activated. As the UAV should not depend on RTK for the maneuvers, RTK integration is deactivated. In this configuration, the task simply feeds through the autopilot navigation data. The navigation data is published using the `IMC::EstimatedState` message.

Chapter 5

Experiments

5.1 Sensor test above water

A platform for was built for testing the behaviour of a monochrome camera and a radar while above water. The test was also to be used for determining how to integrate the radar, camera, and inertial measurements.

5.1.1 Hardware setup

The sensor platform, and the tools necessary for the test, consists of:

- Acrylic plate, $\approx$ 20x60x1cm
- Withrobot oCam 1MGN monochrome camera
- Novelda XeThru X4M03 radar development kit
- Pixhawk Cube with 3DR GPS unit
- Bracket for radar, 3D printed PLA
- Bracket for camera, 3D printed PLA
- USB hub with 4 USB 2.0 ports with power supply
- USB 2.0 cable, 30m, with power supply
- 3x USB cable, 1m
- Mains power extension cord and power strip
- Velcro

- Windows PC
- Crane on pier
- Yellow buoy, ≈ 15cm diameter

The camera and the radar was mounted to the acrylic plate using brackets printed in PLA plastic. The camera was mounted on top of the plate, with a hole drilled in the acrylic plate to fit the camera lens. The radar was mounted underneath the plate. As the Pixhawk and GPS module were already fitted with Velcro, complementary Velcro straps were glued on top of the Acrylic plate. In addition, the USB hub was fastened to the plate using Velcro.

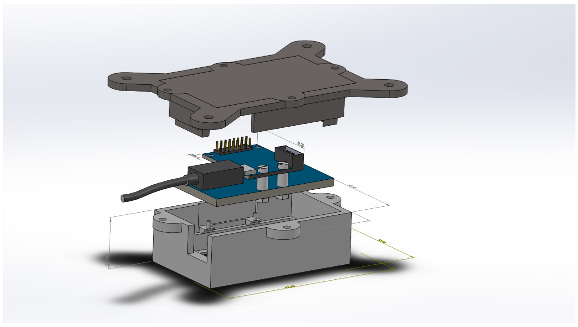


Figure 5.1: CAD model of the housing and bracket for the radar

The Pixhawk, the camera, and the radar was connected to the device ports of the USB hub, using the three 1m USB cables. The 30m USB cable was used to connect the PC to the PC port of the USB-hub. The power supply of the 30m USB-cable was connected to the hub side of the cable.

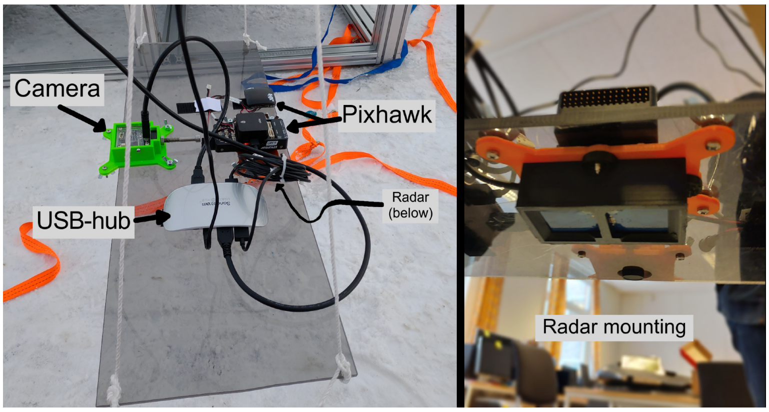


Figure 5.2: Sensor rig

The acrylic plate was suspended from the hook of the crane, using rope attached to each corner of the plate. The power strip and the 30m USB cable was also attached to the hook of the crane. Using the extension chord, the power strip was connected to a mains voltage supply on the pier. The power supplies of the USB cable and the USB hub were connected to the power strip, and the 30m USB cable was connected to the laptop.

5.1.2 Logging sensor data

Camera

The oCam camera manufacturer provides the source code for a C++ windows application for interacting with the camera, named "oCam-wiever_Win". This source code is found on the github repository for the camera:
`https://github.com/withrobot/oCam/`

The source code supports configuring the camera for different resolutions and frame rates. It is also used for setting exposure and brightness, in addition to some settings not used in this test. The program displays the output of the camera, and a button is used to save an image from the camera.

The source code was modified to enable rapid saving of consecutive image frames with timestamps. A button in the graphical user interface was created to activate and disable the logging functionality. The function `timeGetTime()` was used to represent the timestamp of each image, and was set as the image name. It returns a `DWORD` (32 bit unsigned int), which wraps around every 49 days. In order to relate this timestamp to the local system time, the function `GetLocalTime()` was used. At start up, the two time representations are subsequently read, and printed to a file. When post processing the data, this file is used to provide the offset relating the image frame timestamps to the local time.

Radar

The radar manufacturer, Novelda, provides a "ModuleConnector" library interface to the radar from Matlab. This library contains functions for setting up the radar, and acquiring and logging data frames.

A ModuleConnector object is created by calling the function
`mc = ModuleConnector.ModuleConnector(COMPORT, 0)`;
where `COMPORT` is a string corresponding to the port assigned to the radar. The radar interface is acquired by calling
`xep = mc.get_xep();` This radar interface is used to read frames from the radar, as well as to configure the radar. The configuration used is the same as the one described in section 4.2.2.

The logger object is constructed and started using Listing 5.1. It will then save the frames outputted by the radar to a set of files.

Listing 5.1: Setup radar logger

```
recorder = mc.get_data_recorder();
recorder.set_session_id('test_recording');
data_float_type = ...
    ModuleConnector.DataRecorderInterface.DataType_FloatDataType;
% Start recording.
recorder.start_recording(data_float_type,output_dir);
```

While the logging is performed, the frames acquired from the `xep` object are processed and plotted.

Pixhawk

The Pixhawk is interfaced from the Mission Planner program (`http://ardupilot.org/planner/`). This is a program for commanding the behaviour of the autopilot and monitoring the inertial navigation system (INS) through telemetry. In addition, it records the telemetry output of the Pixhawk. It was primarily used to ensure that the INS state was stable before the test maneuvers were performed.

The Pixhawk passively logs most of it's internal states, including INS, to files on its SD card. These files can be extracted using Mission Planner, as well as exported to a Matlab compatible format.

5.1.3 Testing maneuvers

The laptop was used to log and monitor the output of the camera, the radar and the Pixhawk. The application for monitoring the camera was used to set an appropriate exposure for the lighting conditions. The radar was set up using settings determined during testing in an office environment.

The crane lifted the sensor rig above the water and performed several steady descending and ascending maneuvers. In addition, a few descend/ascend maneuvers were performed while significant roll/pitch motion was induced on the sensor rig. The lowest altitude during these maneuvers were about 1.3m.

While the crane performed the descending and ascending maneuvers, the buoy was thrown into the water in the field of view of the camera. Due to the water current, the buoy drifted back to the pier, and out of view of the camera, in about 30s. The rope attached to the buoy was then used to retrieve the buoy and throw it back into the field of view of the camera. This process of throwing the buoy was repeated for the duration of the crane maneuvers.

5.1.4 Post processing

Radar altitude

Matlab was used to play back the radar log generated during the test maneuvers. A `DataPlayer` object was generated using
`player = ModuleConnector.DataPlayer(metafilename);`
From this object, a virtual radar interface was constructed using the functions
`mc = ModuleConnector.ModuleConnector(player,0);`
`xep = mc.get_xep();`
At this point, the `xep` objects acts as a regular radar interface without the capability of setting the radar configuration. The virtual radar will begin outputting frames at a call to the `DataPlayer` object.
`player.set_playback_rate(rate);`
The `rate` is used to scale the playback speed with respect to the real time speed at which it was recorded.

An adaptive cluttermap method is applied to the data to remove the constant background composition of the frames. A side effect is that slowly varying parts of the frame will also be suppressed. As the altitude of the radar was not kept stationary during the maneuvers, this did not lead to problems.

The adaptive cluttermap is described by eq. (5.1)

$$A_{filtered}[k] = (\alpha)A_{measured}[k] + (1-\alpha)A_{filtered}[k-1] \tag{5.1}$$

where $\alpha \in (0,\ 1)$. At the first iteration, the filtered amplitude is initialized to the measured amplitude.

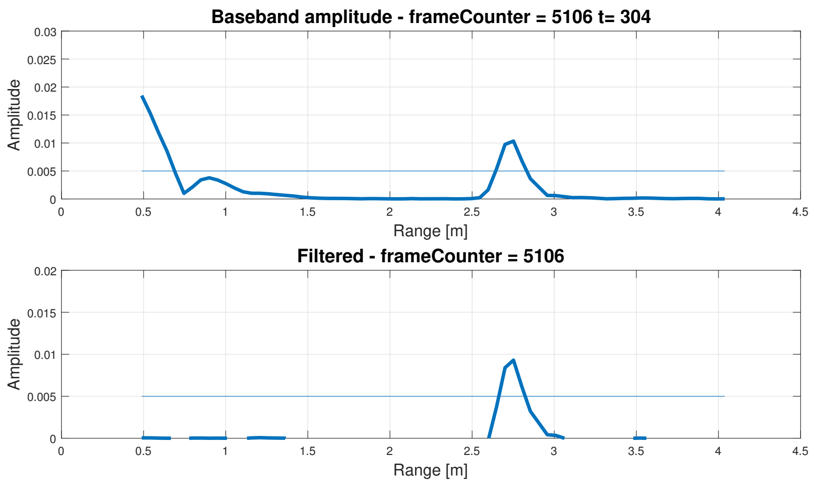


Figure 5.3: Radar output, filtered = adaptive cluttermap

The filtered output of the adaptive cluttermap is used for altitude estimation. The maximum value is chosen as an altitude candidate. If the amplitude of the candidate exceeds a threshold (0.005), it is accepted. The altitude is then set to the range of the candidate. If no new candidate is accepted during the next second, the altitude is assumed to be outside the range of the radar, which was set to 4m. In this case, the altitude is set to 5m. This is not meant as a proper estimate, but it makes it easier to spot when the altitude is out of range during analysis of plots.

Pixhawk attitude

The original plan was to use the SD card of the Pixhawk to extract INS logs. However, the log files appeared to be corrupt. As an alternative, the telemetry logs on the laptop were used. Telemetry data is sent at a significantly lower rate than the internal log. The telemetry log of attitude was sent at the default rate, $4Hz$. It would have been possible to increase the telemetry rate, but this was not done as telemetry was not intended for post processing.

The Mission Planner software was used to export the telemetry data to a Matlab compatible format. By navigating to the Telemetry logs tab and loading the appropriate log, the log could be played back and plotted. A plot was made containing the attitude (pitch, roll, yaw) data. The data in the plot was exported to a ".mat" file using the "Create Matlab file" button.

The ".mat" file was loaded in Matlab. The timestamp was represented using `double` values, with unknown mapping to local time. However, the plot in Mission Planner, used local time on the x-axis. The start and end time of the data was read off the plot. In Matlab, the time `double` time axis was then remapped to local time using linear interpolation.

Camera detection

The camera was calibrated using the Matlab Single Camera Calibrator App. This generated the camera intrisic matrix K, and a radial distortion function with two parameters.

As mentioned in section 5.1.2, the file names of the images are used to represent their timestamp. In addition, a file was used to provide an offset between the timestamps and local time. This file was firstly loaded in matlab, and a time remapping function was made.

The images were then loaded, wile generating the appropriate local time. For each image, the following operations were made:

- Undistortion
- Binarization
- Erosion
- Candidate (blob) rejection
- Dilation

- Circular Hough transform

As there is significantly different brightness between the sea and the buoy target, a binarization is applied at an appropriate threshold. This is performed using the function

```
img = imbinarize(img, level);
```

`level` was chosen as 0.3. Pixels darker than $0.3 * 255$ are then set to 0, while brighter pixels are set to 1.

Erosion is then applied to the image to remove noise and small bright areas in the image. A structuring element approximating a disk with a radius of 5 pixels is created, and stored in `se_erode`. The erosion is performed using:

```
img = imerode(img, se_erode);
```

Dilation is applied to attempt to reconstruct the shape of the regions still containing active pixels. A new structuring element, `se_dil` is created. It approximates a disk with a radius of 10 pixels. The increased size is to ensure that dark spots on the buoy are filled in. The dilation is performed by calling:

```
img = imdilate(img, se_dil);
```

Due to parts of the pier being in the image at several instances, the above approach might include large blobs on the pier. Blobs with unreasonably large area are therefore discarded.

```
img = img - (bwareaopen(img, max_area));
```

`max_area` is the required number of pixels in a blob that should be discarded, and is set to 40'000.

A circular Hough transform is applied to the blob regions, operating on the original grayscale image. The search radii are hard coded to include the expected pixel size of the buoy over the range of altitudes in the test. To reduce the computational cost, the radii can be chosen based on the altitude. The Hough transform outputs center, radius and a certainty metric.

```
[centers, radii] = imfindcircles(roi, [r0 r1], ...
                   'ObjectPolarity','bright')
```

`r0=10` and `r1=30` represent the range of search radii. `roi` is the original image cropped to a region of interest.

Target position and surface coordinate frame

After having performed the detection stage on an image, position estimation is to be performed. To synchronize the time of the image, radar and Pixhawk, the altitude and attitude were interpolated to the time of the image.

A coordinate system fixed to the sea surface directly below the camera was drawn in the image. This was done by specifying homogeneous coordinates of rectangles representing the axes. The points were transformed from a surface fixed NED frame to the camera frame using the transformation T_N^C

$$T_N^C = \begin{bmatrix} R_N^C & R_N^C t \\ 0_{1x3} & 1 \end{bmatrix} \tag{5.2a}$$

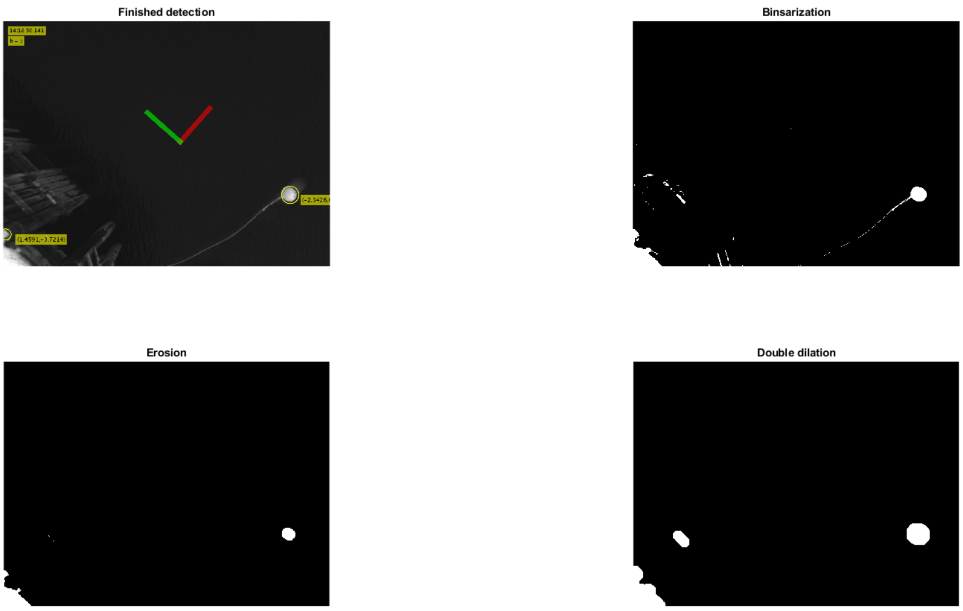


Figure 5.4: Detection algorithm, region of interest

$$t^T = [0, 0, h] \tag{5.2b}$$

Once in the camera frame, the points are projected to image coordinates using the camera intrisic matrix K. The rectangles are then drawn using the pixel coordinates of the world points and the function `insertShape(...)`

The method described in section 3.2 and section 4.2.3 is used to estimate the target position of the detected centers. The position is calculated with respect to the drawn surface frame. The camera to body rotation R_B^C was approximated as $[\phi,\ \theta,\ \psi]^T = [0°,\ 0°,\ 90°]^T$. The attitude angles from the Pixhawk formed the body to NED rotation R_B^N.

Movie generation

After processing an image, information such as altitude, time, and detected horizontal positions were printed on the image. The images were then saved to an output folder. The "ffmpeg" (`https://ffmpeg.org`) command line tool was then used to generate a movie of the output. This was used to better visualize the behaviour of the altitude and position estimation.

While no guarantee is made for the future availability of the url, the movie has been uploaded to:
`https://youtu.be/1PROmScl7UI`

5.1.5 Results and discussion

Radar altitude

The estimated altitude is shown in fig. 5.5. The initial section where the altitude is about 1m correspond to the rig being above the pier. The constant 5m sections correspond to no altitude candidate being accepted for at least a second. While no ground truth is available,

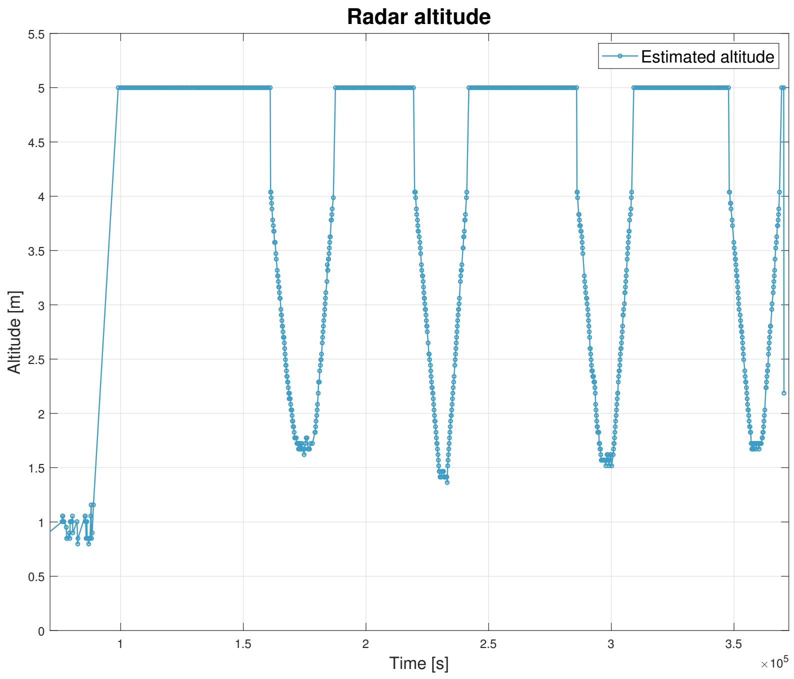


Figure 5.5: Radar altitude estimation from sensor rig test

the general trend of the descending and ascending altitude fits the maneuvers performed with the crane. In addition, the behaviour of the altitude printed in the generated movie is observed to correspond to the perspective height of the camera.

The method of using an adaptive cluttermap for altitude estimation appears to produce sensible results during the test maneuvers. However, the maneuvers were all in a steady decent or ascent. In the multirotor application, it is desired that the multirotor holds a given altitude for an extended period of time. This is to allow the target model filter to converge to accurate estimates of the target state. As this has not been tested, the feasibility of using an adaptive clutter map is not proven.

When at a constant altitude, it expected that the radar output is relatively stationary. As the adaptive cluttermap will eventually remove the stationary signal, it is likely that the altitude tracking will be lost. However, this can't be fully concluded without further test data.

Detection and position estimation

The movie generated in section 5.1.4 shows that the buoy is consistently detected when it is in the image. However, several false positives are present. These false positives were related to pier, usually corresponding to snowy patches. No false positives present on the sea surface. It is therefore assumed that if only the sea and the buoy had been in the field of view, no significant false positives would be present.

The motion of the drawn coordinate frame is observed to oppose the motion of the camera. There is some drifting, as well as some abrupt changes in the motion of the coordinate system. The drifting might be caused by projecting the coordinate system directly below the camera, instead of using an (approximately) inertial NED frame. As the camera was not horizontally stationary, the coordinate system projected to the sea surface can not be expected to be stationary either. Due to the rope suspension of the sensor rig, the rig acted as a pendulum on the crane. The induced roll and attitude maneuvers therefore also induced significant horizontal travel as well.

The abrupt changes in the motion of the coordinate system is likely due to the low rate of the telemetry attitude data. Additionally, the some of the telemetry data might be lost.

The position estimate is observed to correctly relate to the coordinate system projected to the sea surface.

5.1.6 Applicability to the problem statement

While no ground truth is available for either the altitude or the position of the target, the test serves the purpose of validating the concept.

The observed motion of the coordinate frame, opposing the motion of the camera, validates the rotations between camera and NED frame. The accuracy of this opposing motion, and its effect on error in position estimates is not investigated.

As the detection algorithm was constructed during post processing of the test data, no presumption regarding generalization is made. In particular, varying light conditions is likely to affect the ability to detect the buoy without producing false positives. Additionally, different wave and wind conditions may induce shapes on the sea leading to false positives.

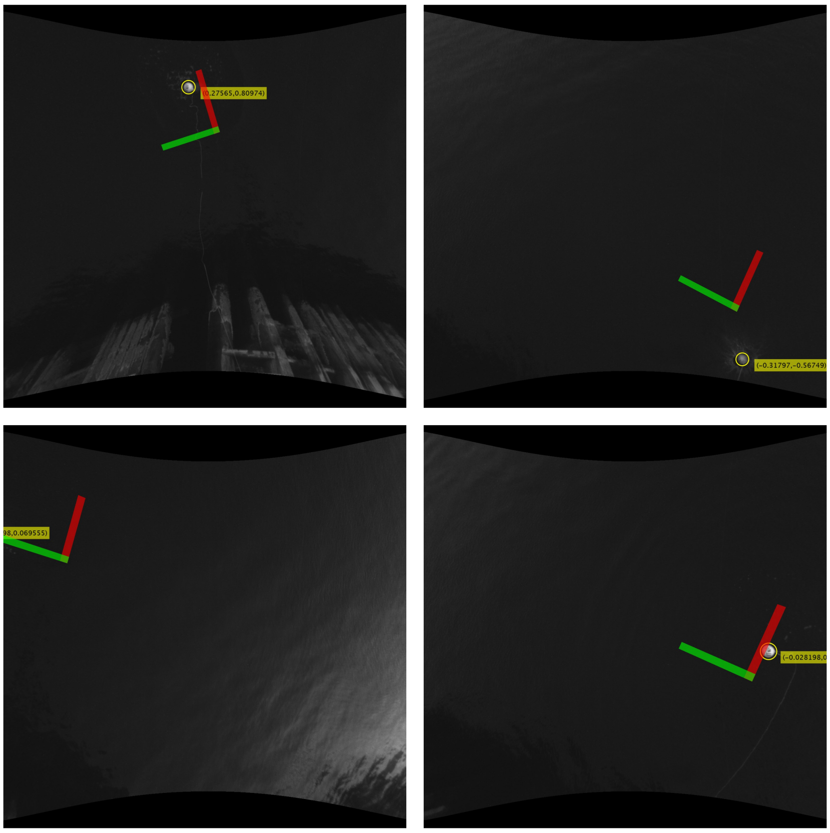


Figure 5.6: Coordinate system opposing the motion of the camera

5.2 DUNE flight control simulation

A simulation setup is configured to test the Supervisor and the Path Control modules. The simulation presented here is performed on a desktop PC. While not covered in this section, the same setup has been used to test the Camera module and the Target model filter module.

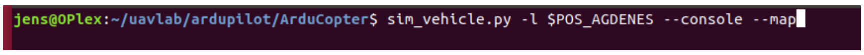
The simulation consist of a Software in The Loop (SIL) configuration of DUNE, and an Ardupilot simulation of a multirotor with autopilot software. The communication between DUNE and Ardupilot is performed over a TCP socket.

5.2.1 ArduPilot multirotor simulation

The ArduPilot simulation setup section is reused from (Joberg, 2018)

The ArduPilot multirotor simulation is launched by the running the python script:
`sim_vehicle.py`
from the terminal while in the `ardupilot/ArduCopter` directory. The python script is added to the PATH environment in the installation procedure of the ArduPilot software. The initial location of the the multirotor is set using the argument:
`-l [lat],[lon],[geodetic_alt],[heading]`
Where the unit of the altitude is in meters, and the rest of the parameters are in degrees. The location used in this experiment is at Agdenes Airport, and the corresponding location argument is:
`-l 63.62831245,9.72534,50,60`
For convenience, the location data is stored in an environment variable, and the final command to start the ArduPilot simulation is shown in fig. 5.7. The additional arguments launches a live map of the UAV and an information console, which outputs useful in flight information. However, these arguments may be omitted.

During startup, MAVProxy is launched in the terminal. MAVProxy is a ground control station (GCS) tool for the UAV, and commands can be used to control various ArduPilot functionalities. In this experiment MAVProxy will only be used to take off and switch the ArduPilot mode, which is used as a trigger in the DUNE state machine. In order to take off, the UAV must be armed, and supplied with positive throttle. This is done with the consecutive commands `arm throttle` `rc 3 1600` Once the UAV is airborne, the autonomous system can take control. This is enabled by calling:
`mode GUIDED`
Further information on the basic functionalities of MAVProxy is documented on the ArduPilot webpage:
`http://ardupilot.github.io/MAVProxy/html/index.html`



```
jens@OPlex:~/uavlab/ardupilot/ArduCopter$ sim_vehicle.py -l $POS_AGDENES --console --map
```

Figure 5.7: Command to launch ArduPilot simulation

5.2.2 DUNE setup

Constant bearing target generator

A new DUNE task is created to replace the Target model filter, named Simulators.TargetGenerator. The purpose of this task is to generate a reproducible constant bearing guidance target for the Path controller to track.

The TargetGenerator task replaces the Target model filter module's interface to the Supervisor. However, the interface to the camera and the radar is ignored. The task state is controlled by the `IMC::VehicleCommand` message with info field "track". If the command field is `VC_EXEC_MANEUVER`, the module is activated, and disabled if it is `VC_STOP_MANEUVER`.

The task generates position and velocity setpoints for maneuvering the vehicle in a horizontal circular motion. It is represented as a scaled unit circle. At the activation of the task, the phase is set to zero, and the vehicle position is saved as (x_0, y_0, z_0).

Two configuration parameters are supplied to the task: `v_reduction` and `radius`. `v_reduction` increases the amount of time the vehicle should use to traverse the circle, and `radius` scales the radius of the circle.

The phase θ of the circle is calculated as:

$$\theta = \frac{t}{v\_reduction} \tag{5.3}$$

The desired position and velocity is then calculated by eq. (5.4)

$$x = x_0 + radius * \big(-1 + \cos(\theta)\big) \tag{5.4a}$$

$$y = y_0 + radius * \sin(\theta) \tag{5.4b}$$

$$z = z_0 \tag{5.4c}$$

$$v_x = -\sin(\theta)\frac{radius}{v\_reduction} \tag{5.4d}$$

$$v_x = \cos(\theta)\frac{radius}{v\_reduction} \tag{5.4e}$$

$$v_z = 0 \tag{5.4f}$$

The position and velocity is placed in a `IMC::ConstantBearingTarget` message and dispatched.

Configuration file

A DUNE configuration file, named "ntnu-hexa-H2O.ini", is created to enable the appropriate tasks with desired parameters. The line
`[Require uav/arducopter.ini]`
is placed at the top to enable the basic tasks required for multirotor configurations. The relevant modules are enabled with the lines in Listing 5.2.

Listing 5.2: Module configurations

```
[Supervisors.H2O_Pickup]
Enabled                                  = Always
Entity Label                             = Supervisor_H2O_Pickup
```

```
[Control.Path.ConstantBearing]
Execution Frequency                  = 10
Enabled                              = Always
Entity Label                    = Constant_Bearing_Guidance

[Simulators.TargetGenerator]
Execution Frequency                  = 10
Enabled                              = Always
Entity Label                         = CBG_Target_generator
Speed reduction                      = 4
Radius                               = 4
```

The DUNE program is then launched from the DUNE build folder. The profile used is AP-SIL (Ardupilot SIL). The command to launch the configuration is:
`./dune -c ntnu-hexa-H2O -p AP-SIL`

5.2.3 Simulating maneuver

Once the simulations of both DUNE and Ardupilot are started, the ardupilot INS needs to be properly initialized. When initialization of INS is done, the message
`EKF2 IMU{0, 1} is using GPS`
will be displayed in the MAVProxy console. After this, a manual take off is performed, before putting the Ardupilot into `GUIDED` mode. This will activate the control system.

Upon receiving control of the multirotor, the Supervisor module will activate the Path Control module and command a waypoint at a vertical increment to the current position. When reaching the waypoint, the radar calibrate command is sent. As the radar module is not enable, the `IMC::CLI` message expected by the supervisor is manually sent. The text field of the message is set to `return radar calibrate`. The multirotor then descends to the position where the Supervisor assumed control of the vehicle.

After descending, the supervisor moves to the tracking state, and control is handed over to the Target Generator. The target generator then sends the position and velocity of a target moving in a circular manner as described in section 5.2.2

Results

The vehicle is observed to correctly increase its altitude for the radar calibration in fig. 5.8. After the calibration return message was received, the vehicle moved back to the starting position. The circular target tracking was then begun. As shown in fig. 5.8, a smooth circle trajectory was performed.

The absolute value of the position error is shown in fig. 5.9. After having converged, the position error is observed to be less than 4cm.

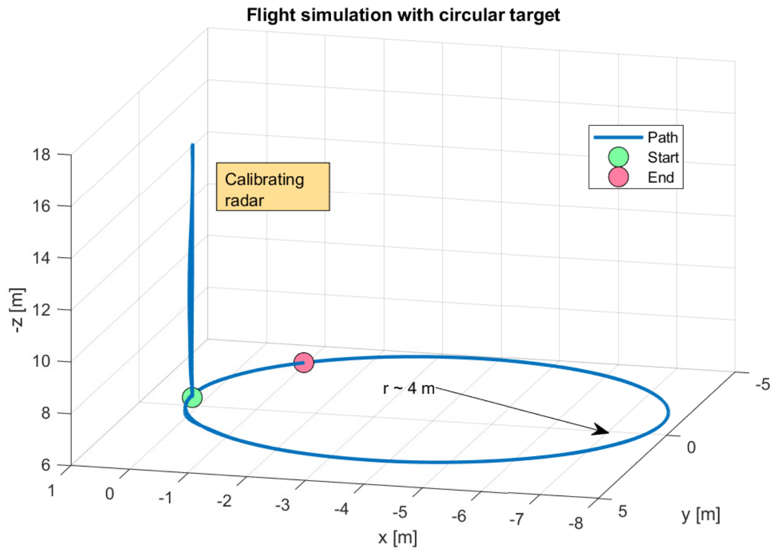


Figure 5.8: Dune flight simulation maneuver

Additional simulations

While the radar, camera and model filter modules were omitted from the experiment setup above, they have been included in other simulations. Unit tests of the modules have been performed in an office environment using an ArUco marker. By presenting the ArUco marker to the right in the image, the vehicle is observed to move to the right, independently of its heading. Similar results are achieved by presenting the ArUco marker to the left, top and bottom of the image. The target filter module is also observed to converge to a steady state when the image is stationary in the image.

5.3 Flight test

A flight test was conducted where the Odroid and the sensors were connected to a Pixhawk on an DJI S1000 octacopter. The payload consisted of

- Odroid XU4
- Radar with bracket
- Camera with bracket
- Traco DCDC voltage converter
- Logic level shifter

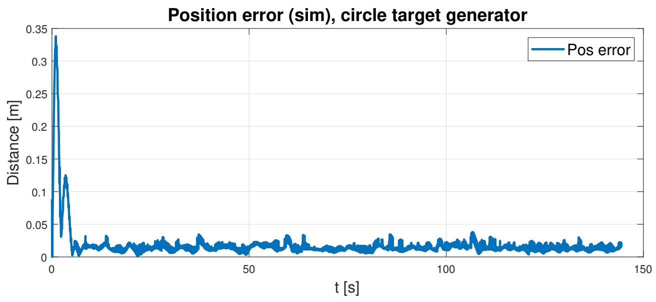


Figure 5.9: Position error during circular tracking maneuver

- 3.3V voltage regulator
- Container box
- 2x M5 Rocket (wireless bridge)

5.3.1 Hardware setup

Serial connection Odroid-Pixhawk

The MAVLink telemetry link between the Odroid and the Pixhawk is realized as a TCP connection over a serial UART link. The Odroid operates on 1.8V logic while the Pixhawk operates on 3.3V logic. A logic level converter was therefore placed between them.

The UART Rx and Tx wires of the Odroid were connected to the low voltage (LV) logic pins of the converter. The 1.8V VCC and ground wires of the Odroid were connected to the LV high reference and ground pins of the converter.

Similarly, the Rx and Tx wires of the Pixhawk were connected to the high voltage (HV) logic pins of the converter. The Pixhawk ground wire was connected to the converter ground pin. However, no 3.3V VCC was available on the Pixhawk to supply the HV reference. Therefore, a 3.3V voltage regulator was placed between a 5V VCC on the Pixhawk and the converter HV reference.

Payload container

A container box was used to mount the payload components in. The radar bracket created in fig. 5.1 was placed underneath the container. The camera bracket was modified to accommodate mounting points for the DCDC converter and the level shifter. The bracket is illustrated in fig. 5.10. The camera, level converter and DCDC were fastened to the bracket, before the bracket was mounted inside the payload container. Finally, the Odroid was mounted inside the container.

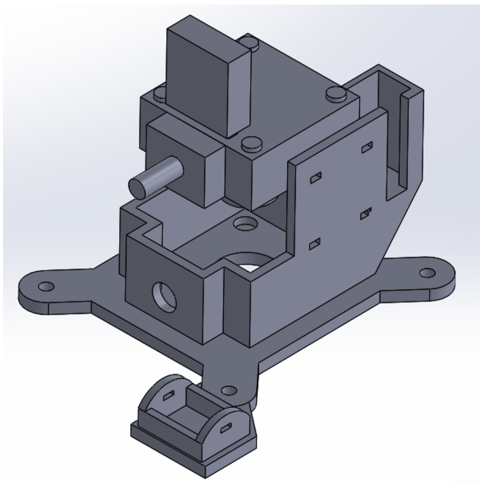


Figure 5.10: Mounting bracket for camera, DCDC and level shifter

An XT60 connector was used to connect the input voltage wires of the DCDC converter to a battery on the octacopter. The 5V output wires of the DCDC converter was fitted with a DC barrel jack. This jack was connected to the input power of the Odroid. The camera was connected to the Odroid using a USB 3.0 port, and the radar was connected to a USB 2.0 port.

Finally, the payload container was mounted to rails beneath the center of the multirotor. The UART cable from the payload was connected to a telemetry port of the Pixhawk using a 6 pin DF13 headers. The DCDC was connected to a battery. The Ethernet port of the Odroid was connected to a network switch, in order to communicate with the M5 Rocket on the multirotor.

5.3.2 DUNE configuration

The configuration used in section 5.2.2 was expanded upon to enable the camera, the radar and the target model filter module. The DUNE task `Transports.SerialOverTCP`, which is enabled by the `arducopter.ini` file, was configured to use the `/dev/SAC2` UART interface.

In addition, UDP streaming of every ¡n¿ camera frame and radar frame was implemented. The decimation factors ¡n¿ were set as configuration parameters.

As the hardware specific tasks should be run, the DUNE profile used was “Hardware”. This configuration was launched using:

```
./dune -c ntnu-hexa-H2O.ini -p Hardware
```

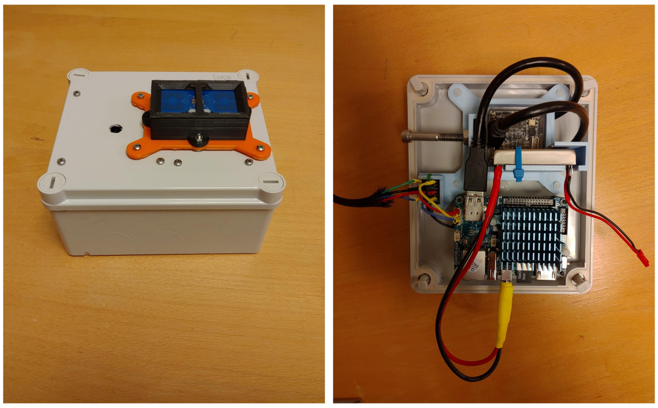


Figure 5.11: Payload overview

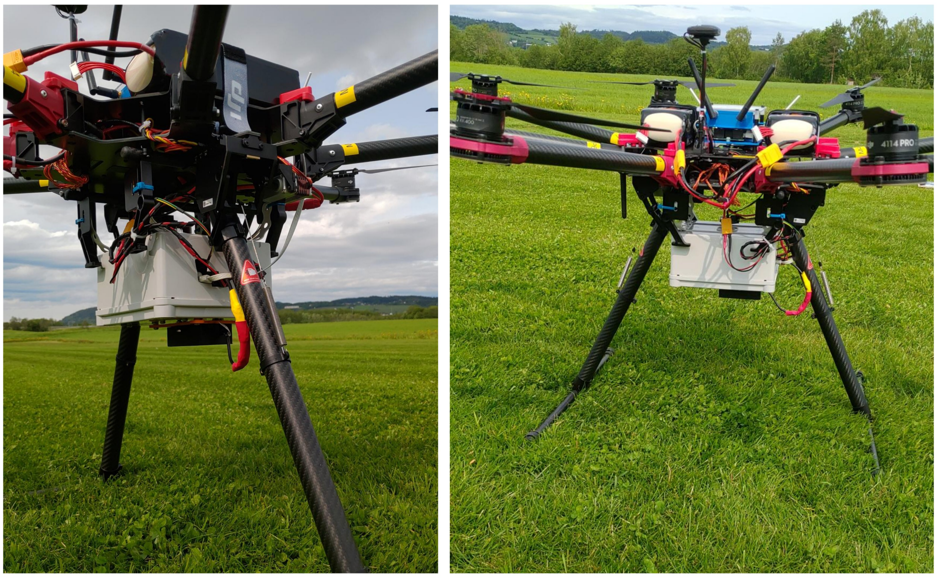


Figure 5.12: Mounting of payload to multirotor

5.3.3 Maneuvers

The original plan was to perform unit tests of the radar and the camera. However, an issue occurred with the camera, and the image output quickly deteriorated. The images became unusable before any tests were performed.

In addition, it was discovered that the grass and soil of the airfield highly attenuated the radar signal. The reflections only became visible at altitudes below 2.5m. The radar signals above 1.5m were hard to distinguish clearly from noise. As such, no altitude estimation was performed with the radar.

Instead, it was chosen to reproduce the simulation maneuver in section 5.2, using the constant bearing target generator task. The radar and camera modules were running during thees maneuvers to attempt to log some usable data. In addition, the calibration procedure of the radar was performed without the need for a manual return message.

Procedure

The pilot performed the take off with the multirotor, and brought it to a safe altitude. The autopilot was then set to `GUIDED` mode, and DUNE received control of the multirotor. The multirotor then increased it's altitude, and begun the radar calibration procedure. When the procedure was complete, the multirotor moved back to the position where it received control. The Target Generator task was then activated, and the multirotor moved in a circular maneuver. After several circles, the pilot assumed control of the vehicle, and performed the landing.

5.3.4 Results

The vehicle position during the segment where DUNE was in control of the vehicle is presented in fig. 5.13. The position error observed during the circular tracking is presented in fig. 5.14.

5.3.5 Discussion

Circular tracking procedure

The vehicle motion shown in fig. 5.13 is significantly more noise than the simulation results in fig. 5.8. This is likely due to the absence of noise sources, such as wind and gusting, in the simulation. In addition, it is likely that the controller used in the simulation is better tuned to its model, than the real counterpart.

An additional source that could have deteriorated the performance, is that the Path Control and Target Generator modules were run at lower rates during the flight test. During sim-

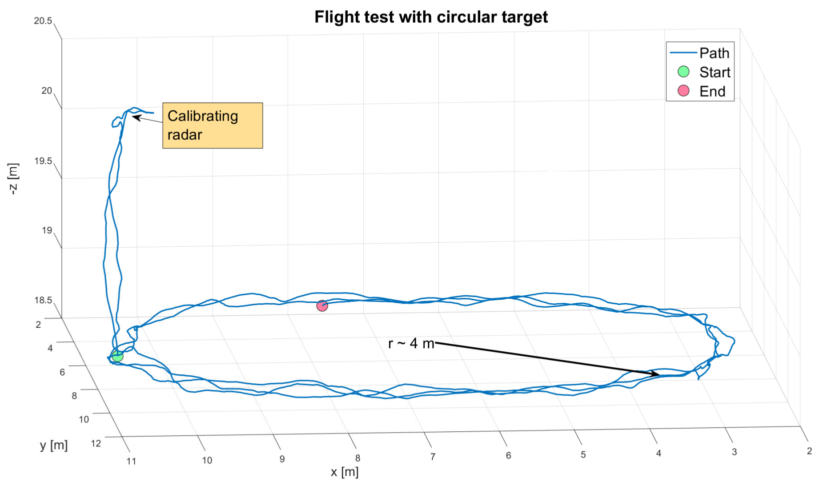


Figure 5.13: DUNE controlled flight maneuver

ulation, it has been observed that halving the rates will more than double the error. The rates were reduced in order to free up some resources for the camera and radar modules.

The position error illustrated in fig. 5.14 is observed to be mostly within 20cm. In conjunction with the circle plot, it appears as the controller did not converge to a steady state. Instead, it seems to oscillate slightly. This might be improved by tuning the autopilot velocity controller, as well as the transient behaviour of the constant bearing module.

Camera issue

The camera failure had not been experienced during office testing. Some of the parameters that changed from testing was:

- Increased temperature
- Structural stress on USB cable
- Vibration from multirotor
- Brighter light conditions.

During office testing, the payload was usually left open, such that airflow was available for cooling. In addition, when closing the payload lid, the lid would put significant pressure on the camera side of the USB connection. This connection was bent at an angle after testing.

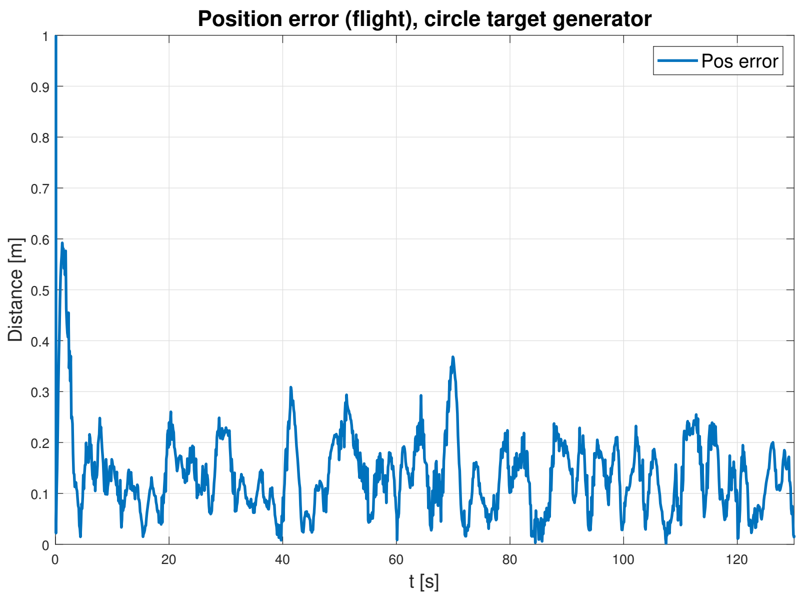


Figure 5.14: Position error during circular tracking

Chapter 6

Conclusion

A state machine governing the stages of the flight has been formulated. In order to realize the state machine, an architecture supporting the functionality required by the state machine, is proposed. The set of modules, and their interfaces has been implemented in DUNE. As part of this implementation, libraries for interfacing an X4M03 and a Withrobot oCam camera has been integrated in DUNE.

A triangulation method for integrating camera detections, altitude estimation and attitude angles has been developed and implemented in DUNE. This triangulation method provides position estimates, which is then used as measurement in a constant velocity Kalman filter.

The sensor rig described in section 5.1 was used to validate the feasibility of the triangulation method. In addition, an object detection method was developed, as well as a method for radar altitude estimation. However, the methods were made during post-processing, and no assumptions about generalization is made for performance in different conditions.

Simulations were performed to test the DUNE modules. In order to test the constant bearing path controller, a target generator was implemented. This generator provides position and velocity setpoints for a target traversing a circle with configurable radius and velocity. The precision of the maneuver was very promising, with a position error of a few centimeters.

A flight test was conducted with a configuration corresponding to the simulation test. The same type of circular maneuver was attempted. However, the accuracy was much worse in real conditions. Some potential sources of the reduced performance can be improved upon, such as sampling rates, and controller tuning. However, other error sources which are harder to improve upon may be present too.

6.1 Further work

While sanity checks have been made for most of the modules, ground truth data has not been acquired as part of this project. Future iteration should verify the metrics of the performance potential of the systems.

The architecture proposed allows for major reworks to single modules, as long as the interface is kept similar. The target model filter module should be expanded upon to better represent the sea state, as only the sea current is modeled at this point. A camera detection algorithm more suitable to the final target should be made as well.

Bibliography

Araar, O., Aouf, N., Vitanov, I., 2017. Vision based autonomous landing of multirotor uav on moving platform. Journal of Intelligent & Robotic Systems 85 (2), 369–384.

ArduPilot, 2018. Autopilot suite. `https://ardupilot.org`, accessed: 8th Dec. 2018.

Borowczyk, A., Nguyen, D.-T., Phu-Van Nguyen, A., Nguyen, D. Q., Saussié, D., Le Ny, J., 2017. Autonomous landing of a multirotor micro air vehicle on a high velocity ground vehicle. IFAC-PapersOnLine 50 (1), 10488–10494.

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

Corke, P., 2017. Robotics, Vision and Control: Fundamental Algorithms In MATLAB, Second Edition, 2nd Edition. Springer Publishing Company, Incorporated.

Falanga, D., Zanchettin, A., Simovic, A., Delmerico, J., Scaramuzza, D., 2017. Vision-based autonomous quadrotor landing on a moving platform. In: Proceedings of the IEEE International Symposium on Safety, Security and Rescue Robotics, Shanghai, China. pp. 11–13.

Fossen, T. I., 2011. Handbook of marine craft hydrodynamics and motion control. John Wiley & Sons.

Frølich, M., 2015. Automatic ship landing system for fixed-wing uav. Master's thesis, NTNU.

Joberg, J., 2018. Specialization project, multirotor pickup of objects in the sea.

Johansen, T. A., Fossen, T. I., 2017. The exogenous kalman filter (xkf). International Journal of Control 90 (2), 161–167.
URL `https://doi.org/10.1080/00207179.2016.1172390`

Lange, S., Sünderhauf, N., Protzel, P., 2008. Autonomous landing for a multirotor uav using vision. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2008). pp. 482–491.

Lange, S., Sunderhauf, N., Protzel, P., 2009. A vision based onboard approach for landing and position control of an autonomous multirotor uav in gps-denied environments. In: Advanced Robotics, 2009. ICAR 2009. International Conference on. IEEE, pp. 1–6.

Lasson, Ø. R., 2018. Autonomous landing of multi-rotor uav. Master's thesis, NTNU.

Line, V., 2018. Autonomous landing of a multirotor uav on a platform in motion. Master's thesis, NTNU.

Ling, K., Chow, D., Das, A., Waslander, S. L., 2014. Autonomous maritime landings for low-cost vtol aerial vehicles. In: Computer and Robot Vision (CRV), 2014 Canadian Conference on. IEEE, pp. 32–39.

MAVLink, 2018. Micro aerial vehicle link, protocol. `https://mavlink.io/en/messages`, accessed: 8th Dec. 2018.

Nguyen, T. H., Cao, M., Nguyen, T.-M., Xie, L., 2018. Post-mission autonomous return and precision landing of uav. In: 2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV). IEEE, pp. 1747–1752.

Novelda, 2018. XeThru X4 Radar User Guide - Rev. A.