Simultaneous localization and mapping with the AR.Drone

Nick Dijkshoorn

July 14, 2012

Master's Thesis for the graduation in Artificial Intelligence

Supervised by Dr. Arnoud Visser

Abstract

The small size of micro aerial vehicles (MAVs) allows a wide range of robotic applications, such as surveillance, inspection and search & rescue. In order to operate autonomously, the robot requires the ability to known its position and movement in the environment. Since no assumptions can be made about the environment, the robot has to learn from its environment. Simultaneous Localization and Mapping (SLAM) using aerial vehicles is an active research area in robotics. However, current approaches use algorithms that are computationally expensive and cannot be applied for real-time navigation problems. Furthermore, most researchers rely on expensive aerial vehicles with advanced sensors.

This thesis presents a real-time SLAM approach for affordable MAVs with a down-looking camera. Focusing on real-time methods and affordable MAVs increases the employability of aerial vehicles in real world situations. The approach has been validated with the AR.Drone quadrotor helicopter, which was the standard platform for the International Micro Air Vehicle competition. The development is partly based on simulation, which requires both a realistic sensor and motion model. The AR.Drone simulation model is described and validated.

Furthermore, this thesis describes how a visual map of the environment can be made. This visual map consists of a texture map and a feature map. The texture map is used for human navigation and the feature map is used by the AR.Drone to localize itself. A localization method is presented. It uses a novel approach to robustly recover the translation and rotation between a camera frame and the map. An experimental method to create an elevation map with a single airborne ultrasound sensor is presented. This elevation map is combined with the texture map and visualized in real-time.

Experiments have validated that the presented methods work in a variety of environments. One of the experiments demonstrates how well the localization works for circumstances encountered during the IMAV competition. Furthermore, the impact of the camera resolution and various pose recovery approaches are investigated.

Parts of this thesis have been published in:

N. Dijkshoorn and A. Visser, "Integrating Sensor and Motion Models to Localize an Autonomous AR. Drone", *International Journal of Micro Air Vehicles*, volume 3, pp. 183-200, 2011

A. Visser, N. Dijkshoorn, M. van der Veen, R. Jurriaans, "Closing the gap between simulation and reality in the sensor and motion models of an autonomous AR.Drone", *Proceedings of the International Micro Air Vehicle Conference and Flight Competition (IMAV11)*, 2011. **Nominated for best paper award**

N. Dijkshoorn and A. Visser, "An elevation map from a micro aerial vehicle for Urban Search and Rescue - RoboCup Rescue Simulation League", *to be published on the Proceedings CD of the 16th RoboCup Symposium, Mexico*, June 2012

Winner of the RoboCup Rescue Infrastructure competition, Mexico, June 2012

Acknowledgements

I would like to thank my supervisor Arnoud Visser for his great support and guidance. I like to thank Parrot S.A. for providing an AR.Drone for the competition. Martijn van der Veen and Robrecht Jurriaans took the initiative to compete in the IMAV competition and did both independent research on obstacle avoidance [1] and force field navigation [2]. I like to thank Carsten van Weelden for his experiments to validate the motion model of the AR.Drone. I like to thank Bas Terwijn and Edwin Steffens for helping setting up a laser rangefinder. Furthermore, I would like to thank Duncan Velthuis for his help during experiments. I am thankful to my parents for their support and encouragement throughout the years.

Contents

1 Introduction

A major goal of robotics is to develop mobile robots that can operate fully autonomously in real world situations [3]. These autonomous robots can be used for a wide range of applications. For example, cleaning, inspection, transportation tasks or medical and construction assistance. Robots can also operate in dangerous environments (e.g., life rescue or pollution control) without risking human lives. Although much progress has been made, a truly autonomous robot that can operate in the real world, has not been developed yet.

One of the main prerequisites of an autonomous robot is the ability to known its position and movement in the environment. Since no assumptions can be made about the environment, the robot has to learn from its environment. This ability has been identified as a fundamental problem in robotics. The process of incremental map construction and using it for localization is called **Simultaneous Localization and Mapping (SLAM)**. Its main function is to aggregate observations obtained by sensors in order to obtain information of the environment and store it in a map. This location information is used by other subsystems of the robot, such as interaction with the environment.

A wide range of SLAM methods have been presented. These methods include various sensor configurations to obtain map information as well as knowledge about the robots location. Available solutions for certain environments and sensor configurations are well understood. For other environments and sensor configurations, open problems remain. One of them is SLAM with micro aerial vehicles (MAVs), which have a limited sensor suite due to their weight constraints.

Figure 1.1: The Aeryon Scout is an unmanned micro aerial vehicle, used for tactical, over-the-hill aerial intelligence. The platform is controlled by a touchscreen interface and is able to fly pre-planned flight paths using GPS positioning.

A **micro aerial vehicle (MAV)** is a class of unmanned aerial vehicles (UAV). Their small size allows a wide range of robotic applications, such as surveillance, inspection and search & rescue. An example of a MAV developed for surveillance and inspection is the Aeryon Scout[1] (Figure 1.1), which is already being deployed in the field[2]. Particularly interesting are small quadrotor helicopters, which are lifted and propelled by four rotors. They offer great maneuverability and stability, making them ideal for indoor and urban flights. Due to technical developments in the last years, small quadrotors with on-board stabilization like the Parrot AR.Drone can be bought off-the-shelf. These quadrotors make it possible to shift the research from basic control of the platform towards intelligent applications that require information about the surrounding environment. However, the limited sensor suite and the fast movements make it quite a challenge to use SLAM methods for such platforms.

1.1 International Micro Air Vehicle competition

The International Micro Air Vehicle competition (IMAV) is an effort to stimulate the practical demonstration of MAV technologies. The competitions have as goal to shorten the road from novel scientific insights to application of the technology in the field. Since the Summer 2011 edition of IMAV, teams can borrow a Parrot AR.Drone quadrotor helicopter. This allows teams to focus research on the artificial intelligence part of the competitions.

Currently, IMAV has three distinct competitions: two indoor competitions and one outdoor competition. One of the indoor competitions is the Pylon challenge (Figure 1.2). The objective of the Pylon challenge is to navigate a MAV in figure-8 shapes around two poles. The competition rules[3] have been created to stimulate the level of autonomy; significantly more points are given to fully autonomous flights.

The main bottleneck for indoor autonomy is reliable indoor positioning. Team are allowed to use external aids to solve their positioning problem, at the cost of points. Examples of external aids are visual references (markers) and radio positioning beacons. One of the contributions of this thesis is the development of basic navigation capabilities for the AR.Drone, without relying on external aids. These navigation capabilities can be used during the IMAV competitions, allowing fully autonomous flights.

1.2 RoboCup Rescue

Another effort to promote research and development in robotics is RoboCup [4]. The RoboCup competions have as goal to stimulate research by providing standard problems where wide range of technologies can

[1] http://www.aeryon.com/products/avs.html

[2] http://www.aeryon.com/news/pressreleases/271-libyanrebels.html

[3] http://www.imav2011.org/images/stories/documents/imav2011-summerediton_indoor_challenges_v3.0.pdf

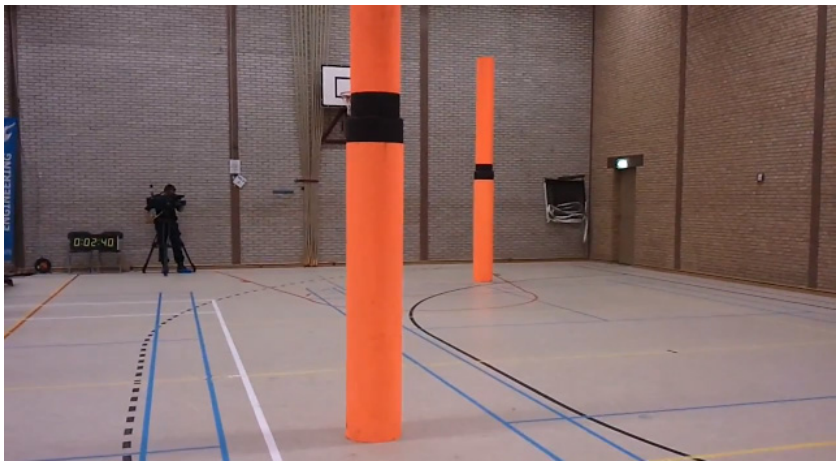


Figure 1.2: The IMAV2011 Pylon challenge environment, which is located in a sports gym. The objective is to navigate a MAV in figure-8 shapes around two poles that are $10m$ *apart. Each pole has a height of approximately* $4m$*.*

be integrated and examined. Unlike the IMAV competitions, the RoboCup competitions are not limited to MAVs.

The RoboCup Rescue league [5] is part of the RoboCup competitions. This league aims at the development of robotic technology that could help human rescuers in the aftermath of disasters like earthquakes, terroristic attacks and other extreme situations. In addition to a Rescue league with real robots, the Virtual Robots Competition was started, which uses a simulator instead of real robots. Simulation has some significant advantages above real competitions, such as the low costs and the ability to easily construct different disaster scenarios.

The main bottleneck for robot simulation is the requirement of realistic motion and sensor models. The simulation environment selected for the Virtual Robots Competition is USARSim [6, 7], which includes models of a wide range of robotic platforms and sensors. Unfortunately, the support of aerial vehicles in USARSim is little, while MAVs offer great advantages above other robots due to the versatile scouting capabilities. In 2011, the expensive AirRobot quadrotor was the only aerial vehicle available in USARSim. One of the contributions arising from this thesis is a simulation model of the affordable AR.Drone quadrotor. The resulting model can be used by teams to make use of the advantages of MAVs in disaster scenarios.

1.3 Objectives and research questions

One goal of robotics is to develop mobile robots that can operate robustly and fully autonomously in real world situations [3]. One of the main prerequisites of an autonomous robot is the ability to know its location and movement in the environment [8]. Since no assumptions can be made about the environment, the robot has to learn from its environment.

Simultaneous Localization and Mapping using aerial vehicles is an active research area in robotics. However, current approaches use algorithms that are computationally expensive and cannot be applied for real-time navigation problems. Furthermore, other researchers rely on expensive aerial vehicles with advanced sensors like a laser rangefinder. Focusing on real-time methods and affordable MAVs increases the employability of aerial vehicles in real world situations. In case of semi-autonomous robots, building and visualizing an elevated and textured map of the environment offers additional feedback to the teleoperator of the vehicle.

The main research question therefore is to determine a real-time Simultaneous Localization and Mapping approach that can be used for MAVs with a low-resolution down-pointing camera (e.g., AR.Drone). This main research question is divided into several sub-questions:

- How to construct a texture map and a feature map from camera frames?
- What is the relation between camera resolution and localization performance? Is a camera resolution of 176×144 pixels enough to localize against a map?
- What is the performance and robustness of different methods to estimate the transformation between a camera frame and a map?
- Is localization on regular basis possible for circumstances encountered during the IMAV competition?
- How to construct an elevation map with a single ultrasound sensor?

In order to perform (global) localization, a map of the environment has to be constructed. Considering the limited sensor suite of affordable quadrotors (e.g., AR.Drone), camera images are often the best information source for localization and mapping. This poses the questions how to construct a map from camera frames. The AR.Drone is equipped with a very low-resolution camera, which poses the question if localization against a map is possible with such cameras. The recent developments in miniaturizing high-resolution cameras enables MAVs to be equipped with high-resolution cameras. This leads to the question how the (increasing) image resolution will affect the localization performance. When a map is constructed, the camera frames are matched against the map to perform (global) localization. A robust transformation is computed to describe the relation between the vehicle's position and the map. Different transformation estimation methods can be used and have to be compared. The performance of the localization method is also depending on the environment. Circumstances encountered during the IMAV competition are challenging due to the repeating patterns (lines) and the lack of natural features. Therefore, the localization method is evaluated for such circumstances. Finally, an elevation map is constructed using a single airborne ultrasound sensor. Since no method has been proposed by others, the question arises how to construct this map.

1.4 Contributions

The first contribution of this thesis is a framework that aids in the development of (intelligent) applications for the AR.Drone. The framework contains an abstraction layer to abstract from the actual device, which allows to use a simulated AR.Drone in a way similar to the real AR.Drone. Therefore, a simulation model of the AR.Drone is developed in USARSim. The main contribution is a new approach that enables a MAV with a low-resolution down-looking camera to navigate in circumstances encountered during the IMAV competition. An algorithm is presented to robustly estimate the transformation between a camera frame and a map. In addition to the navigation capabilities, the approach generates a texture map, which can be used for human navigation. Another contribution is a method to construct an elevation map with an airborne ultrasound sensor.

1.5 Outline

Chapter 2 and 3 give an overview of the background theory that is part of the presented methods. This theory includes probabilistic robotics (Chapter 2) and computer vision techniques (Chapter 3). In **Chapter 4**, an overview of related research is given. This research is divided into three segments: SLAM methods for building texture maps, elevation mapping with ultrasound sensors and research conducted with the AR.Drone. An extensive overview of the AR.Drone platform is presented in **Chapter 5**. The platform overview covers the hardware, the quadrotor flight control and finally its Application Programming Interface (API). This API lacks functionalities to employ a SLAM method. Therefore, it is extended to a development framework in **Chapter 6**. In addition to this framework, a realistic simulation model of the AR.Drone is developed. This simulation model allows safe and efficient development and testing of algorithms. **Chapter 7** presents the Visual SLAM method for the AR.Drone. It includes pose estimation, real-time mapping, localization using the map and building an elevation map using a single ultrasound sensor. **Chapter 8** describes the conducted experiments and presents the results. In **Chapter 9**, this thesis concludes by answering the research questions and summarizing directions for future research.

Probabilistic Robotics 2

In Chapter 2 and 3, an introduction to background theory for this thesis is given. The work presented in this thesis relies on **probabilistic robotics** (Thrun et al [9]), which is introduced in this chapter. The work also relies on **computer vision** (Hartley and Zisserman [10]), which is introduced in Chapter 3. A clear vocabulary and corresponding mathematical nomenclature will be established that will be used consistently in the remainder of this thesis. Probabilistic robotics uses the notation of Thrun et al [9].

Sensors are limited in what they can perceive (e.g., the range and resolution is subject to physical limitations). Sensors are also subject to noise, which deviates sensor measurements in unpredictable ways. This noise limits the information that can be extracted from the sensor measurements. Robot actuators (e.g., motors) are also subject to noise, which introduces uncertainty. Another source of uncertainty is caused by the robot's software, which uses approximate models of the world. Model errors are a source of uncertainty that has often been ignored in robots. Robots are real-time systems, limiting the amount of computation that can be done. This requires the use of algorithmic approximations, but increases the amount of uncertainty even more.

For some robotic applications (e.g., assembly lines with controlled environment), uncertainty is a marginal factor. However, robots operating in uncontrolled environments (e.g., homes, other planets) will have to cope with significant uncertainty. Since robots are increasingly deployed in the open world, the issue of uncertainty has become a major challenge for designing capable robots. *Probabilistic robots* is a relatively new approach to robotics that pays attention to the uncertainty in perception and action. Uncertainty is represented explicitly using the calculus of probability theory. This means that probabilistic algorithms represent information by probability distributions over a space. This allows to represent ambiguity and degree of belief in a mathematically sound way. Now, robots can make choices (plan) with respect to the uncertainty that remains, or chose to reduce the uncertainty (e.g., explore) if that is the best choice.

2.1 Recursive state estimation

Environments are characterized by *state*. A state can be seen as a collection of all aspects of the robot and the environment that can impact the future. The state also includes variables regarding the robot itself (e.g., pose, velocity, acceleration).

Definition 1. *Environments are characterized by **state**, which describes all aspects of a robot and the environment that can impact the future. A state can be either **static state** (non-changing) or **dynamic state** where certain state variables tend to change over time. A state is denoted x. The state at time t is denoted x_t.*

Common state variables are:

- The robot *pose*, which is its location and orientation relative to a global coordinate frame;
- The *robot velocity*, consisting of a velocity for each pose variable;
- The *location and features of surrounding objects in the environment*. The objects can be either *static* or *dynamic*. For some problems, the objects are modeled as *landmarks*, which are distinct, stationary features that provide reliable recognition.

A core idea of probabilistic robotics is the estimation of the state from sensor data. Sensors observe only partial information about those quantities and their measurements are affected by noise. State estimation tries to recover state variables from the data. Instead of computing a single state, a probability distribution is computed over possible world states. This probability distribution over possible world states is called *belief* and is described on the next page.

Definition 2. ***State estimation*** *addresses the problem of estimating quantities (state variables) from sensor data that are not directly observable, but can be inferred. A probability distribution is computed over possible world states.*

A robot can **interact** with its environment by influencing the state of its environment through its actuators (*control actions*), or it can gather information about the state through its sensors (*environment sensor measurements*).

Definition 3. $z_{t_1:t_2} = z_{t_1}, z_{t_1+1}, z_{t_1+2}, \ldots, z_{t_2}$ *denotes the set of all* ***measurements*** *acquired from time t_1 to time t_2, for $t_1 \leq t_2$.*

Definition 4. $u_{t_1:t_2} = u_{t_1}, u_{t_1+1}, u_{t_1+2}, \ldots, u_{t_2}$ *denotes the sequence of all* ***control data*** *from time t_1 to time t_2, for $t_1 \leq t_2$.*

The control data conveys information regarding the change of state. For example, setting a robot's velocity at $1m/s$, suggests that the robot's new position after 2 seconds is approximately 2 meters ahead of its position before the command was executed. As a result of noise, 1.9 or 2.1 meters ahead are also likely new positions. Generally, measurements provide information about the environment's state, increasing the robot's knowledge. Motion tends to induce a loss of knowledge due to the inherent noise in robot actuation.

The evolution of state and measurements is performed by **probabilistic laws**. State x_t is generated stochastically from the state x_{t-1}. If state x is complete (i.e., best predictor of the future), it is a sufficient summary of all that happened in previous time steps. This assumption is known as the *Markov Assumption* and is expressed by the following equality:

$$p(x_t|x_{0:t-1}, z_{1:t-1}, u_{1:t}) = p(x_t|x_{t-1}, u_t) \tag{2.1}$$

where $p(x_t|x_{t-1}, u_t)$ is called *state transition probability* and specifies how environmental state evolves over time as a function of robot control u_t.

Definition 5. *The **Markov Assumption** states that past and future data are independently if one knows the current state x. This means the values in any state are only influenced by the values of the state that directly preceded it.*

The process by which measurements are modeled is expressed by:

$$p(z_t|x_{0:t}, z_{1:t-1}, u_{1:t}) = p(z_t|x_t) \tag{2.2}$$

where $p(z_t|x_t)$ is called *measurement probability* and specifies how measurements are generated from the environment state x.

Another core idea of probabilistic robotics is **belief**, which reflects the robot's knowledge about the environment's state. Belief distributions are posterior probabilities over state variables conditioned on the available data. The Markov Assumption implies the current belief is sufficient to represent the past history of the robot. The belief over state variables is expressed with $bel(x_t)$:

$$bel(x_t) = p(x_t|z_{1:t}, u_{1:t}) \tag{2.3}$$

A *prediction* of the state at time t can be made before incorporating a measurement. This prediction is used in the context of probabilistic filtering and is denoted as follows:

$$\overline{bel}(x_t) = p(x_t|z_{1:t-1}, u_{1:t}) \tag{2.4}$$

This equation predicts the state at time t based on the previous state posterior, before incorporating the measurement at time t.

Definition 6. ***Belief** reflects the robot's knowledge about the environment's state, through conditional probability distributions. Belief over state variable x_t is denoted by $bel(x_t)$.*

2.1.1 Algorithms

The most general algorithm for calculating beliefs is the **Bayes filter**. The input of the algorithm is the (initial) belief bel at time $t-1$, the most recent control u_t and measurement z_t. First, a *prediction* for the new belief is computed based on control u_t (the measurement is ignored):

$$\overline{bel}(x_t) = \int p(x_t|u_t, x_{t-1}) bel(x_{t-1}) dx_{t-1} \tag{2.5}$$

The second step of the Bayes filter is the *measurement update*:

$$bel(x_t) = \eta p(z_t|x_t) \overline{bel}(x_t) \tag{2.6}$$

Where η is a normalization constant to ensure a valid probability. Now, $bel(x_t)$ is the robot's belief about the state after the measurement and control data are used to improve the robot's knowledge. Both steps of the algorithm are performed recursively for all measurements and control commands.

The most popular family of recursive state estimators are the **Gaussian filters**. Gaussian filters represent belief by multivariate normal distributions (approximated by a Gaussian function). The density (probability) over the state space x is characterized by two parameters: the mean μ and the covariance Σ. The dimension of the covariance matrix is the dimensionality of the state x squared. Representing the posterior by a Gaussian has some important consequences. First, Gaussians are unimodel and have a single maximum. However, this is characteristic for many tracking problems, where the posterior is focused around the true state with a certain margin of uncertainty.

In order to implement the filter described above, two additional components are required. In Section 2.2, the *motion model* is described. In Section 2.3, the *measurement model* is described.

2.2 Motion model

Probabilistic motion models describe the relationship between the previous and current pose and the issued controls (commands) by a probability distribution. This is called the *state transition probability*. The state transition probability plays an essential role in the prediction step of the Bayes filter.

Definition 7. *A probabilistic **motion model** defines the state transition between two consecutive time steps $t-1$ and t after a control action u_t has been carried out. It is expressed by the posterior distribution $p(x_t|x_{t-1}, u_t)$.*

Two complementary probabilistic motion models are the *Velocity Motion Model* and the *Odometry Motion Model*. The Velocity Motion Model assumes that a robot is controlled through two velocities: a rotational velocity ω_t and translational velocity (v_t).

$$u_t = \begin{pmatrix} v_t \\ \omega_t \end{pmatrix} \tag{2.7}$$

Algorithms for computing the probability $p(x_t|x_{t-1}, u_t)$ can be found in [9]. The Odometry Motion Model assumes that one has access to odometry information (e.g., wheel encoder information). In practice, odometry models tend to be more accurate, because most robots do not execute velocity command with the level of accuracy that is obtained by measuring odometry. Technically, odometry readings are not controls because they are received after executing a command. However, using odometry readings as controls results in a simpler formulation of the estimation problem.

Both motion models are not directly applicable to MAVs. For example, quadrotor helicopters are controlled through pitch, roll and yaw (as explained in Section 5.1). A conversion between angles to velocities is required to use the Velocity Motion Model for a quadrotor helicopter. For ground robots, the odometry reading can be obtained by integrating wheel encoder information. Because a quadrotor helicopter has no contact with the ground, odometry readings are not directly available. Instead, different sources of information are used to obtain odometry information. For example, the AR.Drone uses a down-pointing camera to recover odometry information (Section 5.3.2).

2.3 Measurement model

Probabilistic measurement models describe the relationship between the world state x_t and how a sensor reading (observation) z_t is formed. Based on the estimated world state x_t, a belief over possible measurements is generated. Noise in the sensor measurements is modeled explicitly, inherent to the uncertainty of the robot's sensors. To express the process of generating measurements, a specification of the environment is required. A map m of the environment is a list of objects and their locations:

$$m = \{m_1, m_2, \ldots, m_N\} \tag{2.8}$$

Where N is the total number of objects in the environment. Maps are often *features-based maps* or *location-based maps*. Location-based maps are volumetric and offer a label for any location. Features-based maps only describe the shape of the environment at specific locations, which are commonly objects. Features-based maps are more popular because the representation makes it easier to adjust (refine) the position of objects.

Definition 8. *A probabilistic **sensor model** describes the relation between a world state x_t and a sensor reading z_t given an environmental model m in form of a posterior distribution $p(z_t|x_t, m)$.*

2.4 Localization

Robot localization is the problem of estimating the robot's pose (state) relative to a map of the environment. Localization is an important building block for successful navigation of a robot, together with *perception*, *cognition* and *motion control*.

When a robot is moving in a known environment and starting at a known location, it can keep track of its location by integrating local position estimates (e.g., odometry). Due to uncertainty of these local estimates, the uncertainty of the robot's absolute location increases over time. In order to reduce the growing uncertainty, the robot has to retrieve its absolute location by localizing itself in relation to a map. To do so, the robot uses its sensors to make observations of its environment and relates these observations to a map of the environment.

A probabilistic approach to localization uses belief to represent the estimated global location. Again, updating the robot position involves two steps. The first step is called the *prediction update*. The robot uses its state at time $t-1$ and control data u_t to predicts its state at time t. This prediction step increases the uncertainty about the robot's state. The second step is called the *perception update*. In this step, the robot uses the information from its sensors to correct the position estimated during the prediction phase, reducing the uncertainty about the robot's state.

2.5 Simultaneous Localization and Mapping

A more difficult subclass of localization occurs when a map of the environment is not available. In this case, the robot's sensor information is used to both recover the robot's path and build a map of the environment. In the robotics community, this problem is called **Simultaneous Localization and Mapping (SLAM)**. A solution to this problem would make a robot truly autonomous. SLAM is a difficult problem because both the estimated path and constructed map are affected by noise. Both of them become increasingly inaccurate during travel. However, when a place that has been mapped is revisited, the uncertainty can be reduced. This process is called *loop-closing*. Additionally, the map can be optimized after a loop-closure event is detected.

2.5.1 Solution techniques

In practice, it is not possible to compute a robot's belief (posterior probabilities) analytically. Therefore, a number of approximation techniques exist. This section describes the most popular solution techniques for SLAM problems.

Solution techniques can be divided in two major branches of belief representation: *single-hypothesis belief* and *multi-hypothesis belief*. **Single-hypothesis trackers** represent belief by a single world state. This estimate is associated by a measure of certainty (or variance), allowing the tracker to broaden or narrow the estimated region in the state space. The main advantage of the single-hypothesis representation is the absence of ambiguity, simplifying decision-making at the robot's cognitive level (e.g., pathplanning). Due to the absence of ambiguity, the trackers fail to model ambiguities adequately.

Multi-hypothesis trackers represent belief not just as a single world state, but as a possibly infinite set of states. The main advantage of the multi-hypothesis representation is that the robot can explicitly maintain uncertainty regarding its state. This allows a robot to believe in multiple poses simultaneously, allowing the robot to track and reject different hypotheses independently. One of the main disadvantages of a multi-hypothesis representation involves decision-making. If the robot represents its position as a set of points, it becomes less obvious to compute the best action.

Furthermore, solution techniques can be divided in two branches of time constraints: *online computing* and *offline computing*. **Online** techniques solve a problem in real-time and must guarantee response within strict time constraints. One advantage of online techniques is the ability to use the response as input for decision-making (e.g., navigation), which is essential for autonomous robots. Due to the strict time constraints, a limited amount of computations can be performed, which limits the complexity of solution techniques.

Offline techniques have less strict time constraints, which allow increased complexity. For example, additional refinements can be performed that would be too slow for online techniques. One of the main

disadvantages of offline techniques involves decision-making. Because a response is not available in real-time, the robot has to wait or make a decision without incorporating the response.

Kalman Filter (KF)

The most popular technique for implementing a Bayes filter is probably the Kalman Filter (1960) [11], often used for improving vehicle navigation. The filter has a Single-hypothesis belief representation and can be performed in real-time. The Kalman Filter is based on the assumption that the system is linear and both the motion model and measurement model are affected by Gaussian noise. Belief at time t is represented by a multivariate Gaussian distribution defined by its mean μ_t and covariance Σ_t. This means the x_t from the Bayes filter (Equations (2.5) and (2.6)) has been replaced by μ_t and Σ_t.

The Kalman Filter assumes the system is linear: the state transition A (Equation 2.1), the motion model B and the sensor model C (Equation 2.2) are linear functions solely depending on the state x or control command u, plus a Gaussian noise model Q:

$$
\begin{aligned}
&\textbf{Predict} \\
&\overline{\mu}_t = A_t\mu_{t-1} + B_t u_t && \text{a-priori mean estimate} \\
&\overline{\Sigma}_t = A_t\Sigma_{t-1}A_t^T + R_t && \text{a-priori covariance estimate} \\
&\textbf{Update} \\
&K_t = \overline{\Sigma}_t C_t^T (C_t \overline{\Sigma}_t C_t^T + Q_t)^{-1} && \text{Kalman gain} \\
&\mu_t = \overline{\mu}_t + K_t(z_t - C_t\overline{\mu}_t) && \text{updated (a posteriori) state estimate} \\
&\Sigma_t = (I - K_t C_t)\overline{\Sigma}_t && \text{updated (a posteriori) estimate covariance}
\end{aligned}
\tag{2.9}
$$

The Kalman Filter represents the belief $bel(x_t)$ at time t by the mean μ_t and covariance Σ_t. When a measurement is received, a new belief $bel(x_t)$ is predicted based on the previous belief $bel(x_{t-1})$, the control data u_t and both the state transition model and motion model. This predicted belief describes the most likely state $\overline{\mu}_t$ at time t and the covariance $\overline{\Sigma}_t$. The measurement is not yet included, because it is a predicted belief. The update step transforms the predicted belief ($\overline{\mu}_t$, $\overline{\Sigma}_t$) into the desired belief (μ_t, Σ_t), by incorporating the measurement z_t. The Kalman gain K_t specifies the degree to which the measurement is incorporated into the new state estimate. The key concept used for Kalman Filtering is the *innovation*, which is the difference between the actual measurement z_t and the expected measurement $C_t\overline{\mu}_t$, which is derived from the predicted state. A measurement z_t that is far off the predicted measurement, is less reliable, thus less incorporated in the new state estimate.

The standard Kalman Filter requires a linear system (i.e., a linear combination of Gaussians results in another Gaussian), which is insufficient to describe many real-life problems. Therefore, variations of the original algorithm have been proposed that can cope with different levels of non-linearity. These variations approximate the motion and sensor models in a way to make them linear again.

Extended Kalman Filter (EKF)

The Extended Kalman Filter is a variant on the Kalman Filter that can be used for non-linear systems. It tries to approximate *non-linear* motion and sensor models to make them linear. The state transition probability and the measurement probabilities are governed by the non-linear functions g and h:

$$\begin{aligned} x_t &= g(u_t, x_{t-1}) \\ z_t &= h(x_t) + \delta_t \end{aligned} \tag{2.10}$$

These functions replace the matrix operations A, B and C of the regular Kalman Filter (Equation 2.9).

They key idea of the EKF approximation is called *linearization*. A first-order linear Taylor approximation is calculated at the mean of the current belief (Gaussian), resulting in a linear function. Projecting the Gaussian belief through the linear approximation results in a Gaussian density.

Predict

$$\begin{aligned} \overline{\mu}_t &= \boldsymbol{g(u_t, \mu_{t-1})} && \text{a-priori mean estimate} \\ \overline{\Sigma}_t &= \boldsymbol{G_t}\Sigma_{t-1}\boldsymbol{G_t^T} + R_t && \text{a-priori covariance estimate} \end{aligned}$$

Update (2.11)

$$\begin{aligned} K_t &= \overline{\Sigma}_t \boldsymbol{H_t^T}(\boldsymbol{H_t}\overline{\Sigma}_t\boldsymbol{H_t^T} + Q_t)^{-1} && \text{Kalman gain} \\ \mu_t &= \overline{\mu}_t + K_t(z_t - \boldsymbol{h(\overline{\mu}_t)}) && \text{updated (a posteriori) state estimate} \\ \Sigma_t &= (I - K_t\boldsymbol{H_t})\overline{\Sigma}_t && \text{updated (a posteriori) estimate covariance} \end{aligned}$$

TORO

An example of a recent algorithm, especially designed for visual slam, is TORO [12]. As stated in Section 2.5, both the estimated path and constructed map are affected by noise. Both of them become increasingly inaccurate during travel. However, when a place that has been mapped is revisited (loop-closure), the uncertainty can be reduced and the map can be optimized to reduce its error. This optimization procedure is computationally expensive since it needs to search for a configuration that minimizes the error of the map.

A solution technique for the map optimization problem is TORO. This technique is used in a comparable setting (visual SLAM with a MAV), as described in Section 4.1. The TORO algorithm operates on a graph-based formulation of the SLAM problem, in which the poses of the robot are modeled by nodes in a graph. Constraints between poses resulting from observations or from odometry are encoded in the edges between the nodes. The goal of TORO is to find a configuration of the nodes that maximizes the observation likelihood encoded in the constraints. Gradient Descent (GD) [13] seeks for a configuration of the nodes that maximizes the likelihood of the observations by iteratively selecting an edge (constraint) $< j, i >$ and by moving a set of nodes in order to decrease the error introduced by the selected constraint.

The TORO algorithm uses a tree-based parameterization for describing the configuration of the nodes in the graph. Such a tree can be constructed from trajectory of the robot. The first node is the root of the tree. An unique id is assigned to each node based on the timestamps of the observations. Furthermore, the parent of a node is the node with the smallest id (timestamp) and a constraint between both nodes. Each node i in the tree is related to a pose p_i in the network and maintains a parameter x_i. The parameter x_i is a 6D vector and describes the relative movement (rotation and translation) from the parent of node i to node i itself. The pose of a node can be expressed as:

$$P_i = \prod_{k \in \mathcal{P}_{i,0}} X_k \tag{2.12}$$

Where $\mathcal{P}_{i,0}$ is the ordered list of nodes describing a path in the tree from the root to node i. The homogenous transformation matrix X_i consists of a rotational matrix R and a translational component t.

Distributing an error ϵ over a sequence of n nodes in the two-dimensional space can be done in a straightforward manner. For example, by changing the pose of the i-th node in the chain by $\frac{i}{n} \times r^{2D}$. In the three-dimensional space, such a technique is not applicable. The reason for that is the non-commutativity of the three rotations. The goal of the update rule in GD is to iteratively update the configuration of a set of nodes in order to reduce the error introduced by a constraint.

The error introduced by a constraint is computed as follows:

$$E_{ji} = \Delta_{ji}^{-1} P_i^{-1} P_j \tag{2.13}$$

In TORO's approach, the error reduction is done in two steps. First, it updates the rotational components R_k of the variables x_k and second, it updates the translational components t_k. The orientation of pose p_j is described by:

$$\mathcal{R}_1 \mathcal{R}_2 \dots \mathcal{R}_n = \mathcal{R}_{1:n} \tag{2.14}$$

where n is the length of the path $\mathcal{P}_{ji}$. The error can be distributed by determining a set of increments in the intermediate rotations of the chain so that the orientation of the last node j is $\mathcal{R}_{1:n}B$. Here B is a matrix that rotates x_j to the desired orientation based on the error. Matrix B can be decomposed into a set of incremental rotations $B = B_{1:n}$. The individual matrices B_k are computed using a spherical linear interpolation (slerp) [14].

The translational error is distributed by linearly moving the individual nodes along the path by a fraction of the error. This fraction depends on the uncertainty of the individual constraints (encoded in the corresponding covariance matrices).

Despite TORO is presented as an efficient estimation method, it cannot solve realistic problems in real-time [12], which makes it an offline solution technique. All work presented in this thesis has a strong emphasis on online techniques, which allows the work to be used for navigation tasks like the IMAV Pylon challenge (Section 1.1). For this reason, a TORO-based optimization algorithm was not used in this thesis.

Nevertheless, the TORO algorithm is described here, because it is being used by Steder's work (as described in Section 4.1). Furthermore, a single-hypothesis belief representation was chosen for its simplicity.

The theory described in this chapter is used in the work presented in Section 7.

Computer Vision 3

In this chapter, the introduction to required background theory for this thesis is continued. This chapter introduces **computer vision** (Hartley and Zisserman [10]). A clear vocabulary and corresponding mathematical nomenclature will be established that will be used consistently in the remainder of this thesis. Computer vision uses the notation of Hartley and Zisserman [10].

Laser range scanners are often used for robot localization. However, this sensor has a number of problems (e.g., limited range, deal with dynamic environments). Vision has long been advertised as a solution. A camera can make a robot perceive the world in a way similar to humans. **Computer vision** is a field that includes methods for acquiring, processing, analyzing and understanding images. Since a camera is the AR.Drone's only sensor capable of retrieving detailed information of the environment, computer vision techniques are essential when performing robot localization with the AR.Drone.

3.1 Projective geometry

Projective geometry describes the relation between a scene and how a picture of it is formed. A projective transformation is used to map objects into a flat picture. This transformation preserves straightness. Multiple geometrical systems can be used to describe geometry.

Euclidean geometry is a popular mathematical system. A point in Euclidean 3D space is represented by an unordered pair of real numbers, (x, y, z). However, this coordinate system is unable to represent vanishing points. The *Homogeneous coordinate system* is able to represent points at infinity. Homogeneous coordinates have an additional coordinate. Now, points are represented by *equivalance classes*, where points are equivalent when they differ by a common multiple. For example, $(x, y, z, 1)$ and $(2x, 2y, 2z, 2)$ represent the same point. Points at infinity are represented using $(x, y, z, 0)$, because $(x/0, y/0, z/0)$ is infinite. In Euclidean geometry, one point is picked out as the *origin*. A coordinate can be transformed to another coordinate by translating and rotating to a different position. This operation is known as a **Euclidian transformation**. A more general type of transformation is the **affine transformation**, which is a linear transformation (linear stretching), followed by a Euclidian transformation. The resulting transformation performs moving (translating), rotating and stretching linearly.

In computer vision problems, projective space is used as a convenient way of representing the real 3D world. A **projective transformation** of projective space $\mathbb{P}^n$ is represented by a linear transformation of homogeneous coordinates:

$$X' = H_{(n+1)} \times_{(n+1)} X \tag{3.1}$$

where X is a coordinate vector, H is a non-singular matrix and X' is a vector with the transformed coordinates.

In order to analyze and manipulate images, understanding about the image formation process is required. Light from a scene reaches the camera and passes through a lens before it reaches the sensor. The relationship between the distance to an object z and the distance behind the lens e at which a focused image is formed, can be expressed as:

$$\frac{1}{f} = \frac{1}{z} + \frac{1}{e} \tag{3.2}$$

where f is the focal length.

The **pinhole camera** has been the first known example of a camera [15]. It is a lightproof box with a very small aperture instead of a lens. Light from the scene passes through the aperture and projects an *inverted image* on the opposite side of the box. This principle has been adopted as a standard model for perspective cameras. In this model, the *optical center* C (aperture) corresponds to the center of the lens. The intersection O between the optical axis and the image plane is called *principal point*. The pinhole model is commonly described with the image plane between C and the scene in order to preserve the same orientation as the object.

The operation performed by the camera is called the **perspective transformation**. It transforms the 3D coordinate of scene point P to coordinate p on the image plane. A simplified version of the perspective transformation can be expressed as:

$$\frac{f}{P_z} = \frac{p_x}{P_x} = \frac{p_y}{P_y} \tag{3.3}$$

from which p_x and p_y can be recovered:

$$p_x = \frac{f}{P_z} \cdot P_x \tag{3.4}$$

$$p_y = \frac{f}{P_z} \cdot P_y \tag{3.5}$$

When using homogeneous coordinates, a linear transformation is obtained:

$$\begin{bmatrix} \lambda p_x \\ \lambda p_y \\ \lambda \end{bmatrix} = \begin{bmatrix} fP_x \\ fP_y \\ P_z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \tag{3.6}$$

From this equation can be seen that it is not possible to estimate the distance to a point (i.e., all points on a line project to a single point on the image plane).

A more realistic camera model is called the **general camera model**. It takes into account the rigid body transformation between the camera and the scene, and *pixelization* (i.e., shape and position of the camera's sensor). Pixelization is addressed by replacing the focal length matrix with a *camera intrinsic parameter matrix* A. The rigid body transformation is addressed by adding a combined rotation and translation matrix $[R|t]$, which are called the *camera extrinsic parameters*.

$$\lambda p = A[R|t]P \tag{3.7}$$

or

$$\begin{bmatrix} \lambda p_x \\ \lambda p_y \\ \lambda \end{bmatrix} = \begin{bmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \tag{3.8}$$

Camera calibration is the process of measuring the intrinsic and extrinsic parameters of the camera. As explained in the previous paragraph, these parameters are required to calculate the mapping between 3D scene points to 2D points on the image plane. When scene points P and image points p are known, it is possible to compute A, R, t by solving the perspective projection equation. Newer camera calibration techniques use a planar grid (e.g., chessboard-like pattern) instead of 3D points, to ease the extraction of corners. The method requires several pictures of the pattern shown at different positions and orientations. Because the 2D positions of the corners are known and matched between all images, the intrinsic and extrinsic parameters are determined simultaneously by applying a least-square minimization.

3.2 Feature extraction and matching

In order to perform robot localization using vision sensors, the robot must be able to relate recent camera frames to previously received camera frames (map). However, all measurements have an error, which complicates the matching. A strategy to deal with uncertainty is by generating a higher-level description of the sensor data instead of using the raw sensor data. This process is called *feature extraction*.

Definition 9. ***Features*** *are recognizable structures of elements in the environment. They are extracted from measurements and described in a mathematical way. Low-level features are geometric primitives (e.g., lines, points, corners) and high-level features are objects (e.g., doors, tables).*

Features play an essential role in the software of mobile robots. They enable more compact and robust descriptions of the environment. Choosing the appropriate features is a critical task when designing a mobile robot.

3.2.1 Feature extraction

A local feature is a small image patch that is different than its immediate neighborhood. Difference can be in terms of intensity, color and texture. Commonly, features are edges, corners or junctions. The most important aspect of a feature detector is *repeatability*, which means the detector is able to detect the same features when multiple images (with different viewing and illumination conditions) of the same scene are used. Another important aspect of a feature detector is *distinctiveness*, which means the information carried by a patch is distinctive as possible. This is important for robust feature matching. A good feature is

invariant, meaning that changes in camera viewpoint, illumination, rotation and scale do not affect a feature and its high-level description.

Scale-invariant feature transform (SIFT) [16] is probably the most popular feature extractor. It is invariant to scale, rotation, illumination and viewpoint. The algorithm can be outlined as follows. First, an internal representation of an image is computed to ensure scale invariance. Secondly, an approximation of the Laplacian of Gaussian is used to find interesting keypoints. Third, keypoints are found at maxima and minima in the Difference of Gaussian from the previous step. Fourth, bad keypoints (e.g., edges and low contrast regions) are eliminated. Fifth, an orientation is calculated for each keypoint. Any further calculations are done relative to this orientation, making it rotation invariant. Finally, a high-level representation is calculated to uniquely identify features. This *feature descriptor* is a vector of 128 elements, containing orientation histograms.

For some applications or mobile platforms, the SIFT extractor is too slow. Therefore, faster variants are proposed, like **Speeded Up Robust Feature (SURF)** [17]. The standard version of SURF is several times faster than SIFT and claimed by its authors to be more robust against different image transformations than SIFT. The important speed gain is achieved by using integral images, which drastically reduces the number of operations for simple box convolutions, independent of the chosen scale.

The algorithm can be outlined as follows. Initially, an integral image and Hessian matrix [18] are computed. The Hessian matrix is a square matrix of second-order partial derivatives of a function, describing the local curvature of a function. Blob-like structures are detected at locations where the determinant matrix is maximum. The images are repeatedly smoothed with a Gaussian kernel and then sub-sampled in order to create a pyramid of different resolutions. In order to find interest points in the image and over scales, a non-maximum suppression in a neighborhood is applied. The maxima of the determinant of the Hessian matrix are then interpolated in scale and image space. A descriptor is constructed that describes the distribution of the intensity content within the interest point neighborhood. First-order Haar wavelet [19] responses in x and y direction are used rather than the gradient, exploiting integral images for speed. The size of the descriptor is reduced to 64 dimensions.

Another feature extractor is the **GIST** descriptor [20], which characterizes several important statistics about a scene. The GIST descriptor measures the global distribution of oriented line segments in an image. This makes the descriptor well suited for determining the type of environment. The GIST feature is computed by convolving an oriented filter with the image at several different orientations and scales. The scores of the convolution at each orientation and scale are stored in array.

3.2.2 Feature matching

To perform robot localization, extracted feature descriptors need to be matched against the feature descriptors that are part of the map. Ideally, each feature extracted from the current frame is correlated against the

corresponding feature inside the map.

A *matching function* is used to compute the correspondence between two feature descriptors. The matching function depends on the type of feature descriptor. In general, a feature descriptor is a high-dimensional vector and matching features can be found by computing the distance using the L_2 norm, which is defined as:

$$L2(a,b) = \sqrt{\sum_{i=1}^{N} |a_i - b_i|^2} \tag{3.9}$$

where N is the dimensionality of the descriptor vector.

The matching function is used to find matches between two sets of descriptors. A popular matcher is the *Brute-force descriptor matcher*. For each descriptor in the first set, this matcher finds the closest descriptor in the second set by trying each one. However, the brute-force matcher becomes slow for large descriptors sets, due to the complexity of $O(n^2)$. For large descriptor sets, the *Fast Library for Approximate Nearest Neighbors (FLANN)* [21] matcher performs more efficient matching. It uses an Approximate Nearest Neighbors algorithm to find similar descriptors efficiently. The authors state that the approximation has proven to be good-enough in most practical applications. In most cases it is orders of magnitude faster than performing exact searches.

In this thesis, the Brute-force descriptor matcher is used. The estimated pose of the vehicle is used to select a part of the map (descriptor set), which reduces the size of the descriptor set. Furthermore, the resolution of the map (descriptor set) is reduced by keeping only the best feature for a certain area inside the map. For such limited descriptor sets, the Brute-force descriptor matcher performs quite well.

3.2.3 Popular application: image stitching

A popular computer vision application that heavily relies on features is *image stitching*, which is the process of combining multiple photos with overlapping fields of view. The result is a segmented panorama or high-resolution image.

The first step involves extracting features for all images. Since most features are invariant under rotation and scale changes, images with varying orientation and zoom can be used for stitching. Images that have a large number of matches between them are identified. These images are likely to have some degree of overlap. A homography $\boldsymbol{H}$ (projective transformation) is computed between matching image pairs. This homography describes how the second image needs to be transformed in order to correctly overlap the first image. Different types of homographies can be used to describe the transformation between image pairs, depending on the level of displacement between images.

The homography $\boldsymbol{H}$ cannot be computed from all feature matches directly. False matches would reduce the accuracy of the estimated homography. Instead, a robust estimation procedure is applied. **RANSAC (Random Sample Consensus)** [22] is a robust estimation procedure that uses a minimal set of randomly

sampled correspondences to estimate image transformation parameters, finding a solution that has the best consensus with the data. Commonly, four random feature correspondences are used to compute the homography $\boldsymbol{H}$ of an image pair. This step is repeated N trials and the solution that has the maximum number of inliers (whose projections are consistent with $\boldsymbol{H}$ within a tolerance ϵ pixels) is selected.

When the homographies of all image pairs are estimated, a global optimization method can be applied to reduce drift or other sources of error. Finally, the images are transformed according to the corresponding homographies, resulting in a seamless panorama or high-resolution image.

3.3 Visual Odometry

An interesting combination of computer vision and mobile robotics is *Visual Odometry (VO)*, which is the process of estimating the motion of a mobile robot using one or more onboard cameras. Below, the case of a single camera and 2D feature coordinates is described. There are two main approaches to compute the relative motion from video images: *appearance-based* methods and *feature-based* methods. Appearance-based methods use intensity information, while feature-based methods use higher-level features (Section 3.2). In this thesis a feature-based approach was chosen, because high-level features are used for localization against a map. Therefore, parts of the localization code can be reused for a Visual Odometry algorithm.

The set of images taken at time t is denoted by $I_{0:n} = \{I_0, \ldots, I_n\}$. Two camera positions at consecutive time instants $t-1$ and t are related by a rigid body transformation:

$$T_t = T_{t,t-1} = \begin{bmatrix} R_{t,t-1} & t_{t,t-1} \\ 0 & 1 \end{bmatrix} \tag{3.10}$$

where $R_{t,t-1}$ is the rotation matrix and $t_{t,t-1}$ is the translation matrix. The set $T_{1:n} = \{T_1, \ldots, T_n\}$ contains all subsequent motions. The set of camera poses $C_{0:n} = \{C_0, \ldots, C_n\}$ contains the transformation of the camera with respect to the initial coordinate frame at time $t = 0$.

The objective of Visual Odometry is to recover the full trajectory of the camera $C_{0:n}$. This is achieved by computing the relative transformation T_t from images I_t and I_{t-1}. When all relative transformation are concatenated, the full trajectory of the camera $C_{0:n}$ is recovered. An iterative refinement procedure like *windowed-bundle adjustment* [23] can be performed to obtain a more accurate estimate of the local trajectory.

The first step of VO is the extraction of image features f_t from a new image I_t. In the second step, these features are matched with features f_{t-1} from the previous image (multiple images can be used to recover 3D motion). The third step consists of computing the relative motion T_t between time $t-1$ and t. For accurate motion estimation, feature correspondences should not contain outliers, as described in Section 3.2.2.

The geometric relations between two images I_t and I_{t-1} are described by the *essential matrix* E:

$$E_t \simeq \hat{t}_t R_t \tag{3.11}$$

where R_t is the rotation and $\hat{t}_t$ is the translation matrix:

$$\hat{t}_t = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \tag{3.12}$$

The rotation and translation of the camera can be extracted from E. A solution when at least eight feature correspondences are available, is the Longuet-Higgins's eight-point algorithm [24]. Each feature correspondence gives a constraint of the following form:

$$\begin{bmatrix} \tilde{u}\tilde{u}' & \tilde{u}'\tilde{v} & \tilde{u}' & \tilde{u}\tilde{v}' & \tilde{v}\tilde{v}' & \tilde{v}' & \tilde{u}' & \tilde{v}' & 1 \end{bmatrix} E = 0 \tag{3.13}$$

where $E = [e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9]^T$, $\tilde{u}$ and $\tilde{v}$ are the x and y coordinates in the first image and $\tilde{u}'$ and $\tilde{v}'$ are the x and y coordinates in the second image.

Stacking these constraints gives the linear equation $AE = 0$, where A is the camera intrinsic matrix. This equation can be solved by using singular value decomposition (SVD) [25], which has the form $A = USV^T$. Now, the projected essential matrix $\overline{E}$ can be found as the last column of V:

$$\overline{E} = Udiag\{1, 1, 0\}V^T \tag{3.14}$$

The rotation and translation can be extracted from $\overline{E}$:

$$R = U(\pm W^T)V^T \tag{3.15}$$

$$\hat{t} = U(\pm W)SU^T \tag{3.16}$$

where

$$W^T = \begin{bmatrix} 0 & \pm 1 & 0 \\ \mp 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.17}$$

Now, the camera position at time t can be recovered by concatenating the relative transformations:

$$C_t = \prod_{k=t}^{1} T_k \tag{3.18}$$

The theory described in this chapter is used in the work presented in Section 7. Projective geometry is used in Section 7.2 to warp the camera images on a map. Furthermore, feature extraction is used to build a map. Future matching is used in Section 7.3 to perform localization against a map. Section 7.4 uses Visual Odometry to recover the motion of the AR.Drone.

4 Related research

One of the most fundamental problems in robotics is the Simultaneous Localization and Mapping (SLAM) problem. This problem arises when the robot does not have access to a map of the environment and does not know its own pose. This knowledge is critical for robots to operate autonomously. SLAM is an active research area in robotics. A variety of solutions have been developed. Most solutions rely on large and heavy sensors that have a high range and accuracy (e.g., SICK laser rangefinder). However, these sensors cannot be used on small (flying) vehicles. As a result, researchers focused on using vision sensors, which offer a good balance in terms of weight, accuracy and power consumption. Lightweight cameras are especially attractive for small flying vehicles (MAVs).

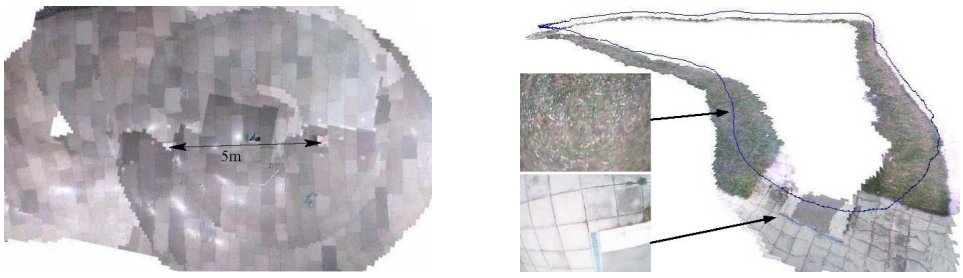
Almost all publications related to this research describe a methodology that is able to learn the environment, but do not produce a visual map. An exception is the research by Steder, which builds a visual map of the environment.

4.1 Visual-map SLAM

Visual SLAM for flying vehicles

The approach presented in this thesis is inspired by Steder et al. [26], who presented a system to learn large visual maps of the ground using flying vehicles. The setup used is comparable to the AR.Drone, with an inertial sensor and a low-quality camera pointing downward. If a stereo camera setup is available, their system is able to learn visual elevation maps of the ground. If only one camera is carried by the vehicle, the system provides a visual map without elevation information. Steder uses a graph-based formulation of the SLAM problem, in which the poses of the vehicle are described by the nodes of a graph. Every time a new image is acquired, the current pose of the camera is computed based on both visual odometry and place revisiting. The corresponding node is augmented to the graph. Edges between these nodes represent spatial constraints between them. The constructed graph serves as input to a TORO-based network optimizer (Section 2.5.1), which minimizes the error introduced by the constraints. The graph is optimized if the computed poses are contradictory.

Each node models a 6 degree of freedom camera pose. The spatial constraints between two poses are computed from the camera images and the inertia measurements. To do so, visual features are extracted from the images obtained from down-looking cameras. Steder uses Speeded-Up Robust Features (SURF) (Section 3.2.1) that are invariant with respect to rotation and scale. By matching features in current image to the ones stored in the previous n nodes, one can estimate the relative motion of the camera (Section 3.3). The inertia sensor provides the roll and pitch angle of the camera. This reduces the dimensionality of each



(a) Map constructed by a blimp (non-rigid airship)

(b) Map constucted by a stereo camera platform. A sensor platform was mounted on a rod to simulate a freely floating vehicle.

Figure 4.1: Visual maps obtained with the visual SLAM system presented in [26] (Courtesy Steder et al. [26]).

pose that needs to be estimated from $\mathbb{R}^6$ to $\mathbb{R}^4$. For each node, the observed features as well as their 3D positions relative to the node are stored. The constraints between nodes are computed from the features associated with the nodes.

In the case of place revisiting, they compare the features of the current frame with all previous features. To speed up this potentially expensive operation, multiple filters are used. Firstly, only the features from robot poses that lie within Tipaldi's confidence interval [27] are used. Secondly, only the best features from the current image are used (i.e., features with the lowest descriptor distance during visual odmetry). Finally, a k-D tree is used to efficiently query for similar features, together with the best-bins-first technique proposed by Lowe [16].

The camera pose is computed as follows. Using known camera calibration parameters, the positions of the features are projected on a normalized image plane. Now, the altitude of the camera is computed by exploiting the similarity of triangles. Once the altitude is known, the yaw (Section 5.1) of the camera is computed by projecting map features into the same normalized image plane. When matching two features from the camera image against two features from the map, the yaw is the angle between the two lines on this plane. Finally, the feature positions from the camera image are projected into the map according to the known altitude and yaw angle. The x and y coordinates are determined as the difference between the positions of the map features and the projections of the corresponding image points.

Both visual odometry and place revisiting return a set of correspondences, from which the the most likely camera transformation is computed. First, these correspondences are ordered according to the Euclidean distance of their descriptor vectors, such that the best correspondences are used first. The transformation $T_{c_a,c_b}$is determined for each correspondence pair. This transformation is then evaluated based on the other features in both sets using a score function. The score function calculates the relative displacement between the image features and the map features projected into the current camera image. The solution with the highest score is used as estimated transformation.

Feature correspondences between images are selected using a deterministic PROSAC [28] algorithm. PROSAC takes into account a quality measure (e.g., distance between feature descriptors) of the correspondences during sampling, where RANSAC draws the samples uniformly.

Steder's article describes how to estimate the motion of a vehicle and perform place revisiting using a feature map. However, this article and Steder's other publications do not describe how the visual map is constructed and visualized. Furthermore, it lacks how an elevation map is constructed and how this elevation map can be used to improve the position estimates (e.g., elevation constraints). A great disadvantage of the described method is the computational cost of the optimization method, which cannot be performed online (i.e. during flight). This reduces the number of practical applications.

Online Mosaicking

While Steder uses a Euclidean distance measure to compare the correspondences of the features in two images, Caballero et al. [29] indicate that this is a last resort. They are able to make a robust estimation of the spatial relationship on different levels: homogeneous, affine and Euclidean (as described in Section 3.1). The article addresses the problem for aerial vehicles with a single camera, like the AR.Drone. A mosaic is built by aligning a set of images gathered to a common frame, while the aerial vehicle is moving. A set of matches between two views can be used to estimate a homographic model for the apparent image motion.

Figure 4.2: Mosaic constructed with a KARMA autonomous airship flying at $22m$ altitude (Courtesy Caballero et al. [29]).

The homography that relates two given images is computed from sets of matched features. Depending on the scene characteristics (e.g., the parallax effect and small overlap between images), the homography computation could become a difficult problem. A classical solution to improve the results is to introduce additional constraints to reduce the number of degrees of freedom of the system of equations. This is accomplished through a hierarchy of homographic models, in which the complexity of the model to be fitted is decreased whenever the system of equations is ill-constrained. An estimation of this accuracy will

be given by the covariance matrix of the computed parameters.

If most of the matches are tracked successfully, a *complete homogeneous transformation* (8 degrees of freedom) is estimated. Least median of squares (LMedS) is used for outlier rejection and a M-Estimator [30] to compute the final result. When the number of correspondences is too low, an *affine transformation* (6 degrees of freedom) is estimated. LMedS is not used, given the reduction in the number of matches. Instead, a relaxed M-Estimator (soft penalization) is carried out to compute the model. Only when the features in the image are really noisy a *Euclidean transformation* (4 degrees of freedom) is estimated. The model is computed using least-squares. If the selected hierarchy level is not constrained enough (e.g., the M-Estimator diverges by reaching the maximum number of iterations), the algorithm decreases the model complexity. Once the homography is computed, it is necessary to obtain a measure of the estimation accuracy. Caballero uses a 9×9 covariance matrix of the homography matrix, which is explained in [10].

The motion of the vehicle is computed using the estimated homographies. This method assumes that the terrain is approximately flat and that cameras are calibrated. H_{12} is the homography that relates the first and the second view of the planar scene. Both projections can be related to the camera motion as:

$$H_{12} = AR_{12}(I - \frac{t_2 n_i^T}{d_1})A^{-1} \tag{4.1}$$

where A is the camera calibration matrix, t_2 is the relative translation of the second view, n_1 is an unitary vector normal to the plane in the first camera coordinate frame, d_1 is the distance from the first camera to the plane and R_{12} is the rotation matrix that transforms a vector in the first camera coordinate frame into a vector expressed in the second camera coordinate frame. If the camera calibration matrix A and the distance to the ground d_1 are known, it is possible to extract the motion of the vehicle. Triggs [31] proposes a robust algorithm based on the singular value decomposition of the calibrated homography, defined by:

$$H_{12}^u = A^{-1} H_{12} A \tag{4.2}$$

A classical static mosaic is not appropriate, as the uncertainties should be taken into account along the mosaicking process. Therefore, the mosaic is augmented by including stochastic information, resulting in a set of images linked by stochastic relations.

The position of the image inside the mosaic is obtained by multiplying the current homography by all the previous homographies. This estimation will drift along time, but can be compensated by building a mosaic. By comparing the current image with images previously stored in the mosaic, loop-closures can be detected and used to eliminate the accumulated drift. The crossover detection consists of finding one image whose Mahalanobis distance is within a certain empirical range. Once an image is detected, a feature matching is used to compute the alignment between both images. In the general case, the task of matching images taken from very different viewpoints is difficult and computationally costly. For this reason, the image is warped to match the image from the mosaic, which simplifies the problem. The computed ho-

mography is used to obtain the correct alignment. Now, the relations among the most recent image and the corresponding image from the mosaic can be updated.

The relations between images are maintained and updated by using an Extended Kalman Filter (Section 2.5.1). The state vector including the mean and covariance of all images is defined as:

$$x^- = [x_1, x_2, ..., x_n]^T = [h_{01}, x_1 \cdot h_{12}, ..., x_{(n-1)n} \cdot h_{(n-1)n}]^T \tag{4.3}$$

The correction obtained when a loop-closing is detected is used to update the state. All states affected by the correction are now updated by the Extended Kalman Filter.

Similar to Steder's research, Caballero uses a probabilistic representation of the estimated poses. Both approaches optimize the map after a loop-closure event is detected, but use different optimization methods. Unfortunately, these optimization methods are computationally expensive and difficult to perform while flying. Steder exploits the information from the inertia sensor to reduce the complexity of the pose estimation problem, without decreasing the degree of freedom of the estimated pose.

4.2 Airborne elevation mapping with an ultrasound sensor

Multiple techniques have been presented that successfully employ an airborne radar sensor for building elevation maps (e.g., [32, 33]). Due to its large footprint, a radar sensor cannot be mounted on small UAVs. Unlike a radar sensor, an ultrasound sensor can be carried by a small UAV. This sensor provides feedback for altitude stabilization and obstacle detection. According to my knowledge, no publication addresses the problem of elevation mapping using a single airborne ultrasound sensor. However, the problem is addressed using advanced (e.g., multibeam) ultrasound sensors. These advanced ultrasound sensors are commonly used for obstacle detection and navigation by underwater robots. Underwater robots observe the seabed from above, very similar to the way airborne vehicles observe the floor from above. This similarity suggests that methods developed for underwater mapping can be used for airborne mapping.

Seabed mapping

Seabed mapping is a research area in which ultrasound sensors have been used for building elevation maps. Some examples are [34, 35, 36, 37]. Kenny et al. [38] give an overview of seabed-mapping technologies. Elevation mapping is essential for an autonomous underwater vehicle designed to navigate close to the seabed. The map is being used for obstacle avoidance, path planning and localization.

Given the limited range and applicability of visual imaging systems in an underwater environment, sonar has been the preferred solution [39] for observing the seabed. Most approaches use a side scan sonar system that returns an image instead of a single distance measurements. Side scan uses a sonar device that emits conical or fan-shaped pulses down toward the seabed. The intensity of the acoustic reflections are

recorded in multiple transducers. When stitched together along the direction of motion, these slices form an image of the sea bottom within the coverage of the beam. Modern side scan devices offer high-resolution images of the seabed on which objects of at least $10cm$ may be detected at a range of up to $100m$.

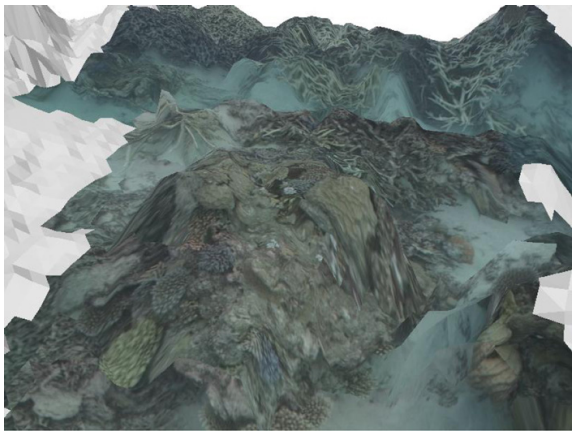


Figure 4.3: Seabed terrain model built by projecting the texture of the visual images onto a surface model generated by the sonar data. The grey areas represent portions of the terrain that are only observed by the sonar. (Courtesy Williams and Mahon [40]).

A particularly interesting application of sonar-based elevation mapping is presented in [40]. The article presents a SLAM system that uses information from the on-board sonar and vision system of an Unmanned Underwater Vehicle (UUV). The system estimates the vehicle's motion and generates a model of the structure of the underlying reefs. Unlike the AR.Drone, the pencil beam scanning sonar of the UUV returns high-resolution images instead of a single distance measurement.

In cases where a priori maps are not available, a mechanism must exist for building a map while simultaneously using that map for localization. During motion, the vehicle builds a complete map of landmarks and uses these to provide continuous estimates of the vehicle location. By tracking the relative position between the vehicle and identifiable features in the environment, both the position of the vehicle and the position of the features can be estimated simultaneously. An Extended Kalman Filter is used for stochastic estimation of the vehicle's position and map feature locations. The state is represented by an augmented state vector $\hat{x}^+(t)$, consisting of n_v states.

$$\hat{x}^+ = \begin{bmatrix} \hat{x}_0^+ \\ \vdots \\ \hat{x}_{n_v}^+ \end{bmatrix} \tag{4.4}$$

Each state $\hat{x}_t^+$ consists of a vehicle state $\hat{x}_v$ and the observed features $\hat{x}_{f,i}, i = 0, ..., n_f$, where n_f is the

number of features.

$$\hat{x}_t^+ = \begin{bmatrix} \hat{x}_v \\ \hat{x}_{f,0} \\ \vdots \\ \hat{x}_{f,n_f} \end{bmatrix} \tag{4.5}$$

The covariance matrix for the state at time t is defined as:

$$\mathbf{P}_t^+ = E[(x_t - \hat{x}_t^+)(x_t - \hat{x}_t^+)^T | \mathbf{Z}^k] \tag{4.6}$$

It was assumed that the roll and pitch of the vehicle were negligible, based on observations of the vehicle performance during in-water tests and results from the design of the vehicle. The vehicle pose at time step t is represented by

$$\hat{x}_v^+(t) = \begin{bmatrix} \hat{x}_v^+(t) & \hat{y}_v^+(t) & \hat{z}_v^+(t) & \hat{\psi}_v^+(t) \end{bmatrix}^T \tag{4.7}$$

where $\hat{x}_v^+(t)$, $\hat{y}_v^+(t)$ and $\hat{z}_v^+(t)$ are the x, y and z coordinates of the vehicle at time t and $\hat{\psi}_v^+(t)$ is the vehicle's yaw at time t. The features states are represented by

$$\hat{x}_i^+(t) = \begin{bmatrix} \hat{x}_i^+(t) & \hat{y}_i^+(t) & \hat{z}_i^+(t) \end{bmatrix}^T \tag{4.8}$$

where $\hat{x}_i^+(t)$, $\hat{y}_i^+(t)$ and $\hat{z}_i^+(t)$ are the x, y and z coordinates of a feature.

Ultrasound and vision information can be combined to aid in the identification and classification of natural features present in the environment. For maximum overlap of both sensors the ultrasound sensor is mounted directly above the high-resolution vision system. The ultrasound sensor scans the seabed directly below the vehicle and is used to generate profiles of the surface. The distance measurements received by the ultrasound sensor are then processed to identify distinctive shapes which are used to initialize new features in the SLAM map. The observation of range R and bearing θ are combined with the estimated value of the vehicle pose $\hat{x}_v(t)$ and the measured position $\begin{bmatrix} x_s & y_s & z_s \end{bmatrix}^T$ of the sonar relative to the camera frame.

$$\begin{bmatrix} \hat{x}_i \\ \hat{y}_i \\ \hat{z}_i \end{bmatrix} = \begin{bmatrix} \hat{x}_v + x_s cos\hat{\psi}_v - (y_s + Rcos\theta) sin\hat{\psi}_v \\ \hat{y}_v + x_s sin\hat{\psi}_v + (y_s + Rcos\theta) cos\hat{\psi}_v \\ \hat{z} + z_s + Rsin\theta \end{bmatrix} \tag{4.9}$$

Once the ultrasound features have been initialized, they are tracked using observations from the vision system. When a new feature is initialized, the sonar footprint is projected into the visual frame of the camera. The center of this footprint is then used to identify a high contrast feature in the image within the area covered by the sonar. These visual features are tracked from frame to frame using the Lucas and Kanade feature tracking technique [41]. Observation of elevation z_e and azimuth z_a (orientation) are provided to the SLAM algorithm. The estimated elevation and azimuth are computed using the current

estimate of the vehicle pose and feature position:

$$\begin{bmatrix} \hat{z}_e \\ \hat{z}_a \end{bmatrix} = \begin{bmatrix} (\hat{x}_i - \hat{x}_v)cos\hat{\psi}_v + (\hat{y}_i - \hat{y}_v)sin\hat{\psi}_v \\ -(\hat{x}_i - \hat{x}_v)sin\hat{\psi}_v + (\hat{y}_i - \hat{y}_v)cos\hat{\psi}_v \end{bmatrix} \tag{4.10}$$

The difference between the actual observation received from the camera and the predicted observation is the innovation, which is used in computing the posterior state estimate. The current implementation makes use of very simple features. These features are sufficient for tracking purposes, but are not applicable to detect loop-closures.

4.3 Research based on the AR.Drone

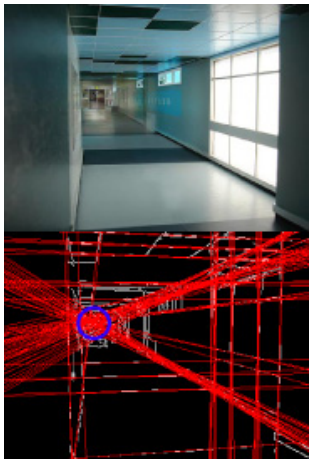
Because the AR.Drone is a quite recent development, the number of studies based on this platform is limited.

Autonomous corridor and staircase flights

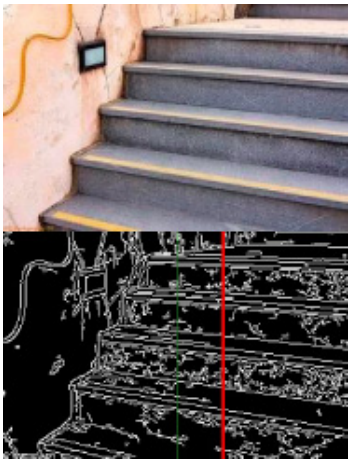
A recent publication is from Cornell University [42], where an AR.Drone is used to automatically navigate corridors and staircases based on visual clues. Their method classifies the type of indoor environment (e.g., corridors, staircases, rooms and corners) and then uses vision algorithms based on perspective clues to estimate the desired direction to fly. This method requires little computational power and can be used directly without building a 3D model of the environment.

The type of environment is determined using two algorithms. The first method uses GIST features [20] (Section 3.2.1) with a Support Vector Machine (SVM) learning algorithm for classification. GIST features are well suited for this task because they directly measure the global distribution of oriented line segments in an image, which takes advantage of the long lines found in indoor images. The second method computes a confidence estimate from both the staircase and corridor vision algorithms. The confidence values are compared and the highest is used to select the environment type.

When navigating corridors, parallel lines appear to converge. The points where lines appear to converge are called vanishing points. Vanishing points are used to locate the end of the corridor. The Canny edge detector [43] is used to detected edges and a probabilistic Hough transform [44] is used to find lines. The vanishing points (end of the corridor) are found by searching for the image region that has the highest density of pair-wise line intersections. A probabilistic model with Gaussians is constructed to model the noise in the location of a vanishing point. Staircases can be navigated by flying to the center of the staircase. The AR.Drone's onboard intelligence will automatically stabilize the altitude. Lines that represent the staircase are found by classifying the line segments as horizontal or vertical and looking for the largest horizontal-line cluster in the Hough transform space. Once the cluster of horizontal lines is detected, the mean of the lines' endpoints is calculated to retrieve the desired direction.



(a) (Top) Original corridor image. (Bottom) Corridor processed with Canny edge detector and Hough transform to find the vanishing point (blue circle).



(b) (Top) Original image of staircase. (Bottom) Processed image where the bold red line marks the location of the staircase.

Figure 4.4: Autonomous corridor and staircase flights based on line detection (Courtesy Bills et al. [42]).

Map&replay bearing-only navigation

Krajník et al. [45, 46, 47] present a simple monocular navigation system based on the map&replay technique. For this technique, a robot is navigated by a human through an environment and creates a map of its environment. After the map is created, the robot starts at the start position and repeats the learned path. The method can navigate a robot along a given path while observing only one landmark at a time, making it more robust than other monocular approaches.

The authors simplify the problem by splitting the route in straight line segments (i.e., the robot travels in straight lines and rotates between segments). For each segment a set of visual landmarks is remembered, consisting of salient features extracted from the frontal camera image. Similar to Steder, the authors use Speeded-Up Robust Features (SURF) to detect and represent salient features. When entering a segment, the robot is rotated to match the rotation during the mapping phase. During the navigation along a segment, the robot establishes correspondences of the currently seen and previously mapped landmarks and computes differences in the expected and recognized positions for each such correspondence. The robot steers in a direction that reduces those differences while moving straight at a constant speed until its odometry indicates that the current segment has been traversed. The heading is determined by a histogram voting procedure.

5 *Platform: Parrot AR.Drone*

A preliminary step is to find a suitable robot platform that enables both the development and testing of indoor aerial SLAM methods. A straightforward choice is to use a *quadrotor helicopter*, commonly designed to be unmanned aerial vehicles (UAVs). Their small size and extremely maneuverability allows both indoor and outdoor flights. Furthermore, quadrotors do not require mechanical linkages to vary the rotor blade pitch angle, which simplifies the design.

Nowadays, small quadrotors with on-board stabilization can be bought off-the-shelf. These quadrotors make it possible to shift the research from basic control of the platform towards intelligent applications. The platform selected is the Parrot AR.Drone[1] quadrotor helicopter. The main advantages of this quadrotor are its robustness and affordable pricetag. The AR.Drone is equipped with a front-camera and a down-looking camera that provide live video streaming. Two complementary algorithms make use of the down-looking camera for improved stabilization. Furthermore, the AR.Drone is equipped with an ultrasound sensor and an inertial unit that measures pitch, roll, yaw and accelerations along all axes. The vehicle is controlled by sending commands over a Wi-Fi connection.

In this chapter, the AR.Drone is evaluated as a robotic platform. The flight control of a quadrotor is described, AR.Drone's hardware is listed and the intelligent onboard software is described. Moreover, the open Application Programming Interface (API) is examined.

5.1 Quadrotor flight control

The mechanical structure of a quadrotor consists of four rotors attached to a body frame (see Figure 5.1). Each pair of opposite rotors (pair $1, 3$ and pair $2, 4$) is turning the same direction. One pair is turning clockwise and the other counter-clockwise. Each rotor produces both a thrust T and torque τ about its center of rotation, as well as a drag force D_b opposite to the vehicle's direction of flight. *Thrust* T is a force that is generated by expelling or accelerating mass in one direction. The accelerated mass will cause a force of equal magnitude but opposite direction on that system. *Torque* τ is the tendency of a force to rotate an object about an axis. *Drag* D_b is the force that opposes the motion of an aircraft through the air. This force depends on velocity of the vehicle and de-accelerates the vehicle if insufficient thrust is generated. Together the motors should generate sufficient vertical thrust to stay airborne, which is indicated by the gravity force mg in the direction e_D.

For movements, the quadrotor relies on differential torque and thrust. *Pitch*, *roll* and *yaw* are used in flight dynamics to indicate the angles of rotation in three dimensions about the vehicle's center of mass (see Figure 5.2). If all rotors are spinning at the same angular velocity, the differential torque is zero, resulting

[1] `http://ardrone.parrot.com`

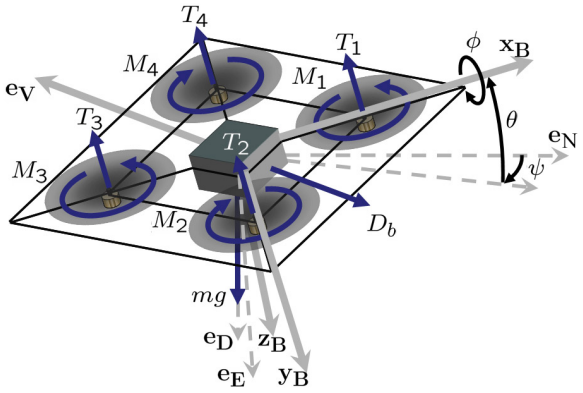


Figure 5.1: Free body diagram of a quadrotor helicopter (Courtesy Hoffman et al. [48]). Note that a right-handed orthogonal coordinate system is used with the z-axis pointing down. Each of the 4 motors has a thrust T_i and momentum M_i. Together the motors should generate sufficient vertical thrust to stay airborne, which is indicated by the gravity force mg in the direction e_D. Differential thrust between the motors can provide roll ϕ and pitch θ torques, which lead to an angle of attack α. This can result in fast movements of the helicopter (e.g. in the horizontal plane) in the direction e_V which a resulting drag force D_b.

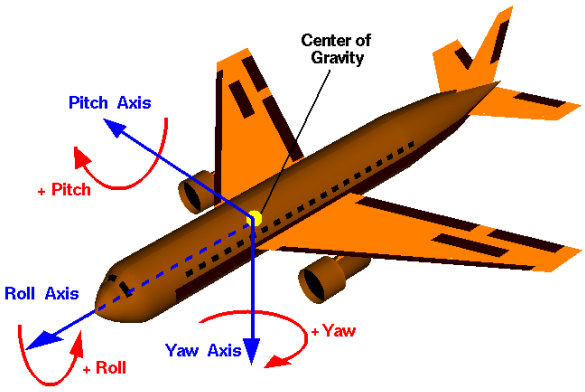


Figure 5.2: Roll, yaw and pitch axis orientation for an aerial vehicle.

in no angular acceleration about the yaw axis. Varying the angular velocity of each rotor pair yields angular acceleration about the yaw axis. If the total thrust is kept constant, the vertical velocity remains unchanged. A vertical movement is achieved by increasing or decreasing the thrust from each motor by the same amount, so that total thrust changes but differential torque on the body remains zero. A horizontal movement is achieved by maintaining a non-zero pitch θ or roll angle ϕ. Angular accelerations about the pitch and roll axes can be generated separately. Each pair of blades rotating in the same direction controls one axis, either roll or pitch. Increasing thrust for one rotor while decreasing thrust for the other will maintain the torque balance needed for yaw stability and induce a differential torque about the roll or pitch axes.

5.2 Hardware

Figure 5.3: The AR.Drone quadrotor helicopter with protection hull attached.

The AR.Drone (Figure 5.3) is a remote-controlled consumer quadrotor helicopter developed by Parrot SA[2]. The body is made of a carbon fiber tube structure and high resistance plastic. A protection hull is made of Expanded Polypropylene (EPP) foam[3], which is durable, light in weight and recyclable. The hull provides protection during indoor flights. The propellers are powered by four brushless motors (35,000 rpm, power: 15W). Energy is provided by a Lithium polymer battery with a capacity of 1000 mAh, which allows a flight time of approximately 10 minutes.

The AR.Drone carries an internal computer with a 468MHz ARM9-processor and 128MB of RAM, running a custom Linux operating system. A mini-USB connector is included for software flashing purposes and to attach add-ons (e.g., GPS sensor). An integrated 802.11g wireless card provides network connectivity with an external device that controls the vehicle. A remote control is not included. Instead, a regular WiFi-enabled device can be used to control the AR.Drone. The AR.Drone was initially designed for Apple platforms (e.g., iPhone, iPod and iPad) and became available on other platforms during 2011. It is also possible to control the AR.Drone from a Linux or Windows PC with the software designed for application developers.

[2] www.parrot.com

[3] http://en.wikipedia.org/wiki/Polypropylene

5.2.1 Sensors

The AR.Drone has different types of sensors. These sensors are used for automatic stabilization. In addition, a front camera provides the user with visual feedback from the vehicle.

Inertial measurement unit

The AR.Drone features a 6 degrees of freedom (DOF) inertial measurement unit. It provides the onboard software with pitch, roll and yaw measurements. These measurements are used for automatic pitch, roll and yaw stabilization and assisted tilting control. The measurement unit is a *micro electro-mechanical system* (MEMS) and contains a 3 axis accelerometer, a 2 axis roll and pitch gyrometer and a single axis yaw gyrometer.

The **accelerometer** is a BMA150[4] made by Bosch Sensortec. An accelerometer outputs g-forces (acceleration relative to free-fall) as a quantity of acceleration. It is based on the phenomenon that the (observed) weight of a mass changes during acceleration. A typical MEMS accelerometer is composed of movable *proof mass* with plates that is attached through a mechanical suspension system to a reference frame. Movable plates and fixed outer plates represent capacitors. The deflection of proof mass is measured using the capacitance difference. The accelerometer has three perpendicular axes. Each axis can only measure acceleration in the direction of the axis. An accelerometer at rest relative to the Earth's surface will indicate approximately 1 g upwards, because any point on the Earth's surface is accelerating upwards relative to the local inertial frame. To obtain the acceleration due to motion with respect to the Earth, this gravity offset must be subtracted and corrected for effects caused by the Earth's rotation relative to the inertial frame. Since the accelerometer can measure gravity, it can be used as a tilt sensor.

The AR.Drone has a 2 axis roll and pitch **gyrometer** and a single axis yaw gyrometer. The gyrometer measures angular velocity in degrees per second, based on the principles of angular momentum. In order to estimate the absolute angle θ, the angular velocity signal Ω needs to be integrated with respect to time. A major problem with this method is that bias errors in the angular velocity signal will cause the integrated angle value to drift over time, since all gyroscopes have at least a small amount of bias error in their angular rate signal.

Regular gyroscopes use a spinning wheel. However, a small device like the AR.Drone cannot afford to have a spinning wheel. Instead, a tiny MEMS gyroscope is used. It comprises of a plate, called the *proof mass*, that vibrates (oscillates). When the vehicle rotates, the proof mass gets displaced in the x, y, and z directions by Coriolis forces. A processor senses the proof mass displacement through capacitor plates located underneath the proof mass, as well as finger capacitors at the edges of the package. The AR.Drone

[4] http://www.bosch-sensortec.com/content/language1/downloads/BMA150_DataSheet_Rev.1.5_30May2008.pdf

uses a IDG-500 Dual-Axis[5] gyroscope for the pitch and roll angles of the drone. The IDG-500 gyro has two separate outputs per axis for higher speed motions (range: $500°/s$, sensitivity: $2.0mV/°/s$) and lower-speed precise movements (range: $110°/s$, sensitivity: $9.1mV/°/s$). The yaw orientation is measured using a high precision gyrometer, the Epson Toyocom XV-3500CB[6]. This sensor has a range of $100°/s$ and a sensitivity of $0.67mV/°/s$.

Ultrasound altimeter

An ultrasound sensor provides altitude measures for automatic altitude stabilization and assisted vertical speed control. The ultrasound sensor is attached on the bottom of the AR.Drone and points downward in order to measure the distance to the floor. A packet of ultrasonic soundwaves is transmitted towards the floor, which reflects the sound back to the sensor. The system then measures the time t for the echo to return to the sensor and computes the distance d to the target using the speed of sound:

$$d = \frac{c \times t}{2} \tag{5.1}$$

where $c \approx 343m/s$ is the speed of sound in air.

Ultrasound range measurements suffer from some fundamental drawbacks which limit the usefulness. These drawbacks are inherent to the principle of an ultrasonic sensor and their commonly used wavelengths. Borenstein and Koren [49] describe some of these drawbacks in the context of obstacle avoidance. A limitation is the small range in which the sensor is able to operate. The ultrasound sensor in the AR.Drone has an effective range of approximately $20cm$ to $6m$. Another limitation is that the ultrasound sensor is unable to obtain precise directional information about objects. Sound propagates in a cone-like manner, where opening angles are commonly between 20 to 40 degrees. Due to this property, the sensor acquires entire regions of constant depth instead of discrete depth points. So, the ultrasound sensor can only tell there is an object at the measured distance somewhere within the measured cone.

Another limitation is the influence of the surface's orientation on the sensor's performance. When the ultrasoundwave is emitted toward a parallel surface of an obstacle, most of the sound energy is reflected perpendicular to the surface and will be detected by the sensor. However, if the surface of the obstacle is tilted relative to the sensor, then only an undetectably small amount of energy will be reflected toward the sensor. Also, the acoustic properties of the material in range have direct impact on the sensor's performance (e.g., foam can acoustically absorb the soundwaves). A final limitation is the limited operating frequency. Before a soundwave can be emitted, the previously emitted soundwave has to be detected by the sensor. This reduces the minimal operating speed of the AR.Drone's ultrasound sensor to $300/(2 \times 6) = 25Hz$.

The opening angle of the AR.Drone's ultrasound sensor is not documented. Appendix A describes an

[5] http://www.sparkfun.com/datasheets/Components/SMD/Datasheet_IDG500.pdf

[6] http://www.eea.epson.com/portal/pls/portal/docs/1/424992.PDF

experiment that was performed to determine the opening angle. The average opening angle was determined as $25.03°$ with a standard deviation of $1.45°$.

During flight, the AR.Drone can take a broad range of angles[7]. The floor is not always perpendicular to the orientation of the ultrasound sensor, which may influence the altitude measurements. A small experiment was performed to measure the effect of the AR.Drone's angle of attack α and the error ϵ of the measured altitude. This experiment was performed above a flat floor. The angle of attack α was changed slowly while keeping the AR.Drone at a fixed altitude. A relation between the angle α and measured altitude was observed. The following (first degree) polynomial fit describes the relation between the angle α in degrees and error ϵ of the measured altitude:

$$\epsilon = 0.2214 \times \alpha \;\; cm \tag{5.2}$$

Cameras

The AR.Drone is equipped with two CMOS cameras: a front camera and a bottom camera. Both cameras support live video streaming at 15 frames per second.

The front camera has a resolution of 640×480 pixels (VGA) and a wide $93°$ field of view. This front camera is used to automatically detect other drones during multiplayer games or to provide video feedback on a screen (e.g., smartphone). The bottom camera has a resolution of 176×144 pixels (QCIF) and a $64°$ field of view. The video frequency of this camera is 60 frames per second to reduce motion blur and improve the applicability of intelligent algorithms. Despite the high frequency, the frames are streamed at 15 frames per second.

The bottom camera plays a central role in the AR.Drone's onboard intelligence. It is used for horizontal stabilization (i.e., automatic hovering and trimming) and to estimate the velocity of the drone. More details about the onboard intelligence can be found in Section 5.3.

5.3 Onboard intelligence

Usually quadrotor remote controls have multiple joysticks and knobs for controlling pitch, roll, yaw and throttle. It generally takes hours of practice to safely perform basic maneuvers like take-off, trimming, hovering with constant altitude and landing. Thanks to the onboard computer and sensors, take-off, hovering, trimming and landing are now completely automatic and all maneuvers are assisted.

Because the quadrotor system is unstable, feedback from the sensors to the control system is required. This raises the issue of state estimation. The vehicle's control system must be robust to cope with various

[7]The AR.Drone's default maximum angle of attack is 12°

disturbances that can be encountered in practice. Redundancy in the state estimation is the solution in this case. The AR.Drone has multiple methods to compute estimated states. However, the onboard software (firmware) is closed source and not documented by the manufacturer. In August 2011, the manufacturer's academic partner *MINES ParisTech for navigation and control design*, published an article [50] which globally describes the navigation and control technology inside the AR.Drone.

5.3.1 Sensor calibration

The AR.Drone uses low-cost sensors, which introduce issues like bias, misalignment angles and scale factors. These errors are not negligible and differ between AR.Drones. For this reason, a manual board calibration is performed at the factory. This calibration process computes the unknown parameters using a least-square minimization method:

$$Y_m = AY_v + B \tag{5.3}$$

where Y_m is the measurement and Y_v the true value of the sensed variable.

Not all errors are compensated during the factory calibration. The AR.Drone's navigation board is mounted on damping foam. To further compensate the misalignment, *onboard calibration* is performed. This is done by considering that the mean acceleration of the UAV must be equal to zero on a stationary flight. A least-squares minimization method is used to determine the micro-rotation. After each landing, this onboard calibration is automatically performed to compensate for the misalignment due to the displacement of the damping foam during the landing shock.

After inertial sensor calibration, sensors are used inside a complementary filter to estimate the attitude and de-bias the gyroscopes. The de-biased gyroscopes are used for vision velocity information combined with the velocity and altitude estimates from a vertical dynamics observer. The velocity from the computer vision algorithm is used to de-bias the accelerometers, the estimated bias is used to increase the accuracy of the attitude estimation algorithm. Finally, the de-biased accelerometer gives a precise body velocity from an aerodynamics model.

5.3.2 State estimation

To address the problem of state estimation, the AR.Drone is equipped with embedded sensors that are described in the previous section. When combining the sensor measurements in data fusion algorithms, relatively good state estimates are obtained. Multiple techniques form a tightly integrated vision and inertial navigation filter that can be summarized as described hereafter. After the inertial sensors are calibrated, the sensors are used inside a complementary filter to estimate the attitude and de-bias the gyroscopes. The de-biased gyroscopes are used for vision-based velocity information provided by the vertical camera. The vision-based velocity is used to de-bias the accelerometers. This estimated accelerometer bias is used to

increase the accuracy of the attitude estimation algorithm. Eventually, the de-biased accelerometer gives a precise body velocity from an aerodynamics model. Below, some techniques are described in more detail.

Visual odometry

The images provided by the vertical camera are used to estimate the 3D velocity of the vehicle. This process is called Visual Odometry and is introduced in Section 3.3. According to [50], two complementary algorithms are available. Depending on the scene content or the expected quality of their results, one is preferred to the other.

The first algorithm is a multi-resolution scheme. The methodology is described in [51]. It computes the optical flow over the whole picture range using a kernel to smooth spatial and temporal derivatives. Attitude changes between two images are ignored during the first resolution refinement step. The error is finally efficiently canceled in most cases by subtracting the displacement of the optical center induced by the attitude change alone. The second algorithm estimates the displacement of several interesting image points (features). For this, FAST image features [52] are used. An iteratively weighted least-squares minimization procedure is used to recover the displacements. The IRLS estimation [53] is carried out by assuming the depth of the scene is uniform. This optical flow based method is used by default. When the scene is deemed suitable for the corner detector and the flight speed is low, the system switches to the second algorithm for increased accuracy.

The inertial sensors are used to compensate for the small rotation between consecutive camera frames. This is a significant help for the determination of the optical flow in the vision algorithm. A specific linear data fusion algorithm combining sonar and accelerometer information is implemented to give accurate vertical velocity and position estimates.

Aerodynamics model

Unfortunately, the vision-based velocity is noisy and relatively slowly updated compared to the vehicle dynamics. The estimated velocity is improved with the help of an aerodynamics model. The accelerometers and gyroscopes are used together as inputs in the motion dynamics to estimate the attitude and velocities. When the aerodynamics velocity and the vision velocity are simultaneously available, the accelerometer bias is estimated and vision-based velocity is filtered. Alternatively, when vision-based velocity is unavailable, only the aerodynamics estimate is used with the last updated value of the bias.

The dynamic behavior of a quadrotor is quite complex. In particular, the aerodynamics of the propellers and the motion of the rigid frame can interfere to produce a coupled dynamical model. This model is beyond the scope of this thesis. In summary, linear drag term exists from the interaction between the rigid body and the rotors and this term is reinforced by tilt phenomenon, which changes a lift force component in drag. These induced-drag effects are non-negligible and they yield interesting information on the velocity

of the system. The induced forces are directly measured by the accelerometers, and through the model, can be used to reliably estimate the velocities of the vehicle.

Section 7.1 describes how the estimated velocities can be used to estimate the position of the AR.Drone.

5.3.3 Controls

The AR.Drone performs actions based on the estimated state and the input from the pilot. A pilot controls the linear and lateral velocity with an input signal s that specifies the pitch of the drone as a factor (between 0 and 1) of the maximum absolute tilt θ_{max}. The altitude is controlled with an input signal that specifies the desired vertical speed as a factor (between 0 and 1) of the maximum vertical velocity v_{max}. A finite state machine is used to handle different modes of the vehicle (e.g., flight, hovering, take off and landing).

The control is realized by two nested loops, the *Attitude Control Loop* and the *Angular Rate Control Loop*. The Attitude Control Loop computes an angular rate from the difference between the estimated attitude and the desired attitude (which is based on the velocity requested by the pilot). The angular rate is tracked with a Proportional Integral (PI) controller. The Angular Rate Control Loop controls the motors with simple proportional controllers. When no user input is given, the AR.Drone goes in hovering mode, where the altitude is kept constant and velocity is stabilized to zero. Transition from flight mode to hovering mode is realized with a motion planning technique, designed to obtain zero speed and zero attitude in short time and without overshooting. For this, a feed-forward control uses inversion of the governing dynamics. The hovering mode is maintained by the Hovering Control Loop, which consists of a PI controller on the estimated speed.

The Attitude Control Loop is also responsible for altitude stabilization (i.e., maintaining a fixed distance between the floor and vehicle). For this, feedback from the ultrasound sensor is used. When an obstacle is in range of the ultrasound sensor, the measured altitude decreases. The AR.Drone responds by increasing its absolute altitude to maintain a fixed measured altitude. Consequently, the AR.Drone decreases its absolute altitude when the obstacle is out of range. However, this behavior was not always observed during experiments. For example, when a floating object ($70cm$ above the floor) becomes in range of the ultrasound sensor. In these cases, the raw ultrasound measurements show a sudden large change, while the onboard filtered altitude measurements show a small change (Figure 5.4). Now, AR.Drone does not change its absolute altitude and remains at the same absolute altitude. This observation suggests that the altitude stabilization is based on a filtered derivative of the ultrasound measurements. The filter minimizes the impact of sudden changes that occur for a short period of time.

Similar to altitude stabilization, the ultrasound sensor is used for take-off and landing. When a take-off command is issued, the engines are started automatically and the AR.Drone increases its altitude until a pre-determined altitude is reached (which is $50cm$ by default). When this altitude is reached, the vehicle switches to hovering mode. During the take-off procedure, the AR.Drone is likely to generate some (ran-

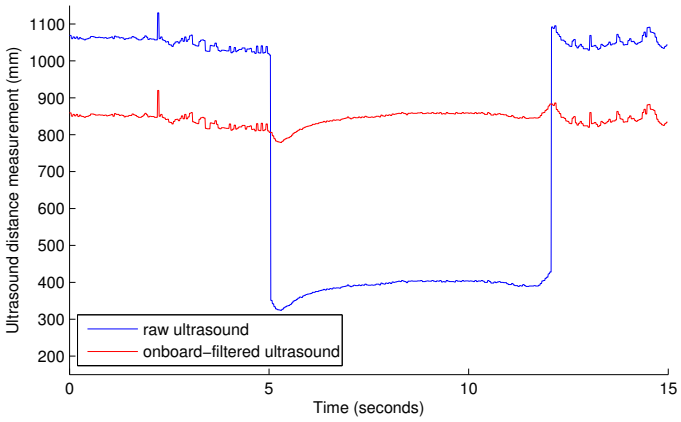


Figure 5.4: Comparison between the AR.Drone's raw and filtered ultrasound measurements when an object floating at $1m$ *comes in range of the ultrasound sensor.*

dom) horizontal movements. These movements are often not observed by the AR.Drone's internal state estimation, probably due to the bad lightning conditions for the vision-based algorithms. In case of a landing command, the AR.Drone decreases its altitude until it has landed on the floor. The vertical velocity is decreased when the drone comes near the floor to prevent a heavy shock. The engines are stopped automatically.

5.4 Open Application Programming Interface

The AR.Drone API[8] (Application Programming Interface) is the reference project for developing applications for the AR.Drone. It includes SDK (Software Development Kit) source code written in C, multiplatform examples and documentation. The API does not include software that is embedded on the AR.Drone.

Communication with the AR.Drone is done through four main *communication services*, which are implemented in the SDK.

1. Controlling and configuring the drone is done by sending *AT commands* on regular basis;

2. Information about the drone (e.g., status, altitude, attitude, speed) is called *navdata*. The navdata also includes filtered and raw sensor measurements. This information is sent by the drone to its client at a frequency of approximately 30 (demo mode) or 200 times per second;

3. A video stream is sent by the AR.Drone to the client device. Images from this video stream are decoded using the codecs (decoders) included in the SDK.

[8] `https://projects.ardrone.org`

4. Critical data is communicated over a channel called the *control port*. This is a TCP connection to provide reliable communication. It is used to retrieve configuration data and to acknowledge important information such as the sending of configuration information.

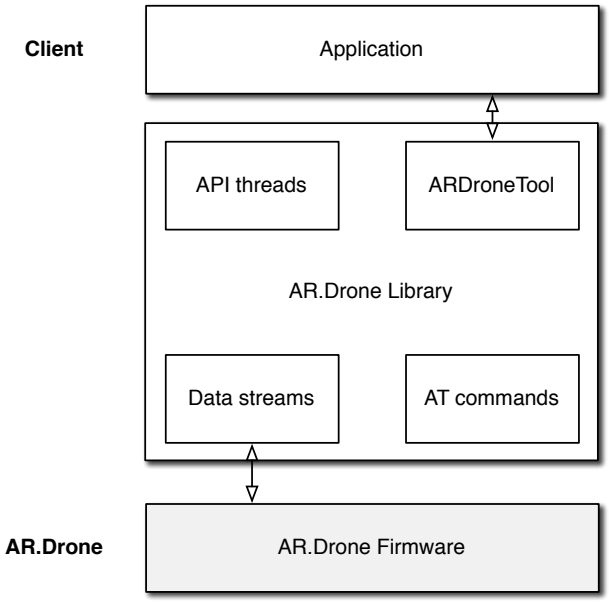


Figure 5.5: Layered architecture of a client application built upon the AR.Drone SDK.

The *AR.Drone Library* is part of the SDK and provides high level APIs to access the drone. Its content can be summarized as following:

- *SOFT*, includes header files describing the communication structures;
  - *ARDroneTool* a set of tools to easily manage the drone, e.g., communication initialization, input-device management, high-level drone control functions and navdata receiving and decoding system;
- *VPSDK*, a set of general purpose libraries: video processing pieces, multiplatform wrappers for system-level functions, multiplatform wrappers for communication functions and helpers to manage video pipelines and threads;
- *VLIB*, the video processing library. It contains the functions to receive and decode the video stream.

Unfortunately, the AR.Drone API has some drawbacks. The API was mainly developed for the iPhone application. Support for other platforms (e.g., Linux and Windows) was added in an early release and not updated for a while. In order to make recent versions of the API working under Windows, a significant amount of fixes is required. Another drawback is the bulkiness. The API consists of many files and lines of undocumented code. This makes it difficult to comprehend what is happening inside the code.

For these reasons, custom software libraries have been released by developers. ARDrone-Control-.NET[9] provides a .Net SDK for Windows. Another developer has written a MeeGo[10] application to control the AR.Drone on Nokia devices. The AR.Drone logic is written in C++. Another interesting project is an Urbi driver[11] in order to control and program the ARDrone from the Urbi platform. Urbi is an open-source software platform to control robots or complex systems in general. The goal of Urbi is to help making robots compatible and simplify the process of writing programs and behaviors for those robots.

[9] https://github.com/shtejv/ARDrone-Control-.NET

[10] http://www.developer.nokia.com/Community/Blogs/blog/kate-alholas-forum-nokia-blog/2010/12/23/ar-drone-with-meego

[11] http://www.psykokwak.com/blog/

6 *Development environment*

As indicated by Michael et al [54], an accurate simulation of a quadrotor is a valuable asset, which allows safe and efficient development of algorithms. Additionally, it gives direct access to ground truth values and allows to design repeatable experiments.

This chapter proceeds as follows. First, the AR.Drone simulation model is presented. This simulation model consists of a motion model, visual model and sensor model. In order to validate the methods presented in Section 7 on both the real and the simulated AR.Drone, an abstraction layer is required. This abstraction layer is part of the framework proposed in Section 6.2.

6.1 Simulation model

The simulation environment selected is USARSim [6, 7], which is based on the Unreal Tournament game engine[1]. It allows realistic simulations and includes a versatile environment editor.

6.1.1 USARSim simulation environment

USARSim is an open source high fidelity robot simulator that can be used both for research and education. It builds upon the Unreal Tournament game engine, a widely used and affordable state of the art commercial game engine. The engine provides realistic rendering and physical simulation. USARSim in itself is a set of models and a hierarchy of classes defining the simulation of robots, sensors and actuators. The simulator is highly configurable and extendible: users can easily add new sensors or robots. The level of effort devoted to validation has been a distinguishing feature of USARSim. Each of its major constituents (e.g., robot kinematics, interaction with the environment, sensors, and camera video) have been subject to ongoing validation testing [55, 56, 57, 58, 59].

USARSim was originally developed aiming to Urban Search And Rescue simulation. The RoboCup Rescue league is part of the RoboCup[2] initiative. This league aims at the development of robotics artifacts that could effectively help human rescuers in the aftermath of disasters like earthquakes, terroristic attacks and other extreme situations. In addition to a Rescue league with real robots, the Virtual Robots Competition was started, which uses USARSim as simulation environment.

Another key feature of USARSim is the *UnrealEd*[3] versatile environment editor. The editor is part of Unreal Tournament and can be used to create and modify virtual environments. It includes geometrical modeling tools which can import and export models compatible with most commercially available mod-

[1] `http://www.unrealengine.com`
[2] `http://www.robocup.org`
[3] `http://nl.wikipedia.org/wiki/UnrealEd`

eling software packages. The details needed for an experiment or competition can be easily added (e.g., roads, fire).

6.1.2 Motion model

The AR.Drone is a stabilized aerial system (Section 5.3). When no control signals are given the quadrotor hovers on the same location. This accomplished by a feedback loop which uses the ultrasound sensor (for altitude) and the bottom camera (for horizontal position). The simulation makes use of this assumption. When no control signal is given, the AR.Drone stays at the same location. When a control signal for a linear (forward) or lateral (sideward) velocity is given, it calculates the force needed to reach that velocity (assuming that the drag force D_b increases linearly with the velocity). When the control signal stops, the drag force D_b slows the quadrotor down until it hovers again. The USARSim quadrotor model uses the Karma physics engine (part of the Unreal Engine [60]) to simulate the force and torque acting on the aircraft. Yet, only the overall thrust is calculated, the differential thrust is not used. When moving in the horizontal plane, a real quadrotor changes its angle of attack (which is the defined as the angle between direction of motion e_V and the body frame e_N [61]). The Karma physics engine does not need this angle to calculate the resulting horizontal movement. Yet, this angle of attack has direct consequences for the viewing directions of the sensors, so the roll and the pitch should be adjusted in correspondence with horizontal movements. The active control of the AR.Drone is incorporated in the value of the dragforce D_b.

Control signals for vertical and rotational movements (around the z-axis) are calculated in the same manner. For vertical movements not only the drag force D_b is taken into account. In this case also the gravitational force mg is included in the equation. Rotations around the z-axis stop quite quickly when no control signal is given. For this rotational movement a strong drag force $D_r = 20 \times D_b$ is used to model the additional inertia. The principal elements of inertia are calculated correspondingly to $(0.0241, 0.0232, 0.0451)kg \cdot m^2$, assuming a homogeneous distribution of the mass.

The result is a simulation model which maneuvers in a way similar to the actual AR.Drone, as demonstrated in Section 8.1.

6.1.3 Sensor model

The USARSim simulator has a set of configurable sensors, which include all the sensors used on the AR.Drone. Each sensor is triggered at fixed time intervals (every $200ms$ by default). The AR.Drone's time interval is reduced to $5ms$, producing up to 200 sensor measurements per second.

The ultrasound sensor emits a number of traces, which in combination form a cone. First the sensor sends out one trace in the direction of its orientation remembering the measured range. Then it does several conical measurements and calculates the minimal measured distance. If the angle between a trace and the surface normal is smaller than angle $\alpha_{maxIncidence}$ the measurement is ignored.

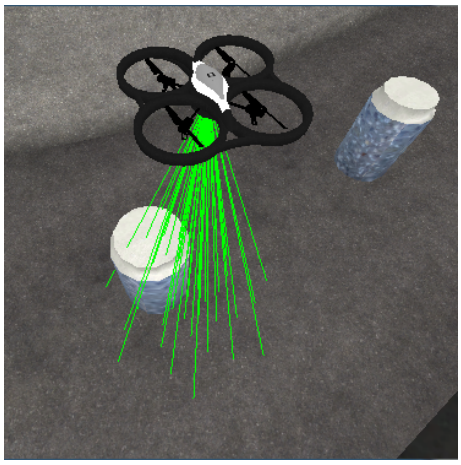


Figure 6.1: The ultrasound sensor is modeled by emitting a number of traces, which in combination form a cone.

The acceleration sensor computes the body accelerations using the AR.Drone's velocity:

$$a_t = (v_t - v_{t-\Delta t})/\Delta t \tag{6.1}$$

where Δt is the time between measurements and v is the velocity of the AR.Drone. The velocity of an object is modeled explicitly by the simulator.

The camera is modeled by creating a viewpoint in the Unreal engine. The field of view is set to $64°$ and the resolution is set to 176×144 pixels, matching the specifications of the AR.Drone's bottom camera. Camera calibration of the virtual camera was performed using OpenCV's Calibration tool[4]. The calibration pattern was placed in a virtual environment and the AR.Drone maneuvered above it.

Figure 6.2: Calibration of a virtual camera inside the USARSim simulator.

USARSim supports dynamic lightning to produce realistic images and shadows. However, the camera

[4] http://opencv.willowgarage.com/documentation/camera_calibration_and_3d_reconstruction.html

sensor lacks a noise model and does not emulate the effect of automatic white balancing that is performed in most cameras (including the AR.Drone's camera). An article of the early developments during this thesis [62] describes the influence of white balancing on image stitching and how to mimic the real images as close as possible.

6.1.4 Visual model

In addition to the motion and sensor model, a visual model of the AR.Drone has been developed. A highly detailed 3D model of the AR.Drone was provided by Parrot SA. However, this model was unsuitable for simulation due to the computational complexity. A simplified model (Figure 6.3) was made using Blender[5]. This model is optimized for simulation and consists of only 3142 vertices. Both the simulated and real system have the same dimensions $(0.525, 0.515, 0.115)m$.

Figure 6.3: 3D model of the Parrot AR.Drone. This is a model optimized for simulation and is based on the highly detailed model provided by Parrot SA.

6.2 Proposed framework

The AR.Drone API (Section 5.4) is the reference project for developing applications for the AR.Drone. It provides basic functionality for communicating with the AR.Drone. In order to perform advanced tasks (e.g., sensor data processing and automated drone control), a framework is proposed. This framework includes an abstraction layer to abstract from the actual device. Interfaces are used to connect the framework to an actual or virtual device. Due to this abstraction, both the real and simulated AR.Drone can be used to validate the methods presented in the next chapter.

The main functionalities of the framework can be summarized as follows:

- Object-oriented programming: robots are represented with an object and are configured using parameters;
- Abstraction from actual device: both the real and simulated AR.Drone can be connected to the framework;
- Dataset recording and playback;

[5] http://www.blender.org/

- Queued sensor data and camera frame processing;
- Keyboard and 3D mouse controls;
- Autonomous waypoint navigation;
- Real-time 3D map visualization with freeflight over the map

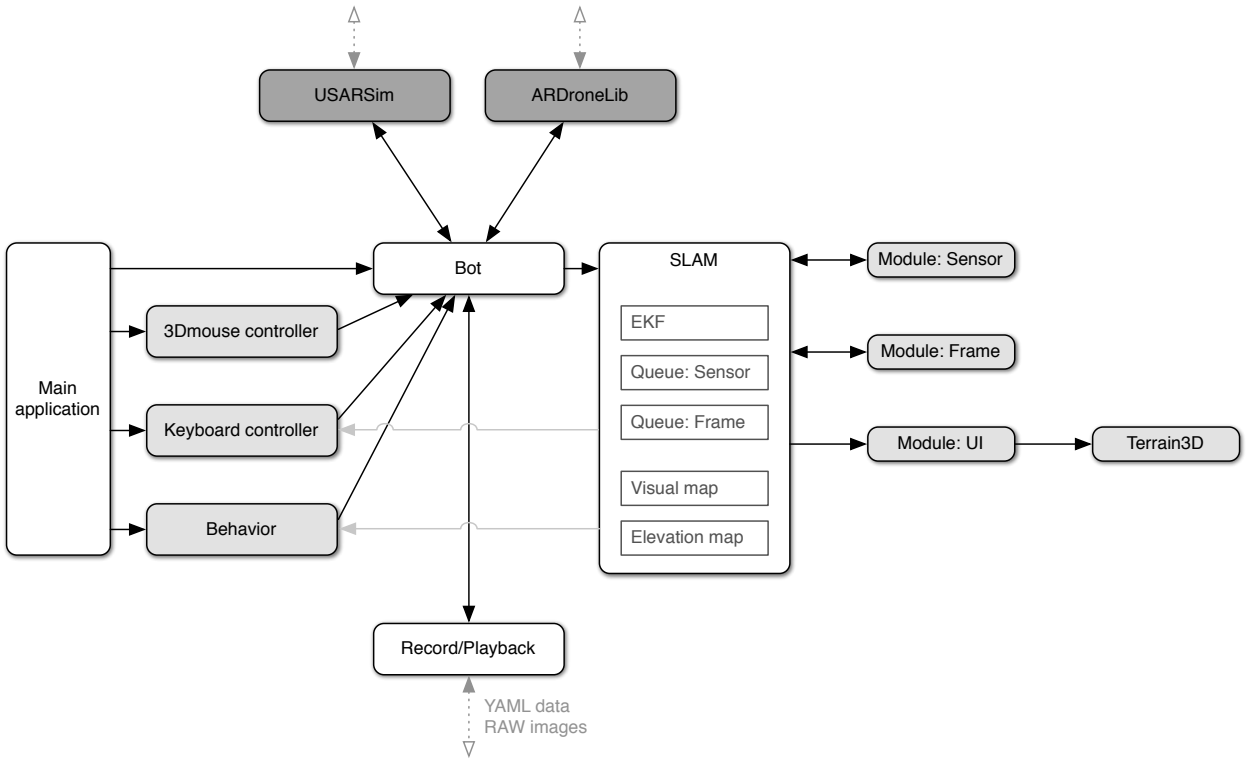


Figure 6.4: Schematic overview of the proposed framework. Each node represents an object. A light gray node indicates the object is running in a dedicated thread. A dark gray node indicates an interface to a device.

A schematic overview of the framework is given in Figure 6.4. A central component of the framework is the Bot object, which represents a robot. Multiple robots can be controlled simultaneous (e.g., a real and simulated AR.Drone). The framework is limited to a single real AR.Drone's due to the lack of multi-robot support in the ARDroneLib. A robot is controlled by calling member functions of the corresponding bot object. Currently, three control methods are implemented: a *keyboard controller*, a *3D mouse controller* and a *behavior controller*. Both the keyboard controller and 3D mouse controller can be used to manually fly the AR.Drone and manipulate the behavior of the SLAM system. The *behavior controller* is used to autonomously fly the AR.Drone. It provides waypoint navigation to autonomously navigate the AR.Drone along a predefined route in world coordinates. Feedback from the SLAM system (e.g., robot's estimated state) is used by the behavior controller to select the appropriate commands for the robot.

In addition to the controllers, a record and playback functionality is present. Playback of a dataset is a valuable asset for testing. All control commands sent to the drone and data received from the drone (i.e., navdata and camera frames) can be recorded to a dataset. The data is stored in YAML[6] format. When playing back a dataset, the data is pushed to the bot object in a way similar to data that is received from a live robot. A timer is used to push the data at the appropriate timestamps, mimicking the original data as closely as possible.

An abstraction layer is used to abstract a bot from the actual device. This enables the control of both a real and simulated AR.Drone in the same manner. *Interfaces* are used to map a bot object to a (simulated) device. When a bot object is created, an interface has to be selected. This can be either a real AR.Drone (using the ARDroneLib), simulated AR.Drone (USARSim) or use no interface at all for dataset playback (i.e., there is no communication with a robot). The interface is responsible for transforming data received in device-specific format to a generic data-structure used by the framework. Also, the control commands that are sent to the robot are transformed from generic format to device-specific format.

The data received from a robot (i.e., navdata and camera frames) is pushed to the SLAM object, which implements the SLAM system presented in Chapter 7. Navdata is added to the Sensor Queue, camera frames are added to the Frame Queue. If the Frame Queue is not empty when a new frame is received (i.e., another frame is being processed), the frame is dropped by default. The advantages of a queue are twofold: it allows data buffering and passing data to another thread to prevent blocking. The latter is very important for a SLAM system, since blocking results in wasted data or delayed data. The SLAM object has three modules: Sensor, Frame and UI. Each module runs in a separate thread to prevent blocking and allows simultaneous data processing. Both the Sensor and Frame module are able to update the state (EKF) independently.

The **Sensor module** processes navdata (e.g., altitude and velocity estimates) from the AR.Drone at 200Hz. When a navdata package is received, the EKF predicts a new state from the previous estimate (Section 2.5.1). Each navdata package is processed before it can be used to update the estimated state of the AR.Drone. This processing task includes the transformation of measurements to the global reference frame and the detection of obstacles (Section 7.5). The processed navdata is used to generate a measurement for the EKF. Elements of the measurement matrix that cannot be (reliably) extracted from the processed navdata, are extracted from the predicted state and added to the measurement matrix with a higher covariance (uncertainty). Finally, the measurement is used to update the state of the EKF.

The **Frame module** processes camera frames. When a camera frame is received, the EKF predicts a new state from the previous estimate (Section 2.5.1). Each camera frame is processed to extract information that is used to update the estimated state of the AR.Drone. Processing includes Visual Odometry (Section 7.4), updating the map (Section 7.2) and recovering the global position of the AR.Drone by matching the

[6] http://en.wikipedia.org/wiki/YAML

frame against the map (Section 7.3). The extracted information (i.e., velocities or global position) is used to generate a measurement for the EKF. Elements of the measurement matrix that cannot be extracted from the camera frames (e.g., altitude), are extracted from the predicted state and added to the measurement matrix with a higher covariance (uncertainty). Finally, the measurement is used to update the state of the EKF.

The average processing time of a single frame is $60ms$ and causes a delay between input and updating the state. This introduces a synchronization issue between state updates: when a update from the Frame module is applied, the estimated state is already updated by the Sensor module with more recent information that originates from the navdata. This issue is solved by pausing the Sensor module (processing) while a frame is being processed. After the state update by the Frame module is applied, the Sensor module is resumed and the queued navdata information is processed.

The **UI module** is responsible for visualizing the map. It initializes a Terrain3D object and pushes local map updates to it. The Terrain3D object renders the map (Figure 6.5). It allows 3D rendering of a textured elevation map. Both the visual map and elevation map are integrated to a single map. The code is based on the work by Chad Vernon [7] and uses DirectX 9 for rendering. The estimated position of the AR.Drone is visualized with a red arrow.

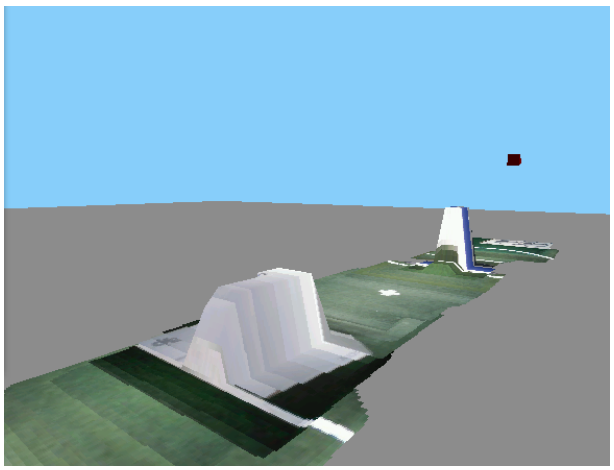


Figure 6.5: Textured elevation map rendered by the Terrain3D object. The red arrow indicates the position and orientation of the AR.Drone.

[7] http://www.chadvernon.com/blog/resources/directx9/moving-around-a-3d-world

Visual SLAM with the AR.Drone 7

This chapter presents methods to build a visual elevation map and use this map to make the AR.Drone localize itself. In order to initialize a visual localization and mapping algorithm, a good estimate of the position based on the internal sensors of the AR.Drone is essential. Section 7.1 describes how the pose is estimated. When an estimate of the pose is available, a map can be constructed using the method described in Section 7.2. Section 7.3 describes how this map is used by the AR.Drone to localize itself and reduce the error of the estimated pose. This localization method is extended in Section 7.4 to operate as a visual odometry algorithm. Finally, Section 7.5 describes an experimental method to build an elevation map using a single airborne ultrasound sensor.

7.1 Pose estimation

The AR.Drone has several sources of information available about its movements and the majority of its computing power is dedicated to combine these sources into reliable estimates (Section 5.3). In the proposed methods, this information is combined into a state vector, with all information needed to initialize visual localization and mapping.

The AR.Drone is equipped with a number of sensors, which give frequent updates about the motion. The inertial sensor measures the body accelerations and angular velocities. A proprietary filter on board of the AR.Drone converts the angular velocities to an estimated attitude (orientation). An ultrasound sensor is used to measure the altitude of the vehicle. In addition to the body accelerations, the AR.Drone sends an estimate of its estimated body velocity (Section 5.3). This estimate is based on the inertia measurements, aerodynamic model and visual odometry obtained from the relative motion between camera frames. The details of this algorithm are proprietary information of the manufacturer.

The information from the sensors are used in an Extended Kalman Filter (EKF, Section 2.5.1) to estimate the current pose. An EKF has been considered [63] the facto standard in the theory of nonlinear state estimation. The EKF state vector comprises a position vector p^W, velocity vector v^W, acceleration vector a^W and attitude (orientation) vector q^W. All vectors are 3-dimensional Cartesian coordinates. The position vector p^W includes the estimated altitude, which is based on the AR.Drone's ultrasound measurement plus the estimated elevation of the floor (Section 7.5). An additional altitude vector h^W is added to the EKF state vector, which contains the unmodified ultrasound measurement without the estimated elevation. Furthermore, it contains first-order and second-order derivatives of the ultrasound measurement. These derivatives are used by the elevation mapping method (Section 7.5).

$$x = [p^W v^W a^W q^W h^W] \tag{7.1}$$

The sensor data from the AR.Drone is used to fill a measurement matrix. This measurement matrix is used by the EKF to update the state (Section 2.1). The attitude (orientation) information from the AR.Drone's proprietary onboard filter is written to the attitude vector of the measurement matrix. This is possible because the attitude measurements are filtered and debiased onboard of the AR.Drone (Section 5.3). The position of the AR.Drone cannot be estimated directly and is derived from the estimated velocities. The velocity estimate sent by the AR.Drone is written to the velocity vector of the measurement matrix. Based on the previous position p_{t-1}, the state's velocity v_t and time between measurements Δt, the new position p_t can be calculated as follows:

$$p_t = p_{t-1} + v_t \times \Delta t + a_t \times 0.5\Delta t^2 \tag{7.2}$$

where Δt is the variable time (seconds) between the last two measurements. Instead of using the AR.Drone's onboard velocity estimate, the visual odometry method proposed in Section 7.4 can be used to determine the AR.Drone's velocity.

In theory, the velocity can be estimated by integrating the body acceleration measurements. However, the low-cost acceleration sensor from the AR.Drone provides unreliable data, which would result in large velocity errors. Therefore, the body acceleration measurements are not written to the EKF's measurement matrix.

Similar to the estimated position, the vertical velocity estimate is used to estimate the vehicle's altitude. However, the ultrasound altitude measurement can be integrated into the EKF's measurement matrix to improve the altitude estimate. The ultrasound sensor is not sensitive to drift because it provides an absolute altitude measurement. Relying only on the ultrasound sensor is not optimal since the measurements are affected by the material and structure of the floor (Section 5.2.1).

7.2 Mapping

When an estimate of the AR.Drone's pose is available, a map can be constructed to make the AR.Drone localize itself and reduce the error of the estimated pose. The map consists of a texture map and a feature map. The texture map is used for human navigation and the feature map is used by the AR.Drone to localize itself.

7.2.1 Texture map

Now that we have an estimate of the AR.Drone's position and attitude in the world, we can use this information to build a texture map of the environment. The AR.Drone is equipped with a down-looking camera that has a resolution of 176×144 pixels. The frames captured by this camera can be warped on a flat canvas to create a texture map. Directly merging the frames on the canvas is not possible, because individual

frames can be taken from a broad range of angles and altitudes. Instead, perspective correction is applied and all frames are normalized in size and orientation.

An initial approach of the mapping algorithm was published [62] and presented at the IMAV2011 conference. This approach uses image stitching [64] to create a texture map. Knowledge about the AR.Drone estimated state is not used. Because the transformations of all previous frames are used to compute the transformation of the recent frame, the method is sensitive to incorrect transformations. Therefore, a new mapping method is developed.

Before describing how a frame is warped, some basics about image formation are repeated. The AR.Drone's camera is modeled using a pinhole camera model (Figure 7.1). In this model, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

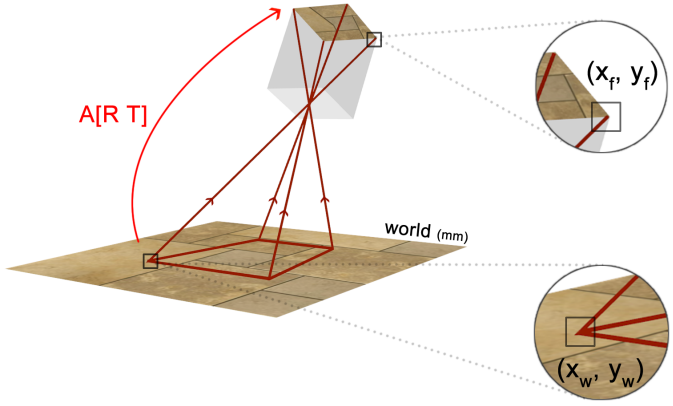


Figure 7.1: In the pinhole camera model, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$\begin{bmatrix} x_f \\ y_f \\ 1 \end{bmatrix} = A[R|t] \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \tag{7.3}$$

where x_f and y_f represent a 2D point in pixel coordinates and x_w, y_w and z_w represent a 3D point in world coordinates. The 3×3 *camera intrinsic* matrix A includes the camera's focal length and principal point. The 3×4 joint rotation-translation matrix $[R|t]$ includes the *camera extrinsic* parameters, which denote the coordinate system transformations from 3D world coordinates to 3D camera coordinates. Equivalently, the extrinsic parameters define the position of the camera center and the camera's heading (attitude) in world coordinates.

In order to warp a camera frame on the correct position of the canvas, the algorithm needs to know which area of the world (floor) is captured by the camera. This is the inverse operation of the image

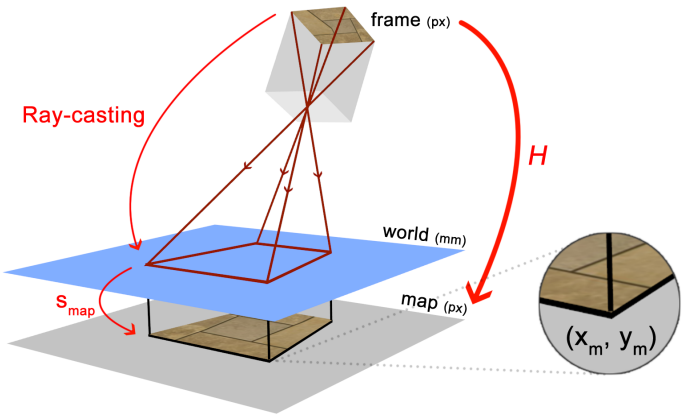


Figure 7.2: Texture map: warping a camera frame on the map's canvas. The frame's corners (px) are mapped to 2D world coordinates that lie on the world's plane. The 2D world coordinates of the frame are mapped to the map's canvas coordinates (px). Equation 7.7 is used to compute the perspective transformation H to transform all pixels of the frame to the corresponding positions on the map's canvas.

formation described above. Instead of mapping 3D world coordinates to 2D pixel coordinates, 2D pixel coordinates are mapped to 3D world coordinates (mm). It is impossible to recover the exact 3D world coordinates, because a pixel coordinate maps to a line instead of a point (i.e., multiple points in 3D world coordinates map to the same 2D pixel coordinate). This ambiguity can be resolved by assuming that all 3D world points lie on a plane ($z_w = 0$), which makes it possible to recover x_w and y_w. The 2D world coordinates of the frame's corners are obtained by casting rays from the four frame's corners (top of Figure 7.2). The 2D world coordinate that corresponds to a frame corner is defined as the point $(x_w, y_w, 0)$ where the ray intersects the world plane ($z_w = 0$). Both x_w and y_w are computed as follows:

$$\begin{bmatrix} x_w \\ y_w \\ 0 \end{bmatrix} = \frac{(p_0 - l_0) \cdot n}{l \cdot n} \tag{7.4}$$

where $n = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix}^T$ is a normal vector to the world plane and $p_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$ is a point on the world plane. l is a vector in the direction of the ray and $l_0 = p^W$ is a point where the ray intersects the camera plane. l is computed as follows:

$$l = \left| RA^{-1} \begin{bmatrix} x_f \\ y_f \\ 1 \end{bmatrix} \right| \tag{7.5}$$

Both the camera's extrinsic and intrinsic parameters are required for this operation. The extrinsic parameters (position and attitude of the camera) are provided by the EKF state vector. The camera intrinsic

parameters are estimated using OpenCV's camera calibration tool[1]. The implementation is based on the work from [65, 66].

Now, a relation between the pixel coordinates and world coordinates is known. However, the 2D world coordinates (mm) need to be mapped to the corresponding 2D pixel coordinates (x_m, y_m) of the map's canvas. A canvas with a fixed resolution of $4.883mm/px$ is used.

$$\begin{bmatrix} x_m \\ y_m \end{bmatrix} = s_{map} \cdot \begin{bmatrix} x_w \\ y_w \end{bmatrix} \tag{7.6}$$

where $s_{map} = 1/4.884$.

Now, the map's pixel coordinates of the frame corners are known (bottom of Figure 7.2). In order to transform a frame to the map's canvas, a relation between the frame's pixels and map's pixels is required. A transformation from the frame's corner pixel coordinates and the corresponding pixel coordinates of the map's canvas can be expressed as:

$$\begin{bmatrix} x_{m,i} \\ y_{m,i} \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_{f,i} \\ y_{f,i} \\ 1 \end{bmatrix} \tag{7.7}$$

The local perspective transformation H is calculated by minimizing the back-projection using a least-squares algorithm. OpenCV's *findHomography*[3] is used to solve the perspective transformation H. This transformation describes how each frame's pixel needs to be transformed in order to map to the corresponding (sub) pixel of the map's canvas. The transformation is used to warp the frame on the map's canvas. By warping the frame on a flat canvas, implicit perspective correction is applied to the frames. OpenCV's *warpPerspective*[2] is used to warp the frame on the map's canvas.

In this way a texture map can be built. This map consists of a set overlapping textures. The placement and discontinuities of the overlapping textures can be further optimized by map stitching, as described in [62]. Here the texture map is used for human navigation. For localization a feature map is used, as described in the next section.

7.2.2 Feature map

In addition to the texture map described above, a grid of image features (feature map) is created using a method that is presented in this section. This feature map will be used for localization purposes, as described in Section 7.3. The inertia measurements and visual odometry provide frequent position estimates. If the AR.Drone is able to relate a video frame to a position inside the feature map, the vehicle is able to

[1] http://opencv.itseez.com/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html
[2] http://opencv.itseez.com/modules/imgproc/doc/geometric_transformations.html

correct this drift long enough to build a map as large as the indoor arena of the IMAV competition (as described in Section 8.2.3).

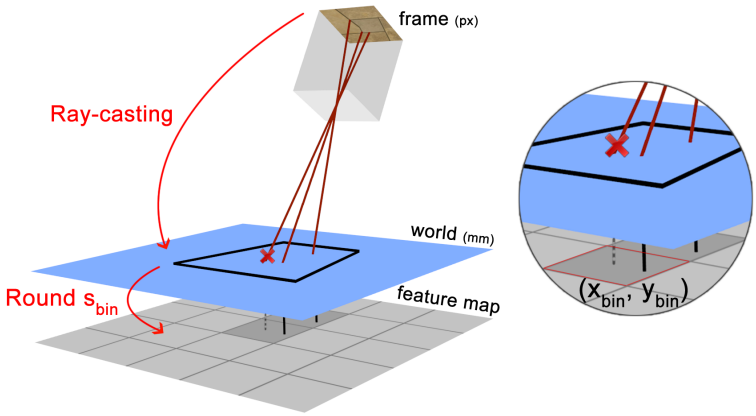


Figure 7.3: Feature map: adding visual features to a feature grid. Each feature found in the camera frame is mapped to the corresponding 2D world coordinates. The 2D world coordinates are mapped to the grid cell the feature belongs to. For each cell, only the best feature is stored.

A 2D grid with a fixed resolution of $100 \times 100mm$ per cell is used. From each camera frame the method extracts Speeded-Up Robust Features (SURF) [17] that are invariant with respect to rotation and scale. Each feature is an abstract description of an interesting part of an image (Section 3.2). A SURF feature is described by a center point in image sub-pixel coordinates and a descriptor vector that consists of 64 floating point numbers.

Each feature that is detected in a camera frame is mapped to the corresponding cell of the feature map. A feature's center point (x_f, y_f) is transformed to its corresponding position in 2D world coordinates (x_w, y_w), visible in the top of Figure 7.3. This is done by casting a ray from the features pixel coordinates in the frame. The method is similar to the method used for casting ray's from the frame's corners (Section 7.2.1). Finally, the 2D world position (x_w, y_w) of each feature is transformed to the corresponding cell indices (x_{bin}, y_{bin}), visible in the bottom of Figure 7.3.

$$\begin{bmatrix} x_{bin} \\ y_{bin} \end{bmatrix} = Round(s_{bin} \cdot \begin{bmatrix} x_w \\ y_w \end{bmatrix}) \tag{7.8}$$

where $s_{bin} = 0.01$ for a cell size of $100 \times 100mm$.

For each cell, only the best feature (e.g. with the highest response) is kept and the other features are dropped. If a cell already contains a feature descriptor ($grid_{x,y} \neq \emptyset$), the cell is ignored.

$$grid_{x,y} = \begin{cases} \arg\max_{d \in D_{x,y}} response(d) & \text{if } grid_{x,y} = \emptyset \\ grid_{x,y} & \text{else} \end{cases} \tag{7.9}$$

where $D_{x,y}$ is the set of features that is mapped to cell x_{bin}, y_{bin}.

The remaining (best) feature descriptors, including the corresponding world coordinates (x_w, y_w), are added to the corresponding grid cells.

7.3 Localization

The feature map created in the previous section can be used for absolute position estimates, at the moment of loop-closure (when the AR.Drone observes a location it has observed before). This allows to correct the drift that originates from the internal sensors of the AR.Drone. The sensor measurements provide frequent velocity estimates. However, the estimated position will drift over time, because the errors of the velocities estimates are being accumulated. The feature map that is created can be exploited to reduce this drift, because localization against this map provides absolute positions of the vehicle. These absolute positions are integrated into the EKF and improve the estimated position and reduce the covariance of the state.

When a camera frame is received, SURF features are extracted. Each feature consists of a center position in pixel coordinates (x_f, y_f) and a feature descriptor. A feature's center point (x_f, y_f) is transformed to its corresponding position in 2D world coordinates (x_w, y_w), visible in the top of Figure 7.3. This is done by casting a ray from the features pixel coordinates in the frame. The method is similar to the method used for casting ray's from the frame's corners (Section 7.2.1).

The next step is matching the feature descriptors from the camera frame against the feature descriptors from the feature map. When the feature map is quite large, this process becomes slow. However, the estimated position of the vehicle can be used to select a subset of the feature map. This can be done by placing a window that is centered at the vehicle's estimated position. The covariance of the estimated position can be used to determine the size of the window. The set of frame descriptors (query descriptors D_q) is matched against the map descriptors (training descriptors D_t). Matching is done using a brute force matcher that uses the L^2 norm as similarity measure. For each query descriptor d_q from the frame, function $C(d_q)$ selects the training descriptor d_t from the map that minimizes the L^2 norm:

$$C(d_q) = \arg \min_{d_t \in D_T} L^2(d_q, d_t) \tag{7.10}$$

where D_T is the set of map descriptors within a window around the estimated position. The L^2 distance between two descriptors a and b is defined as:

$$L^2(a, b) = \sqrt{\sum_{i=1}^{N} |a_i - b_i|^2} \tag{7.11}$$

where $N = 64$ is the length of the SURF descriptor vector.

Each query descriptor (frame) is matched against the descriptor from the training descriptors (map)

that is most similar. Please note it is possible that multiple descriptors from the frame are matched against a single descriptor from the map.

For each match $C(d_q, d_t)$ the 2D world coordinates (x_{w,d_q}, y_{w,d_q}) and (x_{w,d_t}, y_{w,d_t}) of both descriptors are already computed. These point pairs can be used to calculate a transformation between the query points (frame) and the training points (map). This transformation describes the relation between the EKF estimated vehicle position (described in Section 7.1) and the position according to the feature map.

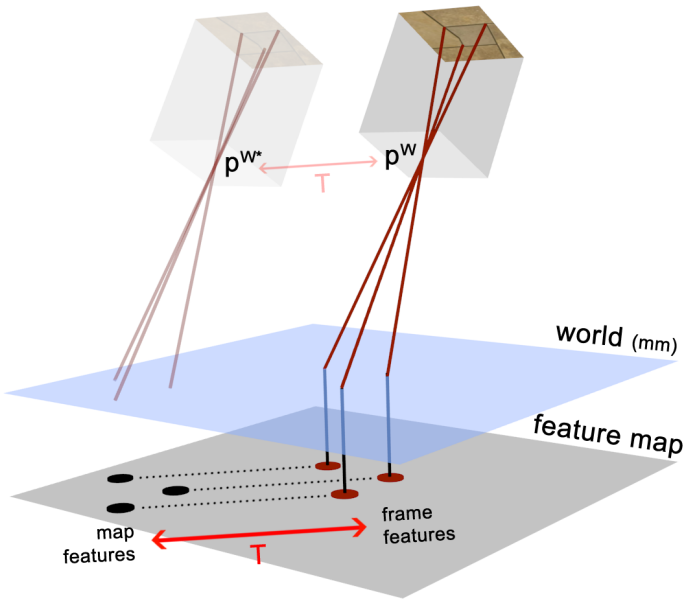


Figure 7.4: Localization: features from the camera frame (red circles) are matched against the features from the feature map (black circles). All the feature points (pixels) are converted to 2D word coordinates. The translation T between both feature sets is calculated and used to correct the estimated position of the AR.Drone.

7.3.1 Pose recovery approaches

Different types of transformations can be used to describe the relation between the point pairs (e.g., perspective transformation or affine transformation). However, if not all of the point pairs fit the transformation (due to outliers), the initial transformation estimate will be poor. Random Sample Consensus (RANSAC) [22] is used to filter the set of matches in order to detect and eliminate faulty matches (point pairs). RANSAC tries different random subsets of corresponding point pairs. It estimates the transformation using this subset and then computes the quality of a transformation by counting the number of inliers.

Since the point pairs are normalized, the perspective deformation and scale differences between the point pairs (matches) are already removed. The remaining degrees of freedom are the translation and rotation between the point pairs. Perspective transformation and affine transformation provide more degrees of freedom than required. This is potentially dangerous, because an incorrect transformation may still result

in a high percentage of inliers. For example, this may happen when multiple query descriptors (from the frame) are matched against a single training descriptor (from the map). In that case, a perspective or affine transformation is found that scales all points to a single position. This transformation does not correspond with the actual translation and rotation, but is a valid transformation according to the RANSAC algorithm.

Instead of computing a full perspective, affine or Euclidean transformation, a more pragmatic approach is used in this thesis. Here, only the translation in x and y direction is estimated. The best translation is chosen with a modified RANSAC algorithm, which uses covariance between matches instead of the number inliers as quality measure. The rotation is estimated independently, as described at the end of this section.

A translation hypothesis T is computed by taking a subset of three random point pairs and calculating the mean translation:

$$T_x = \frac{\sum_i^N (x_{w,d_t,i} - x_{w,d_q,i})}{N} \tag{7.12}$$

$$T_y = \frac{\sum_i^N (y_{w,d_t,i} - y_{w,d_q,i})}{N} \tag{7.13}$$

where $N = 3$ is the number of point pairs. The standard deviation of translation T is defined as:

$$\sigma_x = \sqrt{\sum_i^N ((x_{w,d_t} - x_{w,d_q}) - T_x)^2} \tag{7.14}$$

$$\sigma_y = \sqrt{\sum_i^N ((y_{w,d_t} - y_{w,d_q}) - T_y)^2} \tag{7.15}$$

The confidence c of translation T is computed using the standard deviation:

$$c_T = 1 - \sigma_x/\theta_d - \sigma_y/\theta_d \tag{7.16}$$

where θ_d = 200. Similar to RANSAC, this step is repeated k times to test multiple hypotheses. After all repetitions are performed, the translation with highest confidence T_{best} is used as final translation.

The advantages of this approach are threefold. By using the confidence, instead of the number of inliers (as mostly done in classic RANSAC), a more robust estimate of the translation can be made. Furthermore, this approach eliminates the additional degrees of freedom that can results in incorrect transformations (as described above). An additional benefit of the two degrees of freedom approach is its efficiency, allowing a great number of RANSAC iterations to find the optimal subset of point pairs.

If the confidence c_T of the translation T_{best} exceeds a threshold, the transformation T is added to the estimated position p^W to compute the corrected position p^{W*}. This corrected position is integrated in the EKF as measurement with low covariance. Please note this method requires that the estimated yaw of the vehicle is close to the actual yaw. A large difference between estimated and actual yaw results in low confidence c_T. Therefore, the next paragraph describes a method to reduce the drift in the estimated yaw. Performing this optimization on regular basis should sufficiently correct for drift in the estimated yaw.

If the confidence is exceptionally high (i.e., good matches are found), the selected point pairs are used to determine the rotation between the estimated yaw and the real yaw of the vehicle. This rotation is used to correct the drift of the yaw. The rotation is computed using Kabsch algorithm [67]. This method computes the optimal rotation matrix that minimizes the RMSD (root mean squared deviation) between two sets of points. These sets of points are the same two sets of points used to compute the translation, as described above. This rotation is added as an offset to all future yaw measurements.

In Section 8.3, an experiment is performed to validate that the proposed pose recovery method outperforms three other approaches.

7.4 Visual odometry

The AR.Drone's onboard intelligence estimates the velocity using two complementary visual odometry algorithms and an aerodynamic model of the AR.Drone. However, its limited processing power requires the use of lightweight features and algorithms, which may affect the accuracy of the estimated velocities. Running a visual odometry algorithm offboard relaxes the computational limitations, allowing more complex (better) features and algorithms. In this section, the localization method presented in the previous section is modified to operate as a visual odometry algorithm. Instead of matching the last frame against the feature map, the last frame is matched against the previous frame. The resulting velocity estimates can be merged with the onboard velocity estimates to improve the accuracy of estimated velocity.

When a camera frame f_t is received, SURF features are extracted. Each feature consists of a center position in pixel coordinates (x_f, y_f) and a feature descriptor. A feature's center point (x_f, y_f) is transformed to its corresponding position in 2D world coordinates (x_w, y_w). This is done using ray-casting, as described in Section 7.2.1. Contrary to the previous sections, the origin of the ray is set to $(0, 0, z)$, such that the estimated position of the AR.Drone does not influence the visual odometry calculations.

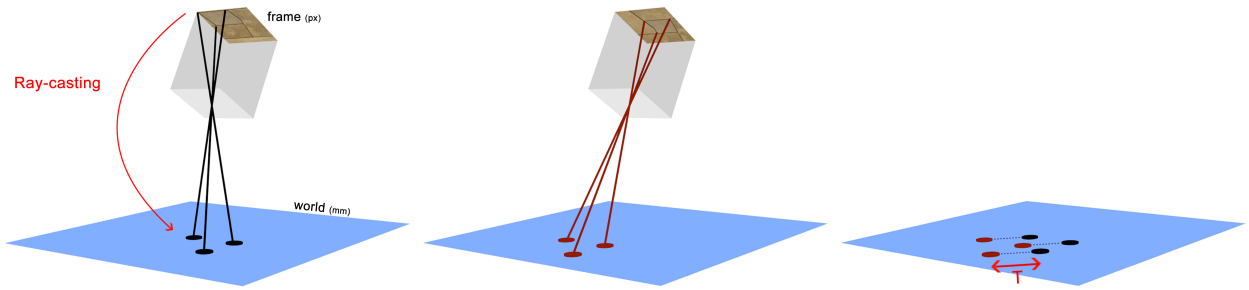
The next step is matching the feature descriptors from the last frame f_t against the feature descriptors from the previous frame f_{t-1}. The set of descriptors D_t from the current frame f_t is matched against the descriptors D_{t-1} from the previous frame f_{t-1}. Matching is done using a brute force matcher that uses the L^2 norm as similarity measure. For each descriptor d_t from frame f_t, function $C(d_t)$ selects the descriptor d_{t-1} from the previous frame f_{t-1} that minimizes the L^2 norm:

$$C(d_t) = \arg \min_{d_{t-1} \in D_{t-1}} L^2(d_t, d_{t-1}) \tag{7.17}$$

The L^2 distance between two descriptors a and b is defined as:

$$L^2(a, b) = \sqrt{\sum_{i=1}^{N} |a_i - b_i|^2} \tag{7.18}$$

where $N = 64$ is the length of the SURF descriptor vector.



(a) Frame f_{t-1} (b) Frame f_t (c) Feature coordinates of frame f_{t-1} and f_t

Figure 7.5: Overview of the visual odometry algorithm. In 7.5(a) features from the previous frame f_{t-1} are transformed to corresponding 2D world coordinates using ray-casting. In 7.5(b) this step is repeated for the newest frame f_t. In both frames the same three features are found, at different locations in the frame. These differences indicate the AR.Drone has moved. Features from this new frame f_t (red circles) are matched against the features from the previous frame (black circles). In 7.5(c) a transformation is computed between the world coordinates of both feature sets. This transformation describes the movement T of the AR.Drone between both frames.

Each descriptor from frame f_t is matched against the most similar descriptor from the previous frame f_{t-1}. For each match $C(d_t, d_{t-1})$ the 2D world coordinates (x_{w,d_t}, y_{w,d_t}) and $(x_{w,d_{t-1}}, y_{w,d_{t-1}})$ of both descriptors are already computed. These point pairs can be used to calculate a transformation between the points from the last frame and the previous frame. This transformation describes the movement (e.g., translation) of the AR.Drone between the last two frames. The transformation is computed using the method described in Section 7.3.1. Now, the velocity is computed using:

$$V_x = Tx/\Delta t \tag{7.19}$$

$$V_y = Ty/\Delta t \tag{7.20}$$

where V is the velocity and Δt is the time between the last two frames.

7.5 Elevation mapping using an ultrasound sensor

An elevation map can be used to improve navigation capabilities of both aerial and ground robots. For example, ground robots can use elevation information to plan routes that avoid obstacles. As stated in Section 4.2, no publications were found that address the problem of elevation mapping using a single airborne ultrasound sensor. The fundamental limitations of an ultrasound sensor make it hard to build an elevation map. Two issues are described below.

The first issue is the unknown true altitude z_{true} of the vehicle. When a MAV is flying above a perfectly flat floor, the measured altitude z_{sensor} is equal to the true altitude z_{true}. However, when an obstacle comes

in range of the ultrasound sensor, the measured distance z_{sensor} decreases and is not equal to the true altitude z_{true}, meaning that z_{true} cannot be derived from the ultrasound sensor directly. When the true altitude is unknown, it is impossible to determine the floor's elevation from the distance measurements using the following equation:

$$elevation = z_{true} - z_{sensor} \tag{7.21}$$

Another limitation of using an ultrasound sensor for elevation mapping is the resolution of the sensor. As described in Section 5.2.1, an ultrasound sensor is unable to obtain precise directional information about objects. Sound propagates in a cone-like manner. Due to this property, the sensor acquires entire regions of constant depth instead of discrete depth points. As a result, the ultrasound sensor can only tell there is an elevation at the measured distance somewhere within the measured cone. This makes it hard to accurately determine the contours of obstacles.

A third difficulty is related to the altitude stabilization of the AR.Drone (Section 5.3.3). The AR.Drone will change its absolute altitude when an altitude difference is detected. This complicates the detection of elevations, since it cannot be assumed the absolute altitude will remain approximately the same when flying above an object.

The remaining part of this section describes a method to extract elevation information from an airborne ultrasound sensor. The true altitude of the AR.Drone is modeled as the combination of the measured altitude z_{sensor} and the estimated elevation of the floor δ below the vehicle:

$$z_{true} = \delta + z_{sensor} \tag{7.22}$$

where the elevation δ is unknown and needs to be recovered from the ultrasound measurements.

A key part of recovering δ is a finite-state machine that describes *elevation events*. A schematic overview of the finite-state machine is given in Figure 7.6. This finite-state machine has three different states: $\{\text{NONE}, \text{UP}, \text{DOWN}\}$. State NONE indicates no significant elevation changes are occurring, state UP indicates a significant increase of elevation is detected and state DOWN indicates a significant decrease of elevation is detected.

Elevation changes are detected using a filtered second order derivative (acceleration) of the ultrasound measurement z_{sensor}. Obstacles that enter or leave the range of the ultrasound sensor result in sudden changes in measured ultrasound distance. These changes are detected when the second order derivative exceeds a certain threshold $\gamma_{elevationEvent}$, triggering an elevation event. The threshold $\gamma_{elevationEvent}$ was carefully chosen such that altitude corrections performed by the AR.Drone's altitude stabilization are not detected as being elevation events. The second order derivative is filtered by adding it to the altitude vector h^W of the EKF state vector. An elevation event ends when the sign of the second order derivative switches. This happens when the AR.Drone altitude stabilization starts to change the absolute altitude to compensate for the change in measured altitude. Now, the elevation change can be recovered by subtracting

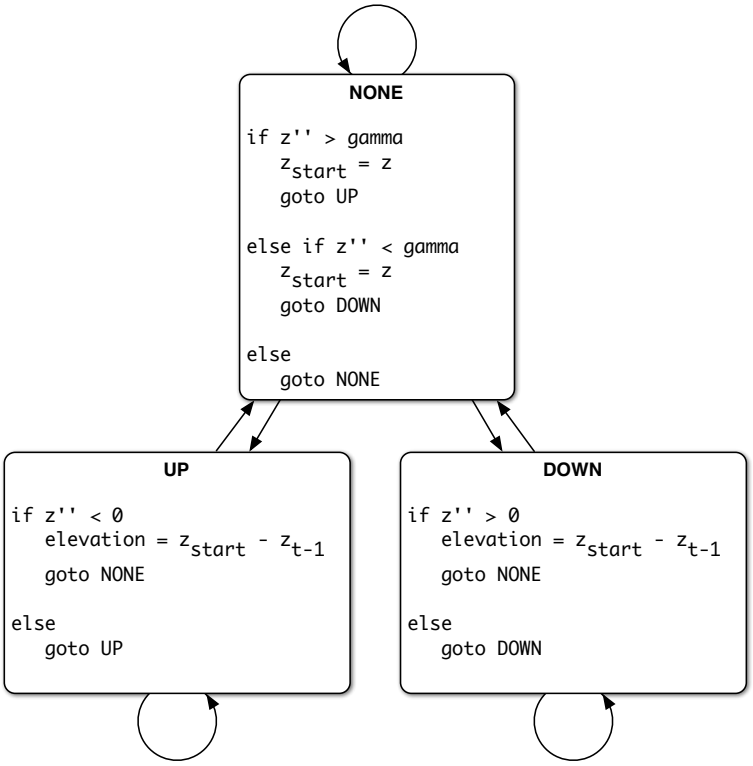


Figure 7.6: Schematic overview of the finite-state machine used for elevation mapping. z'' is the second order derivative of the ultrasound distance measurement.

the measured distance at the end of the elevation event from the measured distance before the elevation event was triggered:

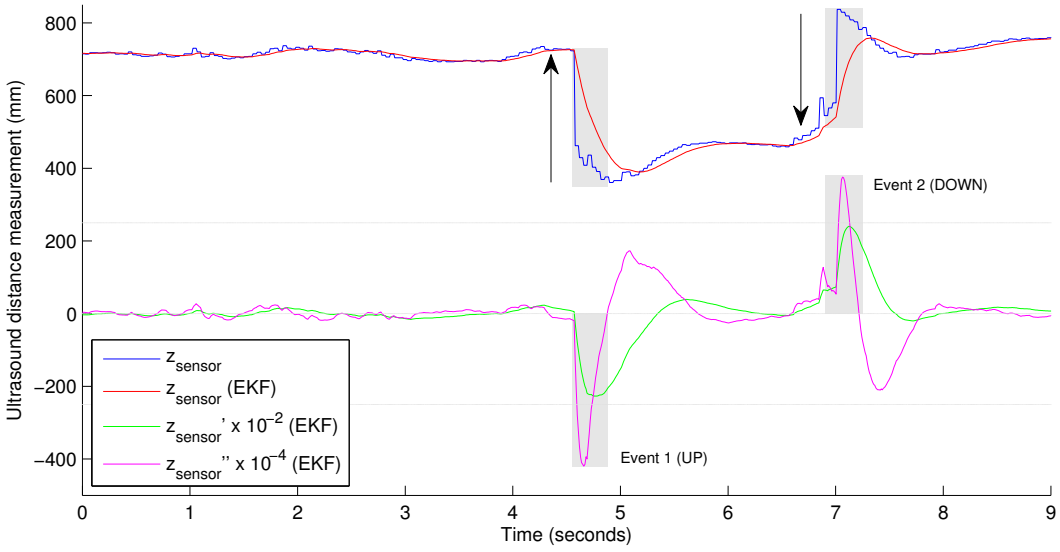
$$\Delta\delta_t = z_{sensor,t-\Delta t-1} - z_{sensor,t-1} \tag{7.23}$$

where $\Delta\delta_t$ is the elevation change during an elevation event, Δt is the duration of an event, $z_{sensor,t-\Delta t-1}$ is the distance measurement before the event started and $z_{sensor,t-1}$ is the distance measurement when the event ended. The total elevation is given by:

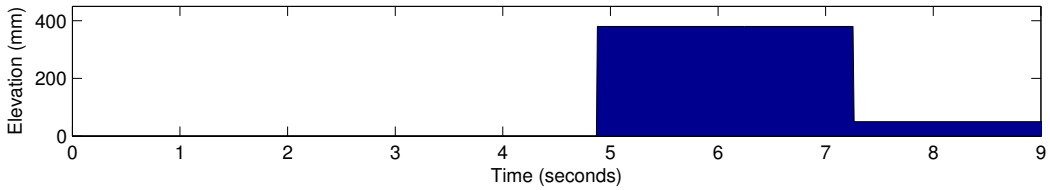
$$\delta_t = \delta_{t-1} + \Delta\delta_t \tag{7.24}$$

The elevation information is stored in a grid that is similar to the feature map described in Section 7.2.2. For each ultrasound distance measurement, elevation δ_t is computed and stored in the grid cells that correspond to the world coordinates where a line perpendicular to the AR.Drone body intersects the world plane. These world coordinates are the position where the center of the ultrasound sensor's cone hits the floor. Because the exact size of an object is unknown, the elevation is written to all grid cells within a radius $\gamma_{elevationRadius}$ around the intersection point. This process is visualized in Figure 7.8(a) and 7.8(b).

This approach may lead to cases where the size of an obstacle is overestimated in the elevation map, as can be seen in Figure 7.8(b). Therefore, an additional refinement step is added. If no elevation is measured



(a) Response



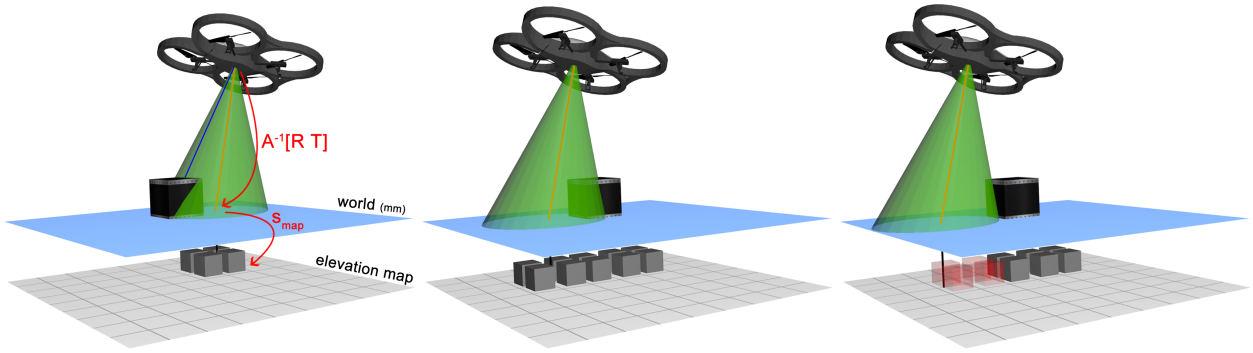
(b) Corresponding elevation

Figure 7.7: a) Response of the ultrasound sensor when flying over an object of approximately $(60, 60, 40)mm$*. The lightgray lines indicate the threshold* $\gamma_{elevationEvent}$ *and null-line. When the second order derivative (magenta line) exceeds the threshold, an event is started (lightgrey rectangle). An event ends when the derivative swaps sign. Each arrow indicates the change in elevation caused by the event. The first event increases the elevation when entering an object and the second event decreases the elevation when leaving the object. Between both events, the AR.Drone performs an altitude correction, as can be seen by the relatively slow increasement of distance. This increasement is not infomative about the elevation and is ignored by the elevation mapping method. b) Elevation* δ *below the AR.Drone over time. The elevation increases to approximately* $40cm$ *when flying above an obstacle. The elevation is decreased when the obstacle is out of the ultrasound sensor's range. There is a small error between both elevation events, resulting in a small false elevation* $(\pm 50mm)$ *after the AR.Drone flew over the obstacle and is flying above the floor again.*

($\delta_t \approx 0$), it can be assumed there is no obstacle inside the cone of the ultrasound sensor. Using this assumption, all grid cells within the cone can be resetted to zero elevation and locked to prevent future changes. This refinement step is visualized Figure 7.8(c). The radius of the cone is computed using the following equation:

$$r = tan(\alpha_{ultrasound} \times z_{sensor}) \tag{7.25}$$

where r is the radius of a cone of height z_{sensor} and $\alpha_{ultrasound}$ is the opening angle of the ultrasound sensor.



(a) Obstacle enters range (b) Obstacle in range (c) Obstacle out of range

Figure 7.8: Overview of the elevation map updates. The green cone indicates the range of the ultrasound sensor. The red line inside the cone represents the center of the cone, perpendicular to the AR.Drone's body. In 7.8(a) and 7.8(b) an elevation is measured. All grid cells within a radius $\gamma_{elevationRadius}$ around the center of the cone (red line) are updated to store the measured elevation. 7.8(c) Describes the refinement step. When no elevation is measured, all grid cells within the cone (red cubes) are resetted to zero elevation.

8 Results

In this chapter, the performance of both the simulation model and visual SLAM method are analyzed in a series of experiments. In Section 8.1, the USARSim simulation model is validated. I like to thank Carsten van Weelden for performing these experiments. He flew a set of maneuvers with the actual AR.Drone and simulated AR.Drone. The differences between the maneuvers are studied in detail. In Section 8.2, the accuracy of the estimated position is evaluated. A set of 8-shapes is flown with the AR.Drone and the estimated positions are compared against the groundtruth from a laser rangefinder. In Section 8.3, the proposed pose recovery approach is benchmarked against other pose recovery approaches. In Section 8.4, the influence of the camera resolution is investigated. Finally, in Section 8.5 the accuracy of the elevation map is evaluated in terms of elevation error and the error in the estimated size of objects.

8.1 Simulation model

To evaluate the USARSim simulation model, a set of maneuvers is flown with the actual AR.Drone and simulated AR.Drone. The differences between the maneuvers are studied in detail. To enable multiple repetitions of the same maneuver it is described as a set of time points (milliseconds since initialization) each coupled to a movement command. Orientation, altitude and horizontal speed are recorded at a frequency of 200Hz during the maneuvers. These are gathered through the AR.Drone's internal sensors and the onboard algorithms, which are also used by the controller to operate the drone. The filtered output of the gyroscope is used to estimate the orientation. The filtered output of the ultrasound distance sensor is used to estimate the altitude. The onboard velocity estimates are used as the horizontal (linear and lateral) speeds. The simulator has equivalent sensors. In addition, simulation can provide ground-truth data. Also for the real maneuvers an attempt was made to generate ground truth via an external reference system; the movements were recorded with a synchronized video system consisting of four firewire cameras, capturing images at 20 frames per second at a resolution of 1024×768 pixels. The position of the AR.Drone in each frame has been annotated by hand.

Corresponding to NIST guidelines [68] a set of experiments of increasing complexity was performed. For the AR.Drone four different experiments were designed. The first experiment is a simple hover, in which the drone tries to maintain its position (both horizontal and vertical). The second experiment is linear movement, where the drone actuates a single movement command. The third experiment is a small horizontal square. The last experiment is a small vertical square.

Hovering

Quadrotors have hovering abilities just like a helicopter. The stability in maintaining a hover depends on environmental factors (wind, underground, aerodynamic interactions) and control software. If no noise model is explicitly added, the USARSim model performs a perfect hover; when no control signal is given the horizontal speeds are zero and the altitude stays exactly the same.

For the AR.Drone, this is a good zero-order model. One of the commercial features of the AR.Drone is its ease of operation. As part of this feature it maintains a stable hover when no other commands are given, which is accomplished by a visual feedback loop. So, the hovering experiment is performed indoor with an underground chosen to have enough texture for the onboard velocity estimation algorithm.

As experiment the AR.Drone maintained a hover for 35 seconds. This experiment was repeated 10 times, collecting 60.000 movement samples for a total of 350 seconds. Over all samples the mean absolute error in horizontal velocity (the Euclidean norm of the velocity vector) was $0.0422m/s$ with a sample variance of $0.0012m^2/s^2$. The distribution of the linear and lateral velocity components was obtained from the samples.

From the velocity logs the position of the AR.Drone during the 35-second flight was calculated. The mean absolute error of the horizontal position was $0.0707m$ with a sample variance of $0.0012m^2$.

Horizontal movement

In this experiment the AR.Drone is flown in a straight line. It is given a control pulse (i.e., pitch) with a constant signal for 5 different time periods: $0.1s$, $1s$, $2s$, $3s$, and $5s$. Each pulse is followed by a null signal for enough time for the AR.Drone to make a full stop and a negative pulse of the same magnitude for the same period, resulting in a back and forth movement. In Figure 8.1 the red line shows the control signal, the blue line the response (i.e., velocity) of the AR.Drone. The experiment was repeated for 5 different speeds. The control signal s specifies the pitch of the AR.Drone as a factor (between 0 and 1) of the maximum absolute tilt θ_{max}, which was set to the default value[1]. The trials were performed with the values of 0.05, 0.10, 0.15, 0.20, 0.25 for the control signal s.

Robots in USARSim are controlled with a standardized interface, which uses SI units. A robot in USARSim expects a DRIVE command with a speed in m/s and not the AR.Drone native signal s. Thus in order to fly comparable trials the relation between the AR.Drone's angle of attack α and the corresponding velocity v has to be investigated. When flying straight forward in no-wind conditions, the angle of attack α is equivalent with the pitch θ. In order to do this the samples from the logs where the AR.Drone has achieved maximum velocity have to be selected. Closer inspection of the velocity logs show that in each trial there is still constant increase of velocity for the first three pulses. For the last two pulses there is obvious plateauing, which indicates that the last two seconds of the five-second pulses is a good indication for

[1] ARDrone firmware (1.3.3)

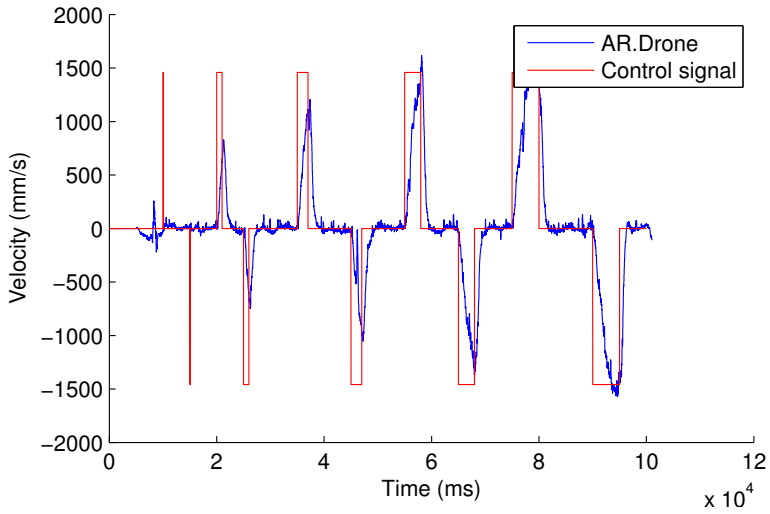


Figure 8.1: Velocity of the real AR.Drone (blue) on a number of control pulses (red) with a pitch of $s = 0.15 \times \theta_{max}$.

the maximum velocity. Therefore the velocity at those last two seconds was used to compute mean absolute speeds $\bar{v}$, which are combined with the mean absolute pitch $\bar{\theta}$ as measured by the MEMS gyroscope. The estimates for $\bar{v}$ and pitch $\bar{\theta}$ are presented in Table 8.1 together with their standard deviation. Extrapolating the mean pitch $\bar{\theta} \simeq 7.5^o$ at control value $s = 0.25$ to the maximum control signal gives an indication of the AR.Drone's maximum pitch $\theta_{max} \simeq 30^o$ value. For typical usage, the angle of attack never exceeds 12^o degrees.

	Control signal s				
	0.05	0.10	0.15	0.20	0.25
$\bar{v}$ (m/s)	0.4044	0.6284	1.4427	1.7587	2.2094
σ_v (m/s)	0.096	0.226	0.070	0.126	0.165
$\bar{\theta}$ (deg)	1.4654	2.9025	4.1227	5.7457	7.4496
σ_θ (deg)	0.455	0.593	0.482	0.552	0.921

Table 8.1: Averaged velocity $\bar{v}$ measured at the end of a 5 seconds pulse of the control signal s, including the corresponding pitch $\bar{\theta}$ as measured by the gyroscope.

To convert the AR.Drone's control signal s to USARSim commands v a least-squares fit through the points of Table 8.1 is made for the linear function $v = c \cdot \theta$, which gives $c = 0.2967$. Equation 8.1 gives the final conversion of a control signal s to a velocity v in m/s given the AR.Drone's maximum pitch θ_{max} in degrees.

$$v = 0.2967 \cdot s \cdot \theta_{max} \tag{8.1}$$

The USARSim model has a parameter P_θ for calculating the angle (radian) given the velocity, which is the value $\hat{P}_\theta = 0.057$, as used in subsequent simulations.

The next experiment checks the acceleration of the real and simulated AR.Drone. An estimate describes how quickly the AR.Drone's controller changes its pitch to match the commanded pitch and how well it can keep it. Samples are selected from $100ms$ after the start of the $2s$, $3s$ and $5s$ pulses till the commanded pitch has been reached. This corresponds to the time-span between which the AR.Drone has started to act on the change in the control signal until it reaches the commanded pitch. The result is illustrated in Figure 8.2.

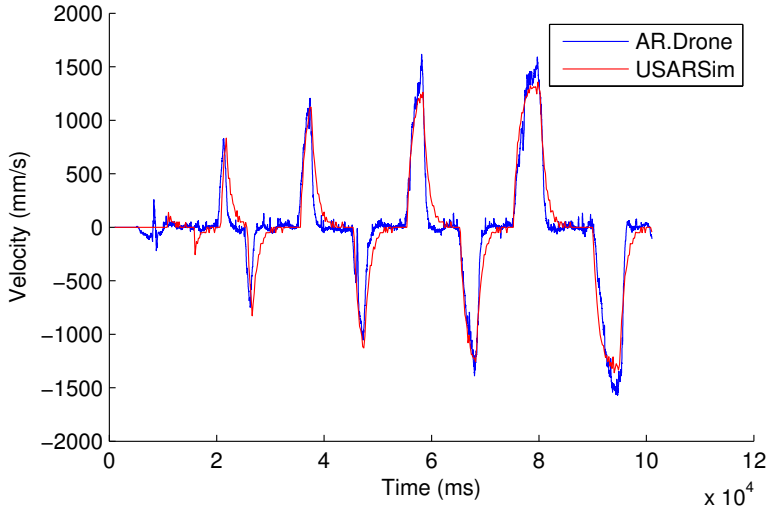


Figure 8.2: Velocity of the real (blue) and simulated AR.Drone (red) on the same control pulses as shown in Figure 8.1.

As one can see, the acceleration shows for the real and simulated AR.Drone nearly the same slope. The deceleration of the simulated AR.Drone is slightly lower. In the real AR.Drone the feedback loop (based on the onboard visual odometry of the ground camera) actively decelerates the system. Overall, the dynamic behavior of the simulator closely resembles the dynamic behavior of the real system.

8.2 Position accuracy

An experiment has been carried out to evaluate the accuracy of the AR.Drone's estimated position. This experiment is performed above a texture-rich floor (Section 8.2.1), a texture-poor floor (Section 8.2.2) and a floor that resembles the circumstances encountered during the IMAV competition (Section 8.2.3). The position can be estimated using different methods and the accuracy of all these methods are evaluated. To ensure fair comparison of the different estimation methods, a flight was recorded (Section 6.2) and played back to generate the same input for all methods.

The first position estimation method integrates the AR.Drone's onboard velocity (AV) measurements to estimate the position (Section 7.1). Recent versions of the AR.Drone firmware calculate an estimated position onboard[2], which is used as second estimation method. The third position estimation method uses the visual odometry method presented in this thesis (Section 7.4). The final estimation method uses the AR.Drone's onboard velocity (AV) for frequent position updates and the presented localization method (Section 7.3) to localize against the map that is constructed during flight.

The estimated position is compared against the groundtruth (real) position to calculate the accuracy of the estimated position. Much attention was paid to collect accurate groundtruth of the AR.Drone's position. A high-end laser rangefinder (LRF)[3] was used to determine the position of the AR.Drone. The LRF accurately measures distance to objects. At a range of $10m$, the accuracy is $\pm 30mm$. A window was placed on the measurements to filter out measurements that originate from the environment. All remaining distance measurements are assumed to originate from the AR.Drone. The mean position of the filtered distance measurements is used as center position of the AR.Drone. Furthermore, a white cylinder with a diameter of $9cm$ and a height of $37cm$ was added on top of the AR.Drone to improve its visibility for the LRF.

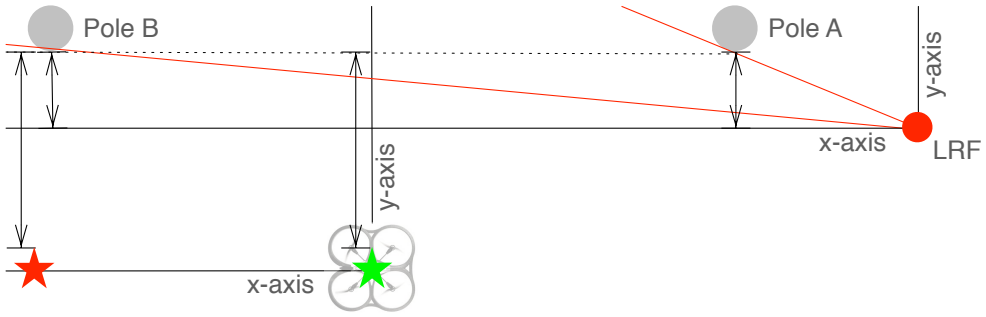


Figure 8.3: The alignment of the laser rangefinder and the AR.Drone. The laser rangefinder is aligned against two poles. The takeoff position is marked with a green star. The red star indicates the marked heading direction of the AR.Drone. A laser pointer is used to align the heading direction of the AR.Drone to the marker position.

The LRF measures the AR.Drone's position in its own reference frame. Therefore, the AR.Drone's reference frame has to be accurately aligned with the LRF's reference frame. This is achieved by rotating the AR.Drone before takeoff, such that it points in exactly the same direction as the LRF. The following method was used to align the LRF and the AR.Drone (Figure 8.3). First, the LRF is aligned against two poles in the environment. The LRF is rotated until both poles obtained an equal y-distance in the LRF's reference frame. Now, the LRF's x-axis is parallel to the poles. Next, the AR.Drone's takeoff position and orientation are determined. The takeoff position is marked on the floor (green star). Another position is marked (red star)

[2]Firmware 1.7.4

[3]Hokuyo UTM-30LX, `http://www.hokuyo-aut.jp/02sensor/07scanner/utm_30lx.html`

to indicate the desired heading direction of the AR.Drone. Both marked positions are parallel to the LRF's x-axis and the two poles. Finally, a laser pointer is used to align the heading direction of the AR.Drone to the marked position.

8.2.1 Texture-rich floor

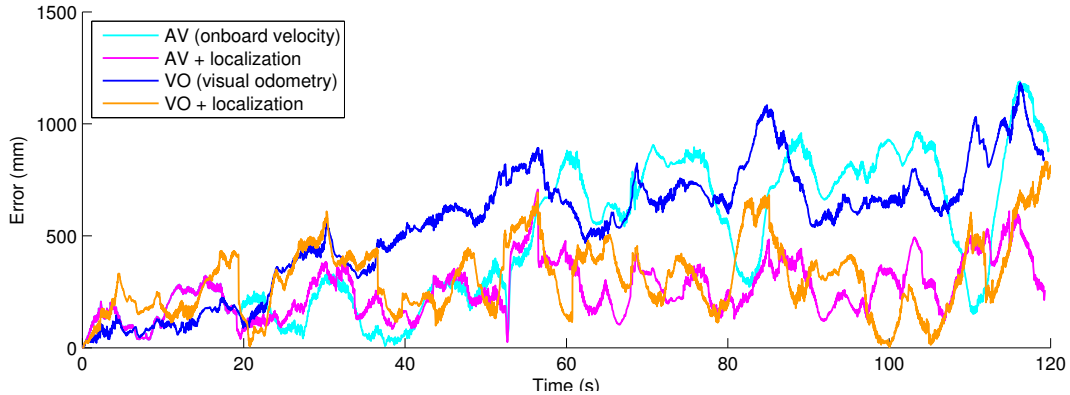
The first experiment was performed above a texture-rich floor (Figure 8.11), to benchmark the methods in preferable circumstances. Magazines were spread out on the floor to provide sufficient texture for the computer vision algorithms. The AR.Drone flew three 8-shapes above the magazines. An 8-shape was chosen to capture an intermediate loop.

Figure 8.4: Photo of the texture-rich floor used for the first experiment. Magazine are spread out on the floor to provide sufficient texture. The dimensions of the 8-shape are approximately $5m \times 2.5m$.

Method	Mean absolute error (mm)	Mean relative error (percentage of trajectory length)
AP (onboard position)	476	0.714%
AV (onboard velocity)	477	0.715%
AV + localization	260	0.390%
VO (visual odometry)	552	0.828%
VO + localization	324	0.486%

Table 8.2: Errors made by the position estimation methods when flying above a texture-rich floor.

The results of this experiment can be found in Table 8.2 and Figures 8.5 (errors), 8.6 (trajectories) and 8.7 (texture map). For the encountered circumstances, all methods are able to estimate the position quite accurately. Both the onboard velocity (AV) method and the onboard position (AP) method produce exactly the same trajectory and error, which implies the onboard position estimate (AP) is based on the velocity that is estimated by the AR.Drone. Therefore, the onboard position estimate is omitted from all plots.



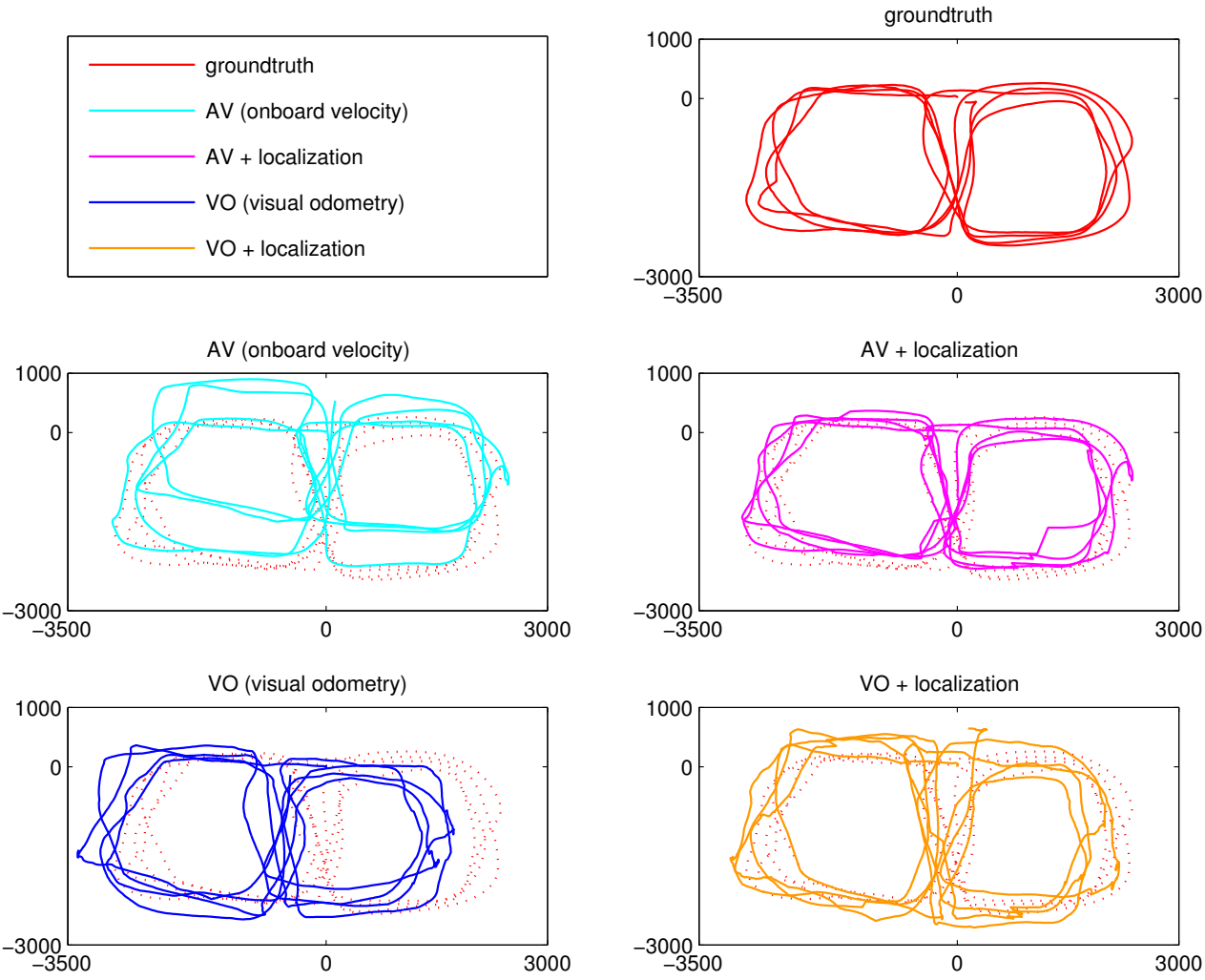


Figure 8.6: Estimated trajectories of the different position estimation methods. The AR.Drone flew three 8-shapes above a texture-rich floor.

Figure 8.7: Texture map created by the texture mapping method (Section 7.2.1). The map is constructed during the first 8-shape. The position estimates are based on the onboard velocity (AV) and the visual localization method (Section 7.3).

The onboard velocity (AV) slightly outperforms the visual odometry (VO) method presented in Section 7.4. An explanation could be that the AR.Drone features an additional optical flow algorithm that is used when the appearance-based method is performing badly. From Figure 8.6 can be seen that the onboard velocity (AV) method has the largest error in the x-direction. Around $54s$ the error is increasing rapidly. Analysis of the dataset pointed out that a small part of a magazine and a large part of a white line were in range of the camera. This resulted in a velocity that was estimated in the wrong direction along the white line, which can be recognized by the small green loop on the right side of Figure 8.6.

The visual odometry (VO) method has the largest error in the y-direction. Around $80s$ the error is increasing rapidly. Analysis of the dataset pointed out that the magazines were out of the camera's range. During this period the velocities could not be updated.

A combination of the onboard velocity (AV) with localization against the map outperforms all other methods. With localization against the map, the error is reduced significantly. From Figure 8.5 can be seen that the maximum error is not exceeding the error made during the first 8-shape. The repetitive error pattern suggests that the error of the estimated position largely depends on the quality (accuracy) of the map, if localization can be performed regularly.

The texture map (Figure 8.7) shows that the AR.Drone's onboard (uncontrollable) white balancing is reducing the contrast of the magazines, which affects the performance of the feature detector. This is a source of error for the visual odometry (VO) algorithm. Furthermore, errors in the ultrasound and IMU measurements are other sources that result in errors in the estimated velocities.

8.2.2 Texture-poor floor

The experiment is repeated above a floor with low texture, to benchmark the methods in difficult circumstances. Therefore, all magazines (Figure 8.11) are removed, resulting in a green floor with few white lines. The AR.Drone flew a single 8-shape.

Method	Mean absolute error (mm)	Mean relative error (percentage of trajectory length)
AP (onboard position)	1294	6.597%
AV (onboard velocity)	1292	6.586%
VO (visual odometry)	1126	5.740%

Table 8.3: Errors made by the position estimation methods when flying above a texture-poor floor.

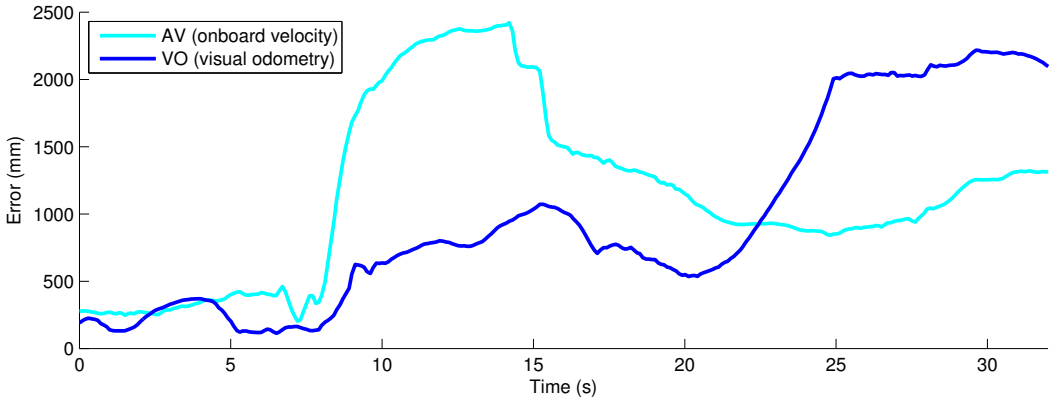


Figure 8.8: Errors made by the position estimation methods when flying above a texture-poor floor.

The results of this experiment can be found in Table 8.3 and Figures 8.8 (errors), 8.9 (trajectories) and 8.10 (texture map). For the encountered circumstances, all methods are unable to estimate the position accurately. Again, both the onboard velocity (AV) method and the onboard position (AP) method produce exactly the same trajectory. Localization was not possible and therefore omitted from the plots.

The visual odometry (VO) method slightly outperforms the onboard velocity (AV) method. One would expect that the AR.Drone's additional optical flow method outperforms appearance-based methods. From Figure 8.9 can be seen that the onboard velocity (AV) method fails to correctly deal with the lines on the floor. The onboard velocity (AV) methods measures a velocity in the wrong direction along the lines, resulting in a large error. One would expect that the onboard aerodynamic model should correct this (i.e., the angle of the AR.Drone can be used to determine the correct direction).

From Figure 8.9 can be seen that the visual odometry (VO) method underestimates the velocities. At places where the AR.Drone changes direction, the method was able to track sufficient features (due to the

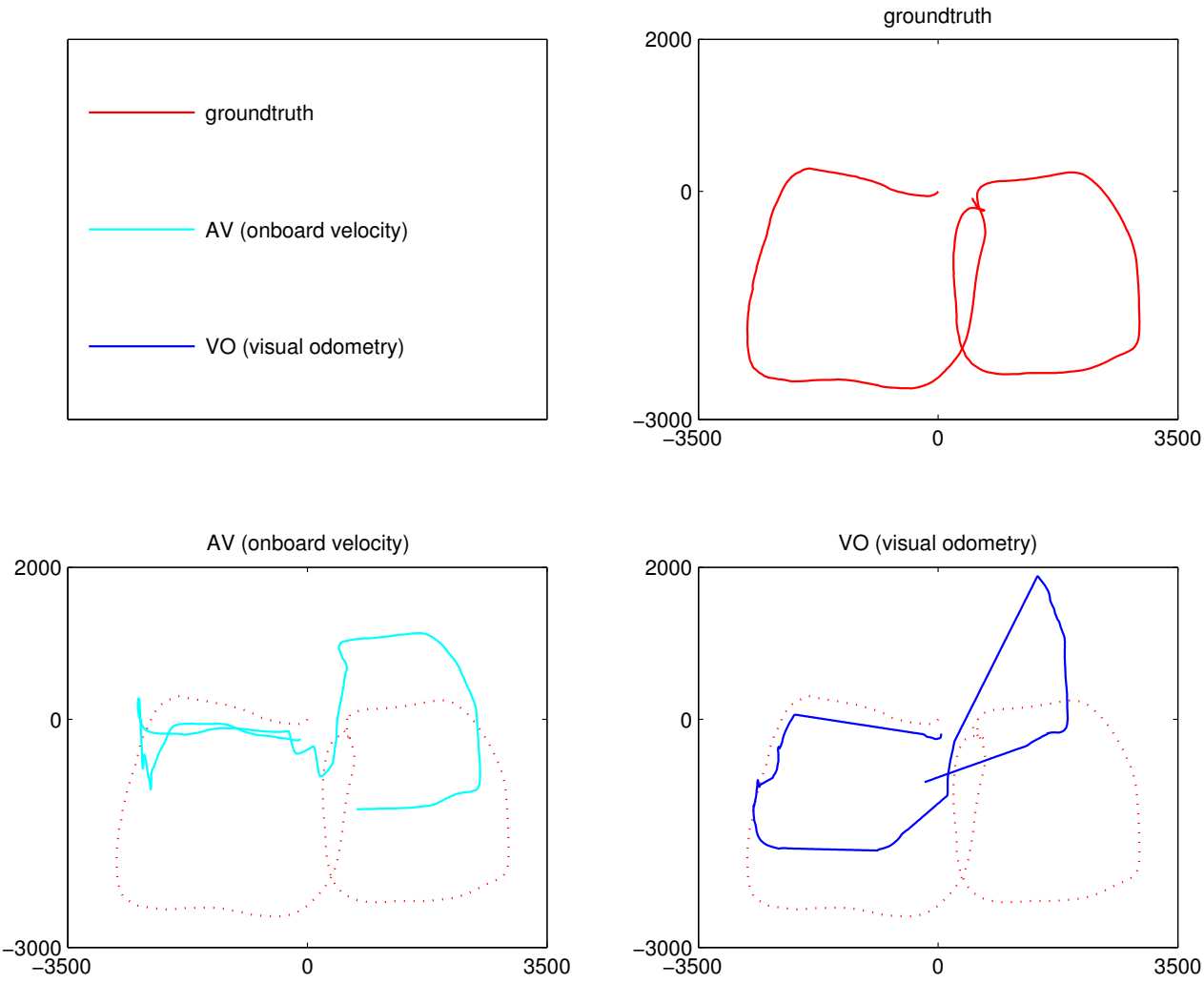


Figure 8.9: Estimated trajectories of the different position estimation methods. The AR.Drone flew three 8-shapes above a texture-poor floor.

presence of corners) and estimate the velocities. At these places, the AR.Drone's velocity was below the average speed due to de-accelerations. Along most straight parts of the trajectory insufficient features were tracked and the (higher) velocity could not be updated. Therefore, the lower velocities during the direction changes are maintained when flying along the straight parts of the trajectory.

8.2.3 IMAV circumstances

The experiment is repeated in a sports hall, to benchmark the methods in circumstances encountered during the IMAV competition. This environment closely resembles the environment of the IMAV2011 indoor competition. The trajectory of the AR.Drone is increased to approximately $12 \times 2.5m$ in order to match the trajectory of the IMAV competition. This trajectory is flown three times. Furthermore, the flight altitude of

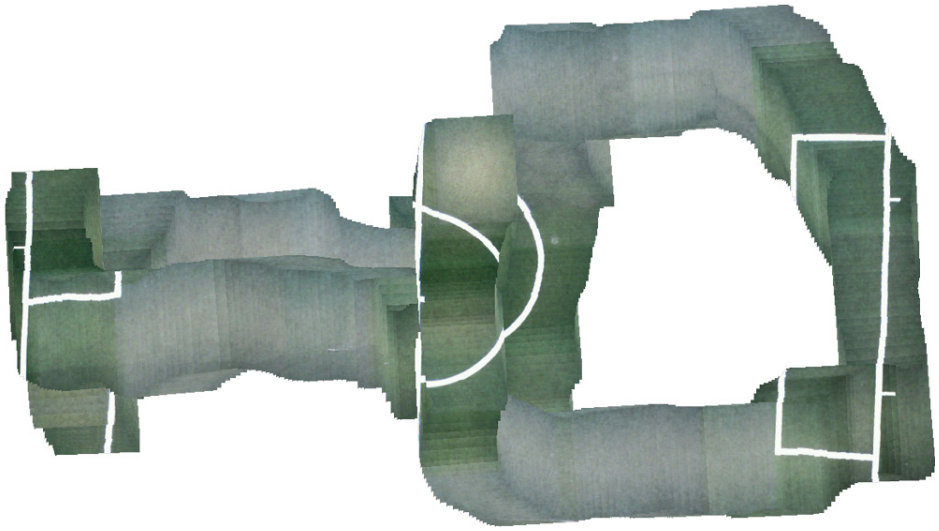


Figure 8.10: Texture map created by the texture mapping method (Section 7.2.1). The position estimates are based on the onboard velocity (AV).

the AR.Drone is increased to $2m$. This allows the camera to capture sufficient visual clues (i.e., lines and intersections) to perform for visual odometry.

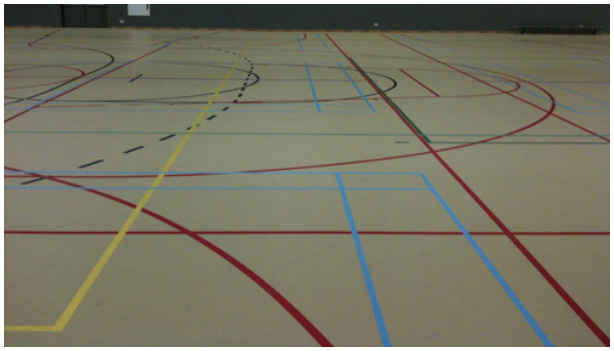


Figure 8.11: Photo of the sports floor, which resembles the circumstances encountered during the IMAV competition.

Method	Mean absolute error (mm)	Mean relative error (percentage of trajectory length)
AP (onboard position)	988	0.960%
AV (onboard velocity)	988	0.960%
AV + localization	461	0.452%
VO (visual odometry)	4251	4.172%
VO + localization	1158	1.136%

Table 8.4: Errors made by the position estimation methods when flying in circumstances encountered during the IMAV competition.

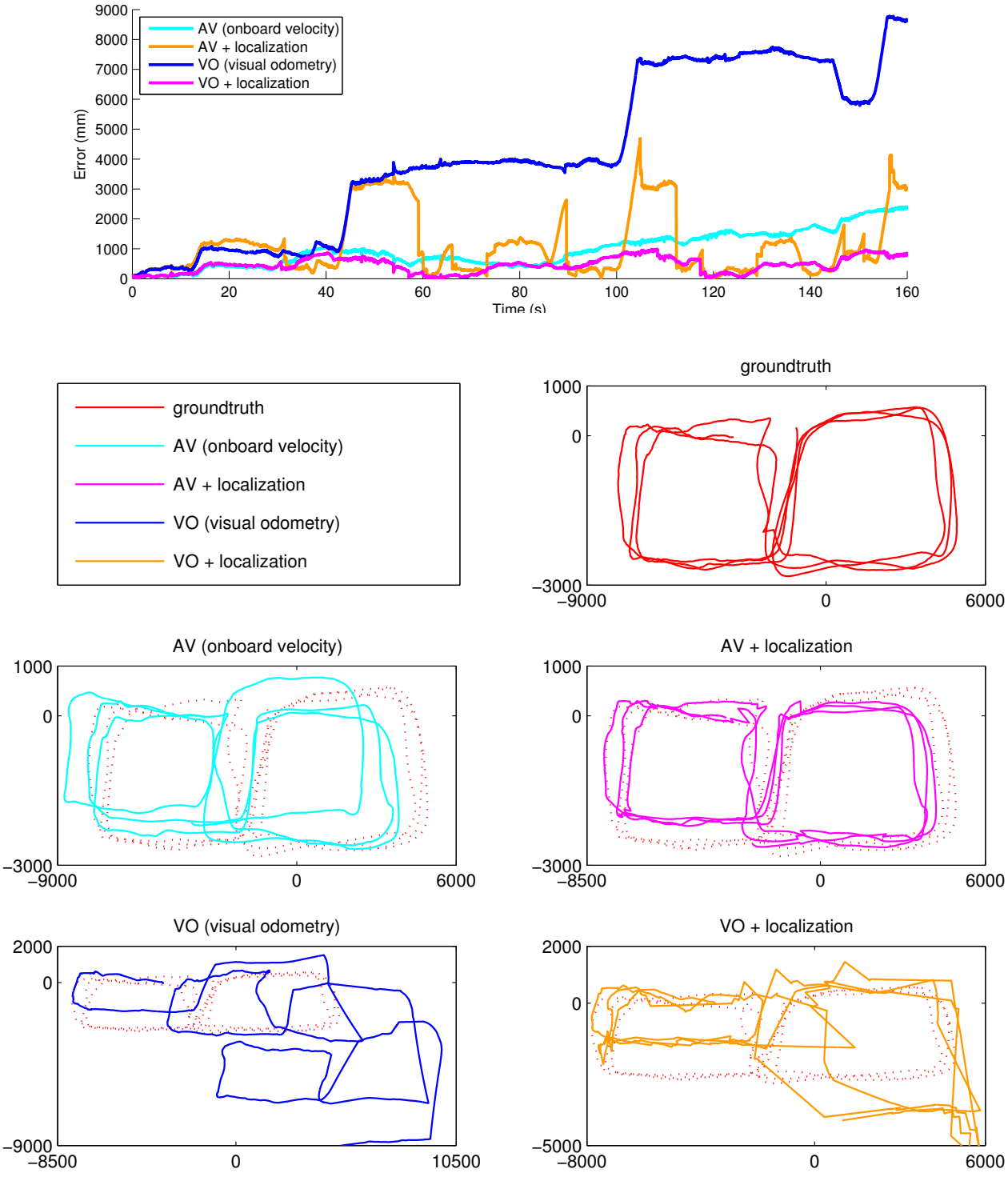


Figure 8.13: Estimated trajectories of the different position estimation methods. The AR.Drone flew three 8-shapes above a sport floor. All units are in mm.

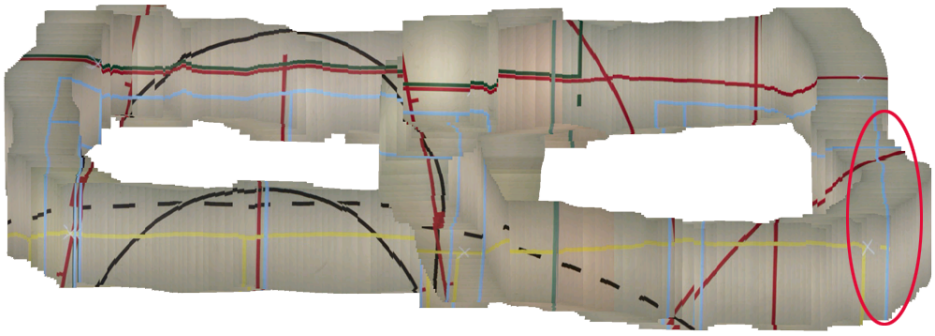


Figure 8.14: Texture map created by the texture mapping method (Section 7.2.1). The map is constructed during the first 8-shape. The position estimates are based on the onboard velocity (AV) and the visual localization method (Section 7.3). The red ellipse indicates an area for which the visual odometry method is unable to estimate the velocity.

The results of this experiment can be found in Table 8.4 and Figures 8.12 (errors), 8.13 (trajectories) and 8.14 (texture map). For the encountered circumstances, not all methods are able to estimate the position accurately. The AR.Drone's onboard velocity (AV) method is able to estimate the position quite well, with an average error of $1m$. The error increases steadily over time due to drift. In Figure 8.13 can be seen that this method underestimates the distance along the left vertical path of the trajectory. During this path, the camera observes parallel lines, for which it is hard to recover the velocity from vision. Similar behavior is visible for the proposed visual odometry method (VO).

The proposed visual odometry method (VO) is having more difficulties. From Figure 8.12 can be seen that the error of the visual odometry method (blue line) is increasing rapidly at certain moments in time. The repetitive behavior suggests the sudden increase in error is correlated to a certain position of the floor. Investigation showed that the visual odometry method was unable to estimate the velocity within a specific area (red ellipse in Figure 8.14). In this area, the floor has too few visual clues.

When using localization against the map, the errors decrease significantly. The localization method is able to recover the AR.Drone's position from the map, which reduces the drift. However, due to the lack of a map optimization method, the map contains errors. These errors affect the accuracy of a localization. Therefore, a repetitive error pattern is visible. For each 8-shape, the errors are approximately bounded to the errors made during the first 8-shape.

These results confirm that the localization method is able to deal with circumstances encountered during the IMAV competition. A feature-based approach to visual odometry is insufficient to estimate the velocity above all areas of a sport floor. Therefore, an additional appearance-based method (like the AR.Drone uses) is required to estimate the velocity at all times.

8.3 Accuracy w.r.t. pose recovery approaches

Section 7.3.1 proposed a new method to robustly recover the translation and rotation between two frames (for visual odometry) or a frame and a map (for localization). This method features two improvements. Instead of computing a full perspective, affine or Euclidean transformation, a more pragmatic approach is used. Furthermore, the proposed approach uses covariance, instead of the number inliers, as quality measure. In this experiment, the proposed pose recovery approach is benchmarked against other pose recovery approaches. The proposed approach has been validated with the proposed covariance quality measure and with the classic *inliers based* quality measure, in order to validate the contribution of the proposed quality measure.

An approach similar to the position accuracy experiment from Section 8.2 is used. The positions are estimated using the Visual Odometry (VO) method from Section 7.4, which relies on pose recovery to determine the velocity. The recorded dataset of the texture-rich floor (Section 8.2.1) is used as input.

The pose recovery approach is benchmarked against OpenCV's robust *Euclidean transformation*, *affine transformation* and *perspective transformation*. The **Euclidean transformation** is computed using OpenCV's *estimateRigidTransform*[4] with the parameter *fullAffine* off to compute a Euclidean transformation. The translation vector from the transformation matrix is used as velocity estimate. The maximum allowed reprojection error is set to $10mm$. Additional filtering is required to remove faulty transformations. A recovered transformation is rejected if the percentage of inliers is below 50% or if the translation is too large ($T \geq 500mm$). The **affine transformation** is computed using OpenCV's *estimateRigidTransform*[5] with the parameter *fullAffine* on. The **perspective transformation** is computed using OpenCV's *findHomography*[6] with a RANSAC-based robust method. The translation vector from the homography is used as velocity estimate.

The results of this experiment can be found in Table 8.5 and Figure 8.15. These results show a relation between the degrees of freedom of an approach and the errors in the estimated positions. Reducing the degrees of freedom results in more robustness against noise. This corresponds with the observations made by Caballero et al (Section 4.1).

OpenCV's pose recovery approaches use the percentage of inliers as quality measure. However, the percentage of inliers may not reflect the actual quality of a recovered pose. For example, a faulty transformation (pose) can maximize the number of inliers, as described in Section 7.3.1. These faulty transformations

[4] http://opencv.itseez.com/modules/video/doc/motion_analysis_and_object_tracking.html?highlight=estimateRigidTransform

[5] http://opencv.itseez.com/modules/video/doc/motion_analysis_and_object_tracking.html?highlight=estimateRigidTransform

[6] http://opencv.itseez.com/modules/video/doc/motion_analysis_and_object_tracking.html?highlight=findhomography

Method	Mean absolute error (mm)	Mean relative error (percentage of trajectory length)
visual odometry (VO)	552	0.828%
visual odometry (VO) - inliers based	734	1.101%
visual odometry (VO) - Euclidean transform	967	1.449%
visual odometry (VO) - affine transform	3784	5.672%
visual odometry (VO) - perspective transform	8923	13.375%

Table 8.5: Errors made by the visual odometry (VO) position estimation method when using different pose recovery approaches.

produce sudden changes in error, as can be seen in Figure 8.15.

The proposed pose recovery approach outperforms the other pose recovery approaches. The best performance is achieved when using the proposed covariance based quality measure. When the confidence based quality measure is replaced with classic inliers based quality measure, the performance decreases. However, the proposed method still outperforms the other methods. This improvement with respect to the Euclidean transform can be dedicated to the pragmatic approach, which reduces the degrees of freedom from four (translation and rotation) to two (translation). The additional improvement gained by the proposed approach can be dedicated to the covariance based quality measure. This quality measure is able to estimate the quality of a recovered pose more accurately, resulting in recovered poses that contain less errors. This is reflected by the reduced number of sudden changes in the plot.

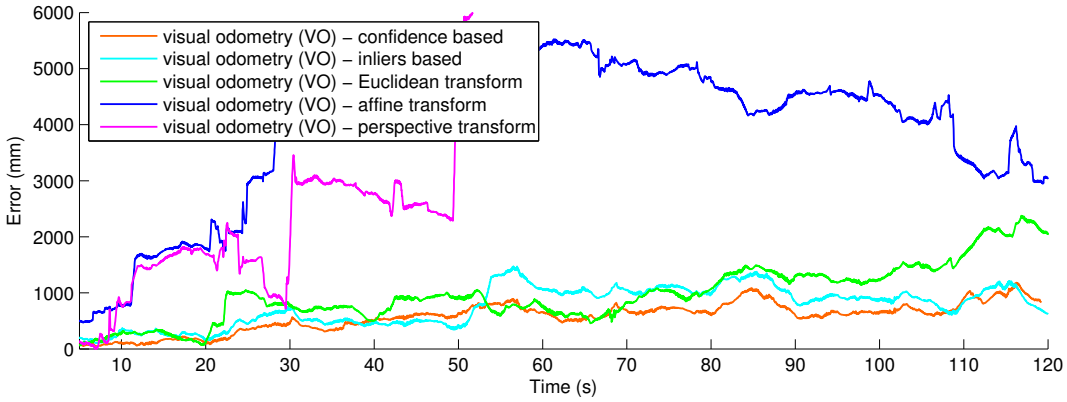


Figure 8.15: Errors between the estimated positions and the groundtruth.

8.4 Accuracy w.r.t. camera resolution

The AR.Drone is equipped with a low resolution down-looking camera. The recent developments in miniaturizing high-resolution cameras enables MAVs to be equipped with high-resolution cameras. This poses the question if the proposed method benefits from an increased image resolution.

In this experiment, the accuracy of the estimated position is evaluated for various camera resolutions. An approach similar to the position accuracy experiment from Section 8.2 is used. The positions are estimated using the Visual Odometry (VO) method from Section 7.4. The recorded dataset of the texture-rich floor (Section 8.2.1) is used as input. The experiment is repeated for different camera resolutions, which are simulated by downscaling the original camera images. The camera calibration matrix is scaled accordingly.

The experiment is repeated with a simulated AR.Drone in USARSim, which enables to both increase and decrease the resolution of the camera. Attention was paid to mimic the AR.Drone's camera[7].

The results of the real AR.Drone can be found in Table 8.6 and the results of the simulated AR.Drone can be found in Table 8.7. For each camera resolution, the number of processed frames is counted. Not all frames can be (accurately) related to its previous frame to determine (extract) the velocity, which is indicated by the third column. The fourth column describes the mean error of the estimated positions. In Figure 8.16, the relation between image resolution and accuracy of the estimated position is plotted. In Figure 8.17, the position error is plotted over time.

Resolution	Nr. frames processed	Nr. frames velocity extracted	Mean error (mm)
33.3%	2275	213	11742
66.6%	2239	1346	2166
90.0%	2259	1847	**531**
100.0%	2168	1945	579

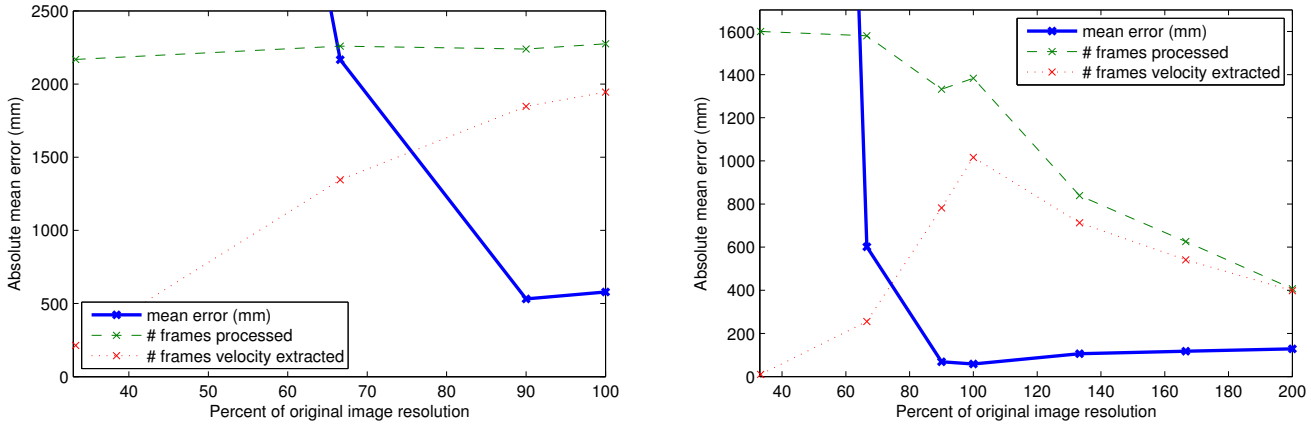
Table 8.6: Real AR.Drone. Mean error of the estimation position for different camera resolutions. For each camera resolution, the number of processed frames is counted. A part of these frames can be used to extract the velocity of the AR.Drone.

From Table 8.6 can be seen that the original camera resolution is sufficient for frequent velocity updates. Out of the 2168 processed frames, 1945 frames (90%) could be used to extract the velocity using the Visual Odometry method. When the image resolution is slightly decreased to 90% of the original resolution, more frames were processed. Algorithm such as the SURF feature extractor requires less processing time, allowing more frames to be processed. However, only 82% of these frames could be used to extract the velocity. When the resolution decreases further, the number of processed frames remains stable, which

[7] A previous article [62] describes the influence of white balancing on image stitching and how to mimic the real images as close as possible.

Resolution	Nr. frames processed	Nr. frames velocity extracted	Mean error (mm)
33.3%	1600	10	15302
66.6%	1580	266	601
90.0%	1332	782	69
100.0%	1383	1016	**58**
133.3%	839	713	105
166.6%	626	541	117
200.0%	407	398	128

Table 8.7: Simulated AR.Drone (USARSim). Mean error of the estimation position for different camera resolutions. For each camera resolution, the number of processed frames is counted. A part of these frames can be used to extract the velocity of the AR.Drone.



(a) Real AR.Drone

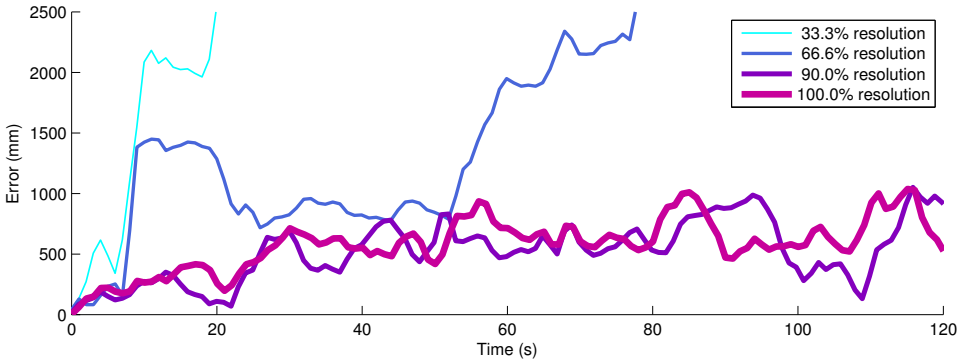
(b) Simulated AR.Drone (USARSim)

Figure 8.16: Mean error of the estimated position with respect to the image resolution. For each resolution, the number of processed frames is counted (green line). A part of these frames can be used to extract the velocity of the AR.Drone (red line).

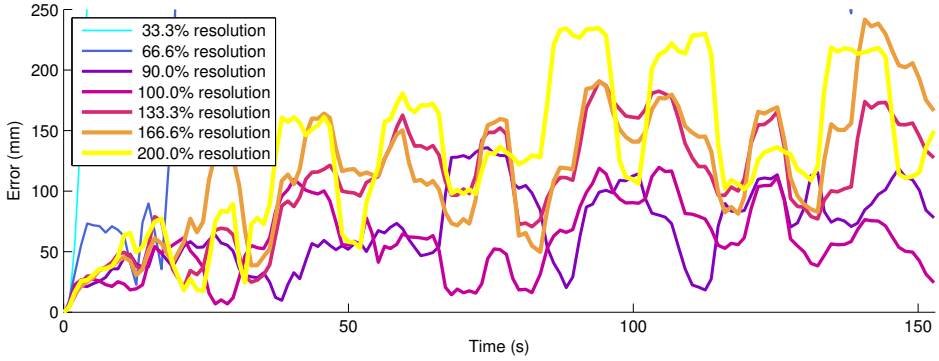
indicates the processing time is not significantly decreasing. However, the number of frames that could be used to extract the velocity is decreasing. This indicates that the camera images with lower resolution are less useful for the Visual Odometry method. This is reflected by the mean error of the different camera resolutions. In general, a lower camera resolution results in a larger error in the estimated position (Figure 8.16(a)). An exception is the camera resolution of 90%, which has a slightly lower error than the original resolution. This exception is probably caused by a coincidental drift that reduces the error.

The results of the simulated AR.Drone can be found in Table 8.7 and Figure 8.16(b). The overall mean errors are smaller than the real AR.Drone, due to the lack of noise generated by the IMU and ultrasound

sensors. Similar to the real AR.Drone, a reduced resolution results in more frames being processed, but less frames that could be used to extract the velocity. When the resolution increases, the number of processed frames decreases due to the increased computational time per frame. However, the percentage of frames that could be used to extract the velocity increases. This indicates that the camera images with a higher resolution are more useful for the Visual Odometry method. For the original resolution, 62% of the processed frames could be used to extract the velocity. When the resolution is doubled, 96% of the frames could be used to extract the velocity. One would expect that an increased image resolution would result in a smaller error. However, due to the increased computational time per frame, less frames can be processed resulting in less frequent velocity estimates. This results in a mean error that is not smaller than the original resolution.



(a) Real AR.Drone



(b) Simulated AR.Drone (USARSim)

Figure 8.17: Error of the estimated position for different image resolutions.

The experiment is repeated on a faster computer[8], to verify that an increased camera resolution is yielding improved position accuracy when velocity estimates can be computed frequently. The results can be found in Table 8.8 and Figures 8.18 and 8.8.

For each resolution, the faster computer is able to process more frames, resulting in more frequent velocity updates. Therefore, the mean error decreases. Similar to the previous experiment, the number of processed frames decreases when the camera resolution increases. This indicates that the faster computer is still unable to process all incoming frames. The video processing implementation, which uses a single thread, is unable to make full use of the CPU's processing power. Despite this fact, the faster computer is able to benefit from the increased image resolution. A camera resolution of 133.3% is producing the smallest mean error, outperforming the mean error from the previous experiment. For this increased resolution, the faster computer is able to process the same amount of frames as the other computer running at the original resolution. When the resolution increases further (166.6% or 200.0%), the mean error increases due to the less frequent velocity estimates.

From these observations can be concluded that the AR.Drone's bottom camera resolution offers a good balance between image quality (usefulness) and processing time. An increased camera resolution is yielding improved position accuracy when the computer is able to process sufficient frames when the image resolution increases. A multi-threaded implementation is expected to provide a performance boost.

Resolution	Nr. frames processed	Nr. frames velocity extracted	Mean error (mm)
100.0%	1556	1145	69
133.3%	1330	961	**54**
166.6%	836	773	82
200.0%	655	624	83

Table 8.8: (Simulated AR.Drone & faster computer). Mean error of the estimation position for different camera resolutions. For each camera resolution, the number of processed frames is counted. A part of these frames can be used to extract the velocity of the AR.Drone.

[8]The original experiment was performed on a Intel Core 2 Duo 2.2 GHz. The experiment is repeated on a faster Intel Core i7-2600 3.4 GHz. The Passmark CPU Mark benchmark indicates a score of 1259 vs 8891 (higher is better). `http://www.cpubenchmark.net/cpu_list.php`

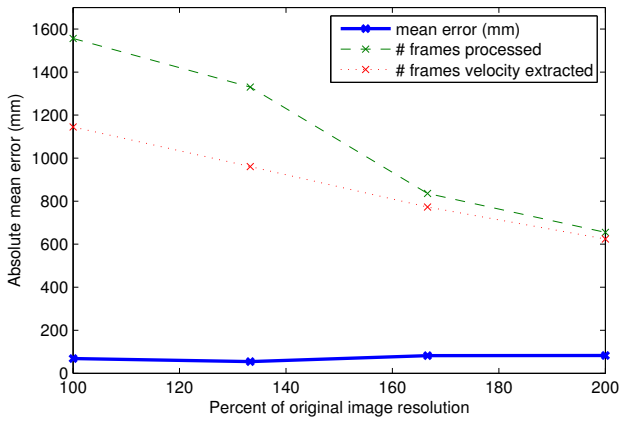


Figure 8.18: (Simulated AR.Drone & faster computer) Mean error of the estimated position with respect to the image resolution. For each resolution, the number of processed frames is counted (green line). A part of these frames can be used to extract the velocity of the AR.Drone (red line).

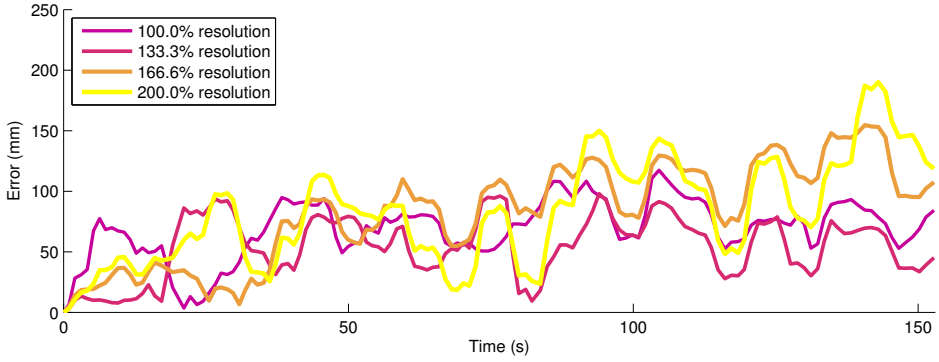


Figure 8.19: (Simulated AR.Drone & faster computer) Error of the estimated position for different image resolutions.

8.5 Elevation accuracy

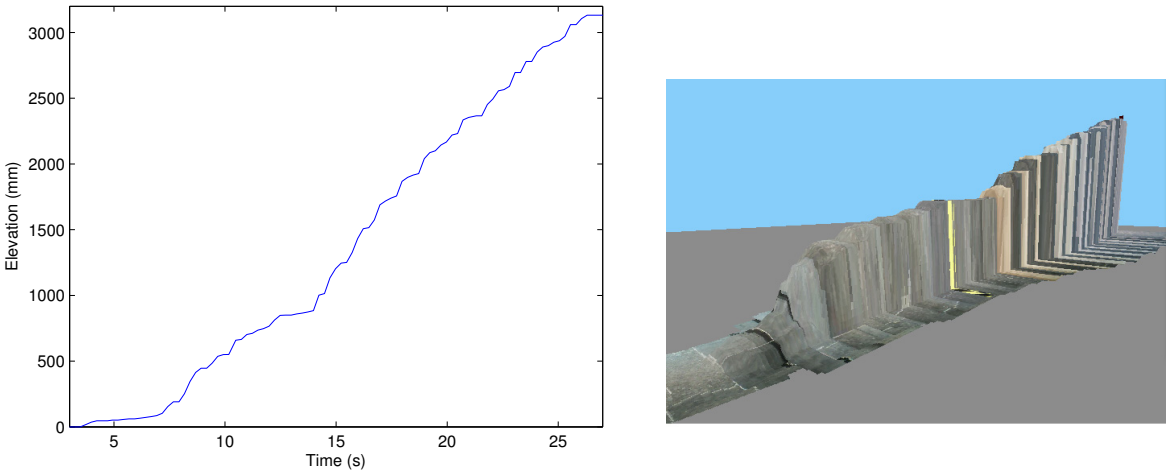
The accuracy of the elevation map is evaluated in terms of elevation error and the error in the estimated dimensions of objects.

Elevation map of a staircase

In the first experiment, the AR.Drone flew with a constant speed over a large staircase (Figure 8.20). The depth of each step is $30cm$ and the height of each step is $18cm$. The total height of the staircase is $480cm$. After the staircase is fully traversed by the AR.Drone, the estimated elevation is compared against the actual height of the staircase.

Figure 8.20: Photo of the staircase which is traversed by the AR.Drone. The depth of each step is $30cm$ *and the height of each step is* $18cm$*. The total height of the staircase is* $480cm$*.*

The results of this experiment can be found in Figure 8.21. After fully traversing the staircase, the measured elevation is $313cm$ and the error is $\frac{480-313}{480} \times 100 = 35\%$. The shape of the measured elevation (Figure 8.21) corresponds with the shape of the staircase (Figure 8.20). However, the approach underestimates the elevation. When traversing the staircase, the AR.Drone's altitude stabilization increases the altitude smoothly, which causes a continuous altitude increase. Therefore, the observed altitude difference within an elevation event is smaller than the actual altitude different caused by an object. Another explanation of the underestimation is the error in the triggering of elevation events (due to thresholds). When an elevation event is triggered at a suboptimal timestamp, the full altitude difference is not observed. Ideally, an elevation UP event is triggered right before an object is detected by the ultrasound sensor, and an elevation NONE event is triggered right after the full height of the object is observed by the ultrasound sensor.



(a) Plot of the detected elevation

(b) Visual elevation map

Figure 8.21: Elevation map of a large staircase (Figure 8.20). In 8.21(a) the measured elevation is plotted over time. In 8.21(b) the elevation map is rendered using the 3DTerrain component of the presented framework.

Elevation map of multiple objects

In the second experiment, the elevation mapping approach is evaluated in terms of error in the estimated dimensions of objects. Objects with different dimensions were laid out on a flat floor. A photo of the floorplan can be found in Figure 8.22. The AR.Drone's trajectory is indicated with a red line and covers all objects.

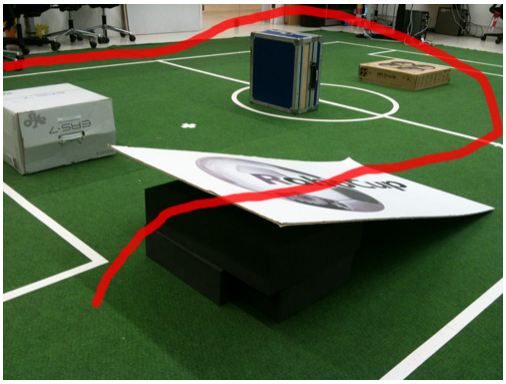


Figure 8.22: Photo of the floorplan with different objects. The dimensions of the objects can be found in Table 8.9. The red line indicates the trajectory of the AR.Drone.

The results of this experiment can be found in Table 8.9 and Figure 8.23. An object's width is evaluated in the trajectory's direction only. For the other direction, the width of an object cannot be estimated accurately,

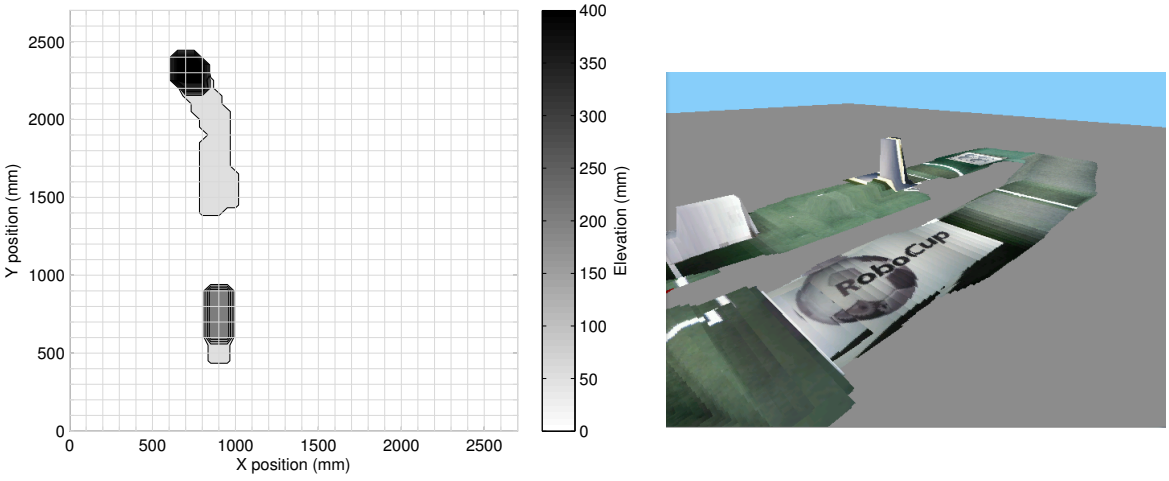
because the refinement step from Section 7.5 cannot be applied.

Object	Dimensions (w×h) (mm)	Estimated dimensions	Absolute error	Relative error
1. White box	460 × 290	390 × 245	70 × 45	**15 × 16 %**
2. Blue box	245 × 490	250 × 420	5 × 70	**2 × 14 %**
3. Brown box	570 × 140	-	-	-
4. Ramp	1380 × 360	-	-	-

Table 8.9: Errors made in the estimated dimensions of objects. The first dimension is the width of an object in the y-direction and the second dimension is the height of an object.

The white and blue boxes are detected. The errors of both the estimated width and height are small. Similar to the previous experiment, the height of an object is (slightly) underestimated. The width of the white box is underestimated, which is probably caused by unreliable velocity estimates due to the lack of sufficient features. Another explanation could be that the visual odometry methods assume a flat world, which does not hold when flying above objects. Between both boxes, a false elevation is detected (clearly visible in Figure 8.23(a)). This elevation is probably triggered by a fast descent, caused by the AR.Drone's altitude stabilization, after the white box is passed.

The brown box is not detected due to its limited height. Therefore, the measured (ultrasound) acceleration is not sufficient to trigger an elevation event. As expected, the ramp was not detected. The gradual elevation change does not produce a significant acceleration.



(a) Plot of the detected elevation

(b) Visual elevation map

Figure 8.23: Elevation map of multiple objects (Figure 8.22). In 8.21(a) a contour map of the elevation is plotted. In 8.21(b) the elevation map is rendered using the 3DTerrain component of the presented framework.

9 Conclusions

This chapter concludes this thesis by summarizing its content and answering the research questions. Section 9.1 describes the contributions of this thesis. Finally, Section 9.2 motivates several interesting directions for future research.

The platform selected in this thesis is the Parrot AR.Drone quadrotor helicopter. A simulation model of the AR.Drone is proposed, which allows safe and efficient development of algorithms. This simulation model consists of a motion model, visual model and sensor model. The validation effort of the hovering and the forward movement shows that the dynamic behavior of the simulated AR.Drone closely resembles the dynamic behavior of the real AR.Drone.

A framework is proposed to perform high-level tasks with the AR.Drone. This framework includes an abstraction layer to abstract from the actual device, which allows to use both the real and simulated AR.Drone to validate the proposed methods. The framework stores video frames and sensor measurements from the AR.Drone in queues, such that algorithms can operate on the data. Furthermore, the framework includes a 3D map visualization and provides functionalities to record and playback data.

This thesis proposed a real-time Simultaneous Localization and Mapping approach that can be used for micro aerial vehicles with a low-resolution down-pointing camera (e.g., AR.Drone). The information from the sensors are used in an Extended Kalman Filter (EKF) to estimate the AR.Drone's pose. The position of the AR.Drone cannot be estimated directly and is derived from the velocity estimates from the AR.Drone. This estimate is based on the inertia measurements, aerodynamic model and visual odometry obtained from the relative motion between camera frames. When an estimate of the AR.Drone's pose is available, a map can be constructed to make the AR.Drone localize itself and reduce the error of the estimated pose. The map consists of a texture map and a feature map. The texture map is used for human navigation and the feature map is used by the AR.Drone to localize itself.

The texture map is constructed by warping a camera frame on a flat canvas. A homography matrix describes how each frame has to be transformed (warped) in order to generate a seamless texture map without perspective distortion. For each frame, the estimated pose and camera matrix are used to determine the area of the world which is observed by the camera. This area is described by four 2D coordinates (a flat world is assumed). Now, the homography is estimated from the four 2D world positions and their corresponding frame's pixel coordinates.

The feature map consists of a grid with SURF features. For each camera frame, SURF features are extracted. The estimated pose and camera matrix are used to compute the 2D world position of each feature. Based on the 2D world position, a feature's cell inside the grid is computed. Only the best feature of a cell is stored.

This feature map is used by the AR.Drone to localize itself. For each camera frame, SURF features

are extracted. Similar to the mapping approach, the corresponding 2D world position of each feature is computed. Each feature is matched against the most similar feature from the feature map. From the feature pairs, a robust transformation between the 2D world coordinates is estimated.

An approach is proposed to compute the transformation between a frame and a map. The method is very similar to RANSAC, but uses covariance instead of the number inliers as quality measure. Three random feature pairs are selected. From these pairs, the 2D world coordinates are used to compute the mean translation. The covariance describes the quality of a translation. These steps are repeated multiple times and the best translation is selected. If a translation is very good (i.e., low covariance), Kabsch algorithm is used to refine the AR.Drone's estimated rotation. Experiments show that the proposed method significantly outperforms OpenCV's implementation to estimate a perspective, affine or Euclidean transformation. By reducing the degrees of freedom and using a beter quality measure, the robustness against noise increases.

A slightly modified version of the localization approach is used to perform visual odometry (estimate the velocity from camera images). Instead of matching the last frame against the feature map, now the last frame is matched against the previous frame. This estimated velocity can be used instead the AR.Drone's onboard velocity estimate. Experiments show that the accuracy of the proposed visual odometry method is comparable to the AR.Drone's onboard velocity estimate.

Others experiments show the AR.Drone down-pointing camera offers good balance between image quality and computational performance. When the resolution of the AR.Drone's camera is reduced, the accuracy of the estimated position decreases. The reduced image quality results in less frames that could be used to robustly recover the velocity. A simulated AR.Drone was used to investigate an increased camera resolution. An increased image resolution does not necessarily produce more accurately estimated positions. While relatively more frames could be used to used to robustly recover the velocity, the absolute number of processed frames decreased due to the increased computational complexity. Therefore, the velocities are estimated less frequent.

Another experiment shows that localization is possible for circumstances encountered during the IMAV competition. When flying the figure-8 shape three times, the maximum position error was $1m$.

In addition to the texture and feature map, this thesis proposed a method to construct an elevation map using a single airborne ultrasound sensor. Elevations are detected through sudden changes in ultrasound measurements. These sudden changes are detected when the filtered second order derivative exceeds a certain threshold. Experiments show that the relative error when flying over small objects is below 20%. When flying over a large staircase, the relative error of the estimated altitude was 35%.

9.1 Contributions

The first contribution of this thesis is a framework that aids in the development of (intelligent) applications for the AR.Drone. The framework contains an abstraction layer to abstract from the actual device. This enables to use a simulated AR.Drone in a way similar to the real AR.Drone. Therefore, a simulation model of the AR.Drone is developed in USARSim. The main contribution is a novel approach that enables a MAV with a low-resolution down-looking camera to navigate in circumstances encountered during the IMAV competition. An algorithm is presented to estimate the transformation between a camera frame and a map. In addition to the navigation capabilities, the approach generates a texture map, which can be used for human navigation. Another contribution is a method to construct an elevation with a single airborne ultrasound sensor.

Parts of this thesis have been published in:

N. Dijkshoorn and A. Visser, "Integrating Sensor and Motion Models to Localize an Autonomous AR. Drone", *International Journal of Micro Air Vehicles*, volume 3, pp. 183-200, 2011

A. Visser, N. Dijkshoorn, M. van der Veen, R. Jurriaans, "Closing the gap between simulation and reality in the sensor and motion models of an autonomous AR.Drone", *Proceedings of the International Micro Air Vehicle Conference and Flight Competition (IMAV11)*, 2011. **Nominated for best paper award**

N. Dijkshoorn and A. Visser, "An elevation map from a micro aerial vehicle for Urban Search and Rescue - RoboCup Rescue Simulation League", *to be published on the Proceedings CD of the 16th RoboCup Symposium, Mexico*, June 2012

Winner of the RoboCup Rescue Infrastructure competition, Mexico, June 2012

9.2 Future research

The proposed simulation model of the AR.Drone is validated in terms of hovering and forward movements. Further improvements would require more system identifications (e.g., rotational movements, wind-conditions) and include the characteristics of the Parrot's proprietary controller into the simulation model. The current sensor model uses a set of default USARSim sensors, which do not model noise accurately. This research would benefit from a realistic modeling of sensor noise.

The proposed SLAM method is able to construct a map and localize the AR.Drone in real-time. However, the proposed method lacks a global map optimization method, which reduces the error in the map

when a loop-closure is detected. An interesting research direction is to develop a global map optimization that is able to optimize both the feature map and texture map in real-time.

Experiments have shown that the proposed visual odometry method is unable to accurately estimate the velocity when insufficient texture is visible on the floor. In these circumstances, the AR.Drone's onboard optical-flow based visual odometry method is able to estimate the velocity more accurately. However, salient lines on the floor cause the AR.Drone's optical-flow method to estimate the velocity in the wrong direction. This problem can be solved by integrating additional information (e.g., aerodynamic model) into the optical-flow method, to correct the direction of the estimated velocity. When such a method is incorporated in the proposed SLAM method, it would yield an increased accuracy of the estimated velocities in low-texture conditions.

The elevation mapping experiment has shown that elevation mapping with a single ultrasound sensor is possible. However, the proposed method is sensitive to errors that accumulate. When an object is detected, the elevation is increased. When an object is out of range, the elevation is decreased. When both events do not observe an equal elevation, an incorrect elevation propagates into the elevation map. A possible solution could be a global optimization method, which is able to correct incorrect elevations. For example, large areas for which the elevation does not change can be assumed to be the floor. Furthermore, the current approach is unable to estimate the dimensions of an object in all directions. Visual clues could be used to estimate the dimensions of an object. For example, image segmentation can be used to determine the boundaries of an object. Another possible solution is to create a depth map from optical flow. Just as stereo vision can be used to create a depth map of the environment, the optical flow between two subsequent frames can be used to create a depth map [1].

The proposed framework uses a single thread to process the camera frames. Therefore, the proposed SLAM method is unable to benefit from an increased image resolution. This issue can be solved by processing multiple frames in parallel (using multiple video processing threads), to make better use of a multicore CPU.

Opening angle of the ultrasound sensor

The opening angle of the AR.Drone's ultrasound sensor is not documented. A small experiment has been performed to determine the opening angle.

The AR.Drone was positioned at a fixed altitude of $79cm$ above a flat floor. No obstacle was in range of the ultrasound sensor to make sure the measured altitude is the actual altitude of the AR.Drone. The altitude measured by the ultrasound sensor was $75.2cm$, which equals a systematic error of $3.8cm$. The AR.Drone's bottom camera was used to mark a point on the floor that is exactly below the center of the AR.Drone. This point is equal to the center of ultrasound cone. In order to measure the opening angle, floating objects were moved from outside the cone towards the cone. The objects require a certain distance from the floor to achieve a distance short than the shortest distance between the sonar and the floor. The minimal altitude of an object when assuming a maximum opening angle of 40 degrees is:

$$a = 79cm - (cos(40\,\mathrm{deg}) \times 79cm) = 18.48cm \tag{A.1}$$

Fluctuations in the altitude measurements indicate that the object is entering the cone. The horizontal distance d between the object and cone center (marked point) is the width of the cone and $h = 79cm - a$ is the height of the cone. The angle α of the cone can be recovered with:

$$\alpha = tan^{-1}(d/h) \tag{A.2}$$

The experiment was repeated 10 times with different objects. The average opening angle is $25.03°$ with a standard deviation of $1.45°$.

Source code

B

All source code is available at Google Code: `http://code.google.com/p/mscthesis-ndijkshoorn/`. The recorded datasets, which are stored in YAML format, can be found at the downloads page: `http://code.google.com/p/mscthesis-ndijkshoorn/downloads/list`.

A research log can be found at `http://nickd.nl/wiki/thesis`.

```
#include "global.h"
#include "bot_ardrone.h"
#include "bot_ardrone_keyboard.h"
#include "bot_ardrone_3dmouse.h"
#include "bot_ardrone_behavior.h"


/* global variables */
bool exit_application   = false;
bool stop_behavior      = false;


int main(int argc, char *argv[])
{
    HWND consoleWindow = GetConsoleWindow();
    MoveWindow(consoleWindow, 0, 0, 600, 400, false);


    int nr_bots = 0;
    bot_ardrone *bots[1];

    /* bot 1: ARDRONE  */
    bot_ardrone ardrone(0x01, BOT_ARDRONE_INTERFACE_ARDRONELIB, SLAM_MODE_VEL | SLAM_MODE_MAP  | SLAM_MODE_ELEVMAP);
    bots[nr_bots++] = &ardrone;

    bot_ardrone_behavior autonomous(&ardrone);
    bot_ardrone_3dmouse mouse(bots, nr_bots);
    bot_ardrone_keyboard kb(bots, nr_bots);

    return 0;
}
```

Figure B.1: Source code of the application main file. This code initializes a robot instance and its controllers (autonomous behavior, keyboard and 3D mouse).

Bibliography

[1] R. Jurriaans, "Flow based Obstacle Avoidance for Real World Autonomous Aerial Navigation Tasks", Bachelor's thesis, Universiteit van Amsterdam, August 2011.

[2] M. van der Veen, "Optimizing Artificial Force Fields for Autonomous Drones in the Pylon Challenge using Reinforcement Learning", Bachelor's thesis, Universiteit van Amsterdam, July 2011.

[3] R. Murphy, *Introduction to AI robotics*, The MIT Press, 2000.

[4] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda and E. Osawa, "Robocup: The robot world cup initiative", in "Proceedings of the first international conference on Autonomous agents", pp. 340–347, ACM, 1997.

[5] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou and S. Shimada, "Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research", in "Systems, Man, and Cybernetics, 1999. IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on", volume 6, pp. 739–743, IEEE, 1999.

[6] "Workshop on Robots, Games, and Research: Success stories in USARSim", in "Proceedings of the International Conference on Intelligent Robots and Systems (IROS 2009)", (edited by S. Balakirky, S. Carpin and M. Lewis), IEEE, October 2009, ISBN 978-1-4244-3804-4.

[7] S. Carpin, M. Lewis, J. Wang, S. Balakirsky and C. Scrapper, "USARSim: a robot simulator for research and education", in "Robotics and Automation, 2007 IEEE International Conference on", pp. 1400–1405, IEEE, 2007.

[8] R. Talluri and J. Aggarwal, "Position estimation for an autonomous mobile robot in an outdoor environment", *Robotics and Automation, IEEE Transactions on*, volume 8(5):pp. 573–584, 1992.

[9] D. Fox, S. Thrun and W. Burgard, "Probabilistic robotics", 2005.

[10] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[11] R. Kalman, "A new approach to linear filtering and prediction problems", *Journal of basic Engineering*, volume 82(Series D):pp. 35–45, 1960.

[12] G. Grisetti, S. Grzonka, C. Stachniss, P. Pfaff and W. Burgard, "Efficient estimation of accurate maximum likelihood maps in 3d", in "Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on", pp. 3472–3478, IEEE, 2007.

[13] E. Olson, J. Leonard and S. Teller, "Fast iterative alignment of pose graphs with poor initial estimates", in "Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on", pp. 2262–2269, Ieee, 2006.

[14] T. Barrera, A. Hast and E. Bengtsson, "Incremental spherical linear interpolation", *Proc. SIGRAD*, volume 13:pp. 7–13, 2004.

[15] H. Gernsheim, A. Gernsheim and H. Gernsheim, *The history of photography: from the camera obscura to the beginning of the modern era*, McGraw-Hill, 1969.

[16] D. Lowe, "Object recognition from local scale-invariant features", in "Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on", volume 2, pp. 1150–1157, Ieee, 1999.

[17] H. Bay, A. Ess, T. Tuytelaars and L. V. Gool, "Speeded-Up Robust Features (SURF)", *Computer Vision and Image Understanding*, volume 110(3):pp. 346 – 359, 2008, ISSN 1077-3142, doi:DOI:10.1016/j.cviu.2007.09.014.

[18] A. Van Den Bos, "Complex gradient and Hessian", in "Vision, Image and Signal Processing, IEE Proceedings-", volume 141, pp. 380–383, IET, 1994.

[19] E. Stollnitz, A. DeRose and D. Salesin, "Wavelets for computer graphics: a primer. 1", *Computer Graphics and Applications, IEEE*, volume 15(3):pp. 76–84, 1995.

[20] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope", *International Journal of Computer Vision*, volume 42(3):pp. 145–175, 2001.

[21] M. Muja, "FLANNFast Library forApproximate Nearest Neighbors", 2009.

[22] M. Fischler and R. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography", *Communications of the ACM*, volume 24(6):pp. 381–395, 1981.

[23] C. Engels, H. Stewénius and D. Nistér, "Bundle adjustment rules", *Photogrammetric Computer Vision*, volume 2, 2006.

[24] H. Longuet-Higgins, "A computer algorithm for reconstructing a scene from two projections", *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms, MA Fischler and O. Firschein, eds*, pp. 61–62, 1987.

[25] V. Klema and A. Laub, "The singular value decomposition: Its computation and some applications", *Automatic Control, IEEE Transactions on*, volume 25(2):pp. 164–176, 1980.

[26] B. Steder, G. Grisetti, C. Stachniss and W. Burgard, "Visual SLAM for flying vehicles", *Robotics, IEEE Transactions on*, volume 24(5):pp. 1088–1093, 2008, ISSN 1552-3098.

[27] G. Tipaldi, G. Grisetti and W. Burgard, "Approximate covariance estimation in graphical approaches to SLAM", in "Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on", pp. 3460–3465, IEEE, 2007.

[28] O. Chum and J. Matas, "Matching with PROSAC-progressive sample consensus", in "Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on", volume 1, pp. 220–226, Ieee, 2005.

[29] F. Caballero, L. Merino, J. Ferruz and A. Ollero, "Unmanned Aerial Vehicle Localization Based on Monocular Vision and Online Mosaicking", *Journal of Intelligent and Robotic Systems*, volume 55(4):pp. 323–343, 2009, ISSN 0921-0296.

[30] Z. Zhang, "Parameter estimation techniques: A tutorial with application to conic fitting", *Image and vision Computing*, volume 15(1):pp. 59–76, 1997.

[31] B. Triggs, "Autocalibration from planar scenes", *Computer VisionECCV'98*, pp. 89–105, 1998.

[32] A. Foessel-Bunting, "Radar sensor model for three dimensional map building", *Proc. SPIE, Mobile Robots XV and Telemanipulator and Telepresence Technologies VII, SPIE*, volume 4195, 2000.

[33] M. Weiß, J. Ender, O. Peters and T. Espeter, "An airborne radar for three dimensional imaging and observation-technical realisation and status of ARTINO", *EUSAR 2006*, 2006.

[34] A. Johnson and M. Hebert, "Seafloor map generation for autonomous underwater vehicle navigation", *Autonomous Robots*, volume 3(2):pp. 145–168, 1996.

[35] O. Strauss, F. Comby and M. Aldon, "Multibeam sonar image matching for terrain-based underwater navigation", in "OCEANS'99 MTS/IEEE. Riding the Crest into the 21st Century", volume 2, pp. 882–887, IEEE, 1999.

[36] B. Zerr and B. Stage, "Three-dimensional reconstruction of underwater objects from a sequence of sonar images", in "Image Processing, 1996. Proceedings., International Conference on", volume 3, pp. 927–930, IEEE, 1996.

[37] J. Evans, N. I. of Standards and T. (US), *Three dimensional data capture in indoor environments for autonomous navigation*, US Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, 2002.

[38] A. Kenny, I. Cato, M. Desprez, G. Fader, R. Schüttenhelm and J. Side, "An overview of seabed-mapping technologies in the context of marine habitat classification", *ICES Journal of Marine Science: Journal du Conseil*, volume 60(2):pp. 411–418, 2003.

[39] P. Blondel and B. Murton, *Handbook of seafloor sonar imagery*, Wiley Chichester,, UK, 1997.

[40] S. Williams and I. Mahon, "Simultaneous localisation and mapping on the Great Barrier Reef", in "Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on", volume 2, pp. 1771–1776, IEEE, 2004.

[41] B. Lucas and T. Kanade, "with an Application to Stereo Vision", *Proceedings DARPA Image Understanding Workrhop*, pp. 121–130, 1998.

[42] C. Bills, J. Chen and A. Saxena, "Autonomous MAV Flight in Indoor Environments using Single Image Perspective Cues", in "International Conference on Robotics and Automation (ICRA)", 2011.

[43] J. Canny, "A computational approach to edge detection", *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):pp. 679–698, 1986.

[44] N. Kiryati, Y. Eldar and A. Bruckstein, "A probabilistic Hough transform", *Pattern recognition*, volume 24(4):pp. 303–316, 1991.

[45] T. Krajník, J. Faigl, V. Vonásek, K. Košnar, M. Kulich and L. Přeučil, "Simple yet stable bearing-only navigation", *Journal of Field Robotics*, volume 27(5):pp. 511–533, 2010.

[46] J. Faigl, T. Krajnık, V. Vonásek and L. Preucil, "Surveillance Planning with Localization Uncertainty for UAVs", .

[47] T. Krajník, V. Vonásek, D. Fišer and J. Faigl, "AR-Drone as a Platform for Robotic Research and Education", *Research and Education in Robotics-EUROBOT 2011*, pp. 172–186, 2011.

[48] G. M. Hoffmann, H. Huang, S. L. Wasl and C. J. Tomlin, "Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment", in "Proc. of the AIAA Guidance, Navigation and Control Conference", 2007.

[49] J. Borenstein and Y. Koren, "Obstacle avoidance with ultrasonic sensors", *Robotics and Automation, IEEE Journal of*, volume 4(2):pp. 213–218, 1988.

[50] P. Bristeau, F. Callou, D. Vissière and N. Petit, "The Navigation and Control technology inside the AR. Drone micro UAV", in "World Congress", volume 18, pp. 1477–1484, 2011.

[51] B. Lukas and T. Kanade, "An iterative image registration technique with an application to stereo vision", in "Image Understanding Workshop", 1981.

[52] E. Rosten, R. Porter and T. Drummond, "Faster and better: A machine learning approach to corner detection", *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, volume 32(1):pp. 105–119, 2010.

[53] E. Michaelsen and U. Stilla, "Pose estimation from airborne video sequences using a structural approach for the construction of homographies and fundamental matrices", *Structural, Syntactic, and Statistical Pattern Recognition*, pp. 486–494, 2004.

[54] N. Michael, D. Mellinger, Q. Lindsey and V. Kumar, "The GRASP Multiple Micro-UAV Testbed", *Robotics Automation Magazine, IEEE*, volume 17(3):pp. 56 –65, sept. 2010, ISSN 1070-9932, doi:10.1109/MRA.2010.937855.

[55] O. Formsma, N. Dijkshoorn, S. van Noort and A. Visser, "Realistic simulation of laser range finder behavior in a smoky environment", *RoboCup 2010: Robot Soccer World Cup XIV*, pp. 336–349, 2011.

[56] S. Carpin, J. Wang, M. Lewis, A. Birk and A. Jacoff, "High fidelity tools for rescue robotics: results and perspectives", *RoboCup 2005: Robot Soccer World Cup IX*, pp. 301–311, 2006.

[57] J. Wang, M. Lewis, S. Hughes, M. Koes and S. Carpin, "Validating usarsim for use in hri research", in "Proceedings of the Human Factors and Ergonomics Society Annual Meeting", volume 49, pp. 457–461, SAGE Publications, 2005.

[58] S. Carpin, M. Lewis, J. Wang, S. Balakirsky and C. Scrapper, "Bridging the gap between simulation and reality in urban search and rescue", *Robocup 2006: Robot Soccer World Cup X*, pp. 1–12, 2007.

[59] S. Carpin, T. Stoyanov, Y. Nevatia, M. Lewis and J. Wang, "Quantitative assessments of USARSim accuracy", in "Proceedings of PerMIS", volume 2006, 2006.

[60] S. Carpin, J. Wang, M. Lewis, A. Birk and A. Jacoff, "High Fidelity Tools for Rescue Robotics: Results and Perspectives", in "RoboCup 2005: Robot Soccer World Cup IX", (edited by A. Bredenfeld, A. Jacoff, I. Noda and Y. Takahashi), *Lecture Notes in Computer Science*, volume 4020, pp. 301–311, Springer Berlin / Heidelberg, 2006, doi:10.1007/11780519_27.

[61] T. Yechout, S. Morris, D. Bossert and W. Hallgren, *Introduction to Aircraft Flight Mechanics: Performance, Static Stability, Dynamic Stability, and Classical Feedback Control*, American Institute of Aeronautics and Astronautics, Reston, VA, 2003.

[62] A. Visser, N. Dijkshoorn, M. van der Veen and R. Jurriaans, "Closing the gap between simulation and reality in the sensor and motion models of an autonomous AR.Drone", in "Proceedings of the International Micro Air Vehicle Conference and Flight Competition (IMAV11)", September 2011.

[63] S. Julier and J. Uhlmann, "Unscented filtering and nonlinear estimation", *Proceedings of the IEEE*, volume 92(3):pp. 401–422, 2004.

[64] A. Levin, A. Zomet, S. Peleg and Y. Weiss, "Seamless image stitching in the gradient domain", *Computer Vision-ECCV 2004*, pp. 377–389, 2004.

[65] Z. Zhang, "A Flexible New Technique for Camera Calibration", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 22:pp. 1330–1334, 2000.

[66] J.-Y. Bouguet, *Visual methods for three-dimensional modeling*, Ph.D. thesis, California Institute of Technology, May 1999.

[67] W. Kabsch, "A solution for the best rotation to relate two sets of vectors", *Acta Crystallographica Section A*, volume 32(5):pp. 922–923, Sep 1976, doi:10.1107/S0567739476001873.

[68] A. Jacoff, E. Messina, H.-M. Huang, A. Virts and A. Downs, "Standard Test Methods for Response Robots", ASTM International Committee on Homeland Security Applications, January 2010, subcommittee E54.08.01.