Leiden University

Computer Science

Mobile radio tomography:
Object detection using autonomous unmanned vehicles

Name: Leon Helwerda
Date: March 9, 2018
1st supervisor: Walter Kosters (LIACS)
2nd supervisor: Joost Batenburg (CWI & MI)

MASTER'S RESEARCH PROJECT REPORT

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Radio tomography applies low-energy radio signals to measure and reconstruct physical properties of objects within an environment. Currently, this technique is based on measurements from static sensors. We propose a mobile setup that could improve the reconstruction. We explore the possibilities for using unmanned vehicles to create a mobile sensor network. The vehicle must detect objects and avoid collisions during a scanning mission. We investigate available software and hardware. A new toolchain determines a trajectory using distance sensors. We implement safety checks, mapping components and visualizations. The vehicle is simulated within different environments. Our experiments show that preplanned missions can only function within an unknown environment if we use artificial intelligence to make quick decisions during the mission.

Contents

1 Introduction **2**
- 1.1 Problem statement . . . 4
  - 1.1.1 Motivation . . . 4
  - 1.1.2 Applications . . . 5
- 1.2 Approach . . . 5
  - 1.2.1 Team . . . 6
- 1.3 Overview . . . 6

2 Related work **7**
- 2.1 Toolchains . . . 7
- 2.2 Hardware . . . 7

3 Drones and other vehicles **9**
- 3.1 Requirements . . . 9
- 3.2 Drone regulations . . . 10

4 Implementation **11**
- 4.1 Overview of the components . . . 11
- 4.2 Simulations . . . 12
  - 4.2.1 Vehicle . . . 12
  - 4.2.2 Geometry . . . 13
  - 4.2.3 Environment . . . 14
  - 4.2.4 Visualization . . . 15
  - 4.2.5 Distance sensor . . . 16
  - 4.2.6 Servo . . . 18
- 4.3 Missions . . . 18
  - 4.3.1 Monitor . . . 19
  - 4.3.2 Memory map . . . 20
  - 4.3.3 Planned missions . . . 20
  - 4.3.4 Guided missions . . . 21

5 Experiments **22**
- 5.1 Setup . . . 22
- 5.2 Results . . . 23

6 Conclusions **24**
- 6.1 Further research . . . 24

References **25**

1 Introduction

Radio tomography is an emerging collection of techniques that can detect and localize objects and people within an environment without any markers or devices on the objects of interest themselves. The technique makes use of wireless transmission sensors that send and receive signals. The signals are then intersected and attenuated by the objects, depending on their density. The gathered signal strength measurements allow us to derive information about properties of the objects and use *radio tomographic imaging* (RTI) to reconstruct an image that resembles the actual situation [24].

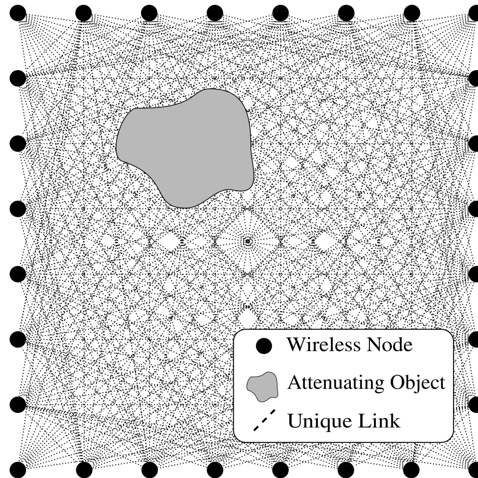


Figure 1: A radio tomography network with static sensors that send and receive signals. The objects inside the network are reconstructed using the measurements [24].

This network, as illustrated in Figure 1, requires many static sensors in order to reconstruct objects with enough detail and less noise [12]. This means that deployment of a network at a random location is impractical. By using mobile sensors, we can make network more dynamic and require fewer assumptions about the environment. This also decreases the necessary number of physical sensors by placing them at different locations after every signal sweep. This allows more freedom to scan the environment, so that we can adaptively determine the regions of interest.

A dynamic setup brings new challenges that we need to research. Reconstruction becomes more computationally expensive and the result may be noisy due to interference. Problems arise if not all measurements are available. We need to gather enough signal strength measurements to perform 2D reconstruction, which can be extended to 3D reconstruction. Instead of having static positions at which we always measure, we now need to determine a strategy for the positions while scanning.

We propose this *mobile radio tomography* variant and solve the problem of positioning the sensors through the use of *drones*. These are flying vehicles that carry a small payload and move between locations. There is an increase in the use of drones and other small, unmanned vehicles for various real-life applications. Reconnaissance missions can be used in, e.g., the security and military sectors. Particular contexts are crowd control [19] and identification of objects and people. Drones with mounted cameras can also be used for professional filming. In scientific research, we can apply the concepts related to autonomous movement and *artificial intelligence* (AI) to perform dynamic experiments. This allows us to turn the static sensor network into a mobile setup.

We can remotely operate a drone, such as the one shown Figure 2, or let it fly around autonomously. A plane or a miniature driving car also support these modes using specific hardware suited for controlling the rotors, motors and servos on the physical device. A *flight controller* acts as an interface between the decision-making actor and the physical mechanisms. The AI actor can issue commands remotely if the flight controller is capable of receiving those wireless signals. A remote decision maker does have the problem of latency before the command is acknowledged, and thus a local AI actor running on the vehicle can be preferable. We achieve this using a *companion computer* that is directly connected to the flight controller and can run the AI actor.

Figure 2: An example bare-bones quadcopter drone [2].

This imposes hardware limits on the AI, since the small computer does not have the necessary clock cycle frequency or specific optimized chip components to perform overly complex calculations in real-time. Image recognition and reconstruction tasks fall outside the reach of this equipment. We opt for simple sensors that give a distance measure to other objects without lots of processing, unlike a full-scale camera.

Although the vehicle can operate autonomously, the realization of a *ground station* is still desirable. The ground station monitors the current status of the vehicle and a human operator can take action in case of problematic behavior. The ground station also performs the reconstruction tasks. We then need bidirectional communication using a lightweight protocol. Additionally, a swarm-based solution is possible when unmanned vehicles inform each other about dangers or features of interest.

Our research focuses initially on implementing an autonomous vehicle control toolchain. The code simulates, plans and runs missions for unmanned vehicles, and makes intelligent decisions based on sensor information during the mission. We then use this toolchain to perform experiments and proof-of-concept missions in order to verify whether it is working as expected, or in which cases the mission is unable to decide on a safe trajectory within its environment.

This research project is created in association with the LIACS Institute of Advanced Computer Science of Leiden University as well as the CWI research centre for mathematics and computer science in Amsterdam, under the supervision of Walter Kosters and Joost Batenburg. We provide a complete profile of the research team in Section 1.2.1.

1.1 Problem statement

The project as a whole centers around the problem of *mobile radio tomography.* The major parts of this problem are autonomous control of unmanned vehicles, sensors that send and detect a form of radiation through objects, and the reconstruction of this information into a visualization of the internal contents of the objects. We use a simplified variant of the WiFi messaging protocol known as ZigBee, which operates at a frequency spectrum around 2.4 GHz [17].

This thesis states the first subproblem in more detail and attempts to find solutions for this part only. However, the context of this subproblem is of importance in its elaboration, since the task is to plan a trajectory around the relevant objects in order to scan them.

Flight planning and unmanned vehicle operations have their own subproblems which we need to worry about. The planning phase revolves around whether preplanned paths or dynamically-chosen trajectories in fact solve the problem at hand. We assess the benefits and disadvantages of these types of missions. During the mission itself, we need to keep the vehicle in a safe position so that it can perform the necessary tasks. The vehicle must visit a certain sequence of locations. We assess the sources of information that are available to detect our position within the environment, such as GPS, altimeters and distance sensors.

We also need to tackle physical constraints of the hardware. These restrictions include operating speed and maximum load of the vehicle. The cable connections to the peripherals and sensors must be secured tightly so that it does not become loose during flight. Finally, the software-level interfaces between the companion computer and flight controller have to function quickly and correctly.

1.1.1 Motivation

As mentioned in the introduction in Section 1, the use of autonomous vehicles presents many research opportunities. We can transform a static experiment into a dynamic or mobile approach. In our case, we wish to replace a sensor network that operates with a large number of sensors at fixed positions. The new setup instead uses fewer sensors that are moved around.

This setup should be able to replace the static sensor network with fewer nodes and still perform at least the same kinds of measurements as the static network. There may be a downgrade in the real-time reconstruction of the detected objects, since the sensors cannot be in multiple places at the same time. Also, the data needs to be sent to a ground station before it can be processed. The main point is to be able to perform scans at different altitudes and more locations, in an effort to improve the reconstruction quality altogether.

The intention is to make this novel setup easy to deploy with a low budget. The distance sensors and tomography communication chips are quite cheap, and a companion computer can also be purchased easily with the necessary power cables and battery packs. Prices for drones and other radio-controlled vehicles range from budget customer models to elaborate professional products. The most expensive part may very well be the flight controller which binds all the physical components together and can ensure a safe flight. We use readily available sensors and other hardware, so that the principle can be reused in real-world applications with the use of, e.g., WiFi antenna stations.

An important rationale of our research is that it is a novel approach toward a widely investigated problem where multiple scientific fields meet. Radio tomography has a basis of mathematical models and physical properties. Meanwhile, drone operations also make use of such concepts in trajectory planning and flight movement. This allows for an interesting combination of theoretical and mathematical foundations, realized within the boundary of computer science by means of an AI actor, and taking place within a physical environment.

1.1.2 Applications

Radio tomographic imaging is a technique for reconstructing an image of objects within a specified area by sending out waves of radiation and detecting them at collectors. An object located between the sensors attenuates and reflects the radiation, causing a different signal strength at the collector. There is a large foundation of tomography theory to make the reconstruction feasible. There exist applications in the medical diagnostics field [3].

Whereas medical imaging uses high-energy radiation to accurately detect whether and to what extent a ray is blocked by the objects of interest, we reconstruct tomogram images using waves in the electromagnetic spectrum that have less energy. Emitting the waves then requires less power, making it feasible to use a battery source for powering the sensors.

With a lower power also comes a much lower risk of permanent damage caused by radiation. This opens up possibilities for tomography in applications that frequently involve biological materials such as humans without concerns for radiation dose. In fact, many of these applications correspond to the use cases mentioned in the introduction in Section 1. We can use tomography to detect whether there are people inside a building, which is useful for burglar detection and other security purposes. If the building is on fire, for example, the fire brigade can detect from a distance whether any people need rescue.

There are drawbacks to the low-power tomography approach that hinders its widespread application. The waves might not fully penetrate the objects. It is also difficult to distinguish between an attenuated signal and one that is received fine. Noise filtering is essential to achieve a recognizable result. Even detecting anything relevant is great. The resulting images only show “blobs” of detected objects. This has the benefit of preserving the privacy of people within the scan region since there are no determining features [24].

To improve the reconstruction phase, we can use additional data collected using the unmanned vehicle. The distance sensor is not only meant for avoiding collisions with objects, but to detect the exterior walls of the building, for example. This convex hull can be used to ensure that the reconstructed image is calculated to be inside these boundaries.

1.2 Approach

The project is split up in multiple phases related to operating unmanned vehicles, communicating between the tomography sensors [17] and the reconstruction and visualization. Section 1.1 summarizes these phases, and Figure 3 provides a high-level overview of them.

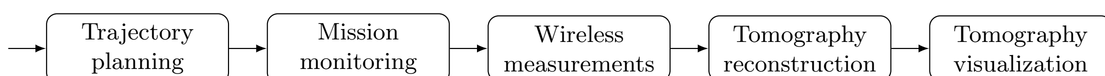


Figure 3: Flow diagram of the high-level phases of the toolchain

This thesis focuses on the first phases of autonomous unmanned vehicle control. We state the desired applications, research existing hardware and software support, implement a software toolchain that plans and monitors missions, and experiment with the entire setup.

A large part of the research involves simulations before moving on to a physical setup. This roadmap is mostly for safety reasons, so that we test the implementation without crashing the vehicle into real objects. This also allows for precise comparisons of missions. We can quickly tune parameters with less potential for errors [23].

1.2.1 Team

In order to research all the fields related to the drone tomography project and to divide the tasks among the people responsible for it, we formed a research group consisting of members from Leiden University as well as CWI Amsterdam. The group consists of the following members (in alphabetical order):

- Joost Batenburg (CWI Amsterdam): Supervisor, diverse knowledge of tomography theory and radio tomography imaging (RTI)
- Folkert Bleichrodt (CWI Amsterdam): Researcher in RTI and robot movement
- Leon Helwerda (Leiden University): MSc student, focus on drone operations
- Walter Kosters (Leiden University): Supervisor, diverse knowledge in the field of artificial intelligence
- Tim van der Meij (Leiden University): MSc student, focus on ZigBee sensor control and communication

The group is frequently assisted by Daniel Pelt, Zhichao Zhong and Xiaodong Zhuge, all from CWI Amsterdam, with useful feedback on theoretical basis of tomography, drone research and antenna properties. We also appreciate the initial help and suggestions from Willem Jan Palenstijn (CWI Amsterdam) and Alyssa Milburn. This allows our research to base upon and continue with earlier projects [16, 18].

1.3 Overview

The remainder of this thesis is structured as follows. We investigate existing software and hardware that may be of use in our implementations in Section 2. We provide insights into the scope of the thesis, namely drone operations, and state concrete definitions that are relevant within this scope in Section 3. This includes formal requirements of the physical vehicle in Section 3.1 as well as a study of applicable drone regulations in Section 3.2.

We then present an overview of the implementation for unmanned vehicle control, as part of our toolchain in Section 4. The implementation comprises of several components that are related to simulations of the environment, actual missions, or both. We describe the components related to these two elements in Sections 4.2 and 4.3, respectively.

Finally, we report on several experiments with the toolchain in Section 5. We provide details of the setup of each experiment in Section 5.1 and present the results in Section 5.2. We then conclude the project's thesis with some observations during the experiments in Section 6. In Section 6.1, we mention the problems that the project continues to research as well as additional subjects that may become future research.

2 Related work

In this section, we delve into previous experimental research on drone operations. This is mostly a practical overview of existing solutions, rather than a pure literature study. We discuss and elaborate on the workings of several software toolchains in Section 2.1, and review the flight controller hardware that they run on in Section 2.2.

2.1 Toolchains

There exists a variety of software collections related to steering and driving unmanned vehicles remotely or autonomously. In particular, recent efforts focus on multiple unmanned aerial vehicles (UAVs), such as drones, miniature planes and rover cars, using an interface that is compatible between all of them. There are open source software packages that are designed to interface with each other and remain as versatile as possible for the end user.

The toolchain that we focus on is a collection of three software layers. The first package is ArduPilot [2], which operates on the lowest level, namely the flight controller. It can control rotors of drones, flies fixed-wing planes and powers servo motors of *rover* cars. ArduPilot knows how to make a UAV take off by itself, can keep a mission plan based on waypoint locations that the vehicle visits in order, and can return to the launch site relatively easy. The flight controller can be connected to various peripherals. The ArduPilot code supervises these sensors and takes very simplistic autonomous decisions.

ArduPilot provides a simulator to test the vehicle's functioning with various simulated scenarios, such as a GPS sensor failure and random faulty readings. However, both the real binary program and the simulator cannot be started standalone, since it needs to receive parameters and other information regarding the mission plan. This can be received via various communication channels from a ground station, which can be a directly-linked companion computer or another program in simulation.

This *telemetry* interface makes use of a self-contained communications protocol known as MAVLink [14]. The protocol supports various kinds of packets that can be sent to or received by the flight controller, either regarding status updates or commands to change the vehicle mode or operating parameters.

The companion computer can also interface with MAVLink using various programming language bindings. MAVLink comes together with a Python package known as MAVProxy, which can create minimal ground stations via additional links.

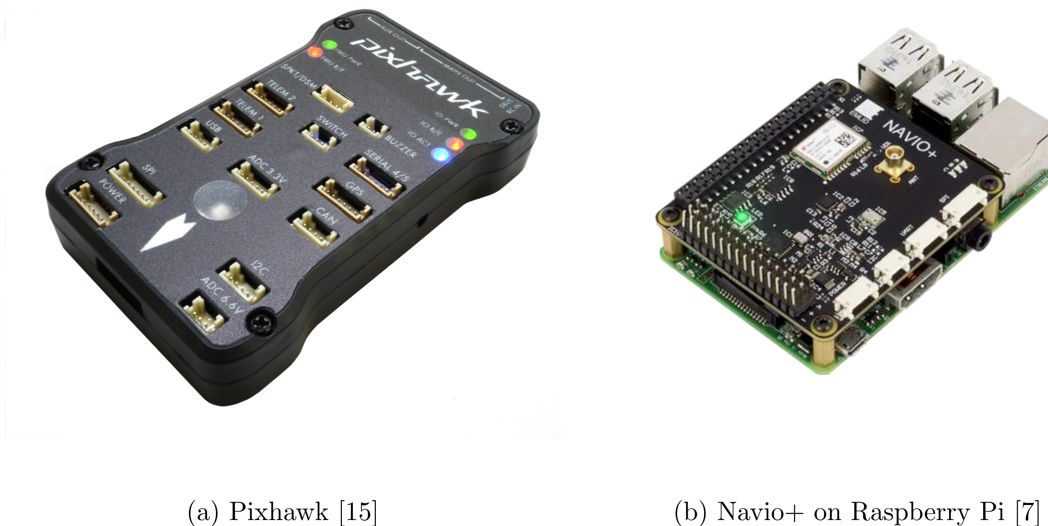
This interface works seamlessly with the Python modules related to DroneKit [1], which allows user-created scripts and modules to communicate with the flight controller program using an abstracted interface. We receive information regarding the vehicle position as the status updates come in. For example, we be notified of low battery levels automatically.

2.2 Hardware

In order to control an unmanned vehicle with our own mission scripts, we need a flight controller that can communicate with a companion computer using the toolchain described in Section 2.1. It also has to be compatible with the vehicle's motor control, which is a matter of having the correct wiring.

In the open source community for autonomous or remote control of drones and rovers, there are several hardware lines for flight controllers. In this section, we discuss two relevant ones: the Pixhawk from PX4 [15] and the Navio+ from Emlid [7]. Both are compatible with the ArduPilot flight controller and the MAVLink protocol. They can connect with various peripherals through external ports.

Often, the flight controller is combined with a companion computer, which is usually a Raspberry Pi (RPi). This miniature computer has the dimensions, computational power and hardware interfaces that we want. In the remainder of this section, we compare the two flight controllers as a means of determining which best fits our needs.



(a) Pixhawk [15]

(b) Navio+ on Raspberry Pi [7]

Figure 4: Commonly used flight controller hardware in unmanned vehicles.

The Pixhawk is a mature flight controller, shown in Figure 4a. It has failsafes and internal backup systems, to ensure that it remains safe in, e.g., a flight. It can connect with many peripherals, and the ArduPilot documentation uses the Pixhawk as a reference for many demonstrations on how to assemble it. Although it is relatively small, it does have an awkward model. It cannot mount a companion computer, instead it needs to connect to the Raspberry Pi using wiring. Also, there may be some deficiencies with the RC pins that make it potentially unsafe to connect peripherals that require a high power.

Navio+ is a more modern but well-backed project from Emlid. It has power monitoring and a triple redundant power supply, so that the flight controller always has power in a reasonable mission radius. There is a bunch of documentation from Emlid [7], which is not always very specific. The Navio+ connects directly with the Raspberry Pi via the general purpose input and output (GPIO) pins by mounting on top of it, so that it is smaller and wiring as shown in Figure 4b. The RPi GPIO pins extend to ports for peripherals so that either the flight controller or the RPi itself can access them. The additional *servo rail* has its own power supply for motor and servo control, so that it cannot steal power from the main hardware.

3 Drones and other vehicles

This section introduces the main concepts regarding autonomous and remotely controlled vehicles. We distinguish the different types of unmanned vehicles and specify their benefits and drawbacks.

In Section 3.1, we investigate the expectations of the vehicle and the hardware mounted on it and deduce the physical requirements. Section 3.2 describes the current regulation regarding flights with drones and other UAVs within applicable jurisdictions.

3.1 Requirements

The use of drones is a very compelling idea, however one should be cautious not to decide too quickly. Unmanned vehicles come with various features and exist in many price ranges. The ability to mount a camera on a drone may be helpful for replaying the mission, even if we do not use image processing to detect objects. However, a fully advanced system with a large number of rotors, servos and mount points is exorbitant for our research.

It makes sense to describe what the project necessitates as a means for selecting what kind of unmanned vehicle fits with the purpose. The following properties should be taken into account:

- The vehicle must be able to propel itself autonomously without wires. The vehicle may have miniature dimensions, with no driver on board. Instead, the motor and steering controls must be compatible with the hardware flight controllers described in Section 2.2 and the software toolchains running on it, as shown in Section 2.1.
- The vehicle should also have some form of remote control. The toolchain supports additional ground stations that can take over control in case the autonomous mode fails, which is desirable for safety but not required in controlled situations.
- The vehicle should operate at a reasonable speed. Our missions are not lengthy, but the vehicle should have a reasonable range depending on the battery capacity and motor power. This cannot take such a long time that the experiment's measurements are no longer useful.
- The vehicle must be able to carry its payload without hindering its operations. While the vehicle does not need to carry much else than the flight controller, companion computer and mounted peripherals such as sensors, it must carry these components without damaging or losing them.
- The vehicle must have a way to stabilize itself such that it can stand still at one position. This is useful for accurate measurements. Especially when it is close to obstacles, the vehicle may need to halt and decide a new route in place.
- There must be a means to point the distance sensor in different directions on the horizontal plane. As explained in Section 4.2.5, the distance sensor works in a straight line, and rotating it allows us to detect more of the environment. We can achieve this by mounting the sensor on a servo, or by rotating the vehicle as a whole, if possible.

3.2 Drone regulations

There are few constraints to the use of autonomously or remotely driven small ground vehicles. One must avoid public roads and busy locations. This means such miniature rover cars can be driven around inside or outside, which is helpful for comparisons between test runs and actual demonstrations.

On the other hand, there are laws and regulations regarding flights with unmanned aerial vehicles. These rules differ by jurisdiction and can be more or less stringent depending on location. In our project, we only plan flights above the mainland of the Netherlands, therefore we describe the regulations that hold in the jurisdiction of the Netherlands.

As of July 1st, 2015 the laws and regulations were tightened, which further restricts the use of drones for recreational or non-professional commercial use. Unlawful use or breaking these regulations results in a fine or a confiscation of the drone [5].

Drones or UAVs are sometimes called *model air planes* or lightweight unmanned air planes. The following restrictions apply to these vehicles [21]:

- A drone must give priority to planes, helicopters, gliders, balloons and airships.
- In other cases where two UAVs cross each other at or around the same altitude, the UAV must grant priority to the vehicle at its right hand side. (Note that this regulation does not provide a solution in the case of an impending frontal collision.)
- During flight, the vehicle and the surrounding air must be clearly visible from the ground. This means the weather conditions and the surface must be appropriate — flying over a dense forest might not be allowed in this sense. Furthermore, the flight can only be performed during the daylight period, with an extra 15 minutes margin before sunrise and after sunset. The controller of the vehicle must keep a clear sight of the model plane.
- It is not allowed to fly over urban areas with contiguous buildings, works of art, or industrial areas, including harbors. Also restricted are crowded areas, railway lines and roads where the speed limit is 80 km/h or more (including highways). It is advised to keep a 150 meter distance from these areas [5]. It is not allowed to fly within a 3 km radius of any kind of airport. A drone may not fly higher than 120 m.

For professional drone operators, there exists a special certificate which simultaneously requires them to adhere to more criteria, but also allows a little more leniency with the rules. In any case, the following restrictions that normal planes have, no longer apply:

- There is no need to have an altimeter or other navigation devices, and a flight plan does not need to be submitted. Also, the vehicle does not need to reply to requests from an air traffic control station.
- The captain does not need to be on board of the unmanned vehicle.
- It is allowed to make photos from the sky. A license to make aerial recordings is no longer necessary. It is still not allowed to make photos of military bases [5].

The important bottom line here is that there are no regulations regarding flying a drone indoors, at least as long as the owner of the building allows such activities. It is therefore possible to start testing a drone within a large hall. This does have risks involved with nearby objects and people, thus the coverage of insurances should be examined beforehand.

Flying a drone outside buildings is a lot more restricted in this sense. The only suitable locations are large, open spaces without any roads or other infrastructure nearby. Perhaps a location at a university campus could still suffice, given that there is some spacing between buildings or undeveloped pieces of land. We must still keep a safe distance from railways and busy roads.

This means that for safety, one would have to travel to a more natural landscape. In the Netherlands, one is often close by roads or railways, and having to avoid them makes it difficult to actually reach that location.

4 Implementation

In this section, we describe our contributions to the toolchain regarding drone operations and simulations. We first introduce the new features in an overview of the toolchain, where we describe the components in a summarized form. This overview is given in Section 4.1. We explain the components that are mostly related to the simulations in Section 4.2, and the mission components that run in simulations or physical tests are laid out in Section 4.3.

All components are written in the Python programming language and are meant to either run on the vehicle's companion computer, or on a developer computer for simulation.

4.1 Overview of the components

The overall toolchain consists of a number of components related to the distance sensor, the environment and the trajectory path missions. Some parts of each component are related to simulations only, some of them only function in a physical environment with the actual hardware, and some mission components function in any environment.

The rationale between this hybrid setup is that it becomes very easy to swap out different components with each other, which makes it easy and predictable to work with the same code for both simulations and physical runs. This reduces the chance that problems arise with the components themselves in actual demonstrations, since the code has already been tested thoroughly.

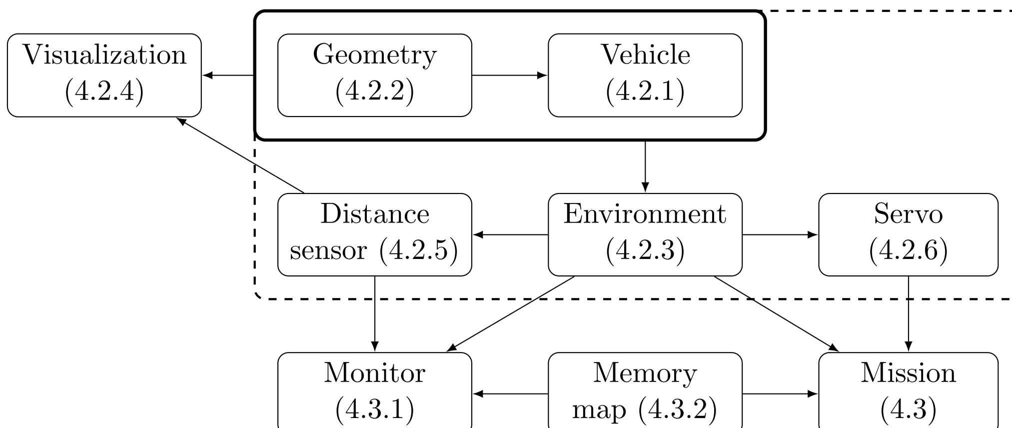


Figure 5: Diagram of components in the toolchain. The arrows indicate that the latter component depends on the first component.

The simulated components fall into a number of groups built on top of each other. Figure 5 summarizes the dependencies between these groups and components. In the core, there is a physics and geometry engine that a simulated vehicle can make use of. This is used not only by the simulated vehicle but also by the environment simulator. On top of these two components is a group of sensor simulation classes. Finally, we develop visualizations for the simulations.

The simulated environment and sensor components have a physical variant that only works on the companion computer of the unmanned vehicle, after it has been connected to all peripherals. Additionally, there are components that control the current behavior of the vehicle and determine its mission. These components work either within a simulation or on the physical version.

4.2 Simulations

Among the core simulations are a simple simulation engine for a vehicle (Section 4.2.1), a number of utilities for geometry and physics simulation (Section 4.2.2), and a scene loader and environment simulation helper component, as described in Section 4.2.3.

On top of this, we provide a simulated version of the distance sensor that makes use of the scene to determine distances (Section 4.2.5). A simulated servo component doubles as a tracking object for the current servo state: more details on this are given in Section 4.2.6. The environment and current state of a vehicle are visualized with components as shown in Section 4.2.4.

4.2.1 Vehicle

The vehicle component is an optional part of the simulation that implements simplistic movement. As we detail in Section 2.1, there already exist tools for simulating drones, planes or rovers. The ArduPilot engine [2] works with the actual binaries that are also compatible with the flight controller hardware in Section 2.2. It puts the software binaries inside a simulation loop, also called Software in the Loop (SITL), which makes the program believe that the hardware is physically moving around while it is actually inside a simulated environment.

The problem with the ArduPilot simulator is that it is slow to start up. Although the binaries only need to be compiled once for each vehicle type in the optimal case, the time that the program needs to set up is very long. This is due to numerous components being initialized and the load time of some simulated libraries.

Additionally, while ArduPilot is able to simulate the randomness that a vehicle would experience in reality, such as imperfect sensor readings and GPS antenna failures, it does not provide precise means to simulate an environment. It is possible to load terrain data with altitudes determined by satellites. This is very coarse satellite data for our purposes, at a resolution of around one pixel per 90 meters [8]. Also, a very simple polygon-defined *geofence* can be provided to the vehicle, which disallows the vehicle from leaving the bounding box of the geofence. We rather want to disallow the vehicle from *entering* certain physical objects. Both the terain data and the geofence are stored on the flight controller, instead of defined and verified by the simulator itself.

For these reasons, we implement our own “mock” vehicle which is able to emulate most of the simple movement schemes that the flight controller also performs. The replacement engine has support for the same command sequence storage defined in ArduPilot and the MAVLink protocol [14] described in Section 2.1. When given a target location, the vehicle first changes its attitude to turn itself in the right direction and then moves to the given location. The movement updates happen slowly based on intervals between checks of the location. We also keep the current velocity of the vehicle in mind. Changing the speed of the vehicle makes it move in its current direction until other orders arrive.

The mock vehicle component supports various ArduPilot vehicle modes. It also tracks the home location and enables other components to check whether a location update is correct, such as whether the vehicle did not move into an object.

4.2.2 Geometry

The geometry component is a physics engine, featuring a large number of methods that perform calculations and conversions with coordinates, distances and angles.

Navigational tools such as flight controllers define the *bearing* as the direction in which a vehicle is moving. This direction is defined in terms of three axes, pointing in the north, east, and upward direction. In standard convention of right-handed Cartesian coordinate systems, these directions are also called z, y, and x axes, respectively. A vehicle has a certain direction that need not be in one of these perpendicular directions, therefore we define this direction by means of three angles: the roll, yaw and pitch. These rotate around the z, y, and x axes, respectively. However, the order in which they are applied in order to get the actual *attitude* of the vehicle is roll, pitch, and then yaw [4].

We make use of these angles in various conversion methods, which also need to track whether the given angles are bearings, which start at 0° facing forward and increase clockwise, or mathematical angles, which start at 0° facing orthogonally rightward and increase clockwise. We convert between the variants as follows: a bearing β in radians is equal to the angle α in radians when $\alpha = -(\beta - \frac{\pi}{2}) \mod 2\pi$ and vice versa, $\beta = -(\alpha - \frac{\pi}{2}) \mod 2\pi$.

We also have methods for comparing angles, such as calculating the difference between two angles a_1, a_2 and finding the direction in which a_1 reaches a_2 the quickest. These methods also accept angles that are in different periods of the unit circle. The radial difference ignoring periodicity is $(a_1 - a_2 + \pi) \bmod 2\pi - \pi$. The sign of the difference is unrelated to which angle is smaller than the other. Instead, it indicates whether counterclockwise or clockwise rotation brings a_1 to a_2 the fastest, based on negative or positive sign, respectively.

Aside from bearings and angles, the other geometry utilities mostly deal with locations and distances. The geometry has two modes for the coordinate system: all locations can either be defined in meters or using geographic coordinates of the WGS84 system [20]. In the former system, we calculate distances between locations using Euclidean distance, and determine new locations based on trigonometric angles and distances. The WGS84 system takes the curvature of the earth into account, while the basic operations remain reusable. The locations can be relative to a home location, or a global coordinate.

Finally, the geometry contains various intersection methods. This allows detecting whether a given point is within a 2D polygon, a 3D plane or a 3D polygon. By extension, we can detect the intersection of a line, a ray or a line segment with any of these shapes. Section 4.2.5 provides more details on the use of these methods.

4.2.3 Environment

We implement an environment simulator that makes use of the physics simulators from Sections 4.2.1 and 4.2.2, respectively. This component keeps provides a central location for retrieving current vehicle properties. Also, the environment instantiates and tracks the sensors related to the environment, such as the XBee sensor interfaces, the distance sensor from Section 4.2.5 and the servo container described in Section 4.2.6.

The simulated sensors can be easily replaced by physical versions. Whether we use a real or virtual distance sensor depends on the type of environment we have, whereas the XBee sensors can work with either the simulated socket interface or the actual communication sensors connected to USB ports [17]. It can also work standalone with no sensors at all.

The simulated environment in particular has the option of loading a scene file. If it is not used, a simple environment with some wall-like objects and poles is created. The importable scene files have to be Virtual Reality Modeling Language (VRML) files. VRML is a stable and fairly old specification for structured storage of information about static objects and their behavior within a 3D scene [11]. An example scene is shown in Figure 6.

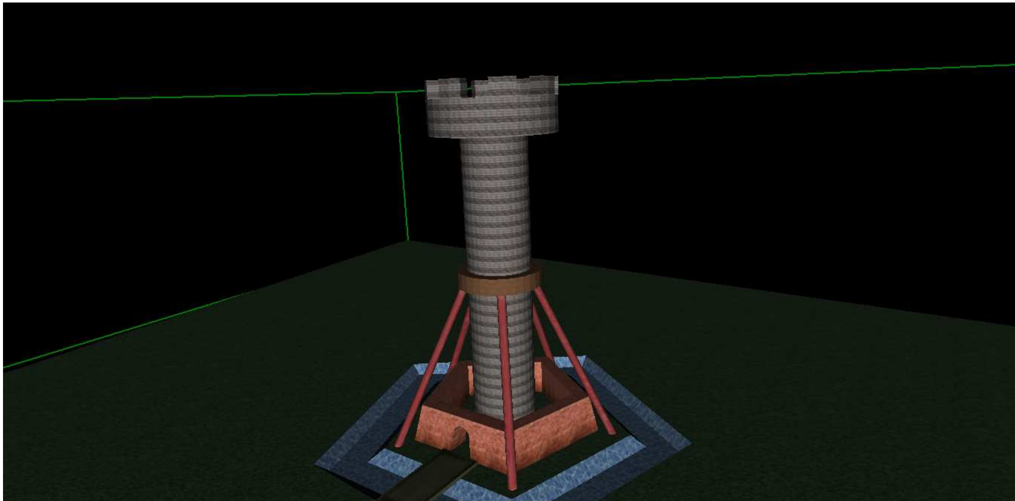


Figure 6: Visualization of a VRML file: a castle with a large pillar. Screenshot from the `view3dscene` model browser from the Castle game engine [13].

Developers can easily create 3D game engines using VRML files. The simple but extensible format has been used for simulation specifically for drones [10]. There are multiple libraries available to import them. Simpler polygon file formats only support defining one object without scene information. While VRML has been superseded by newer formats such as X3D, it is still supported and many scenes are available or exportable to the format [13].

The VRML files contain nodes of various types, and the nodes can be nested in one another. A simple use case for this is a `Transform` node, which defines rotation and scaling operations and contains other nodes that are translated according to the affine matrices that are determined from these operations. Objects can also be defined using a name and reused later on in the file, except when nested within the node itself. This means that the *scene graph* of all the nodes and their dependencies is a directed acyclic graph.

A node of the `Shape` type contains coordinate information. These define the polygons that make up the object, as well as texture locations and so on. We are only interested in the

polygon coordinates of these actual nodes, and the transformations we have to apply to make up the entire scene.

We load all these objects so that it can be used by the distance sensor as described in Section 4.2.5 and for visualization in Section 4.2.4.

4.2.4 Visualization

The ArduPilot simulator has a simple access script with options to monitor the mission via a terminal, including outputs from the missions described in Section 4.3. Secondly, one can verify whether the flight controller MAVLink connections are instantiated via another terminal emulator. Finally, there is an option to view a map of the surroundings around a home location on Earth.

While this map is useful for viewing the relative distances of mission waypoint locations, it does not provide enough visual cues for investigating certain problems or scenarios. The top-down overview map still has its own purposes, and in fact is an inspiration for parts of the memory map. We describe the memory map itself and its visualization separately in Section 4.3.2.

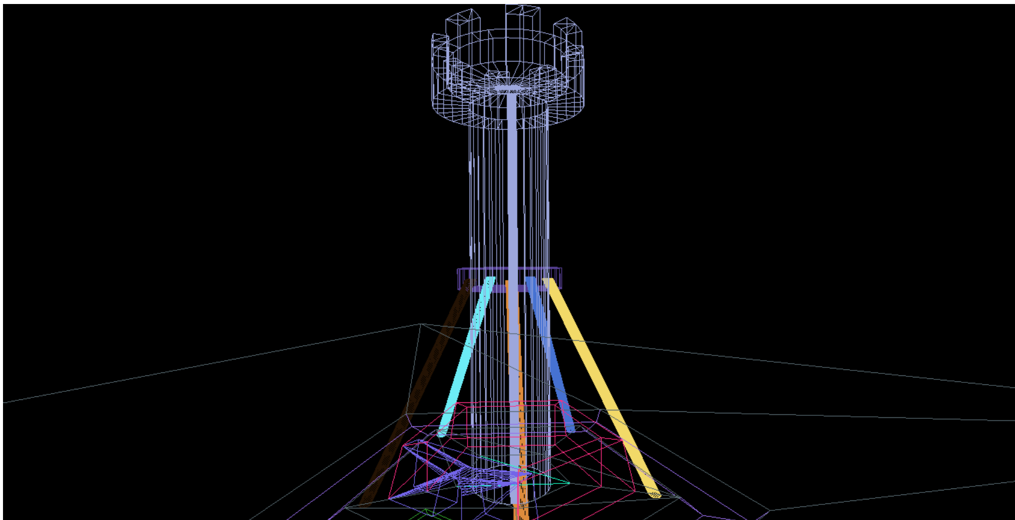


Figure 7: Visualization of the `castle` VRML file as polygons within the viewer.

The additional visualization that we need is based on the simulated environment with the 3D scene shown in Section 4.2.3. After we load a VRML file, we can display the polygons using OpenGL in a viewer window as shown in Figure 7. During a simulated mission, we can display what the vehicle would currently see at its given position and angles. There is also a script that instantiates the viewer and provides interactive control over the camera within the environment using the keyboard or the mouse.

Controlling the camera in either case is a matter of its own. While the position could be achieved by simply translating the entire scene loaded into the viewer in the opposite direction, the same cannot be done with the rotational angles. This is because the yaw, pitch and roll are applied one after another, making each axis relative to the previous one. There is also the problem of *gimbal lock*, where one axis of rotation becomes “locked” with another one due to deficiencies in the rotation matrices. Such an event would make it difficult to control the camera correctly.

Instead of using matrix transformations, OpenGL provides a method of controlling the camera using a number of vectors. Not only the position vector is stored, but also the vectors relative to the camera that face upward, forward and rightward. We can convert the current yaw, pitch and roll to orthonormalized vectors, and then convert them back again after movement to receive updated angles [4].

4.2.5 Distance sensor

The unmanned vehicle uses an ultrasonic distance sensor as its main source of information for the purpose of object detection. This sensor sends sound waves in one direction, and waits for the echo signal to return. This method works in a nearly-straight line. It detects almost all types of objects up to a certain distance, regardless of texture. If the angle of incidence is too large, then the sensor may not receive the echo signal at its location [6]. This simplistic directional sensor is a limitation to the sight of the vehicle [17]. We can improve the range of the vehicle's scanning ability by rotating it around itself in one place, which is especially possible with drones. We can also mount multiple distance sensors on the vehicle and use servos to rotate the sensors themselves, as explained in Section 4.2.6.

We focus on simulating the distance sensor within a virtual environment. We simulate the range of the physical distance sensor by ignoring distances above a certain value. We use geometry and intersection calculations to determine the distances algebraically.

There are three types of objects that we can simulate in our environment. In Section 4.2.3, we introduced the VRML format, which produces objects based on 3D polygon shapes. Determining the distance to these polygons is possible, but non-trivial. We instead first describe two simpler types of virtual objects and how we detect their distances.

The simplest object that we can detect is a cylindrical pole. It stands perpendicular to the ground, and it has a certain circular radius and height. When a UAV flies above this height, it is not visible and the vehicle never collides with it. Otherwise, the pole is only visible if the vehicle's angle is between the two angles of the sides of the pole as seen from the vehicle's location. We can easily determine these angles using the position of the vehicle and the radius of the object. One add margins to allow the distance sensor to detect or miss when it is flying near the pole's height or close to the sides of the pole.

We can also apply the concept of objects defined by their height to more arbitrary shapes. We can define an object by a polygon, where each point contains the 3D positioning information of a vertical edge reaching up to that point. We define the polygon P as an ordered vector $P = \langle p_0, p_1, \ldots, p_{n-1} \rangle$ of $n \geq 2$ points. Then each pair of points $(p_i, p_{i+1 \bmod n})$ — including the pair of points consisting of the end and start point — in fact generates a *face* of a 3D object. The object basically consists of vertical walls, whose topmost edges can slope diagonally, but always connect to the next wall at an equal height.

We need to perform several steps to determine the distance to such an object. We first check whether the vehicle's angle α is within the minimum and maximum angles from the vehicle to the points of the polygon, ignoring height differences and making sure that the periodicity of the angles is the same by checking in which *quadrant* of the coordinate system each angle goes. Then for each pair $(p_i, p_{i+1 \bmod n})$ we calculate several line components within a 2D system, ignoring altitude: the slope of the vehicle's angle $m_v = \tan(\alpha)$ with α an angle in radians, and its latitude intercept $b_v = y_v - m_v \cdot x_v$, where y_v and x_v are the vehicle's latitude and longitude coordinates, respectively.

For the edge pair p_i and $p_{i+1 \bmod n}$ we also determine the line of the edge e. In case the longitudes of the points are the same, the line is vertical within the 2D system, thus $m_e \rightsquigarrow \infty$ and $b_e = 0$, and we use $x_s = x_i$ and $y_s = m_v \cdot x_s + b_v$ as the intersection point coordinates. Otherwise, the edge line has slope

$$m_e = \frac{y_{i+1 \bmod n} - y_i}{x_{i+1 \bmod n} - x_i}$$

and

$$b_e = \begin{cases} y_i - m_e \cdot x_i & \text{if } y_i \leq y_{i+1 \bmod n} \\ y_{i+1 \bmod n} - m_e \cdot x_{i+1 \bmod n} & \text{if } y_i > y_{i+1 \bmod n} \end{cases}$$

If the two slopes m_v and m_e are equal, then the ray from the vehicle never intersects with the edge. Otherwise, the latter case has an intersection point

$$x_s = \frac{b_e - b_v}{m_v - m_e} \qquad\qquad y_s = m_v \cdot x_s + b_v$$

We then determine the distance d to this point on the same altitude using Euclidean distance and calculate the altitude using the vehicle's pitch β using $z_s = z_v + \beta \cdot d$. We still have to check whether the intersection point is actually on the edge, which is the case using our construction if and only if the distance between the two points is no larger than the distances from the detected point and the edge points.

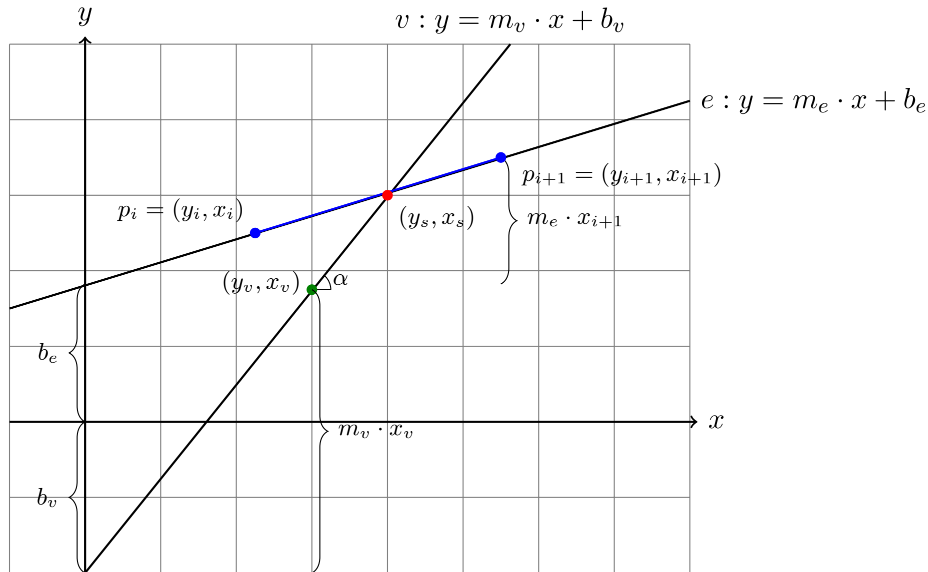


Figure 8: Example coordinate system with vehicle and edge line functions.

Using this construction, which is summarized in Figure 8, the object has detectable walls, but no roof. We can make it solid using an altitude check and a point in polygon algorithm using the topmost polygon P. We pass a ray from the current vehicle position eastward, and count the number of intersections with edge pairs $(p_i, p_{i+1 \bmod n})$ from P. The vehicle is inside the object if and only if an odd number of such intersections occur [22].

The distance calculations for these simple objects are somewhat reusable for the VRML object shapes. Once we determine an intersection point, we can check whether it is inside one of the polygon faces of an object by projecting the point and polygon in 2D, ignoring the least relevant coordinate. Determining this and the potential intersection points requires some preliminary steps, however.

A VRML polygon needs at least 3 points in order to be considered complete. The polygon $P = \langle p_0, p_1, p_2, \dots, p_{n-1} \rangle$ usually lies on one 2D plane within the 3D space. We determine this plane's normal vector r by calculating two vectors $u = p_0 - p_1$ and $w = p_0 - p_2$ and taking their *cross product*:

$$\begin{aligned} r = u \times w = (&u_1 \cdot w_2 - u_2 \cdot w_1, \\ &u_2 \cdot w_0 - u_0 \cdot w_2, \\ &u_0 \cdot w_1 - u_1 \cdot w_0) \end{aligned}$$

We calculate the vector ℓ of the ray extending from the vehicle's position q by taking the difference of two points on the line: $\ell = q - q^*$. Then the ray and the plane intersect if and only if the dot product $r \cdot \ell$ is strictly positive.

We obtain the intersection point t through some additional vector calculations:

$$\begin{aligned} c &= \frac{-(r \cdot (p_0 - q))}{r \cdot \ell} \\ t &= q + c \cdot \ell \end{aligned}$$

However, if $c < 0$ then the point is actually on the line extending from the vehicle in the other direction and should not be detected by the distance sensor. The point t must still be projected and checked whether it is inside the projected polygon, ignoring the ith coordinate, where $i = \operatorname{argmax}_j |r_j|$, so that the projected surface area of the polygon is maximal.

4.2.6 Servo

A *servo* is a small motor that can rotate an axle, which is often used to steer autonomously. The simulated distance sensor mentioned in Section 4.2.5 always points into one direction, and measures the distance in a straight line. A physical version shows similar behavior [17]. This may be problematic in case there is an object that is close to the vehicle's path, but not completely straight ahead. It is definitely more secure if we can scan 360 degrees of the surroundings continuously.

Instead of rotating the entire vehicle in place, which increases the time needed to perform the mission, one can physically mount a distance sensor upon a servo motor. The servo can be steered by the flight controller based on commands in the mission. Servos operate by sending a power signal at high and low voltages at a certain frequency. The width of the pulses sent in this way results in an average voltage between the low and high voltages. This then determines the angle that the servo maintains. This technique, known as *pulse width modulation* (PWM), controls the servo and makes it possible to read out its current state. We convert the desired angle and the current PWM value back and forth in this way. The current angle affects the calculations related to the distance sensor.

4.3 Missions

The mission components determine the trajectory that the vehicle should take in order to achieve its goals, which include visiting a number of preselected waypoints and safely returning home without colliding into other objects. Whereas the components described in Section 4.2 mostly deal with simulations and abstractions of reality, the mission has

physical control over the vehicle. It is at a higher level than the physical engine and sensor interfaces. We make use of the sensors to deduce what the vehicle should do to execute the mission without failures. The mission is a basic AI agent that operates step-wise, determining its next step based on its current situation.

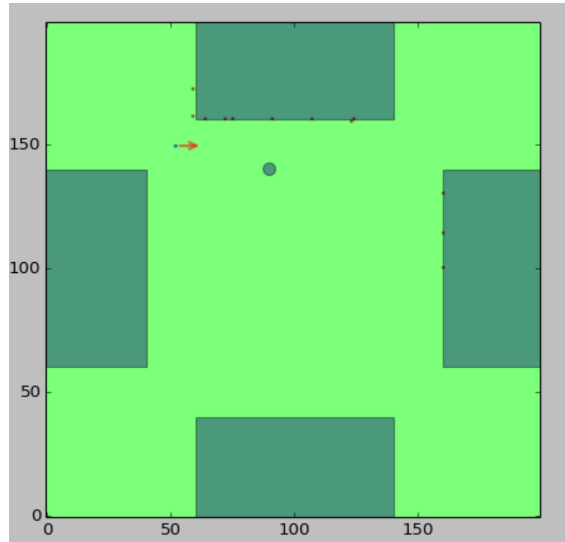


Figure 9: Memory map plot of a simple projected environment. We detect a few points on the northern (top) and eastern (right) objects. The arrow shows the vehicle's positioning.

We categorize the individual missions into two types: pre-planned automatic missions and dynamic missions that react to new situations. Both types operate autonomously. We show several examples of both modes in Sections 4.3.3 and 4.3.4, respectively. The component that controls the entire mission by checking sensors at a regular interval is aptly named the monitor. We introduce this component in Section 4.3.1, before describing the memory map that the monitor keeps track of in Section 4.3.2.

4.3.1 Monitor

The monitor keeps track of the sensor information and passes this along to the actual mission. Like other components related to mission control, the monitor activates on a regular schedule. The monitor supervises the measurements on each step. It determines whether the measured distance to objects is too far away, so that it is dropped in case the distance sensor has a glitch in its exactness. Also, a very low distance to objects halts the vehicle so that it must either rescan or completely stop its mission. Otherwise, we pass the data on to the mission for additional safety checks.

The monitor also updates the memory map with the distance measurements and can display a plot of the map. We provide more details on this feature in Section 4.3.2. The monitor is also responsible for tracking mission progress, and for safely returning home when reaching the final goal or when problems arise. This includes landing the vehicle in case it is a UAV or halting all the motors.

Certain parts of the monitor can run on their own thread in a multiprocessing environment. We use this to let the XBee communications sensor control its own scheduling loop while still integrating all sensors into the mission monitor.

4.3.2 Memory map

The memory map keeps track of detected objects. It plays a vital role in the missions that use it to determine a path around the objects in order to avoid collisions. The mission monitor from Section 4.3.1 is responsible for constantly updating as new measurements from distance sensors arrive.

Assuming that the vehicle operates on one altitude level, the memory map can be defined as a two-dimensional array that stores a grid representation of the known environment. Each cell in the grid relates to a square area at the given altitude. The value that we store in the cell determines whether there is likely to be an obstacle within this area.

The memory map is finite, thus there is a *space size* in which we assume the vehicle to be operating. If the vehicle goes outside this area, then we cannot ensure the vehicle's safety. It then stops or returns to its starting location.

We can configure both the space size and the *resolution* of the cells, which is deduced from the number of cells per meter along each dimension. This allows missions within a confined space to have some more precision in the memory map, so that it may be able to find its way around objects with relative ease. Missions in an open environment have a lower resolution and are thus more careful with any oncoming objects.

Although we currently only support a two-dimensional memory map, we can easily extend it to include the altitude component as well. The memory map keeps in mind that the vehicle may be slightly tilted, causing the detected object to be at a different position and altitude. The reason for an initial 2D approach is that the memory map can be easily inspected and visualized as shown in Figure 9. A three-dimensional map is only useful if the vehicle operates on multiple altitudes, which is usually not the case for a rover on a flat surface or a simple drone flight.

4.3.3 Planned missions

The first and simplest vehicle mode is the automated mission, designated as the `AUTO` mode in the ArduPilot toolchain. This mode works by sending a sequence of commands to the flight controller, involving a number of waypoint locations. After the vehicle is set up, the vehicle has to move to the given locations in that order. If the vehicle is a UAV, then it first takes off to the specified operating altitude.

This mission is easy to set up by programmatically defining the waypoints, for example by converting their distances to the starting position to actual coordinates using the geometry utilities from Section 4.2.2. The flight controller then determines the paths to be taken to reach them. An example mission of this type is the `Square` mission, which flies in a square around the central home location. This is based on a simple DroneKit example [1]. One can also use a mission planner to pick the waypoints on a map, or replay a mission performed by a human.

A disadvantage to automatic mode is that one cannot deviate from the chosen paths aside from skipping a waypoint. The `AUTO` mode does not keep collisions in mind. In order to load a new mission, the vehicle has to wait until all the new waypoints are loaded and validated, which costs time. Instead, it is possible to let the companion computer track the waypoints in the mission on its own, of which we shown an example in the `Pathfind` mission in Section 4.3.4.

4.3.4 Guided missions

The `GUIDED` mode allows more freedom during missions than the `AUTO` mode of Section 4.3.3 does. This is because this mode does not make use of a predefined mission passed in the sequence of commands, but instead accepts commands on the go. These commands can steer, rotate and move the vehicle into certain directions, adjust the operating altitude and speed, and so on.

This allows the mission and the monitor to react on circumstances during the mission that were not precisely foreseen beforehand. The possibly dangerous situations can then be handled by routines and AI algorithms that decide on the next step that should be the safest and most useful according to the AI actor.

A very simplistic mission that operates in this mode is the `Browse` mission. When the vehicle is *browsing*, it attempts to stabilize itself upon one position. It then turns itself around so that the distance sensor from Section 4.2.5 can check its surroundings, step by step. After rotating in a loop, this gives us some information about the potentially unsafe environment we are in. This could not be performed in an `AUTO` mission since yaw changes interfere with the automatic path traversal. While the `Browse` mission is quite useless on its own, it is a helpful test to see if the vehicle functions correctly without any serious danger. Also, this mission is a building block for other missions, that can use the browsing routine when those missions are actually at unsafe locations. This allows them to achieve the goals of visiting locations and returning home safely. Finally, the `Browse` mission can make use of the servos from Section 4.2.6 to only rotate the distance sensors and not the whole vehicle. We deduce which servo to use and what angle to pass it from the desired yaw angle.

Another mission, `Search`, actually moves around in the environment. It attempts to stay close to the relevant objects, but not crash into them. To do this, we first browse around the vehicle using the `Browse` mission. We then decide to move into a direction that is close to but not directly onto a detected point. The chosen direction is the one with the furthest point and with the most open space next to it in a weighted equation. This should keep the vehicle within a zone of interest around the convex hull of the object, and not fly into the interior of objects. The `Search` mission starts browsing again after traveling a little further than the chosen point, or when it is close to a new object. `Search` is mostly based on readings from the distance sensor from the current browsing sweep, and does not make full use of the memory map.

Finally, the `Pathfind` mission does rely on the memory map in order to determine and correct a path. `Pathfind` works with waypoint locations much like the planned missions from Section 4.3.3. The vehicle must visit the waypoints in order, but the path to it can be altered in case there are objects blocking the current path. We use the A* algorithm [9] to find a new path in case we detect a nearby object. To improve the algorithm, we browse the surroundings when we are close to an object while staying in one place. We then use all the detected points in the memory map to specify disallowed paths. The A* algorithm works on graphs in general, and each point in the memory map has eight neighbors in the cardinal and diagonal directions. The monotonous cost and estimation functions are determined by the distance between the two points. We achieve a slight speed-up by using the indices of the memory map rather than locations themselves, at the cost of some preciseness.

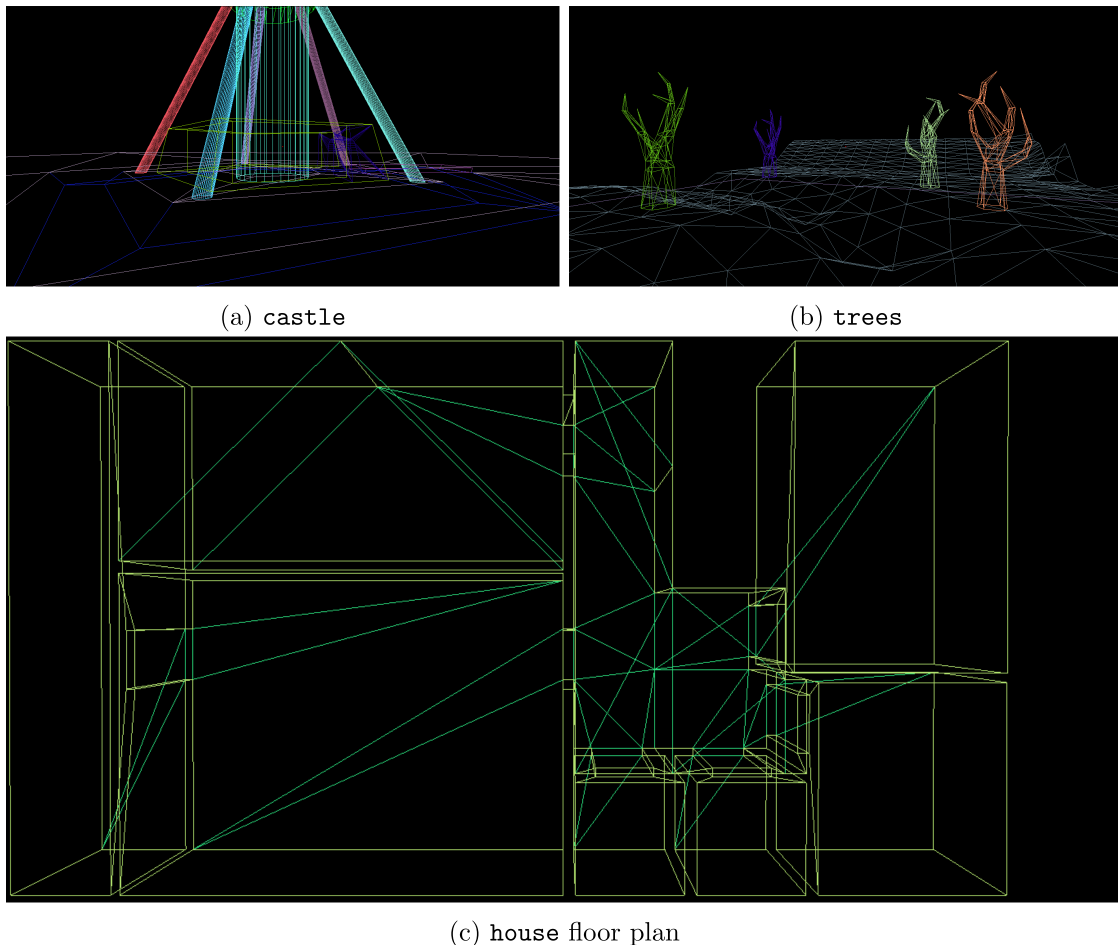
5 Experiments

We propose a number of experiments to determine the functioning and effectiveness of our implementation from Section 4. We perform these experiments in a simulator. We use different vehicle parameters that may influence the outcome. We provide the vehicle with various environments and determine whether the missions function normally in them.

To find out how much of the environment a mission detects, we calculate the number of detected objects in the memory map. We specify the experimental setup in Section 5.1. In Section 5.2, we provide the results and observe some of the behavior of the tested vehicles.

5.1 Setup

The experiments mainly focus on the effectiveness and completeness of the memory map, which we describe in Section 4.3.2. We also investigate whether the mission succeeds and how much time it takes to finish, which can be determined using an in-program timer.



(a) `castle` (b) `trees`

(c) `house` floor plan

Figure 10: Visualization of the environments in which the vehicle is tested. The starting positions for the `castle` and `trees` scenes and the floor plan of the `house` are shown.

Comparing the memory maps between various runs is more difficult in this regard. We can look at the monitor plots from Section 4.3.1, but it does not provide a ground for exact measurements. Instead, we use the memory map data itself.

Every time a mission detects an object and stores it in the memory map, we expect other missions to detect that object in the same environment as well. Most missions do not cover the entire surface of their operating altitude, thus they never detect every single point. However, we can determine which mission detects the most points.

The most important goal of our missions at this point is to finish safely. Thus the vehicle should never move into an object that it had not detected. Our environment simulation has a feature that ends the mission in case the vehicle moves through an object's polygon, so we can determine this without the memory map. Still, we want to see whether the memory map is actually filled with that point in this case, since that can signal a problem with the detection or with the vehicle moving in directions that are not necessarily safe.

In order to speed up and simplify the experiment setup, we run the simulated vehicle under our own mock vehicle simulation from Section 4.2.1. We do not use the randomization that the ArduPilot simulator provides, since it can influence the results in multiple ways, including detecting points more often than in the baseline situation. We use different simulated environments from VRML files, which are shown in Figure 10. These are the `castle` pillar scene (Figure 10a), a closed scene within a `house` with a floor plan shown in Figure 10c, and a scene with uneven ground and `trees` (Figure 10b).

5.2 Results

We run almost 600 experiments using different permutations of parameters for the vehicle and the environment. These parameters include safety distances, memory map resolutions and space sizes. We change the default monitoring frequency of 0.5 seconds for the `house` environment to 0.25 seconds, so that the vehicle can quickly take actions. We describe the influence of some of these parameters and show the best results of each mission.

	`castle`	`trees`	`house`
`Square`	6	7	11
`Browse`	13	49	68
`Search`	140	65	258
`Pathfind`	74	37	109

(a) Best counts

	`castle`	`trees`	`house`
`Square`	00:01:46	00:00:00	00:00:10
`Browse`	00:08:49	00:10:02	00:10:02
`Search`	00:09:01	00:04:51	00:03:58
`Pathfind`	00:10:43	00:00:34	00:13:06

(b) Best times

Table 1: Best memory map counts and their respective mission durations for each mission and environment. Red colors mean that the mission failed.

In Table 1, we show the best results for each mission and scene with regard to the number of detected objects in the memory map. It is immediately visible from Table 1b that the `Square` mission does not perform well in these environments, because it quickly flies into an object. We chose the starting locations in such a way that the missions would at least come across an object. There is no other safe option than to end the mission in a preplanned mission from Section 4.3.3, so this is expected.

The `Browse` mission detects a few more objects in the environment as seen in Table 1a, despite not leaving its initial location. The `Search` mission does move around and detects many objects. The downside that this mission does not visit specific waypoints, which can hinder the tomography measurements. It also has a high chance of stopping prematurely due to it staying too close to walls.

The `Pathfind` mission instead tries to avoid walls and appears to be the safest in this regard. It still fail to finish its mission if it cannot find a suitable path, but if there is one then it also detects a reasonable number of objects on its way. The mission is however slow, because it has to wait when a new path has to be calculated.

These results should be reproducible in other environments as well, but the success of the missions is sensitive to the chosen parameters. The size of the search space depends completely on the environment, and needs to be tuned specifically for them. A higher memory map resolution works well in all runs. It is interesting to find that low padding and closeness parameters (both at 10 cm) gives a higher chance of success on most missions except `Pathfind`, which needs higher values so as not to trap itself in corners.

6 Conclusions

In this paper, we explore the problems and possibilities related to unmanned vehicle control in the context of a mobile radio tomography setup. We investigate the available toolchains and hardware that are able to operate motors, servos and other peripherals during a mission. The goal of the mission is to visit locations where we scan features of interest.

For this purpose, we need to detect objects during the mission without crashing into them. We implement a toolchain that determines a trajectory based on measurements from distance sensors and takes actions accordingly. We test the vehicle's behavior in simulations with different environments and parameters.

The experiments suggest that it is important to find a balance between an automated mission that simply follows actions and a free-form search that attempts to find the best locations to detect and avoid objects. Safety checks must stop the vehicle before it would move into the object. A mission that avoids these situations performs better than simplistic missions. Some of the current missions are able to detect objects and navigate their way in a virtually modeled environment, but it can take a long time to reach the goals, especially if we have little prior knowledge of a potentially hazardous environment.

6.1 Further research

While the experiments show the potential of the missions within a simulation, the goal is of course to have an actual vehicle moving around. In a physical environment, additional problems surface. The physical distance sensor may have problems with highly reflective or permeable textures. A comparison to other detection methods can be future research. The vehicle might also be inexact in its movement, such as a rover with imperfect wheels. The flight controller should be able to adjust for this. It uses multiple sources for calculating the location of the vehicle, including GPS and gyroscopes. A study into other solutions for accurate positioning is also useful. We can make assumptions that certain areas are safe and other areas contain the object of interest.

The missions can be repurposed for the WiFi tomography setup, however it may be better to create missions specific for this purpose. We can follow a trajectory that helps improve the reconstruction the most by moving the vehicles to obtain specific lines between them. The more of these lines intersect, the better the reconstruction becomes. We can create, adjust and compare such missions to see what performs well.

References

[1] 3D Robotics. DroneKit. Developer tools for drones. http://dronekit.io/ (accessed September 15, 2015).

[2] ArduPilot developers. Open source autopilot. http://ardupilot.com/ (accessed September 15, 2015).

[3] F. Bleichrodt. "Improving robustness of tomographic reconstruction methods". PhD thesis. 2015. ISBN: 978-94-6259-869-0.

[4] J. Doty. Determining yaw, pitch and roll from up and forward vectors. http://www.jldoty.com/code/DirectX/YPRfromUF/YPRfromUF.html (accessed October 26, 2015).

[5] Drones.nl. Wetgeving. *Dutch*. http://www.drones.nl/wetgeving/ (accessed September 11, 2015).

[6] ElecFreaks. HC-SR04 User Guide. http://www.elecfreaks.com/store/download/product/Sensor/HC-SR04/HC-SR04_Ultrasonic_Module_User_Guide.pdf (accessed January 26, 2016).

[7] Emlid. Navio+. http://www.emlid.com/ (accessed November 19, 2015).

[8] T. G. Farr et al. *The shuttle radar topography mission*. In: Reviews of Geophysics 45.2 (2007). DOI: 10.1029/2005RG000183.

[9] P. Hart, N. Nilsson, and B. Raphael. *A formal basis for the heuristic determination of minimum cost paths*. In: IEEE Transactions on Systems Science and Cybernetics 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[10] Z. Huang, A. Eliëns, and C. Visser. *3D agent-based virtual communities*. In: Proceedings of the Seventh International Conference on 3D Web Technology. ACM, 2002, pp. 137–143. DOI: 10.1145/504502.504525.

[11] ISO. VRML97: The Virtual Reality Modeling Language, Part 1: Functional specification and UTF-8 encoding. ISO/IEC 14772-1. 1997.

[12] O. Kaltiokallio, M. Bocca, and N. Patwari. *Enhancing the accuracy of radio tomographic imaging using channel diversity*. In: IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS). 2012, pp. 254–262. DOI: 10.1109/MASS.2012.6502524.

[13] M. Kamburelis. Castle game engine: view3dscene. http://castle-engine.sf.net/view3dscene.php (accessed November 17, 2015).

[14] L. Meier. MAVLink Micro Air Vehicle Communication Protocol. http://www.qgroundcontrol.org/mavlink/start (accessed October 27, 2015).

[15] L. Meier. PX4. Pixhawk autopilot. https://pixhawk.org/modules/pixhawk (accessed November 19, 2015).

[16] T. van der Meij. "Constructing an open-source toolchain and investigating sensor properties for radio tomography". Bachelor thesis, Leiden University. 2014.

[17] T. van der Meij. "Mobile radio tomography: constructing an open-source framework with wireless communication components". Research project report, Leiden University. 2016.

[18] A. Milburn. "Algorithms and models for radio tomographic imaging". Bachelor thesis, Leiden University. 2014.

[19] T. Müller and M. Müller. *Vision-based drone flight control and crowd or riot analysis with efficient color histogram based tracking*. In: SPIE Proceedings. Vol. 8020. 2011, pp. 1–14. DOI: 10.1117/12.884213.

[20] National Geospatial-Intelligence Agency. *Department of Defense World Geodetic System 1984, Its Definition and Relationships With Local Geodetic Systems*. Technical report. NIMA TR8350.2. 1997.

[21] Overheid. Regeling modelvliegen. *Dutch*. HDJZ/LUV/2005-2297. http://wetten.overheid.nl/BWBR0019147/ (accessed September 11, 2015).

[22] M. Shimrat. *Algorithm 112: position of point relative to polygon*. In: Communications of the ACM 5.8 (1962), p. 434. DOI: 10.1145/368637.368653.

[23] 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: International Micro Air Vehicle Conference and Competitions. 2011, pp. 40–47.

[24] J. Wilson and N. Patwari. *Radio tomographic imaging with wireless networks*. In: IEEE Transactions on Mobile Computing 9.5 (2010), pp. 621–632. DOI: 10.1109/TMC.2009.174.