Joshua David Springer

Computational methods for autonomous multirotor drone landing

Presented in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Department of Computer Science
School of Technology
Reykjavik University

0000-0003-0137-1770	Joshua Springer
ISBN 978-9935-539-80-9	Print version
ISBN 978-9935-539-81-6	Electronic version

This thesis has been read and approved by:

Gylfi Þór Guðmundsson — Date
Supervisor
Assistant Professor
Reykjavik University

Marcel Kyas — Date
Co-Supervisor
Assistant Professor
Reykjavik University

Joseph Timothy Foley — Date
Internal Committee Member
Assistant Professor
Reykjavik University

Ingibjörg Jónsdottir — Date
External Committee Member
Associate Professor
University of Iceland

Sebastian Scherer — Date
External Committee Member
Associate Research Professor
Carnegie Mellon University

Stefanos Vrochidis — Date
External Examiner
Researcher B
Center for Research and Technology Hellas

Acknowledgements

My sincere thanks go to Gylfi for his regular discussions that provided fantastic technical input, actionable advice, stubborn optimism and irony, and for his continuous logistical help – not least of which included several trips around Iceland to walk on lava flows with me in freezing rain and snow to collect data and conduct tests. Your support has been absolutely critical.

Similarly, my thanks go to Marcel for his continued financial and administrative support, *attention to detail*, groundedness, and sponsorship of grant applications that were necessary for this research.

I also thank Joe for his financial support in being able to participate in fieldwork in the Icelandic highlands, for purchasing drone equipment, and for in-depth review of this manuscript.

I would like to thank Hannes for being a great source of friendly positivity, advice, and supplies, especially in terms of manufacturing a lot of the custom parts that I have had to make over the years. You have taught me a lot of techniques that I will take with me, and you have been quite a positive force that I truly appreciate.

I also call back to my mentor for the final project course in undergrad – Eddie Weeks – plenty of whose techniques and approaches I use and appreciate every day.

To the department of geology at the University of Iceland, especially Inga and Will, I thank you for the opportunity to work with you to conduct fieldwork at the eruption sites in Reykjanes. This has broadened my understanding of the phenomena and helped my own research quite a lot. I would similarly like to thank the RAVEN team for the opportunity to conduct fieldwork in the Icelandic highlands, and Vatnajökulsþjóðgarður for permission to work in the Holuhraun region.

Ég þakka Skarpa og Völlu fyrir góðvild þeirra og fyrir að hjalpa mér að læra íslensku.

To Shawn, Sveinn, Shalini, Duncan, Benedikt, Karina, and Billy, thank you for your friendship, moral support, and the many great times we have shared and will continue to share.

To my teammates and friends from the Mongol Rally, especially Gary, Nick, Nikolaj, and Markus, not a day goes by that I don't remember our experiences fondly, and they have brightened my dark, Icelandic winters many times.

To my family, especially dad, mom, Jeremy, Megan, Jonathon, Jackie, Tegan, and grandma, thank you for being a great source of knowledge, support, happiness, and meaning.

Finally, my profound thanks go, of course, to Ellie for being a seemingly unending wellspring of support, positivity, and good ideas through thick and thin. You have made the dark times bearable and the bright times brighter. And you’ve been crucial to my being able to finish this. I am happy to have shared tons of my favorite memories with you all over the world, and to continue to do so.

Abstract

This dissertation presents findings on the topic of autonomous multirotor drone landing — one basic area of multirotor drone flight that is not yet fully automated — with emphasis on real world proofs of concept. We conduct two phases of research, focusing first on landing sites that are structured with fiducial markers, and next on unstructured landing sites, where the drone cannot expect to detect existing infrastructure.

The first phase is a continuation of the author's master thesis which proposes an autonomous landing method based on fiducial markers and a gimbal mounted camera that is tested in simulation. We migrate the method from simulation to the real world and then expand on it. The initial migration revealed problems in recognizing the orientation of the fiducial markers in the real world, which was obscured by idealized graphics in simulation. We quantify this orientation ambiguity in several fiducial systems and carry out a real world landing experiment without mitigating the orientation ambiguity to gauge its effects. Finally, we develop a method for avoiding this issue entirely by directing the drone based solely on the angle from the drone to the landing pad. We demonstrate this method in the real world with both visual and infrared fiducial markers. The infrared markers can serve as landing site identification infrastructure both at daytime and nighttime, and can even be unpowered. Our overall contributions on this front are 1. some modifications to existing fiducial systems to mitigate orientation ambiguity and decrease runtime computational requirements, 2. a test of the effects of orientation ambiguity on the feasibility of autonomous landing when depending primarily on fiducial pose estimation, and 3. a minimalistic method for autonomous drone landing that uses fewer data points than existing methods (primarily the pixel position of the marker and orientation of the gimbal, and avoiding use of the altitude and range) to avoid the orientation ambiguity issue and allow flexibility in changing between modalities (visible and infrared) and changing the landing pad size without reconfiguration.

The second phase involves analyzing the terrain beneath the drone to determine if it is safe for landing. We focus on suburban environments as an easier test case, and on lava fields as a more challenging test case that is plentiful and relevant in Iceland. We develop a pipeline for creating appearance-based terrain classifiers to automatically locate safe landing sites with a typical, monocular camera using supervised learning methods for image segmentation. To avoid hand-labeling thousands of images, we generate a synthetic training data set from geometric surveys in analog environments — similar to the target landing environments, but not exactly the same. We train a U-Net to do this task, but the pipeline is modular such that it is possible to use other methods. We evaluate the method synthetically and validate it on real world data to ensure that it has not overfitted to the synthetic data. The method

shows success in classifying the training and validation sets in both scenarios, but can provide erroneous classifications when obstacles are too far away, or when obstacles cannot be easily identified by their appearance (e.g., safe, level gravel safe versus unsafe gravel slanted at 45 deg). We therefore supplement this method with a geometric check that uses a depth camera at low altitude (3.5 m – 5 m) which can prevent unsafe landing at erroneously classified sites. Our contributions on this front are: 1. a full pipeline to create visual terrain classifiers from the data collection stage to the real-world deployment stage, 2. guidelines and scripts for automatically creating the custom synthetic data sets (particularly useful for environments with few or no publicly available topographical data sets), 3. a successful terrain classifier that goes beyond previous methods by being relatively tiny (1-2 MB), operating at varying camera angles and which is applicable to any environment for which it has training data (not limited to, e.g., urban environments), and 4. a real-world demonstration of the method onboard a drone in real time.

Real world testing requires a lot of engineering overhead, which itself is useful for other researchers. Therefore, we describe our systems and payloads in detail so they can be reproduced by others. We carry out autonomous landing experiments using a DJI Spark and the DJI Mobile SDK (MSDK) for autonomous control, which gives insight into how to add custom computers into the control loop when using the MSDK. Finally, we develop 3 payloads for the DJI Matrice 350 that add onboard computers with varying degrees of integration and autonomous control, and we provide guidelines for reproducing them. We target the Raspberry Pi series as the main computing board for these payloads in the hopes that the wider community of drone users can take advantage of them in a number of different fields with relative ease, given the already booming Raspberry Pi community that ensures compatibility with a huge range of other hardware and software. Our contributions on this front are: 1. guidelines for creating a fully-integrated payload for the DJI Matrice series and other DJI drones, designs for the 3D-printed case for such payload with quick-release brackets for easy installation and removal in the field, and many field tests and demonstrations of the payload while testing the autonomous landing methods mentioned above.

If you fell down yesterday, stand up today.
– H. G. Wells

Contents

List of abbreviations

AGL – altitude above ground level
pp. 6, 76, 78, 79, 110

BEC – battery eliminator circuit
pp. 42

BLE – Bluetooth Low Energy
pp. 32

CPU – central processing unit
pp. 19

CSA – Canadian Space Agency
pp. 96

DEM – digital elevation model
pp. 7, 102, 103

DIY – do-it-yourself
pp. 27, 45, 68

DTM – digital terrain model
pp. 7, 102

DoF – degrees of freedom
pp. 6, 39, 45, 46, 76, 78, 79, 89, 157, 158

ENU – east, north, up
pp. 52

ESC – electronic speed controller
pp. 17

FOV – field of view
pp. 105, 125

FPV – first-person view
pp. 18, 61, 75

GCP – ground control point
pp. 163

GCS – ground control station
pp. 27, 31, 68

GDOP – geometric dilution of precision
pp. 44, 45, 100

GDR – gradual decoder refresh
pp. 75

GNSS – global navigation satellite system
pp. 6, 7, 16, 17, 26, 28, 29, 31, 36, 44, 45, 57, 58, 78, 100, 163

GPIO – general purpose input-output
pp. 32, 33, 112

GPS – Global Positioning System
pp. 77, 78, 79, 110

GPU – graphics processing unit
pp. 10, 33, 34, 143

HITL – hardware-in-the-loop
pp. 45

IMU – inertial measurement unit
pp. 17, 26, 28, 31, 66, 67, 103, 104, 145, 146, 158

IR – infrared
pp. xvi, xix, 5, 6, 7, 11, 18, 25, 28, 36, 75, 77, 78, 79, 81, 83, 84, 85, 86, 87, 89, 94, 97, 158

LiDAR – light detection and ranging
pp. xxi, 1, 6, 7, 8, 9, 11, 26, 28, 29, 31, 77, 78, 95, 98, 100, 103, 122, 123, 124, 125, 126, 128, 130, 144, 158, 162, 163, 164

MAV – micro aerial vehicle
pp. 27

MSDK – Mobile Software Development Kit
pp. 29, 56, 58

OS – operating system
pp. 42

OSDK – Onboard Software Development Kit
pp. 29

OTG – on-the-go
pp. 32, 33, 34, 42, 74, 112, 142, 143

PSDK – Payload Software Development Kit
pp. xxi, 29, 68, 70, 73, 74, 110, 112, 114, 118, 119, 121, 141, 143, 161

PWM – pulse-width modulation
pp. 17, 42

QR – quick-response
pp. 37, 38

RAVEN – Rover-Aerial Vehicle Exploration Network
pp. xvi, 96, 94, 95, 98, 100, 103

RC – radio controlled
pp. 41, 42

RGB – red, green, blue
pp. 9, 24, 25, 26, 27, 28, 91, 98, 101, 102, 104, 107, 110, 125, 127, 128, 140, 142, 141, 143, 162, 163

RGBD – RGB + depth
pp. 122, 123

ROS – Robot Operating System
pp. 13, 30, 31, 38, 42, 43, 48, 68, 143, 145

RTK – real time kinematic
pp. 28, 69, 78, 163

SDK – software development kit
pp. 10

SITL – software-in-the-loop
pp. 45

SLAM – simultaneous localization and mapping

pp. 7

SfM – structure from motion
pp. 7, 100, 102, 103

TPU – tensor processing unit
pp. xvii, 10, 33, 34, 109, 111, 116, 121, 130

UART – universal asynchronous receiver-transmitter
pp. 74, 112, 114, 116

UAS – uncrewed aerial system
pp. 15, 18, 103

UAV – uncrewed aerial vehicle
pp. 15, 16, 18

UGV – uncrewed ground vehicle
pp. 16

USV – uncrewed surface vehicle
pp. 16

UUV – uncrewed underwater vehicle
pp. 16

List of figures

List of videos

The following videos can help readers gain an intuition for some of the phenomena discussed. They are currently hosted on Vimeo, but in case they become unavailable in the future, readers can contact the author to retrieve them.

List of tables

Chapter 1:
Introduction

Drones are a rapidly growing industry at the intersection of automation, artificial intelligence, internet of things, and remote sensing. The European Commission anticipates a market value of €14.5 billion and 145.000 related jobs in Europe alone by 2030 [1]. In particular, drones are useful for removing human operators from situations where they would be too expensive or their safety would be at risk. Autonomous and semi-autonomous drones, which require only minimal training to use reliably, enable non-expert operators to perform many remote sensing tasks affordably. For example, drones enable quick and accurate land surveys via light detection and ranging (LiDAR) or photogrammetry. They aid in environmental monitoring, such as inspecting crop health or detecting pests in *precision agriculture* scenarios, and for detecting greenhouse gas emissions at industrial plants or waste management areas. They help operators see farther and more accurately for many tasks such as infrastructure inspection, archaeology, and search and rescue missions. They enable affordable aerial video for the film industry, for real estate, and for personal use.

Automation in this sector is critical. Although consumer drones could be considered *manually* piloted to some extent, much of the lower-level tasks are carried out by the drone as a black box, with the drone receiving only high-level commands from the operator, such as *go forward*, *turn left*, etc., thereby making it safe for non-expert operators to control the drone. In the worst case, the operator can simply let go of the control sticks, and a drone in *position* mode will simply stay still on its own. Monitoring or survey tasks are often extensions of this, where an operator outlines an area to be inspected on a map, and flight control software decomposes this into a grid flight with either continuous data collection, or data collection at specific locations. This type of automation makes repetitive or semi-autonomous tasks easily viable.

Drones are moving towards higher levels of autonomy in order to deal with dynamic or unknown environments, or for long-term deployments without direct human interaction. This comes with its own challenges in terms of maintaining the drone, interacting with ground infrastructure, and evaluating and reacting to the environment. There are also legal considerations regarding the deployment of fully autonomous drone systems, particularly if it means exposing society to the decisions made by a non-human, autonomous entity, or as far as respecting airspace restrictions and regulations. We research the technical side of this topic, with a human operator ready at all times to take control of the drone, but we anticipate further discussions of drone autonomy in the future as part of the larger discussion on autonomous vehicles.

Section 1.1: Problem description & motivation

Many aspects of drone flight have been automated, e.g., takeoff, position maintenance, point-to-point flight, simple obstacle avoidance, target tracking, various other extraneous tasks, and some aspects of landing. This automation is built on sensors that provide, e.g., a position or velocity estimate, images of the drone's surroundings, estimated distances to nearby surfaces, etc. Most consumer-grade drones automate landing by flying to a known-safe coordinate at a presumed-safe altitude, and then descending vertically until hitting the ground at a safe velocity. Some drones descend blindly, unaware of possible obstacles, and therefore depend on the given coordinate being truly safe. Other drones have simple obstacle avoidance sensors and software, allowing them to abort a landing if necessary. Most drones do not have a way of intelligently searching for a previously unknown landing site. The ability to land in locations previously unknown to the drone is necessary for enabling further autonomy in drones, such that they are not fully dependent on specific ground infrastructure or known landing locations without dynamic obstacles.

We consider the landing process in 5 constituent parts representing different problems, assuming the drone has completed its mission successfully and needs to land safely and quickly:

1. **search,** where the drone has to identify a target landing site,
2. **approach,** where the drone moves toward the target landing site horizontally,
3. **commit,** where the drone must decide whether or not to carry out the landing after getting a closer look,
4. **descent,** where the drone reduces altitude above the landing site, and
5. **touchdown,** where the drone makes contact with the landing site and disables the motors.

We focus primarily on the *search* and *commit* phases and give some attention to tracking the target during the *approach* phase, as these are the problems that generally require more computational development. This means we focus primarily on identifying potential landing sites and making a final decision on whether or not the landing site is viable. We do not focus on the low-level control tasks, since modern drones can respond to high-level commands equivalent to *move forward at 1 meter per second*, *spin clockwise at 15 degrees per second*, or *descend in place to the ground and disable the motors*. These are essentially the commands issued by a pilot when flying a drone via a controller. We also do not address the issues of, e.g., improving landing gear placement and traction because, although these are valuable avenues of research, they are better left to mechanical engineers.

The search and commit problems require real time computational effort, and we have decided to depend on *embedded* computation to tackle them. Therefore, we aim to make our methods efficient enough to run on lightweight hardware with low power requirements, build corresponding drone payloads, and deploy them on a drone in the real world as a proof of concept. While it is possible to offload these tasks to ground infrastructure which tends to be subject to less stringent power limitations, offloading the computation adds overhead in transmission and places heavy limitations on the operational range of the drone, e.g., imposing a maximum distance away from the ground station or cellular network infrastructure that the drone can move, and thereby influencing the form of potential missions. In our view, a method for searching for a previously unknown, viable landing site should be included as a basic function of the drone's flight controller and sensor packages, similar to some drones' current ability to abort a landing using onboard obstacle detection systems without delegating autonomy to a ground station over an often tenuous wireless data connection. The decision of whether or not to land at a particular location is obviously important to the drone itself because it can result in a crash landing from which the drone cannot take off again, but it has further implications in that it can damage the environment physically, i.e., with its propellers and its own mass, and chemically with its batteries that are usually made with lithium.

Identification and tracking of safe landing sites is key both to aiding human operators in semi-autonomous flight, and to enabling fully autonomous mission cycles, where the drone can operate truly without human intervention over long periods of time. This sort of mission structure has applications in plenty of fields, with the best known likely being package delivery. In order for a package to be successfully delivered by drone at quick notice, the drone has to locate a viable drop-off point with high accuracy, and often in urban areas, where there are many dynamic obstacles and degraded positioning accuracy as a result of urban canyons. Here, it is critical to identify and avoid hazards using real time, local information from the drone's surroundings, instead of depending on static maps or purely on geographic positioning. Further, such autonomous missions enable drones' remote sensing abilities to be further leveraged, such that they can be deployed long-term in uninhabited, non-urban environments for search and rescue, meteorological or geological data collection, or even drone-based Mars exploration. In these situations, the drone cannot expect to detect any particular kind of infrastructure to make the landing easier, so it must find its own way. Significant effort has been directed towards autonomous landing in academia, exploring different technologies for marking landing sites so the drone can easily recognize them in nature, and for analyzing areas the drone has never seen to determine whether they are suitable for landing. On the industrial side, some drones are being fitted with primitive solutions to accept or reject landing sites that they have not yet seen, or with sophisticated docks to guide them to a landing, enclose them for protection during storage, and deploy them again to execute missions sent by remote operators. There is room for improvement in terms of the types of algorithms used, in terms of optimization of the algorithms for embedded systems, and in terms of exposing the functionality to drone operators in semi-autonomous scenarios, which we explore in this research.

Finally, we consider two main paradigms, depending either on *structured* or *unstructured* landing sites. We define *structured* landing sites as those that include a marker or beacon that is known to the drone beforehand, such that it can use dedicated sensors or computer vision techniques to locate an artificial landing site that has been prepared by a human. We define *unstructured* landing sites as those that have no such infrastructure, and therefore the drone must use different methods for determining which areas are safe and unsafe. We explore the state of the art in both areas, expand on a few existing methods, and test our methods in the real world. At a high level, we test methods that depend on markers in the structured case, and test both geometric and appearance-based methods in the unstructured case.

Although we begin by developing and testing methods in simulation environments, we pair these tests with real world proofs of concept. In some cases, we demonstrate full autonomous landings on real drones, and in other cases, we demonstrate the landing site identification methods in real time onboard the drone, without conducting the actual landing itself. Real world proofs of concept are the most time-consuming part of this research, and it is tempting to explore solutions in the simulation environment only, because of how much time, money, and effort it can save. However, real world proofs of concept are necessary to eliminate potential biases, to reveal problems hidden by the simulation environment, and to give a holistic view of the requirements for embedding solutions onto the drone. We link to several videos (see the list of videos on page xix) which illustrate these real world tests and drone-based data collection events better than images can.

Section 1.2: Research summary

We need to approach the goal of autonomously landing multirotor drones with two different paradigms in mind – structured and unstructured landing sites – that need to be considered separately. Therefore, we explore the following *general* research areas:

- **G1:** how can a drone autonomously land at a structured landing site?
- **G2:** how can a drone autonomously land at an unstructured landing site?

Further, we distinguish between testing in simulation and testing in the real world, as there is a major difference between them, and there is significant effort required to migrate a solution from simulation to the real world. We proceed with two major design principles in mind:

1. We want to demonstrate our methods in the real world, as this provides the ultimate sanity check as to whether they work or not.
2. We want our methods to run onboard the drone itself, such that the system does not suffer from the range limitations and latency issues inherent in offloading the computation.

Regarding **G1**, there are a number of academic projects exploring autonomous landing at structured landing sites. In some cases they approach the scenario as a control problem, focusing primarily on generating coherent, reliable control signals to direct a drone to land on either a stationary or moving target. In other cases, they focus on the sensors and infrastructure required to identify the landing pad, i.e., infrared beacons (emitters), lasers, or markers – visual or IR patterns [2], [3]. We constrain our research efforts to scenarios where the landing site is denoted by a marker, as this sort of passive infrastructure does not necessarily require any type of power and can often be recognized via the most common drone peripheral sensor – a monocular camera – meaning that it could be integrated into most drone platforms without requiring additional sensors. Most of the preceding work in this field uses a downward-facing camera, or a camera fixed rigidly to the drone to identify the landing pad and direct the drone towards it using the landing pad's pixel position in the camera frame [4], [5], [6]. Although this does work in many cases, the camera's limited field of view means that the drone cannot detect the landing pad unless it is nearly directly above it. This has resulted in detection loss during landing attempts in non-ideal conditions. For example, the drone can simply lose sight of the landing pad if it (or the moving land pad) drifts too far to one side. Similarly, if the camera is fixed rigidly to the drone, then a sudden wind gust can cause the drone to pitch or roll significantly, thereby erroneously pointing the camera away from the landing target. To circumvent this limitation, some projects propose using multiple cameras (at fixed orientations) such that, e.g., a forward-facing camera enables the drone to identify the landing pad from farther away, and a downward-facing camera enables it to continue identifying the landing pad during descent [7], [8]. It is also possible to mitigate issues with losing track of the landing pad or pose estimation errors by using multiple markers and, e.g., a voting scheme [9], [10], [11].

Constraining **G1** even further, we arrive at our first research question:

> **RQ1:** how can a drone use an actuated camera to actively track a marker for autonomous landing?

One approach is to use the 6 degrees of freedom (DoF) pose of the marker as perceived from the camera images. Unfortunately, this cannot be determined unambiguously, meaning that the control signals generated during such a scenario are often erroneous. One project gets around this issue by creating a proprietary fiducial marker system [12] that uses specially designed patterns to make the pose unambiguous, resulting in successful autonomous landings with closed source software [13]. Although other approaches often estimate the whole 6 DoF marker pose, they typically avoid using the orientation (the position is reliable), and usually require some other data source (either a sensor or global navigation satellite system (GNSS)) to estimate the altitude above ground level (AGL) or height above the landing pad [14], [15], [16]. One method uses a sophisticated, gimbal-mounted sensor with LiDAR and cameras in the IR and visible spectra to identify an actively heated IR fiducial marker [17].

Our contributions toward **RQ1** are as follows: We conduct an evaluation of 5 open source fiducial marker systems to determine how unstable their 6 DoF poses are and if they are able to run in real time on embedded hardware. We find that they all exhibit this pose ambiguity at different rates, but are all able to run on our embedded hardware. We go on to use these marker systems in a real world autonomous landing scenario despite their erroneous control signals. We use a small drone in a safe and controlled environment to gauge the extent to which the pose ambiguity and erroneous control signals affect the landings. We find that pose ambiguity itself does not ultimately *preclude* successful autonomous landings, but it does decrease the accuracy. Finally, we develop a method for circumventing the pose ambiguity problem altogether by directing the drone's approach to the landing pad based on the tilt and pitch angle from the drone to the landing pad as calculated by the gimbal's orientation and the position of the landing pad in the camera frame (both of which can be known unambiguously). This is essentially an abstraction from the methods proposed in the related work that use only the landing pad's pixel position in a downward-facing camera. It does not require the height above the landing pad, AGL, or GNSS as other methods do, but instead uses the combination of marker pixel size and zoom factor as a heuristic for distance, which reduces data requirements. We conduct a real world test to show how this method can generalize to multiple cameras with ease, such that our drone system can leverage its multi-camera payload for close-range landing with a wide-angle lens, long-range landing with a zoom camera, and nighttime landing with an IR camera. We test landing targets in both the visual and IR spectra, and demonstrate a completely passive IR landing pad that can allow for nighttime landing with no powered infrastructure.

G2 describes a harder problem than **G1** overall, as more responsibility is placed on the landing site evaluation algorithm since it cannot depend on recognizing known infrastructure. Determining which landing sites are safe and unsafe in otherwise unstructured terrain is more difficult not only conceptually, but also in terms of implementation, computational requirements, and data requirements. There are three overarching paradigms here:

1. picking a safe landing site from an existing model of the environment,
2. picking a safe landing site from a local terrain model generated during flight, and
3. examining utterly new locations.

Generally, picking safe landing sites from known models of the environment can be done efficiently from, e.g., a digital elevation model (DEM) or digital terrain model (DTM). After selection, sites can be approached using visual tracking or GNSS alone, for a lightweight, emergency landing site detection method [18] [19]. Such methods can also incorporate heat maps of cell phone activity [20] and other data types to attempt to avoid humans.

In many cases, a database of known obstacles is either unavailable or out of date, and an adaptation of the approaches above is to generate a local terrain model simultaneously with other mission tasks and then incorporate similar techniques as before to execute the landing. One approach is to use the takeoff location as the target landing location, record the visual appearance of it on takeoff using a camera, and match its visual appearance during landing to position the drone more accurately than a GNSS can [21]. This method is relatively efficient compared to many others but limits mission structure so that the drone can only autonomously land at its takeoff location. Another method depends on simultaneous localization and mapping (SLAM), i.e., collecting geometric information about the environment using cameras or other sensors to determine which areas are suitable for landing [22]. This can be done with a single monocular camera using a structure from motion (SfM) approach, which generates a 3D model of the terrain from many different monocular views collected systematically [23], [24], [25]. Several studies use local LiDAR surveys to geometrically evaluate a scene for landing, and the geometric evaluation is typically based on slope and roughness. One approach is to conduct a survey of a relatively wide area in the vicinity of a desired landing site and choose the best potential landing site by fitting planes in the resulting point cloud which correspond to the locations of landing skids, which was tested on a full scale helicopter with an onboard computer [26]. A similar approach is to continually scan and keep track of potential landing sites during the course of an otherwise normal mission. Similar methods are tested in the real world [27] and in simulation [28]. One method uses stereo depth cameras to generate an elevation map for geometrically evaluating a scene and a monocular camera to determine whether a surface is rigid or not [29]. In the stereo case, the method rejects the landing site if any part of it is unsafe. However, the stereo camera fails to reject particularly difficult sites such as tall grass or water, which motivate the use of a monocular camera to test for rigidity. One hybrid approach uses a stereo depth camera attached to a small helicopter to create a 3D map of the environment and searches it for safe landing sites [30]. Then, the monocular camera is used to track an identified safe landing site over time to direct the helicopter to a landing. However, the stereo depth camera can miss small obstacles such as power lines. Terrain reconstruction with LiDAR and stereo depth cameras requires transformations to remove biases resulting from the orientation of the sensor, which is not usually level throughout the duration of a flight.

One sophisticated approach that incorporates vision-based systems uses several detectors: one for identifying potential landing sites from known geographical models, a visual detector for common obstacles like vegetation or trees, a visual detector for people, a map manager to place detected obstacles in the geographical model, and a path planner to decide where to go [31]. The vision components are neural networks that were pre-trained or trained on existing data sets with some manual refinement, and the system was tested in AirSim.

Using a preference for minimal assumptions, we prefer not to depend on the existence of a database of obstacles, or pre-existing terrain models that can go out of date. Further, LiDAR is not typically available on most drones, and therefore the requirement for a LiDAR sensor is somewhat heavy. Additionally, methods that require pre-landing surveys significantly influence mission structure by requiring extra time or continuous computing effort throughout the mission.

We therefore constrain **G2** to arrive at our second research question:

> **RQ2:** how can a drone efficiently and in real time land in an unstructured area that it has not previously seen?

Some methods take on the difficult task of evaluating completely unknown landing sites by their appearance, instead of their geometry. A simple method is to use histogram-based thresholding as a makeshift way of detecting obstacles in a target region, in addition to comparing the appearance of the target region over several frames to detect motion, which has been used with success to to determine the safety of package delivery locations in an urban environment [32]. One method uses a ResNet backbone to classify an entire scene based on its probability to contain each of 7 classes representing physical things with an associate safety rating, e.g., *safety* contains the label “lawn,” *general* contains the labels “wilderness” and “wheat field,” and *dangerous* contains “crowded place,” “vehicle intensive place,” “road,” and “water area” [33]. The scenes are classified with success, but the labels are not localized in the scene, so the classifications and the labels of the associated dataset are coarse, i.e., they cannot give a pixel location of a target landing site, and the method is not deployed on a drone. One method uses an appearance-based classifier to simultaneously classify the potential landing areas in an urban environment and estimate the 3D depth of the camera scene, expecting the camera to be at a particular orientation facing forward and down [34]. It is trained on synthetic data from Google Maps and tested in a lab environment, but could theoretically be deployed on a drone.

Although we do not contribute in the context of sophisticated landing gear, such work is still interesting and useful in this context. Drones tend to have simple landing gear – usually skids or point landing gear that are generally not compliant to the surfaces they eventually meet. Most of the related work therefore focuses on identifying areas in the terrain that are exceptionally advantageous to accommodate the simple landing gear. However, for a more holistic approach, it is justified to create new landing gear that adapt to the rough surfaces that tend to be common in some of the more exotic areas that we would like to land. A common approach to this end is to create actuated point landing gear that conform to the surface, which have been used in multiple cases to allow drones to land on uneven surfaces [35], [36], [37].

Our contributions toward **RQ2** are as follows. We develop a process for creating *appearance-based* terrain classifiers that use images to identify safe landing sites in unstructured terrain that we test the method in the real world (see **RQ3**). We choose this paradigm to re-use the most common peripheral sensor, the monocular camera, eliminating extra sensor requirements at runtime. To generate datasets to train the classifiers, we conduct geometric terrain surveys in analog environments that are *similar enough* to the environments where the drone will eventually need to land, but not the same. We test both LiDAR surveys (faster, but require more expensive sensors) and photogrammetry surveys (slower and less accurate, but more affordable) from suburban areas around the Capital Area in Iceland for this data generation. We then train a visual classifier on this dataset offline and deploy it directly on the drone after training. This method therefore requires only red, green, blue (RGB) cameras, but can be augmented if the resources are available. We test the method using a U-Net neural network architecture with success, although the process itself is agnostic with respect to any particular visual classifier architecture. The appearance-based method occasionally misclassifies landing sites, so we develop a method for geometrically evaluating landing sites before committing to a landing, using a downward-facing stereo depth camera, which is relatively lightweight (requiring two monocular cameras), as compared to, e.g., LiDAR. We test the methods together on our drone at several sites in solidified lava flows, using the appearance-based method for longer-distance landing site evaluation, and using the geometric method for final evaluation at low altitude. The methods show success in evaluating the landing sites. Time limitations prevented us from fully autonomously executing these unstructured landings, but we have developed the required components. To conduct fully autonomous landings in the lava flows requires a slow, cautious process of integrating the components and gradually testing them in the field, but it does not require development of major new functionality.

Lastly, as a result of our design principles of demonstrating our methods in the real world and running them onboard the drone, we have our third research question:

> **RQ3:** how can our structured and unstructured landing methods run onboard a drone in real time?

This research is directed primarily by the selection of drone platforms and computational hardware available to us, which is determined by financial considerations and availability. We develop two custom-built hobbyist hexacopters with embedded deep learning computational hardware to test our early methods. However, we found that these had limitations in their positioning accuracy, battery life, and weatherproofing. As a result, we moved on to a DJI Spark indoors, creating a makeshift computational system that used embedded hardware while actually offloading the processing to a ground control station because of the inability to attach payloads to the Spark itself. For the majority of the projects in this research, we had the opportunity to use a DJI Matrice 350. As a result of the previous discontinuation of the Manifold 2 – the proprietary computational payload that was available for the Matrice series in the past – we needed to develop our own computational payloads for testing. We chose to use the Raspberry Pi series as our embedded computational platform for affordability and compatibility with a wide variety of software, as well as a vibrant community of users and developers. We present 3 computational payloads for use with the Matrice 350, with varying degrees of integration and computational capabilities. Since the primary overhead in using these payloads is simply decoding the high-resolution video streams in real time, we explore possibilities for using hardware decoding, with some success. Further, we test a tensor processing unit (TPU) (the Google Coral) and graphics processing unit (GPU) (the Jetson Nano) for accelerated embedded processing in order to run deep learning inference and to carry out geometric calculations in real time. These are relatively weaker boards compared to those commonly used in the related work for **RQ2**, which are typically the Jetson TX2, Jetson Xavier, Orin NX, and similar boards. We create applications within the DJI software development kits (SDKs) to not only run our solution onboard the Matrice 350, but also to stream the desktop of the onboard computer to the drone's controller such that we can see the processed video in real time and interact with the computer using the controller's touchscreen. We could not find a similar payload in our research. Finally, we create 3D-printed cases for the payloads that are cost-effective, easy to make, install in the field with quick-release mechanisms (requiring no screws), and fit in the Matrice's carrying case for safe transportation and integration with typical drone deployment workflows. We provide designs for the cases and installation instructions for the software, in the hopes that others can more affordably create fully integrated payloads for their DJI drones, which can often appear as closed black boxes with no hope of customizability. Ultimately, we create a payload that runs our solutions for both **RQ1** and **RQ2** simultaneously onboard the drone.

Section 1.3: Publications

The following publications comprise the main content of this research. We describe our experiments to quantify orientation ambiguity in fiducial marker systems [38]. Then, we outline an autonomous landing experiment in the presence of such orientation ambiguity [39]. We describe the PhD's trajectory in a workshop paper and suggest the idea of autonomous landing on solidified lava flows after attending fieldwork in the Icelandic highlands [40], and then we write a journal extension of the workshop paper [41]. We test an autonomous landing method using both IR and visual fiducial markers while avoiding orientation ambiguity, which required the development of a preliminary computational payload for our DJI Matrice 350 [42]. We detail an upgraded and fully integrated payload for our Matrice 350 and provide designs and guidelines on how to reproduce it [43]. We go on to create an appearance-based landing site evaluation method that uses LiDAR data to generate synthetic visual data sets on which we train a lightweight, image segmentation classifier [44].

[38] J. Springer and M. Kyas, "Evaluation of Orientation Ambiguity and Detection Rate in April Tag and WhyCode," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, Naples, Italy: IEEE Computer Society, Dec. 2022, pp. 281–286. doi: 10.1109/IRC55401.2022.00054.

[39] J. Springer and M. Kyas, "Autonomous Drone Landing with Fiducial Markers and a Gimbal-Mounted Camera for Active Tracking," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, Naples, Italy: IEEE Computer Society, Dec. 2022, pp. 243–247. doi: 10.1109/IRC55401.2022.00047.

[40] J. Springer, "Autonomous Multirotor Landing on Landing Pads and Lava Flows," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, 2022, pp. 425–428. doi: 10.1109/IRC55401.2022.00081.

[41] J. Springer and M. Kyas, "Autonomous Drone Landing: Marked Landing Pads and Solidified Lava Flows," *International Journal of Semantic Computing*, vol. 18, no. 2, pp. 283–299, 2024, doi: 10.1142/S1793351X24300061.

[43] J. Springer, G. Þ. Guðmundsson, and M. Kyas, "Lowering Barriers to Entry for Fully-Integrated Custom Payloads on a DJI Matrice," in *2024 International Conference on Content-Based Multimedia Indexing (CBMI)*, 2024, pp. 1–5. doi: 10.1109/CBMI62980.2024.10859236.

[44] J. Springer, G. Þ. Guðmundsson, and M. Kyas, "Toward Appearance-Based Autonomous Landing Site Identification for Multirotor Drones in Unstructured Environments," in *MultiMedia Modeling*, I. Ide, I. Kompatsiaris, C. Xu, K. Yanai, W.-T. Chu, N. Nitta, M. Riegler, and T. Yamasaki, Eds., Singapore: Springer Nature Singapore, 2025, pp. 198–211.

The following pre-print is similarly a major part of this research, but has not yet been accepted for publication:

[42] J. Springer, G. Þ. Guðmundsson, and M. Kyas, "A Precision Drone Landing System using Visual and IR Fiducial Markers and a Multi-Payload Camera," 2024. [Online]. Available: https://arxiv.org/abs/2403.03806

The following publications are tangentially related to this research but are not main contributions:

[45] B. B. Carr *et al.*, "Evaluating the Use of Unoccupied Aircraft Systems (UASs) for Planetary Exploration in Mars Analog Terrain," *The Planetary Science Journal*, vol. 5, no. 10, p. 231, Oct. 2024, doi: 10.3847/PSJ/ad781e.

[46] M. Kyas, J. D. Springer, J. T. Pedersen, and V. Chkoniya, "Data Analysis in the Shipping Industry: eShip Case Study – Problem Statement," in Handbook of Research on Applied Data Science and Artificial Intelligence in Business and Industry., Hershey, PA, USA: IGI Global, 2021, pp. 381–400. doi: 10.4018/978-1-7998-6985-6.ch018.

The publications and their corresponding research questions are summarized in Table 1.

Publication	Research Question
[38], [39], [42]	1,3
[40], [41]	1,2,3
[43]	3
[44]	2,3

Table 1: Correspondence between publications and research questions.

Section 1.4: Document overview

The rest of this document is structured as follows:

Chapter 2 discusses many of the background topics upon which the research is built. We discuss drone types, terminology, software, sensors, relevant embedded computational boards, and underlying technologies such as simulation environments and Robot Operating System (ROS).

Each section in Chapters 3 and 4 describes a particular research project. If there is a corresponding publication, we include the content of that publication (mostly unmodified, but adapted for this document) and cite the publication. In the Conclusion of each section, we briefly summarize its research implications.

Chapter 3 discusses our research in the *structured* landing site paradigm, starting from the author's master thesis and motivating our insistence on demonstrating methods on real world data and hardware. It then includes a background section on fiducial markers. We evaluate several fiducial marker systems in terms of their suitability for autonomous landing and continue with an experiment that uses those systems naively for autonomous landing. We finish with a full demonstration of an autonomous, marker-based landing method on a sophisticated drone with an onboard payload.

Chapter 4 describes our research in the *unstructured* landing site paradigm. Naturally, we test in the environments available to us in Iceland, including suburban and rural environments as well as solidified lava flows, owing to the recent eruptions on the Reykjanes peninsula and the widespread use of drones in geology. Further, the monochromatic, rugged, fresh lava represents a particularly challenging terrain in which to land drones, so it is a difficult problem. We discuss our geometric and appearance-based methods of landing site evaluation. We provide instructions for how to make our affordable, modular, computational payload for the DJI Matrice series, which is currently the flagship enterprise drone series from DJI, the world leader in commercial drones.

Finally, in Chapter 5 we discuss the implications of each of our projects in terms of the research questions, and we discuss some potential areas for future exploration.

Chapter 2:

Background

Section 2.1: Drone types & terminology

The term *drone* is not well defined in common parlance but generally refers to any type of robotic vehicle that does not have an onboard operator, e.g., aircraft, ground vehicles, marine surface vehicles, and underwater vehicles. In this research, we use the term *drone* to refer to uncrewed multirotor vehicles (multicopters), such as the well-known *quadcopter*, but we provide a short overview to describe each type.

Aerial drones are typically referred to as uncrewed aerial vehicles (UAVs) and the entire system including the vehicle and its typical, ground-based control infrastructure are referred to as uncrewed aerial systems (UASes). *Fixed-wing* drones, i.e., airplanes, depend primarily on airspeed to create lift over their wings and control their orientation using control surfaces on the wings and other parts of their fuselages. They are useful in long-range missions as a result of their relative power efficiency, but they require infrastructure such as runways for takeoff and landing. *Rotary-wing* drones, i.e., helicopters or multirotor drones, create lift by spinning their wings or turbines independently of their fuselage, so they can create lift even while the drone itself remains still. A helicopter vectors its thrust by changing the pitch of the blades of its main rotor depending its position. The large rotor creates a torque on the fuselage of the helicopter and requires an additional tail rotor to cancel it out. In this arrangement, the main rotor controls the helicopter's pitch and roll, and the tail rotor controls the yaw. Multirotor drones, on the other hand, vector thrust by changing the speed of their motors, which each have fixed-pitch propellers. The motors come in counter-rotating pairs so that the torque from one can cancel the torque of another, thereby allowing the drone to control its yaw without an equivalent to a tail rotor. There are many types of hybrid aerial drones that take advantage both of fixed-wing efficiency and of multirotor vertical takeoff and landing. Rotary-wing drones are typically less suited for long-range missions than fixed-wing drones because they are usually less power efficient, but their ability to hover and maneuver vertically gives them greater flexibility. They also require much less infrastructure because they can take off and land vertically, making them particularly suitable for missions in challenging environments, e.g., requiring landing and taking off on a boat or in a forest. Missions for aerial drones are often defined in terms of waypoints to which the drone navigates using primarily a global navigation satellite system (GNSS). Aerial drones in autonomous or semi-autonomous missions often fly above the altitude of obstacles, such that obstacle avoidance is not a primary consideration when creating a mission.

Land-based drones are typically referred to as uncrewed ground vehicles (UGVs). These usually move using wheels, tracks, or legs, i.e., they tend to be rovers, tanks, or crawlers. Although their missions are often outlined using waypoints, UGVs tend to put more effort into obstacle avoidance and have to consider the *traversability* of the terrain on which they are traveling, since their environment is often much more cluttered than the open spaces that aerial drones enjoy. They have fewer restrictions on weight and power than aerial drones because they can conserve power by simply stopping.

Marine surface drones, also known as uncrewed surface vehicles (USVs) or boats, have similar structures to UGVs and can sometimes even use the same control software. They tend to operate in open spaces, can depend on GNSS, and often need less obstacle avoidance capabilities than UGVs. They require more waterproofing than UAVs and UGVs to survive in the marine environment. Submarine drones, i.e., uncrewed underwater vehicles (UUVs), naturally have a very strong emphasis on waterproofing. They can only use radio-based navigation when at the surface of the water, as water attenuates radio signals very quickly. Thus, for remote control and data transmission, they typically depend on physical tethers, which limits their range. In order to know their position during autonomous operation, they often collect GNSS data at the surface and then calculate how far they have moved after diving by integrating their velocity over time using a technique known as *dead reckoning*. They must surface regularly, however, as a position calculated with dead reckoning will become less and less accurate over time.

Section 2.2: Basic multirotor hardware

Multirotor drones require a set of basic hardware regardless of the exact type, and we will cover the most typical case, which is that of an electric drone with brushless motors. There are many variations on this paradigm, including different types of motors, fuel sources, and layouts which are out of scope for this discussion. First is the body, or fuselage, which is usually made of carbon fiber or plastic for strength and lightness. The drone rests on its underside or on landing gear in the form of skids

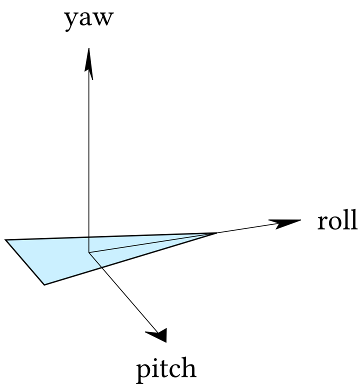


Figure 1: Basic angles describing the orientation of an aircraft. If the blue arrow is an aircraft, then pitch, roll, and yaw describe its orientation in the corresponding axes.

or point legs. In the center of the body is a compartment for the control electronics, power distribution board, and a battery, which is most commonly based on a lithium polymer composition for its energy density. Lithium polymer batteries require mild operating temperatures and can become unusable if too cold. Therefore, some batteries have internal heaters for operating in cold environments. Extending from the body is one arm for each motor or pair of motors, in the case of, e.g., octocopters with motors on top and bottom. Each motor has a corresponding electronic speed controller (ESC) to provide power based on control signals — typically pulse-width modulation (PWM) signals — from the flight controller.

The flight controller's job is to control the position and orientation of the drone based on a high-level control policy or flight mode. For example, if the drone is in a *position hold* mode, it will try to keep the drone in its current position in spite of external factors, e.g., wind, and internal factors, e.g., hardware variations in the motors or speed controllers. The flight controller manages the position of the drone *indirectly*, based on the drone's orientation, i.e., its *pitch*, *roll*, and *yaw* (see Figure 1). It has an inertial measurement unit (IMU) to determine the drone's orientation in space, based on estimation of the gravity vector from an accelerometer, angular velocity from a gyroscope, and heading from a compass. It fuses and filters these data to create a higher quality estimate of the drone's orientation. There are several methods for determining the drone's position in space, but the most common is GNSS. If the flight controller detects a difference between its target position and its actual position, it changes the speed of the motors using the ESCs, resulting in a change of orientation, and eventually resulting in a change of position. This is the primary mechanism for positional control for hovering and reaching waypoints. Since the orientation of a drone is constantly changing, it is advantageous to attach sensors to the drone via a gimbal, which can control its orientation independently of that of the drone. There are two main functions of the gimbal: first, it stabilizes the sensor mounted to it, such that the sensor is not subject to as much vibrational noise or extraneous orientation changes; second, it aims the camera independently of the drone, such that the drone can simply hover in place while the camera inspects its environment from all angles.

A critical part of a UAS is the ground control station, which issues commands to the UAV. In many cases, this is a human operator with a controller to give the drone high-level commands in the form of *velocity targets*, e.g., *move forward at 1 m/s*, by moving control sticks. More sophisticated systems have two-way communication between the drone and the controller, such that the controller sends commands and receives data from the drone, e.g., telemetry data such as altitude, velocity, remaining battery capacity, etc., or first-person view (FPV) video. Although these may be sent via dedicated transmission infrastructure on simpler drones, they have generally been merged into a single transmission link on more sophisticated drones.

Section 2.3: Drone sensors & applications

In many fields, drones are used as remote sensing platforms, and the end user primarily cares about the data the drone will collect — not the drone itself. Drones are cheap remote sensing platforms compared to human-based data collection. Often, they can fit in a backpack and be deployed at a moment's notice to collect data above treetops, over water or lava, in volcanic plumes, on cliffsides, inside caves or in radiation areas. They remove human operators from danger while still providing accurate manual and automated data collection capabilities.

Section 2.3.1: Monocular cameras

Arguably the most common of drone peripheral sensors is the monocular camera, and a common use case for such cameras is collecting visible-light aerial photos or video for cinematography, advertising, personal memories, etc. This is also the primary sensor for image analysis algorithms that the drone can use to react to its environment.

Less common, but still widespread is the IR camera, which is useful for remote IR thermometry. Taking into account reflectivity (the tendency of an object to reflect IR radiation regardless of its own temperature) and emissivity (the tendency of an object to emit IR radiation because of its own temperature) of a surface, as well as the intensity of IR radiation detected by the camera's IR sensor, one can estimate the temperature of the surface [47]. This has many applications, e.g., locating animals in outdoor scenes, detecting leaks in pipelines, detecting geothermal activity [48], [49] and fires, even estimating crop water status in precision agriculture [50], etc. IR cameras provide a night vision capability, but can be used in the daytime as well, e.g., as in Figure 48.

Hyperspectral cameras provide insight into the chemical makeup of the material they are viewing. By providing intensities over a large range of wavelengths at each pixel, as well as a calibration from a neutral surface of known material, one can calculate the absorption rates of an object and thereby correlate it to known absorption rates for various materials. This gives an idea of the object's chemical makeup [51].

Section 2.3.1.1: Codecs

Camera sensors themselves produce maps where each pixel of the field of view has been assigned a color (or some other intensity, based on the modality of the camera). This map, or a minimally processed version, is what is produced when shooting images in `.raw` format. Although `.raw` images preserve the authentic representation of the original scene, they are space-inefficient as a result of containing redundant data, e.g., the full dynamic range of brightness in each of the channels, and being uncompressed or minimally compressed. This is not a prohibitive issue for storing images to, e.g., a memory card on a camera because storing many MB of data per second is not a problem in that context. On the other hand, streaming a video in `.raw` format from a camera to a computer over a network would likely monopolize the entire network bandwidth and then some, meaning that the video would be processed with latency and preclude the possibility of using the network infrastructure for other tasks.

It is so inefficient to transport video in uncompressed format that some of the most common tasks in video processing are to encode the video to a more efficient format before transmission, usually using a lossy compression algorithm, and then to decode it to an approximation of its original form before further processing. Components to encode and decode video for this purpose are called *codecs*, and two of the most common video compression standards are H264 [52] and its non-backwards-compatible successor H265 [53]. Performing compression and decompression with video codecs is not a trivial task, and it can occupy a significant portion of processor time if done in software. On many embedded components that work with video, e.g., smartphones, tablets, and many embedded computational boards, this work is done on specialized, dedicated hardware components that are very efficient and free the central processing unit (CPU) of this kind of overhead. However, once such functionality is implemented in hardware, it becomes unflinchingly rigid, particularly with regards to its hard limits on video resolution. A system that is decoding or encoding video outside the specifications of the hardware must fall back to software for a more flexible implementation at the cost of processor time, which can be significant. This can often happen when working with proprietary systems that do not follow the default specifications.

Although standards exist to unify the community in its approach to certain tasks such as video encoding and decoding, not everyone follows these standards to the same extent, and everyone has their own reasons. The most modern computational boards might follow the absolute latest video compression standards by, e.g., getting rid of an H264 hardware decoder in favor of an H265 hardware decoder, thereby allowing it to optimize its performance over that of H264 and interact with other modern systems. On the other hand, components that exist as a part of a larger ecosystem with its own idiosyncrasies — particularly in terms of hardware components that are difficult to change once deployed in products — may be developed with backwards compatibility in mind as a higher priority than using the latest video codec. Therefore, they might prefer to stick with H264 decoding hardware instead of upgrading. This can cause issues when using the most modern hardware to interact with specialized proprietary systems, as one may lose hardware decoding capabilities entirely because of mismatched features.

Section 2.3.1.2: Monocular camera models

Computer vision tasks often require using camera images to make decisions about the real world based on spatial analysis of it, which requires methods for determining real world coordinates from camera coordinates. Camera models relate the 2D coordinates of the camera frame to the 3D coordinates of the physical world. The *pinhole camera* model, illustrated in Figure 2, is based on the idea that all rays of light pass through an infinitesimally small hole before meeting the camera sensor. The distance from the hole to the camera sensor is the *focal length*, f, and the image that is projected on the sensor is upside-down. The actual camera sensor is *behind* the hole, such that it is at position $-f$ in our setup. To simplify the arithmetic, we analyze the image seen by a virtual sensor that is upright and *in front* of the hole at a distance f.

Let u, v be the horizontal and vertical pixel coordinates of a point in the camera frame, which is implicitly at a distance f in front of the camera sensor. The coordinate (u, v) essentially describes an angle offset from the z axis of the camera. We want to calculate the 3D positions of objects $(x, y, z)^T$ in the real world, where the x axis extends to the camera's right, the y axis extends to the camera's top, and the z axis extends in front of the camera. There are infinitely many solutions for the real 3D coordinates, for a given value of (u, v), depending on the actual scale of the object in the image. However, if we know one of x, y, or z, then we can derive the two others according to Equations 1 and 2.

$$u = f\frac{x}{z} \tag{1}$$

$$v = f\frac{y}{z} \tag{2}$$

In most cases, the camera frame is not square, so we determine scaling factors s_x and s_y to create virtual focal lengths in the x and y dimensions to represent the

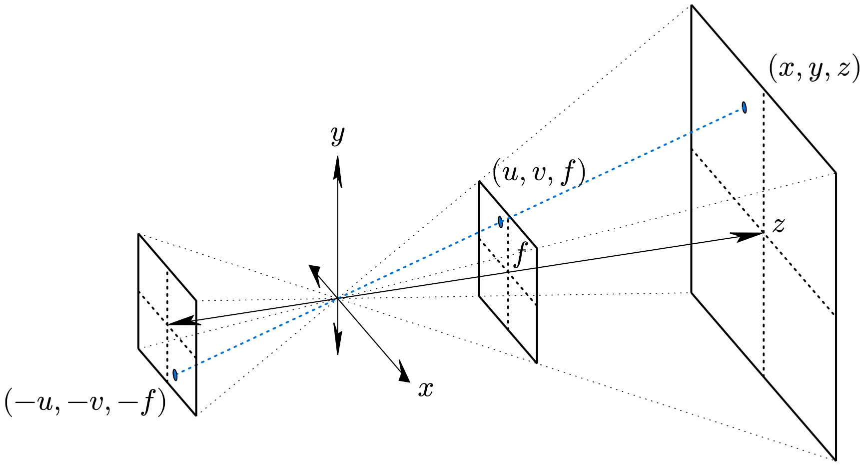


Figure 2: Pinhole camera model. The actual scene observed by the camera is described in (x, y, z) coordinates. All the light from the scene passes through a hole at the origin. The camera's sensor is positioned at a distance f (the focal length) behind the origin with respect to the scene, and it is inverted and mirrored. For simplicity, we instead work with a virtual sensor positioned at a distance f in front of the origin and which is oriented in the same way as the scene. We use this model to associate image coordinates (u, v) with real world coordinates (x, y, z).

different resolutions. We calculate separate focal lengths $f_x = f s_x$ and $f_y = f s_y$. Putting all this together, we arrive at Equation 3 which describes the ideal pinhole camera model:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3}$$

Next, since it is typically the case in computer vision that $u, v \geq 0$ and their origin is at the top left of the camera frame, extending to the bottom right, we add the *principal points* u_c and v_c to denote the center of the camera frame in each dimension, as in Equation 4.

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & u_c \\ 0 & f_y & v_c \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{4}$$

Further, because of manufacturing imperfections, the sensor can be skewed such that the u and v axes are not perpendicular, but rather offset from perpendicularity by an angle θ. Then, we add a *skew factor*, $\alpha = f_x \tan(\theta)$, arriving at the final pinhole model as described in Equation 5:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & \alpha & u_c \\ 0 & f_y & v_c \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{5}$$

Here, the matrix including the skew factor, the focal lengths and principal points for each dimension, as shown in Equation 6, is called the *camera intrinsics matrix*.

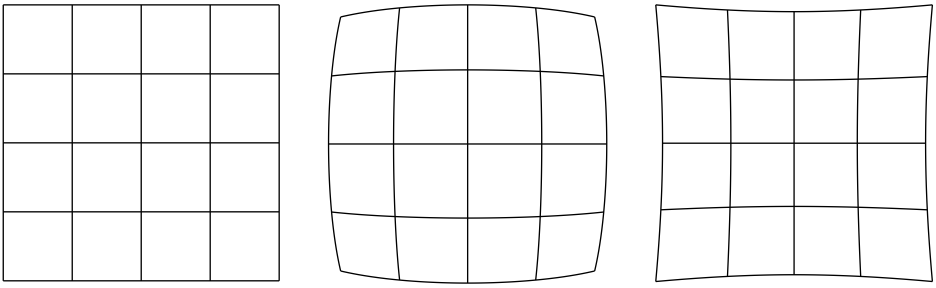
$$K = \begin{pmatrix} f_x & \alpha & u_c \\ 0 & f_y & v_c \\ 0 & 0 & 1 \end{pmatrix} \tag{6}$$

In order to direct all of the light rays through the hole of our pinhole camera, we of course need a lens, and, in the real world, lenses can introduce distortion to the image that the camera receives. In order to properly analyze the images, we therefore need to pin down the type and extent of the lens distortion. For example, Figure 3 illustrates radial distortion. Figure 3a shows a canvas that exhibits no distortion. Figure 3b shows a canvas that exhibits *barrel* distortion, where magnification decreases as a function of the distance from the center of the image. Figure 3c shows a canvas that exhibits *pincushion* distortion, where magnification increases as a function of the distance from the center of the image. Radial distortion can be described by the simple vector in Equation 7 in our case:

$$D = (k_1, k_2, p_1, p_2, k_3) \tag{7}$$

Here, k_1 and k_2 describe the radial distortion, and k_3 describes tangential distortion. Further information on the distortion coefficients can be found in [54], [55].

It is possible to calibrate a camera, i.e., calculate the camera intrinsic parameters and distortion coefficients, by using images of objects of known shapes and sizes, and, as shown in Figure 4, the classic method is to use an arrangement of black and white squares [54], [56]. OpenCV provides an implementation of this calibration procedure that uses a “calibration chessboard” [57]. Given such a chessboard of 11x8 squares, as shown in Figure 4a, the calibration algorithm detects the 10x7 vertices between them, as shown in Figure 4b. Using prior knowledge of the fact that the chessboard is planar, and that the side length of each square is 15.9 mm, we can determine the camera intrinsics matrix and the distortion parameters. We collect several pictures, covering the full field of view of the camera, and deflecting



(a) Canvas with no distortion. (b) Barrel distortion. (c) Pincushion distortion.

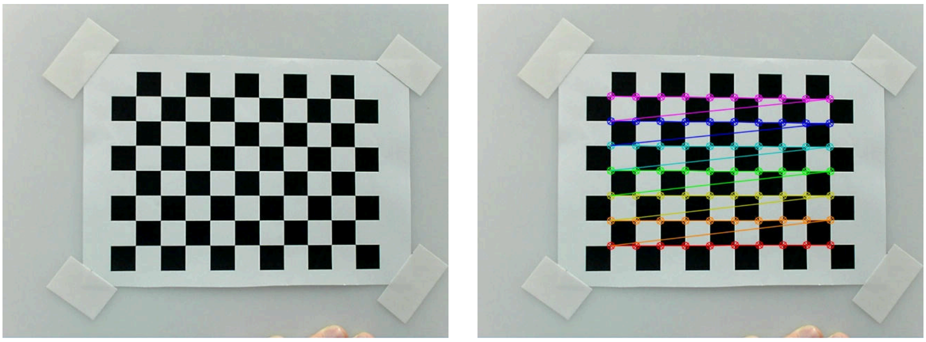
Figure 3: Types of radial camera distortion.

the chessboard at various angles. As an example, we calibrated a webcam with a resolution of 640x480 pixels using 289 images. The matrix K from Equation 6 takes on the value shown in Equation 8 and the distortion parameters as in Equation 9.

$$K = \begin{pmatrix} 544.106717 & 0.000000 & 342.772181 \\ 0.000000 & 545.207925 & 243.291791 \\ 0.000000 & 0.000000 & 1.000000 \end{pmatrix} \tag{8}$$

$$D = \begin{pmatrix} -0.045895 & -0.157692 & 0.005568 & 0.008672 & 0.364941 \end{pmatrix} \tag{9}$$

We see from Equation 8 that the skew (α) is 0, so the horizontal and vertical axes of the camera are perpendicular. The principal points $(u_c, v_c) \approx (342, 243)$ are indeed approximating the exact middle of the 640x480 camera frame. The focal lengths $f_x \approx 544$ and $f_y \approx 545$ are reasonably close to one another, implying that the pixels in the camera frame are essentially square.



(a) Camera calibration chessboard.

(b) Vertices detected.

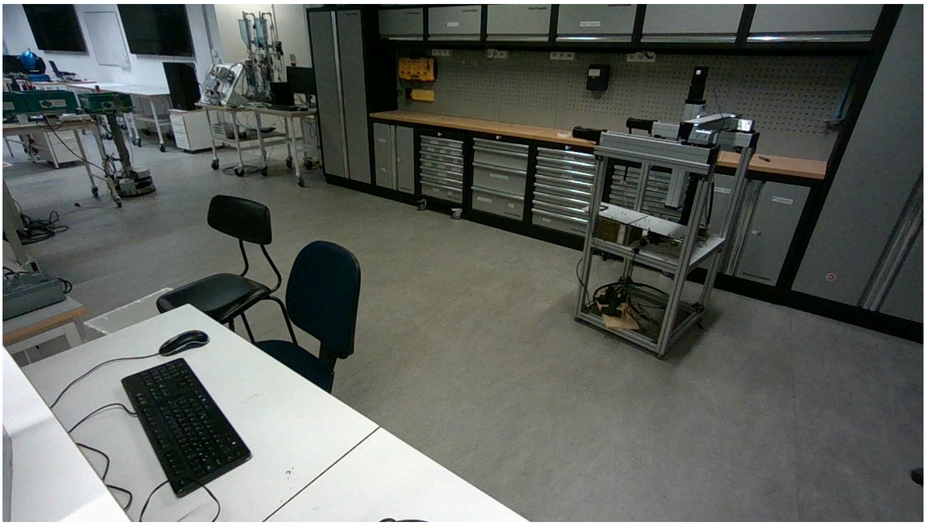
(c) Camera calibration chessboard near camera frame edges.

(d) Camera calibration chessboard at high deflection.

Figure 4: Calibration procedure in OpenCV. OpenCV can automatically determine the intrinsic and distortion parameters of a camera from a set of pictures of the chessboard with a known number of vertices and squares of known side length (Figure 4a). In Figure 4b the detector has found the corners of the individual chessboard squares (the vertices) and highlights them to show a detection has been made. For a more accurate calibration, it helps to have detections near the edges of the camera frame, as in Figure 4c, and to have high angles of deflection, as in Figure 4d.

Section 2.3.2: Stereo cameras

An arrangement of two side-by-side monocular cameras can provide deeper spatial insight into a given scene than that of a single monocular camera, through comparison of the differences in their simultaneous images and the knowledge of the



(a) RGB image.

(b) Left infrared image.

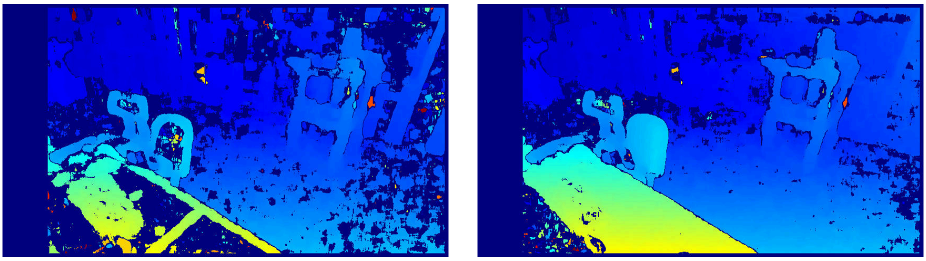
(c) Right infrared image.

(d) Left infrared image (emitter enabled).

(e) Right infrared image (emitter enabled).

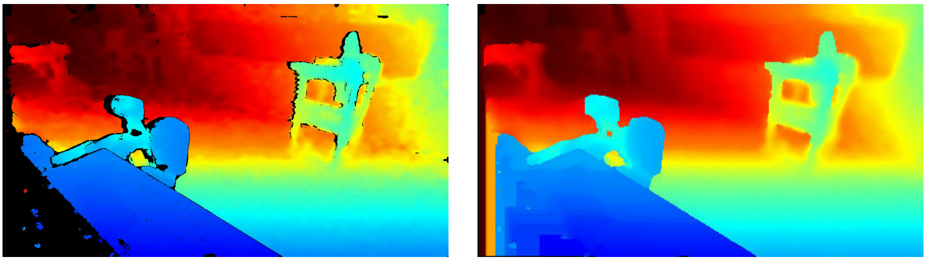
Figure 5: Raw images from a stereo depth camera. The RGB image (5a) gives an overview of the scene and allows the 3D representation to be colorized. The left (5b) and right (5c) infrared images come from cameras that are offset at a precise horizontal translation, which allows the system to derive a 3D representation of the scene. In some cases, the scene may not be adequately clear in the infrared images for a 3D reconstruction. In those cases, it can help to enable an infrared emitter to create high-contrast points in the scene; these are visible as point sources on the surface of the desk in the bottom pictures (5d and (5e).

relative displacement from one camera to the other, as inspired by human eyes. Figure 5 shows the raw images from an Intel RealSense D455 stereo depth camera, which contains three camera sensors: a central, RGB camera (Figure 5a) and two IR cameras — one positioned to the left (Figure 5b) and one to the right (Figure 5c). The RGB camera provides a conventional, monocular view of the scene, and the infrared cameras provide slightly different views of the scene as a result of their translation from each other. The differences in the pixel positions of objects from the left frame to the right — referred to as *disparity* — help to determine the objects' 3D positions relative to the camera, as shown in Figure 6. Figure 6a visualizes the result of a disparity calculation from Figures 5b and 5c. The matcher finds high disparity at the outlines of most of the objects in the scene. However, highly self-similar surfaces, e.g., the monochrome white desk surface and the gray floor, have undefined segments where the matching algorithm was unable to find common points between the left and right images. This camera has an IR laser emitter positioned between the two IR cameras that shines point sources into the scene, as visible in Figure 5d and Figure 5e, which can help to calculate disparity scenes that are particularly dark or self-similar. With the emitter enabled, the matcher is able to better calculate the disparity even on the self-similar surfaces, as shown in Figure 6b. Note that in the disparity maps, the resolution of the populated canvas is smaller than that of the left



(a) Disparity map from Figures 5b and 5c (emitter disabled).

(b) Disparity map from Figures 5d and 5e (emitter enabled).



(c) Depth map (unfiltered) derived from Figure 6b.

(d) Depth map (filtered) derived from Figure 6b.

Figure 6: Disparity and depth maps corresponding to Figure 5. The raw disparity map, shown in Figure 6a does indeed perceive many of the 3D features of the scene from Figures 5b and 5c, but is replete with holes, particularly in places where there is not enough contrast to associate points between the left and right IR cameras. With the IR emitter enabled, as in Figures 5d and 5e, many of these holes are filled, as shown in Figure 6b. Depth maps use the disparity to determine the actual distance from the camera to each point, as shown in Figure 6c, where cooler colors are closer and warmer colors are farther away. Using some filtering and hole-filling methods, the camera can create a cleaner depth map as in Figure 6d.

and right images. This is because of the way the disparity is calculated in a moving window or kernel, which is undefined at the edge of the screen and matched to one side.

Finally, if we know the translation and rotation from one camera to the other, we can correlate displacement to distance, thereby deriving a 3D depth map of the scene, as shown in Figure 6c, where the color at each pixel denotes the z axis distance from the camera, i.e., how far the surface at a particular pixel is sticking out straight in front of the camera. Some of the holes from the problematic areas in the disparity map from Figure 6b propagate to the depth map — especially around particularly sharp edges, spatial discontinuities, and the desk surface in the bottom left of the scene, which is almost entirely outside of the right camera's field of view. As shown in Figure 6d, we can apply some filters to the depth map to make it more usable by other entities that need to analyze the scene. Some useful filters to apply here are Gaussian smoothing, temporal filters, and hole-filling filters. The Gaussian smoothing helps to get rid of erroneous spikes under the assumption that the scene is a continuous surface. Temporal filters help to reduce noise in the depth map by aggregating depth maps from different instants over time. This is important because plenty of factors, including slight mismatches in the timestamps of the left and right images, high frequency blinking lights, or other transient phenomena can make the depth map appear erroneously wavy. Finally, various hole-filling filters can help to make an educated guess on the depth by, e.g., aligning the depth map to the RGB image, matching problematic locations by color to surfaces that have been successfully identified, and extending those surfaces to fill the holes.

Section 2.3.3: LiDAR

A common tool for surveying with drones is light detection and ranging (LiDAR), which detects distance using the time-of-flight of a laser pulse. When working in a single dimension, i.e., detecting distance from one point to one other point, this type of sensor is typically called a laser rangefinder. 2D LiDARs expand on the 1D case by stacking several laser rangefinders in a line, or by rotating a single laser rangefinder quickly. Rotating 2D LiDARs are useful for, e.g., obstacle avoidance systems on mobile robots operating in a plane. In the 3D case, a LiDAR sensor is typically implemented as a set of laser rangefinders arranged in a line and swept orthogonally across the surface being modeled, like a scanner. It may also be spinning, as in Video 3. LiDAR sensors are often used for terrain sensing and modeling, as they can quickly scan large areas with centimer-level accuracy. Combined with a GNSS system and IMU, the terrain can also be reconstructed, allowing for realistic modeling and measurement.

LiDAR modules produce geometric data in the form of *point clouds*, which are sparse representations of the occupied areas and their positions as detected by the module's laser. With a color sensor such as an RGB camera, it is possible to colorize the points to have a more complete representation of the scene. In order to use the point cloud in subsequent tasks, it is sometimes necessary to transform it into a continuous surface, e.g., by conducting a Poisson reconstruction to create a corresponding mesh [58].

Section 2.4: Open source flight control

Open source flight control software and hardware is often used to control custom drones. There are many open source flight control software suites [59], and we will go over the most relevant ones for this research. MAVLink is an open source communication protocol used by many flight control software suites [60]. It provides a message definition to allow micro aerial vehicles (MAVs) to communicate with ground control stations (GCSes) and with each other. Users can also add their own, custom message definitions. MAVLink implementations are available in many programming languages, and there are extensions for message signing, error correction, and message forwarding via proxy. ArduPilot is a widely used, MAVLink-enabled software suite with variants for different vehicle types, e.g., rovers, boats, submarines, copters, planes, and many hybrids [61]. PX4 is a similar autopilot software to ArduPilot, developed by ETH Zurich with a more permissive license structure [62].

ArduPilot and PX4 can use similar hardware, e.g., the Navio2 [63] or Cube Orange and many other boards. Flight control is a safety critical real time task, so both ArduPilot and PX4 can run in NuttX [64] real time, embedded OS. ArduPilot can additionally run in ChibiOS [65] and on Linux.

Section 2.5: Enterprise solutions

Sometimes, a do-it-yourself (DIY) solution will not provide adequate performance, and users will opt for an enterprise solution with a professional development team behind it. Enterprise drones, although *much* more expensive than in-house projects, tend to provide better stability in terms of hardware, software, and flight performance. For example, closed systems allow developers to tune the flight controller for a particular set of hardware and adapt it to particular ground control software, controlling for edge cases. They can more fully explore the limits of their hardware and add in important features like waterproofing or in-built battery heaters. The advantage of using an enterprise solution is that a developer or researcher can view the drone itself as a finished system requiring no development and can therefore focus on research which is typically an *extension* of the drone's basic functionality. In other words, there is significantly less overhead effort put into the drone platform itself. Here, we discuss enterprise solutions that are directly relevant to this research.

Section 2.5.1: DJI Matrice 350, H20T, and L2

The DJI Matrice 350 is currently DJI's flagship quadcopter, with a nominal 55-minute maximum flight time, 9.2 kg maximum takeoff mass (of which 2.7 kg can be payload), IP55 rating (dust and water protection) [66]. It has the ability to connect to a real time kinematic (RTK) base station for increased precision in its GNSS positioning. It comes with a large touchscreen controller, which also has waterproofing for outdoor deployment. Although the drone is rather large, its arms can fold in on its body to save room, after which it can pack into a secure carrying case for transportation to field sites. There are infrared distance sensors and stereo cameras on the 6 sides of the fuselage, which is in the form of a rectangular prism. It has multiple payload mount points, both for DJI proprietary payloads and for custom payloads that use DJI's software development kits (to be discussed next). It has no onboard computer that users can extend for their own purposes. The Matrice 350 has multiple points for attaching payloads, each with a specialized port for power and data. Proprietary payloads for the Matrice 350 typically attach to a payload port and require DJI's proprietary SkyPort or X-Port connectors, which are essentially breakout connectors that provide vibration dampening and a gimbal mount. The Matrice 350 also provides a topside E-port that is designed for onboard computers. These ports allow for different functionality, as the payload ports are designed for components that need to collect data on their own and may need to forward that data to the controller, while the E-port is designed for components to take data from the drone, control the drone, or control other payloads. (See the Matrice 350 manual [66] for more information about these ports.) A critical difference between the ports (at the time of this research) is that payloads are not able to stream video to the controller via the E-port, and payloads are not able to retrieve video from the drone or control other payloads via the payload ports. What this means is that it is not possible, using a single port, to retrieve video or other high-bandwidth data from the drone, process it, and forward it to the controller.

DJI produces enterprise payloads for their drones, and the relevant ones for this research are the H20T [67] and the L2 [68]. The H20T is a multi-camera payload, with one wide-angle camera for basic video, one zoom camera for long-range video, and one IR camera. It also has a laser rangefinder for measuring distances. The H20T is particularly useful for photogrammetry in the RGB and IR spectra, and for IR thermometry. Further, it has an automatic tracking feature to keep the camera centered on a particular location without human input even while the drone is moving, which is helpful when surveying a particular location accurately. The L2 is a LiDAR module with an RGB camera and a high-precision IMU.

Section 2.5.2: MSDK

DJI provides a Mobile Software Development Kit (MSDK) to allow developers to interact with their drones via tablet applications [69]. This fits within their controller-and-tablet paradigm, where the controller provides the connection to the drone, and the tablet provides a more sophisticated graphical interface from which to view video and other information from the drone, as well as manage it. Major use cases for the MSDK include applications for sophisticated surveying or automation of data collection that require no changes to the drone itself or its payloads, and that are not negatively affected by potential latency or temporary loss of connectivity.

Section 2.5.3: PSDK

DJI provides a Payload Software Development Kit (PSDK) for allowing developers to create new payloads that interact with the drone [70]. Payload-based applications are set up as `cmake` projects, where the user writes high-level code on top of open source sample applications and pre-compiled libraries that DJI distributes [70]. The samples outline the following functionalities:

- camera emulation (for streaming data to the controller)
- camera management (i.e., zooming, taking pictures or videos, managing camera-based storage, retrieving files, etc.),
- data transmission
- flight controller interaction (subscription to data topics such as GNSS position, orientation, and flight control including waypoint-to-waypoint flight, takeoff, and landing)
- liveview (viewing a video feed *on the payload*)
- gimbal control
- perception (access to the video streams from the onboard stereo cameras for obstacle avoidance)
- widgets (for viewing text on the controller screen or for adding buttons)
- and many others.

The pre-compiled libraries contain lower-level code for interacting closely with the drone itself, and they are not available to users. The PSDK was preceded by the Onboard Software Development Kit (OSDK) [71] and as such there is large overlap and a lot of compatibility, but DJI has migrated the OSDK functionality to the PSDK and recommends to switch [72]. Major use cases for the PSDK include creating additional means of data collection, automating real time tasks that require low latency and more consistent connectivity, and giving the drone autonomy within a closed system that does not require a ground station. Examples of custom PSDK products include third-party camera and LiDAR payloads, parachutes, water samplers, liquid sprayers, gas detectors, and cellular network connectors for direct streaming.

Section 2.6: Robot Operating System (ROS)

ROS is a set of open source middleware, algorithms, and developer tools to create reliable robotic systems [73], [74]. ROS applications are configured as a set of nodes, each executing a modular computing task and communicating synchronously with other nodes using a publisher-subscriber model in a many-to-many setup. Messages are of predefined types, e.g., one node may publish a *pose* message, which is composed of a *point* specifying the position and a *quaternion* specifying the orientation. Both the point and the quaternion are also message types, and they are populated by primitives such as `float64`. Further, ROS has *services* for relatively small, simple, consistent calculations using asynchronous message communication. The newer iteration, ROS 2, introduces *actions* for longer, slower behaviors that correspond to real-world actions, such as autonomous navigation. It also offers an API to help developers enforce real time computing constraints. A key benefit of ROS's architecture and tools is the ability to quickly set up reliable network communication between physical computational boards with minimal configuration, which is particularly useful when creating heterogeneous embedded system prototypes. ROS 2 extends this to add configurable reliability attributes to the messages, allowing developers to set various levels of necessity and persistence in ensuring a particular message arrives at its destination. ROS has historically been used in innumerable robotic applications such as autonomous drones, rovers, submarines, warehouse robots, and more [74]. It also has integration with open source and proprietary simulators, to help with prototyping systems in simulation and then quickly transitioning them to the real world.

Section 2.7: Simulation environments

Simulators are useful tools for developing robotic systems and algorithms, as they reduce financial, logistical, and environmental constraints during the development process. For example, if a drone crashes in a simulator because of an error in its control algorithm, one can simply reset the simulator, instead of picking up the pieces of the drone, spending the time and money to reconstruct it, and then testing again in the field. For this reason, many robotic systems are pre-prototyped in simulation, and many simulators have native ROS integration to facilitate this. Further, simulators offer a fast way to create synthetic data sets that mimic the real world. We discuss Gazebo and Isaac Sim as the relevant simulators to this research.

Section 2.7.1: Gazebo

Gazebo is a common, open source simulator for developing robotic systems that interact with the world around them [75]. It uses a physics engine to provide accurate reproductions of the effects of actuators and contact between objects. It has many plugins for common use cases, e.g., for ROS integration, and for sensors such as IMUs, contact sensors, altimeters, magnetometers, GNSS, monocular cameras, stereo cameras, and LiDARs. Many such plugins allow users to configure basic attributes of those systems, e.g., operational range, resolution, framerate, etc., as well as noise, and camera intrinsic and distortion parameters, to provide a more realistic depiction of what data such sensors might collect when deployed on a physical system. Users can download openly available models, modify them, or create their own, specifying appearance and dimensions in 3D object files, and describing their physical attributes for the Gazebo engine. Importantly, dynamic models such as the Iris quadcopter are openly available and connect with ArduPilot, PX4 (see Section 2.4), and other relevant software in this field such as GCS software, MAVLink-enabled libraries, and Python libraries such as DroneKit [76]. Screenshots from Gazebo are shown in Figures 10 and 11.

Section 2.7.2: Isaac Sim

Isaac Sim is a modern, realistic simulator developed as part of the NVIDIA Omniverse [77]. It has many of the same features as Gazebo, e.g., realistic physics and visual simulation, ability to add custom models and sensors, integration with ROS, and dynamic models that are programmatically controllable and can interact with software outside of the simulation. Isaac Sim is particularly useful for training AI models. Its synthetic camera sensor provides the ability to create not only simulated visible images, depth maps, LiDAR point clouds, etc., but also sophisticated segmentation masks that automatically label each pixel in the image according to the object to which it belongs, and can provide bounding boxes for each object. Using this method, it is possible to create image classifiers, evaluate their performance in the simulator, and then transition them to the real world.

Section 2.8: Embedded computational hardware

A main theme in this research is how to run autonomous landing solutions in an embedded computing environment onboard the drone. Therefore, we experiment with a few of the most popular embedded computing boards on the market, with a focus on overall computing capability, small form factor, minimal power requirements, and compatibility with a large array of software and hardware. We are limited by supply chain, finances, and time, so we constrain our experiments to the select boards as outlined in this section.

Section 2.8.1: Raspberry Pi

The Raspberry Pi boards (see Figure 7) are a very popular family of single-board computers with a large community of users and developers. They are cheap enough for hobbyists but powerful enough for prototypes or even products, depending on the particular case. The fact that they are so widely used means that the community has already done much of the work in making them compatible with many different types of software and hardware. For example, there are many “hats” — external hardware packages that attach to the general purpose input-output (GPIO) pins — and extend the functionality of the Raspberry Pi, e.g., stepper motor drivers, high-definition audio cards, meteorological and gas sensor packages, neural and tensor processing units, and inertial measurement sensor packages (e.g., Navio2 [63]), to name a few. Raspberry Pis provide a cheap, low-power, accessible, embedded computing solution in many fields, e.g., automation, smart agriculture, monitoring, education, robotics, remote sensing, and more [78]. We cover 3 of the latest models of the standard Raspberry Pi board: the Raspberry Pi 3B+, (Figure 7a), Raspberry Pi 4B (Figure 7b), and Raspberry Pi 5 (Figure 7c). These can all run the Raspberry Pi OS, which is a Debian Linux system developed specifically for the Raspberry Pi.

The Raspberry Pi 3B+ has a 1.4 GHz, 64-bit, 4-core Cortex-A53 CPU with a 1 GB of RAM, 2.4 GHz and 5 GHz WiFi capabilities, Bluetooth 4.2 and Bluetooth Low Energy (BLE), Ethernet, HDMI (full size), 4 USB 2.0 ports, a CSI camera port, a DSI port for touchscreens, stereo audio output and a composite video port, and a micro SD card port. It uses a micro SD card both for storage and booting, and it uses a 10 W (5 V, 2 A) power supply. Its main power port is a micro USB port, but it can also be powered over the GPIO pins. It has hardware H264 decoders (see Section 2.3.1.1) and encoders at up to 1080p resolution and 30 FPS [79].

The Raspberry Pi 4B has a 1.5 GHz, 64-bit, 4-core Cortex A72 CPU with 1, 2, 4, or 8 GB of RAM. It improves on the Raspberry Pi 3 in upgrading to Bluetooth 5.0, gaining two USB 3.0 ports (of 4 total), an extra HDMI port (micro), and a faster H264 hardware decoder (60 FPS). Further, it adds H265 hardware decoding at 4K resolution and 60 FPS, for a more modern standard. It is more power-hungry, requiring 15 W (5 V, 3 A). Its main power port, a USB C port, is also a USB on-the-go (OTG)

(a) Raspberry Pi 3B+.

(b) Raspberry Pi 4B.

(c) Raspberry Pi 5.

Figure 7: The Raspberry Pi boards.

port, such that it can function as a USB device that creates a network connection to a host computer for easy access [80].

The Raspberry Pi 5 has a 2.4 GHz, 64-bit, 4-core Cortex-A76 CPU with 2, 4, 8, or 16 GB of RAM. It improves on the Raspberry Pi 4 by adding a VideoCore VLL GPU, support for dual 4K display output, a PCIe interface for fast data transmission, two MIPI transceivers for cameras or displays, and a real time clock. It uses 25 W (5 V, 5 A). It retains the H265 hardware decoder but not the H264 hardware decoder [81].

Over time, the Raspberry Pi family has become increasingly capable, with faster CPUs and higher RAM capacities allowing them to run increasingly complex algorithms. The main data storage and boot drive has evolved from only an SD card, to either an SD card or USB device, to either an SD card, USB device, or PCIe device. USB OTG ports are not only convenient, but tend to simplify component connections in embedded prototypes, resulting in higher reliability. Unfortunately, the higher power requirements and lack of H.264 hardware decoding can make the Raspberry Pi 5 less usable in embedded scenarios. However, it is ultimately still a cornerstone of embedded computing in the academic, hobbyist, and commercial sectors.

Section 2.8.2: Google Coral

The Google Coral family includes a number of computational boards that come in multiple form factors, including a single-board computer mimicking the Raspberry Pi, a USB accelerator that works as a peripheral device, and several PCIe modules. We cover the single-board computer and USB accelerator as they are relevant to this research. Figure 8 shows some computing boards that are more capable than the Raspberry Pi series for embedded machine learning.

The Google Coral Dev Board [82] (see Figure 8a) is a single-board computer with a Cortex-A53 64-bit, 4-core CPU, with 1 or 4 GB of RAM, 8 or 16 GB of onboard eMMC storage, 2.4 and 5 GHz WiFi, Ethernet, Bluetooth 4.2, one type-A USB 3.0 host port, one type-C USB 3.0 OTG port, stereo audio output, HDMI (full size), and a 40-pin GPIO header. Importantly, it contains an Edge TPU for efficient embedded

(a) Google Coral Dev Board. (b) Google Coral USB Accelerator. (c) Jetson Nano Development Kit.

Figure 8: Edge computing boards for embedded AI. The Google Coral Dev board and USB accelerator provide a TPU and the Jetson Nano provides a GPU.

machine learning inference with TensorFlow Lite [83], e.g., object detection, pose estimation, image segmentation, and key phrase detection. It runs a Debian-based distribution of Linux called Mendel. The USB accelerator [84] is a simply a USB case and interface for the Edge TPU, to be used by a host running its own operating system.

The Dev Board is designed for prototyping with embedded machine learning, but its smaller community and specific operating system reduce compatibility and add overhead when interacting with different hardware or using software that is outside of common use cases. The USB accelerator (see Figure 8b) allows users to leverage the ease of development, larger community, and compatibility of the Raspberry Pi with the embedded inference speed of the Edge TPU. In the Google Coral paradigm, users can train their models in the TensorFlow [85] framework on a desktop computer or similar, then transform the model to a TensorFlow Lite model using the Edge TPU compiler, and then deploy it to their Edge TPU. Transforming the model to a TensorFlow Lite model involves some loss of functionality through quantization and simplification of some operations, but users can typically achieve comparable performance to the original model, with the ability to run it relatively quickly in embedded scenarios.

Section 2.8.3: Jetson Nano

The Jetson Series is NVIDIA's family of computational boards for embedded artificial intelligence and machine learning. We use the Jetson Nano Developer Kit (see Figure 8c), which includes a version of the Jetson Nano board mounted to a carrier board providing peripherals similar to those of the Raspberry Pi. The Jetson Nano developer kit has a 64-bit, 1.4 GHz Cortex-A57 MPCore CPU with 4 GB of RAM. The carrier board comes with a slot for an SD card as its main storage and boot drive, Ethernet port, a display port, HDMI port (full size), a micro USB OTG port, 4 type-A USB 3.0 ports [86]. It has a 128-core GPU with CUDA cores. The carrier board itself is larger than the other embedded computational boards mentioned. It can run Ubuntu Linux and NVIDIA provides pre-configured OS images for users.

Chapter 3:

Structured landing sites

Section 3.1: Introduction

Structured landing sites use some sort of infrastructure to distinguish themselves from the surrounding environment, such as a helipad with a large H symbol on it, or an airplane runway with a long row of bright lights leading a pilot in the correct direction at night. The underlying assumption is that, if the pilot or system detects a location using some predefined method, that location is safe for landing. Recall our 5 stages of landing from Section 1.1: search, approach, commit, descent, and touchdown. The basic advantage of structuring the landing site is that the *search* phase is simple and, often, computationally cheap: simply scan the environment for the known and obvious characteristics of landing sites. The landing site can be adapted such that it depends on common, basic drone sensors, and such that the drone can accurately estimate its positional offset from the landing site in physical units, instead of, e.g., just identifying the landing site's pixel coordinates in an image from the drone's camera. Such landing sites often provide some sort of identifier as well, making it easy to distinguish between multiple known landing sites. In some cases, especially in the case of landing pads on ships or other moving vehicles, the landing site may even actively transmit its location to an aircraft, so the aircraft can approach it as if it were a normal waypoint, bypassing the need for extra sensors. Shifting some of the effort of the landing task to the landing site itself reduces the effort required by the aircraft, which often has comparatively more stringent power and computational limitations.

Structuring the landing site is, however, a double-edged sword. First, it restricts the number of landing locations to those with the necessary infrastructure, implying that all of the missions depending on a given landing method must end at a set of predefined locations. Second, the landing locations must be prepared, which implies overhead in construction and maintenance costs. Further, in many cases the landing sites must either be powered or depend on some sort of natural energy source such as sunlight, implying even more overhead to power the infrastructure or implying that the landing site might not be operable all the time. These are important disadvantages to consider, but there are many scenarios where structured landing sites are a perfectly reasonable approach, such as package delivery to a known location, or landing on a ship or other vehicle. Therefore, we explore them in this chapter.

In the context of autonomous drones, landing locations tend to be fitted with patterns or beacons that make it straightforward for the drone to recognize them in real time from a distance with a low risk of false-positive detections, e.g., IR beacons or visual markers. Then, with a (typically downward-facing) monocular camera, the drone can easily identify the beacon or marker based on its known characteristics, without having to make sophisticated inferences about an otherwise unfamiliar environment. However, more complex systems may actively transmit their positions to the drone as mentioned earlier. Landing drones at structured landing sites often allows drones to achieve a higher positional accuracy upon landing than simply using GNSS, which is subject to drift, interference from nearby physical formations, and dilution of precision resulting from not being able to receive signals from enough satellites on a cloudy day. Further, dynamic obstacles, e.g., humans or animals who mistakenly wander onto a landing site, would obscure the landing site from the drone in many cases leveraging IR or visual methods, providing a sort of inherent obstacle avoidance “feature” that GNSS alone would lack.

One important distinction that separates different types of landing infrastructure is whether it is *active* or *passive*. We refer to *active* infrastructure as emitting a signal of some sort, i.e., methods depending on beacons or data transmission, as mentioned earlier. *Passive* infrastructure is similarly that which inherently conforms to some easy method of identification, such as visual markers. Active infrastructure is often effective in more cases, since it does more work to enable the landing, but it typically requires more power, maintenance, and sophisticated mechanisms than passive infrastructure. For this reason, we prefer to depend on passive infrastructure. We focus primarily on *fiducial marker* systems, as they are a flexible and widely used method for robotic localization, with many possibilities for modification. We use fiducial markers to direct the drone throughout the search, approach, descent, and commit phases of landing. We focus on the challenge of using an actuated, gimbal-mounted camera to make the search and approach phases more flexible, which makes estimation of the landing pad’s pose more difficult because of the ambiguity in recognizing the marker’s orientation.

Section 3.1.1: Chapter overview

This chapter proceeds as follows:

- Section 3.2 discusses *fiducial markers*, which are the method we explore for marking our structured landing pads.
- Section 3.3 describes our first attempt at implementing a real world version of a method for autonomous landing with fiducial markers and a gimbal-mounted camera that was developed in simulation. It describes the advantages and disadvantages of using simulation for development of such methods.
- Section 3.4 discusses experiments regarding the quantification of a limitation of fiducial systems that we refer to as *orientation ambiguity*. It also discusses modifications to those fiducial systems carried out in an effort to both mitigate orientation ambiguity and reduce runtime requirements for the embedded computational boards needed in this research.
- Section 3.5 explores the feasibility of autonomous landing using fiducial markers in spite of orientation ambiguity. It outlines our first successful testing platform, discusses its autonomous landing performance using the same systems as in Section 3.4, and compares the results.
- Section 3.6 describes our first, custom computational payload for testing on a DJI Matrice 350 in Section 3.7. We expand on this payload in Section 4.3.
- Section 3.7 proposes a method to avoid orientation ambiguity altogether, by depending primarily on the pixel position of the marker and the orientation of the camera, and it also leverages camera zoom for long-range detection. It describes two platforms built for testing the method in the real world, and discusses the performance of that method. It also shows the benefits of using fiducial markers in the infrared spectrum, including the potential for both active and passive landing infrastructure that allow for precision landing at nighttime.

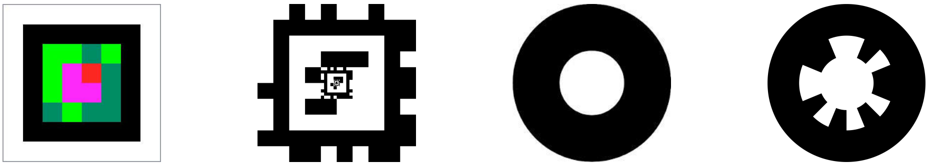
Section 3.2: Background: fiducial markers

Fiducial markers are commonly used in computer vision for marking objects such that they can be recognized and localized easily, from a computational point of view. Figure 9 shows several fiducial markers. A common use case for such markers is to affix them to objects or locations of interest to a robotic system, to allow the system to easily extract information about its location or interact with the object. Using each marker system's dedicated software, a computer can identify the markers in typical, monocular images from RGB cameras, which are a relatively cheap and common sensor. The markers are (typically high-contrast) 2D patterns that often take on a square or circular shape, and they usually encode an ID. Their unnatural appearance helps to reduce false-positive detections in nature. Since they typically encode an ID, the objects to which they are affixed can be easily distinguished. The best known example of a fiducial marker is probably the quick-response (QR) code (shown in Figure 9a), which is designed to have a relatively large data payload compared to other fiducial markers, and is commonly used to store web links for quick scanning via a mobile phone camera. The high payload capacity means that QR codes have a dense encoding compared to other fiducial markers, and therefore that it is harder to recognize them in non-ideal conditions, e.g., from far away (implying a small pixel size), at a high angle of deflection, or with motion blur.

The true strength in these markers is the ability to extract 3D information from them via 2D images. In many fiducial systems, if a marker's side length or diameter is known, as well as the distortion parameters of the camera viewing it, one can derive the marker's position and orientation in 3D space relative to the camera using only monocular images. Markers designed for this purpose tend to have lower data capacities than QR codes such that they are easier to recognize in non-ideal conditions, but many use a similar black-and-white square layout, such as ArUco [87] (Figure 9b), ARTag [88] (Figure 9c), and April Tag [89] (Figure 9d). Chroma Tag [90] (Figure 9e) explores the possibility of adding color to this structure. WhyCon [91] (Figure 9g), WhyCode [92] (Figure 9h), and other marker systems such as STag [93] use a circular form. Other examples of fiducial systems are discussed in [94], which compares many of them in terms of errors in pose estimation, jitter, performance under occlusion, computing speed, detection distance and angle of deflection.

April Tag [89], [95] has multiple marker families that are composed of black and white squares. The April Tag 36h11 family, shown in Figure 9d, has a square black border surrounding a 6x6 grid of black and white squares in the center. The name "36h11" denotes 36 ID bits with a minimum Hamilton distance of 11 between each one to avoid false-positive detections. This family has widespread community usage, and April Tag itself has a ROS connector to make it easily usable for robots. Refinements of April Tag produced flexible marker layouts, including the ability to place ID bits outside of the black border, the ability to classify squares as undefined (so they can contain information that does not detract from marker detection), and the ability to use custom placement of the squares [96]. This led to April Tag

(a) QR code. (b) ArUco. (c) AR Tag. (d) April Tag (36h11).



(e) Chroma Tag. (f) April Tag (48h12). (g) WhyCon. (h) WhyCode.

Figure 9: Examples of fiducial markers. There are many spin offs, but many fiducial markers tend to have a square-based, black-and-white layout. Some have added colors, and some have gone with a circular layout.

48h12, shown in Figure 9f, which has several outer ID squares and 4 undefined inner squares where smaller markers can be embedded. If we define such an arrangement of concentric markers as, e.g., `landing_pad_1`, then we can identify it over a wider range of distances, such as when the camera is too far away from the landing pad to recognize the inner markers, or too close to recognize the outer ones.

WhyCon [91], shown in Figure 9g, takes a different approach from that of April Tag, with a simpler definition as a black outer ring with a white inner circle, and it also requires a simpler detector. The WhyCon system cannot distinguish between WhyCon markers, as they do not have any ID. WhyCode [92] solves this problem by adding a Manchester-encoded binary ID that is wrapped around the inner white ring. Since this system can distinguish markers by ID, it can handle markers of different sizes. Importantly, this system separates the recognition and identification steps, meaning that it can recognize that there is a marker in a particular location even if it is too far away to decode its ID.

The April Tag and WhyCode systems both provide 6 DoF pose estimates for their markers. WhyCon can provide only a 5 DoF pose estimate because WhyCon markers do not have an inherent origin in their z axis, from which to determine their orientation. WhyCode's simpler detector requires lighter computational resources than April Tag. It is also less sensitive to occlusions caused by intersection with the drone's landing skids, but also exhibits more problems in the rotational components of its pose estimates (to be discussed in Section 3.4).

Fiducial markers are particularly useful in autonomous landing because they provide a cheap, easy, out-of-the-box way for the drone to localize itself with respect to the landing pad using its monocular camera.

Section 3.3: The gap between simulation and reality

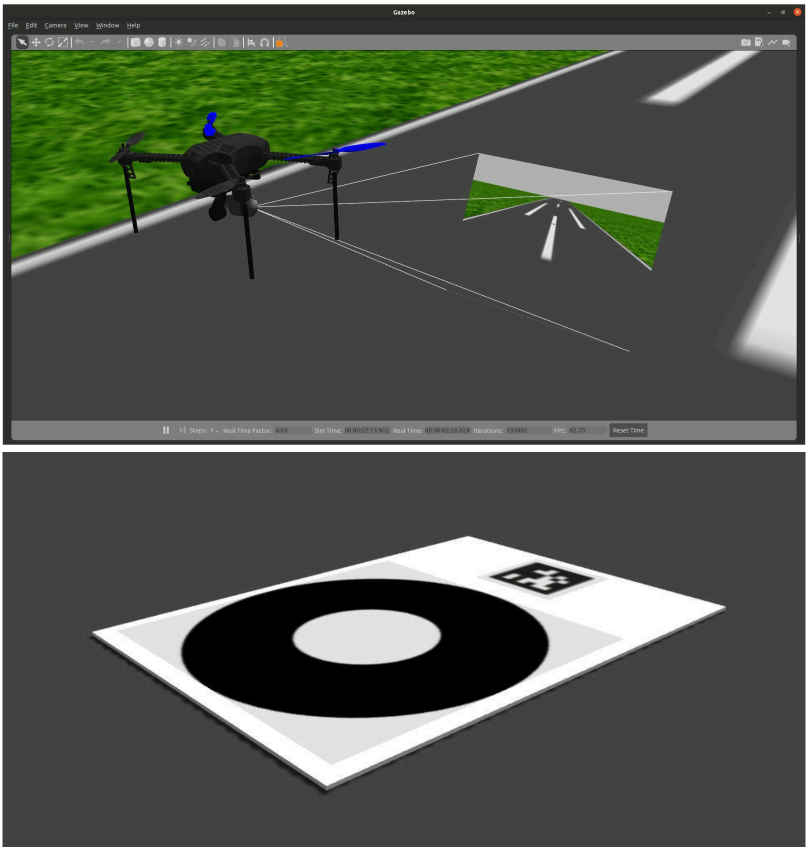
This research is a continuation of the author's master thesis, which presents a method for autonomous drone landing using fiducial markers *in simulation*:

> [97] J. Springer, "Autonomous Landing of a Multicopter Using Computer Vision," 2020. [Online]. Available: http://hdl.handle.net/1946/36422

Gazebo Simulator (see Section 2.7.1), provided a safe, idealized environment in which to learn about and develop software for the ArduPilot (see Section 2.4) software suite, and to create ROS (see Section 2.6) modules to carry out the required computation in a modular, organized, and efficient way. Although one might prefer to have a proof of concept on a real drone, there are some important reasons for developing autonomous control methods in simulation first:

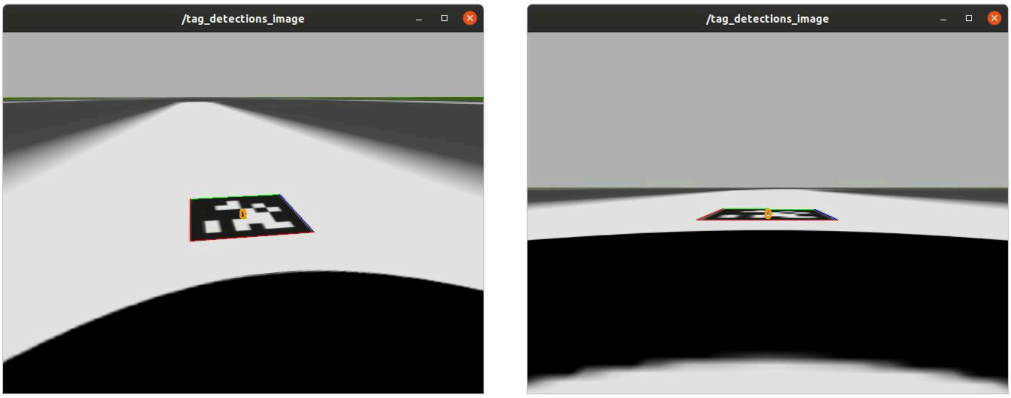
- the cost of crashes is reduced to a simple reset of the simulation environment, instead of a high price tag and many hours of repairs,
- the hardware setup and specifications can be precisely known and can be quickly modified just by editing configuration files,
- logistical considerations, e.g., travel between the lab and a flying location, airspace restrictions, and inclement weather, are eliminated.

All of these aspects make simulation an ideal environment for rapidly developing control methods without the punishing attrition from prohibitively expensive crashes during the process. Importantly, it also helps one to become attuned to the *personality* of the autopilot software one is using i.e., the often critically under-documented and unexpected behavior by the software in edge case scenarios, which is not predictable by the user, though it may be known by the developer. The product of this master thesis was a method for autonomous landing using fiducial markers, which depended on the *Iris* quadcopter model and ArduPilot, as shown in Figures 10 and 11. Generally, it locates the landing pad by identifying its markers in the camera's field of view, calculates the relative position from the drone to the marker,



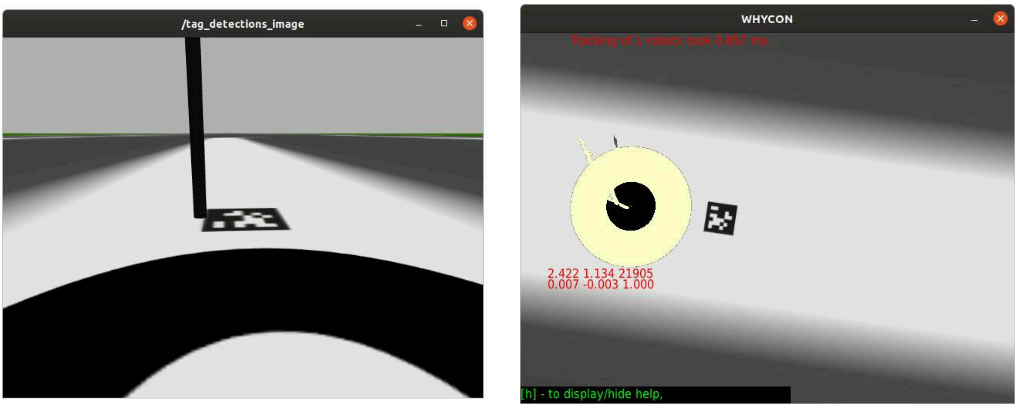
Steps: 1 Real Time Factor: 0.83 Sim Time: 00 00:02:13.902 Real Time: 00 00:02:26.624 Iterations: 133902 FPS: 62.70

Figure 10: Screenshots from the author's master thesis method developed in Gazebo. Top: *Iris* quadcopter and virtual camera sensor/view. Bottom: landing pad with larger WhyCon marker (circular) and April Tag marker (square).



(a) April Tag 36h11, normal detection.

(b) April Tag 36h11, high-deflection detection.



```
Tracking of 1 robots took 0.857 ms.
2.422 1.134 21905
0.007 -0.003 1.000
[h] - to display/hide help,
```

(c) April Tag 36h11 detection blocked by minor occlusion.

(d) WhyCode detection successful with occlusion.

Figure 11: Demonstration of sensitivity to occlusion in April Tag and WhyCon. (From the author's master thesis [97].) The detector places an ID of 0 over the marker in Figure 11a and highlights the border of the marker, showing a detection. It can function well even at high angles of deflection, such as in Figure 11b. However, Figure 11a shows that April Tag is very sensitive to slight occlusions that result in contiguous black regions touching the marker's black border, thus making it appear less square and preventing detection. On the other hand, WhyCon maintains its detection even when the shadow of the drone intersects the black, outer ring; it just treats it as a sort of deformation in the marker. In Figure 11d, the marker has been highlighted/inverted by the detector.

and generates a target velocity accordingly. The drone then approaches the marker and descends.

Section 3.3.1: Real world implementation

The natural first step of this PhD research was to migrate the method from simulation into the real world, which was supposed to be the simpler task when compared to developing the method itself, but which turned out to be a much longer and more tedious process than expected. We began by constructing two Tarot 680 hexacopter platforms, according to a plan laid out in the thesis. These custom, hobbyist kits were well reviewed in the community and had the payload capacity to carry a companion board. The drones are primarily radio controlled (RC) vehicles, so they have a typical hardware setup as shown in Figure 12. The RC receiver takes commands from the pilot's transmitter and relays them to the flight controller, which generates PWM commands to control the speed of each motor. A telemetry radio allows for higher level communication, which the operator can access via a laptop, giving commands such as ordering the drone to takeoff, or to fly to a specific coordinate. We use two separate power systems in order to shield the flight controller and companion board from surges caused by the motors. We use a gimbal to aim the camera.

The flight controller unit (FCU) on both drones is a Raspberry Pi 3 B+ with a mounted Navio2 [63] shield, which provides inertial measurement data. The autopilot software, *ArduPilot* [61], runs as a system service in the Emlid distribution of the operating system (OS), which is designed specifically for the Navio2. We equipped one drone with a Google Coral development board, and the other with a Jetson Nano to act as companion boards, handling image processing in a computational environment isolated from that of the flight controller which is real time critical. A ROS module called MAVROS [98] communicates with ArduPilot via its MAVLink [60] language in order to enable networked communication with the drone's companion board. Both companion boards have USB OTG ports, simplifying the network connection between them and the Raspberry Pi.

We designed and 3D printed a component mounting plate (Figure 13a) to mount the computational electronics, and adapters to fit the companion boards' custom camera modules into gimbals (Figure 13c and Figure 13d), which were meant to hold GoPro cameras. We additionally 3D printed a canopy [99] (Figure 13b) to protect the computational components. These parts are shown in Figure 13. The completed drones are shown in Figure 14.

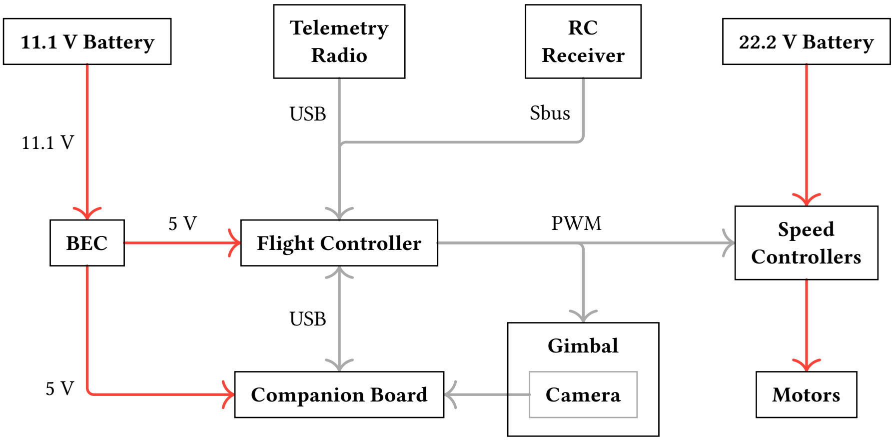
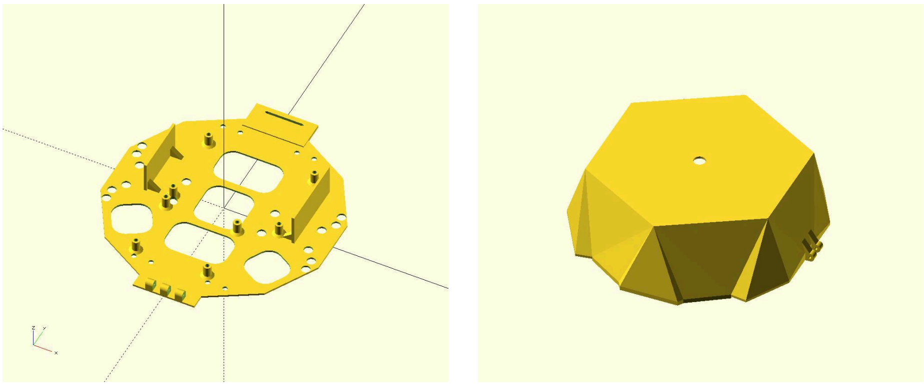
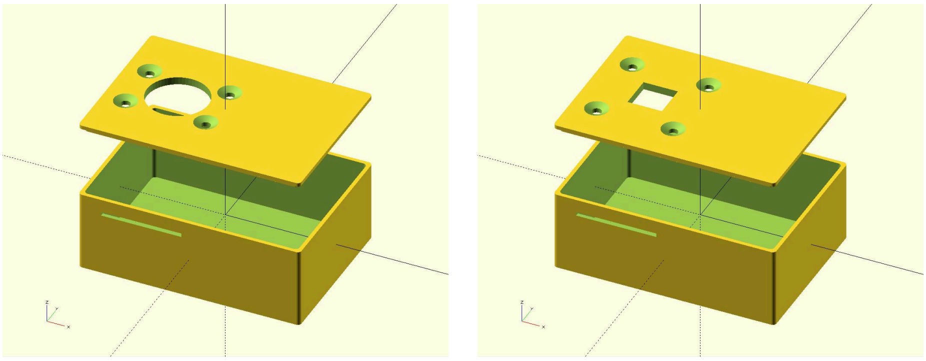


Figure 12: Hardware setup for the Tarot 680 hexacopters. Data connections are in gray, and power connections are in red. An 11.1 V battery powers the flight electronics via a battery eliminator circuit (BEC), and a 22.2 V battery provides power to the gimbal, speed controllers, and motors in an isolated system. The drone receives commands from an operator via an RC receiver, and sends and receives higher-level messages using a telemetry radio. The flight controller manages the behavior of the drone, and a companion board interprets the image from the camera and sends commands to the flight controller. A gimbal aims and stabilizes the camera.



(a) Component mounting plate.

(b) Tarot 680 canopy.



(c) Jetson Nano wide-angle camera module case.

(d) Google Coral camera module case.

Figure 13: 3D printed components for the Tarot 680 hexacopters. Figure 13a shows a mount for the electronic components, and Figure 13b shows the canopy that covers the electronic components. Figure 13c and Figure 13d show adapters to fit the Jetson Nano and Google Coral camera modules into Go Pro form factors for the gimbal.

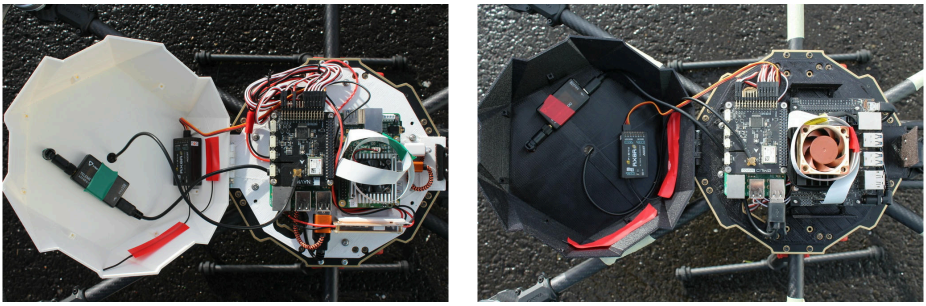
The method for autonomous landing in the author's master thesis is implemented as a set of ROS modules, which we transferred directly to the physical drones, however some components had to be adapted, e.g., to connect the Jetson Nano's camera module to the ROS network required specific modules. Installing the ROS software stack on the Jetson Nano and Google Coral can be a challenge because of the number of dependencies and the specific operating systems which do not have as much popularity as typical Linux distributions that ROS targets, such as basic Ubuntu. Because of this, pre-compiled ROS modules are often not available, and so the whole toolchain may have to be installed from source, which takes significant time and troubleshooting effort.

Section 3.3.1.1: Results

These drones flew reliably in manual mode and were able to track the markers using the gimbal, as illustrated in Figure 15, which shows that the camera is able to keep the pixel position of the center of the marker in frame over several tracking attempts in the lab. We also verified this functionality in flight.

(a) Tarot 680 with Google Coral.

(b) Tarot 680 with Jetson Nano.

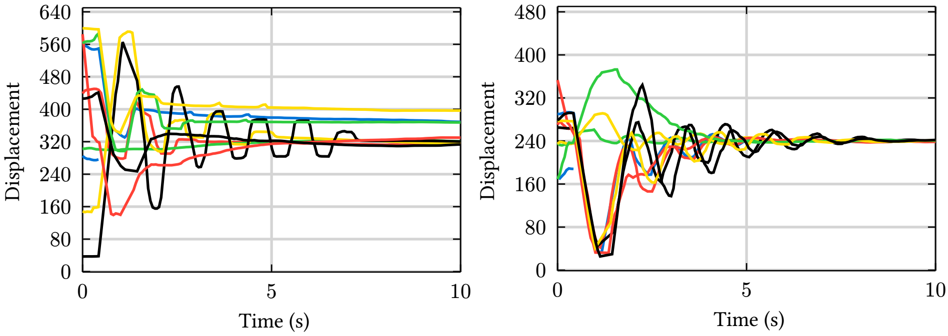


(c) Electronics with Google Coral.

(d) Electronics with Jetson Nano.

Figure 14: The finished Tarot 680 hexacopters with onboard flight controllers and companion boards.

However, the drones were never able to carry out autonomous landings for two primary reasons. The first, and less conceptually interesting problem for this research, is that the drones' onboard GNSS system was unable to receive signals from enough satellites to attain an adequately low geometric dilution of precision (GDOP) — in other words, it was unable to calculate its position accurately enough. The result was that the autopilot software refused to enter `auto` mode, thereby preventing the drone from responding to commands from the autonomous landing method. Raising the GDOP limit risks erratic and dangerous behavior by the drone



(a) Coral drone x axis tracking performance.

(b) Coral drone y axis tracking performance.

Figure 15: Tracking performance from the Google Coral drone. It shows that the position of the marker in the camera frame converges to the center of the frame, or near the center of the frame. The tracker is tuned tightly, leading to some decaying oscillations. In some cases, the gimbal reaches its actuation limit, so it cannot completely center the marker in the x axis.

because its position estimate changes unpredictably over time. Later boards were able to access more GNSS satellites, thereby receiving positioning data from more satellites and achieving adequately low GDOP in Iceland. This problem was not apparent in Gazebo, as it abstracts from these sorts of issues by design, isolating the development process of the control method from the lower-level issues of the drone hardware itself. Although this problem could possibly have been solved by adding an RTK system, this was unavailable at the time and goes against the design principles of this research, since the RTK system requires extra, powered infrastructure and limits mission range.

The second problem — and the more difficult one — is that the simple, idealized appearance of the objects (the fiducial markers in particular) and environment within Gazebo partially obscured the issue of orientation ambiguity in fiducial marker systems. The pose (the position and orientation) of the landing pad was calculated using the full 6 DoF pose of the fiducial markers attached to it, but the orientation component is actually unreliable, as we will study in the next section. This resulted in erroneous control signals that directed the drone towards the wrong landing location. However, within the Gazebo environment, this was noticeable only in the WhyCode system. The WhyCon and April Tag systems did not exhibit orientation ambiguity to the same extent, such that successful autonomous landings were possible using the method. Unfortunately, orientation ambiguity reappeared in the real world once the system was using non-idealized camera sensors with inexact distortion parameters (sometimes made even worse because of auto-focus), slight blurring from camera vibration, fiducial markers that were printed on paper and mounted to boards such that they were not perfectly planar, glares and shadows, etc. These complications prolonged the testing period — anticipated to last only for the summer — well into autumn, when it became infeasible due to Iceland’s harsh weather. As a result of this setback, we developed a preference to avoid simulation in general and preferred to make conclusions based primarily on experiments in the real world, or at least pair conclusions from simulation with real world proofs of concept, as will be shown in the rest of this work. A common workaround for this extreme measure is to instead pair simulated experiments with in-lab testing of a control method using real hardware, i.e., hardware-in-the-loop (HITL) instead of purely software-in-the-loop (SITL), which gives empirical evidence for what runtime performance is to be expected. However, such testing can still conceal challenges that may appear in the real world. Therefore, in order to test our methods holistically, we prefer to deploy them on real world systems.

Unfortunately, real world testing involves an incredible amount of overhead, which was the main time sink in this research. The most salient task in this regard is building the computational system and integrating it into the drone. This is already a highly involved task in the case of DIY drones and can become prohibitively troublesome in the case of enterprise drones. For example, developers who primarily need to deploy a sensor package may prefer to duplicate power supplies and telemetry equipment in order to get real time data from their drone without using the power and transmission infrastructure that their drone already provides. We spend a lot of time integrating our systems with our drone platforms to avoid this kind of duplication. Further, while the basic effect of severe weather is to limit the days on which one can actually conduct aerial tests, weather considerations also force weatherproofing of any computational systems one might add to a drone. The cases for the computational systems also must be refined to, e.g., quick-release systems for rapid deployment because prolonged assembly time in the field (often with frozen fingers) inevitably leads to clumsy assembly, dropped and lost screws, and general inconsistency. Lastly, Reykjavik University is situated directly next to Reykjavik Airport, placing it deep within a no-fly zone and meaning that all drone fieldwork in this research required an extra step of transportation to a different field location, in addition to drone pilot certifications.

Section 3.3.2: Conclusion

This project did not provide an answer to **RQ1**, but did provide some insight into the real world difficulty of using fiducial markers, particularly with regard to ambiguity in the orientation components of their perceived pose, and motivated further investigation. It provided some insight into **RQ3**, in that the autonomous landing method from the author's master thesis did technically run on the embedded hardware, and the drone platforms were able to fly with the hardware.

Section 3.4: Fiducial markers and orientation ambiguity

The difficulties in the previous section warranted further examination into fiducial systems, particularly because other publications had successfully used fiducial systems for autonomous landing in the past. The primary difference is that those projects tended to avoid using the marker's 6 DoF pose, of which the rotational components are subject to sudden apparent flips in the x and y axes. We refer to this phenomenon as *orientation ambiguity*. Because of orientation ambiguity, commands generated using the full marker pose were unstable and did not direct the drone reliably towards the landing pad. Therefore, we set out to determine if some common fiducial systems exhibited this issue less than others, or if we could modify them to do so, in order to make commands generated from such marker detections more reliable in a deeper way than, e.g., simply filtering them. We work towards quantifying the prevalence of orientation ambiguity in 5 fiducial marker systems by treating the marker as a physical object whose pose must be continuous over time, and then detecting discontinuities in its position and velocity which indicate axial flips.

This section describes in detail the following publication, which is a quantification of *orientation ambiguity* exhibited by several fiducial systems, as well as an evaluation of such systems for real time, embedded performance on a Raspberry Pi 4:

> [38] J. Springer and M. Kyas, "Evaluation of Orientation Ambiguity and Detection Rate in April Tag and WhyCode," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, Naples, Italy: IEEE Computer Society, Dec. 2022, pp. 281–286. doi: 10.1109/IRC55401.2022.00054.

Section 3.4.1: Related work

Irmisch [100] has analyzed the performance of April Tag and WhyCon markers in terms of their abilities to determine correct position in real world experiments. The experiments show that both April Tag and WhyCon have relatively low position estimate errors. The issue of ambiguity in planar marker orientation is a known problem, and various methods exist which attempt to mitigate it. One example is an edition of ARTag called LentiMark [12], which uses special Moiré patterns on the outside of the tags to determine a correct orientation. Another study [101] proposes to determine the correct solution using an averaging algorithm with multiple views of the markers, instead of using only a single image. The method in [102] proposes to consider the marker as a moving object, and therefore reduces orientation ambiguity by intelligently choosing the orientation using a motion model.

These studies acknowledge the issue of orientation ambiguity and have proposed some strategies to mitigate it, which mostly focus on augmenting the pose estimation process. We prefer not to use such significant augmentations, but rather to use primarily existing, open source technologies and to quantify the prevalence of orientation ambiguity in those systems with regard to our specific scenario. We do allow for minimal modifications that integrate smoothly into the conventional pose estimation process.

Section 3.4.2: Methods

We captured videos of an arrangement of 4 fiducial markers, as shown in Figure 16, where the markers remained stationary while the camera translated and rotated around the markers in multiple ways, keeping all markers fully in frame at all times. After collecting these videos, we analyzed them offline to detect orientation ambiguity and to determine runtime detection rate on our Raspberry Pi 4, which has a larger community and is generally easier to use than our previous Jetson Nano and Google Coral.

(a) April Tag 48h12.

(b) WhyCode.

(c) April Tag 24h10.

(d) WhyCode Multi.

Figure 16: Fiducial markers evaluated for detection rate and orientation ambiguity. The markers were placed close together without their labels. Each marker remained entirely the camera's field of view for every frame of every video.

Section 3.4.2.1: ROS setup and detection of orientation ambiguity

We conducted offline processing using customized versions of the April Tag and WhyCode ROS modules, which we edited to expose specific attributes of the marker detections:

1. A **position target** denoting the translation in meters from the camera to the center of the marker in 3 components *right*, *forward*, and *up* (originally called *east*, *north*, *up*, imitating the directions of a compass centered on the drone as part of the convention in ArduPilot). This message is supposed to be used in autonomous landing scenarios, and therefore it makes the warranted assumption that the marker is flat on the ground and face-up.
2. The Euler components of the marker's **orientation**, i.e., yaw, pitch, and roll, for straightforward calculation of angular speed.
3. The **normalized pixel position** $u_n, v_n \in [-1, 1]$ which denote the x and y positions of the center of each marker, respectively. These are simpler inputs to the tracking system that will be used to aim the camera at each marker when we test for autonomous landing later, in Section 3.5.

Like the related work [102], we analyze marker pose with an implicit motion model, detecting orientation ambiguity by the discontinuities (i.e., sign flips and other spikes) that it causes in the pitch and roll components of a marker's orientation, which propagate to subsequent calculations that depend on the marker orientation — the position targets. They cause the system to perceive artificially high linear and angular speeds. We compare the fiducial systems on the basis of how many discontinuities they generate in both the translational and rotational components of the marker's pose when viewed as a time series. For the translational component, discontinuities occur when the inequality in Equation 10 is true:

$$\frac{p_{x,i+1}}{p_{x,i}} < \theta_l < 0 \tag{10}$$

where $p_{x,i}$ and $p_{x,i+1}$ are x components of the position targets in a single dimension at frame i and $i + 1$ respectively. We consider that a linear discontinuity occurs when Equation 10 holds for at least one of the x,y, and z components. When it is true, the linear displacement of the marker is sufficiently opposite in one frame compared to the previous frame, implying that the marker has jumped. However, this inequality erroneously implies discontinuities when any of the components are near 0 because of small movements and measurement noise. So, we pair it with a check for spikes in the marker's angular speed:

$$s_a = \frac{\mathrm{dist}(q_i, q_{i+1})}{\Delta t} > \theta_a > 0 \tag{11}$$

where s_a is the marker's angular speed, q_i and q_{i+1} are the orientations of the marker at frames i and $i + 1$ respectively (implemented as quaternions), "dist" is the quaternion distance function (which gives the magnitude of the angle between the two orientations), and Δt is the change in time between frames i and $i + 1$. When Equation 11 is true, it shows a spike in the marker's angular speed. When the inequalities in both Equation 10 and Equation 11 are true, we consider the marker system to have exhibited orientation ambiguity on the given frame. We define a *discontinuity rate*, $r_d = \frac{d}{n}$, for each system for each video, where d is the number of discontinuities exhibited and n is the number of frames in each video.

For this comparison, we collect 33 videos of the fiducial marker arrangement in Figure 16, so that each marker system is treated in the same way. Each marker is printed with a side length of 30 cm and affixed to a planar surface with clear lighting. The camera can move around the marker arrangement, but always keeps every marker fully in the field of view for every frame. The camera stays still during a calibration video, which functions as a control and allows us to check that each marker system is configured correctly, which we verify by checking that they all show a z distance of 0.96 m. In other videos, we vary the distance between 1 m and 3 m and move the camera in specific ways: translating left-right, up-down, and forward-backward; rotating left-right, and up-down; and orbiting left-right, and up-down. We move the camera manually, but slowly and deliberately. After initial experimentation, we are able to guarantee that the angular speed does not exceed $\theta_a = 1 \frac{\mathrm{rad}}{\mathrm{s}} \approx 57 \frac{°}{s}$, while angular discontinuities often do, so we set this as the threshold. Similarly, since we maintain a continuous motion, we assert after initial experimentation that the θ_l should be -0.8, in order to avoid classifying linear discontinuities erroneously (except at near-0 values of the translational components of the position targets).

Section 3.4.2.2: Tested and modified fiducial systems

Some systems prefer using more complex notions of the marker pose, i.e., [12] which adds components in order to more accurately perceive the orientation from a single frame, and [101] which takes multiple views of the marker into account. We prefer to only extract the orientation from a single frame and not to add fundamentally new components to the marker, such as Lentimark's Moiré patterns [12]. Our modifications are limited to adding extra phases of sampling and considering multiple markers at at once.

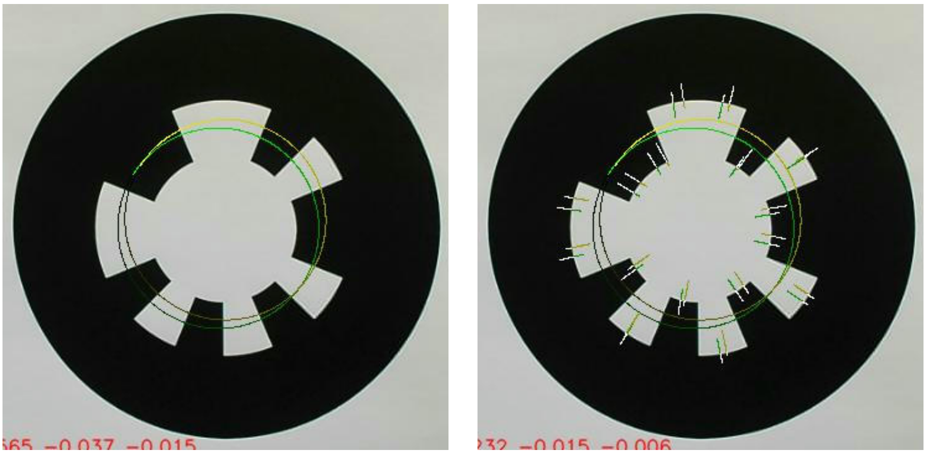
We evaluated orientation ambiguity rates on 5 systems — 2 default and 3 with modifications. The first system, **April Tag 48h12** (see Figure 16a), is one of the defaults, and we test it as a replacement for April Tag 36h11, used in the master thesis method, because it allows for concentric marker embedding. During the tests in the original method, the drone would occasionally lose its detection of the landing pad as a result of the landing pad being too large for the camera's field of view. The paper proposing April Tag 48h12 [96] specifically suggests this marker for autonomous drone landing for this reason, so it was a natural contender.

The second system, **April Tag 24h10** (see Figure 16c), is a non-default April Tag variant that follows the structure of April Tag 48h12 with fewer bits. We test this in an effort to make the April Tag system run faster with limited computational resources, as April Tag 48h12 had a low runtime detection rate on our Raspberry Pi 3 during initial experiments. The April Tag system works by loading a hash table of all possible IDs of the relevant family (which includes 4 different rotations of each ID) into a single, large hash table in memory at runtime. This hash table was more than 1GB, meaning that it exceeded the RAM size of the Raspberry Pi 3 and required swap space, making the runtime detection rate on the order of 2 Hz. April Tag 48h12 has a width of 10 squares in its definition, with the 4 central squares undefined, giving 42,211 possible markers. The April Tag 24h10 family has only 7, with the central square undefined, giving only 18 possible markers, and resulting in a much smaller hash table. In the scenario of autonomous drone landing with fiducial markers, it is not inherently a priority to have thousands of different markers, so we test this warranted tradeoff for a potential increase in execution speed. This marker system was generated using the April Tag Generation library [103].

The third system, **WhyCode Orig** is a refined version of WhyCode [104], [105]. The WhyCode detector works by detecting black, elliptical regions with white, elliptical regions inside them, and then comparing the regions based on their semi-axes, with a threshold defining how much they should match. The is a reliable method for marker detection, and the *position* of the marker can be derived from the detected entity if the size is known. However, the *orientation* of the marker cannot be determined unambiguously using its semi-axes, as 2 solutions exist for the same pair of semi-axes under the warranted assumption that the marker is facing the camera. The original WhyCode paper [92] left this problem for later, calculating both orientations but simply returning the first one always. Ulrich added a method for distinguishing which solution to the orientation was more likely, based on projecting an ellipse onto the marker in the predicted location of the center of the ID "teeth." Then, the arc length of each tooth compared to its expected arc length could be used to determine which solution more accurately fit the picture, especially in extreme cases where the sampling circle may not intersect some teeth. An illustration of the two sampling ellipses is shown in Figure 17a.

The fourth system, **WhyCode Ellipse**, extends the idea of sampling the arc length of predicted locations of the ID teeth but samples them radially, in the hopes that this will give more sensitivity to the "centeredness of each solution." After the initial ID sampling occurs, we can predict the locations of the edges of each ID tooth. The IDs are Manchester-encoded based on their height, i.e., each ID bit has one "tall" and one "short" white region, as measured from the center of the marker. Given a marker detection in an image with the ID determined according to the default WhyCode method, we conduct a secondary sampling phase by starting from the center of the marker and sampling outward along a radius, looking for a white-to-black transition. We do not have to sample the entire radius, but can simply sample in a neighborhood of the expected transition. We then determine how aligned the corresponding orientation solution is to the marker by determining the variance in the positioning of the transition along the sampling lines, with lower variance being better. This method is illustrated in Figure 17b, where the sampling lines are drawn in over the image such that they appear white where the marker is black, and they appear either yellow or green where the marker is white, depending on if they correspond to one possible solution or the other. Intuitively, a well-centered solution should place the white-to-black transitions in the middle of the radial sampling lines. For example, in the detection shown in Figure 17b, the green solution is chosen because the white-to-black transition occurs closer to the center of the sampling lines. This implies a lower variance as compared to the yellow solution, where the transition actually happens closer to the start or end of each sampling line depending on its angle around the marker.

The fifth system, **WhyCode Multi**, is based on the fact that the *positions* of the WhyCode markers are relatively accurate, and that it is possible to unambiguously determine the orientation of the plane connecting 3 or more points. An example marker arrangement is shown in Figure 16d, where each marker has a different ID from the others and from the marker used to evaluate WhyCode Orig and WhyCode Ellipse, in order to avoid collisions, since all marker systems are tested on the same video test cases that include all markers in Figure 16. This system is essentially a wrapper around the WhyCode Orig system, which extracts the positions of each marker in the arrangement, determines the orientation of the plane connecting them, and then assigns that orientation to both the plane and to each marker.



(a) Example detection from the WhyCode system defined in [104] ("WhyCode Orig").

(b) Example detection from "WhyCode Ellipse"

Figure 17: Example detections from the 2 single-marker WhyCode systems. WhyCode Orig samples the ID teeth in their arc length, while WhyCode Ellipse samples them radially, hoping to improve the system's ability to determine which solution is better aligned to the marker.

The modifications that we make to these systems are not as significant as in the related work, which is important because it means that they can provide a pose estimate for the relevant landing pad without adding specialized augmentations to the appearance, and without making the pose estimates dependent on multiple frames. These requirements would limit the types of landing pads the drone could use and would also influence the landing process itself.

Section 3.4.3: Results

Aggregated results comparing the overall discontinuity rates of each system are presented in Table 2 and in Figure 18. WhyCode Ellipse achieves the lowest discontinuity rate over the systems, although it is not statistically separable from April Tag 48h12. WhyCode Orig and WhyCode Multi exhibit higher orientation ambiguity, and April Tag 24h10 exhibits very high orientation ambiguity. Table 3 and Figure 19 show aggregated detection rates. WhyCode Orig is the fastest, and WhyCode Ellipse has a predictable slowdown as a result of its extra sampling phase. WhyCode Multi is the next fastest, with a similarly predictable slowdown as a result of its need to detect multiple markers simultaneously. April Tag 48h12 and April Tag 24h10 show similar detection rates, as the test platform (a Raspberry Pi 4) had enough RAM (2 GB) to deal with the large hash table of April Tag 48h12, so the smaller size of the April Tag 24h10 family did not give any benefit.

Figure 21 illustrates the effects of orientation ambiguity in the position targets that denote the relative displacement from the drone to the landing pad. In this case, the position targets are denoted in an east, north, up (ENU) format, corresponding respectively to the directions right, forward, and up, from the camera's perspective.

System	$\overline{r_d}$	$\hat{\sigma}$
WhyCode Ellipse	0.014	0.028
April Tag 48h12	0.026	0.039
WhyCode Orig	0.056	0.083
WhyCode Multi	0.064	0.102
April Tag 24h10	0.161	0.165

Table 2: Mean discontinuity rates for each of the fiducial systems in our evaluation. Lower is better. We calculate the discontinuity rate, $r_d = \frac{d}{n} \in [0, 1]$, for each system for each video, where d is the number of discontinuities exhibited and n is the number of frames in each video. Then, $\overline{r_d}$ is simply the mean of r_d for the fiducial system over all videos.

System	F (Hz)	$\hat{\sigma}$ (Hz)
WhyCode Orig	26.232	0.444
WhyCode Ellipse	26.096	0.365
WhyCode Multi	22.881	3.509
April Tag 48h12	16.270	1.527
April Tag 24h10	16.154	1.760

Table 3: Mean detection rates for each of the fiducial systems in our evaluation.

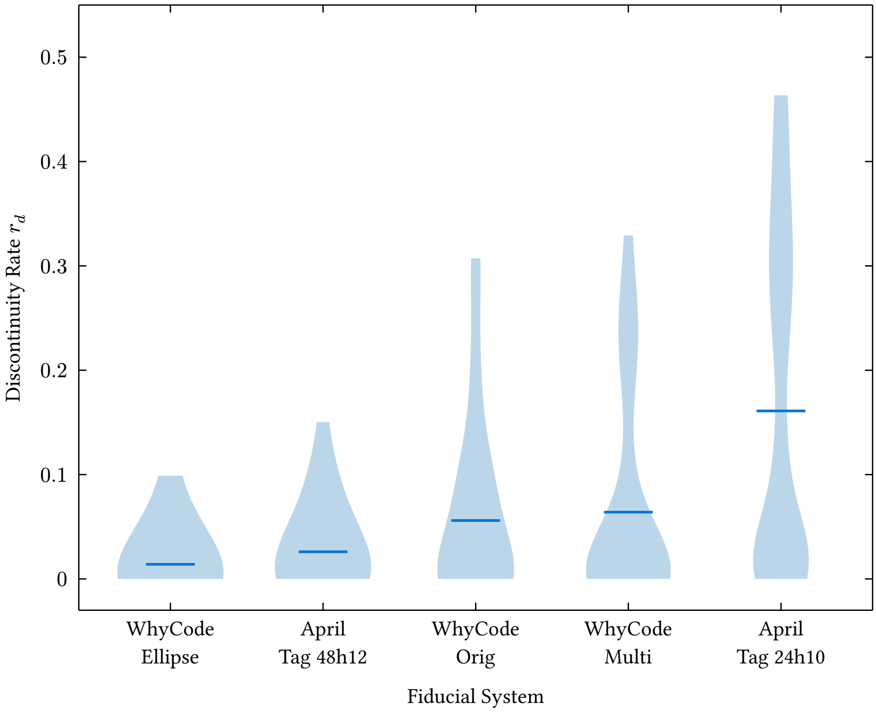


Figure 18: Distribution of discontinuity rates for all systems over all test cases. The systems are sorted from left to right by increasing mean discontinuity rate. Means are indicated by horizontal lines.

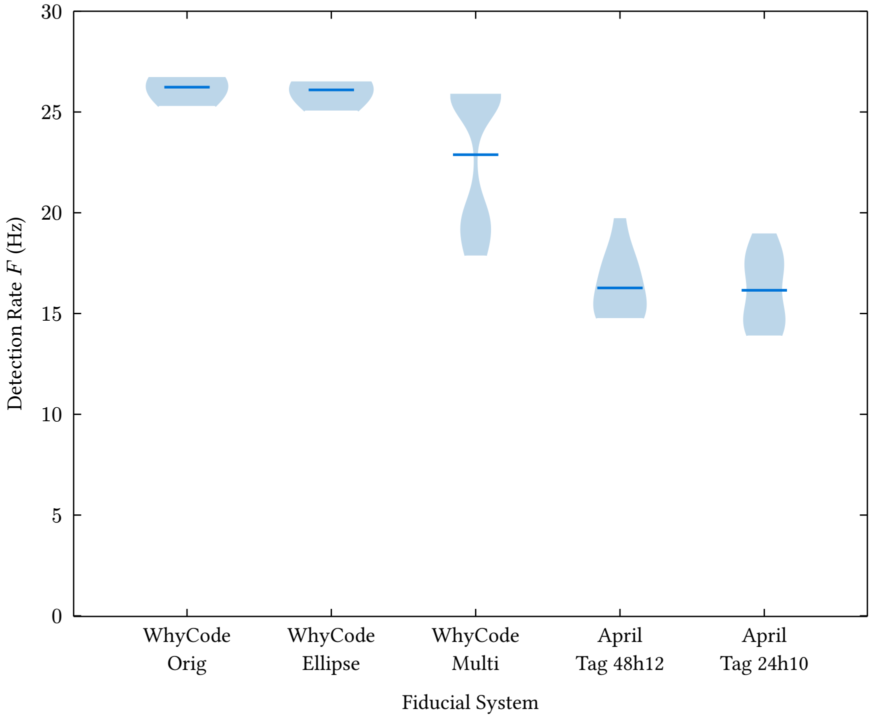


Figure 19: Distribution of detection rates for all systems over all test cases. The systems are sorted from left to right by decreasing mean detection rate. Means are indicated by horizontal lines.

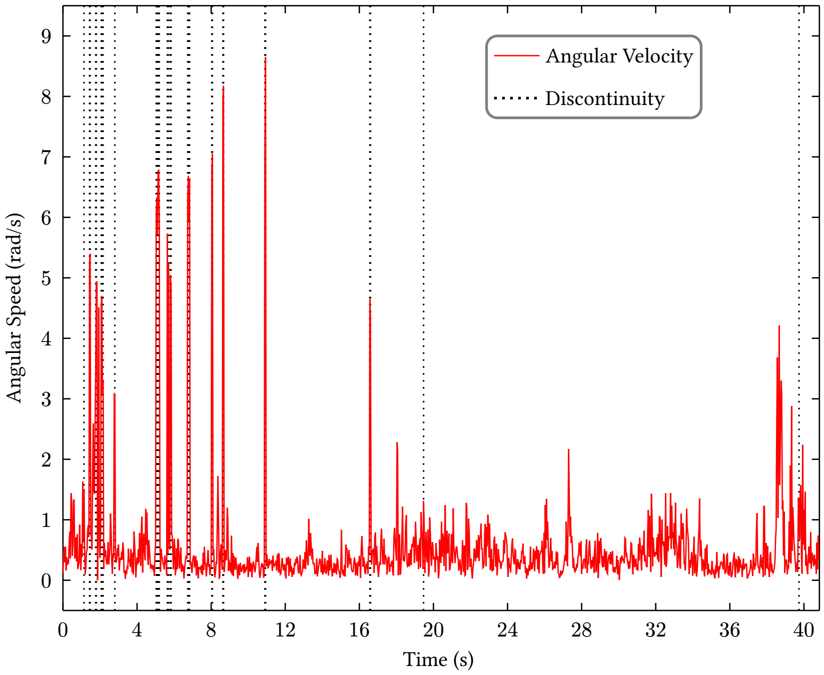
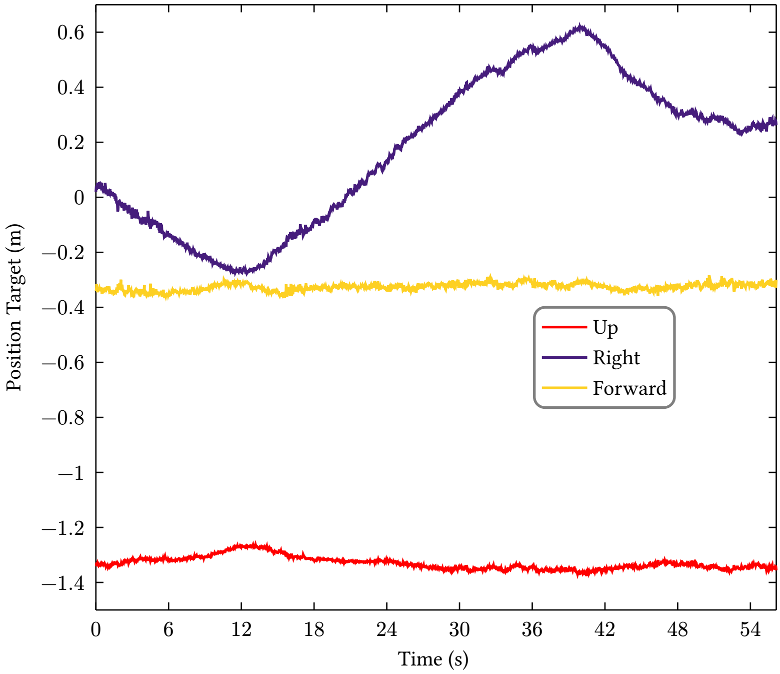
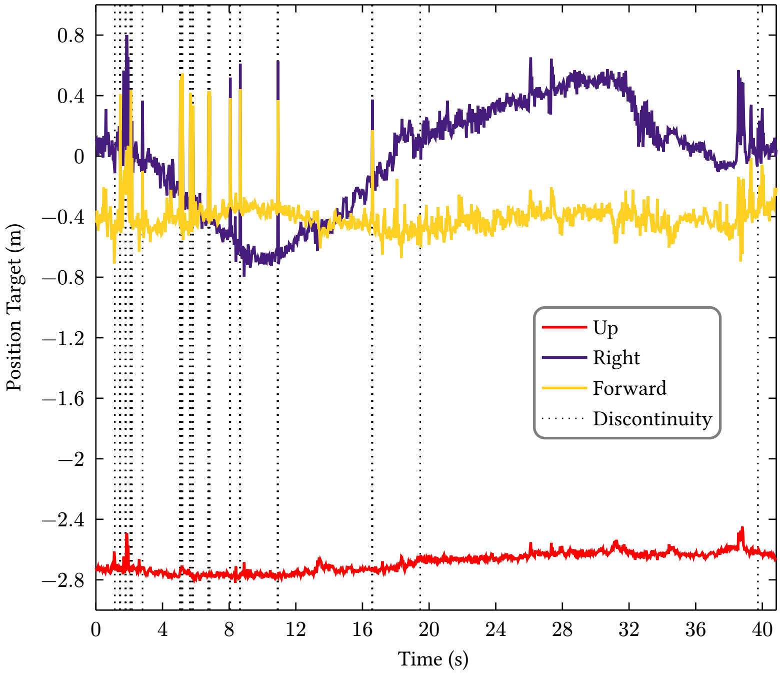


Figure 20: Angular speed corresponding to the test case with April Tag 48h12 shown in Figure 21b. This illustrates the spikes that occur as a result of orientation ambiguity. Some of the spikes in angular velocity do not correspond to sufficiently large spikes in the position targets, so they are not considered to be discontinuities.



(a) An example test case with no discontinuities.



(b) An example test case with several discontinuities, corresponding to Figure 20.

Figure 21: Plots of position targets over time for two test cases with April Tag 48h12. The camera is moved right and left, while keeping the marker in the center of the frame. The test case in Figure 21a is relatively smooth with only minor noise, whereas the test case in Figure 21b shows sign flips in the Up and Right position targets as a result of orientation ambiguity. These *discontinuities* are marked by dotted, vertical lines.

It shows two test cases in which the camera follows generally the same motion: translating left, then right, then back to the center, while rotating to keep the marker arrangement in the center at all times. In Figure 21a, the April Tag 48h12 system shows a correct trend in the east component of the position target, as the camera moves left and right. The camera maintains a distance of about 1.25 m from the marker. However, in a similar version of that test case taken from 1.75 m away from the marker, shown in Figure 21b, the system exhibits many discontinuities as a result of orientation ambiguity. These primarily occur as sign flips in the east and north components of the position target and are marked with dotted lines. Further, Figure 20 shows the spikes in the angular speed of the marker that occur with the orientation ambiguity in this same test case. These phenomena are representative of the effects of orientation ambiguity in the other systems, including the higher prevalence as the camera moves farther from the marker, since it is harder for the detector to determine the marker's orientation given smaller pixel areas.

Section 3.4.4: Conclusion

We evaluated 5 fiducial systems in terms of orientation ambiguity and detection rate. Some of the systems had simple modifications that did not fundamentally change their structure, but did, in the case of WhyCode, reduce the orientation ambiguity rate. This gives insight into **RQ1**, in that we expect every system to generate some erroneous control signals during autonomous landing, and that April Tag 24h10′s high orientation ambiguity rate would mean many such erroneous control signals and likely problematic behavior of the drone. It also gives insight into **RQ3**, in that all the systems can run in real time on a Raspberry Pi 4, which can be embedded onto a drone.

The natural next step, as we lay out in the next section, was therefore to use these systems in a real world autonomous landing scenario in a safe way that mitigated the potential dangers of the erroneous control signals.

Section 3.5: The effects of orientation ambiguity on autonomous landing

Since all of the tested fiducial systems from the previous section were subject to orientation ambiguity, and since the only known ways for eliminating the orientation ambiguity are with additional techniques outside of conventional fiducial image sampling, we decided to conduct another real world test of our experiment in Section 3.3 with a safer drone indoors. This would allow us to directly observe the effects of the erroneous control signals. We chose a DJI Spark, which, although it had neither the capacity nor interfaces to use an onboard computer, could be controlled autonomously via DJI's MSDK. The drone was so small and harmless that indoor experiments with a potentially erratic drone were not infeasible. Further, the drone's control system was advanced enough such that it could hold its position reliably and be controlled using velocity targets indoors without GNSS, because of its vision system.

This section summarizes the following publication, which tests the fiducial systems from Section 3.4 in an autonomous landing scenario:

> [39] J. Springer and M. Kyas, "Autonomous Drone Landing with Fiducial Markers and a Gimbal-Mounted Camera for Active Tracking," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, Naples, Italy: IEEE Computer Society, Dec. 2022, pp. 243–247. doi: 10.1109/IRC55401.2022.00047.

Section 3.5.1: Related work

Some projects have accomplished precision drone landing with fiducial markers and a camera that is downward-facing and rigidly-mounted, or gimbal-mounted (for stabilization) but oriented vertically downwards. In some cases [10], [11], they use multiple ArUco markers of different sizes to maintain detection of the landing pad as the drone approaches very close. In the case of [4], the drone detects a single, custom X-shaped marker with 2 fixed cameras — one pointing down and the other pointing forward and down. In [7], the drone has a front-facing camera and a downward facing camera, and detects a single marker (AR Tag, April Tag respectively) for landing on a moving boat. One project [5] uses GPS for an initial approach, and an April Tag for final approach. One method [6] uses a DJI Phantom 4, which has a gimbal-mounted camera, but keeps the gimbal pointed straight down at the ground when detecting fiducial markers. The drone in [9] uses a fixed, downward facing camera, but does address the issue of orientation ambiguity using many co-planar April Tag markers, to determine the position of the landing pad through voting. In [8], the drone has a single camera facing forwards and down, and detects April Tag markers on its landing pad. Finally, one method [13] uses a closed-source, marker system called LentiMark [12], which mitigates the orientation ambiguity problem using Moiré patterns on the outside of otherwise conventional AR Tags. They embed a single LentiMark marker inside of an AR Tag, mount them on a post, and land a drone autonomously while tracking the marker during landing with a gimbal-mounted camera. The drone lands on the ground in front of the post.

Some of these projects report visual loss of the marker during landing, as a result of the lack of camera tracking. They use cameras at fixed, known orientations, but sometimes use multiple cameras to accomodate a longer detection range. LentiMark enables marker tracking during landing but depends on a closed-source, augmented marker system.

Section 3.5.2: Experimental setup

Most of the related work uses one or more fixed cameras, or many fiducial markers to get around the problem of losing track of the marker and mitigate the orientation ambiguity problem. Following our preference for minimalism, we maintain an overall structure of using only a single, gimbal-mounted camera (since it is commonly found on modern drones), and only a single marker (or set of concentric markers) per site. We then quantify the effects of the erroneous control signals caused by orientation ambiguity in several autonomous landing scenarios, and compare the performance of the fiducial marker systems we evaluated in Section 3.4, which are all open source. We propose two modified marker systems that have not previously appeared in autonomous landing scenarios. Our method actively tracks the marked landing pad with a gimbal-mounted camera independently of the drone's movement, enabling both autonomous precision landing in unfriendly environments (e.g. wind) and the ability to safely search for the marker by simply spinning in place and tilting the camera up and down.

We use a DJI Spark [106] (as shown in Figure 22) for this experiment. It is a small quadcopter (143 mm x 143 mm x 55 mm without propellers and guards) that is safe to fly indoors. The DJI Spark has a positioning system that can use GNSS, but it also has a downward-facing vision system with which it can maintain its position without connecting to those external positioning systems, as long as the surface below is clearly patterned and lit. This vision system generally supports accurate, stable flight, such that if one nudges it with a finger while it is hovering, it will return automatically to within millimeters of its original location. It does, however, have issues with accuracy if the surface is too self-similar or reflective, so we conduct experiments only above a non-reflective, cement floor with several asymmetric markings. The Spark has a front camera mounted on a 2 axis gimbal with adjustable tilt (i.e., pitch) and stabilization in the roll axis.

Since the drone is small, it is not able to carry a payload, and it has no interface to interact with one. We therefore use the MSDK to interact with the Spark via an Android tablet connected to its controller. However, we can still simulate using our Raspberry Pi 4 companion board by adding it into the control loop, with a bit of hacking. The data flow is shown in Figure 23 and as follows:

- The drone and the controller communicate through a proprietary, wireless telemetry link, streaming video, location, speed, attitude, etc. from the drone to the controller, and streaming commands from the controller to the drone.
- The controller connects to the Android app via a USB connection to an Android tablet where a customized version of DJI's MSDK video decoding software sample is running, which uses the hardware decoding capabilities of the tablet. (The controller only has micro USB, and the tablet only has USB C, so we used a micro USB to USB A cable connecting to a USB A to USB C cable.) The

Figure 22: The DJI Spark quadcopter used in these experiments, with a banana for scale.

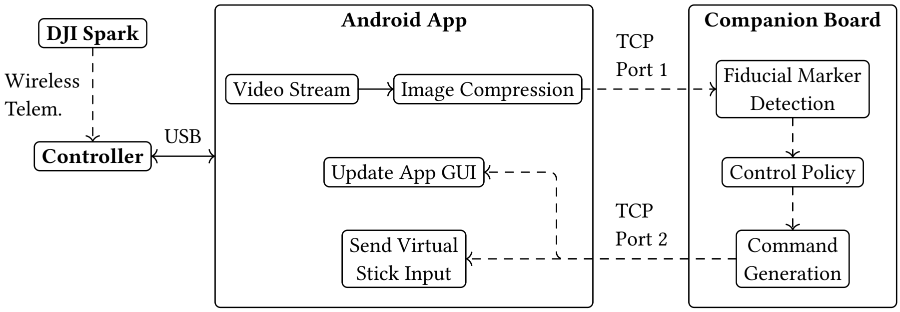


Figure 23: The data flow for the DJI Spark experimental setup.

app decodes video frames and sends them at a lower frame rate (7 Hz) to the companion board, which is connected over a WiFi network — the only other available interface with adequately high bandwidth.

- The companion board analyzes the video frames to detect the landing pad and determine its **position target** and **normalized pixel position** as laid out in Section 3.4.2. It then determines a command according to a control policy to be discussed later, and forwards the command back to the Android app. The commands can be *takeoff*, *land*, or a velocity target command.
- The Android app updates the GUI to give a visual representation of the command for debugging, and, if the command is a velocity target, converts it to a `VirtualStick` object, such that it can be used as if a pilot were moving the sticks on the controller. It then forwards the command to the controller.
- The controller then forwards the command to the drone.

Section 3.5.3: Methods

We conduct 20 landings for each of the 5 fiducial systems tested in Section 3.4.2.2, i.e., April Tag 24h10, April Tag 48h12, WhyCode Ellipse, WhyCode Multi, and WhyCode Orig. At the start of each landing attempt, the drone is placed 2.5 m from the landing pad, facing directly away from the landing pad so that it must search for it after takeoff. The landing pad is rotated 18° after each attempt to simulate approaches from 20 different angles. We consider a landing to be successful if it requires no human intervention except for initialization, and if the drone touches down fully on the landing pad without touching the ground. The entire process is automated, after initialization, from takeoff to landing, and we compare the distances from the drone's camera to the center of the marker after touchdown in order to rank the fiducial systems.

The companion board generates commands according to a control policy, which is a simple proportional controller with several phases. Its inputs are the **position targets** and **normalized pixel positions** of the landing pad, and its outputs are velocity targets for the drone. The `VirtualStick` commands for the drone simulate stick inputs from the controller and correspond to a target velocity, which we clip to the maximum interval $[-1, 1]$. The **position target** of the landing pad determines the translational velocities of the drone, i.e., forward/backward, left/right, and up/down, in order to direct the drone's approach toward the landing pad. The **normalized pixel position** determines the rotational velocity of the drone in the yaw dimension and of the gimbal tilt, in order to track the landing pad over time. The control policy is as follows:

- The system starts in the *takeoff* phase, during which the drone ascends in place to an altitude of 1.2 m. The gimbal position is unchanged.
- The drone then transitions to *search*, where it spins in place while tilting the camera up and down at a slow, constant rate until it locates the landing pad.
- It then transitions to *approach*, where it moves toward the landing pad quickly, without changing altitude. This phase has a "dead zone" where planar movement is disabled within a small distance from the landing pad to converge more quickly (instead of, e.g., oscillating like a typical proportional controller). The gimbal adjusts to keep the landing pad in the center of the camera frame.
- Next, it enters the *yaw align* phase, where it simply spins in place to align with the yaw of the landing pad. This phase also has a deadzone. The gimbal adjusts to keep the landing pad in the center of the camera frame.
- Once aligned with the landing pad, the system transitions to *descent*, where the drone decreases altitude while correcting its horizontal position above the marker. The gimbal adjusts to keep the landing pad in the center of the camera frame.

- At a minimum altitude, the system enters *landing commit* mode and sends a **land** command to the drone, which then automatically descends in place until detecting touchdown and disabling the motors. The gimbal points straight forward.
- After turning off the motors, the system enters the *landed* phase. The gimbal position is unchanged.

We test the landing system using each of the 5 fiducial systems, keeping all other factors the same. We compare the fiducial systems on the number of successful landings they produce, and on the accuracy of the landing, defined as the ability of the system to minimize the distance from the camera to the center of the landing pad after touchdown. All fiducial markers have the same width at the widest point.

Section 3.5.4: Results

April Tag 24h10, April Tag 48h12, WhyCode Ellipse, and WhyCode Orig all achieved 20 successful landings, while WhyCode Multi unfortunately achieved none. Figure 24 shows the distributions of landing error for each *successful* marker system; smaller distances mean more accurate landings. For one landing without discontinuities, Figure 25 shows the normalized pixel positions of the landing pad, Figure 26 shows the position target, i.e., the distance from the drone to the landing pad, and Figure 27 shows the control signals and phases for the landing. For a landing *with* discontinuities, Figure 28 shows the position target, and Figure 29 shows the control signals and phases.

Section 3.5.4.1: Fiducial system performance

April Tag 48h12 provides the most accurate landings — between 0 m and 1.5 m error (offset from center) —, while April Tag 24h10, WhyCode Ellipse, and WhyCode Orig tend to result in landings with about 0.1 m – 0.2 m error. April Tag 24h10 generates a lot of erroneous control signals, but its concentric marker structure allows it to track the landing pad at low altitudes (< 0.4 meters). WhyCode Ellipse and WhyCode Orig exhibit less orientation ambiguity but require higher *landing commit* heights, below which the rest of their landing is blind, which could be problematic in future, outdoor experiments with windy conditions.

Video 1 shows a demonstration with the Spark. The left half shows an external view of the drone and two landing pads. The right shows the view from the Android application, which includes the FPV feed from the drone and the control effort in each dimension. The drone takes off from one landing pad, searches for the other and approaches and lands on it, and then repeats the process to end up at the initial landing pad. The discontinuities are also visible, particularly in the second approach.

WhyCode Multi unfortunately did not demonstrate any successful landings, although it did perform successfully in the *search* phase and the initial part of the

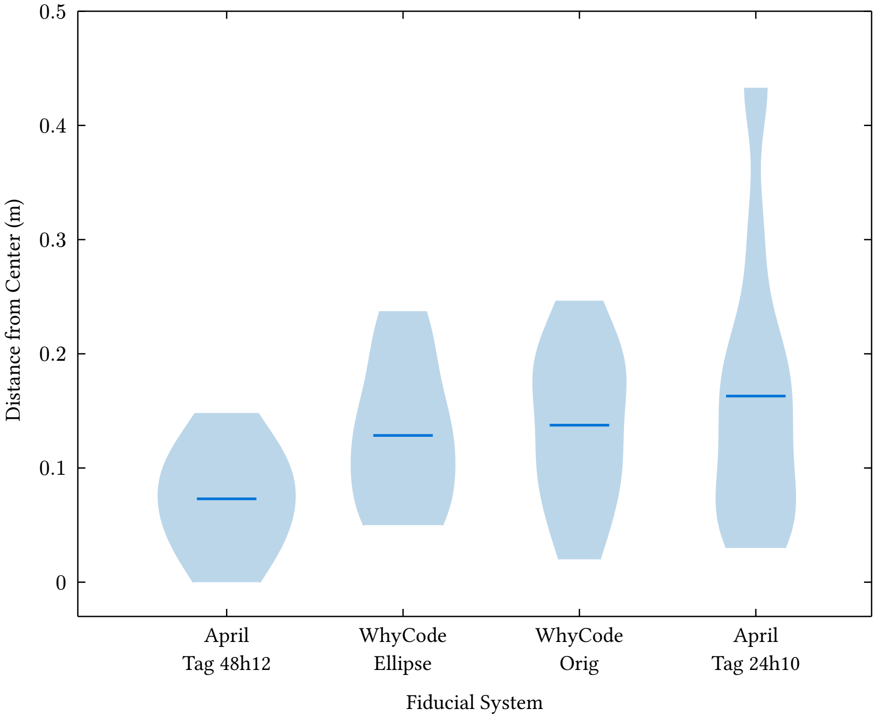


Figure 24: Distributions of landing error for each *successful* system. This is measured as the distance from the camera to the center of the landing pad after touchdown. Lower values mean more accurate landings. The systems are sorted from left to right by increasing mean distance from center. Means are indicated by horizontal lines.

approach phase. However, it overshot the landing pad every time. While it outputs correct position targets in lab experiments, its position targets are highly dependent on the angle from which the marker is viewed, and the camera for the Spark can only point between 0° (straight forward) and −85° down. It is possible that the drone could recover from an overshoot if the gimbal were able to point backwards at the marker. The prevalence of orientation ambiguity for each system correlates to the distance from the center of the landing pad to the drone's camera after landing, because of the resulting, erroneous control signals. Further, when the drone is almost directly above the landing pad, orientation ambiguity is most common and can cause the drone to move from one side of the landing pad to the other in stable oscillation — hence the deadzone.

All systems occasionally lost sight of the landing pad, contributing to a small number of landing failures. If this happened in a mission scenario, the system could simply begin searching for the landing pad again. Visual loss of the April Tags typically occurred upon initial acquisition of the landing pad, when they were harder to identify as a result of their small pixel area (because of the longer distance) and motion blur (because of the panning and tilting camera). After initial identification, however, they provide reliable detection even at very close distances because of their concentric marker embedding, and the drone can commit to the landing later,

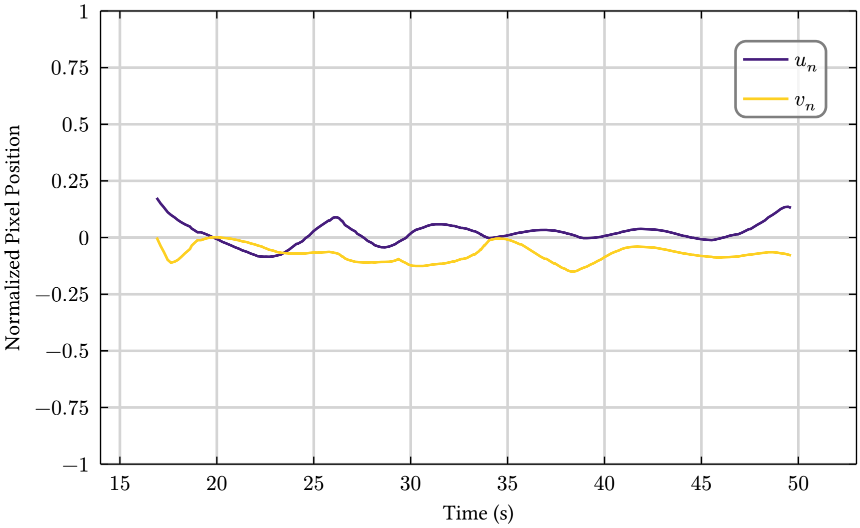


Figure 25: Normalized pixel positions of the landing pad during a representative example landing, corresponding to Figures 26 and 27. The drone is able to track the marker (i.e., $(u_n, v_n) \approx (0, 0)$) by aiming the camera via gimbal as the drone approaches the landing pad.

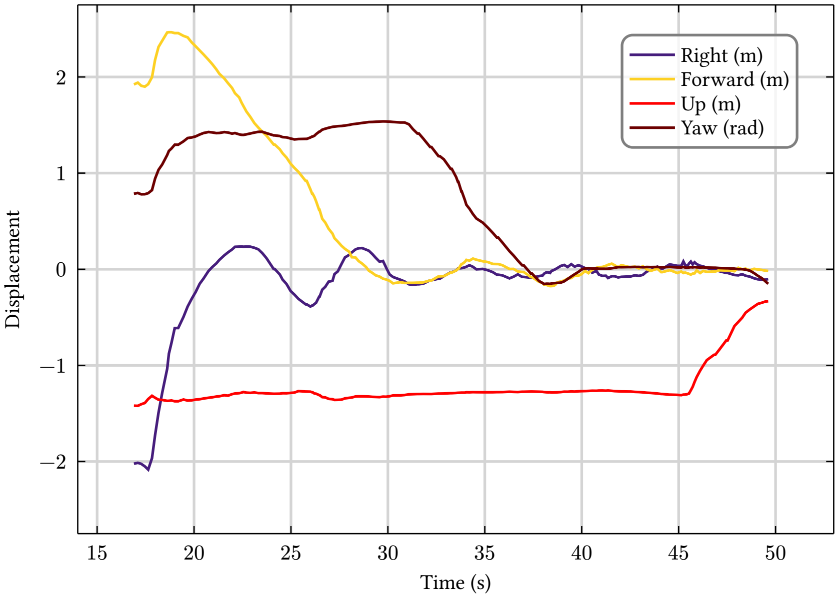


Figure 26: An example landing trajectory without discontinuities, corresponding to Figures 25 and 27. The Right position target decreases as the drone turns to face the landing pad. The Forward position target decreases slowly as the drone approaches the landing pad. The drone then rotates to align with the landing pad, reducing its yaw displacement. Finally, the Up position target approaches 0 as the drone descends.

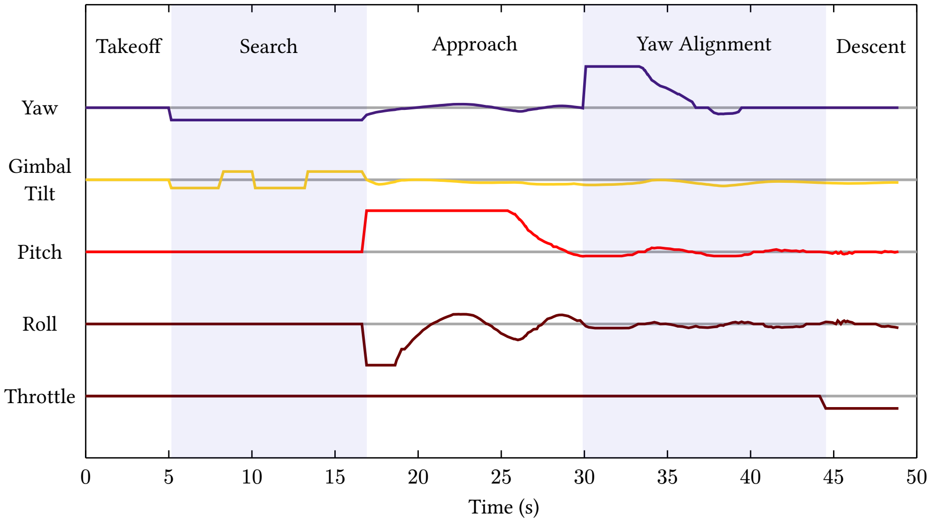


Figure 27: `VirtualStick` commands (yaw, gimbal tilt, pitch, roll, throttle) sent to the drone during an example landing, corresponding to Figures 25 and 26. These commands describe velocity targets in each dimension, i.e., a positive value for pitch means "move forward", and a negative value for throttle means "descend." *Takeoff*: control outputs are neutral. *Search*: the drone rotates counterclockwise in the yaw, and the gimbal tilts up and down. *Approach*: the drone tracks the landing pad with the yaw and gimbal tilt and approaches with pitch and roll. *Yaw align*: the drone tracks the landing pad with gimbal tilt, maintains its position above the landing pad, and aligns to the landing pad's yaw. *Descent*: the drone maintains its horizontal position and descends.

thereby relinquishing control at a safer altitude. On the other hand, the WhyCode markers exhibit reliable long-distance detection even under motion but eclipse the camera's field of view completely at close range. Finally, some visual loss occurs simply as a result of sun glares or shadows — an inherent problem in identifying markers in the visible spectrum.

Perhaps the most interesting result of this study is that the orientation ambiguity problem does not prohibit autonomous precision landing using a gimbal-mounted camera that is tracking the marker. Orientation ambiguity caused discontinuities in the control outputs, as predicted in Section 3.4 but they caused only minor disturbances — not destructive interference. Importantly, most of the discontinuities occur when the drone is almost directly above the landing pad, since the orientation is hardest to perceive when the camera is normal to the fiducial marker. In these cases, the drone's planar radius to the landing pad is small, so it is likely inside the deadzone, where the erroneous signals have no effect on the drone's behavior. Filtering or outlier detection, if applied to the detected relative position of the landing pad, may help remove these erroneous control outputs. However, this would ultimately treat only the *symptoms* of the problem, without addressing its root. One potential

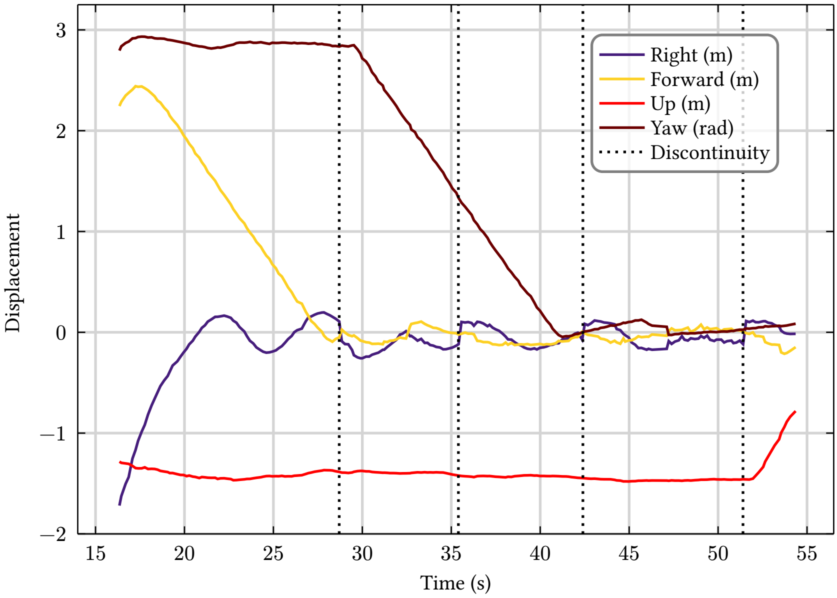


Figure 28: An example landing trajectory with discontinuities, corresponding to Figure 29. Discontinuities are shown by vertical lines at $t \in \{28.7, 35.4, 42.4, 51.4\}$ (s), and they propagate to the control signals shown in Figure 29.

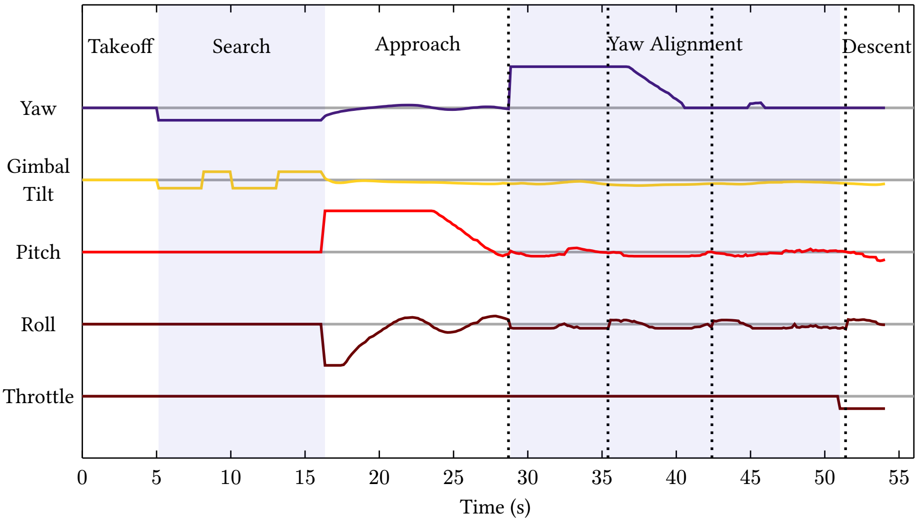
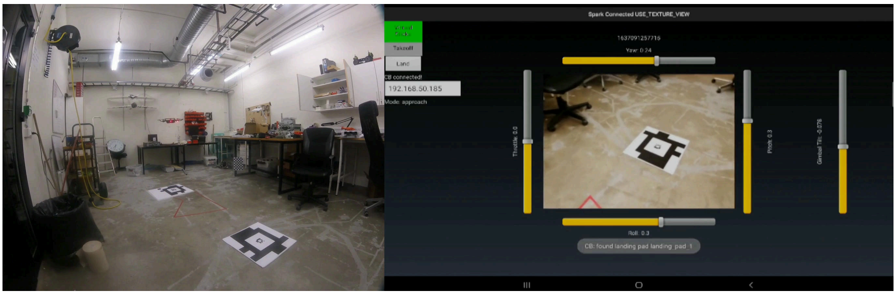


Figure 29: The `VirtualStick` commands during the landing with discontinuities from Figure 28. Discontinuities, shown by vertical lines, occur particularly in the roll commands and cause the drone to persistently oscillate around the landing pad.



Video 1: The Spark landing demonstration with two April Tag 24h10 fiducial markers as landing pads. Available at https://vimeo.com/j0shua/spark-fiducial-landing. The main point of the video is to demonstrate the effects of orientation ambiguity. The video shows two views side-by-side, with an external view of the drone and two landing pads on the left, and the view of the Android app controlling the drone along with its first-person video on the right. First, the drone takes off from its position on one of the landing pads, flies to a low altitude, finds the takeoff site with its camera, and re-aligns itself above its takeoff site after having drifted to the side. Then, it begins searching for the next landing pad by spinning in place and tilting its camera up and down. It finds the other landing pad, approaches it, and lands. It then repeats the process to arrive at the first landing pad again, exhibiting some erroneous control signal flips along the way, as a result of orientation ambiguity.

way of eliminating the problem (in future work, which we do not implement) would be to use the detected translational position of the marker only, and to determine its orientation indirectly, using the reliable orientation of the camera with an attached IMU and the assumption that the landing pad is flat on the ground.

Section 3.5.4.2: Drone platform difficulties

The DJI Mobile SDK connects to an app on a tablet which is useful for common tasks like generating a GUI, for displaying video, and for receiving user input. However, the tablet is not representative of the hardware that will eventually be embedded onboard a drone, and it is difficult to run the necessary fiducial software within the Android environment. So, the app must offload video frames to a Raspberry Pi 4 via a WiFi connection (no more wired interfaces were available). This transmission required converting the frames from color `.bmp` to grayscale `.webp` at 20% quality to achieve an adequately fast system with a maximum frame rate of about 7 Hz, and caused an inconsistent latency of between 0.5 and 2 seconds from image acquisition to control output. The low frame rate and inconsistent processing times left artifacts in the experiments, such as the oscillations in the roll signal of Figure 27. As the drone approaches the landing pad head-on, it overcorrects for left-right positional offset because it does not receive the video stream quickly enough.

Overall, this project aimed just to prove the landing method, and not create an ideal hardware platform. However, in future experiments, this latency can be solved by using a different drone platform with an onboard processing unit and using only wired connections, thereby avoiding image compression and transfer. In such a context, the control policy could be less conservative, with higher limits on the control outputs and more tightly tuned gimbal tracking.

Section 3.5.5: Conclusion

We have shown that autonomous precision landing is possible with fiducial markers and a gimbal-mounted camera for active tracking. Even with high processing latency, 4 of 5 tested systems have demonstrated success in the real world. While our platform does not allow for an onboard companion board, all essential image processing and control signal generation occurs in real time onboard a Raspberry Pi, meaning that the entire system can be embedded onto a larger drone. We have also demonstrated that the gimbal-mounted camera setup allows the drone to search for the landing pad by simply spinning in place and sweeping its gimbal up and down, thereby safely and efficiently scanning a large area.

This project gave a partial answer to **RQ1**, showing that even in spite of orientation ambiguity, it is possible to conduct successful autonomous landings, but that the orientation ambiguity will cause erroneous control signals. It gives further insight into **RQ3**, in that the method successfully ran on the Raspberry Pi, and it further motivates the notion of embedding the hardware onto the drone to avoid latency in the control system and its effects.

The next step is to re-conduct the experiments with the following changes: embed the companion board into a larger drone to reduce the system's latency and use the unambiguous orientation of a gimbal as measured by an IMU to work around the orientation ambiguity altogether. We see development of an embedded companion board in the next section (Section 3.6), and the corresponding real world experiment in Section 3.7.

Section 3.6: Matrice 350 and Raspberry Pi 4 payload

New funding from the Reykjavik University Infrastructure Fund allowed us to purchase a DJI Matrice 350 with an H20T camera, such that we were able to improve on the testing setup of Section 3.5. The Matrice 350 does not have an onboard computer, and existing computational payloads such as the Manifold 2 [107] have been discontinued and were expensive. We developed a cheap, onboard computational payload for the Matrice to use in subsequent autonomous landing experiments. At the time of this work, the Matrice 350 had just been released and there were not guidelines on how to connect to it with a Raspberry Pi in the way we lay out, although there were some guides for connecting to older versions of the Matrice, using different ports and older versions of the Raspberry Pi. These guides typically demonstrated simpler tasks but gave an insight into how to work with the system. It is, unfortunately, not trivial to migrate from older versions to newer ones, so this took some experimentation. We took advantage of the Matrice's payload capacity and other onboard infrastructure — as well as the PSDK — to integrate the computational system into the drone, thereby improving on our previous setup with the DJI Spark's computational system and its problems with inconsistent latency, unnecessary links in the data processing chain, and a heterogeneous system with both Android apps and ROS. However, this improvement comes with its own growing pains and development difficulties, which we also discuss. Developing payloads for DJI drones is non-trivial, and the payload for these experiments is the first chapter in a long saga. After using the payload in this section, we improved it and created another iteration, which we go over in Section 4.3.

Section 3.6.1: Motivation

Here, we are intentionally making a notable change from open source flight control software to DJI's closed system. In the earlier part of this research, we have built multiple operational drone platforms using components within this ecosystem, (e.g., PX4 [62], ArduPilot [61], the Navio2 [63] and Cube Orange [108]). As much as we prefer open source software and hardware, the distributed nature of the development of the software and the fact that the hardware and software are not built by the same organizations result in a product that is not as stable as we would like. These systems are easier — on the surface — to interact with programmatically, for example:

- streaming video from a custom camera is almost plug-and-play with the Herelink [109] telemetry system,
- adding custom data messages to the MAVlink [60] protocol (see Section 2.4) is basic functionality included in the protocol and can propagate through the autopilot system itself, as well as message-passing proxy servers and even GCS software for real-time telemetry,
- there are a number of velocity target and position target interfaces available for autonomous control — including through middleware such as ROS,
- there is plenty of free autopilot software, GCS software [110], [111], and middleware available.

However, the software is often prone to glitches, and the DIY nature of it tends to mean that the resulting drones are also prone to mishaps. Further, all of the overhead of making the drone platform stable becomes a task of the operator, who may not have the available resources, time, or indeed the skills to make an utterly reliable drone. In the case that one does have the ability to create a custom drone, there is typically a first step of building the drone and tuning its control system before it is capable of stable flight. Next, much of the damage to drones happens not in flight, but rather in storage and transportation, particularly as a result of most custom drones not having a form-fitting, rigid, protective case. Further, most such drones are not weatherproofed, such that they can suffer attrition from exposure to dust and moisture.

The DJI ecosystem, although closed or “black box”, solves many of these issues, but comes at a significantly higher price point. Such drones are stable out of the box, and constructed with holistic considerations of the software, hardware, and use cases. Basic drone construction and tuning is essentially delegated to a team of industry professionals. In the case of DJI’s flagship Matrice 350 that we are targetting (but also many others), many of the peripheral tasks are already expertly done, e.g.,

- the fuselage, motors, payloads, controller, and RTK tower are sealed and weatherproofed,
- the carrying case is form-fitting and can protect the drone even while it is bouncing around in a car driving off-road,
- the batteries have rigid cases and internal heaters for cold-weather operation, and they record performance statistics to automatically monitor degradation over time,
- payloads and batteries attach with quick-release mechanisms for easy installation instead of using, e.g., XT connectors.

Unfortunately, interacting with DJI drones programmatically involves getting used to a very specific and often unintuitive proprietary system whose documentation is often cryptic, unclear, or even wrong. This is discouraging for even the most diligent programmers, and it can be a prohibitively difficult barrier for people in domains other than computing or engineering. It is therefore understandable why many people decide to avoid DJI’s apparent black-box ecosystem altogether.

Our conclusion from our own experiences and those of our colleagues is, nevertheless, that the money saved from using custom drones and open source software is not currently worth the overhead in a scenario where the primary objective is something other than researching low-level drone operations. DJI offers streamlined, ready-to-go drone systems whose high initial cost is outweighed by their reliability and performance over time. Open source drones tend to have lower initial cost and a higher running cost, in terms of both materials and time. Therefore, given the difficulties in working with open source drones in Section 3.3 and the DJI Spark in Section 3.5, we chose to take the risk of developing computational payloads for the DJI Matrice instead, which was a difficult and major task lasting more than 1.5 years in parallel with much of the research in this thesis. It was worth it.

Section 3.6.2: Design requirements

Here, we discuss the development of the payload from software and hardware perspectives. These experiments required the following functionality from the payload:

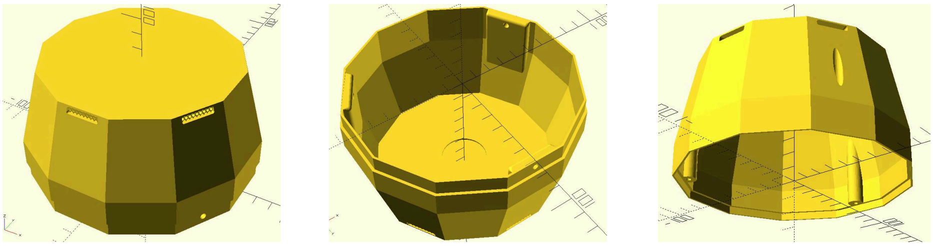
1. view the video stream from the camera in real time,
2. manage the camera source and zoom,
3. aim the camera via the gimbal,
4. control the drone via commands of takeoff, landing, and target velocity,

5. provide an interface for the drone operator to change settings,
6. integrate into the drone's power and telemetry system to avoid duplication,
7. protect against dust and rain, and
8. install quickly without tools.

Requirements 1-5 stem directly from the task at hand, and requirements 6-8 are aesthetic choices made from our previous experience operating drones in Iceland. Integrating into the drone's power and telemetry systems mean that the payload requires no additional battery or transmitters, which would greatly complicate the payload and make it heavier and more delicate. Protecting against dust and rain is necessary in Iceland because the weather is harsh and unpredictable, sometimes changing from sunny to hailing during the course of a single drone flight. Installation should be as simple and quick as possible (as with proprietary DJI payloads) so the operator can focus on operating the system instead of, e.g., struggling to attach a screw with freezing cold hands, or struggling to find a screw after dropping it on the ground.

Section 3.6.3: Case & computational hardware

We chose to use a Raspberry Pi 4 as our main computational board, given that our previous tests with it had shown that it can handle the necessary computation in real time (see Section 3.5), and given the vibrant Raspberry Pi community which meant that it was compatible with many different software suites. The capacity of the Matrice's power system, which contains two TB-65 batteries with 5880 mAh at 44.76 V [66], is $2 \cdot 5880\ \text{mAh} \cdot 44.76\text{V} \approx 526\ \text{Wh}$. The recommended power supply for the Raspberry Pi 4 is 5V at 3A, giving 15W [80]. If we assume the Raspberry Pi draws this maximum power for the entire duration of the drone flight (about 45 minutes), then the total energy consumption of the board as a portion of the total available capacity over a single flight is $\frac{15\text{W} \cdot 0.75\text{h}}{526\ \text{Wh}} \approx 2.1\%$. Therefore, the added power draw is insignificant. Further, the fact that it has hardware H264 decoding implied that the CPU would be free to execute only application-specific processes, instead of handling video codec overhead. The Matrice has an E-port on its topside mounting location, which is geared toward onboard computers that communicate with the drone using the PSDK, so this was our natural choice for an interface to the drone and for the placement of the payload. The E-port connector uses a proprietary protocol that is apparently coaxial cable in a USB C form factor. It is exposed via a development board as a UART for low-bandwidth communication and a USB C port for high-bandwidth communication, e.g., HD video. See Section 2.5.1 for more details on the drone and necessary peripherals. Although the E-port development board does provide power at a nominal 5V, we found over time that, on all of our units, it dropped significantly to 4.7V under load. This caused undervoltage warnings and CPU throttling on the Raspberry Pi, which made the processing pro-



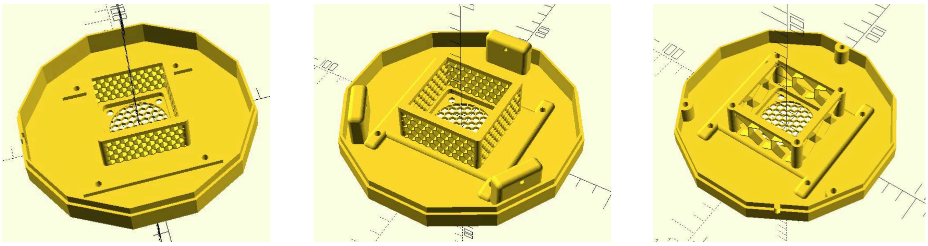
(a) Iteration 1 (top view).
(b) Iteration 1 (bottom view).
(c) Iteration 2.

Figure 30: The major iterations of the Matrice payload case canopy. Figure 30a and Figure 30b show the top and bottom views of the canopy with lateral mounting screw holes that join to the bottom in Figure 31b. After becoming unaligned by little more than a millimeter, these screws broke off the pillars in the bottom piece, leading to the vertical mounting screws in Figure 30c. All of these canopies have air intake holes positioned radially around the top and running to the center through ducts with honeycomb to attempt to reduce moisture intake.

hibitively slow. We opted instead to use an additional DC-DC converter to transform the E-port's 12V source to a more reliable 5V, 8A output, which solved this problem.

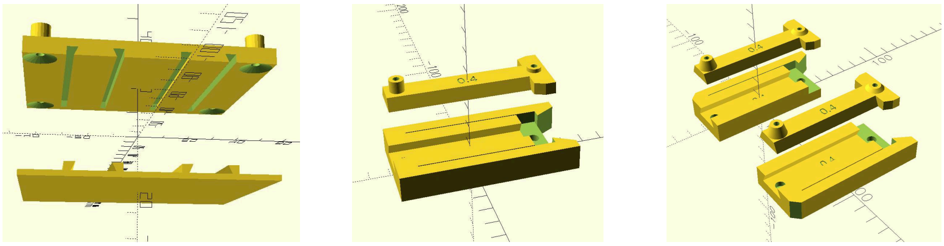
Construction of the physical payload case is tedious, slow, and nearly out of scope for research in computer science, but it plays a necessary, supportive role in real world testing. It is critical to ensure that the payload is structurally sound, weather-resistant, modular, and simple enough to use that it is nearly impossible to forget setup steps. Anticipating an iterative development process, we preferred to use 3D printing as a primary manufacturing method for the payload case. This made it relatively quick and cheap to create, refine, and recreate parts of the case that needed improvement. Although buying production metal components provides better final quality, it does not allow the necessary freedom in development, especially given the supply chain and shipping challenges of developing this in Iceland.

The general shape of the payload case imitates the main canopy of the Tarot 680 hexacopters from Section 3.3. It is a tapered dodecagonal prism cut into a top piece



(a) Iteration 1
(b) Iteration 2
(c) Iteration 3

Figure 31: The major iterations of the Matrice payload case bottom. From left to right: Figure 31a shows the initial bottom for test fitting to the Matrice and placement of the lid without fastening. The Raspberry Pi 4B and E-port development board attach to the top of the honeycomb structure with heavy tape, an exhaust fan mounts in the cavity, and there is a small notch for the E-port development board's connector to pass through to the Matrice. Figure 31b shows the bottom after revisions making the bottom thinner to decrease print time to about 4 hours and for fastening the lid using vertical pillars and horizontal screws that extend outward to the outside of the lid. These were weak and easily broke from lateral force because of the horizontal print layers. Figure 31c shows the final design of the payload bottom, which has stronger, vertical fasteners for the lid and a coarser honeycomb for the fan, which further decreases print time while adding some strength.



(a) Iteration 1. (b) Iteration 2. (c) Iteration 3.

Figure 32: The major iterations of the quick release brackets for connecting the payload case to the Matrice 350. Figure 32a shows the prototype, which is made from 2 flat plates with 4 tapered tracks that join after sliding them together. The screws attaching the bracket to the drone are countersunk for more clearance. This bracket did work, but suffered from binding as a result of warping when mounting to the drone, as the starboard and port screw pairs on the drone do not seem to be parallel, but rather angled slightly outward. Figure 32b shows a larger quick-release bracket with a slight built-in tolerance of 0.4 mm (hence the “0.4”), which eliminated the binding. It has tabs on both sides of the track that lock the bracket in place. One of these brackets was used on each side, but this made it exceedingly difficult to remove the payload because the operator must simultaneously press on 4 of the tabs simultaneously. Figure 32c shows the final version of the quick-release bracket, which primarily removes the locking tab on the inner side, such that the whole payload comes loose by unlocking the two outermost tabs.

(see Figure 30) and a bottom piece (see Figure 31) with a lip connecting them to protect against ingress. After several iterations, we added an exhaust hole in the bottom piece, an exhaust fan, and a cage around the fan to act as a mount for the Raspberry Pi, E-port development board, and DC-DC converter. Similarly, we designed quick-release brackets (Figures 32 and 33) to connect the payload to the drone without using screws in the field. These are formed in 2 pieces by interlocking V-track rails and a print-in-place elastic latch on one rail that locks onto a tab on the other rail. Slim brackets on the drone connect to the top, forward payload mount point and do

(a) Unlocked. (b) Pushing apart the elastic tabs. (c) Locked.

Figure 33: A demonstration of the Matrice payload quick-release mechanism (iteration 2). It is printed in 2 pieces with interlocking V-tracks. The inner piece has a backstop that pushes two print-in-place tabs apart as it is inserted, and they then hold it in place. The screws connecting the quick-release brackets to the payload and drone are accessible from the interface between the two brackets, such that they are prevented from backing out because of vibration as long as the pieces are joined.

(a) Payload bottom view. (b) Matrice top view. (c) Mounted payload.

Figure 34: The final Matrice payload case, printed in orange for visibility while flying. It has quick release brackets on its bottom (left), and the Matrice has rails on top (center) that the payload mounts onto (right) at the top, forward payload mount point.

not affect the drone's position in the snug carrying case. Screws connect the payload top, bottom, and quick release brackets together during assembly, but installation in the field does not require screws and can be done in a few seconds. Screw holes are printed into the payload design, and threaded, brass inserts are heated and pressed into them such that the printed PLA plastic locks them into place. Screws connecting the brackets to the payload case and to the drone are tightly covered up when the mechanism is locked, such that they cannot back out from the drone's vibrations. The final payload is shown in Figure 34. Although we did not conduct formal tests of the payload case's physical abilities, it successfully flew many missions in normal weather and several missions in a yellow weather warning with significant wind, rain, and hail, without coming loose or suffering water ingress. During our tests with the payload installed on the drone, it was possible to pick up the entire assembly via the payload and shake it with no components breaking or coming loose. We designed the components in OpenSCAD and they are freely available [112].

Section 3.6.4: Software

Many of the payload's software tasks are relatively straightforward, i.e., decoding a video stream, analyzing the frames, and sending velocity commands. On the other hand, interacting with the Matrice via the PSDK is more complicated because of its proprietary quirks and its often cryptic documentation and code samples. The PSDK is being updated quickly, so some specifics of the development of this payload will go out of date quickly. However, they give an insight into how to work DJI's black box systems.

PSDK applications are set up as `cmake` projects that can use the open source, medium-level code provided in the GitHub repository [70], as well as pre-compiled libraries that give access to the lower-level functionality in the drone, but whose source code is not available. A user writes high-level code on top of these libraries to interact with the drone. Some of the basic functionality is well-documented and includes code samples, but anything outside of the norm requires direct communication with DJI or weeks of poking around to determine which function or order of operations can inexplicably make something work. This payload is relatively simple, so we only face minor difficulties, but we will continue in Section 4.3 to develop a more sophisticated payload with more difficulties. When developing a PSDK app, we typically worked in a lab environment with the drone's propellers removed, or with the arms folded in the storage position, such that the drone could not take off even if the motors erroneously started spinning.

Configuration of a PSDK app is generally done with macros within specific header files, e.g., one can configure the basic universal asynchronous receiver-transmitter (UART) connection for low-bandwidth data transfer by defining in `dji_sdk_config.h`:

```
#define CONFIG_HARDWARE_CONNECTION DJI_USE_ONLY_UART
```

and by defining in `hal_uart.h`:

```
#define LINUX_UART_DEV1 "/dev/ttyACM0"
```

However, many of the macros do not seem to have an effect in many scenarios, (e.g., the `USER_BAUD_RATE` in `dji_sdk_app_info.h`) and may be specific to particular configurations — but this is not clear from the documentation [70]. It is not relevant to include all of the specific quirks here, but we mention some of them to give an idea of some of the basic challenges faced immediately when starting to develop a PSDK application.

Since the UART is a standard interface, it requires very little configuration on the side of the companion board's operating system. Once connected via UART, the companion board has access to control the drone's motion, aim the gimbal, subscribe to data topics from the flight controller, and create widgets for human interaction on the controller screen. However, it cannot view the drone's video stream, as this requires a high-bandwidth data connection, of which the PSDK provides 2 options: USB bulk devices and network devices. Given our particular setup with a Raspberry Pi and E-port development board, it was easier to set up a network device using the Raspberry Pi's OTG USB C port, which treats the E-port development board as a host and creates a `usb0` network. After minor configuration, i.e., installing some dependencies and activating the `dwc2` driver for OTG and `g_ether` for the `usb0` network, the Raspberry Pi is able to view the video stream from the drone.

The PSDK uses `ffmpeg` to decode the video stream from the drone, and it defaults to running the decoder on the CPU, which requires roughly a constant 70% usage on all cores, with latency that accumulates over time. To speed things up, we attempted to take advantage of the Raspberry Pi's `h264_v4l2m2m` hardware decoder, which performs well on the FPV camera built into the drone's body (see the large, wide-angle lens the center of the drone's front face in Figure 34c). However, in order to use and aim the camera in further experiments, we have to use gimbal-mounted payload cameras, such as our H20T, which has a significantly different video stream. In testing streams from our H20T and a borrowed P1 (a similar, gimbal-mounted payload containing a high-definition camera) the hardware decoder continuously gave ambiguous errors of the form

```
corrupt decoded frame in stream 0
```

showing that the decoder encountered an error in decoding the frame and threw out the result. Discussions with the Raspberry Pi developers[1] suggested (but do not confirm, because of a lack of documentation on the hardware decoder itself) that the hardware decoder is simply unable to decode gradual decoder refresh (GDR) streams, such as those from the H20T. However, further discussions revealed a firmware parameter `MMAL_PARAMETER_VIDEO_DECODE_ERROR_CONCEALMENT` which is `true` by default and causes the decoder to not pass on frames if it has encountered errors in decoding them. Figure 35 shows the results of changing this parameter to `false` and recompiling the Raspberry Pi kernel. We could successfully use the hardware decoder to decode the 1920x1080 stream from the P1 (see Figure 35a) in real time and view it with minimal CPU overhead. However, our payload, the H20T, still showed significant issues because of its resolution of 1920x1440, as the hardware decoder apparently has a maximum pixel height of 1080. (The resolution of all streams was 1920x1440, even that of the IR camera, which outputs at a true resolution of 640x512, meaning that it was upsized before transmission.) The result is that the bottom 360 rows of pixels were not successfully decoded (see Figure 35b), and that we gave up on trying to accomplish this. Ultimately, we had to compromise by decoding every frame of the 30 FPS on the CPU, which still required significant CPU time, but then exported at about 17 FPS to RGB frames and immediately resized them to 800x600 pixels before further processing. This left about 30% of CPU time for executing task-oriented code, i.e., landing pad detection and flight control. The code both for setup and for the application itself is available on GitHub [113]. We accessed this payload with a laptop over SSH using a battery-powered WiFi router in the field, which is not ideal, but which was the easiest interface we had available.

For conservative testing, we preferred to enable a geofence around the drone, restricting its possible flight to a radius of 15 m from the takeoff point, but this prevented the drone from responding to all autonomous commands because it was apparently too close to the edge of its allowed airspace. After increasing the radius, the drone began to respond to commands from the companion board.

---

[1] https://github.com/raspberrypi/linux/issues/5166

(a) P1.

(b) H20T.

Figure 35: Results of decoding the video streams from the P1 (successful) and H20T (unsuccessful) in hardware. The P1 stream successfully decoded after disabling concealment of errors in firmware, but the H20T's resolution of 1920x1440 was larger than the decoder's apparent max of 1920x1080, and the bottom pixels were corrupted.

Section 3.6.5: Conclusion

We successfully created a computational payload for the Matrice 350 that can access the drone's camera and control its behavior. This project did not answer any research questions, but serves as valuable groundwork for exploring **RQ3** in the next section, particularly with regards to being able to execute an autonomous landing solution onboard the drone to reduce latency as in the previous section.

Section 3.7: Avoiding orientation ambiguity

After developing the custom computational payload for our Matrice 350 in the previous section, we set out to conduct a similar experiment to that of Section 3.5. Our primary goal was to set the system up to flexibly use fiducial markers to mark the landing pad, while entirely avoiding the orientation ambiguity issue. We focus on minimizing data requirements as compared to previous work to make our method as general as possible. Other projects tend to, understandably, use enough data to estimate the physical pose of the marker relative to the drone, whether this is by measuring

1. the drone's AGL as a heuristic for its height above the marker,
2. the straight-line distance (range) from the drone to the marker,
3. or the 6 DoF marker pose.

We prefer not to use the AGL as it only corresponds to the drone's height above the landing pad in the case of flat ground, and otherwise can confound the relative position estimate. We similarly prefer not to use the range because it requires a rangefinder which is not typically available on most drones, so it restricts the method's generalizability. As we have seen in Sections 3.4 and 3.5, using the 6 DoF marker pose will result in erroneous control signals. Therefore, we simply do not attempt to estimate the physical distance to the landing pad directly, but instead depend only the direction from the drone to the landing pad. This means that the drone often must halt its descent, align to the landing pad horizontally, and then begin its descent again. However, it also means that the method requires little data and therefore is adaptable to many different scenarios. We exploit this adaptability to explore the advantages of using the multi-sensor H20T, which has a basic wide-angle RGB camera, a zoom RGB camera, and an IR camera, which allow the drone to identify the landing pad from long distances and at night. For our real world tests, we use the payload developed in Section 3.6.

This section presents the findings of the following pre-print:

> [42] J. Springer, G. Þ. Guðmundsson, and M. Kyas, “A Precision Drone Landing System using Visual and IR Fiducial Markers and a Multi-Payload Camera,” 2024. [Online]. Available: https://arxiv.org/abs/2403.03806

Section 3.7.1: Related work

The most basic paradigm of autonomous landing with a camera and a marked landing pad assumes the camera has a downward-facing orientation, either rigidly mounted to the drone or stabilized vertically down on a gimbal. One study uses a camera fixed to a helicopter to locate a custom visual marker on a stationary platform, approaching from a distance of 4 m and achieving an error of 0.7 m [114]. Another study uses a downward facing camera to locate an ArUco marker with an embedded IR beacon and measure the drone’s height using a barometer and rangefinder. This system achieves a mean error of 0.15 m when using both the visual and IR systems, and an error of 0.25 m when using only one system [115]. Another uses an IR beacon on the landing pad and LiDAR to determine the drone’s height, achieving successful landing on a 0.4 m by 0.4 m platform [2]. Similarly, one study uses an IR beacon mounted on a landing pad that is heaving to mimic a ship’s motion and that is detected using a downward-facing IR camera [3]. Instead of using a marker or beacon, a further study proposes to record the appearance of the takeoff location and realign the drone with it using visual matching after executing a mission [21]. One study uses a fixed, forward-facing camera for initial approach, and a fixed, downward-facing camera for final descent. This system collaborates with the moving landing pad (marked with an AR Tag) to share Global Positioning System (GPS) position, velocity, and acceleration data (in simulation) [116]. A similar study uses a custom marker, one downward-facing camera, and one forward-facing camera to land on a 1.5 m by 1.5 m platform marked with a custom visual marker [4]. While this paradigm is employed successfully, it suffers from the basic weakness that it is difficult to detect and maintain detection of the marker if the camera is fixed or only pointing downward because the camera moves according to the movements of the drone. Additionally, the camera’s limited field of view makes it difficult to find the marker if it cannot move independently.

Another paradigm of autonomous landing uses a gimbal-mounted camera that is actuated to track the landing pad independently of the drone's orientation, which increases the robustness of the landing pad detection but complicates the landing process. Typically, they estimate the full pose of the marker, requiring the camera intrinsic parameters and marker dimensions. They also tend to depend on altitude as measured by an extra sensor, or, e.g., on the relative GPS positions between the drone and the landing pad. One study presents a hybrid method that uses a collaborative GPS system (both on the moving landing pad and the drone) for long-range approach, a gimbal-mounted camera for initial visual approach, and a downward-facing, wide-angle camera for final descent [5]. Another study uses a gimbal-mounted camera to detect the 6 DoF pose of an AR Tag on a moving ship deck, initially approaching using the GPS data of both the ship and the drone, and achieving an error of 0.2 m [14]. A further study uses a gimbal-mounted camera to track a landing pad with a custom visual marker and approaches it using the gimbal angles and height as measured by an altitude sensor [15]. Similarly, it is possible to use a gimbal-mounted camera to detect a custom visual marker and approach using the camera's orientation, the marker's pose, and the drone's AGL, measured by an auxiliary sensor [16]. Finally, a similar study uses an omnidirectional IR beacon mounted on the landing pad and approach it using the gimbal-mounted camera's attitude and the drone's height as measured from a LiDAR [117]. They can detect and approach from a distance of 18 m. A proprietary extension of AR Tag called Lentimark [12] makes the full 6 DoF marker pose reliable, though it is normally subject to orientation ambiguity, and this makes it possible to land simply using the marker pose with a gimbal-mounted camera [13]. Each study presents the performance of its method in terms of landing accuracy differently, so it is not straightforward to compare them. In some cases they present only a single data point showing that they achieved an accuracy of a few centimeters. We list the landing accuracy where we can, such that we have at least some metric with which to compare.

Some studies have created fiducial markers in the IR spectrum. Dogan et al. have used 3D-printed ArUco markers and QR codes to unobtrusively mark objects for augmented reality [118]. Khattack et al. have used laser-cut, acrylic ArUco markers for drone localization in visually degraded environments [119]. Claro et al. created a multimodal, active (i.e., powered) thermal marker based on ARTag for autonomous landing of a drone with GNSS, RTK, LiDAR and visual and thermal cameras, achieving an error of 0.03 m and using a downward-facing camera on a stabilized gimbal [17]. The marker is composed of two materials of differing reflectivities, with one heated and one unheated.

Most of the related work depends on the AGL or range (straight-line distance to the landing pad) which must be measured by auxiliary sensors, e.g., LiDAR or ultrasonic sensors. AGL may not correspond directly to the altitude above the landing pad if the terrain is not flat, such that the system may miscalculate the position of the drone relative to the landing pad. The method described in [15] is similar to what we propose, but we improve upon it by not requiring the height, by adapting it to multiple cameras, and by using IR fiducial markers. Additionally the IR beacons are not necessarily distinguishable from each other — often characterized simply by areas of significant IR radiation, such that any IR emission might appear as a landing pad [2], [3]. This issue is partially addressed in [17], with an actively heated marker and a downward-facing camera, but we believe there is also potential in using actuated cameras and passive (i.e., unpowered) IR infrastructure that exploits high reflectivity. Finally, most of the related work does not focus on the initial search for the landing pad or on detection re-acquisition if it is lost. Instead, it treats autonomous landing as a low-level control problem, detailing methods for sensor fusion and filtering to develop reliable pose estimation and trajectory calculation. However, many modern drones make this unnecessary by providing *position* or *velocity* modes that execute high-level commands, e.g., *forward x m/s*, using either GPS or optical flow for velocity measurement. Furthermore, many of the methods depend on expensive, sophisticated, or collaborative ground infrastructure, whereas we minimize the required data and sensors.

Section 3.7.2: Methods

Section 3.7.2.1: Required data

We attempt to minimize the required data wherever possible, thereby minimizing constraints on sensors and infrastructure as compared to previous work. This method uses April Tag [89], [95], [96] fiducial markers to denote and recognize the landing pad, but it does not depend on pose estimation as in Section 3.4 and Section 3.5. Further, it does not depend on the AGL or range as in most of the related work that uses an actuated camera. Instead, it identifies the marker in the camera image and determines the *pan* angle φ and *tilt* angle θ from the drone's heading to the landing pad as an abstraction from the pose. Avoiding estimation of the landing pad's 6 DoF pose removes the possibility for orientation ambiguity to affect the process at all and reduces data requirements, so that the system can be more dynamic with the sizes and spectra of landing pads, the camera sensor we use at any given time, and the zoom factor. We consider also the difference ψ between the drone's yaw and the landing pad's yaw for pre-descent alignment to the landing pad, the zoom factor Z (queried from the flight controller) for intermediate calculations such as the camera's field of view at a given time, and the landing pad's pixel size S_p for specific control state transitions and zoom management. Figure 36 illustrates the required angles. Although it would be convenient, we avoid using AGL and the straight-line range R from the drone to the landing pad because these are often not available on most drone platforms.

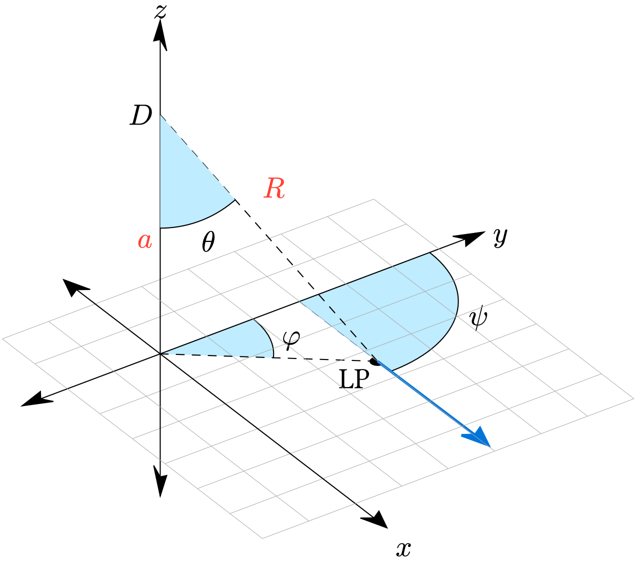


Figure 36: The angles defining the control policy for the drone (D) to approach the landing pad (LP). φ represents the pan, and θ the tilt, from the drone to the landing pad. The heading of the drone is in the direction of the y-axis. The landing pad is at the base of the vector (LP), and the vector extends in the direction of the landing pad's yaw. The angle ψ represents the relative yaw from the drone's heading to the landing pad's heading. The altitude a and range R are not used or needed for this method.

We determine φ and θ by tracking the landing pad with a gimbal-mounted camera and comparing its orientation to that of the drone. We query the pan angle of the gimbal, φ_g, and the tilt angle of the gimbal, θ_g, directly from the flight controller. Since the gimbal is most often not aimed directly at the landing pad, but skewed by some angle as a result of the drone's motion, we also consider the landing pad's pixel position (u, v) and the additional angle offsets (φ_u, θ_v) implied by the fact that it is off-center in the camera frame. Then we calculate φ and θ as in Equations 12 and 13.

$$\varphi = \varphi_g + \varphi_u \tag{12}$$

$$\theta = \theta_g + \theta_v \tag{13}$$

We calculate φ_u and θ_v by considering the field of view of the camera in each dimension, as illustrated in Figure 37, where FOV_u and FOV_v represent the fields of view of the camera in the horizontal and vertical dimensions respectively. Then, we normalize the pixel positions of the landing pad's center into the interval $[-0.5, 0.5]$ and multiply by the field of view in the relevant dimension, as in Equations 14 and 15 where u_v and v_c represent the horizontal and vertical pixel centers respectively.

$$\varphi_u = \frac{u - u_c}{2u_c} \cdot \mathrm{FOV}_u \tag{14}$$

$$\theta_v = \frac{v - v_c}{2v_c} \cdot \mathrm{FOV}_v \tag{15}$$

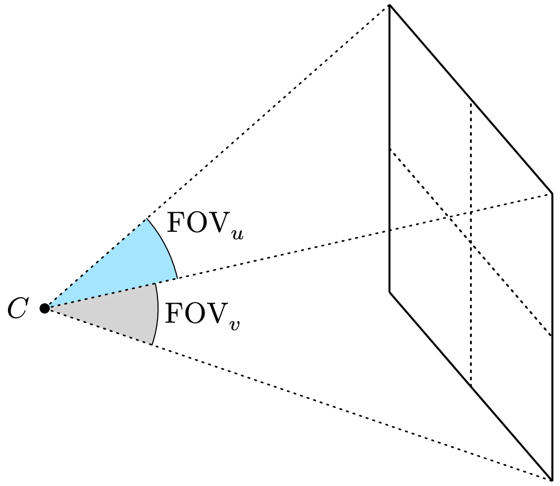


Figure 37: Horizontal and vertical fields of view. If the camera is at position C, then its field of view is the angle to the edges of the scene that the camera can view. In many cameras, the horizontal and vertical fields of view, FOV_u and FOV_v are not equal, so we measure both.

The fields of view for the wide-angle and IR cameras are constant, but the field of view for the zoom camera needs to be calculated dynamically, based on the current zoom factor which is available from the flight controller. We calculate the fields of view as in Equations 16 and 17, where w and h are the camera's sensor width and height respectively, and f is its focal length. For the wide-angle and IR cameras, f is constant, and for the zoom camera, $f = ZF_b$ where F_b is the base focal length. We query Z from the flight controller.

$$\mathrm{FOV}_u = 2\arctan\left(\frac{w}{2f}\right) \tag{16}$$

$$\mathrm{FOV}_v = 2\arctan\left(\frac{h}{2f}\right) \tag{17}$$

Calculating the fields of view for each camera — and calculating the field of view for the zoom camera *dynamically* — makes it possible to switch between the camera sensors at will, enabling the drone to, e.g., identify the landing pad from far away with a high zoom factor, and then track it during its full approach while zooming out to keep it fully within the frame.

Section 3.7.2.2: Control policy

The autonomous landing system follows the control policy outlined in Figure 38. We assume the drone starts the landing process near — but not directly above — the landing pad. It starts to look for the marker by holding the camera still, facing forward and down (*static search*). It then proceeds to search in a slow, clockwise yaw rotation, tilting the camera down (*search down*) and up (*search up*). If the system detects the landing pad in any of the search states, it centers the landing pad in the camera's field of view (*aim camera*), and then points the drone at the landing pad in the yaw dimension (*aim drone*). At this point, the drone can *approach* simply by moving forward, making slight left or right adjustments, and tracking the landing pad with the camera. Once the drone is above the landing pad, it stops actively tracking the marker (but continues to zoom in or out to keep the landing pad in the camera frame), points the gimbal vertically down, and aligns to the landing pad's yaw (*yaw align*). It then aligns horizontally (*horizontal alignment*) and begins its *descent*. Once the zoom factor is adequately low and the pixel size of the landing pad is adequately high, the drone *commits* to the landing, descending until hitting the ground and disabling the motors. It then transitions to the *landed* phase. If the landing pad is no longer detected at any point during *aim camera*, *aim drone*, *approach*, *yaw alignment*, or *horizontal alignment*, the system zooms out (*zoom out*$_1$), to reacquire the landing pad. If it succeeds, it switches back to the previous state and resumes the landing process; otherwise, it restarts from *static search*. If the landing pad is no longer detected during the *descent* stage, the drone recovers by zooming

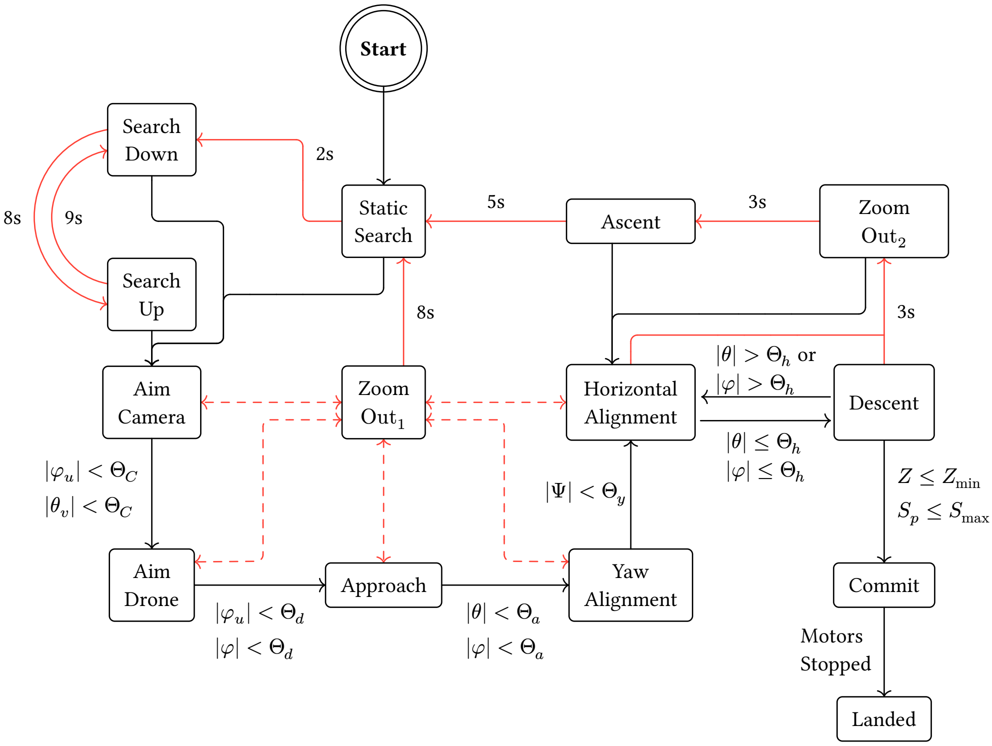


Figure 38: The control policy dictating the behavior of the drone during the landing process. For *Aim Camera*, *Aim Drone*, and *Approach*, transition conditions are based on $\Theta_c = \Theta_d = \Theta_a = 3°$. For *Yaw Alignment*, $\Theta_y = 1°$. For *Horizontal Alignment*, $\Theta_h = 2°$. For *Descent*, $Z_{\min} = 2$, and $S_{\max} = 32\%$. For *Commit*, the flight controller indicates motor stop. Black edges indicate transitions resulting from continuous detection and fulfillment of transition requirements. Solid red edges indicate transitions resulting from detection loss. Dashed red edges indicate transitions resulting from detection loss that can be reversed if the marker is detected soon enough (< 3s).

Mode	Forward (m/s)	Right (m/s)	Up (m/s)	Yaw ($\frac{\circ}{s}$)	Pan	Tilt	Zoom
Static Search	0	0	0	0	$0\frac{\circ}{s}$	$0\frac{\circ}{s}$	none
Search Down	0	0	0.0	5	$0\frac{\circ}{s}$	$-10\frac{\circ}{s}$	none
Search Up	0	0	0.0	5	$0\frac{\circ}{s}$	$10\frac{\circ}{s}$	none
Aim Camera	0	0	0.0	0	$1.2\theta_u{}^\circ$	$-1.2\theta_v{}^\circ$	auto
Aim Drone	0	0	0.0	φ	$5\theta_u\frac{\circ}{s}$	$5\theta_v,\frac{\circ}{s}$	auto
Approach	$2.82\sin(\theta)$	$1.41\sin(\varphi)$	0.0	0	$5\theta_u\frac{\circ}{s}$	$5\theta_v\frac{\circ}{s}$	auto
Yaw Alignment	0	0	0.0	$-\psi$	0°	0°	auto
Horizontal Alignment	$-0.10\theta_v$	$0.10\theta_u$	0.0	0	0°	0°	auto
Descent	$-0.05\theta_v$	$0.05\theta_u$	-0.5	0	0°	0°	auto
Commit	0	0	-0.5	0	0°	0°	none
Landed							
Zoom Out	0	0	0.0	0	$0\frac{\circ}{s}$	$0\frac{\circ}{s}$	out
Ascent	0	0	0.5	0	$0\frac{\circ}{s}$	$0\frac{\circ}{s}$	none

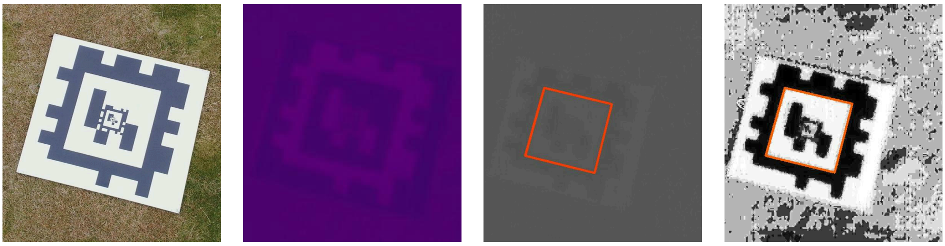
Table 4: Control signal calculation for each of the phases of the Matrice's control policy.

out (*zoom out*$_2$) and, if necessary, ascending (*ascent*), and returning to *horizontal alignment* if it reacquires it; otherwise, it restarts from *static search*.

Each state of the control policy has parameters to specify velocity targets for the drone in the forward, up, and right dimensions, clockwise yaw rate, and the target pan and tilt speed (or angle) of rotation of the camera, as well as the mode's zoom policy (see Table 4). The pan and tilt are controlled in terms of rotational speed in modes where the landing pad is being actively tracked, and in terms of angles when the camera should be pointed straight ahead, straight down, or upon initial recognition of the marker (when it is advantageous to immediately aim the camera directly at the marker without constraining pixel speed). The control signals are constrained before execution as follows: the forward command is in $[-0.5, 2.0]$ m/s, the right command is in $[-1.0, -1.0]$ m/s, the up command is in $[-0.5, 1.0]$ m/s, and the yaw rate command is in $[-10, 10]\frac{\circ}{s}$. For the zoom modes, the mode *none* means the zoom factor does not change, the mode *out* means the zoom factor decreases slowly, and the mode *auto* means the system minimally adjusts the zoom factor to keep the pixel size of the landing pad between 20% and 80% of the total video stream pixel size. Pixel sizes out of this range make the detection unreliable: if the pixel size is too small, the system may not detect it; if it is too large, the drone's movements can push the marker out of the camera's field of view. We minimally adjust the zoom factor because zooming increases pixel speed and blur, making the detection less reliable. Changing the zoom factor over the lower-bound of the zoom camera triggers a stream change to the wide-angle camera or vice versa. Zoom policies do not apply when using the IR camera, as it has no optical zoom.

Section 3.7.2.3: Landing pads

We conduct tests with physical landing pads in the real world. Initial experimentation with a large, visual landing pad showed that it exhibited an adequate temperature gradient to be detected even in IR images, as shown in Figure 39. Therefore, we test both visual and IR markers, since our testing platform allows for it. We test 3 landing pads, as shown in Figure 40. The first is a large April Tag 48h12 arrangement with IDs 2, 1, and 0 in order of decreasing size. It is printed on vinyl, and the largest marker is 1 m across. The IR landing pads are made of steel and aluminum in the shape of an April Tag 36h11 with ID 564 because its lack of islands and thin corners meant it could be cut out of a single sheet of metal. We cut the IR landing pads on a plasma cutter out of 3 mm sheet metal. The active IR landing pad, shown in the bottom right of the arrangement of 3 landing pads, is made of steel and heated with a number of Peltier plates on the back. These point sources of heat can be seen through the metal and result in uneven heating, however this did not inhibit the detector. Originally, we chose steel for its lower reflectivity and higher emissivity as compared to aluminum, but initial experiments showed that the reflectivity was too high, and the emissivity too low, for the marker to appear clearly in the IR spectrum. We proceeded by covering it in electrical tape, so that its appearance in the IR camera view comes almost entirely from the warm electrical tape, with the steel body acting simply as a heat conductor. This landing pad is designed to appear warm where the April Tag would be white, and should be placed on a relatively cool surface for contrast. The passive IR landing pad, shown in the bottom left of the 3-marker arrangement, is cut from aluminum for its high reflectivity. This landing pad depends on a differential between the IR radiation emitted from the sky and



(a) Visible spectrum.

(b) IR spectrum.

(c) Inverted grayscale, with detection.

(d) Detection with histogram equalization.

Figure 39: Inspiration for creating landing pads in the IR spectrum. A small temperature differential forms over the visual April Tag marker under sunlight, since the black ink absorbs more energy from the sunlight than the white ink. However, the image is not clear enough in the IR spectrum, and offers no benefit over the visible spectrum for this landing pad. We therefore made specific IR landing pads as well, as shown in Figure 40. From left to right: Figure 39a shows the visual landing pad in isolation, Figure 39b shows the same scene in the IR spectrum, Figure 39c shows the IR image inverted with a successful April Tag detection overlaid, and Figure 39d shows the detection with histogram equalization for clarity. The thermal difference is only visible when the landing pad is viewed in isolation, as the camera adjusts the dynamic range for the current scene; it is no longer visible near larger IR differences, e.g., in Figure 40.

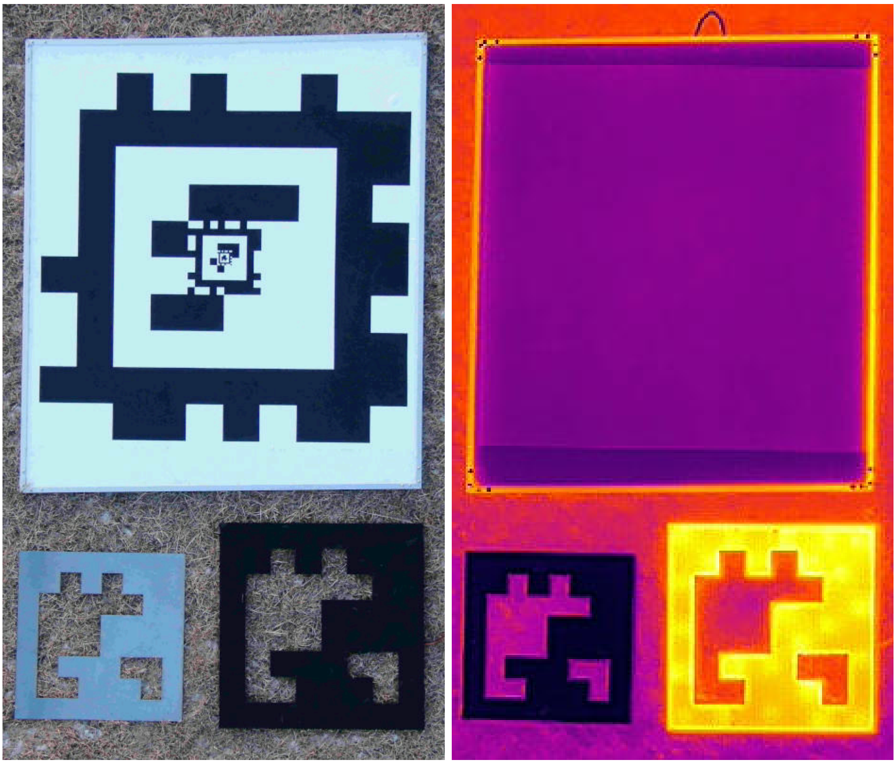


Figure 40: The final landing pads for this experiment in the visible spectrum (left) and IR spectrum (right). The top landing pad is made of vinyl, is in the visible spectrum and is used for long- and close-range detection in the daytime. The bottom right is a heated landing pad made of steel and covered in electrical tape to increase its emissivity. The bottom left is made of aluminum and is unheated.

from the ground. In this case, the sky emits significantly less IR radiation than the ground, so the marker appears very "cold" in comparison, even though it is at a similar temperature. Since it has the negated appearance of the April Tag marker, we invert the image before passing it to the April Tag detector.

Section 3.7.3: Results

Like much of the related work, we would like to have a metric for the quality of the autonomous landings that our system carries out. Therefore, we define the following protocol, which is similar to that proposed by Jiang et al. [15]: we start each test by manually positioning the drone at a horizontal distance between 5 m and 168 m from the landing pad, and altitude of 5 m to 102 m above the landing pad. We then give control of the drone to the computational payload, which directs the landing according to the policy in Figure 38. while we simply monitor its behavior. The error of the landing is measured as the distance from the center of the landing pad to the point directly under the drone's camera after touchdown.

We conducted 26 tests over 4 days with air temperatures ranging from −8° C to 3° C, wind conditions less than 5 m/s, on a grassy field in Iceland with light snow. Table 5

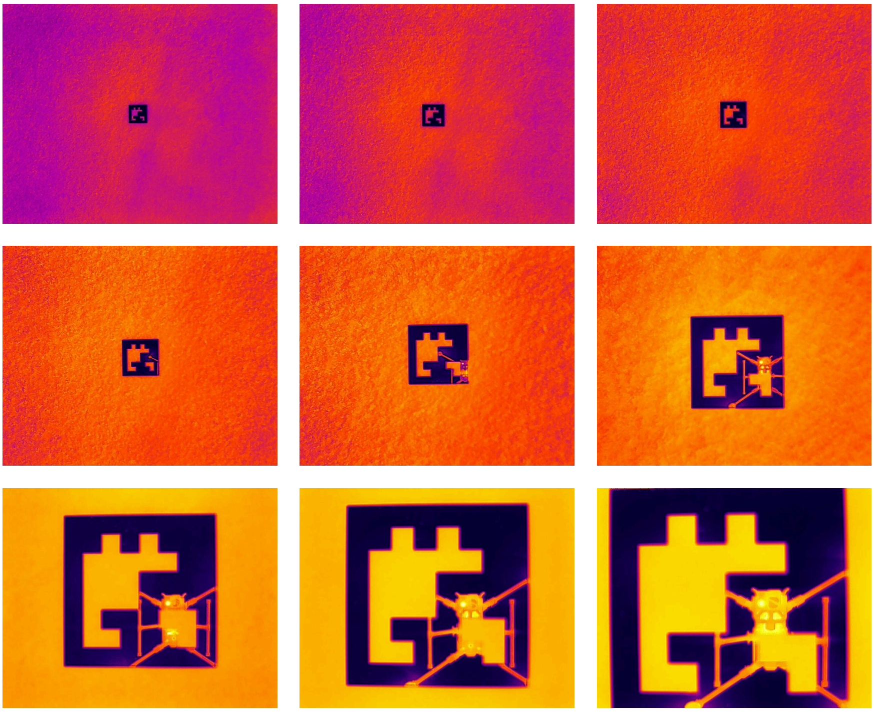
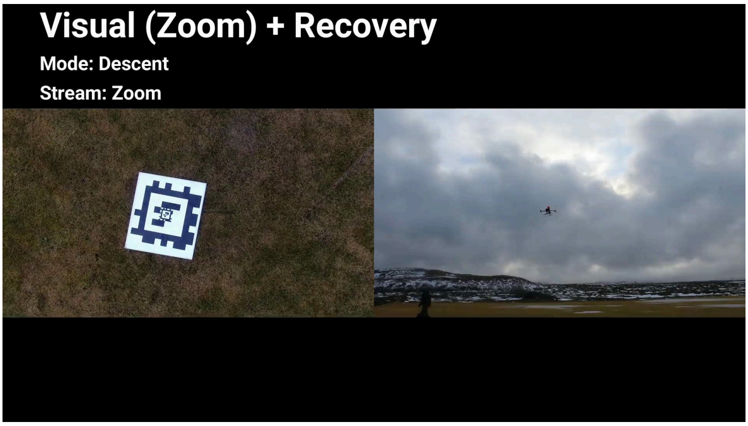


Figure 41: Example of reflectivity in the passive IR landing pad. It becomes obstructed by the reflection of the drone's own IR radiation as it approaches. The consequence is that the system can use the default April Tag method for approach and initial descent but must commit to the landing before obstruction. The images are artificially colorized; brighter areas represent areas that are perceived to be hotter, and darker areas represent areas perceived to be colder. In reality, the aluminum landing pad is at ambient temperature, but reflecting the IR radiation (or lack thereof) from the cold sky, giving a contrast between it and the comparatively warmer, higher emissivity ground. The images are taken at night during winter, i.e., at between $-5°$ C and $-8°$ C ambient temperature on a grassy field.

illustrates this, where μ_E is the mean error, σ_E is the standard deviation of the error, n is the number of tests per type, and Max Alt. and Max Dist. are the maximum starting altitudes and horizontal distances for the landing tests. All tests were successful, i.e., the drone landed at least touching the landing pad, with a mean error of 0.19 m, which is similar to the accuracy achieved by Jiang et al. [15], although Claro et al. [17] achieve lower landing error at about 0.02 m. This demonstrates the viability of the method and each of the landing pad types. Video 2 shows landings with each of the landing pads and an antagonistic landing site scenario at the end.

Type	μ_E (m)	σ_E (m)	n	Max Alt. (m)	Max Dist. (m)
Visual	0.16	0.10	11	102	168
Active IR	0.14	0.14	5	15	15
Passive IR	0.26	0.17	10	15	14

Table 5: Landing accuracy for the Matrice 350 per fiducial system. No system gives a significant advantage over the other in terms of accuracy, but the zoom does, unsurprisingly, allow the visual system to work from significantly longer distances than the IR.



Video 2: The Matrice landing demonstration with visual and IR fiducial markers. Available at https://vimeo.com/j0shua/matrice-fiducial-demo-visir. The main point of this video is to demonstrate how to eliminate orientation ambiguity while still using the gimbal to search for the landing pads. The video is in 4 parts, starting with the drone already in flight, and beginning to search for the landing pad. The type of landing pad, the control policy mode, and the type of video stream are all annotated in the top left. The first part shows a visual landing starting from far away, using the visual zoom. The drone searches by spinning in place and tilting the camera up and down until it finds the landing pad by recognizing the attached marker. It then approaches the landing pad, aligns to it, and descends until touchdown. This process is repeated for the active and passive IR landing pads in the second and third parts. The fourth part shows another approach with the zoom camera, where the landing pad is intentionally obscured and tossed around to demonstrate recovery in different phases of the control policy.

The system carries out landings from much longer distances than previously reported using the visual landing pad and zoom camera. The IR camera's range is limited by its lack of zoom, but these experiments demonstrate the viability of using both active and passive IR fiducial markers for autonomous landing, including at nighttime. The active IR landing pad provides consistent performance and is clearly recognizable. The passive IR landing pad requires no power source but reflects the drone's IR radiation (see Figure 41) causing occlusion when horizontally aligned at low altitudes and necessitating the *commit* state, where the drone no longer needs to detect the marker in order to descend. The performance of the passive IR landing pad may be weather-dependent and requires further testing. Since the control policy is the same for all landing pads, the only major performance difference is the maximum detection distance. Our landing accuracy is comparable to that reported in the related work but does not improve upon it. Greater accuracy could be achieved with a tighter control policy, possibly at the expense of longer landing times.

Section 3.7.3.1: Case study: antagonistic landing site

Successes in the first experiment led to a more antagonistic second experiment where the landing pad was intentionally obscured both during the approach and descent states to test the system's ability to recover from detection loss. A demo is available in the last part of Video 2. Figure 42 shows the control signals for a landing with the visual landing pad and the zoom camera. The system reacquires the landing pad each time by zooming out or ascending to increase the area covered by its field

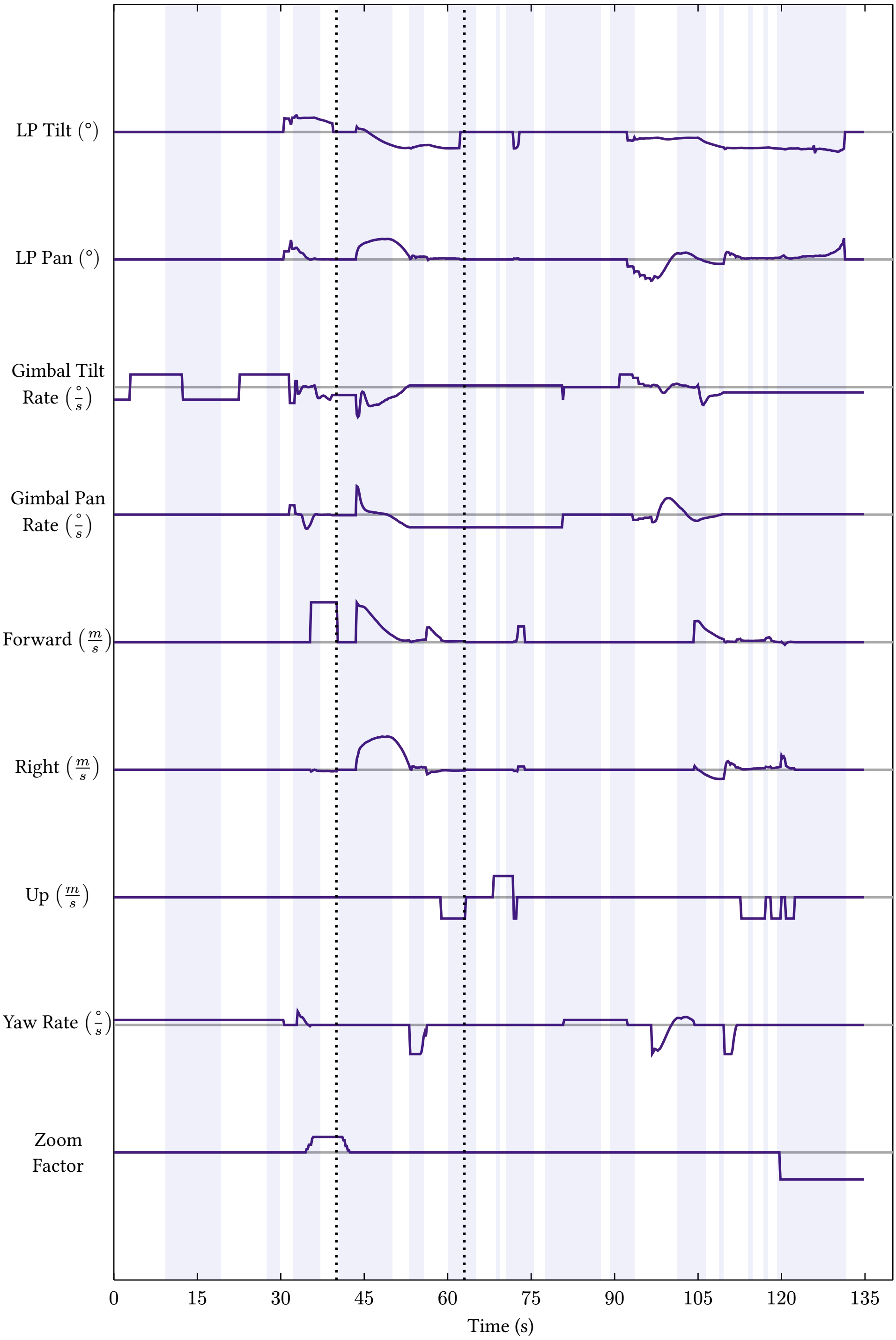


Figure 42: Control signals for the Matrice during an antagonistic landing scenario where the drone must search for the landing pad as it is intentionally obscured. Color changes indicate mode changes; vertical lines indicate intentional obscuration.

of view, restarts the search process entirely when necessary, and ultimately lands successfully. The drone starts at a distance of 20 m and altitude of 10 m relative to the landing pad. It searches for the landing pad by tilting the gimbal up and down while yawing clockwise from the start until $t = 30$. Then, it detects the landing pad, aiming the camera at it until $t = 32$ and aiming the drone at it until $t = 35$. It begins its approach until the landing pad is intentionally obscured and thrown to the side at $t = 40$, at which point it zooms out and finds the landing pad towards its right. It begins its approach again at $t = 44$, until arriving above it at $t = 53$, when it turns counterclockwise to align with the landing pad's yaw. It conducts a horizontal alignment at $t = 56$ and begins its descent at $t = 58$. At $t = 63$, the landing pad is again intentionally obscured and thrown to the side, causing the drone to enter the *zoom out 2* state. At $t = 68$, it begins its ascent and momentarily re-acquires the marker at $t = 71$. It zooms out and begins the search process again at $t = 78$, first with static search until $t = 80$, then search down until $t = 90$, and search up until $t = 92$, at which point it re-acquires the marker. It aims the camera again until $t = 96$, then approaches until $t = 109$, then aligns to the landing pad's yaw until $t = 111$. It alternates between horizontal alignment and descent 3 times until $t = 122$, as it is gradually getting closer to the landing pad in the horizontal and vertical dimensions, and during this time it switches to the wide-angle stream with $Z = 1$. Finally, it commits to the landing and descends vertically until it touches down on the landing pad and disables the motors at $t = 134$. This demonstrates that the system is able to recover from detection losses by zooming out, ascending, or restarting the search process.

Section 3.7.4: Conclusion

These experiments show the benefits of refining the system from Section 3.5. Avoiding the full 6 DoF pose estimation of the fiducial markers eliminates the erroneous control signals caused by orientation ambiguity and reduces data requirements, while still allowing for safe landings using only the pan and tilt directions from the drone to the landing pad. It also allows for easily switching between camera sources, so that the drone system can use the wide-angle, zoom, and IR cameras available through the H20T. The zoom camera enables the drone to detect the landing pad from long distances, and the IR camera enables it to detect the landing pad both at daytime and nighttime. The passive IR landing pad is detectable at night without power. The control policy allows the drone to restart the landing process if sight of the landing pad is lost. Although this method requires fewer data data sources than the related work, it still achieves similar landing accuracy at about 0.19 m offset from the center of the landing pad after touchdown.

Importantly for our subsequent research, this method showed that it is possible to conduct autonomous landing with an initial, safe search phase using minimal data:

the orientation of the gimbal, knowledge of some of the camera intrinsic parameters, and the pixel position of the marker. This allows flexibility in the type of target that the drone has to identify, and we can apply this method to future experiments if we can just determine a pixel position for the target.

This project provided an answer to **RQ1** in that the landing method itself was successful, did not exhibit erroneous control signals, and used a reliable, robust control policy. It similarly provided an answer to **RQ3** in that the method ran in real time onboard the drone.

Chapter 4:

Unstructured landing sites

Section 4.1: Introduction: what is a safe landing site?

Most places where a drone could potentially land do not have any kind of infrastructure that the drone can use to recognize them easily as in Chapter 3. Recall our 5 stages of landing from Section 1.1: search, approach, commit, descent, and touchdown. In the unstructured case, the components to develop are again *search* — the task of initially identifying a potential landing site in an otherwise unknown environment — and *commit* — the problem of determining whether the landing site is safe or unsafe upon closer inspection. After identifying a potential landing site for the first time, *approach* is the problem of tracking and control, which can largely be delegated to the drone in simple commands as shown before. Similarly, after the system commits to a landing, the *descent* and *touchdown* can be delegated to the production solutions in the drone's flight controller.

We explore an appearance-based strategy for the *search*, in the hopes that RGB video analysis should be a relatively lightweight method using only basic sensors. One drawback of this approach is that different types of environments have such different appearances that we cannot develop a single appearance-based method for all environments. Consider urban versus non-urban environments. While an appearance-based classifier trained in an urban environment may associate danger with straight lines, right angles, text, etc., these will be wholly absent from non-urban environments where the classifier should instead pick up on trees, ditches, boulders, etc. Legal requirements also vary between such environments: whereas drone systems have an obligation to stay safely away from people in urban environments, they would enjoy greater flexibility in non-urban environments where there are fewer people. Furthermore, the non-urban case necessitates onboard processing,

(a) Hills and steam plumes near the Mývatn Geothermal Area.

(b) Rocky, sandy landscape with Herðubreið (volcano) in the background.

(c) Black sand beach (Vík í Mýrdal).

(d) Grassy, uneven fields in Grímsey.

Figure 43: Examples of some of the varied Icelandic environments from which we can choose to test unstructured landing methods.

since a ground station or cellular network would often be unavailable. Finally, the *commit* stage often requires more certainty than appearance-based methods can currently give. We therefore need a geometric method using stereo depth cameras for a last-minute, final check to decide whether the drone should descend or search for another potential landing site.

This research has been carried out in Iceland, which, although it is a relatively small country, has many different types of landscapes (see Figure 43), and it is changing all the time. For example, it has black sand beaches, canyons, glaciers, grassy fields, hills, mountains, sand sheets, small forests, cliffs, lava flows, and urban and suburban environments. Even if some of the environments might not be suitable or relevant for research with drones, this still affords us many choices for testing. Given current events in Iceland — namely, periodic volcanic eruptions on the Reykjanes peninsula — we choose ultimately to target solidified lava flows as potential unstructured landing sites. Aside from being a unique opportunity in terms of available landscapes in the world, it is also particularly challenging. Lava flows are often incredibly rough terrain with only rare flat areas, and with lots of sharp surfaces, pits, cavities, mounds, cliffs, and uneven surfaces. Further, the newer lava flows tend to have fairly monotonic coloring, as compared to an everyday environment that a normal person might be used to, and that lack of coloring can make it difficult to extract information from a visual scene.

Interest in volcanic activity brings geologists from around the world to Iceland and invites collaboration, as geologists often study the eruption sites and use drones for remote sensing. To this end, an autonomous landing solution, or indeed simply an aid to help operators locate viable landing sites, would be helpful in deploying geological sensors in a lava field to gather valuable data quickly. Aside from topographical challenges of finding a good landing site that is clear of obstacles, there is the further consideration that the solidified surface may be dangerously thin with a flowing lava river still under it, as the lava solidifies from the top down. Of course, landing on solidified lava is already challenging, as we will explore in the coming sections, but even flying in the proximity of the eruption generally can be a challenge because of radiant heat, hot gases, steam, and sulphur. Further, since eruptions attract many scientists and tourists, there is often a lot of aerial traffic around the eruption site that clutters the airspace, from personal drones to full size helicopters and airplanes.

Section 4.1.1: Lava flows as a testing ground

Volcanic eruptions are major considerations in Icelandic society. In 2021, a string of eruptions began in the Reykjanes peninsula at Fagradalsfjall[2], about 40 km south-

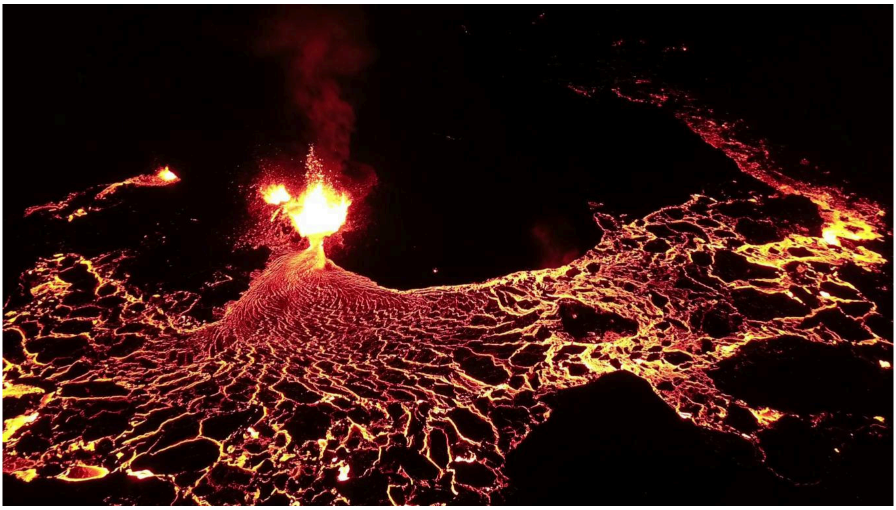


Figure 44: Nighttime, true color, aerial view of an eruption in 2022, with the crater at center-left, and lava pooling around it.

---

[2] A word on Icelandic place names: *Fagradalsfjall* is composed of the words *fagur* (beautiful), *dalur* (valley), and *fjall* (mountain), and this type of fusion is typical in many place names around Iceland, e.g., *Reykjanes*, from *reykja* (smoke) and *nes* (cape, headland); or *Hafnarfjörður*, from *höfn* (harbor) and *fjörður* (fjord). The word *hraun* (lava; rough, uneven surface) is similarly added to indicate lava fields, e.g., *Fagradalshraun*, *Holuhraun*, or *Selhraun*, which will be mentioned later.

west of Reykjavík, with over 50,000 earthquakes leading up to it in the preceding month [120]. The area has seen several further eruptions since (such as in Figure 44), with each lasting from days to months, and eruptions are expected to occur periodically for up to hundreds of years. These eruptions have mostly occurred in unoccupied land, covering many square kilometers of grassy fields, previous lava flows, and gravel. They have even filled valleys and overflowed into other valleys. However, they have recently begun to threaten nearby human infrastructure — in particular, a power plant, tourist attractions, and the town of Grindavík. In response, a large wall of dirt and rock has been constructed in an attempt to redirect the lava and protect the infrastructure. At the time of this writing, sadly, Grindavík has been mostly evacuated of residents, though some businesses are still operating. There has been some damage to the infrastructure despite efforts to stop the lava, including a small fissure opening up behind the wall in Grindavík and damaging houses, as well as molten lava flowing over the road leading to Grindavík and destroying a hot water pipeline servicing much of the nearby population.

Local and foreign geologists regularly study volcanic phenomena in Iceland, and we have had the opportunity to assist and shadow two teams, specifically observing their use of drones, which have become a primary tool for studying these eruptions. The first is the RAVEN project [45], [122], [123], [124]. In 2021, during the first eruption at Fagradalsfjall, we accompanied part of the RAVEN team to the eruption site, helping to deploy a drone containing a specialized thermal camera [121], as shown in Figures 45 and 46. In 2022, RAVEN conducted a fieldwork session in the Icelandic central highlands (see Figure 47). It set up a basecamp at Dreki (see Figure 47a) and had excursions to the 2014-2015 Holuhraun eruption site, Lake Askja (Öskjuvatn), Víti, and a nearby sand sheet and river. The goal of the fieldwork was

Figure 45: Flying the heavy lift drone with an IR camera over the Fagradalsfjall eruption site. © BBC, 2021. Used with permission. [121]

Figure 46: Hand-catching a Matrice 300 at the Fagradalsfjall eruption site because of a lack of safe landing area. © BBC, 2021. Used with permission. [121]

to carry out an analog Mars exploration mission to determine the potential scientific benefits gained by deploying a drone and a rover as a unified team. In this paradigm, the rover (see Figure 47b) acts as a sort of base, and the drone as a probe that can get to hard-to-reach areas like lava flows, where it can collect valuable geological samples by, e.g., drilling cores, or with laser spectrography. This is a prime scenario motivating a power-efficient, embedded processing method for autonomous landing on lava flows. The RAVEN fieldwork was primarily a geological project, but it offered many opportunities for technology demonstrations, e.g., testing laser spectrometers, testing drones and attachments for drilling into rock, conducting LiDAR and photogrammetry surveys, etc. It also gave the opportunity to get a close look at the Holuhraun lava flow not only to get familiar with it, but also to collect topographical data. The main takeaways are that lava flows tend to exhibit many different morphologies, from the particularly rough ʻa'ā to the relatively smooth pāhoehoe, and particularly smooth *lava rise plateaus* which would be best-suited for drone landing.

We have similarly collaborated with the geology department at the University of Iceland in July of 2023 (see Figure 48), helping to conduct several LiDAR surveys at the eruption sites on the Reykjanes peninsula, and providing some geotagged images of moss fires caused by the lava. Some of their work which partially stems from these missions was to determine the effectiveness of LiDAR in evaluating the amount of lava expelled during a given eruption, as well as to classify other geological formations [125]. We tagged along to get a close up view of the eruption site and another idea of what it takes to carry out fieldwork in this area. This eruption site, though only a few kilometers from Reykjavik, required an off-road drive of about 2 hours, for which we in the Reykjavik University Department of

(a) Dreki Basecamp with tents for common areas such as meeting and eating, and a shipping container for storage.

(b) The Mars rover from the Canadian Space Agency (CSA).

(c) Communal tent in a July snowstorm.

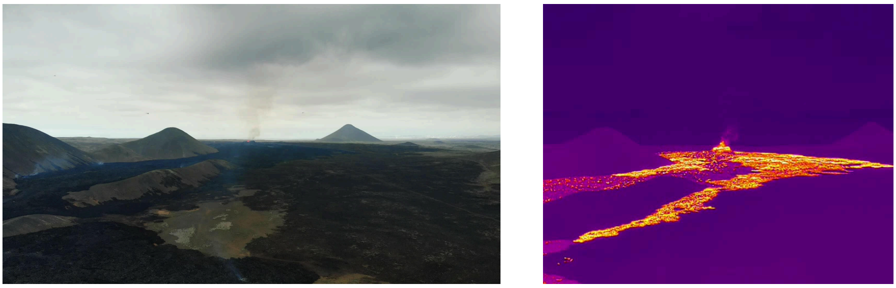
(d) Friable, rough area.

(e) Rough area with large crack.

(f) Very flat, viable landing site.

Figure 47: Images from the RAVEN fieldwork in 2022. The base camp was in personal tents and larger, communal tents at Dreki, with a storage container for equipment transportation, as in Figure 47a. Figure 47b shows the CSA Mars rover for the analog mission. Although the weather in the Icelandic highlands tends to be best in the summer, there can still be snowstorms in July, which we experienced as shown in Figure 47c. During the fieldwork, we walked on the Holuhraun lava flow and collected various data, including topographical data using a depth camera, at sites such as those shown in Figure 47d, Figure 47e, and Figure 47f, which illustrate how varied the landscape can be, and how rough and dangerous it is in some parts. Viable drone landing sites are relatively rare compared to the otherwise rough, cracked, uneven surface.

Computer Science did not have the vehicles. Therefore it was quite a benefit to be embedded with the University of Iceland team. Further, this type of off-road driving is brutal on equipment, with continuous bumps that can throw heavy Pelican cases airborne. This, in addition to the RAVEN fieldwork, motivated us in Section 4.3 to



(a) RGB view of eruption site, July 2023. (H20T, wide) (b) IR view of eruption site, July 2023. (H20T, IR)

Figure 48: Aerial view from fieldwork with the University of Iceland near Litli Hrútur in July 2023. The lava has already hardened on top, turning a dark color, but still emits a lot of IR radiation as a result of the internal heat, which can last from months to years. The ground in the bottom right has a layer of vegetation that has been burnt by fires.

conform our payload designs to fit in the DJI carrying case, which provides adequate protection in such scenarios. It further motivated us to design payloads with quick-release mechanisms for installation with minimal connectors and activity in the field, since storage, transportation, and installation are the most likely times for damage to occur to the drone and its components — not flight.

As it is important to conduct regular fieldwork in order to demonstrate methods in the real world, we have flown our drones many times at a location in Hafnarfjörður (see Figure 49), near Reykjavik in the Capital Area, which is just outside of the suburban area. The location is next to a large, open soccer field, an RC airfield, and an old lava field, making it an ideal location for many different types of testing. The lava field, unfortunately, has a lot of vegetation covering it, such that while it has overall a similar, cracked topography *underneath* as the newer lava flows like Holuhraun and Fagradalshraun, many of these formations are not immediately visible. However,

Figure 49: Aerial view of the testing area in Hafnarfjörður. The old lava field is visible in the foreground, the soccer field in the center, the RC airfield to the center left, and Hafnarfjörður in the background.

it still functions as a good lava practice environment that is relatively close to the university, meaning that we are able to deploy the drone in the real world often because the overhead costs of traveling there are low.

As the eruptions in the Reykjanes peninsula continued over the years, the lava has approached inhabited areas more and more. Of course, from a humanitarian perspective, this is disastrous and means the destruction of infrastructure, houses, and the evacuation of Grindavík, as explained earlier. Our silver lining is that the newest eruption sites are easily accessible from paved road, such that fieldwork on brand new lava flows (see Figure 50) no longer require off-road trips. The result is that, in the latest projects that we conducted, we were able to use LiDAR and video from the newest lava flows, which are not covered in vegetation like the practice area in Hafnarfjörður. The area is more like the Holuhraun lava flow in terms of roughness, variability, and color scheme.

Section 4.1.2: Chapter overview

The rest of this chapter is structured as follows. Section 4.2 discusses in detail our work in geometric landing site evaluation [41] using stereo depth cameras. We discuss some initial observations from the RAVEN fieldwork and another study at the Hafnarfjörður site using a custom drone, as this work was conducted before we obtained the Matrice 350. Section 4.3 describes the work to develop our upgraded Matrice 350 payload [43], that allows the operator to view the onboard computer screen using the Matrice's controller and interact with it in real time using screen clicks — two main features which were lacking in the payload we developed in Section 3.7 earlier. Section 4.4 discusses our work in appearance-based landing site identification as a means of searching the drone's environment using only RGB video [44]. We approach this as an image segmentation problem and train our classifiers using synthetic data generated from LiDAR surveys. Finally, we embed the method on our Matrice payload in Section 4.5, which can run the geometric and appearance-based landing site detection methods simultaneously. We propose to use them together, so that the appearance-based method can identify potential landing sites in a lightweight way during the *search* phase, and the geometric method can give a final sanity check on whether the chosen area is truly safe or not during the *commit* phase.

Section 4.2: Geometric landing site evaluation

This section describes our groundwork and background in determining whether or not a depth camera would be adequate for geometric landing site evaluation in a lava flow. The novelty lies primarily in the application itself. Although we do create our own implementation for doing the depth processing onboard our drone in our own way, the overall procedure is similar to other geometric landing methods that

(a) Dried lava field around a small hill. Much of the area is unsuitable for landing, but some of the smoother, “turtle shell” formations in the upper, center-left are adequately flat and level.

(b) Example of lava that is too rough and friable for landing a drone.

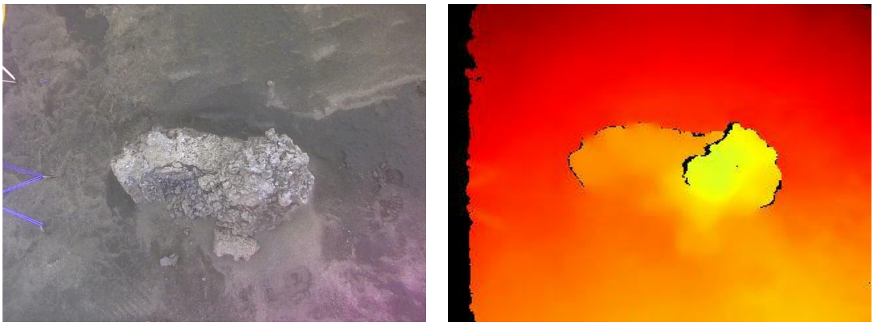
Figure 50: Aerial images from the eruptions in Reykjanes in October 2024. Figure 50a shows the an example of relatively smooth lava in the background which could be a viable landing site, whereas Figure 50b shows particularly rough textures that make up the bulk of the lava flow.)

have used depth cameras — primarily in urban environments. To our knowledge, this is the first application of such a sensor for real time landing site evaluation in a lava flow.

Geometric methods are traditionally used in the case of identifying landing sites in unstructured locations, since they can reliably ascertain the topography of a given area. The main challenge with geometric methods is their computational heaviness, particularly when using LiDAR, or reconstruction using, e.g., structure from motion (SfM). Appearance-based landing site evaluation makes the initial search for a landing site relatively quick and lightweight, but can occasionally fail by falsely identifying an unsafe area as safe. In this section, we discuss geometric landing site evaluation in unstructured scenarios using stereo depth cameras. We have conducted initial tests using a custom Tarot 680 hexacopter platform, as this work occured before we obtained the Matrice 350. We expand on the method and test it further in the field in Section 4.5.

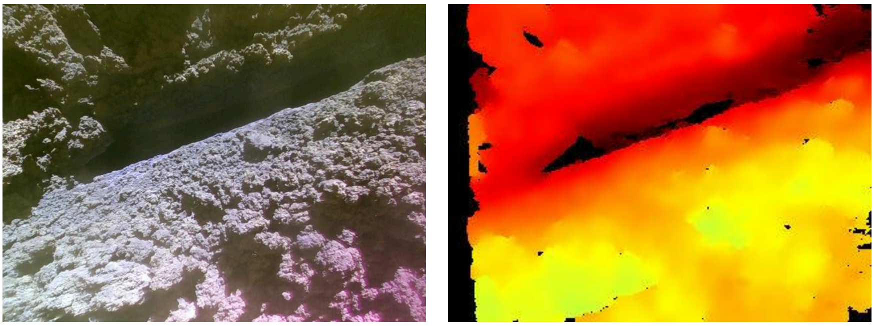
During the RAVEN fieldwork, we tested an Intel RealSense D455 depth camera (see Section 2.3.2) both in handheld scenarios and mounted to a drone as an exploratory look into the viability of depth cameras for identifying safe landing sites in lava flows. Specifically, we were determining whether or not our camera could resolve obstacles reliably given the limited colors and harsh shapes. As shown in Figure 51, it is able to resolve boulders in sandy scenes, and cracks, rough areas, and smooth areas on a lava flow. This work also revealed that drones produced by industry manufacturers were significantly more reliable, on average, than custom-built, in-house drones, and it motivated our eventual switch to enterprise DJI drones in subsequent projects. As this work occurred before we obtained the DJI Matrice 350 of Section 3.6 and Section 3.7, we still had to develop a custom platform to conduct testing in the field. As shown in Figure 52, we modified one of the Tarot 680 platforms from Section 3.3.1 to use a Cube Orange flight controller and Here 3 GNSS system, such that the drone was able to attain adequately low GDOP for reliable semi-autonomous control, which was a problem in Section 3.3.1 with the Navio2. We constructed a custom, 3D-printed gimbal to hold the depth camera (Figure 52b), and added a Raspberry Pi 4 as a companion board to process the data from the depth camera.

Figure 53 shows a visualization of the data collected by the stereo depth camera at Selhraun. The D455 internally processes the stereo images to create a depth map, shown in the top left. Similar to Figure 51, it is able to perceive the crack in the lava field. However, it is not only important to detect such major obstacles, but also to determine if the surface beneath the drone is flat.



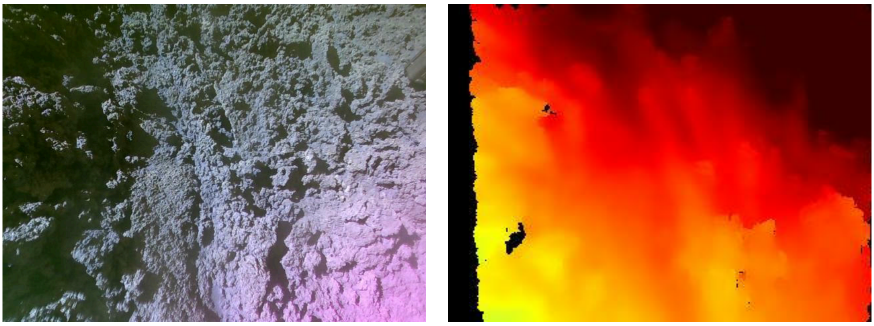
(a) RGB: flat, sandy scene with boulder.

(b) Depth: flat sandy scene with boulder.



(c) RGB: lava flow with large crack.

(d) Depth: lava flow with large crack.



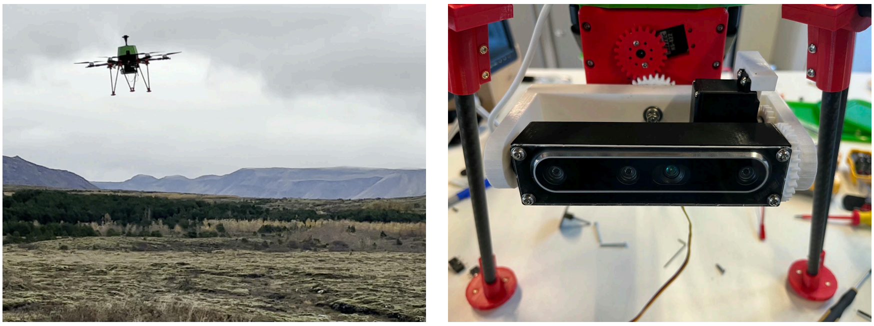
(e) RGB: rough, unsafe area in lava flow.

(f) Depth: rough, unsafe area in lava flow.

(g) RGB: potentially viable landing area.

(h) Depth: potentially viable landing area.

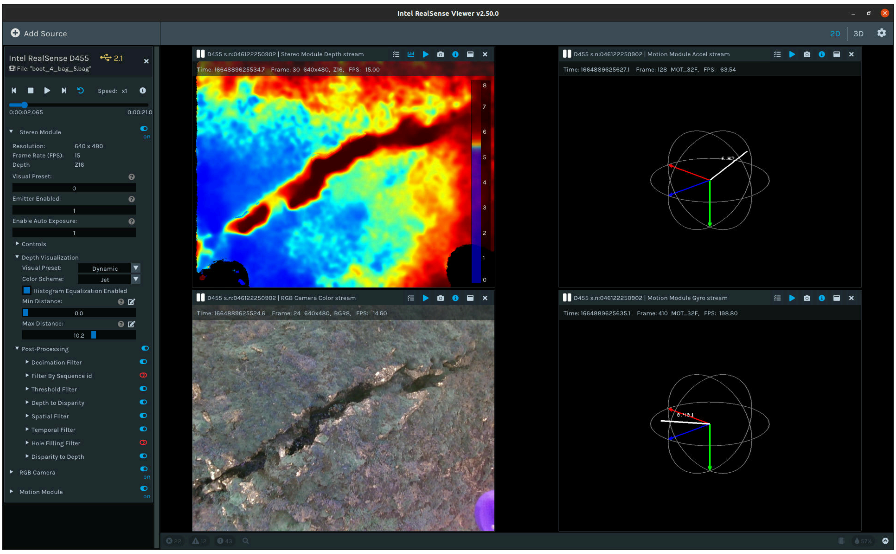
Figure 51: Testing a stereo depth camera at Holuhraun and surrounding areas. The left column shows RGB images (note the glare and lack of color of the scene). The right shows a visualization of the depth image. Brighter colors denote objects closer to the camera, and darker colors denote objects farther away. The depth images are matched to the right, leaving a small, undefined margin on the left.



(a) Tarot 680 with D455 collecting data at Selhraun.

(b) Custom gimbal for D455.

Figure 52: The modified Tarot 680 hexacopter, including an Intel RealSense D455 stereo depth camera.



File: "boot_4_bag_5.bag"

0:00:02.065 0:00:21.0

Stereo Module on

Resolution: 640 x 480
Frame Rate (FPS): 15
Depth Z16
Visual Preset: 0
Emitter Enabled: 1
Enable Auto Exposure: 1

Depth Visualization
Visual Preset: Dynamic
Color Scheme: Jet
☑ Histogram Equalization Enabled
Min Distance: 0.0
Max Distance: 10.2

Post-Processing
Decimation Filter
Filter By Sequence id
Threshold Filter
Depth to Disparity
Spatial Filter
Temporal Filter
Hole Filling Filter
Disparity to Depth

RGB Camera on
Motion Module on

D455 s.n:046122250902 | Stereo Module Depth stream
Time: 1664889625534.7 Frame: 30 640x480, Z16, FPS: 15.00

D455 s.n:046122250902 | Motion Module Accel stream
Time: 1664889625627.1 Frame: 128 MOT...32F, FPS: 63.54

D455 s.n:046122250902 | RGB Camera Color stream
Time: 1664889625524.6 Frame: 24 640x480, BGR8, FPS: 14.60

D455 s.n:046122250902 | Motion Module Gyro stream
Time: 1664889625635.1 Frame: 410 MOT...32F, FPS: 198.80

Figure 53: Screenshot from the Intel Realsense Viewer illustrating the data from the D455. It shows the RGB image in the bottom left, a visualization of the depth data in the top left, and visualizations of the accelerometer and gyroscope data at the top right and bottom right, respectively.

Section 4.2.1: Related work

Geometric landing site evaluation has been the subject of several studies. One approach is to use a database of known obstacles that can be recalled visually, for a lightweight, emergency landing site detection method [18] [19]. It also includes a visual method for recognizing dynamic obstacles and a path planner to attempt a safe, emergency landing very quickly (60s) without ascent. While such databases are often created with, e.g., DEMs or DTMs, they can also incorporate heat maps of cell phone activity [20]. In most cases, a database of known obstacles is either unavailable or out of date, and an extension of such approaches is to generate a local terrain model and then incorporate the same analysis. This can be done with a single monocular camera using a structure from motion (SfM) approach, which generates a 3D model of the terrain from many different monocular views collected systematically [22], [23]. Two methods use an SfM reconstruction for landing site detection specifically in the context of a Mars helicopter, testing offline using data from simulation and from the real world [24], [25].

Several studies use LiDAR to geometrically evaluate a scene for landing, and the geometric evaluation is typically based on slope and roughness. One approach is to conduct a survey of a relatively wide area and choose the best potential landing site based on fitting planes in the resulting point cloud that correspond to the locations of landing skids. This was tested on a full scale helicopter with an onboard computer [26]. A similar approach is to continually scan and keep track of potential landing sites during the course of an otherwise normal mission, and this method ran in real time onboard a drone with a ZED Box Orin NX [126], [127]. Similar methods are tested in the real world [27] and in simulation [28]. The UAS Operations Team in the RAVEN fieldwork manually chose landing sites in DEMs describing the Holuhraun lava flow based on slope, roughness, and terrain ruggedness, and the UAS Implementation Team carried out subsequent, manual landings were carried out near those sites to simulate autonomous Mars landing missions [45].

Stereo depth cameras have been used for geometric evaluation of a scene before landing, e.g., by creating a map of the environment and then searching it for a safe landing spot [30], or simply approaching the landing spot and evaluating it to make a decision as to whether or not to descend [29]. Stereo cameras can often miss small obstacles even from up close, and this effect becomes more prominent as the camera gets farther and farther away from the scene. Further, the depth map must be rotated before use to correct for biases caused by the sensor not being perfectly level.

Section 4.2.2: Methods

We first tested a stereo depth camera in a lava field at a walking height to determine if it can resolve the scenes well enough (see Figure 51). Now, we re-conduct the experiment on a custom drone to test again in flight above a lava field and develop an algorithm for interpreting the data in real time. Stereo depth cameras in this type of scenario are typically used from a relatively higher altitude than we demonstrate, in order to conduct a survey or evaluate a wider scene. We deploy our method from a relatively low altitude (< 4 m), and conduct only a small, initial evaluation in this section. We conduct a further evaluation in Section 4.5. Our method for pre-processing the depth maps is conceptually similar to previous methods [30] [29], but our approach to the decision problem is simpler — essentially a flood fill with tolerance from the central point of the depth map.

In order to determine if the area beneath the drone is topographically safe for landing, we analyze it using the depth map and IMU data output by the depth camera. We approximate the surface beneath the drone by calculating the straight-

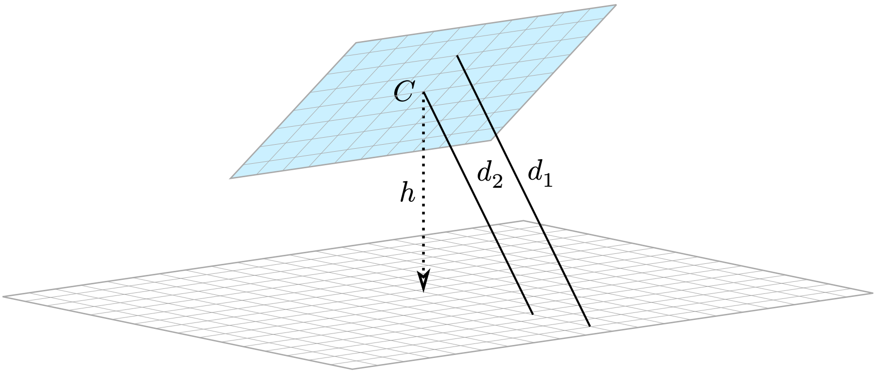
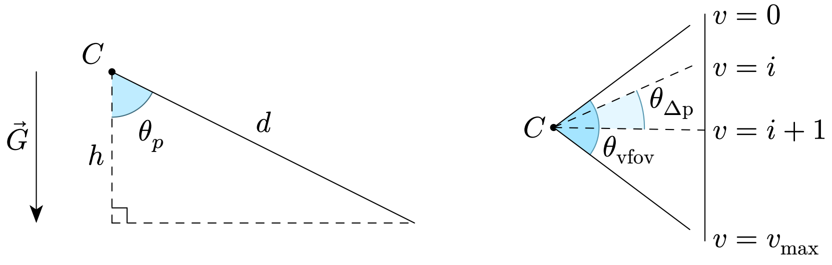


Figure 54: Illustration of artificial gradients in a depth map, resulting from the stereo depth camera not being aligned to the ground. A stereo depth camera at C outputs an array (the blue grid) of points describing the distance from the camera to each point in the scene. For our scenario, we need to calculate the height h from the camera to a given point. If the camera is not aligned perpendicular to the scene, then the distances shown in the output do not represent this height, and they exhibit a gradient, as points at the same height may have different distances. The blue grid is just a visualization of the camera's output — the entire output can be considered to exist at the point C, i.e., d_1 and d_2 extend to approximately the same point: the camera sensor.

line distance from the camera to the ground in the direction of gravity at each pixel. We begin by determining the gravity vector, since the D455 outputs only raw data from the IMU instead of a fused orientation estimate. We calculate the pitch and roll (ignoring yaw) of the stereo depth camera by using an exponential moving average on the accelerometer and gyroscope data from our Intel RealSense D455. (See [128] for the relevant code.) The accelerometer determines the pitch and roll without drift, and the gyroscope allows the system to respond more quickly to motion.

Recall from Section 2.3.2 that a stereo depth camera outputs a *depth map*, i.e., a canvas denoting distances from the camera to the surface shown at each pixel. If the depth camera views a scene under projection, this induces a gradient over the canvas, as shown in the depth map (right side) of Figure 57a. Even though the floor is a flat and level surface, as shown in the RGB image on the left side, the depth image shows a gradient from red to blue over the same, flat surface. This is what we have to remove. This phenomenon is also illustrated in Figure 54, where d_1 and d_2 describe distances from the camera to the same flat surface, but are different because of the non-zero pitch angle of the camera. In order to use the data from the depth camera, we have to convert these distances to vertical heights, i.e., distances in the direction of gravity. Figure 55 shows the basic considerations for finding the height at a single point, assuming the camera is level with respect to gravity in its roll. We can consider Figure 55a, where we already have the gravity vector $\vec{G}$, the pitch angle θ_p as measured from the direction of gravity, and the distance d from the camera to the point the camera is facing. Then, h can be very simply calculated as in Equation 18 below:

$$h = d\cos(\theta_p) \tag{18}$$



(a) Basic angles.

(b) Per-pixel field of view (FOV).

Figure 55: Basic calculations for the idealized situation when the depth camera's roll is level with the ground. Figure 55a: the height can be simply calculated as in Equation 18. Figure 55b: each pixel in the depth map, starting from index 0 at the top, has an equal share of the field of view. Therefore, the apparent pitch at a pixel in row i is offset from the camera's pitch by a corresponding angle.

The orientation of the camera corresponds to a line extending from the camera to the surface at the center of its field of view. Each point in the depth map is offset from the camera's orientation by some value depending on its position in the depth map. To determine this offset, we assume that each pixel describes an equal portion of the field of view of the depth map, as in Figure 55b. For now we assume the camera is level in the roll dimension, i.e., the roll $\theta_r = 0$. Then, we only need to consider the *inherent pitch* of each pixel (u, v) in the depth map, i.e., the pitch offset of a given pixel from the camera's pitch based on the pixel's position in the depth map, as in Equation 19. Here, $\theta_{p,v}$ represents the apparent pitch at row v, θ_p is the pitch of the camera, $\theta_{\Delta p} = \frac{\theta_{\mathrm{vfov}}}{v_{\max}}$ is the portion of the camera's vertical field of view for each pixel, c_v is the index of the center row of the depth map, and v is the row of the current pixel. The vertical offset of a given pixel from the center is $h_1 = v - c_v$, as illustrated in Figure 56. Pixels in the same row have the same inherent pitch.

$$\theta_{p,v} = \theta_p + h_1 \theta_{\Delta p} \tag{19}$$

However, our simple, custom gimbal does not have the stabilization that comes with production gimbals, so we cannot assume the idealized situation where the camera is level. We therefore have to consider the camera's roll, pitch, and the inherent pitch of each pixel simultaneously. Consider Figure 56, which shows a pixel (u, v) in the depth map, which has indices $(0, 0)$ at the top left and its center at (c_u, c_v). Instead of using h_1 as in the previous case, we derive h_2 by rotating by the roll of the camera θ_r. We take the radius from the camera's center to the point (u, v) as shown in Equation 20, and we take the inherent angle from the horizontal axis to the point (u, v), shown in Figure 56, as in Equation 21.

$$r_{u,v} = \sqrt{(u - c_v)^2 + (v - c_v)^2} \tag{20}$$

$$\omega = \arctan\left(\frac{v - c_v}{u - c_u}\right) \tag{21}$$

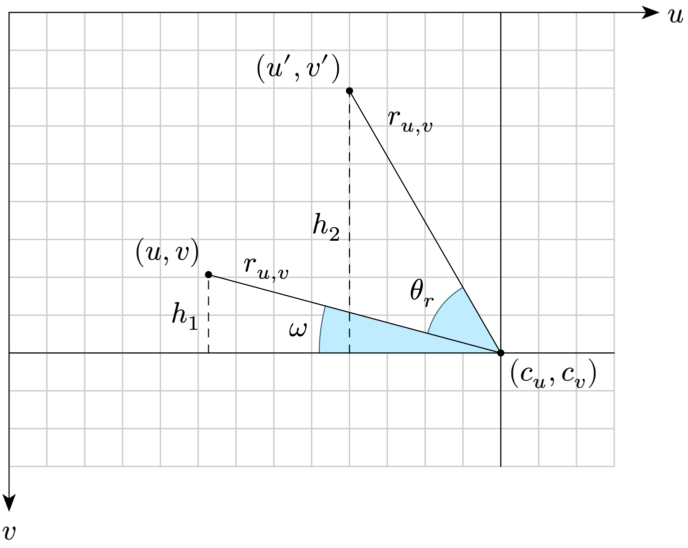


Figure 56: Calculations for the realistic situation when the depth camera is not pointing straight down in both pitch and roll. The *inherent pitch* of a particular pixel can be determined by the vertical offset from the depth map center to the pixel, giving h_1 and implying a radius $r_{u,v}$ at an angle ω from the horizontal center of the depth map. If we consider the roll of the camera, θ_r, then we can determine a point (u', v'), with radius $r_{u,v}$ and angle $\omega + \theta_r$ from the horizontal center of the depth map, which gives an effective pitch of the pixel, $h_2\theta_{v,p}$. Then, the vertical distance to the surface described by that pixel is calculated as in Equation 23.

Then, we calculate h_2 as in Equation 22.

$$h_2 = r_{u,v} \sin(\omega + \theta_r) \tag{22}$$

Then, we can calculate the vertical offset h from the camera to the surface at a given point as in Equation 23.

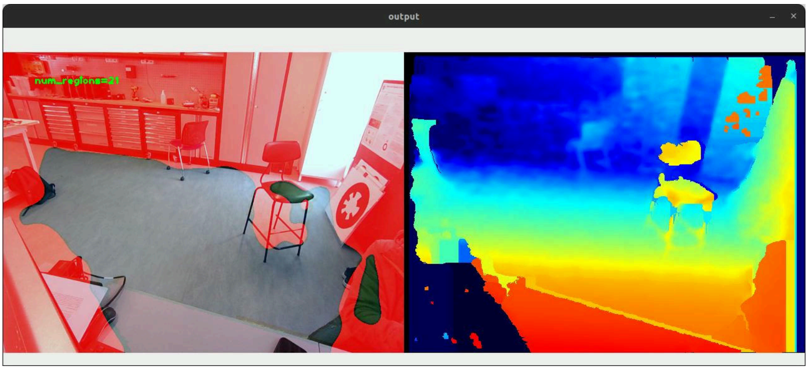
$$h = d \cos\left(\theta_p + h_2\theta_{v,p}\right) \tag{23}$$

In order to label the entire scene according to contiguous regions, we can then flood-fill the result with some tolerance. The product of this method is a scene as shown in Figure 57a.

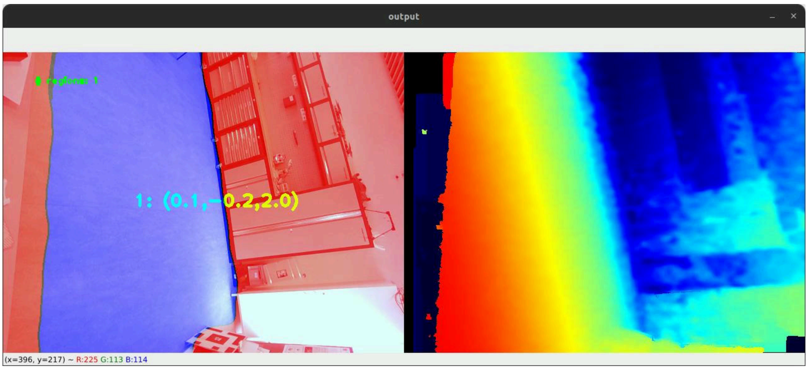
Next, we set the centroid of the largest safe region as the area we would like to approach, and we calculate the physical distance to that point. The forward displacement is similar to the vertical displacement, and we calculate it as in Equation 24.

$$h = d \sin\left(\theta_p + h_2\theta_{v,p}\right) \tag{24}$$

For the forward horizontal displacement, we consider the camera's horizontal field of view, θ_{hfov} and the equal portion of it that is described by each pixel, $\theta_h = \frac{\theta_{\text{hfov}}}{u_{\text{max}}}$, where u_{max} is the width of the depth image. Similarly to h_2, we calculate v_2, the yaw angle offset from the camera's orientation to a given point, as in Equation 25.



(a) Left: RGB scene with unsafe areas labeled in red. Right: depth image visualization.



(b) Left: RGB with safe and unsafe areas labeled in blue and red respectively, with position target. Right: depth image visualization.

Figure 57: Running the geometric landing site evaluation method in a lab environment.

$$v_2 = r_{u,v} \sin(\omega + \theta_r) \tag{25}$$

The right horizontal displacement is then calculated as in Equation 26, leading to the labeled position target vector in Figure 57b.

$$h = d \sin(h_2 \theta_h) \tag{26}$$

Figure 57 shows the method performing in a lab environment, properly identifying the floor as the primary safe landing area, even when tilted. We conducted an informal evaluation of this method by flying the drone over Selhraun to collect data with the depth camera. We then processed that data offline using a Jetson Nano, observing performance such as in Figure 58, where the crack in the lava flow is labeled as red (unsafe), area 1 is labeled as relatively safe (purple), and area 2 is labeled as most safe (blue). The area in the bottom center is unlabeled as a result of being neither safe nor unsafe.

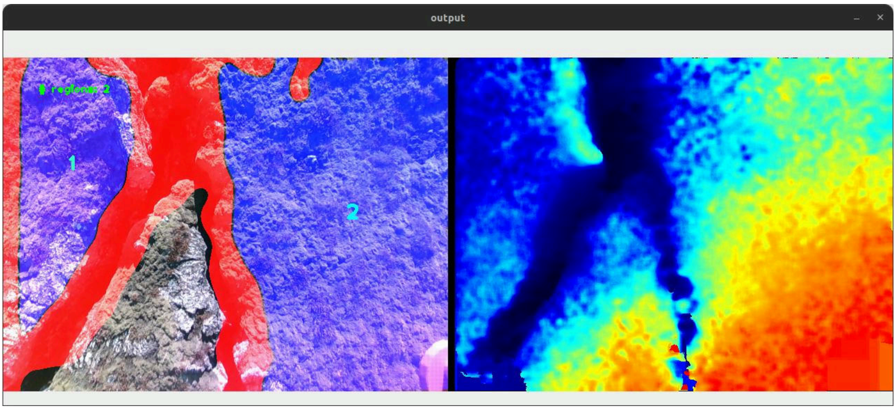


Figure 58: The geometric landing site evaluation using a stereo depth camera on the Tarot drone over Selhraun. The large crack in the center is recognized as unsafe, the region on the bottom is unlabeled as not being perceived particularly safe or unsafe. The regions on the left and right of the crack are both detected as safe.

Section 4.2.3: Conclusion

We created a method to use stereo depth cameras to evaluate landing sites geometrically, and have tested this method informally in a lab environment, on data from a relatively new lava field, and on data from a drone flying above an older lava field. This shows that the stereo depth camera is able to resolve the topography in the lava fields and that the data can be processed in real time on hardware that can be embedded onboard the drone. Although the stereo depth camera is not the most lightweight sensor, it is available natively on several enterprise drone systems, such that it is justifiable to use it even in the context of our preference for minimizing sensor requirements.

This method provides some insight into **RQ2**, since it shows that this type of geometric landing site evaluation can classify unstructured terrain topographically using a stereo depth camera. It also shows that such a method can run on a Jetson Nano, which can be embedded into a drone, providing some insight into **RQ3**.

The story breaks slightly between this section and the next: while we work on geometric landing site evaluation here, we begin to develop a robust computational payload for our Matrice 350 in the next section. This is a necessary extension to Section 3.6, which allows closer interaction with our companion board in flight, and again serves as necessary support for further research. We continue geometric evaluation of landing sites using the Matrice 350 in Section 4.5.

Section 4.3: Building a better payload

The previous autonomous landing experiment with our DJI Matrice 350, discussed in Section 3.7 (see that section for motivation and design decisions), required a custom-built computational payload for onboard image analysis and command generation with a Raspberry Pi 4 as a companion board. This payload worked for what it needed to do, but suffered from the significant limitation that the companion board could not send real time video to the controller, which made it difficult to diagnose problems in the field, primarily because the only way to know if the system had identified a landing pad was through its behavior or via a small text widget on the screen. In more complicated experiments, a single text box does not provide adequate information for an operator to determine whether the system is functioning properly, so we set out to make a new payload that can stream video to the controller in real time in addition to doing all the functionality of the payload in Section 3.6. Further, we use the Raspberry Pi 5, which is newer and has more processing power than the Raspberry Pi 4, and this is important particularly because of the overhead of video decoding and encoding.

This section discusses the following paper about our custom payload with a Raspberry Pi 5 in more detail and with a broader scope that includes embedding the Google Coral TPU for real time inference onboard:

> [43] J. Springer, G. Þ. Guðmundsson, and M. Kyas, "Lowering Barriers to Entry for Fully-Integrated Custom Payloads on a DJI Matrice," in *2024 International Conference on Content-Based Multimedia Indexing (CBMI)*, 2024, pp. 1–5. doi: 10.1109/CBMI62980.2024.10859236.

Section 4.3.1: Functionality goals

The main goal of this work is to iterate and improve on the payload outlined in Section 3.7. However, after discussions with collaborators who also use drones and who are not specialized in the engineering and computer science aspects of drone development, we also set out to make this payload easy to adapt and modify. With this in mind, we formalize the following requirements below, which share a lot of overlap with the payload from Section 3.6:

1. **Portability:** the payload quickly mounts onto the drone without assembly and can be carried in the drone's case as a typical DJI payload.
2. **Ease of software development:** the payload uses common computational hardware.
3. **Power/telemetry integration:** the payload uses the drone's power and telemetry systems, thereby reducing the amount of redundant equipment. This has been an issue in some collaborators' drone projects, as it was easier to add secondary transmission and power equipment rather than spend the time to integrate the added computer and sensors into the drone's existing power and telemetry systems.
4. **Flight/payload control:** the payload can control the behavior of the drone and of other payloads, e.g., it can aim the other payload cameras.

Port	Req. 3	Req. 4	Req. 5	Req. 6	Req. 7
E-port	✓	✓	✓	✓	✗
SkyPort V2	✓	✗	✓	✗	✓

Table 6: Per-port PSDK functionalities as of the production of this payload.

5. **Access to the drone's sensors:** the payload can get input from the drone's many onboard sensors and from other payloads, e.g., composite pose estimates, GPS data, AGL, information from distance and obstacle sensors, etc.
6. **Access to camera feeds:** the payload can view RGB and stereo depth video feeds from other payloads and from the drone in real time.
7. **Real time video processing and streaming:** the payload can process sensor video and can relay it to the pilot's handheld controller in real time.

For **Req. 1**, 3D printing is the easiest method available to us for developing a small form-factor case for these components that is adequately weatherproof. Physical space for safe transportation in the drone's carrying case limits the payload to a bounding box of about 15 cm x 15 cm x 11 cm. We print a simple case in the shape of a tapered dodecagonal prism with air holes at the bottom, and each component mounts vertically to take efficient advantage of the space. The components can be removed independently using a single screw and pulling vertically upwards. The payload attaches to the drone with 3D-printed quick-release latches. The payload case is designed in OpenSCAD and the files are available on GitHub [129].

To address **Req. 2**, we use the Raspberry Pi 5 [81] because it has support for a lot of software and sensors, is inexpensive, and relatively easy to use given its developed community. The mount point for onboard computers and many other custom sensors is on top of the Matrice, and has access to 2 payload ports: an E-port (for onboard computers) and a SkyPort V2 (for typical payloads) — both of which use the PSDK but have different capabilities [130], as shown in Table 6.

Section 4.3.2: Implementation

Section 4.3.2.1: Hardware setup

The E-port provides flight and payload control, the ability to integrate into the drone's power and telemetry systems, access to the drone's instruments (including video streams), but does not offer the ability to stream video to the controller for the pilot to view in real time. On the other hand, the SkyPort V2 can stream video to the controller but lacks the ability to access video feeds from the drone and other payloads, and also lacks the ability to control the drone and other payloads. Therefore, our payload requires accessing both ports. Each PSDK application can only connect to one port at a time, so the payload must run two separate applications

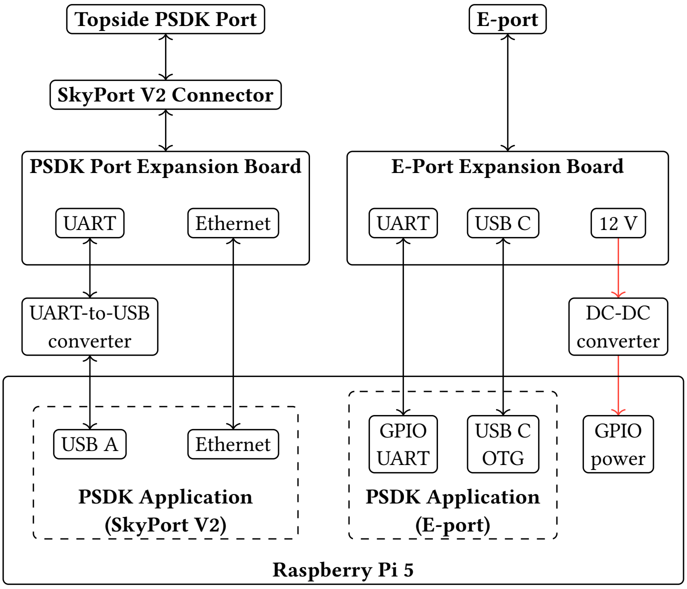


Figure 59: Connection diagram for the Raspberry Pi 5 payload for the Matrice 350. It runs two applications, both of which need their own hardware connection to the drone. The E-port application connects directly to the drone via the E-port expansion board and carries out most of the processing. The PSDK Port application connects from the PSDK Port expansion board to a SkyPort V2 gimbal connector and then to the drone, and it just streams a video feed to the controller to display the Raspberry Pi's desktop.

in parallel — a possibility which was hinted at by the DJI team, but not demonstrated or documented.

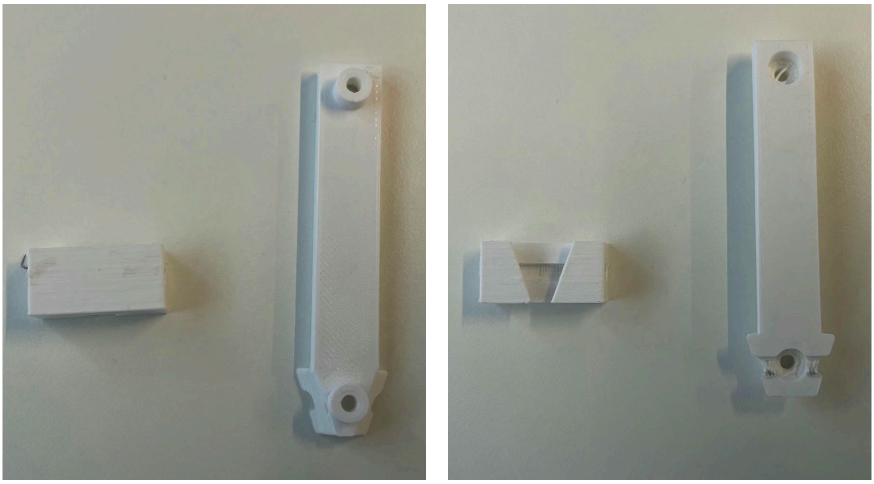
Figure 59 shows the hardware connection diagram of the payload. The Raspberry Pi 5 is the central component. Each application requires two data connections to the drone — a UART for low-bandwidth communication and configuration, and either a network or USB bulk connection for high-bandwidth data transfer such as real-time video. The E-port development board provides power to the Raspberry Pi both over USB C and via a DC-DC converter that converts 12V to 5V, fulfilling **Req. 3**. In terms of power consumption, the Raspberry Pi 5 can consume about 3.56% of the total energy capacity of the Matrice, and the Google Coral about 0.29% (see Table 9). Therefore, even after upgrading from the Raspberry Pi 4 to 5 and adding the Google Coral TPU, the total power consumption is still under 4% of the Matrice's capacity — therefore it is insignificant. We avoid using the 5V source from the E-port development board directly because of the undervoltage issues encountered in Section 3.6. The E-port development board also provides a UART interface to the Raspberry Pi and communicates with the Raspberry Pi over the USB-C port. The UART on the SkyPort V2 adapter board connects to the Raspberry Pi via a USB-to-UART converter and also via an Ethernet cable.

The primary application connects to the drone via the E-port and carries out most of the requirements. However, although it is similar to the application in Section 3.6, it requires a significantly different configuration of the Raspberry Pi operating system, as a result of its simultaneous operation with the secondary application on the SkyPort V2, to be discussed in the next section. As the PSDK is a black box and running simultaneous applications is not a documented, supported functionality, this particular aspect required a lot of experimentation and testing, since there are a lot of moving parts and the applications interfere with each other if not properly isolated, in addition to crashes that can happen inexplicably as a result of background processes in the OS itself.

In order to maximally isolate the applications, we take steps in the hardware setup itself. The SkyPort V2 application has a USB-to-UART converter which connects to the Raspberry Pi's USB A port for configuration and low-bandwidth communication. Since it has only an Ethernet interface for high-bandwidth data transfer, it takes priority on using network devices. On the other hand, the E-port application interfaces with the drone via the onboard GPIO UART for configuration and low-bandwidth communication, and via a USB bulk device over its OTG USB port for high-bandwidth communication. While we would like to have USB-to-UART converters for both applications to help with isolating the applications from the lower-level functions of the Raspberry Pi's operating system, early experiments showed that initializing one application would break the other's access to its USB-to-UART converter. Similarly, while we would like to use network interfaces for both high-bandwidth communication channels, i.e., `eth0` and `usb0`, we found it was more reliable to use only one.

The small form factor and relatively high number of components posed a challenge in the physical design of the payload. Although the case and quick-release brackets are similar to the payload in Section 3.7, the component mounting mechanism is different to accommodate for the lack of space. Once again depending on the only method we have available for reliable, quick manufacture — 3D printing — we designed new component mounting mechanisms in OpenSCAD to allow components to be installed or removed vertically and individually, and we 3D printed them into the case. These mechanisms had to have two parts — one on the payload case and one on the component to mount. While the part attached to the component could be printed in whatever orientation was needed and then fastened to the component, the part attached to the payload case had a fixed orientation and therefore had to have reliable print outcomes in that orientation. After several iterations, we settled on the connector shown in Figure 60, and on the component placement shown in Figure 61.

In addition to complexities in placing the components, we also needed to address the issue of some components simply being too large and needing to be downsized (see Figure 62). The SkyPort V2 port is designed for conventional DJI payloads, which are typically mounted on a gimbal for stabilization. Because of this, the interface for this port is in multiple parts including a gimbal stabilizer that is too large for the payload,



(a) Side 1.

(b) Side 2.

Figure 60: Component mounting mechanism for the Raspberry Pi 5 payload. This allows the components to be installed or removed vertically, individually, and reliably. After putting the long end into the short end, a single screw threads into the cylindrical space formed by half-cylinder spaces in each part.

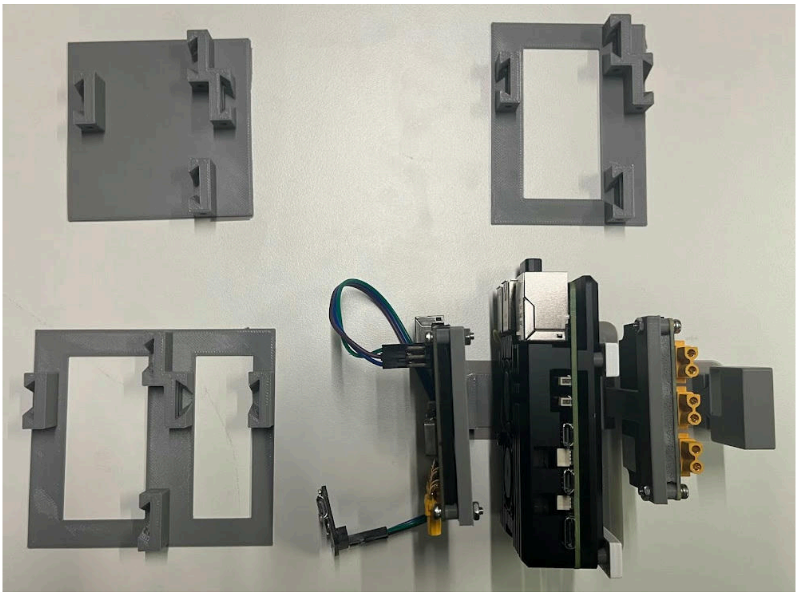
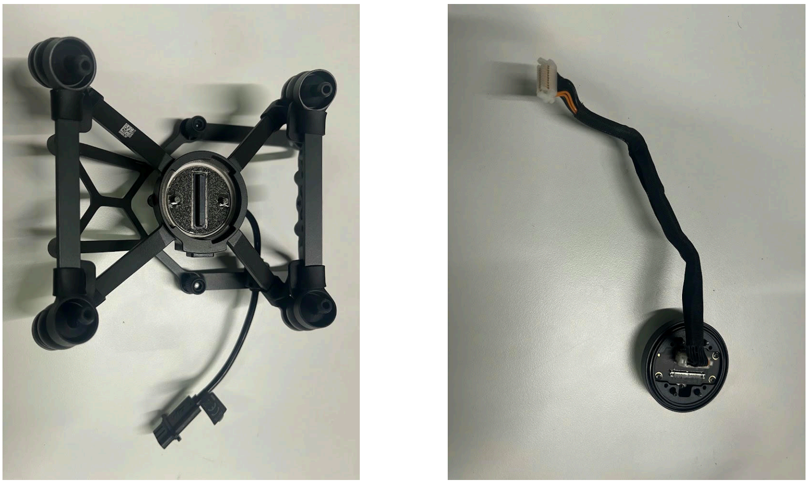


Figure 61: Iterative prototypes developed in determining the correct component placement in the Raspberry Pi 5 payload. After determining the correct placement, we printed the mounting mechanisms directly into the payload case bottom.



(a) Upward gimbal mount.

(b) SkyPort V2 adapter.

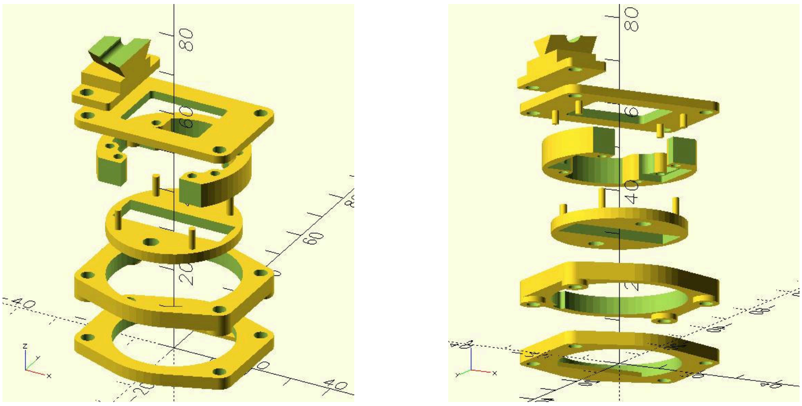
Figure 62: Components to connect devices to the Matrice 350 over the PSDK port. Under normal circumstances, the gimbal mount plugs into the PSDK port and mounts to the upward payload mount point. The SkyPort V2 adapter plugs into the circular port in the middle of the gimbal mount, which is stabilized and designed to hold the load of the gimbal and whatever custom sensor is attached to it. Its cord then plugs into the PSDK port development board and exposes a UART and Ethernet connection for the PSDK applications running on the Raspberry Pi.

shown in Figure 62a. Furthermore, we do not actually require stabilization because we do not need to aim any sensors from our payload, which is primarily a computer. Therefore, we disassemble the gimbal mount and reassemble it with the SkyPort V2 adapter using a set of 3D printed rings as shown in Figure 63. This assembly was tedious to design because it has to depress multiple small buttons simultaneously, while lining up all the components and providing an interface to mount them in the component mounts we had already designed. Ultimately, it produces the small assembly that fits completely inside the payload to the far left of the Raspberry Pi in Figure 64. The payload designs and 3D models are freely available for download on Thingiverse [112].

We install a Google Coral USB accelerator (see Figure 65) for anticipated real time onboard inference using TFLite. Experiments showed that this component, when placed unshielded (as shown in Figure 65a) in the payload and performing inference, emitted enough electrical noise to interfere with the drone’s compass, after which the drone refused to take off. We therefore removed its plastic case and wrapped about 20 layers of aluminum foil around it and added a ferrite choke around its USB cable (as shown in Figure 65b).

Section 4.3.2.2: OS setup

The following outlines the configuration for the Raspberry Pi 5 to make it work with PSDK version 3.8.1, leaving out minor steps, e.g., configuration of environment variables, installation of dependencies, and intermediate compilation of necessary programs, which is described in detail on the GitHub page [131] and automated in



(a) Exploded view, above.

(b) Exploded view, below.

Figure 63: Assembler mechanism for the disassembled components from the gimbal mount and SkyPort V2.

the corresponding configuration script [132]. We have tested it with the Raspberry Pi OS 32-bit version from 15 March 2024. (We have since confirmed that the configuration does not work unedited for later versions, so care should be taken if one upgrades the payload.) The steps are as follows:

- switch from Wayland to X11 desktop environments for easier screen grabbing,
- set screen resolution to 640x480 for faster encoding,

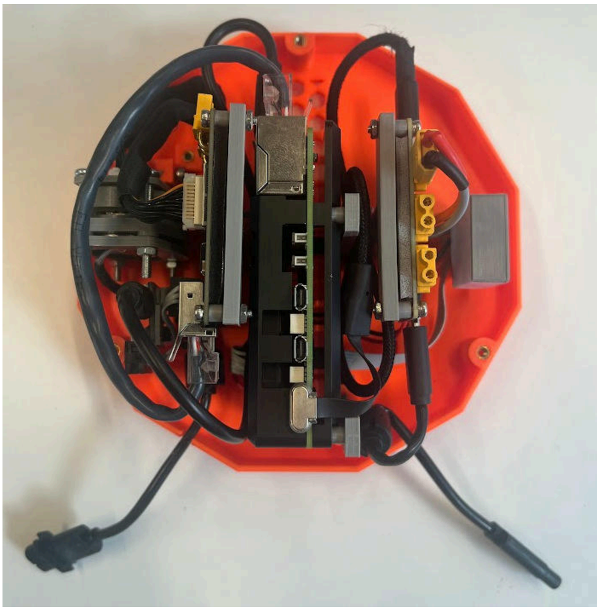
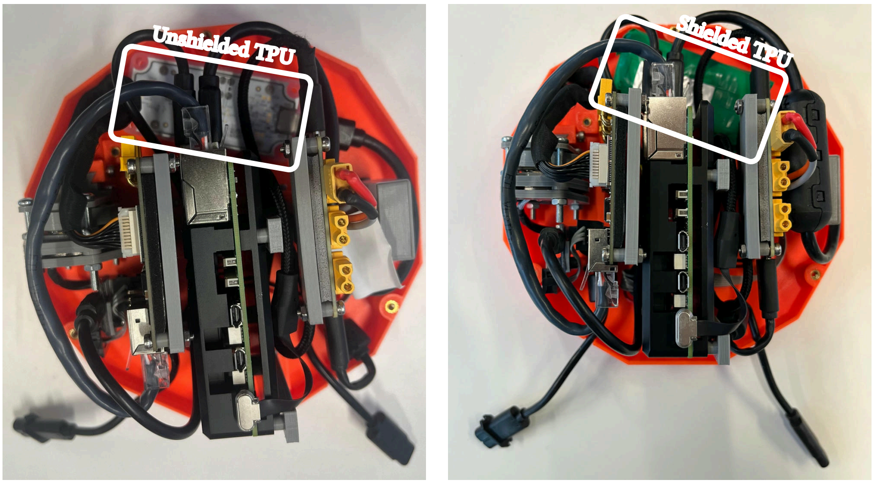


Figure 64: The open Raspberry Pi 5 payload. All components are mounted vertically. In the center is the Raspberry Pi 5 with a heat sink case. To its right is the E-port development board, and to the far right is a DC-DC converter stepping the E-port development board’s 12 V source down to 5 V for the Raspberry Pi. To the left of the Raspberry Pi is the SkyPort V2 development board, and to the far left is the assembly for the SkyPort V2.



(a) Google Coral placed unshielded. (b) Google Coral with aluminum foil shielding.

Figure 65: The Raspberry Pi 5 payload with the Google Coral TPU, which required shielding. Left: the Google Coral TPU in the payload in its stock, white case, which exposed the Matrice to interference such that it refused to take off. Right: The Google Coral TPU wrapped in many layers of aluminum foil and green tape, with a large ferrite choke on its USB cable (far right), which mitigated the interference so that the drone could fly.

- enable `uart0` to be able to use the onboard UART interface,
- enable USB gadget mode by loading the `dwc2` driver on boot,
- enable `libcomposite`, making it easier to configure composite USB devices
- configure the composite USB device on boot and create 2 endpoints (although apparently not documented, the Matrice uses one endpoint for video streams and one endpoint for onboard stereo camera streams),
- compile the E-port application and SkyPort V2 application,
- create service files to run the applications on boot
- add a delay to make the SkyPort V2 application stream the Raspberry Pi's desktop only after waiting for 3 minutes, as a workaround to avoid interference from the OS, which otherwise breaks the video stream.

Although we have created a working configuration, this has been pieced together from multiple tutorials from DJI and from the Raspberry Pi community, and is highly specific to our use case. It cannot be expected that this exact configuration will generalize to other DJI drones, or even to other versions of the Raspberry Pi OS on the Raspberry Pi 5, although it will likely require only some small modifications.

Section 4.3.2.3: PSDK applications

For the E-port application, we primarily need to extract video from the drone's payloads, which is possible through edits in the Jetson Nano C++ example included in the default PSDK code. From Sections 3.6 and 3.7, we know that this fulfils **Req. 4**. We modify it to begin the `Liveview` sample automatically, instead of explicitly navigating a menu. For completeness, we verify that it can also access the feeds from the stereo depth cameras (fulfilling **Reqs. 5** and **6**), which requires configuration of the secondary USB bulk endpoint.

The PSDK port, involving non-standard functionality, is slightly more involved to set up. It primarily follows the existing camera emulation (`camera_emu`) functionality, by configuring the high bandwidth data channel, setting a video stream of format `H264_CUSTOM` (as opposed to the apparently proprietary alternative of `H264_DJI`), and activating the base camera emulation service, as below:

```
  cout << "Setting bandwidth proportion." << endl;
  const T_DjiDataChannelBandwidthProportionOfHighspeedChannel
bandwidthProportionOfHighspeedChannel = {10, 60, 30};
  returnCode = DjiHighSpeedDataChannel_SetBandwidthProportion(
    bandwidthProportionOfHighspeedChannel
  );

  returnCode = DjiPayloadCamera_SetVideoStreamType(
    DJI_CAMERA_VIDEO_STREAM_TYPE_H264_CUSTOM_FORMAT
  );

  returnCode = DjiTest_CameraEmuBaseStartService();
```

We begin a subprocess that captures the Raspberry Pi’s desktop continuously using the `X11` plugin for `ffmpeg` and passes it directly to the main process via a pipe to avoid overhead in setting up shared memory locations, dealing with mutual exclusion, and having to manage multiple processes at boot. This is done as below:

```
// Command to capture screen using ffmpeg
std::ostringstream ffmpeg_command_stream;
ffmpeg_command_stream << "ffmpeg -loglevel quiet -f x11grab -r "
                       << FPS
                       << " -s "
                       << WIDTH
                       << "x"
                       << HEIGHT
                       <<
                       " -i :0.0 -f rawvideo -pix_fmt yuv420p -";
std::string ffmpeg_cmd = ffmpeg_command_stream.str();

// Open subprocess to capture screen
FILE* ffmpeg_pipe = popen(ffmpeg_cmd.c_str(), "r");
```

Finally, in an infinite loop, we read the frames from the desktop video stream, encode them to H264 (in software), and forward them to the controller. To keep the load on the Raspberry Pi’s CPU at or below 10% while providing adequate video quality, we set the desktop resolution to 640x480 pixels and transmit at 24 FPS. We configure `ffmpeg` to encode with the `ultrafast` preset using a bit rate of 1.2 Mbps, and a `GOP` (“Group of Pictures”) of 2, such that the streamer alternates between sending key frames and I-frames, which makes the video transmission smoother and more reliable, though not as lightweight as it could be. We do not send B-frames. This fulfills **Req. 7**.

For more flexible interaction with the payload than the typical widgets available directly in the Matrice controller's GUI, we have modified the screen click callback within the `camera_emu` PSDK module to propagate clicks to the Raspberry Pi, using `xdotool`, such that we can essentially use the controller as a mouse. The modification is as below:

```
static T_DjiReturnCode SetFocusTarget(T_DjiCameraPointInScreen target)
{
    USER_LOG_INFO("set focus target x:%.2f y:%.2f",
      target.focusX,
      target.focusY);

    memcpy(&s_cameraFocusTarget,
           &target,
           sizeof(T_DjiCameraPointInScreen));

    int mouse_x = 640 * target.focusX;
    int mouse_y = 480 * target.focusY;

    char command[100];

    sprintf(command,
      "xdotool mousemove %d %d click 1",
      mouse_x, mouse_y);

    system(command);

    return DJI_ERROR_SYSTEM_MODULE_CODE_SUCCESS;
}
```

Section 4.3.3: Results

The payload appears very similar to that of Section 3.7, as shown in Figure 66. It fits in an otherwise empty slot in the Matrice's carrying case, as shown in Figure 67, making it safe and easy to integrate into the workflow of transporting the drone to the field. The Raspberry Pi desktop screen shows up reliably on the Matrice controller, with a latency of about 300 milliseconds. This is most likely due solely to the slow software encoding on the Pi, since the high-definition video streams from the other cameras have almost no latency. After a (relatively long) delay of 3 minutes after boot, the payload begins streaming the desktop video to the controller, at which time the operator can single-click on a desktop icon that starts the E-port

Figure 66: The Matrice with the Raspberry Pi 5 Payload and L2 LiDAR module in flight.

application to view the drone's onboard sensors and video streams. We show the feed from the drone's downward-facing stereo depth cameras in Figure 68.

There are a number of quirks to deal with in addition to the required startup order of the PSDK applications (and the delay), and the isolation of the hardware interfaces for each application. First, it seems that the system is somewhat intolerant of major changes on the high-bandwidth data connections after an initialization period. This manifests in occasional crashes of the E-port application when repeatedly enabling or disabling the transmission of video streams to the payload. In future payload versions, if multiple video streams are needed, i.e., both the video stream from a payload and the stereo depth cameras, we anticipate the system will be more stable if it simply enables and subscribes to both streams from application start, and that the overhead of decoding both streams will simply be a necessary operating cost. However, further processing, e.g., calculation of a depth map from the stereo images, can still be dynamically enabled or disabled after decoding to save on computational capacity without affecting the interface with the drone. Second, at the time of development of this payload, it was not actually possible to subscribe to the stereo images over a network connection, but rather required the USB bulk connection, and in fact required the second endpoint of such connection, leading to our adaptation of the USB bulk setup script that was provided by DJI for the Matrice 30. Ultimately, we prefer this method because it further isolates the PSDK applications from each other. Third, the SkyPort V2 connector was designed for the Matrice 300 and released before the Matrice 350, and it required a firmware update before being compatible with the Matrice 350. We attempted to conduct this update on the Matrice 350, as it was apparently possible in the DJI Assistant software, but

Figure 67: The Raspberry Pi 5 payload stored in the payload compartment in the Matrice 350′s case. An important aspect in being able to practically use the payload is that it should be able to be transported safely and conveniently, so we designed it to fit into the Matrice's protective case, similarly to the H20T (bottom right). This is a primary consideration because transportation to and from some field sites involves very rough off-road driving during which the entire case is bouncing and shaking for multiple hours, which is a prime opportunity for unintended damages to the payload, particularly if it does not have a proper carrying case.

Figure 68: The view of the Raspberry Pi's desktop on the Matrice's controller. It is displaying the stream from the Matrice's left, downward-facing stereo camera. The streams from these cameras are not visible by default, as they are primarily used for obstacle detection. We expose them to try to make the full functionality of the Matrice available and because we anticipate using them for geometric evaluation of the area beneath the drone later.

this bricked our SkyPort V2 until we were able to redo the update from a Matrice 300 through our local DJI store using a specific version of the DJI Assistant software. Finally, since essentially all independent components of the Matrice 350 are *smart*, from the payloads to the batteries, some repeated runtime errors from our PSDK applications during their development caused errors that propagated unexpectedly deeply through the system. For example, the system has given warnings about errors in the batteries after the PSDK application encountered issues and crashed. We cleared the errors by simply rebooting the drone. Overall, these quirks are minor and we are quite satisfied with the Matrice as a whole, though the PSDK itself could be better documented to save developer time.

Section 4.3.4: Conclusion

We have demonstrated a method for integrating custom computational payloads into the DJI Matrice 350 using a Raspberry Pi 5. The system can access the drone's sensor and video feeds, control the drone's flight and the behavior of other payloads, process video and stream it to the controller in real time. It can also provide basic interaction with the Raspberry Pi via mouse clicks on the controller's screen. We have provided the scripts to configure such a system, so that such functionality may be within reach for researchers from other domains. The payload is cheap and can be easily adapted to deploy other sensors supported by the Raspberry Pi 5. Finally, we have integrated a Google Coral TPU into the payload for real time inference with a deep learning model, as we will see in the next section.

This project does not directly answer any research questions, but provides groundwork for answering **RQ2** and **RQ3** in the next section.

Section 4.4: Appearance-based landing site evaluation

We propose an appearance-based classification approach to the *search* problem of initially identifying a landing site, hoping that this will be a relatively lightweight approach compared to geometric methods. In the previous section, we proposed a computational payload for our Matrice 350 that is designed for running real time deep learning models for classification and transmitting video to the Matrice's controller in real time for debugging. In this section, we lay out the appearance-based classification method itself and test it in the real world using the payload. (We present the payload and the method out of chronological order.)

This section discusses the findings of the following publication:

> [44] J. Springer, G. Þ. Guðmundsson, and M. Kyas, "Toward Appearance-Based Autonomous Landing Site Identification for Multirotor Drones in Unstructured Environments," in *MultiMedia Modeling*, I. Ide, I. Kompatsiaris, C. Xu, K. Yanai, W.-T. Chu, N. Nitta, M. Riegler, and T. Yamasaki, Eds., Singapore: Springer Nature Singapore, 2025, pp. 198–211.

Section 4.4.1: Related work

Landing site detection is similar to the notion of *traversability*, which has been explored extensively on ground vehicles. One survey of ground vehicle traversability methods divides them generally into appearance-based, geometric-based, and mixed methods [133]. Appearance-based methods analyze terrain using visual sensors, whereas geometric-based methods require sensors such as LiDAR or RGB + depth (RGBD) — essentially the output of a stereo depth camera such as in Section 2.3.2 — that can extract a 3D representation of the environment. Some methods show success in traversability analysis, where training data is collected and labeled directly with 3D sensors, and terrain classification is performed with visual sensors only [134],[135]. Some others classify 3D data from LiDAR and RGBD cameras directly [136]. There is also a tendency to classify each pixel into one of three groups, e.g., traversable, not traversable, and unknown. These methods typically do not analyze reconstructed terrain models, but instead, analyze the terrain directly via some sensor and apply that analysis to label RGB images. Many traversability methods are developed and tested in simulation, but the gap from simulation to reality is often not overcome. While appearance-based methods can lack accuracy, they only require simple RGB camera sensors that are abundant, cheaper, and have a lower processing overhead than the hardware needed for geometric-based methods [133]. Many of the methods presented require specialized data sets with multiple data sources, i.e., RGB and at least one LiDAR or RGBD camera, to determine a label for the terrain.

Most work in autonomous landing site identification in unstructured environments is geometric-based, requiring real-time LiDAR or RGBD analysis, full-size GPUs, etc. [137], [138]. However, some methods visually locate their starting location using its visual appearance only [21]. More general appearance-based methods in this context are less widespread, as they often depend on expensive, manually-labeled data sets. In this context, some methods exist to sparsely label video frames manually, and then propagate the labels from frame to frame [139]; others use previously trained neural networks to add new labels to drone videos [140]. These methods have not yet been applied to autonomous landing but have potential in identifying viable landing sites. Many methods are tested only in simulation or laboratory settings on real drone video but are not embedded onto the drone [31],

[140]. SafeUAV serves as an initial proof of concept for our proposed method [34]. It uses existing synthetic data sets from Google, making it possible to quickly extract training data from many locations but making the data less dynamic when the environment has recently changed. Further, this method depends on Google having actually mapped a site similar to those one might want to classify, which limits its use cases. Their setup requires a camera at a fixed angle of 45° below the horizontal, and their classifiers predict the scene's 3D depth and landing safety. Crucially, they consider the problem of embedding their classifiers on a drone in the real world and therefore test them on embedded hardware (an NVIDIA Jetson TX2) and actual drone footage, although they do not deploy their solution on a drone.

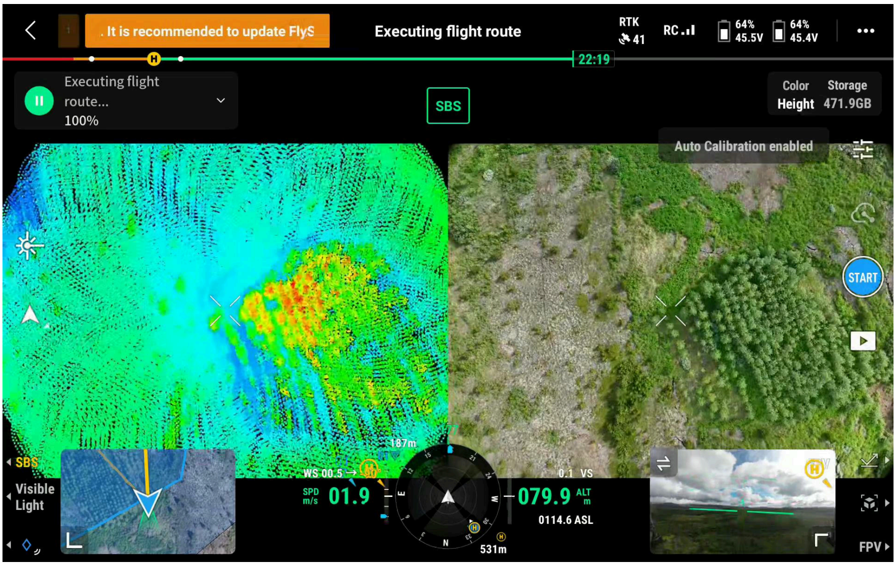
Section 4.4.2: Methods

We take inspiration from much of the related work and seek to create a full-pipeline approach that allows for flexibility in data requirements, minimizes manual labeling as much as possible, and ultimately produces a viable embedded terrain classifier that can run in real-time onboard a drone. To satisfy the real-time aspect, we prefer to use an appearance-based approach such that the classifier can perform inference on RGB images, which are lightweight to analyze when compared to, e.g., LiDAR data. This also makes the method easier to generalize to drone platforms without LiDAR or RGBD cameras. For flexibility in data requirements and ease in manual labeling, we generate intermediate 3D models from which we produce a synthetic data set of RGB images and masks. This gives the advantage that we can use many different data sources to generate the intermediate models, e.g., photogrammetry, LiDAR, RGBD, etc. This also means that we do not necessarily have to collect our data but can use openly available, standard-format, unlabeled data sets from terrain surveys. We also allow for the ability to quickly add manual labels to the intermediate models one time; such labels then propagate to all of the images generated. Importantly, we can vary the angle of our camera, which inherently makes the classifier more flexible than SafeUAV [34]. Finally, while we do not prescribe a particular, optimal classifier architecture, we create a successful U-Net that is relatively tiny (on the order of 1 MB instead of more than 1 GB) and can be deployed on power-efficient hardware compared to all methods described earlier. We also showcase how our classifier can be embedded onboard a drone.

There are several steps in generating our proposed classifier, many of which we have automated — except for logistical tasks such as transporting the drone to the survey location and optional, manual label refinement. We recall the ability of the drone to land autonomously given a simple control policy dependent primarily on the angle from the drone to the landing site, as demonstrated in Section 3.7. Given this method, we limit the scope of the demonstration for the unstructured landing sites purely to the *identification* of such sites, and we avoid demonstration of the full approach and landing in order to save time. The method from Section 3.7. can then be used once a pixel location for the landing site is determined.

Section 4.4.2.1: Data collection phase 1: geometric surveys in analog environments

Breaking from the approach from SafeUAV [34], we develop a process for collecting data ourselves using surveys from our own, select environments, in order to keep the method general and enable it to be applied to environments where data is not readily available and must be collected. The first step is to select an environment that is similar to where the drone will be landing, i.e., an Icelandic outdoor environment in summer, a forest, a beach, a lava flow, etc., where it is possible to carry out a terrain survey. Similarly, it is possible to simply download an openly available terrain survey, e.g., through the WebODM forums. What counts as *similar* enough is yet to be explored, but we essentially look for a source environment in which the hazards and safe areas appear visually similar to those in the target environment. In the source environment, we conduct a drone-based survey of the terrain to create a 3D model. We test 2 techniques: photogrammetry and LiDAR. Photogrammetry surveys produce a series of geotagged images that can be stitched together later into large, composite pictures, i.e., orthomosaics, or can be transformed into colored, 3D



Video 3: The view from the Matrice's controller during a LiDAR survey with the L2 to give an intuition of LiDAR point cloud data. Available at https://vimeo.com/j0shua/lidar-example-m350. The right side shows the visual appearance of the scene for a human-understandable view, and also for colorizing the point cloud. The left side shows a visualization of the point cloud, where lower altitudes are marked with cooler colors, and higher altitudes with warmer colors. The sensor is therefore picking up the fact that the trees are taller than the surrounding area, and indeed it has enough resolution to distinguish between individual trees. The sensor is scanning in a spiral, "non-repetitive" mode, though it is also possible to configure it to scan in a straight-line "repetitive" mode.

models of the surveyed scene by matching key points in each image and extracting depth data. LiDAR produces point clouds representing the scene geometrically, and these can be colorized by registering them with simultaneously collected RGB images. Video 3 shows the Matrice's controller during a LiDAR survey, where the RGB view is on the right and the LiDAR point cloud is on the left. One has the option to use either oblique or orthogonal data collection in these surveys, referring to the notion that the sensor should be pointed straight at the surface being studied, or should view it at an angle, respectively. Oblique data collection supplements orthogonal data collection by providing more reliable data on surfaces that are difficult to see in orthogonal surveys, such as vertical walls [34], but it can result in a lot of duplication of data and longer surveys. Regardless of the method used for the terrain surveys, we gain the raw data required to generate a synthetic data set for image segmentation.

Section 4.4.2.2: Data collection phase 2: real world potential landing sites

The data from the terrain surveys will undergo a lot of transformation and reconstruction before being used to train the landing site classifiers, and there is a risk that the classifiers will overfit to the transformed data but not generalize back to the real world. This requires a validation phase that uses real-world data. However, we found no such readily available data sets or protocols to suit our needs, given the requirements that we outline below. Further, for generality to other potential landing sites, we prefer to develop a method for generating such a data set ourselves. The safety of each landing site depends on the specifications of the drone, and collecting data in this way allows the user to specify these parameters. Our drone — a Matrice 350 — can land safely in an idealized area of 1 m radius that is essentially level and has obstacles less than 10 cm in height.

We create our validation data set using sites from both inside and outside of the terrain surveyed in step 1. We collect 10-second real-world drone videos at each site, and tag each landing site as *safe* or *unsafe*. Some of the site locations are shown in Figure 69. In each video, the drone's camera holds the validation site in the center of its FOV while moving, such that it collects many different frames from many different angles. The camera's tilt is between 45° below the horizon and vertically down. We set the validation site to be either an obstacle or a clearing and record the classifier's predictions of the center pixels of the video for all the frames, adjusting the central region's size according to the validation site's size. We tag each video as a whole according to whether it shows a safe landing site or not, and we compare our manual classification to the classifier's predictions over the frames of the video. The prediction describes the classification of the majority of pixels in the central region of the video, and we aggregate it over time to determine a simple, binary prediction for the entire video. Some of the sites are hand-selected over a range of anticipated

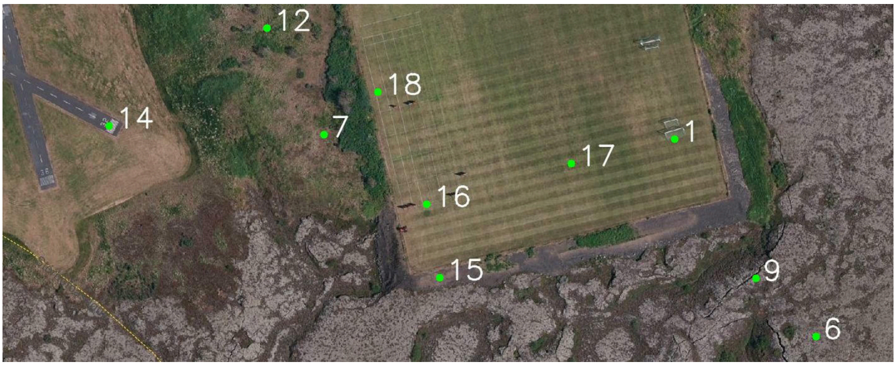


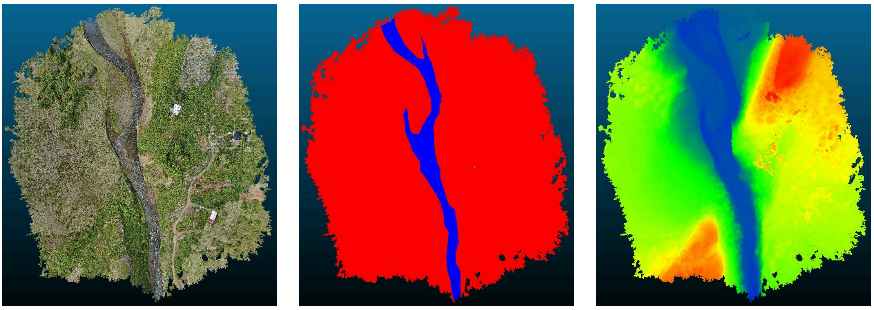
Figure 69: One of three data collection sites for validating the appearance-based landing site evaluation method. We manually and randomly picked 18 validation sites, and we number them as follows, marking safe with an S and unsafe with a U: (1U) – an archery target on a soccer field, (2U) – a bush, (3S) – a flat, dirt area, (4U) – a large, cracked rock mound, (5U) – high vegetation area, (6S) – flat, mossy area in a lava field, (7S) – flat, grassy area, (8S) – flat, mossy area in a lava field, (9U) – crack in a lava field, (10S) – dirt patch in a lava field, (11S) – road, (12U) – person, (13U) – very rough lava field, (14S) – model aircraft runway, (15U) – sloped, gravel edge of a soccer field, (16S) – green spot in a soccer field, (17S) – middle of a soccer field, (18U) – soccer goal. Map source: Loftmyndir ehf [141].

difficulties, considering the appearance-based classifier's lack of geometric understanding of the scene it is classifying. For example, we expect the classifier to easily determine that an open field is safe and that a large crack in the ground is unsafe. On the other hand, we expect that it should be hard to classify slanted dirt areas as unsafe, since they appear visually similar to the safe, level dirt areas. Some areas are chosen by randomly generating coordinates within the testing area.

Section 4.4.2.3: 3D terrain model reconstruction

The second step in generating the data set is to create an RGB mesh that is a colorized, 3D depiction of the terrain. We create this mesh from photogrammetry data using WebODM [142], or from LiDAR data by performing a Poisson reconstruction [58] in CloudCompare [143]. We cannot use the LiDAR point clouds directly because they are sparse representations of the terrain and therefore do not produce good visual representations of continuous surfaces.

The third step is to create a "label mesh" with the same topography as the RGB mesh that is marked according to which regions are safe and unsafe according to our geometric specifications. We sample the RGB mesh to create a point cloud with uniform density to calculate geometric features. This is necessary to remove "fuzziness" in point clouds derived from photogrammetry, and to remove scanning overlap in point clouds collected via LiDAR, since such variations can influence geometric features. We then calculate the normal vectors and geometric features of *verticality* – how slanted a region is – and *surface variation* – how rough a region is [144]. SafeUAV considers horizontal surfaces as safe landing sites. We go beyond this by computing a binary safety metric over the mesh, marking as unsafe all areas with verticality > 0.01, and surface variation > 0.002 (experimentally determined).



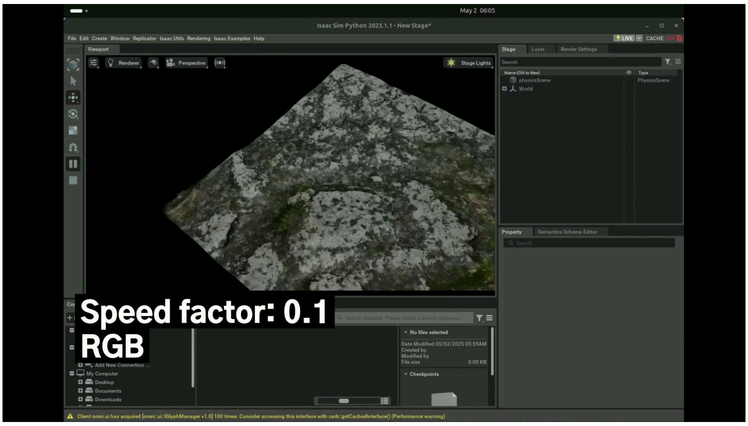
(a) RGB, unsegmented. (b) Manually segmented. (c) Altitude above sea level.

Figure 70: Example of manual segmentation of a river in the summer house dataset by isolating the river and adding a manual classification to it in CloudCompare. Figure 70a shows the RGB mesh as generated from a photogrammetry survey, and Figure 70b shows the segmented river. It is not feasible to isolate the river by simply filtering on the altitude above sea level (as shown in Figure 70c) since the terrain has a significant slope.

We further eliminate unsafe areas that have been classified as safe geometrically, because of their low verticality and surface variation, e.g., lakes and rivers, by simply selecting them in CloudCompare and adding a manual classification to all the points representing the problematic surface. This process is quick, requiring only about 2 minutes to manually label the river in Figure 70. We then apply Gaussian smoothing to the safety metric as the new grayscale texture for the label mesh. Finally, we slice the meshes into smaller, more manageable chunks to process individually on our hardware.

Section 4.4.2.4: Generating synthetic image segmentation data set

The fourth and last step in generating our data set is to create synthetic aerial images representing a drone's view of the terrain and corresponding masks that specify which pixels in those images represent safe landing sites. Using NVIDIA Isaac Sim [77], we create a scene with the RGB and label meshes at the same position and orientation, with only one visible at a time. We position a virtual camera randomly in the scene, and aim it at a random location on the meshes, ensuring that the camera is between a minimum and maximum height above the terrain and between a minimum and maximum angular deflection from vertical down. We set the RGB mesh as visible and take a picture, effectively creating a typical aerial picture of a given terrain area. Then, we set the label mesh as visible and take another picture, creating a safety mask for the RGB image. We repeat this process for each slice to generate the labeled data set as many times as necessary. Figure 71 visualizes this process and Figure 72 shows examples from a dataset, where the terrain is shown on top, and masks are shown on the bottom. White and black areas indicate safe and unsafe areas for landing, respectively. Video 4 shows the "Synthetic Data Set Generation" step of the data set creation pipeline in Figure 71.



Video 4: Data generation process in Isaac Sim. Available at https://vimeo.com/j0shua/appearance-based-classifier-dataset-generation. The video shows the creation of a dataset for image segmentation from examples of 3 "slices" of terrain. The simulator first randomly positions the camera to look at the terrain, then takes a picture of the RGB mesh, and then takes a picture of the label mesh, effectively creating a labeled data set. The first example is slowed down by a factor of 10 for clarity. The second and third are shown at full speed.

The following is a non-exhaustive list of changeable parameters which we have set experimentally and which may require special attention depending on the specific scenario. For the geometric surveys, it is important to consider primarily

1. data collection altitude (ensure it is low enough to allow for required resolution, which varies case by case), and
2. image or point cloud overlap (ensure there are no gaps between survey lines), which will significantly affect the survey quality.

For the terrain reconstruction, consider the

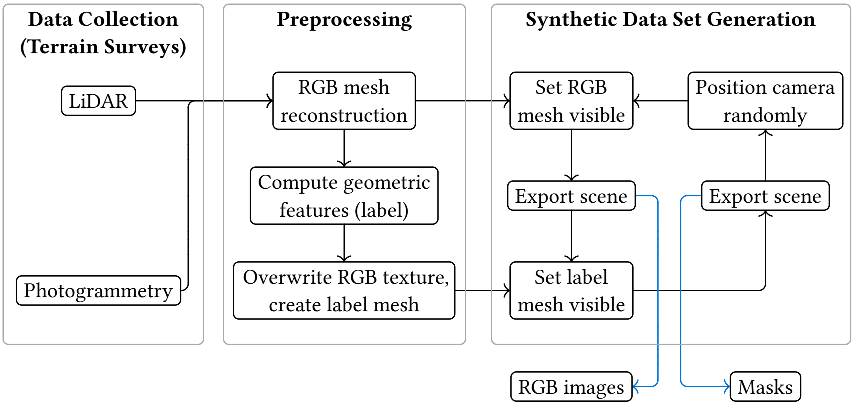


Figure 71: Pipeline for synthetic image segmentation data set generation to train the appearance-based landing site evaluation method. First, we collect data describing terrain from an analog environment using either LiDAR or photogrammetry surveys. Then, we pre-process that data geometrically to determine the locations that are safe for landing. We generate an RGB mesh to visualize the appearance of the terrain, and a *label* mesh with the same topography where safe areas are colored in white and unsafe areas in black. Finally, we import the meshes into a simulator and generate images via a synthetic camera sensor that is positioned randomly around the scene and pointing at the virtual terrain. This creates a set of synthetic images and masks, linked to in blue as the ultimate goal of the process. An example of the images and masks is shown in Figure 72.

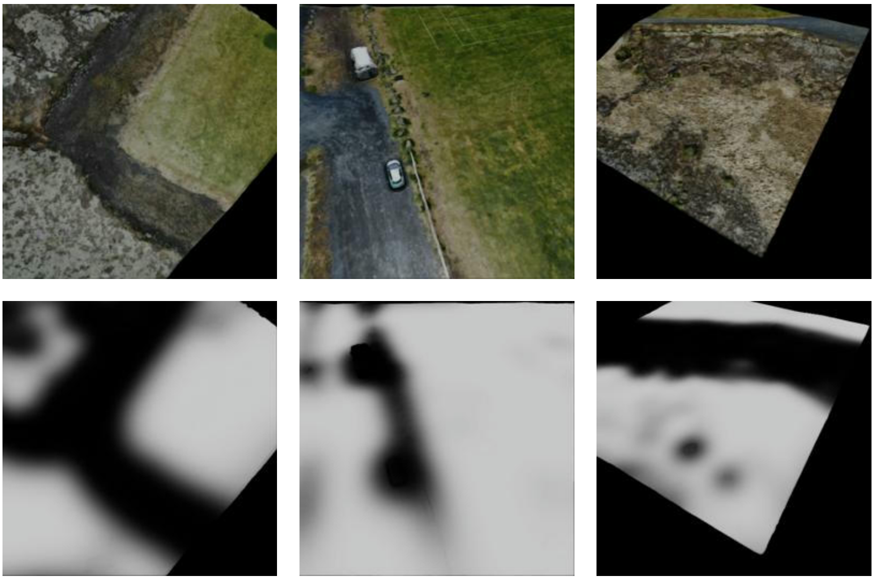


Figure 72: Example images and masks from the synthetic data set as created according to the processing pipeline in Figure 71.

3. method and radius for calculating point cloud normal vectors (a smaller radius can increase sensitivity to details in terrain formations, while a larger radius can act as a sort of low-pass filter that removes noise),
4. Poisson reconstruction radius or Octree depth (similar tradeoffs as above),
5. Poisson reconstruction minimum acceptable point density on which to generate a surface — a higher value can result in more stable surfaces that conservatively represent reality but may have holes, and a lower value can result in fewer holes but more artificial formations resulting simply from a lack of data in some areas),
6. radii and acceptable levels of verticality and planarity (can be tuned for the size and robustness of the drone),
7. terrain slice size (primarily tuned to the computer on which the data set is being generated), and
8. maximum height and deflection of the virtual camera (need to represent the end use case — altitudes and deflections that are out of the training specifications can result in consistently incorrect classifications during testing).

These will significantly affect the quality of the datasets generated, and they may vary case by case.

Section 4.4.2.5: Terrain classifiers

We use deep learning methods to generate classifiers for our image segmentation dataset described in Section 4.4.2.4. Although this is a typical segmentation problem, and there are many deep learning frameworks we could use, we are targeting a specific inference platform: the Google Coral TPU [84] because it is in the payload we developed in Section 4.3. The Google Coral imposes a particular toolchain revolving around TensorFlow [83],[85], as well as a particular set of available operations, input sizes, etc.

We test the U-Net architecture for this task based on its well-known, strong performance in segmentation tasks. However, other architectures could be applicable, e.g., auto-encoders, fully convolutional, and others. Additionally, initial experimentation revealed a few key notions. First, the little available memory on the Google Coral imposed size limitations of about 8 MB on our models and the input image resolution. Second, longer chains of operations, e.g., more sections in the U-Net architecture, significantly affected the frame rate at which the Google Coral could perform inference, even when all of the overhead operations, such as playing or resizing video and passing it to the TPU, were performed on a desktop machine and not on the more limited Raspberry Pi 5 to be used in the end product. Third, using RGB images as input to these small networks quickly resulted in them overfitting to both color and brightness. Therefore, we limit the image input size to 512x512 pixels, process grayscale images instead of RGB, and limit the number of U-Net sections to 3-4.

Section 4.4.3: Results

Section 4.4.3.1: Evaluation of synthetic segmentation dataset

We have collected terrain data in Iceland at 3 locations via photogrammetry, and at 2 locations via LiDAR. We also use two open-source photogrammetry datasets available through WebODM. Table 7 lists the data sets used, and Figure 72 shows example pairs of images and masks from the soccer field dataset. Notably, the data sets from WebODM tend to contain only orthogonal photography, such that the models they produce appear realistic from a top-down view but less realistic from an angle, especially on vertical surfaces. We note additionally that the LiDAR

	Location	Type	Sensor	Data points	Source
1	RC airfield	p.gram	DJI H20T	750	own
2	RC airfield	LiDAR	DJI L2	750	own
3	Sheffield Cross	p.gram	unknown	241	WebODM
4	Soccer field	LiDAR	DJI L2	750	own
5	Soccer field	p.gram	DJI H20T	750	own
6	Summer house	p.gram	DJI H20T	750	own

Table 7: LiDAR and photogrammetry datasets used to train our appearance-based landing site classifiers. Our own photogrammetry (“p.gram”) surveys include both oblique and orthogonal images, whereas the Sheffield Cross data set includes only orthogonal.

reconstructions tend to capture smaller obstacles and deeper cracks much more accurately and make the surveys and subsequent data processing much quicker, on the scale of minutes to hours.

Section 4.4.3.2: Evaluation of synthetic terrain classifiers

We train a 4-stage U-Net with 512x512 resolution on synthetic training and testing images for a maximum of 100 epochs. We use early stopping (monitoring the testing loss) with a patience of 15 epochs and a learning rate of $5 \cdot 10^{-5}$ with the Adam optimizer. We use categorical accuracy and categorical cross-entropy as our accuracy and loss metrics, respectively. The U-Net has four encoder steps, a bridge, and four decoder steps. The encoder block pipeline has a 2D convolution, batch normalization, leaky ReLU, 2D convolution, batch normalization, leaky ReLU, and maxpool. The decoder block pipeline has an upsampling layer, concatenation, 2D convolution, batch normalization, leaky ReLU, 2D convolution, batch normalization, leaky ReLU and maxpool. To conform to the available operations for the Google Coral, we remove batch normalization layers, replace upsampling with transpose convolution, and replace each leaky ReLU layer with a PReLU layer with an untrainable alpha parameter. We train in Keras [145] and quantize classifiers destined for the Google Coral to a TFLite model.

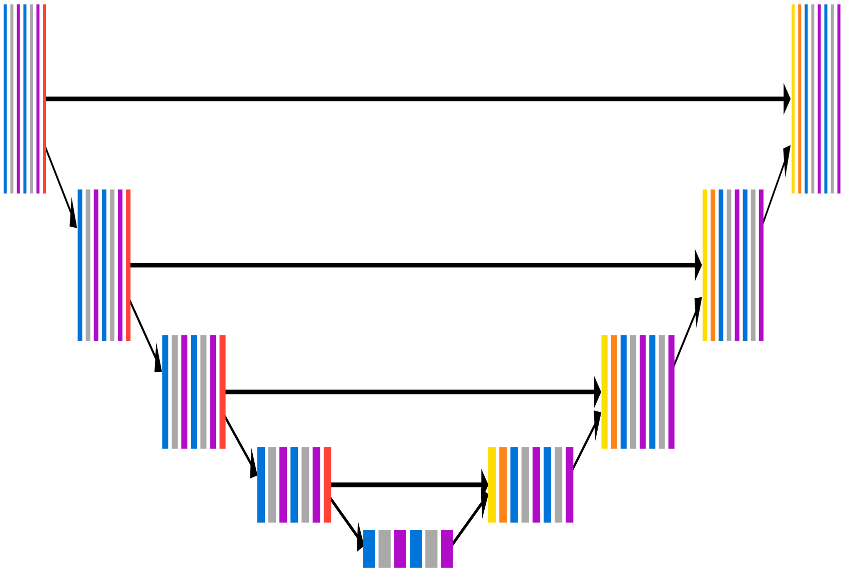


Figure 73: The U-Net architecture. We use encoders (blocks going down) with the following steps: 2D convolution (blue), batch normalization (gray), leaky ReLU (purple), 2D convolution (blue), batch normalization (gray), leaky ReLU (purple), and maxpool (red). We use decoders (blocks going up) with the following steps: upsampling layer (yellow), concatenation (orange), 2D convolution (blue), batch normalization (gray), leaky ReLU (purple), 2D convolution (blue), batch normalization (gray), leaky ReLU (purple) and maxpool (red). The bridge (bottom) is the same as encoder without the maxpool. When exporting to the Google Coral, we remove batch normalization layers, replace upsampling with transpose convolution, and replace each leaky ReLU layer with a PReLU layer with an untrainable alpha parameter.

Classifier	Accuracy		Loss		Real world	
	Training	Testing	Training	Testing	V1	V2
Best U-net	0.667	0.815	0.613	0.373	0.778	0.833

Table 8: Accuracy, loss, and real world validation accuracy for our best appearance-based landing site classifier. V1 and V2 are the proportions of validation sites correctly classified with naive thresholds and manually tuned thresholds, respectively. For V2, the safety threshold is 0.6 and the danger threshold is 0.4.

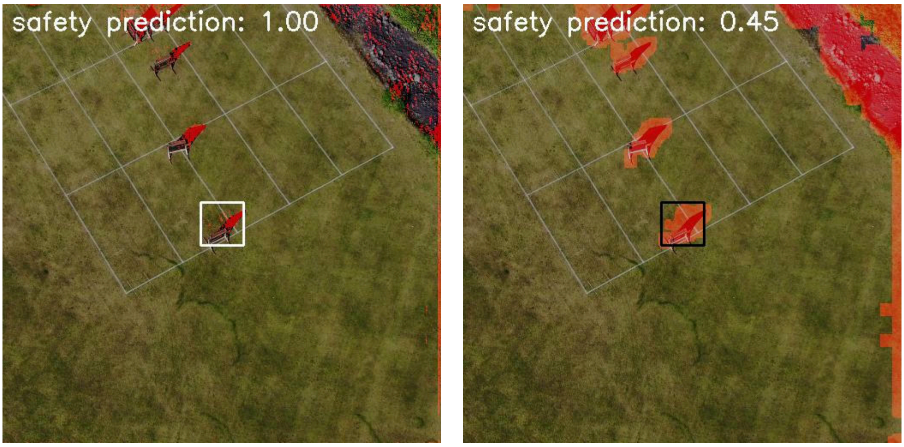
This requires full quantization to unsigned 8-bit integers, instead of the standard 32-bit float and also requires a representative data set to tune the network during the conversion (we use a random selection of the training images as a representative data set). The last step is to use the edge TPU compiler to convert the TFLite model into one compatible with the Google Coral. For each experiment, we select a subset of the datasets in Table 7 for training and testing. We run each experiment 10 times, taking the model with the lowest testing loss. This is feasible because the model is small, and the training time is 10–20 minutes. The result is presented in Table 8.

Section 4.4.3.3: Post processing

We cannot use the output from our classifier directly because it is too sparse. We therefore add some post-processing steps. Figure 74 shows an example prediction from the network at the 10-second mark (the last frame) of a video from the validation set. The network outputs two masks: one for safety and one for danger. For clarity, we only show the danger masks (red) and other parts can be assumed safe. Setting thresholds for converting these to a binary mask is yet another parameter; we conduct a coarse parameter sweep, with the safety threshold θ_s being $[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]$ and the danger threshold being $1 - \theta_s$. The central square region represents a candidate landing site that we want to evaluate – in this case, an archery target (unsafe for landing) – and the evaluation is shown in the top left of the image. As explained in Section 4.4.2.1, this evaluation is determined by whether the majority of pixels in the square are considered safe or unsafe. The square is white if the area is considered safe and black if it is considered unsafe at the particular instant. Although many of the pixels in the box in Figure 74a are red, they are too sparse to reject the site – this can be seen in the safety prediction of 1.00, indicating that it has been deemed safe throughout the entire video. In the spirit of erring on the side of caution, we would like to reject unsafe landing sites despite the typical sparsity of unsafe classifications. We thus propose two post-processing enhancements: E1 is a box blur with the job of patching holes between rejected regions to create contiguous rejected regions, and E2 is temporal smoothing, with the job of keeping track of temporally sparse unsafe classifications. These are shown to reduce the safety prediction in Figure 74b and Figure 74c respectively. We experimented with values ranging from 1–10 frames for the temporal history and 7–19 for the box blur kernel size, ultimately choosing a box blur kernel size of 15 for E1 and a temporal history of 5 frames for E2. While E1 and E2 alone both reduce the safety prediction, only the combination of both completely rejects the unsafe landing site with a prediction of 0.00, as shown in Figure 74d.

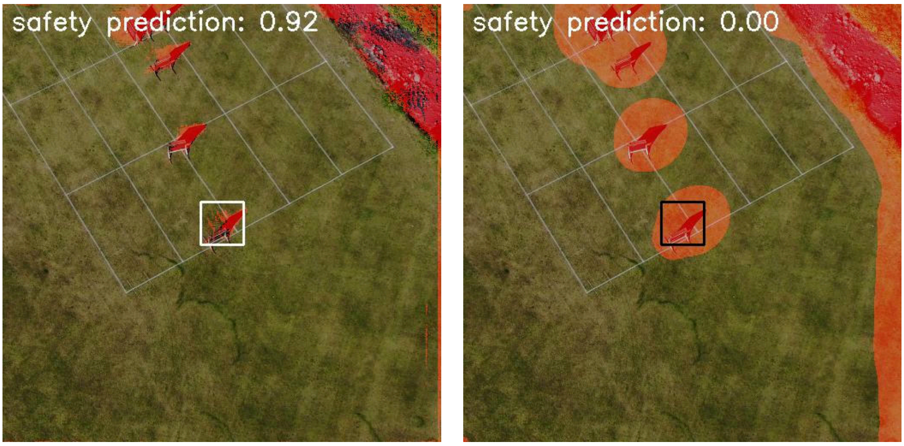
Section 4.4.4: Real world evaluation

Using the landing site safety prediction method defined in Section 4.4.3.3, we conduct a validation phase for the most promising classifiers. These results are



(a) Raw output.

(b) E1: spatial smoothing (box blur).



(c) E2: temporal smoothing.

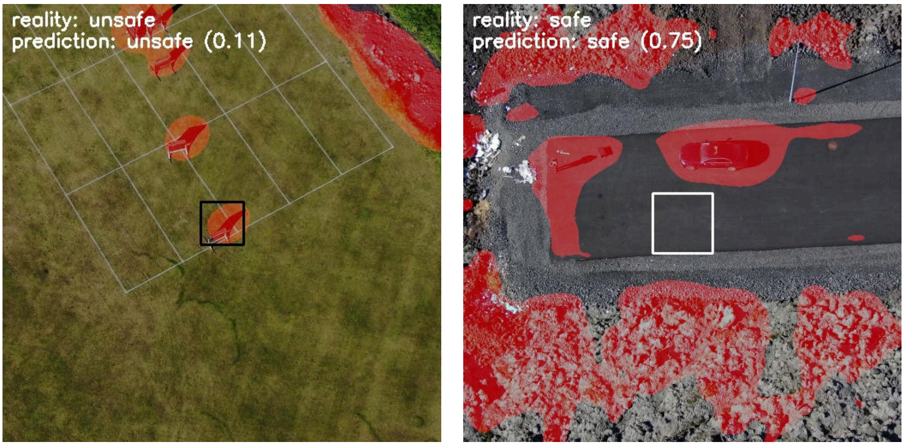
(d) E1 & E2.

Figure 74: Effects of post-processing the predictions from our terrain classifier. Each picture shows the last frame of the same 10 second video with different stages of post-processing. The central square region is the particular area on which we want to generate a binary classifcation of `safe` (white square) or `unsafe` (black square) after processing the video for 10 seconds, during which time the camera is tracking the same region. The classifier is, however, classifying each pixel of the image, and those classified as safe are unlabeled in the output, whereas those classified as unsafe are marked in red. The safety prediction is a number in the interval $[0, 1]$, and the site is considered safe if the safety prediction is greater than 0.5. The raw predictions are accurately pointing out unsafe archery stands and the rough area in the top right, but they are too sparse to classify the central region as unsafe by majority vote. We add spatial smoothing (E1) in Figure 74b and temporal smoothing (E2) in Figure 74c. After combining them, we end up with the smooth predictions in Figure 74d which give reasonable aggregate classifications on the central region.

presented in the last two columns of Table 8 – V1 represents the validation performance with safety and danger thresholds of 0.5, and V2 represents the validation performance with thresholds tuned through a coarse parameter search. We compare the network's aggregated prediction to our knowledge of the area, obtained by going to the area in person, and determine the network's successful prediction rate over the validation locations in Figure 69. Overall, the best U-net correctly classifies 15 out of 18 videos depicting each validation site, i.e., it achieves an accuracy of 0.83, which we consider promising. The classifiers can run at a rate of about 3.4 Hz on the drone payload, which is fast enough for our task given that we can add filtering methods to make the approach smooth.

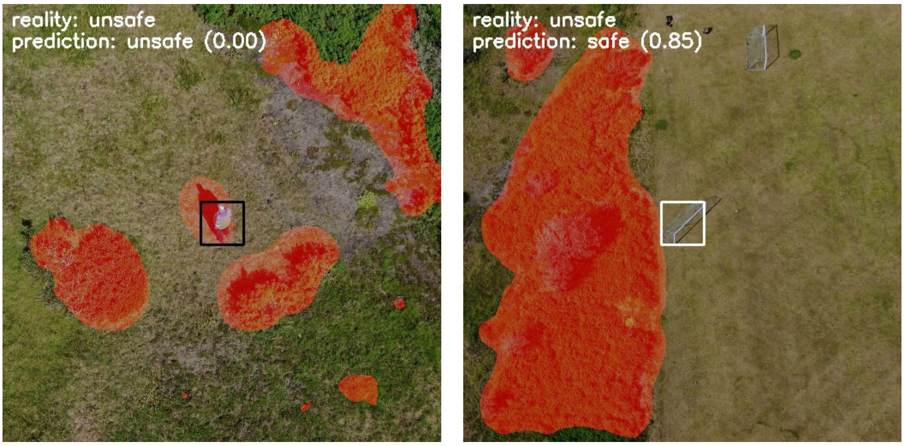
We show and explain the full set of validation sites in Figures 75, 76, and 77. thematically (out of order). Each of the images represents the inference provided by the classifier, aggregated over 10 seconds of video with the post-processing steps described in Figure 74. The pixel size of the target area, denoted by the black or white square in the middle, is 100x100 pixels by default, but we occasionally increase its size slightly to fit larger potential landing sites. We mark each image with the real safety classification (determined by us as oracles), and the classifier's prediction from 0 (least safe) to 1 (safest). The archery targets shown in Figure 75a (and earlier in Figure 74d) represent what we anticipated to be a relatively straightforward validation for the classifier to handle, since the field is generally monotonic and the archery targets stand out as anomalies. This makes it a good top-level demonstration for the classifier, primarily to show the post-processing steps.

Similarly, the classifier distinguishes the safe road from unsafe cars, people, and lava in Figure 75b, additionally rejecting the rough outer area which is an old lava flow. In Figure 75c, the classifier has rejected a person standing in the middle of a generally safe area with some unsafe bushes and other small obstacles. Figure 75d shows an incorrect classification of the soccer goal as safe, though we anticipated it to be clearly unsafe. The classifier also views the salient, white runway markings in Figure 75e as safe, correctly. A particularly green spot in the middle of the soccer field, shown in Figure 75f, is partially classified as unsafe, but ultimately the unsafe region is not large enough for the classifier to reject it. The classifier generally determines that more monotonic regions, such as the middle of the soccer field in Figure 76a, are safe. However, the sloped, gravel edge of the soccer field, shown in Figure 76b – which is absolutely not a safe landing site – is also quite monotonic and is erroneously classified as safe, representing a primary challenge of using an appearance-based method for this task. Figures 76c and 76d show that the classifier can distinguish between short and tall vegetation, correctly classifying the bushes and other high vegetation as unsafe, while classifying the grass as safe. The flat, dirt area in Figure 76e is correctly classified as unsafe, while the rockier and mossier areas around it are considered safe. Figure 76f represents an erroneous classification of a flat, grassy area as unsafe, however, this is likely because of issues in the



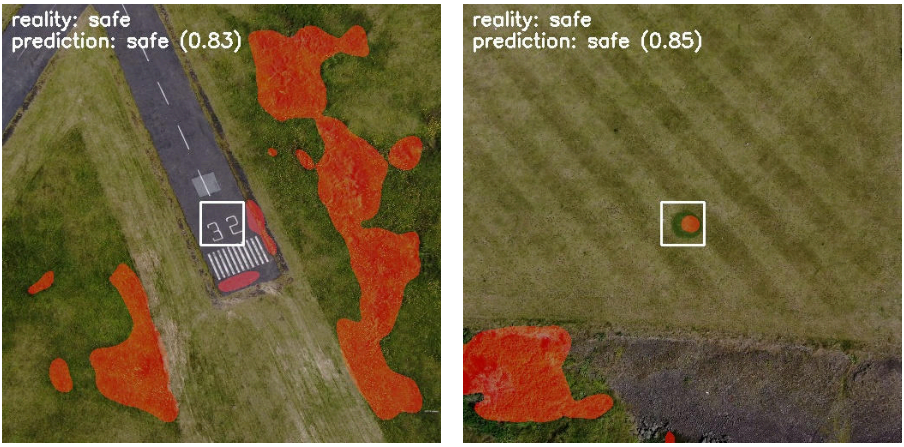
(a) 1U: archery target on soccer field.

(b) 11S: road.



(c) 12U: person.

(d) 18U: soccer goal.



(e) 14S: model aircraft runway.

(f) 16S: green spot.

Figure 75: Validation sites depicting mostly suburban infrastructure. Site 18U is classified incorrectly, however, we verified that this is a result of the scene being too far away for the classifier to resolve. As the drone moves closer, the soccer goal is correctly classified as unsafe. The rest of the classifications are correct.

automatic aiming of the camera, which is pitching up and down slightly (this can be seen in the video). The target area is slightly above the central region.

Figures 77a and 77b represent different flat, mossy areas, which are both classified correctly as safe. Figure 77c shows a correct, safe classification of a dirt patch in a lava field, and Figure 77d shows a correct, unsafe classification of a rough lava field. Figure 77e and Figure 77f show correct, unsafe classifications of cracks in lava fields.

There were some important trends in the real-world evaluation. First, the classifiers almost universally classified a safe runway as unsafe and the unsafe, slanted gravel edge of a soccer field as safe. This seems to be a limitation of the fact that the classifier is appearance-based; the straight, monochromatic white runway markings seem to appear as tall structures, and the gravel does not change appearance significantly when slanted compared to when it is level (e.g., in a parking lot). Further, orientation and altitude are important factors, and performance is unsurprisingly much better if these are within the ranges of the synthetic training data, i.e., between 45 degrees below the horizon and vertically down, and at a distance of between 5 and 20 meters from the target. For example, many classifiers correctly classified tall vegetation as unsafe when adequately close, but as safe when farther away. Finally, the small networks occasionally experience a “burn-in” as a result of saturation in the training process, where they produce persistently safe or unsafe classifications in particular locations of the image, regardless of the content of the image. This is most likely attributable to (1) disadvantageous initial conditions in the training process, and (2) biases in the data set, e.g., the fact that edges of the images are often unsafe as a result of being outside the generated terrain area, as shown in Figure 72. We show the post-processing steps and the appearance-based classifications in order in Video 5.

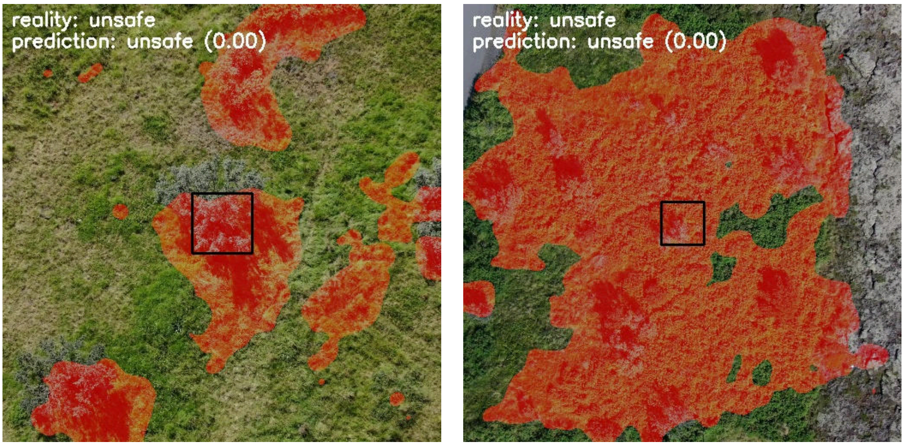
In addition to lab tests, we conducted flights at multiple validation locations in the testing area, with the method running onboard the drone. An example of successful onboard inference via a quantized TFLite model on the Google Coral is shown in Figure 78, where the system is marking the grassy field as safe and the archery stands as unsafe. Of course, this does not tell the whole story, as demonstrated in Figure 79, which shows that, at higher altitudes, the system does not pick up the archery stands as obstacles. This is a testament to 2 things. First, the low resolution of the video input to the classifier means that it is simply difficult to resolve objects from far away. Second, the limited height ranges in the training data mean that the classifier may not recognize an object as unsafe even if it can resolve it visually, if it has not seen something similar labeled as unsafe in the training data. The code for this project is available on GitHub [146].

Section 4.4.5: Conclusion

We presented a pipeline for generating terrain classifiers to find landing sites for a drone autonomously by analyzing video from our drone’s RGB camera. We auto-

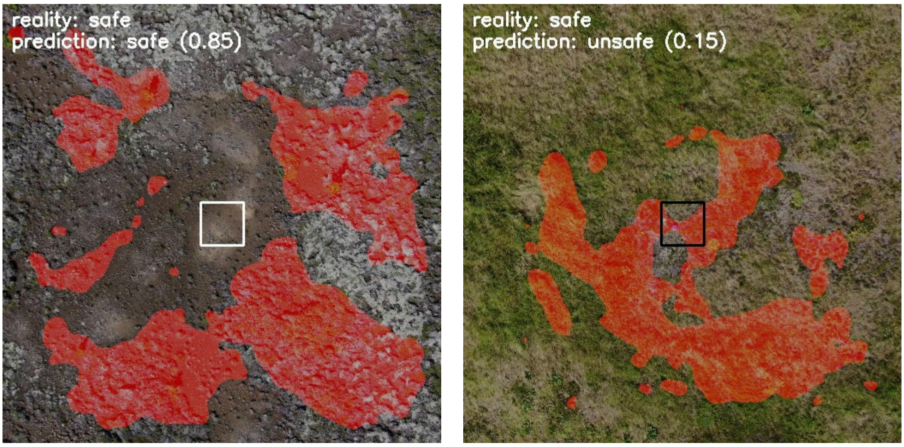
(a) 17S: middle of soccer field.

(b) 15U: sloped gravel edge of soccer field.



(c) 2U: bush.

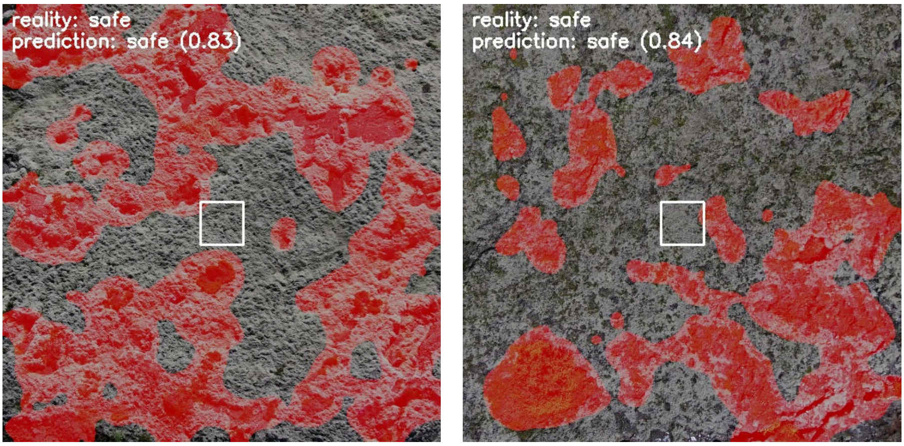
(d) 5U: high vegetation area.



(e) 3S: flat dirt area.

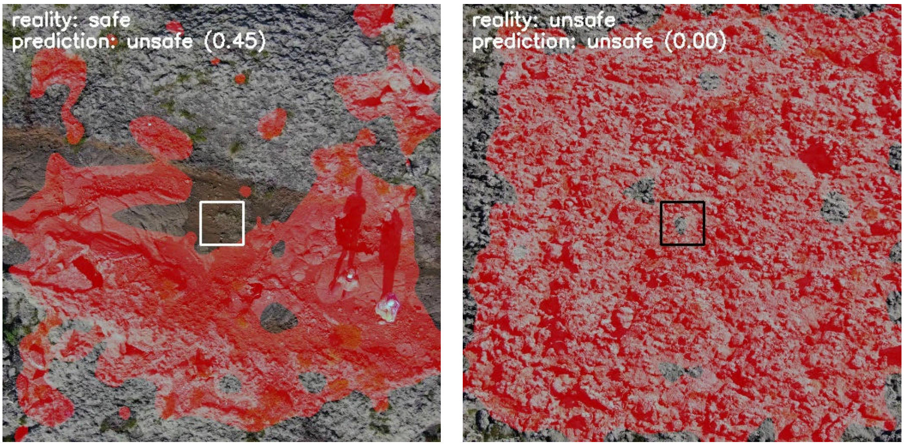
(f) 7S: flat grassy area.

Figure 76: Validation sites depicting natural surfaces. Site 15U is classified incorrectly because its appearance is too similar to that of flat gravel, although it is slanted. This is a basic limitation of appearance-based methods. Site 7S is also classified incorrectly, although this is likely because the camera veers upward erroneously while trying to track the lower, unsafe site (see the validation video). The rest of the classifications are correct.



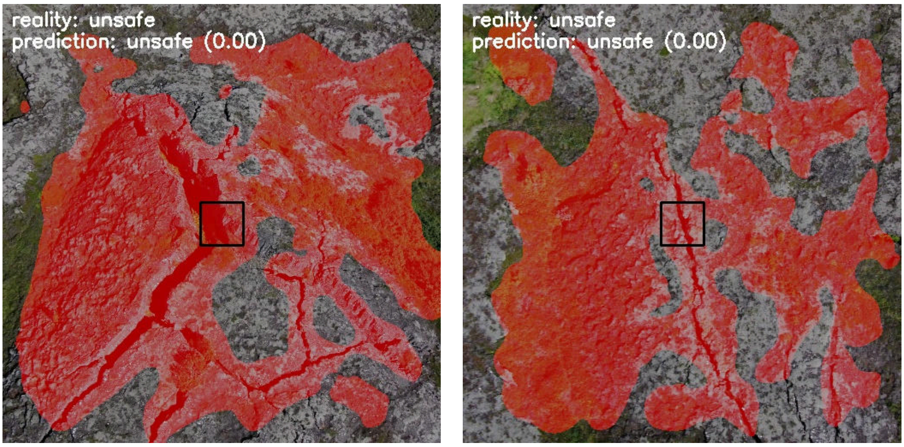
(a) 6S: a flat mossy area.

(b) 8S: a flat mossy area in a lava field.



(c) 10S: dirt patch in lava field.

(d) 13U: rough lava field.



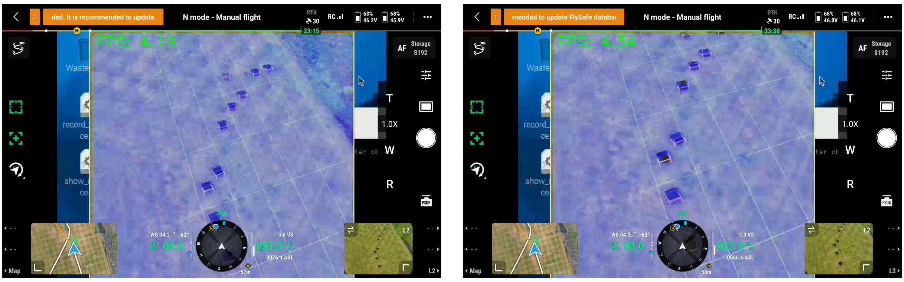
(e) 4U: large cracked rock mound.

(f) 9U: crack in lava field.

Figure 77: Validation sites depicting lava fields. Site 10S is classified incorrectly, although this seems to be another issue of the camera not tracking completely accurately, and the site is classified correctly as the video goes on. We leave it in anyway to not introduce our own biases. Although it is classified overall as unsafe, the target square is white because the site was classified as safe instantaneously during the last frame of the video. The rest of the classifications are correct.

Figure 78: Usable inference made on the Raspberry Pi 5 payload with the Google Coral TPU in flight, and viewed in real time on the controller. The system correctly identifies the grassy field as safe, ignoring the unnatural straight lines and right angles, while rejecting the unsafe areas (archery stands put on their side). The payload is running inference at a rate of about 4 frames per second, as shown in the top left.

matically generated a synthetic dataset for image segmentation by reconstructing



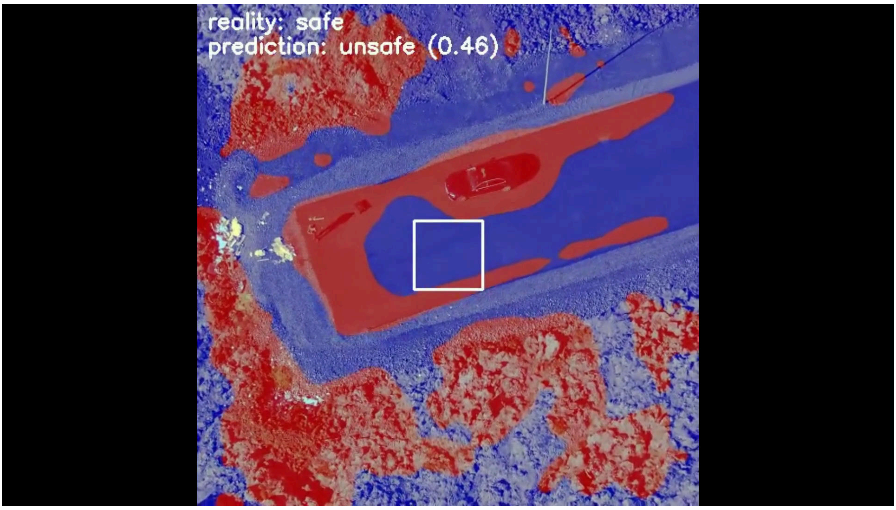
(a) 35.5 m altitude.

(b) 27.0 m altitude.

(c) 20.7 m altitude.

(d) 16.4 m altitude.

Figure 79: The effect of distance on the performance of the classifier in inferring which areas of the scene are safe and unsafe. From an altitude of 35.5 m, the system erroneously views the whole scene as safe, but it gradually becomes more accurate as the drone descends to an altitude of 16.4 m. This is a consideration when determining how to interpret the output from the classifier, which must be aggregated over time, potentially changing view angles and altitude.



Video 5: Demo of appearance-based landing site classification for the real world validation sites. Available at https://vimeo.com/j0shua/appearance-based-classifier-demo. The video starts by showing the effects of the pre-processing steps, which refine the sparse output from our classifier into a more usable form. Areas highlighted in red are classified as unsafe, and those in blue as safe. The video then iterates through each of the validation sites showing the output of the classifier. A square in the middle marks the location being classified, and it is colored in black if the area is instantaneously deemed as unsafe, and white if safe.

the terrain surveyed by the drone, labeling the reconstructed terrain geometrically for landing safety, and generating images and masks in simulation. We used both photogrammetry and LiDAR datasets and used one openly-available photogrammetry dataset from WebODM to show the method's flexibility. We trained a U-Net on the synthetic dataset and quantified its performance in the real world by performing inference on 10-second drone videos of 18 known-safe and known-unsafe validation sites. The U-Nets correctly classified a maximum of 15 validation sites. Finally, we ran the method in real-time onboard a drone equipped with a Raspberry Pi 5 and Google Coral TPU, motivating our creation of a very small classifier.

This project provided insight to **RQ2** by demonstrating a visual classifier that can evaluate landing sites based on their appearance, requiring just RGB images. Of course, it is more subject to inaccuracy than a geometric method, and therefore must be investigated further. This project also provided insight into **RQ3** by showing that the method can be small enough to run onboard the drone.

Future work should include collecting data from more real-world environments, and generating more synthetic data sets for training. More classifiers should be tested, e.g., auto-encoders, fully convolutional networks, and other segmentation methods such as support vector machines – with consideration of the constraints of embedded hardware so they can be embedded onboard a drone. It is also a point of interest to determine whether there is a difference in performance to be gained by either photogrammetry or LiDAR over the other. Finally, we will adapt this method so that it can generate commands to control the drone and execute the landings on its own, similar to the method in Section 3.7.

Section 4.5: Tying it all together

In this section, we describe work towards formally testing the geometric landing site detection method first outlined in Section 4.2, as well as the appearance-based method in Section 4.4 in a newer lava field in the recent eruption area near Grindavík. Here, the idea is that the appearance-based method can be used for longer-distance, initial identification of potential landing sites, and the geometric method can give a better final decision on whether or not to commit to the landing, once the drone has approached. Because of time constraints, we tested prototypes of the methods online, i.e., in the field, and we tested the full versions offline, i.e., in the lab using data collected from the lava flows. We have constructed a payload to run both solutions simultaneously, based on observations from the previous payloads. However, as testing proceeded, we noticed that the geometric evaluation method required slightly more computational resources than before. This is because we have adapted it to the Matrice 350 (which we did not have during the work from Section 4.2), which provides only the raw stereo images and calibration parameters – not the full depth map provided by the Intel RealSense D455. We therefore have to additionally consider the non-trivial calculation of the depth map ourselves. Finally, the Matrice 350 has stereo depth cameras on each of its 6 sides for obstacle avoidance, which provide a sort of landing site evaluation method. However, their method is a black box that is difficult to safely validate, and it seems that the only way to evaluate its performance would be to attempt to carry out landings and check empirically if it refuses to land or not. We avoid this and simply test our own geometric evaluation method instead.

Section 4.5.1: Combined payload design

This payload uses two computational boards for isolated processing. The first is a Raspberry Pi 4, which serves as the connection to the drone. It has connections to both the E-port and PSDK port to access the drone and other payloads and to stream video to the controller in real time, similar to the payload in Section 4.3. The payload requires both the video from the gimbal-mounted camera and synchronized image pairs from the downward-facing stereo depth cameras. Since the H264 video stream from the L2′s RGB camera has lower resolution than that of the H20T, the Raspberry Pi 4 can decode it in hardware, saving a huge amount of CPU time and motivating the choice against using a Raspberry Pi 5 as in Section 4.3, as it can only decode H265 in hardware. The Raspberry Pi further subscribes to the raw stereo depth images, as the Matrice does not provide the processed depth map over the PSDK API. The subscriptions to both the RGB video stream and the stereo stream are activated on

Figure 80: The Jetson Nano payload and constituent components. This is a confined space with a lot of electronics and wires, which required tedious and careful placement. In the top right is the Raspberry Pi 4 which runs two PSDK applications, connecting on the PSDK Port and the E-port, similar to the Raspberry Pi 5 payload mentioned earlier. It streams its desktop to the controller via the PSDK Port. Via the E-port, it retrieves the video feed from the RGB camera on an L2 payload and decodes it in hardware, as it is adequately low resolution. It connects to a Jetson Nano on an A203 carrier board in the center (with the black heatsink) via a USB OTG connection. The Jetson Nano carries out inference via our visual classifier and performs terrain analysis using the Matrice's built-in, downward-facing stereo depth camera. The E-port expansion board is on the right (with a DC-DC converter beneath it), and the PSDK Port expansion board is on the left. The wires to connect to the drone route through the circular holes on the left and right of the payload. The payload connects with the same quick-release mechanisms as shown with the earlier payloads (see Section 3.6). The top, rear of the drone can be seen at the bottom of the picture. On the leftmost and rightmost sides of the fuselage are the left and right monocular cameras forming the upward stereo depth camera for obstacle avoidance. There is one such stereo depth camera on each side of the drone's fuselage.

startup and both run continually throughout the drone's flight, as we have found that dynamically activating and deactivating them to reduce computational effort makes the system unstable and prone to crashes. The Raspberry Pi encodes its own screen and streams it to the controller in real time according to the same method as in Section 4.3.

The Raspberry Pi connects to a Jetson Nano, which has GPU acceleration to run the programs to analyze the depth images of the scene below the drone and to conduct inference on the RGB video using a visual classifier trained according to the method in Section 4.4. The Raspberry Pi connects to the Jetson Nano via the OTG port on an A203 carrier board [147], which creates a network interface that we use for communication via ROS to send image topics. The carrier board provides more space-efficient interfaces for delivering power and data to the Jetson Nano, but is actually meant for the Jetson Nano production model, rather than the Jetson Nano development kit that we have, and therefore most of the interfaces on it are not functional. This is not prohibitive because the OTG port and power pins are all we need to use. The payload contains the necessary development boards for the E-port, PSDK port, DC-DC converter, SkyPort V2 adapter and gimbal assembly (disassembled to save space), and the companion boards, as shown in Figure 80.

Section 4.5.1.1: Power consumption

We summarize the power usage of each of the computational components in Table 9, under the assumption that the boards are running at maximum power consumption constantly. Table 9 shows that the consumption is generally small (< 5%) in each case. For the current payload, which uses a Jetson Nano and Raspberry Pi 4, the consumption would be about $2.14\% + 1.43\% = 3.57\%$.

Section 4.5.1.2: Software

The Raspberry Pi 4 runs both PSDK applications and publishes image topics via a ROS 1 network connection to the Jetson Nano, which runs nodes to process them immediately. The Raspberry Pi selects which solution to run based on the topic it publishes. This setup makes switching between solutions fast and predictable. The Raspberry Pi publishes RGB images for inference as a single image topic, and

	RPi 4	RPi 5	Jetson Nano	Google Coral
Maximum power draw (W)	15	25	10	2
Energy consumption over 45 minutes (Wh)	11.25	18.75	7.5	1.5
Portion per flight	2.14%	3.56%	1.43%	0.29%

Table 9: Worst-case (maximum) energy consumption of each of the companion boards over a typical, 45-minute flight.

publishes depth images as synchronized pairs. For classifying the scene visually, the inference server on the Jetson Nano loads the classifier, carries out inference according to the method in Section 4.4.2.5 and publishes the segmentation mask. Since the Jetson Nano is somewhat old and needs an older version of TensorFlow, it loads the classifier from a protocol buffer (`protobuf`) for compatibility.

For dealing with the stereo depth images, the process is not as straightforward as in Section 4.2, as much of the lower-level processing is done natively on the depth camera itself, whereas the Matrice provides only raw, timestamped stereo images and the intrinsic and extrinsic parameters of the depth camera. To complicate matters slightly, the left and right camera sensors in this setup are oriented such that the horizontal dimension is in line with the axis through the drone from front to back. This makes sense because, since the images from these cameras are wider than they are tall, they would otherwise be blocked on the edges by the drone's landing skids, so this orientation maximizes the usable area in the images. However, existing OpenCV implementations for disparity calculation – one of the main steps in calculating depth from stereo images – require that the images should be positioned side-by-side, rather than the resulting vertical arrangement in this setup. This requires transposing the images and camera parameters. From here, we calculate the disparity and then depth using conventional methods in OpenCV. Similar to Section 4.2, the depth camera does not have stabilization, but is rather attached rigidly to the drone. However, in this case, we take the fused orientation estimate from the drone itself, which provides it as a data topic, instead of calculating it ourselves as before. Finally, we use the depth to conduct a final check as to whether the scene beneath the drone is safe for landing, since Section 4.4.3 demonstrates that the appearance-based classifiers can be mistaken. Indeed, stereo depth cameras can also be mistaken if the scene is not clear enough or is too self-similar, but this still offers a good sanity check on the appearance-based classifier's prediction.

Section 4.5.2: Methods: combined landing site classification

We conduct a small experiment similar to that in Section 4.4.4 to evaluate the appearance-based classifier from Section 4.4 and the geometric method from Section 4.2 on a new lava site near Grindavík. We train the appearance based classifier according to the method from Section 4.4 using new data from an orthogonal LiDAR scan we conducted over an area of roughly 0.88 km^2 just north of Sýlingarfell. We conduct a sanity check at 3 of the sites from Section 4.4.4 for comparison. We choose 9 further sites at the new eruption site near Grindavík (Klifhólahraun) in April 2025, limited primarily by logistical, time, and weather considerations. To test the appearance-based method, we collect videos of the sites between 20 m and 25 m altitude and less than 25 m horizontal distance from the point, using the POI tracking function on the H20T to automatically aim the camera at the landing site. We cut these videos to 10-second clips for evaluation by the appearance-based classifier.

Once directly over the site, having already collected video for the appearance-based method, we begin to collect image pairs from the downward-facing stereo depth camera and the IMU data simultaneously, while descending from 20 m altitude to 2 m altitude. We collect this data as a ROS `.bag` file for offline processing. We determine the vertical distance to the ground according to the method in Section 4.2 and we add the following decision logic to make an automated assessment of the landing site's viability.

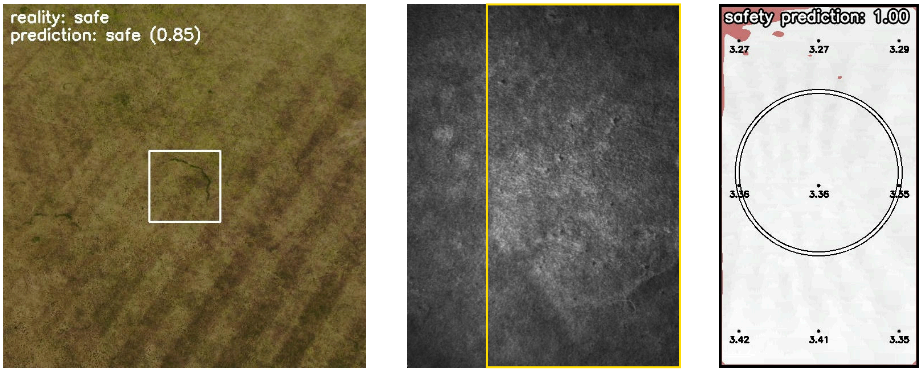
1. We flood-fill an array representing the vertical distances to the ground with a tolerance, using the center of the depth image as the seed point. We choose a tolerance of 10 cm because previous landings demonstrate that this level of terrain variance is safe for our Matrice 350. This gives a mask describing a contiguous, safe area in the depth image. We filter this mask spatially and temporally to remove noise.
2. We move on to projecting a circle with a physical radius of 1 m into the depth image, which we use as the landing area for the Matrice. We choose a radius of 1 m because it provides sufficient clearance on all sides of the drone. To do this, we calculate a functional altitude over ground by averaging the vertical distances to the points in a central region of the depth image. Using the intrinsic parameters of a model describing the depth camera, we can determine the pixel radius of a circle with a physical radius of 1 m, and we can begin to place this circle in the mask.
3. We place the circle representing the landing site at the area in the mask that is farthest from `false` values, since this also gives the most possible clearance. We filter the location of the circle temporally to reduce noise.
4. To create a prediction of landing safety, we determine the number of frames for which 90% of the circle's points overlap with `true` values in the mask, implying the area is geometrically safe, and then divide by the total number of frames over which we carry out the evaluation. Since the depth camera is subject to noise and distortion – especially at long distances – we experimentally determine a maximum and minimum height over which to carry out this evaluation during the descent. It is not useful to start the evaluation prematurely, as the depth camera is unable to resolve the scene with enough resolution from too high an altitude, and similarly cannot resolve the scene with enough (or any) resolution from too low an altitude. Therefore, we have to find a workable middle ground.

Section 4.5.3: Results

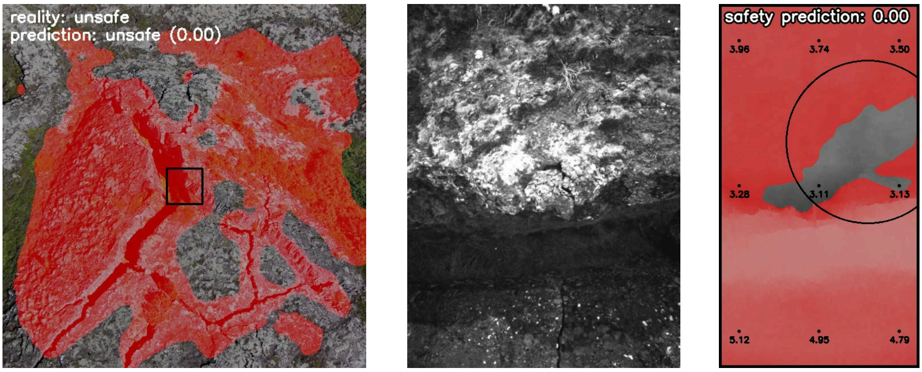
Section 4.5.3.1: Combined landing site evaluation

We conducted a test of the combined landing site evaluations at the soccer field site from Section 4.4, and the performance is shown in Figure 81. The format for each site is as follows: each site gets a line of 3 images: the appearance-based evaluation, the image from the left, downward-facing stereo camera at an altitude of approximately 3.5 m above the site, and a labeled depth image (from left to right). The appearance-based evaluations are conducted as in Section 4.4, with the central square in white or black illustrating whether the landing site is safe or unsafe, respectively. The stereo images primarily serve to illustrate the difficulty of determining subtle features such as slant using a grayscale monocular image, even from a low altitude. The yellow rectangle in Figure 81b outlines the approximate portion corresponding to the depth image in Figure 81c, which is consistent for all further examples. The depth images visualize the vertical distance from the drone to the ground and are annotated with 9 points specifying the distance in meters with text. They are further annotated with a circle representing a potential landing area with a radius of 1 m, and the circle is white-filled or black, denoting whether the site is safe or unsafe, respectively. Aggregated safety predictions are at the top of the images and run from 0 (unsafe) to 1 (safe), with a decision threshold of 0.5. For both methods, areas masked in red are classified as unsafe, and unmasked areas are classified as safe. After examining the depth camera's ability to resolve the scene over our range of altitudes (from 20 m to 2 m), we set the safety evaluation to occur on each frame between 5.5 m and 3.5 m. The screenshots of each of the depth images are taken just after the lower limit is reached, and the final safety prediction is given.

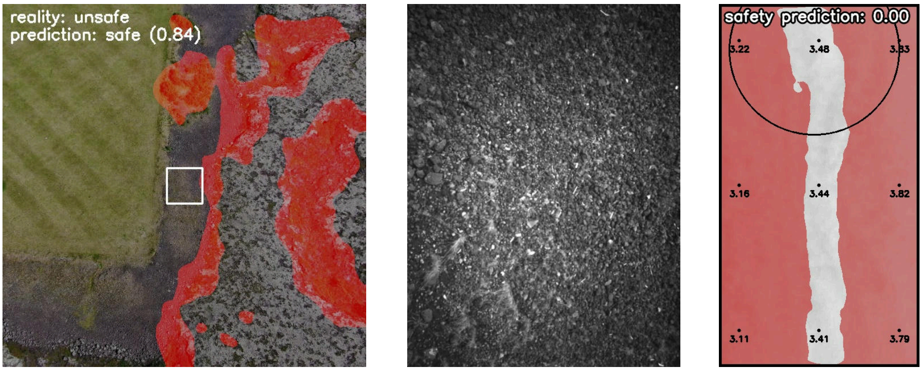
Figures 81a, 81b, and 81c describe site 17S – a flat, level soccer field – which is a good test case for sanity checking. The evaluations both show correct safe classifications of the site, although the geometric evaluation has erroneously marked areas in the top left as unsafe. This is likely a result of the IMU data being slightly out of sync with the stereo images, such that the extremities of the depth image show slight, false variations, since the windy weather meant that the drone was constantly correcting its orientation to stay stationary. Figures 81d, 81e, and 81f describe site 4U – a large, cracked mound in a lava field. The entire scene could not be contained in the stereo images at the required 3.5 m altitude, so only one of the cracks appears in Figures 81e and 81f. Both the appearance-based and geometric evaluations correctly classify the site as unsafe. Figure 81e shows the deep crack at the bottom and a sloped edge on the top, and Figure 81f shows that there is a small area in the scene with less than 10 cm of elevation change from the central point, but this region is not large enough to land on, as shown in that a large portion of the interior of the black circle is considered unsafe. Figures 81g, 81h, and 81i describe site 15U – the



(a) Site 17S: appearance-based method. (b) Closeup (alt. approx. 3.5 m) (c) Geometric method.



(d) Site 4U: appearance-based method. (e) Closeup (alt. approx. 3.5 m) (f) Geometric method.



(g) Site 15U: appearance-based method. (h) Closeup (alt. approx. 3.5 m) (i) Geometric method.

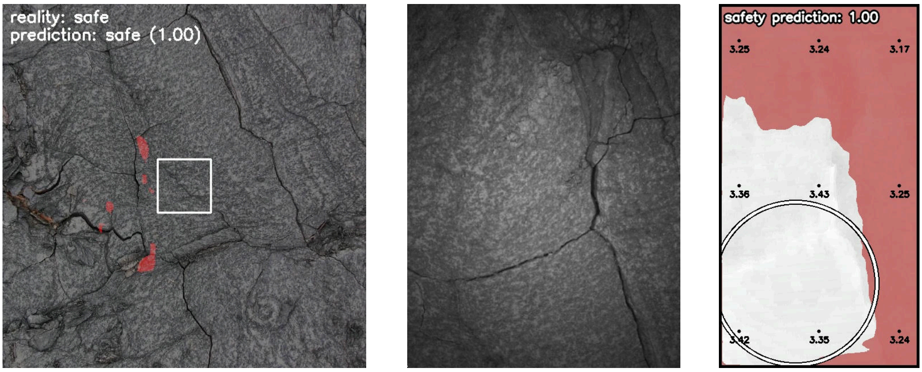
Figure 81: Combined appearance-based and geometric landing site evaluations at sites from Section 4.4.4. From left to right, each line shows the appearance-based evaluation, the downward-facing (left) stereo image at approximately 3.5 m altitude, and the geometric evaluation overlaid on top of the corresponding depth image. Areas masked in red are classified as unsafe, and unmasked areas are safe. The yellow rectangle in Figure 81b marks the portion corresponding to the depth image in Figure 81c, which is consistent for all further examples. Depth images textually specify the distance to the ground over a grid of 9 points. White and black indicate safe and unsafe respectively, for the target regions (squares and circles) in the both evaluations. Geometric evaluations have target regions denoting a circle of 1-m radius. Figures 81a–81c show safe classifications of site 17S – a flat, level soccer field. Figures 81d–81f show unsafe classifications of site 4U, a large, cracked mound in a lava field. Figures 81g–81i show contradictory classifications of site 15U, a gravel slope next to a soccer field, illustrating how the geometric method precludes dangerous landing even when the appearance-based method erroneously gives a safe classification.

gravel slope separating the soccer field from the lava field. The appearance-based method classified this site as safe in Section 4.4 owing to the fact that it cannot distinguish slanted gravel from level gravel by appearance alone. This is particularly illustrated in Figure 81h, which could as easily represent a slanted surface as a level one. In Figure 81i, the drone is oriented such that the ground slopes downward to the right of the image, and therefore only the strip in the center, running top to bottom has less than 10 cm of elevation change. This illustrates that a drone using an appearance-based method for the *search* phase could still abort a dangerous landing before the *commit* phase using the geometric method. This test showed promising results.

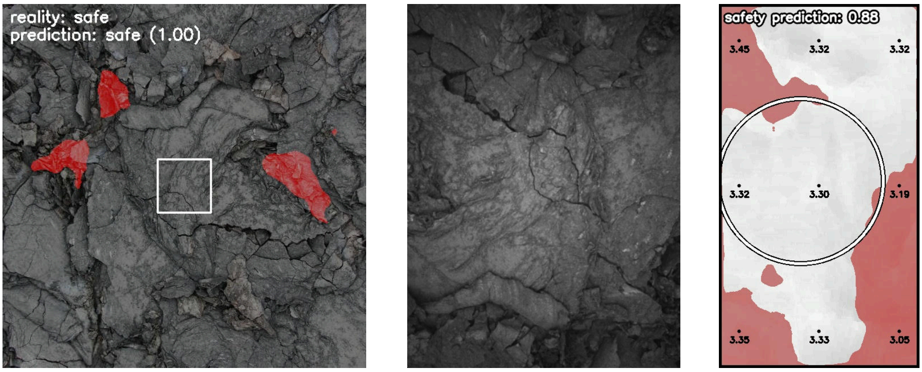
The further 9 validation sites in Figures 82, 83, and 84 are from the eruption area near Grindavík. Sites in Figures 82 and 83 are presented in order from most obviously safe to most obviously unsafe. Sites in Figure 84 are miscellaneous. We attempted to give oracle predictions denoting the safety for each of the sites, but some of the sites were not reachable because of the difficulty in traversing the lava flow. The oracle classifications are shown on the predictions for the appearance-based method, which matches them in every case.

Figure 82 describes 3 landing sites that are classified as safe by us as oracles and by both evaluation methods. Figures 82a, 82b, and 82c show site G9S – a particularly flat and level area whose only semi-dangerous areas are small cracks that can be ignored. The appearance-based method picks up some of the cracks and classifies them as unsafe, but most of the region is safe. The left stereo image and the geometric evaluation are not in the same orientation as the appearance-based evaluation, but describe the area in the bottom right of the appearance-based evaluation. Figures 82d, 82e, and 82f show site G3S – a similarly flat area as G9S, but with more cracks and problematic areas surrounding the central, safe area. The appearance-based method is, again, not in the same orientation as the stereo and depth images, but the stereo and depth images describe the central region of the appearance-based evaluation. Both evaluation methods correctly classify it as safe. Figure 85a shows the Matrice after having landed on this site. Figures 82g, 82h, and 82i show site G7S – a flat-topped pillar in an otherwise very rough lava field. Both methods correctly classify it as safe.

Figure 83 describes more difficult landing sites. Site G4S (Figures 83a, 83b, and 83c) is similar to G9S, but with a dropoff in the bottom right that reduces the radius of its safe area to slightly less than 1 m. Therefore, even though the appearance-based evaluation correctly classifies it as safe, according to the oracle classification, the geometric method classifies it as unsafe because of its conservative radius of 1 m. Figure 85b shows the drone after having landed on this site, as well as the dropoff behind the drone. Sites G8U (Figures 83d, 83e, and 83f) and G1U (Figures 83g, 83h,



(a) Site G9S: appearance-based method. (b) Closeup (alt. approx. 3.5 m) (c) Geometric method.



(d) Site G3S: appearance-based method. (e) Closeup (alt. approx. 3.5 m) (f) Geometric method.

(g) G7S: appearance-based method. (h) Closeup (alt. approx. 3.5 m) (i) Geometric method.

Figure 82: Appearance-based and geometric evaluations of sites G9S, G3S, and G7S in the lava fields near Grindavík, following the format of Figure 81. Note that the cameras in the geometric method may be in a different orientation than that of the appearance-based method, but they describe the same region. All 3 sites are correctly classified as safe by both methods. Site G9S is a large, flat region with only minor cracks that can be ignored. Site G3S is a smaller flat region with deeper cracks and some obstacles. Site G7S is a flat-topped pillar in the middle of an otherwise incredibly rough and unsafe area.

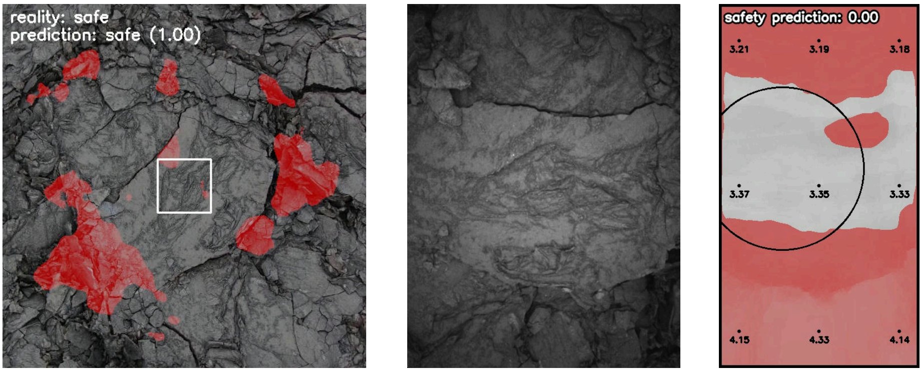
and 83i) are both unsafe areas with plenty of obstacles and roughness, classified unanimously as unsafe by the oracle predictions and by both evaluation methods.

Figure 84 shows 3 miscellaneous landing sites. Figures 84a, 84b, and 84c describe site G6S an older lava field with moss coverage, which is right next to the lava field in the 8 other test cases. The appearance-based classifier is not trained on moss but still classifies this area correctly as safe, while correctly classifying many of the surrounding bushes as unsafe. Similarly, the geometric evaluation classifies it as safe. Site G2U (Figures 84d, 84e, and 84f) and G5U (Figures 84g, 84h, and 84i) both represent level areas with a lot of rocks as obstacles that make both sites unsafe. They are both correctly classified as unsafe by both evaluation methods.

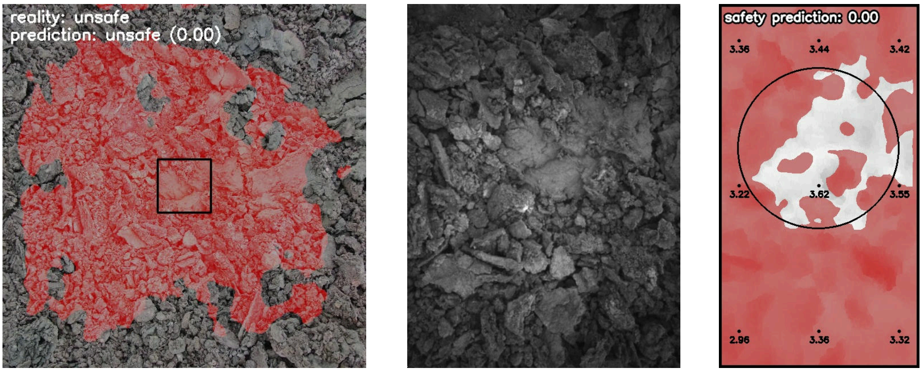
The geometric method matches the oracle predictions in all but a single case – site 4 (Figure 83c), which we classify as safe, but it classifies as unsafe. In this particular case, since we could approach the site and confirm it was adequately flat and large, we landed the drone manually on the surface and obtained a true classification of the site as safe. The geometric method rejected the site because of a dropoff that was within the 1 m radius, and generally it would be better to follow the classification from the geometric method for a final decision on whether to commit to the landing. One consideration from this test case is that downward protrusions outside of a smaller radius could be allowable, as they do not risk hitting the propellers, and the landing skids require a smaller diameter for safe landing. There is plenty of room for improving the simple decision logic that we have implemented. Figure 85 shows *manual* lava landings at sites G3S and G4S.

Video 6 shows the appearance-based classification, then the geometric classification for each landing site in order. The unsafe areas are labeled in red in the appearance-based classification, whereas safe areas are unlabeled for clarity. The geometric classifications include the left stereo image on the left of the screen, and the labeled depth image on the right side, with unsafe areas labeled in red.

Finally, we have tried creating larger U-nets that have bigger kernel sizes in the convolutional layers of their encoder and decoder steps. The underlying structure and number of layers is the same, with 5 encoder and 5 decoder steps with a bridge in between. The number of channels at each encoder step is doubled, and the number of channels at each decoder step is halved. We test 4 sizes, defined by the number of channels in the first encoder layer, which takes on the values 2, 4, 8, and 16. These produce models of size 2.7 MB (small), 9.6 MB (medium), 37 MB (large), and 147 MB (extra large) respectively. We trained 10 of each size, and conducted a coarse sweep to check parameter sizes, similar to Section 4.4.3.3, but with steps of 0.05 instead of 0.1. Since the dataset is quite small, and the smallest size as presented here correctly classified 9 of 9 validation cases, we did not investigate this further, as it most likely requires a larger data set for any conclusions. We do note that there were 5 networks that achieved 9 of 9 correct classifications, that 2 of them were the small size, 1 was



(a) Site G4S: appearance-based method. (b) Closeup (alt. approx. 3.5 m) (c) Geometric method.



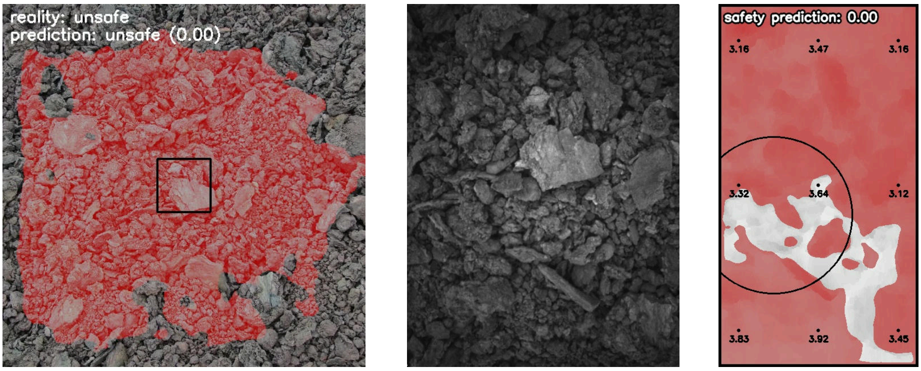
(d) Site G8U: appearance-based method. (e) Closeup (alt. approx. 3.5 m) (f) Geometric method.

(g) Site G1U: appearance-based method. (h) Closeup (alt. approx. 3.5 m) (i) Geometric method.

Figure 83: Appearance-based and geometric evaluations of sites G4S, G8U, and G1U in the lava fields near Grindavík, following the format of Figure 81. Note that the cameras in the geometric method may be in a different orientation than that of the appearance-based method, but they describe the same region. Site G4S is a safe area, as demonstrated by the landing in Figure 85b, but with a large dropoff at the bottom right of the central region. It is classified as safe by the appearance-based method but as unsafe by the geometric method because the contiguous safe region has a radius of less than 1 m. Sites G8U and G1U are both very rough areas with many obstacles and few flat regions, and are both correctly classified as unsafe in each evaluation.

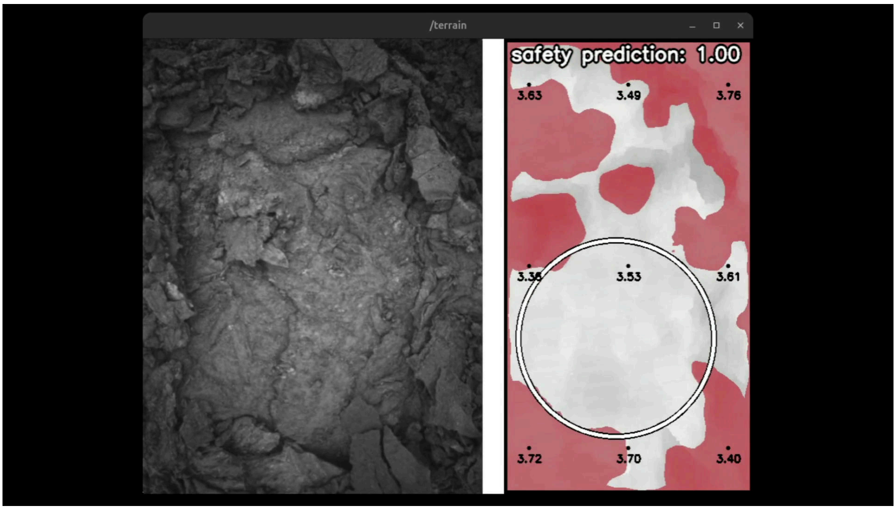
(a) Site G6S: appearance-based method. (b) Closeup (alt. approx. 3.5 m) (c) Geometric method.

(d) Site G2U: appearance-based method. (e) Closeup (alt. approx. 3.5 m) (f) Geometric method.



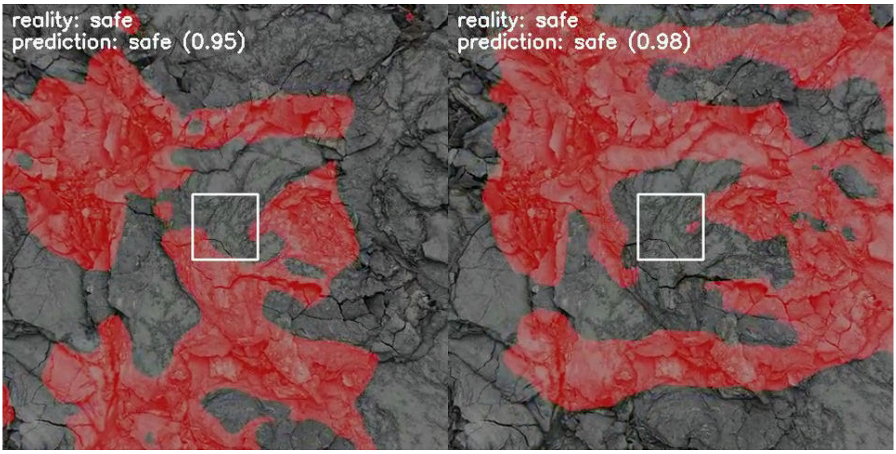
(g) Site G5U: appearance-based method. (h) Closeup (alt. approx. 3.5 m) (i) Geometric method.

Figure 84: Appearance-based and geometric evaluations of sites G6S, G2U and G5U in the lava fields near Grindavík, following the format of Figure 81. Note that the cameras in the geometric method may be in a different orientation than that of the appearance-based method, but they describe the same region. Site G6S is a mossy area next to the new flow that has not been affected by the recent eruptions. It is mostly flat aside from some small shrubs and has a slight incline. It is correctly classified as safe in both evaluations. Site G2U and and G5U are level regions that are unsafe because of their large obstacles. Both sites are correctly classified as unsafe in both evaluations.



Video 6: The combined appearance-based and geometric landing site classifications at the lava flow near Grindavík. Available at https://vimeo.com/j0shua/combined-appearance-geometric-evaluation. The video shows the visual classification followed by the geometric classification for each site. The visual classifications are presented by only overlaying red over the unsafe regions, and the other areas can be presumed safe but are not overlaid with blue for clarity in this monochromatic environment. The geometric classifications show the left stereo camera view on the left, and the labeled depth map on the right. The depth map is presented with histogram visualization for clarity. Unsafe areas are overlaid in red, and a circle is placed over the area that the system determines as the best 1 m radius circular area for landing. The circle is white if the area is deemed safe, and black if it is deemed unsafe.

medium, 1 was large, and 1 was extra large. As an example, a comparison between the best small classifier and the best large classifier is available in Video 7.



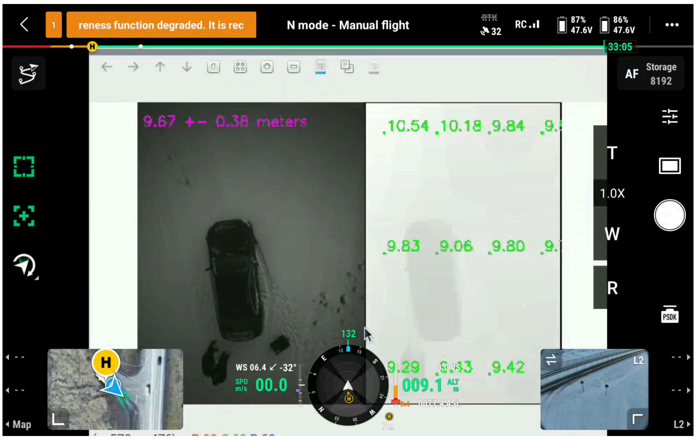
Video 7: A comparison of the best small and large appearance-based classifiers. They are presented side by side, with the small classifier on the left, and the large on the right. The results are very similar: both classifiers classify every site correctly, have a significant overlap in the regions they deem safe and unsafe, and both exhibit biases in which areas of the image canvas are more sensitive to negative or positive classifications. Available at https://vimeo.com/j0shua/comparison-appearance-classifiers.

(a) Site 3. (b) Site 4.

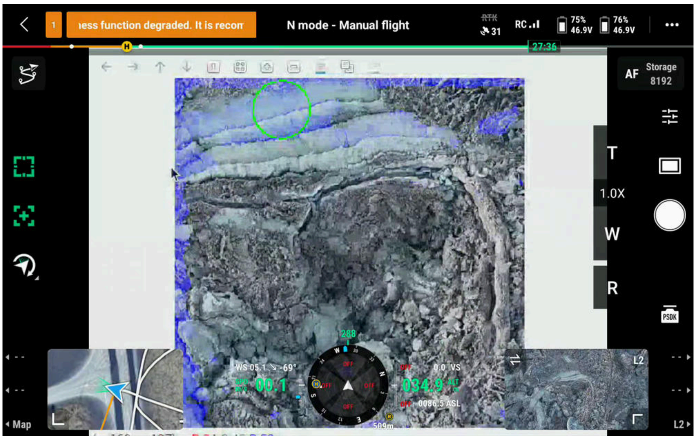
Figure 85: *Manual* lava landings at sites G3S and G4S. Site 3 was classified as safe by both the appearance-based and geometric methods, while Site 4 was classified as safe by its appearance and as unsafe by the geometric method because of the dropoff behind the drone. We carried out the landing to make the most of the fieldwork, since we could verify by sight that it was safe. (Sadly, the drone blends in well with the grey tones of the lava field, and we ran out of orange filament before printing this payload, so there is not much to distinguish the drone from its surroundings.)

Section 4.5.3.2: Payload performance

The drone can maintain flight for more than 30 minutes with the H20T and custom computational payload running simultaneously. We manually cap processing rates to 10 Hz for both the geometric and appearance-based methods. Figure 86 shows the payload's output on the Matrice's controller during flight. The payload is able to carry out the calculation of the depth image with apparent gradients removed according to the method in Section 4.2 at 10 Hz, as shown in Figure 86a. The payload is similarly able to carry out inference at 10 Hz for the appearance-based method, as shown in Figure 86b.



(a) Depth processing over a car.



(b) Performing inference over the lava scene. Safe areas are shown in blue. Unsafe areas are unlabeled. The “best” location to land according to a naive method is shown via the green circle.

Figure 86: In-flight test of the depth processing functionality using the Matrice’s downward-facing depth cameras. The system is able to generate and process the depth images at about 10 Hz, resolving a 3D depiction of the scene.

Section 4.5.4: Conclusion

We conducted an offline analysis of the performance of the geometric and appearance-based landing site evaluation methods, giving an answer to **RQ2**. In the more challenging environment of a new lava flow, the appearance-based method requires more parameters than before but accurately classifies all tested sites if given enough computational resources. The geometric method also requires more computational resources than before, owing to the fact our board has to calculate the stereo depth image on its own, instead of receiving the depth image as input from the sensor itself. Because of this, we have not yet tested the full site selection logic onboard the payload, although we anticipate it should be achievable in real time on the Jetson Nano or similar, more modern boards. The geometric method itself classifies 11 of 12 sites correctly, failing only in a single case as a result of being too cautious, i.e., requiring a landing area with an overly large radius. This is fixable with more sophisticated logic. We conducted two manual landings on the lava, but did not automate the landing process as this requires a long, slow testing phase. However, the landing could be carried out similarly to the method in Section 3.7.

We have created a payload to run both our geometric and appearance-based landing site evaluation methods onboard the drone simultaneously (without the final decision logic), and we have tested them informally in the field on our Matrice 350. This provides insight into **RQ3**, giving a data point for the required hardware and setup for running the solutions in real time onboard a drone.

Chapter 5:

Conclusions & future work

In Section 5.1, we recall our research questions from Section 1.2 and our contributions toward answering them. In Section 5.2, we mention some lessons learned as we faced various difficulties in working with drones and conducting field robotics. Finally, in Section 5.3, we present some ideas for future research.

Section 5.1: Conclusions

Recall our research questions **RQ1**, **RQ2**, and **RQ3** from Section 1.2.

> **RQ1:** how can a drone use an actuated camera to actively track a marker for autonomous landing?

For **RQ1,** we examined landing sites marked with fiducial markers, using a gimbal-mounted camera to increase tracking reliability and detection range as opposed to the common paradigm of using downward-facing cameras. We found that, on our Tarot 680 hexacopter platforms, fiducial marker orientation ambiguity prevented autonomous landing if the drone used the full, 6 DoF marker pose to direct its approach. This led us to conduct an evaluation of the prevalence of orientation ambiguity in several candidate fiducial systems, in an effort to find one that minimized this phenomenon. All of the systems exhibited orientation ambiguity to some extent. We continued by re-implementing our autonomous landing method on a small quadcopter (a DJI Spark), which could be flown inside safely, allowing us to continue testing during the Icelandic winter, and which would mitigate the dangers of erroneous control signals generated by orientation ambiguity. This system showed success in autonomous landing with 4 out of 5 of the fiducial systems, with the fifth system being apparently too dependent on the orientation of the tilt-limited camera to allow for successful landings. Improved flight stability, gimbal stabilization, and video quality likely allowed the Spark perform to where the Tarots could not. Nevertheless, the Spark's landing accuracy was negatively affected in cases of high orientation ambiguity. We proceeded to avoid this issue altogether by making another fiducial landing method that avoided using the 6 DoF marker pose, but still used a gimbal-mounted camera for active tracking. This method depended solely on the angle from the drone's heading to the landing pad, as determined by the orientation of the gimbal and the fiducial marker's pixel position in the camera frame. We created a robust control policy that handles detection loss and demonstrated successful autonomous landings using a DJI Matrice 350 without erroneous control signals caused by orientation ambiguity. Although this method reduces data requirements which , it is still able to achieve landing accuracy comparable to more sophisticated methods, i.e., about 0.19 m offset from the center of the landing pad overall in our tests. (This could be reduced with tighter tuning of the control policy parameters.) Further, the reduction in data requirements means that this method is flexible enough that it allows the system to switch between cameras and even spectra, such that the drone can make full use of its multi-payload camera with wide-angle, zoom, and IR sensors. We tested multiple landing pads and approach distances, using the zoom camera to direct the drone's landing from long distances, and using the IR camera to detect landing pads at night. We tested an active IR landing pad that requires electricity to heat itself above ambient temperature, and a passive IR landing pad that requires no power and depends on high reflectivity. Both allow for nighttime autonomous landing with an IR camera.

> **RQ2:** how can a drone efficiently and in real time land in an unstructured area that it has not previously seen?

For **RQ2,** we prototyped a method for geometrically determining if the terrain underneath the drone is safe for landing or not using a stereo depth camera with an IMU. We tested this method informally in a lab setting using data from multiple lava flows around Iceland. We created a second, lightweight method for appearance-based landing site classification that is trained on synthetic data extracted from LiDAR and photogrammetry terrain scans. The terrain classifiers are small (1-2 MB) image segmentation classifiers using the U-Net architecture, although we do not prescribe a particular classifier architecture for further use. Since these classifiers are trained on synthetic data, we create a small, real world validation data set using videos from suburban areas and from an old lava flow, on which our best classifiers obtained high accuracy. They failed on test cases where

1. the appearance of the landing site is ambiguous without a geometric intuition of the scene, e.g., slanted versus level gravel viewed in isolation,
2. the camera erroneously moved while auto-tracking the potential landing site, and

3. objects in the scene were too far away to be identified (they were then correctly flagged as unsafe once the drone moved closer — to a similar distance as the data on which the classifier was trained).

Overall, an automatically tuned classifier correctly classified 14 of 18 real-world validation sites, while a hand-tuned classifier correctly classified 15 of them in an Icelandic rural environment. An automatically tuned classifier correctly 9 of 9 real-world validation sites in an Icelandic lava flow environment. We evaluated the geometric method in several test cases in both old and new lava fields, on a soccer field, and on the sloped gravel where the appearance-based classifier failed. The geometric method succeeded in all cases but one, where it classified a site as unsafe as a result of overly cautious decision making, which can be corrected with more sophisticated decision logic.

> **RQ3:** how can our structured and unstructured landing methods run onboard our drone in real time?

To answer **RQ3,** we have tested our methods for **RQ1** and **RQ2** on real world, embedded hardware over the course of this research. For our experiment with the DJI Spark, we kept the hardware off of the drone since it did not have the capacity to carry it or connect to it directly. This showed both that our fiducial landing method could run on a Raspberry Pi 4 and that not having the method run onboard the drone introduced latency that affected the landing process. Moving forward, we developed a payload for the Matrice 350 to embed the Raspberry Pi onto it. The subsequent landing tests did not have problems resulting from the earlier latency, and we ran the method at about 17 frames per second in order not to drive the CPU to 100% usage. However, most of the processing power during these tests was dedicated simply to decoding the high-definition H264 video stream from the camera (which must still be done at 30 frames per second), since it was larger resolution than the hardware decoder on the Raspberry Pi could handle. Further, we discovered that not having a live feed of the *processed* video from the Raspberry Pi made it difficult to debug issues during testing. We built a second payload to transmit the view from the Raspberry Pi's desktop (also containing the processed video) and added an interface to interact with it using mouse clicks via the the controller's touchscreen, which made it possible to debug issues during field testing. The companion board on the second payload, a Raspberry Pi 5, also spent most of its processing time decoding video in software. This was due to it lacking a hardware decoder for H264 and instead having one only for the more modern H265 format, which is incompatible. However, with a Google Coral TPU, and as a result of the small size of our classifiers, the payload was able to execute our appearance-based landing site evaluation method in real time, albeit at a framerate of 4 frames per second, which could be improved with more time and extra optimization. Finally, to run both our geometric and appearance-based methods at once, we created a third payload that uses a Raspberry Pi 4 and an updated interface to the drone to decode the video (from a camera with smaller resolution) in hardware and transmit processed video to the drone. It has a Jetson Nano to both perform inference according to the appearance-based method and conduct geometric terrain analysis from the drone's depth cameras.

Section 5.2: Practical lessons learned in field robotics

This research took place in Iceland, which is a severe location for fieldwork, and equally so for field robotics. The weather — and often the landscape — are continual, underlying difficulties one faces in addition to the technical difficulty of doing the actual research. One might set out from Reykjavik with sunny skies and arrive at a field location 40 kilometers away in a blizzard that stops one from exiting the car for 2 hours. In the highlands, one might encounter (as we did) a blizzard in the middle of July with wind speeds upward of 20 m/s. Occasionally, volcanic eruptions change the skyline to which one has been accustomed from years of looking at it, with new mountains ushered in by bright orange nighttime skies. These are often preceded by "swarms" of hundreds to thousands of earthquakes, which might be more accurately described as the land simply shaking for weeks on end. The necessity of traveling to a field site in harsh weather to deploy a system under development means that one has to work in a different way from in a lab environment, and we will go over a few observations here.

All setup in the field should be so simple that it is impossible to forget steps, and quick enough that operators will not be tempted to take shortcuts. DJI has mastered this to some extent, in that even though DJI drones often fold and pack into carrying cases, they also unpack and assemble extremely quickly. Similarly, the necessary attachment mechanisms for the payloads have haptic and auditory feedback to indicate correct installation. We followed this idea with the quick-release brackets on our payloads and did not have any issues with them. On the other hand, in earlier experiments, we spent less time refining the drone platform and setup process, resulting in unsuccessful fumbling with screws in the field with cold fingers. Unruly or overly involved setup processes tend to result in users simply not disassembling drones for travel, which means that they cannot be stowed in secure cases and often suffer from a sort of "hangar rash," i.e., damage from non-ideal handling. This can happen while traveling on rough roads with a lot of shaking, but equally so from just sitting in a lab near equipment or people that move around. This appears to be the case especially with ad hoc payloads that have been attached with, e.g., zip ties, in an effort to save time, although we personally do not have experience with this.

Field tests should be tightly constrained and planned in detail beforehand, with a checklist. Any sort of confounding factors like inclement weather, changing field sites, and even casual conversations can cause one to forget about the agenda and miss steps, only to find out that one is missing data upon returning to the lab.

Habit is an important element that will solve unexpected problems. Software development should typically be done on the platform that will be deployed in the field itself, as opposed to, e.g., cross compilation on a faster machine. If all of the necessary commands for building and installing custom software, such as the PSDK applications in our case, are in the `bash` history, this makes it much easier to fix problems in the field when it might be raining or cold. For example, during one fieldwork session, the PSDK application to stream the screen of the payload was not working, even after rebooting several times. However, the Raspberry Pi inside was indeed booted. A simple `ctrl+r` search to run a `bash` command to reinstall the PSDK application saved the fieldwork session. Using a wireless keyboard, we were able to start the terminal and run the compilation command even without seeing the screen, since we had integrated this into the normal workflow and had memorized a few keystrokes. The command was similar to below:

```
cd ~/git/Payload-SDK && rm -rf ./build && mkdir build && cd build &&
cmake .. && make -j4 ; cd ~/git/Payload-SDK
```

The application then restarted and we were able to receive output from the system, and we would have otherwise had to take another trip to the lab to fix this small issue. Taking another trip to the lab and back wastes not only time, money, and energy, but also can squander valuable good-weather windows that can be rare in Iceland.

The PSDK has plenty of quirks, being a fairly unique, proprietary, and safety-critical system. We have mentioned some such quirks in Section 4.3.3, including idiosyncrasies with updating firmware in multiple components, surprisingly deep error propagation, and difficulty in subscribing to or unsubscribing from high-bandwidth data feeds after an initialization period. Much of this is not documented, and generally it is not advisable to blindly follow documentation, as some might be tempted to do. Gradual, empirical testing of functionality in a safe lab environment is a far better strategy. Most of the time we spent developing code for the drones was in the lab, either with the propellers removed or with the arms simply not latched into place. This ensured that even if it attempted to take off, it would simply be unable to move.

Section 5.3: Future work

Section 5.3.1: Finalize unstructured autonomous landing method

We have created the necessary components for classifying unstructured landing sites as safe or unsafe, and for geometrically determining whether or not to commit to landing on them using a stereo depth camera. Both of these methods can be used to obtain a pixel coordinate for a target landing site, meaning that they could be integrated into the existing landing control policy from Section 3.7, replacing the existing functionality that is based on fiducial marker detection. The *search* phase can be carried out in a similar way, by spinning in place and tilting the camera up and down slowly until finding a site that is adequately safe. Once such a spot is found, the drone can approach while tracking its pixel location from frame to frame using conventional tracking methods based on feature matching, e.g., SIFT [148] or ORB [149], which have open source implementations in OpenCV. Then, it can descend in place to a low altitude and make a decision in the *commit* phase based on the geometric method in Section 4.5.

Section 5.3.2: Create more sophisticated computational payloads

Many more computational boards exist than those we have tested. Given that those we have tested draw very little power compared to the drone's capacity, we believe there is room to embed significantly more powerful computational hardware, extending not only to conventional embedded boards in the NVIDIA Jetson Series, e.g., the Orin, but also to mini-computers, e.g, the Mac Mini.

Section 5.3.3: Map geometric labels directly to video validation data to create generic visual classifiers

In Section 4.4, we discussed a method to train classifiers to recognize safe and unsafe landing areas in RGB drone video. We modeled terrain using LiDAR to extract geometric information from it, from which we could automate the process of labeling safe and unsafe landing sites. However, since our goal was to create a visual classifier that could process drone video and not a point cloud, we had to generate another data set for image segmentation. Therefore, we generated meshes from the LiDAR point clouds (to create a continuous surface), imported those meshes into a simulator, and generated synthetic images from them. With this method, we could quickly propagate labels denoting which areas were safe and which were unsafe, such that our resulting synthetic data set consisted of pairs of RGB images and corresponding segmentation masks, which is usable for training an image segmentation model. Unfortunately, there is a major gap between the appearance of the synthetic data and the appearance of the real world, which is significant since we are training a classifier to evaluate a scene based on its appearance. The synthetic data looks like it could have been taken from an old video game (i.e., not realistic), and the reconstruction of meshes from point clouds adds artifacts and holes that do not exist in reality. All of this means that the metrics for accuracy and loss during the training process correspond only to the synthetic data, and that we have to depend on a separate step for validating the classifiers in the real world.

These limitations motivate us to explore a more sophisticated paradigm for the same problem of using analog environments to train appearance-based classifiers. We propose to propagate geometric labels from LiDAR surveys directly to RGB images taken from the drone. The first step would be to conduct a LiDAR survey in the analog environment in the same way as before, which is necessary to get an idea of the environment's topography. The next step would be to conduct a second survey over the same environment, collecting RGB video, and annotating each frame with the location of the drone, orientation of the gimbal, and relevant camera parameters, e.g., zoom factor, distortion parameters, etc. As we noticed in Section 4.4, appearance-based classifiers can be highly biased according to the data on which they are trained, especially with regards to altitude, and angle. We therefore suggest to continuously vary these parameters during the survey. Then, since the point cloud is also geo-referenced, it is possible to determine the location in the point cloud corresponding to the camera's field of view at a given frame, and propagate the labels to the video directly. We expect that noise in the GNSS data should be insignificant in the context of this process because of the precision gained by using RTK. Similarly, the pitch and roll components of the gimbal orientation are demonstrably reliable. On the other hand, the compass is often subject to interference from the environment and has shown significant deviations between its readings before and after booting, even if the drone has not moved in the intervening time. We therefore anticipate that some realignment will be necessary before propagating the geometric labels to the RGB frames, and therefore it may still be necessary to perform intermediate mesh reconstructions and image matching in order to determine the exact location from which to pull geometric labels. Adding ground control points (GCPs) into the surveys for more accurate global registration, which we were able to avoid in Section 4.4, would also help with the compass error correction. Assuming the frames can be properly aligned to the point cloud, the data set for training the classifier can directly use the drone's RGB video with a corresponding segmentation mask generated from the point cloud.

Training classifiers in this way would have multiple benefits. First of all, the metrics on the classifier's performance during training would be more relevant to the whole process, since they would correspond to the classifier's performance on real world data instead of synthetic data. It would therefore be less necessary to conduct a second stage of validation as we did in Section 4.4.4, and the classifier could instead be validated as often done, with an initial data set split between training and testing data sets, and a secondary data set of the same form for final validation. Further, it would make the whole process more generalizable for generic image segmentation data sets — not just for autonomous landing. For example, the 3DMASC [150] tool enables sophisticated, automated classification of point clouds based on geometric, dimensional, and spectral features, in addition to metadata such as the number of LiDAR returns. This allows explainable classification of, e.g., buildings, artificial ground, power lines, vehicles, vegetation, seabed types, rocks, pebbles, etc. Integrating these classifiers into our label generation process would make it possible to create more sophisticated appearance-based classifiers that could classify generic objects in real time onboard the drone. This has plenty of applications, e.g., identifying, quantifying, and geo-tagging trash in natural environments in real time for pickup by a human. In the case of search and rescue, such a classifier could be used to give the "highlight reels" of an automated video survey to save time helping rescuers to identify anomalies in the natural environment that could reveal a lost person's location. In terms of virtual reality, it could be used to create animations visualizing changing landforms such as the flow of newer lava flows over older lava flows. In short, it broadens the possibilities for real time, spatially aware, appearance-based classifiers.

Section 5.3.4: Use larger, well-known, generative AI models

ChatGPT and DeepSeek have obviously made a huge impact on the world at large. Although these models are relatively heavy, the Raspberry Pi 5 can run the smallest DeepSeek R1 models. It naturally makes sense to wonder if these more general models can give any advantage over the small, specialized models in Section 4.4. As an initial test, we tried to get ChatGPT to mimic our validation step. We submitted the input images for Figures 75a and 76d to ChatGPT's 4o model, which can output images, along with the following prompt:

> Determine which areas in the attached image are unsafe for a drone to land and label them visually.

The model generated the following Python script to process the image, under the assumption that darker areas represent obstacles, and generated the images in Figure 87.

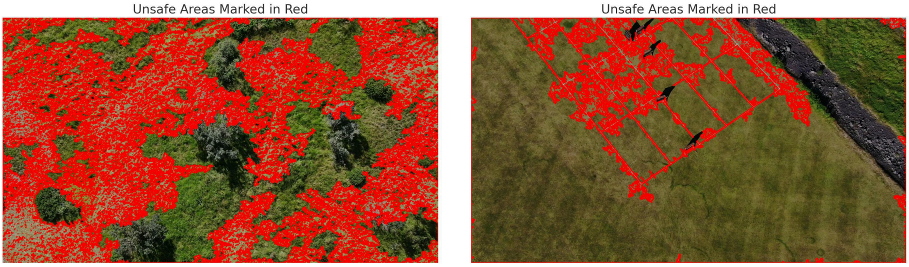
```
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```
# Load the image
image_path = "/mnt/data/
a965f5504c6eb7d61eee960a28e3923e282058592eb4bbefb73cbd478e0eea49.jpg"
image = cv2.imread(image_path)

# Convert to grayscale for processing
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply thresholding to identify trees and obstacles (assuming darker
areas represent obstacles)
_, thresholded = cv2.threshold(gray, 100, 255, cv2.THRESH_BINARY_INV)

# Find contours of obstacles
contours, _ = cv2.findContours(thresholded, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

# Draw contours on the original image
annotated_image = image.copy()
cv2.drawContours(annotated_image, contours, -1, (0, 0, 255), 3)  # Red
for obstacles

# Display the annotated image
plt.figure(figsize=(10, 6))
plt.imshow(cv2.cvtColor(annotated_image, cv2.COLOR_BGR2RGB))
plt.axis("off")
plt.title("Unsafe Areas Marked in Red")
plt.show()
```

In a second case, we provided the following prompt:

> Label the areas in the attached drone photo that would be unsafe for landing by masking them lightly in red.

The model responded with the following image in Figure 88, which seems to be based on the fact that it analyzed the image and noted the general location of the



(a) 35.5 m altitude.

(b) 35.5 m altitude.

Figure 87: Output from ChatGPT attempting to detect unsafe landing sites.

Figure 88: ChatGPT’s labeling of unsafe landing sites in the archery stand image from Figure 75a.

archery stands, but could not accurately annotate them. Of course, this output in itself is not adequate for autonomous landing, but as the models become more capable, they may be directly applicable to the task.

Bibliography

[1] E. Commission, “A Drone Strategy 2.0 for a Smart and Sustainable Unmanned Aircraft Eco-System in Europe,” Brussels, Nov. 2022. [Online]. Available: https://ec.europa.eu/commission/presscorner/detail/en/IP_22_4157

[2] E. Nowak, K. Gupta, and H. Najjaran, “Development of a Plug-and-Play Infrared Landing System for Multirotor Unmanned Aerial Vehicles,” in *2017 14th Conference on Computer and Robot Vision (CRV)*, 2017, pp. 256–260. doi: 10.1109/CRV.2017.23.

[3] N. Xuan-Mung, S. K. Hong, N. P. Nguyen, L. N. N. T. Ha, and T.-L. Le, “Autonomous Quadcopter Precision Landing Onto a Heaving Platform: New Method and Experiment,” *IEEE Access*, vol. 8, pp. 167192–167202, 2020, doi: 10.1109/ACCESS.2020.3022881.

[4] D. Falanga, A. Zanchettin, A. Simovic, J. Delmerico, and D. Scaramuzza, “Vision-based autonomous quadrotor landing on a moving platform,” in *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, 2017, pp. 200–207. doi: 10.1109/SSRR.2017.8088164.

[5] A. Borowczyk, D.-T. Nguyen, A. Phu-Van Nguyen, D. Q. Nguyen, D. Saussié, and J. L. Ny, “Autonomous Landing of a Multirotor Micro Air Vehicle on a High Velocity Ground Vehicle,” *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 10488–10494, 2017, doi: https://doi.org/10.1016/j.ifacol.2017.08.1980.

[6] P. H. Nguyen, K. W. Kim, Y. W. Lee, and K. R. Park, “Remote Marker-Based Tracking for UAV Landing Using Visible-Light Camera Sensor,” *Sensors*, vol. 17, no. 9, 2017, doi: 10.3390/s17091987.

[7] T. Liao, A. Haridevan, Y. Liu, and J. Shan, “Autonomous Vision-Based UAV Landing with Collision Avoidance Using Deep Learning,” in *Intelligent Computing*, K. Arai, Ed., Cham: Springer International Publishing, 2022, pp. 79–87.

[8] S. Chaves, R. Wolcott, and R. Eustice, “NEEC Research: Toward GPS-denied Landing of Unmanned Aerial Vehicles on Ships at Sea,” *Naval Engineers Journal*, vol. 127, pp. 23–35, 2015.

[9] O. Araar, N. Aouf, and I. Vitanov, “Vision Based Autonomous Landing of Multirotor UAV on Moving Platform,” *Journal of Intelligent & Robotic Systems*, vol. 85, 2017, doi: 10.1007/s10846-016-0399-z.

[10] J. S. Wynn, “Visual Servoing for Precision Shipboard Landing of an Autonomous Multirotor Aircraft System,” Brigham Young University. De-

partment of Mechanical Engineering, 2018. [Online]. Available: http://hdl.lib.byu.edu/1877/etd10385

[11] J. Wubben *et al.*, “Accurate Landing of Unmanned Aerial Vehicles Using Ground Pattern Recognition,” *Electronics*, vol. 8, p. 1532, 2019, doi: 10.3390/electronics8121532.

[12] H. Tanaka, K. Ogata, and Y. Matsumoto, “Solving pose ambiguity of planar visual marker by wavelike two-tone patterns,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 568–573. doi: 10.1109/IROS.2017.8202209.

[13] H. Tanaka and Y. Matsumoto, “Autonomous Drone Guidance and Landing System Using AR/high-accuracy Hybrid Markers,” in *2019 IEEE 8th Global Conference on Consumer Electronics (GCCE)*, 2019, pp. 598–599. doi: 10.1109/GCCE46687.2019.9015373.

[14] G. Cho, J. Choi, G. Bae, and H. Oh, “Autonomous ship deck landing of a quadrotor UAV using feed-forward image-based visual servoing,” *Aerospace Science and Technology*, vol. 130, p. 107869, 2022, doi: https://doi.org/10.1016/j.ast.2022.107869.

[15] T. Jiang, D. Lin, and T. Song, “Vision-based autonomous landing of a quadrotor using a gimbaled camera,” *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 233, no. 14, pp. 5093–5106, 2019, doi: 10.1177/0954410019837777.

[16] M. Demirhan and C. Premachandra, “Development of an Automated Camera-Based Drone Landing System,” *IEEE Access*, vol. 8, pp. 202111–202121, 2020, doi: 10.1109/ACCESS.2020.3034948.

[17] R. M. Claro, D. B. Silva, and A. M. Pinto, “ArTuga: A novel multimodal fiducial marker for aerial robotics,” *Robotics and Autonomous Systems*, vol. 163, p. 104398, 2023, doi: https://doi.org/10.1016/j.robot.2023.104398.

[18] P. Lusk, P. Glaab, L. Glaab, and R. Beard, “Safe2Ditch: Emergency Landing for Small Unmanned Aircraft Systems,” *Journal of Aerospace Information Systems*, vol. 16, pp. 1–13, 2019, doi: 10.2514/1.I010706.

[19] B. J. Petty, P. C. Glaab, and L. J. Glaab, “Safe2Ditch Steer-to-Clear Development and Flight Testing,” in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, 2019, pp. 1–10. doi: 10.1109/DASC43569.2019.9081728.

[20] P. F. A. Di Donato and E. M. Atkins, “Evaluating Risk to People and Property for Aircraft Emergency Landing Planning,” *Journal of Aerospace Information Systems*, vol. 14, no. 5, pp. 259–278, 2017, doi: 10.2514/1.I010513.

[21] K. Pluckter and S. Scherer, “Precision UAV Landing in Unstructured Environments,” in *Proceedings of the 2018 International Symposium on Experimental Robotics*, J. Xiao, T. Kröger, and O. Khatib, Eds., in Springer Proceedings in Advanced Robotics, vol. 11. Cham: Springer, 2020, pp. 177–187. doi: 10.1007/978-3-030-33950-0_16.

[22] T. Yang, P. Li, H. Zhang, J. Li, and Z. Li, “Monocular Vision SLAM-Based UAV Autonomous Landing in Emergencies and Unknown Environments,” *Electronics*, vol. 7, p. 73, 2018, doi: 10.3390/electronics7050073.

[23] A. Johnson, J. Montgomery, and L. Matthies, “Vision Guided Landing of an Autonomous Helicopter in Hazardous Terrain,” in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 2005, pp. 3966–3971. doi: 10.1109/ROBOT.2005.1570727.

[24] R. Brockers *et al.*, “Autonomous Safe Landing Site Detection for a Future Mars Science Helicopter,” in *2021 IEEE Aerospace Conference (50100)*, 2021, pp. 1–8. doi: 10.1109/AERO50100.2021.9438289.

[25] P. F. Proença, J. Delaune, Rol, and Brockers, “Optimizing Terrain Mapping and Landing Site Detection for Autonomous UAVs,” in *2022 International Conference on Robotics and Automation (ICRA)*, 2022, pp. 9668–9674. doi: 10.1109/ICRA46639.2022.9811789.

[26] S. Scherer, L. Chamberlain, and S. Singh, “Autonomous landing at unprepared sites by a full-scale helicopter,” *Robotics and Autonomous Systems*, vol. 60, no. 12, pp. 1545–1562, 2012, doi: https://doi.org/10.1016/j.robot.2012.09.004.

[27] L. Yang, C. Wang, and L. Wang, “Autonomous UAVs landing site selection from point cloud in unknown environments,” *ISA Transactions*, vol. 130, pp. 610–628, 2022, doi: https://doi.org/10.1016/j.isatra.2022.04.005.

[28] G. Loureiro, A. Dias, A. Martins, and J. Almeida, “Emergency Landing Spot Detection Algorithm for Unmanned Aerial Vehicles,” *Remote Sensing*, vol. 13, no. 10, 2021, doi: 10.3390/rs13101930.

[29] R. Garg, S. Yang, and S. Scherer, “Monocular and Stereo Cues for Landing Zone Evaluation for Micro UAVs,” 2018, [Online]. Available: https://arxiv.org/abs/1812.03539

[30] J. M. Hintze, “Autonomous Landing of a Rotary Unmanned Aerial Vehicle in a Non-cooperative Environment using Machine Vision,” Master thesis. Brigham Young University. Department of Electrical & Computer Engineering, 2004. [Online]. Available: http://hdl.lib.byu.edu/1877/etd359

[31] C. Symeonidis, E. Kakaletsis, I. Mademlis, N. Nikolaidis, A. Tefas, and I. Pitas, “Vision-based UAV Safe Landing exploiting Lightweight Deep Neural Net-

works," in *Proceedings of the 2021 4th International Conference on Image and Graphics Processing*, in ICIGP '21. Sanya, China: Association for Computing Machinery, 2021, pp. 13–19. doi: 10.1145/3447587.3447590.

[32] A. Alsawy, A. Hicks, D. Moss, and S. Mckeever, "An Image Processing Based Classifier to Support Safe Dropping for Delivery-by-Drone," in *2022 IEEE 5th International Conference on Image Processing Applications and Systems (IPAS)*, 2022, pp. 1–5. doi: 10.1109/IPAS55744.2022.10052868.

[33] H. Du, W. Wang, X. Wang, and Y. Wang, "Autonomous landing scene recognition based on transfer learning for drones," *Journal of Systems Engineering and Electronics*, vol. 34, no. 1, pp. 28–35, 2023, doi: 10.23919/JSEE.2023.000031.

[34] A. Marcu, D. Costea, V. LicăreŢ, M. Pîrvu, E. Sluşanschi, and M. Leordeanu, "SafeUAV: Learning to Estimate Depth and Safe Landing Areas for UAVs from Synthetic Data," in *Computer Vision – ECCV 2018 Workshops*, L. Leal-Taixé and S. Roth, Eds., Cham: Springer International Publishing, 2019, pp. 43–58.

[35] G. Yashin, A. Egorov, Z. Darush, N. Zherdev, and D. Tsetserukou, "LocoGear: Locomotion Analysis of Robotic Landing Gear for Multicopters," *IEEE Journal on Miniaturization for Air and Space Systems*, vol. 1, no. 2, pp. 138–147, 2020, doi: 10.1109/JMASS.2020.3015525.

[36] S. Karaf, A. Fedoseev, M. Martynov, Z. Darush, A. Shcherbak, and D. Tsetserukou, "MorphoLander: Reinforcement Learning Based Landing of a Group of Drones on the Adaptive Morphogenetic UAV," in *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2023, pp. 2507–2512. doi: 10.1109/SMC53992.2023.10394504.

[37] S. Þ. Jónsson, "Self-adapting landing gear for multi-rotor drones," 2023. [Online]. Available: http://hdl.handle.net/1946/44873

[38] J. Springer and M. Kyas, "Evaluation of Orientation Ambiguity and Detection Rate in April Tag and WhyCode," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, Naples, Italy: IEEE Computer Society, Dec. 2022, pp. 281–286. doi: 10.1109/IRC55401.2022.00054.

[39] J. Springer and M. Kyas, "Autonomous Drone Landing with Fiducial Markers and a Gimbal-Mounted Camera for Active Tracking," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, Naples, Italy: IEEE Computer Society, Dec. 2022, pp. 243–247. doi: 10.1109/IRC55401.2022.00047.

[40] J. Springer, "Autonomous Multirotor Landing on Landing Pads and Lava Flows," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, 2022, pp. 425–428. doi: 10.1109/IRC55401.2022.00081.

[41] J. Springer and M. Kyas, “Autonomous Drone Landing: Marked Landing Pads and Solidified Lava Flows,” *International Journal of Semantic Computing*, vol. 18, no. 2, pp. 283–299, 2024, doi: 10.1142/S1793351X24300061.

[42] J. Springer, G. Þ. Guðmundsson, and M. Kyas, “A Precision Drone Landing System using Visual and IR Fiducial Markers and a Multi-Payload Camera,” 2024. [Online]. Available: https://arxiv.org/abs/2403.03806

[43] J. Springer, G. Þ. Guðmundsson, and M. Kyas, “Lowering Barriers to Entry for Fully-Integrated Custom Payloads on a DJI Matrice,” in *2024 International Conference on Content-Based Multimedia Indexing (CBMI)*, 2024, pp. 1–5. doi: 10.1109/CBMI62980.2024.10859236.

[44] J. Springer, G. Þ. Guðmundsson, and M. Kyas, “Toward Appearance-Based Autonomous Landing Site Identification for Multirotor Drones in Unstructured Environments,” in *MultiMedia Modeling*, I. Ide, I. Kompatsiaris, C. Xu, K. Yanai, W.-T. Chu, N. Nitta, M. Riegler, and T. Yamasaki, Eds., Singapore: Springer Nature Singapore, 2025, pp. 198–211.

[45] B. B. Carr *et al.*, “Evaluating the Use of Unoccupied Aircraft Systems (UASs) for Planetary Exploration in Mars Analog Terrain,” *The Planetary Science Journal*, vol. 5, no. 10, p. 231, Oct. 2024, doi: 10.3847/PSJ/ad781e.

[46] M. Kyas, J. D. Springer, J. T. Pedersen, and V. Chkoniya, “Data Analysis in the Shipping Industry: eShip Case Study – Problem Statement,” in Handbook of Research on Applied Data Science and Artificial Intelligence in Business and Industry., Hershey, PA, USA: IGI Global, 2021, pp. 381–400. doi: 10.4018/978-1-7998-6985-6.ch018.

[47] Z. M. Zhang and G. Machin, “Chapter 1 Overview of Radiation Thermometry,” *Radiometric Temperature Measurements: I. Fundamentals*, vol. 42. in Experimental Methods in the Physical Sciences, vol. 42. Academic Press, pp. 22–23107, 2009. doi: https://doi.org/10.1016/S1079-4042(09)04201-5.

[48] P. M. Kant, “Geothermal mapping and remote sensing of surface thermal anomalies at Grændalur area, Hveragerði, SW Iceland,” 2021. [Online]. Available: http://hdl.handle.net/1946/39990

[49] L. M. R. González, “Remote sensing of surface hydrothermal alteration, identification of minerals and thermal anomalies at Sveifluháls-Krýsuvík high-temperature geothermal field, SW Iceland,” 2018. [Online]. Available: http://hdl.handle.net/1946/29516

[50] H. S. Ndlovu, J. Odindi, M. Sibanda, and O. M. and, “A systematic review on the application of UAV-based thermal remote sensing for assessing and monitoring crop water status in crop farming systems,” *International*

Journal of Remote Sensing, vol. 45, no. 15, pp. 4923–4960, 2024, doi: 10.1080/01431161.2024.2368933.

[51] J. Jia, Y. Wang, J. Chen, R. Guo, R. Shu, and J. Wang, “Status and application of advanced airborne hyperspectral imaging technology: A review,” *Infrared Physics & Technology*, vol. 104, p. 103115, 2020, doi: https://doi.org/10.1016/j.infrared.2019.103115.

[52] International Telecommunication Union, “H.264: Advanced video coding for generic audiovisual services (Recommendation ITU-T H.264).” [Online]. Available: https://www.itu.int/rec/T-REC-H.264-202408-I/en

[53] International Telecommunication Union, “H.265: High efficiency video coding (Recommendation ITU-T H.265).” [Online]. Available: https://www.itu.int/rec/T-REC-H.265

[54] Z. Zhang, “A flexible new technique for camera calibration,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000, doi: 10.1109/34.888718.

[55] A. Kaehler and G. Bradski, *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*, 1st ed. O'Reilly Media, Inc., 2016.

[56] J. Weng, P. Cohen, and M. Herniou, “Camera calibration with distortion models and accuracy evaluation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 10, pp. 965–980, 1992, doi: 10.1109/34.159901.

[57] G. Bradski, “The OpenCV Library,” *Dr. Dobb's Journal of Software Tools*, 2000.

[58] M. Kazhdan, M. Bolitho, and H. Hoppe, “Poisson surface reconstruction,” in *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, in SGP '06. Cagliari, Sardinia, Italy: Eurographics Association, 2006, pp. 61–70.

[59] E. S. M. Ebeid, M. Skriver, K. Laursen, K. Jensen, and U. Schultz, “A Survey of Open-Source UAV Flight Controllers and Flight Simulators,” *Microprocessors and Microsystems*, vol. 61, 2018, doi: 10.1016/j.micpro.2018.05.002.

[60] The Dronecode Foundation, “MAVLink Micro Air Vehicle Communication Protocol.” [Online]. Available: https://mavlink.io/

[61] ArduPilot, “Ardupilot: Versatile, Trusted, Open.” [Online]. Available: https://ardupilot.org/

[62] “PX4 Open Source Autopilot.” [Online]. Available: https://px4.io/

[63] Emlid, “Navio2.” [Online]. Available: https://emlid.com/navio/

[64] Apache Software Foundation, “NuttX Real-Time Operating System.” [Online]. Available: http://nuttx.apache.org/

[65] ChibiOS Community, “ChibiOS/RT Real-Time Operating System.” [Online]. Available: http://www.chibios.org/

[66] *Matrice 350 RTK User Manual.* Da Jiang Innovations (DJI), 2023. [Online]. Available: https://dl.djicdn.com/downloads/matrice_350_rtk/Matrice_350_RTK_User_Manual_en.pdf

[67] SZ DJI Technology Co., Ltd., “Zenmuse H20 Series User Manual.” Shenzhen, China, May 2020. [Online]. Available: https://dl.djicdn.com/downloads/Zenmuse_H20_Series/Zenmuse_H20_Series_User_Manual-EN.pdf

[68] SZ DJI Technology Co., Ltd., “ZENMUSE L2 User Manual.” Shenzhen, China, Oct. 2023. [Online]. Available: https://dl.djicdn.com/downloads/Zenmuse_L2/Zenmuse_L2_User_Manual_v1.0_EN.pdf

[69] Da-Jiang Innovations (DJI), “Mobile SDK V5.” [Online]. Available: https://developer.dji.com/mobile-sdk/

[70] Da-Jiang Innovations (DJI), “Build a Drone Aerial-Specific toolkit.” [Online]. Available: https://github.com/dji-sdk/Payload-SDK

[71] Da-Jiang Innovations (DJI), “Onboard SDK V5.” [Online]. Available: https://github.com/dji-sdk/Onboard-SDK

[72] Da-Jiang Innovations (DJI), “DJI Developer Documentation.” Accessed: Mar. 08, 2025. [Online]. Available: https://developer.dji.com/document/30ac6801-db84-46c2-baf2-8ad8d62bf3ba

[73] M. Quigley, “ROS: an open-source Robot Operating System,” in *IEEE International Conference on Robotics and Automation*, 2009. [Online]. Available: http://www.robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf

[74] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, May 2022, doi: 10.1126/scirobotics.abm6074.

[75] N. Koenig and A. Howard, “Design and use paradigms for Gazebo, an open-source multi-robot simulator,” in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, 2004, pp. 2149–2154. doi: 10.1109/IROS.2004.1389727.

[76] 3D Robotics, Inc., “DroneKit-Python.” [Online]. Available: https://github.com/dronekit/dronekit-python

[77] NVIDIA Corporation, “Isaac Sim.” [Online]. Available: https://developer.nvidia.com/isaac-sim

[78] S. E. Mathe, H. K. Kondaveeti, S. Vappangi, S. D. Vanambathina, and N. K. Kumaravelu, “A comprehensive review on applications of Raspberry Pi,”

Computer Science Review, vol. 52, p. 100636, 2024, doi: https://doi.org/10.1016/j.cosrev.2024.100636.

[79] Raspberry Pi Foundation, "Raspberry Pi 3 B+ Product Brief." [Online]. Available: https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf

[80] Raspberry Pi Foundation, "Raspberry Pi 4 B+ Product Brief." [Online]. Available: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf

[81] Raspberry Pi Foundation, "Raspberry Pi 5 Product Brief." [Online]. Available: https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf

[82] Google, "Coral Dev Board." [Online]. Available: https://coral.ai/products/dev-board/

[83] "TensorFlow Lite: Lightweight solution for mobile and embedded devices." [Online]. Available: https://www.tensorflow.org/lite

[84] Google, "Coral Edge TPU." [Online]. Available: https://coral.ai/products/accelerator

[85] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems." [Online]. Available: http://tensorflow.org/

[86] "Jetson Nano." [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano

[87] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, and M. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014, doi: https://doi.org/10.1016/j.patcog.2014.01.005.

[88] M. Fiala, "ARTag, a fiducial marker system using digital techniques," *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 2, pp. 590–596, 2005, [Online]. Available: https://api.semanticscholar.org/CorpusID:5746274

[89] E. Olson, "AprilTag: A Robust and Flexible Visual Fiducial System," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 3400–3407. doi: 10.1109/ICRA.2011.5979561.

[90] J. DeGol, T. Bretl, and D. Hoiem, "ChromaTag: A Colored Marker and Fast Detection Algorithm," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1481–1490. doi: 10.1109/ICCV.2017.164.

[91] M. Nitsche, T. Krajník, P. Čížek, M. Mejail, and T. Duckett, "WhyCon: An Efficient, Marker-based Localization System," in *IROS Workshop on Open Source*

Aerial Robotics, 2015. [Online]. Available: https://core.ac.uk/download/pdf/42583963.pdf

[92] P. Lightbody, T. Krajník, and M. Hanheide, “A Versatile High-performance Visual Fiducial Marker Detection System with Scalable Identity Encoding,” in *Proceedings of the Symposium on Applied Computing*, in SAC '17. Marrakech, Morocco: ACM, 2017, pp. 276–282. doi: 10.1145/3019612.3019709.

[93] B. Benligiray, C. Topal, and C. Akinlar, “STag: A stable fiducial marker system,” *Image and Vision Computing*, vol. 89, pp. 158–169, 2019, doi: https://doi.org/10.1016/j.imavis.2019.06.007.

[94] D. Jurado, R. Muñoz-Salinas, S. Garrido-Jurado, and R. Medina-Carnicer, “Planar fiducial markers: a comparative study,” *Virtual Reality*, vol. 27, pp. 1–17, 2023, doi: 10.1007/s10055-023-00772-5.

[95] J. Wang and E. Olson, “AprilTag 2: Efficient and robust fiducial detection,” 2016, pp. 4193–4198. doi: 10.1109/IROS.2016.7759617.

[96] M. Krogius, A. Haggenmiller, and E. Olson, “Flexible Layouts for Fiducial Tags,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Macau, China: IEEE Press, 2019, pp. 1898–1903. doi: 10.1109/IROS40897.2019.8967787.

[97] J. Springer, “Autonomous Landing of a Multicopter Using Computer Vision,” 2020. [Online]. Available: http://hdl.handle.net/1946/36422

[98] V. Ermakov, *MAVROS – MAVLink extendable communication node for ROS with proxy for Ground Control Station*. 2018. [Online]. Available: http://wiki.ros.org/mavros

[99] User benbenson, “Canopy for Tarot 680 Pro.” [Online]. Available: https://www.thingiverse.com/thing:4191694

[100] P. Irmisch, “Camera-based Distance Estimation for Autonomous Vehicles,” Department of Computer Vision & Remote Sensing, Technische Universität Berlin, 2017. [Online]. Available: https://elib.dlr.de/116211/

[101] S. F. Chng, N. Sogi, P. Purkait, T.-J. Chin, and K. Fukui, “Resolving Marker Pose Ambiguity by Robust Rotation Averaging with Clique Constraints *,” 2020, pp. 9680–9686. doi: 10.1109/ICRA40945.2020.9196902.

[102] P.-C. Wu, Y.-H. Tsai, and S.-Y. Chien, “Stable pose tracking from a planar target with an analytical motion model in real-time applications,” 2014. doi: 10.1109/MMSP.2014.6958793.

[103] M. Krogius, A. Haggenmiller, and E. Olson, “GitHub - AprilRobotics/apriltag-generation: Generate AprilTags in custom layouts.” Accessed: Feb. 26, 2025. [Online]. Available: https://github.com/AprilRobotics/apriltag-generation

[104] J. Ulrich, “Fiducial Marker Detection for Vision-based Mobile Robot Localisation.” [Online]. Available: https://dspace.cvut.cz/bitstream/handle/10467/89879/F3-BP-2020-Ulrich-Jiri-main.pdf

[105] J. Ulrich, A. Alsayed, F. Arvin, and T. Krajník, “Towards fast fiducial marker with full 6 DOF pose estimation,” in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, in SAC '22. Virtual Event: Association for Computing Machinery, 2022, pp. 723–730. doi: 10.1145/3477314.3507043.

[106] Da-Jiang Innovations (DJI), “Spark User Manual.” Oct. 2017. [Online]. Available: https://dl.djicdn.com/downloads/Spark/Spark_User_Manual_v1.6_en.pdf

[107] SZ DJI Technology Co., Ltd., “Manifold 2 Series User Guide.” Shenzhen, China, May 2019. [Online]. Available: https://dl.djicdn.com/downloads/manifold-2/20190528/Manifold_2_User_Guide_v1.0_EN.pdf

[108] CubePilot, “The Cube Orange Flight Controller.” [Online]. Available: https://cubepilot.org/

[109] Hex Aero Technologies, “Herelink Remote Control and Video System.” [Online]. Available: https://hex.aero/products/herelink

[110] QGroundControl Development Team, “QGroundControl: Ground Control Station for UAVs.” [Online]. Available: https://qgroundcontrol.com/

[111] Michael Oborne and the ArduPilot Community, “Mission Planner: Ground Control Station for ArduPilot.” [Online]. Available: https://ardupilot.org/planner/

[112] J. Springer, “DJI Matrice 300/350 Custom Payload Case.” [Online]. Available: https://www.thingiverse.com/thing:6681945

[113] J. Springer, “Payload-SDK App: RPi Autonomous Landing.” [Online]. Available: https://github.com/uzgit/Payload-SDK/tree/apriltag

[114] M. A. Olivares-Mendez, I. F. Mondragón, and P. Campoy, “Autonomous Landing of an Unmanned Aerial Vehicle using Image-Based Fuzzy Control,” *IFAC Proceedings Volumes*, vol. 46, no. 30, pp. 79–86, 2013, doi: https://doi.org/10.3182/20131120-3-FR-4045.00011.

[115] G. Badakis, M. Koutsoubelias, and S. Lalis, “Robust Precision Landing for Autonomous Drones Combining Vision-based and Infrared Sensors,” in *2021*

IEEE Sensors Applications Symposium (SAS), 2021, pp. 1–6. doi: 10.1109/SAS51076.2021.9530091.

[116] R. Polvara, S. Sharma, J. Wan, A. Manning, and R. Sutton, “Towards autonomous landing on a moving vessel through fiducial markers,” in *2017 European Conference on Mobile Robots (ECMR)*, 2017, pp. 1–6. doi: 10.1109/ECMR.2017.8098671.

[117] J. Lim, T. Lee, S. Pyo, J. Lee, J. Kim, and J. Lee, “Hemispherical InfraRed (IR) Marker for Reliable Detection for Autonomous Landing on a Moving Ground Vehicle From Various Altitude Angles,” *IEEE/ASME Transactions on Mechatronics*, vol. 27, no. 1, pp. 485–492, 2022, doi: 10.1109/TMECH.2021.3066643.

[118] M. D. Dogan *et al.*, “InfraredTags: Embedding Invisible AR Markers and Barcodes Using Low-Cost, Infrared-Based 3D Printing and Imaging Tools,” in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, in CHI '22., New Orleans, LA, USA,: Association for Computing Machinery, 2022. doi: 10.1145/3491102.3501951.

[119] S. Khattak, C. Papachristos, and K. Alexis, “Marker Based Thermal-Inertial Localization for Aerial Robots in Obscurant Filled Environments,” 2018, pp. 565–575. doi: 10.1007/978-3-030-03801-4_49.

[120] F. Sigmundsson *et al.*, “Deformation and seismicity decline before the 2021 Fagradalsfjall eruption,” *Nature*, vol. 609, pp. 1–6, 2022, doi: 10.1038/s41586-022-05083-4.

[121] BBC News. Click, “Why Iceland’s volcanoes may aid Mars exploration.” [Online]. Available: https://www.bbc.co.uk/programmes/p09r8nzv

[122] J. Bapst *et al.*, “Mars Science Helicopter: Compelling Science Enabled by an Aerial Platform,” *Bulletin of the AAS*, vol. 53, no. 4, Mar. 2021, [Online]. Available: https://baas.aas.org/pub/2021n4i361

[123] C. Hamilton, “RAVEN : Mars Exploration Drone Testing in Iceland.” [Online]. Available: https://www.lpl.arizona.edu/outreach/multimedia/raven-mars-exploration-drone-testing-iceland

[124] S. Gwizd *et al.*, “Rover-Aerial Vehicle Exploration Network (RAVEN): Mission Planning, Implementation, and Results from the 2022 Rover-Only Field Campaign at Holuhraun, Iceland,” 2023.

[125] G. Pascale *et al.*, “On the Move: 2023 Observations on Real Time Graben Formation, Grindavík, Iceland,” *Geophysical Research Letters*, vol. 51, 2024, doi: 10.1029/2024GL110150.

[126] E. Saldiran, M. Hasanzade, A. Cetin, and G. Inalhan, “Autonomous Emergency Landing System to Unknown Terrain for UAVs,” in *2024 AIAA DATC/IEEE*

43rd Digital Avionics Systems Conference (DASC), 2024, pp. 1–7. doi: 10.1109/DASC62030.2024.10749002.

[127] E. Saldiran, M. Hasanzade, A. Cetin, and G. Inalhan, "Ensuring Operation Time Safety of VTOL UAV: Autonomous Emergency Landings in Unknown Terrain," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 1–15, 2025, doi: 10.1109/TAES.2025.3559899.

[128] J. Springer, "Geometric Landing Site Evaluation (with D455) Repository." [Online]. Available: https://github.com/uzgit/geometric_landing_site_analysis_d455

[129] J. Springer, "Matrice Parts Repository." [Online]. Available: https://github.com/uzgit/matrice_parts

[130] DJI, [Online]. Available: https://developer.dji.com/doc/payload-sdk-tutorial/en/model-instruction/choose-develop-platform.html#aircraft-type-function-difference

[131] J. Springer, "Edited Payload-SDK Repository." [Online]. Available: https://github.com/uzgit/Payload-SDK/

[132] J. Springer, "Raspberry Pi 5 PSDK Configuration Script." [Online]. Available: https://github.com/uzgit/Payload-SDK/blob/master/configure.sh

[133] S. Beycimen, D. Ignatyev, and A. Zolotas, "A comprehensive survey of unmanned ground vehicle terrain traversability for unstructured environments and sensor technology insights," *Engineering Science and Technology, an International Journal*, vol. 47, p. 101457, 2023, doi: https://doi.org/10.1016/j.jestch.2023.101457.

[134] D. Barnes, W. Maddern, and I. Posner, "Find your own way: Weakly-supervised segmentation of path proposals for urban autonomy," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 203–210. doi: 10.1109/ICRA.2017.7989025.

[135] L. Tang, X. Ding, H. Yin, Y. Wang, and R. Xiong, "From one to many: Unsupervised traversable area segmentation in off-road environment," in *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 787–792. doi: 10.1109/ROBIO.2017.8324513.

[136] M. Sánchez, J. L. Martínez, J. Morales, A. Robles, and M. Morán, "Automatic Generation of Labeled 3D Point Clouds of Natural Environments with Gazebo," in *2019 IEEE International Conference on Mechatronics (ICM)*, 2019, pp. 161–166. doi: 10.1109/ICMECH.2019.8722866.

[137] E. Kakaletsis *et al.*, "Computer Vision for Autonomous UAV Flight Safety: An Overview and a Vision-based Safe Landing Pipeline Example," *ACM Comput. Surv.*, vol. 54, no. 9, Oct. 2021, doi: 10.1145/3472288.

[138] L. Chen, Y. Xiao, X. Yuan, Y. Zhang, and J. Zhu, "Robust autonomous landing of UAVs in non-cooperative environments based on comprehensive terrain understanding," *Science China Information Sciences*, vol. 65, no. 11, p. 212202, 2022, doi: 10.1007/s11432-021-3429-1.

[139] A. Marcu, V. Licaret, D. Costea, and M. Leordeanu, "Semantics Through Time: Semi-supervised Segmentation of Aerial Videos with Iterative Label Propagation," 2021, pp. 537–552. doi: 10.1007/978-3-030-69525-5_32.

[140] T. Mitroudas, K. A. Tsintotas, N. Santavas, A. Psomoulis, and A. Gasteratos, "Towards 3D printed modular unmanned aerial vehicle development: The landing safety paradigm," in *2022 IEEE International Conference on Imaging Systems and Techniques (IST),* 2022, pp. 1–6. doi: 10.1109/IST55454.2022.9827665.

[141] Loftmyndir ehf., [Online]. Available: https://map.is/

[142] P. Toffanin *et al.*, "OpenDroneMap/WebODM: 2.5.4." [Online]. Available: https://doi.org/10.5281/zenodo.12775235

[143] "CloudCompare (version 2.12) [GPL software]." [Online]. Available: http://www.cloudcompare.org/

[144] T. Hackel, J. D. Wegner, and K. Schindler, "Contour Detection in Unstructured 3D Point Clouds," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* 2016, pp. 1610–1618. doi: 10.1109/CVPR.2016.178.

[145] F. Chollet and others, "Keras." [Online]. Available: https://keras.io/

[146] J. Springer, "Unstructured Landing Site Identification Repository." [Online]. Available: https://github.com/uzgit/unstructured_landing_site_identification_mmm2025

[147] Leetop Technology Co., Ltd., "A203 V2 Carrier Board for NVIDIA Jetson Nano/Xavier NX/TX2 NX." [Online]. Available: https://www.leetop.top/products/a203-carrier-baord-for-nvidia-jetson-nx-tx2-nx

[148] D. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the Seventh IEEE International Conference on Computer Vision,* 1999, pp. 1150–1157. doi: 10.1109/ICCV.1999.790410.

[149] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *2011 International Conference on Computer Vision*, 2011, pp. 2564–2571. doi: 10.1109/ICCV.2011.6126544.

[150] M. Letard *et al.*, “3DMASC: Accessible, explainable 3D point clouds classification. Application to bi-spectral topo-bathymetric lidar data,” *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 207, pp. 175–197, 2024, doi: https://doi.org/10.1016/j.isprsjprs.2023.11.022.