

Captor Report

Ben Agro

May 4, 2023

Contents

| | | |
|----------|-------------------------------------|-----------|
| 1 | Introduction | 1 |
| 2 | Design Philosophy | 1 |
| 2.1 | Simulation-First | 2 |
| 2.2 | Future-Proofing | 4 |
| 2.3 | Simplicity | 4 |
| 3 | Background And Related Work | 4 |
| 3.1 | Hardware | 5 |
| 3.2 | Software | 5 |
| 3.3 | Theory | 6 |
| 4 | Challenge Task Solutions | 6 |
| 4.1 | Notation | 7 |
| 4.2 | System Overview | 7 |
| 4.3 | Stationkeeping | 8 |
| 4.4 | Waypoint Navigation | 9 |
| 4.5 | Online Obstacle Avoidance | 10 |
| 5 | Results and Evaluation | 14 |
| 5.1 | Stationkeeping | 14 |
| 5.2 | Waypoint Navigation | 14 |
| 5.3 | Online Obstacle Avoidance | 15 |
| 6 | Lessons Learned | 16 |

1 Introduction

This report summarizes our capstone design project. We, team Captors, have designed, built, and programmed an autonomous drone, named CAPTOR, that can understand and navigate within its environment. Our design was motivated and assessed using five *challenge tasks* (CT) of increasing difficulty: drone bring-up (CT1), station-keeping (CT2), waypoint navigation (CT3), obstacle avoidance (CT4), and environment sensing (CT5). Given the vast design space, there are many potential development procedures and solutions to these problems. In this report, we describe our team’s design philosophy, background, and related work, how we approached and solved each CT, the results of our efforts, and the lessons we learned along the way.

2 Design Philosophy

There are three tenets of our design philosophy: (1) simulation-first, (2) future-proofing, and (3) simplicity. This section describes each principle and how it manifested in our team.

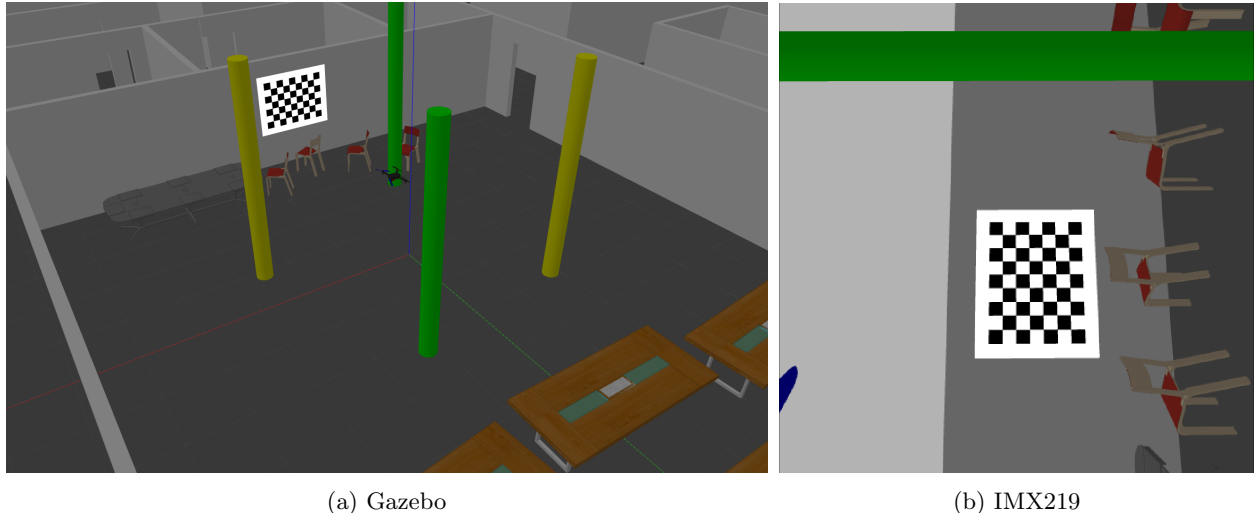


Figure 1: Images from MYHALSIM. Figure 1a shows an external view from Gazebo [1], and fig. 1b shows the simulated IMX219 color camera (after undistortion).

2.1 Simulation-First

The first tenet of our design philosophy is *simulation first*. Simulation provides many advantages during the design process, which we detail below.

Rapid iteration: Limited flight time in the physical Myhal 580 (MY580) testing environment hinders development. Simulation allows for rapid iteration on hardware (e.g., the layout of sensors) and software designs (e.g., autonomy systems).

Robust testing: Simulation allows for much more testing and pre-flight verification of our solutions. For each CT, we had a suite of tests to pass in simulation before testing on the actual drone. Figure 2 shows an example of CT4 in simulation [2].

Testing in simulation reduces the risk of crashes and hardware malfunction, which hinder development. Furthermore, simulation allows us to enumerate different challenge task configurations (e.g., landmark and obstacle locations) to test edge cases we would otherwise want to avoid in real life.

Towards the *simulation-first* design philosophy, we developed MYHALSIM; a high-fidelity simulation of the MY580 testing environment built in Gazebo 11 [1]. See fig. 1 for images from this simulation. MYHALSIM simulates

- MY580 testing environment with miscellaneous objects (tables and chairs), obstacles, and calibration checkboard, (fig. 1a),
- CAPTOR drone, including
 - PX4 flight controller [3] [4],
 - Realsense T265 tracking camera (fig. 1b) with odometry estimates and fisheye images [5],
 - IMX219 color camera (fig. 1b),
 - Vicon dots frame \mathcal{F}_d ,
- Controller for arming the PX4 and changing flight modes,
- Vicon system for the ground truth transformation between the world frame \mathcal{F}_w and the Vicon dots frame \mathcal{F}_d .

In this same spirit, we use *log-replay* to reduce the sim-to-real gap in our testing. Log replay involves collecting data from the real drone to test obstacle detection, mapping, and camera calibration. Figure 3 shows an example frame from log replay data used to develop detection and mapping.

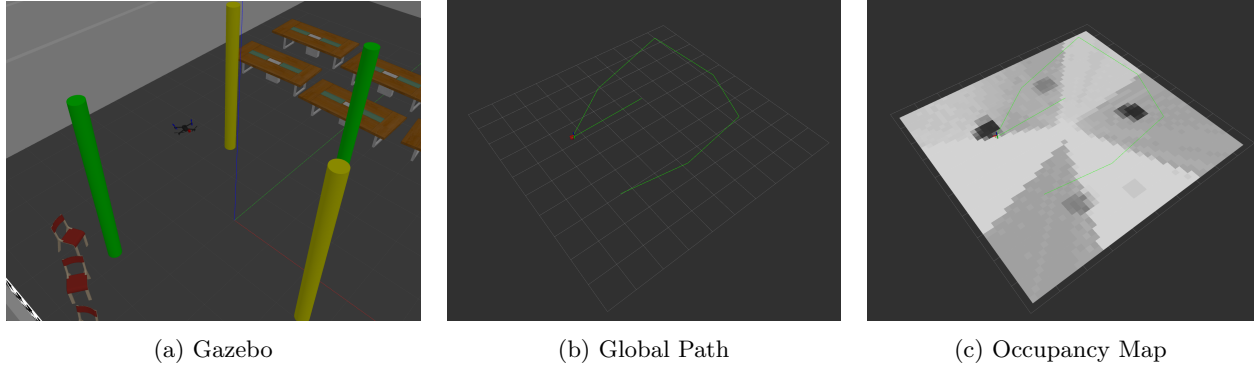


Figure 2: Visualizations from the simulation of CT4.

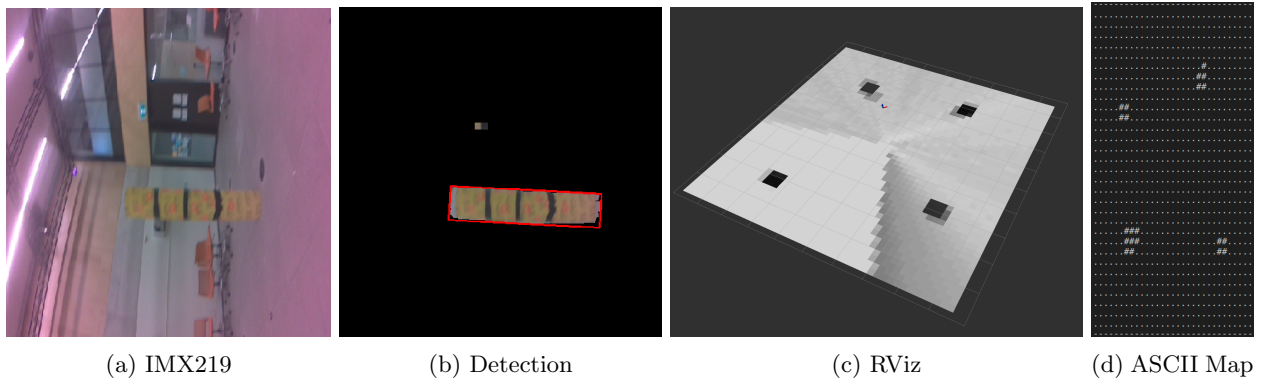


Figure 3: An example of log-replay used to design the detection and mapping system for CT4.

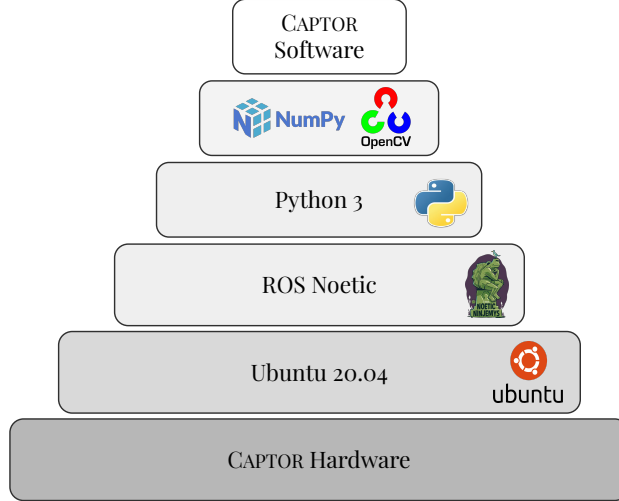


Figure 4: Our hardware and software stack.

2.2 Future-Proofing

The second tenet of our design philosophy is *future-proofing*; we want to prevent our intermediate designs from incurring technical debt in the future, and we design our system with longevity in mind [6]. While we did not have the freedom to select hardware, we chose the latest and greatest operating systems, firmware, and software during our design process:

- We ran a custom Ubuntu 20.04 on the Nvidia Jetson Nano [7], which supports the latest software packages (e.g., Robot Operating System (ROS) Noetic [8]). Furthermore, Ubuntu 20.04 is supported until 2030 (unlike the default Ubuntu 18.04 operating system that ships with the Jetson Nano).
- We used ROS Noetic [8], the latest ROS 1 version.
- ROS Noetic allowed us to program in Python 3 and use all of the latest versions of open-source software packages like OpenCV [9] and NumPy [10].

See fig. 4 for an illustration of our software stack.

On top of employing the latest open-source solutions, we designed the CAPTOR system for each CT with future tasks in mind. For example, for CT2 (stationkeeping), we already implemented a global and local planning framework, which was easily extensible to CT3 and CT4.

2.3 Simplicity

The third tenet of our design philosophy is *simplicity* [2]. We try to avoid over-engineered solutions and use open-source implementations wherever possible, including simulation with Gazebo [1], robot operating system with ROS, visual-inertial odometry using the Realsense T265, flight control using the PX4, computer vision with OpenCV, and fast A* on grids with pyastar2d [11].

Further, we try and *fail-fast* - rapidly rule out possible solutions in parallel to find the most straightforward path forwards. An example of this was in our initial work on CT4. Initially three team members explored the merits of different approaches (occupancy mapping with stereo from the T265, occupancy mapping with the IMX219, and reactive collision avoidance with the IMX219). We quickly determined which idea had the most promise by identifying potential issues with each solution.

3 Background And Related Work

This section outlines the hardware, software frameworks, and background theory used throughout our design work.

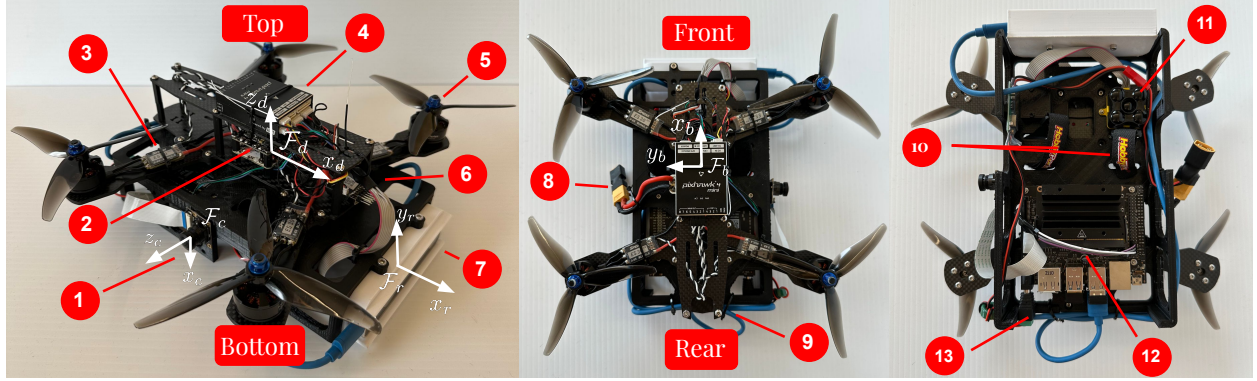


Figure 5: Labelled hardware and frames of CAPTOR.

3.1 Hardware

CAPTOR uses a carbon fiber frame for its lightweight and high rigidity. To accommodate all the peripherals, a custom 3D-printed chassis is attached to the underside of the frame. Figure 5 labels the various peripherals and components mounted on the frame and chassis:

1. **Sony IMX219:** A monocular color camera, useful for computer vision tasks.
2. **PX4 Mini PDB:** A power distribution board (PDB) that supplies power to PX4 Mini and motors.
3. **Spedix ESC:** An electronic speed controller (ESC) that controls the motors.
4. **PX4 Mini:** A flight controller that fuses sensor data to generate a state estimate, handles remote-control (RC) commands and dictates motor speeds based on flight commands.
5. **Gemfan 5" 3 blade propeller:** Four plastic propellers, chosen for safety.
6. **ARCHER R4:** A lightweight RC receiver.
7. **Intel RealSense T265:** A stereo camera that performs visual-inertial odometry to generate a pose estimate.
8. **XT60 battery connector:** Wiring that splits power from LiPo battery to PX4 Mini PDB and Jetson Nano.
9. **Vicon Mount:** The location where motion capture (Vicon) dots are mounted on the drone, used by the Vicon system to obtain "ground truth" pose estimates.
10. **Battery straps:** Used to secure the battery to the chassis.
11. **TeraRanger Evo 60m:** A time-of-flight sensor used to measure the distance to ground with a minimum range of 0.6 m and maximum range of 60 m.
12. **NVIDIA Jetson Nano 4GB:** A small computer used for autonomous drone control, which can communicate with the PX4 through the `mavros` framework [12].
13. **Barrel jack and voltage regulator:** Used to supply regulated power from the battery to Jetson Nano.

3.2 Software

The Jetson Nano was our development platform, running a custom Ubuntu 20.04 OS image [7]. Our development used the following software:

- **ROS Noetic:** the most recent ROS release, which supports Python 3. We use the following ROS packages:
 - **rospy:** a pure python client for ROS.

- **mavros**: provides a communication driver with the PX4 autopilot, allowing us, among many other things, to send position commands and pose estimates to the PX4 and receive the fused PX4 state estimate [12].
- **tf**: allows for convenient querying of transformations by maintaining the relationship between frames in a tree structure buffered in time.
- Python 3, and the following open-source packages:
 - **NumPy**: An array-based computing framework for fast multi-dimensional array operations and matrix math in Python [10].
 - **OpenCV**: An open-source computer vision library with Python bindings for all computer vision-related tasks such as undistortion, camera calibration, and image transformations [9].
 - **pyastar**: A blazing fast grid based A* implemented in C++ and wrapped for Python [11].

3.3 Theory

This section gives a brief background theory on the ideas and algorithms used during our design process. Please refer to the cited sources for more information.

- **Rigid Transformation Matrices**: Matrices that live in the special Euclidian group $\mathbf{T}_{ab} \in SE(3) = \left\{ \begin{bmatrix} \mathbf{C}_{ab} & \mathbf{p}_a^{ba} \\ \mathbf{0}^T & 1 \end{bmatrix} \in \mathbb{R}^{4 \times 4} \mid \mathbf{C}_{ab}^T \mathbf{C}_{ab} = \mathbf{1}, \det(\mathbf{C}) = 1 \right\}$ that represent rotations and translations. \mathbf{T}_{ab} represents a transformation matrix that turns a point expressed in frame b , \mathbf{p}_b , into a point expressed in frame a , \mathbf{p}_a : $\mathbf{p}_a = \mathbf{T}_{ab} \mathbf{p}_b$. Transformations can be composed via matrix multiplication: $\mathbf{T}_{ab} = \mathbf{T}_{ac} \mathbf{T}_{cb}$ [13].
- **Pin Hole Camera Model**: is an idealized camera model described as $\mathbf{y} = \mathbf{K} \mathfrak{D}(\frac{1}{\mathbf{e}_3^T \mathbf{T}_{sw} \mathbf{p}_w} \mathbf{T}_{sw} \mathbf{p}_w)$, where \mathbf{p}_w is the observed object in the world frame, \mathbf{T}_{sw} is a transformation from world frame to sensor frame, \mathfrak{D} is a non-linear distortion function (usually the plum-bob model), and \mathbf{K} is the intrinsic camera matrix [13]. See [13] for more details.
- **Occupancy grid mapping**: A mapping technique using a 2D or 3D grid where each cell holds the occupancy probability. Most formulations use a recursive update in log-odds (logits) space:
$$l(c_k | y_{1:t}, x_{1:t}) = l(c_k | y_{1:t-1}, x_{1:t-1}) + \begin{cases} +\alpha & \text{if } y_t \text{ indicates the cell is occupied} \\ -\beta & \text{otherwise} \end{cases}, \text{ where } x_t, y_t \text{ are the}$$
 robot state and measurement at time t , c_k represents the occupancy of the k^{th} cell, and $l(c_k | y_{1:t}, x_{1:t}) = \log \left(\frac{p(c_k | y_{1:t}, x_{1:t})}{1 - p(c_k | y_{1:t}, x_{1:t})} \right)$ is the log-odds [14].
- **Extended Kalman Filter (EKF)**: is an optimal estimation algorithm used to estimate the state of a system given measurements. Like the standard Kalman filter, the algorithm considers state and measurement noise as Gaussian. However, the Extended Kalman Filter operates on non-linear systems by linearizing them [15]. The EKF is the state estimator used by the PX4 [3].
- **Visual-Inertial Odometry (VIO)**: is a technique that combines data from visual and inertial sensors to estimate the motion of a robot in real-time. The T265 tracking camera uses VIO to estimate its pose [16].

4 Challenge Task Solutions

This section describes our solutions to each of the challenge tasks. First, we summarize the structure of our system. Then, for each task, we provide a brief list of objectives, a functional analysis, and our solution details. Please refer to the official scoring documents and scripts [here](#) for a highly detailed description of each task [17].

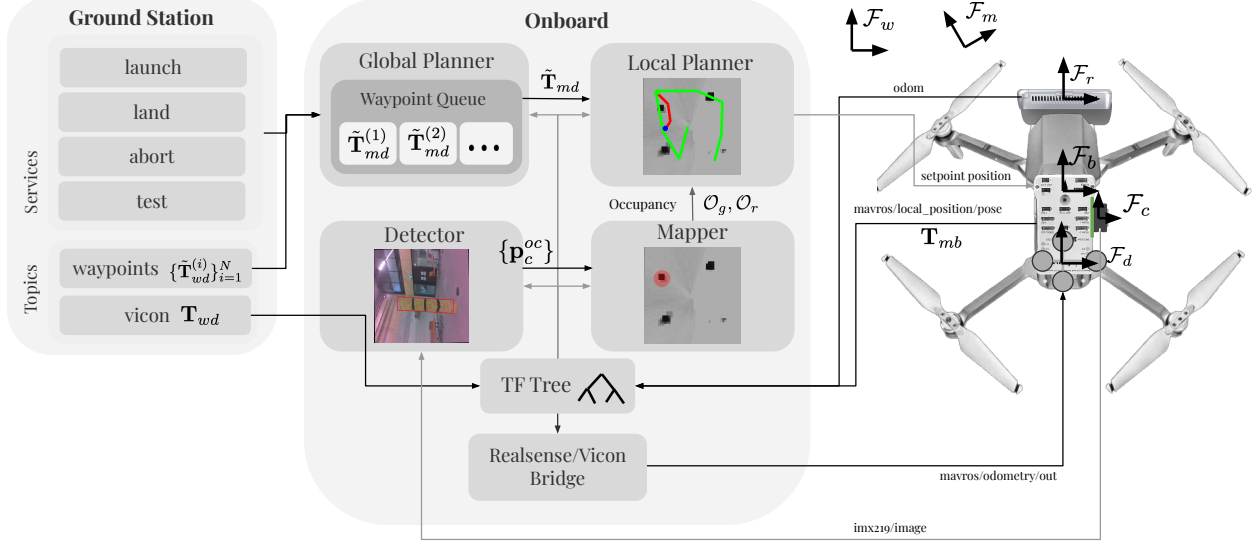


Figure 6: A system overview diagram of CAPTOR. Various utility nodes are not shown, like the IMX219 camera publisher. Reference frames are for illustration purposes only (see fig. 5 for more accurate frames).

4.1 Notation

This section describes the notation used during our description of the CAPTOR system and challenge task solutions.

We use \mathbf{T}_{ab} to denote a transformation matrix transforming from coordinates in frame b to frame a . We denote “desired” poses with a tilde, e.g., $\tilde{\mathbf{T}}_{ab}$. \mathbf{p}_c^{ba} denotes the translation from point a to point b , expressed in frame c . We denote the yaw angle of a pose with the letter θ .

\mathcal{F}_w , referred to as the world frame, is fixed in the center of the MY580 testing room and is the fixed frame used by the Vicon system. \mathcal{F}_d is the frame defined by the Vicon dots model placed on the CAPTOR, meaning the Vicon system measures \mathbf{T}_{wd} . \mathcal{F}_b is the frame fixed to the origin of the PX4. \mathcal{F}_c is the frame fixed to the optical center of the IMX219 camera, with the z-axis pointing along the optical axis and the x-axis pointing towards the bottom of CAPTOR. \mathcal{F}_m is the fixed map frame, which is taken as the initial \mathcal{F}_b when the system starts. \mathcal{F}_r is the frame fixed to the Realsense T265 from which the VIO estimate is provided. Refer to fig. 5 for an illustration of $\mathcal{F}_c, \mathcal{F}_r, \mathcal{F}_d, \mathcal{F}_b$.

4.2 System Overview

This section describes the CAPTOR system at a high level. Further details are included in our description of the solution to each challenge task. Refer to fig. 6 for a diagram of the main modules in CAPTOR.

CAPTOR consists of two main parts: the *ground station* and the *onboard* system. The ground station is responsible for sending high-level semantic commands to the drone, such as `launch`, `land`, `abort`, and `test` through ROS service calls. The ground station can also send information through ROS topics, including desired waypoints $\{\tilde{\mathbf{T}}_{wd}^{(i)}\}_{i=1}^N$ for testing, and ground truth pose information from the Vicon system (the latest measured \mathbf{T}_{wd}).

The onboard system runs on the Jetson Nano using ROS and is responsible for enacting the commands from the ground station. It has a transform tree (TF tree) which stores a buffer of static and dynamic transformations in a tree structure. Sub-modules of the onboard system can query this TF tree for relevant transformations at a specific time. We feed the following transformations to the TF tree:

- \mathbf{T}_{mb} , the fused pose estimate from the PX4.
- \mathbf{T}_{bd} , the fixed transformation between the PX4 and Vicon dots frame.
- \mathbf{T}_{bc} , the fixed transformation between the PX4 and the IMX219 frame.

- \mathbf{T}_{rb} , the fixed transformation between the T265 odom frame and the PX4.
- and \mathbf{T}_{wd} , the pose estimate from the Vicon system.

We will describe the sub-modules of the onboard system during our description of the challenge task solutions in the order they were developed and added to CAPTOR.

4.3 Stationkeeping

CT2 was *stationkeeping*.

Task Description: The goal is to maintain a stable hover at a fixed altitude (height). After being sent a `launch` command from the ground station, CAPTOR is required to autonomously ascend to an altitude of 1.5 m (The z value of \mathbf{p}_w^{dw} , as measured by the Vicon system). After sending the `test` command, the drone must maintain a stable hover for 30 s. Finally, the drone should autonomously land after sending the `land` command. The requirement is to keep the heading within $\pm 5^\circ$ of the initial heading when `test` is called, and altitude within $1.5 \text{ m} \pm 0.1 \text{ m}$, and the horizontal position within $\pm 0.15 \text{ m}$ of the initial position at the start of `test`.

Functional Analysis: CT2 breaks down into four main functions:

- **Launch:** The drone must be able to control itself to a pose 1.5 m above the ground (usually 1.5 m above the current pose).
- **Land:** The drone must be able to safely land by controlling itself to a pose on the ground (usually directly below the current pose).
- **Hover:** The drone must be capable of maintaining its current pose. We assess this capability by measuring the maximum translational and yaw deviation from the desired pose during the hovering period of 30 s (lower is better).
- **State Estimate:** To achieve the previous functions, the drone must accurately estimate its pose in the world.

Solution: To meet these requirements, and with future CTs in mind, we implemented a *Realsense/Vicon bridge*, a *Global Planner*, and a *Local Planner* in the onboard system, which we describe below:

Realsense / Vicon Bridge: This node implements the state estimate functionality by taking in a pose estimate (from either the T265 or Vicon) and passing it to the PX4 for use in its EKF pose estimator.

When using Vicon, the bridge node requires \mathbf{T}_{db} to be present in the TF tree. The bridge subscribes to the `TransformStamped` message from the Vicon system \mathbf{T}_{wd} , computes $\mathbf{T}_{wb} = \mathbf{T}_{wd}\mathbf{T}_{db}$, and publishes that transformation through the `mavros/odometry/out` topic as a `Odometry` message. Thus, the PX4 can fuse the Vicon position estimate into its state estimate. We set the `Odometry` message velocity and covariance values to zero because the Vicon does not provide velocity information, and we assumed the poses to be perfect. Note that when using Vicon, \mathcal{F}_m will coincide with \mathcal{F}_w .

When using the T265, the bridge node requires \mathbf{T}_{rb} in the TF tree. The bridge subscribes to the `Odometry` message from the T265, transforms it into \mathcal{F}_b , and publishes the resulting `Odometry` message to `mavros/odometry/out` to be fused into the PX4 state estimate.

Different parameter files are loaded into the PX4 when flying with Vicon or the T265 for pose information. When flying with Vicon, we disable all other sources of pose information, such as the Terabee. When flying with the Realsense, we use the Terabee for height information.

With this bridge node active, the PX4 can accurately estimate its pose, allowing for position hold mode when manually flying CAPTOR.

Global Planner: The Global Planner is responsible for implementing the various service calls by maintaining a thread-safe first-in-first-out queue of waypoints in the map frame $Q = \{\tilde{\mathbf{T}}_{md}^{(i)}\}_{i=1}^N$ to visit and the current desired waypoint $\tilde{\mathbf{T}}_{md}$. When the current desired waypoint is not set, the Global Planner pops the next waypoint off Q , if available. The Global Planner calls the Local Planner at 20 Hz with the current pose of the dots frame in the map frame \mathbf{T}_{md} and the desired pose $\tilde{\mathbf{T}}_{md}$, and the Local Planner send a control command to the PX4 to move towards that desired pose. The Global Planner is agnostic to the internals of the Local Planner, allowing for flexibility in implementation. Before calling the Local Planner, the Global Planner checks if it has reached the current waypoint by calling `pose_is_close(\mathbf{T}_{md} , $\tilde{\mathbf{T}}_{mb}$)`, which returns `True` if \mathbf{T}_{md} is within 0.04 m of the desired position and if the yaw angle θ of the drone is within 5° of the desired yaw. `False` is returned otherwise. If CAPTOR is close to the current waypoint and Q is not empty, the current waypoint is set to the next waypoint popped off of Q . If Q is empty, we maintain the current waypoint.

For CT2, we implemented the `launch` and `land` services in the Global Planner. The launch service added a single waypoint at $x = 0, y = 0, z = 1.5 \text{ m}, \theta = 0$, while the land command added a single waypoint at $x = 0, y = 0, z = 0, \theta = 0$, both expressed in the map frame \mathcal{F}_m .

Local Planner: The Local Planner takes in the current pose of the Vicon dots in the map frame \mathbf{T}_{md} and the current desired waypoint $\tilde{\mathbf{T}}_{md}$ and publishes a message to the PX4 to command the drone to move towards that desired waypoint. For CT2, the Local Planner simply sends a pose command to the PX4 that puts the base frame at the desired pose in the map frame $\tilde{\mathbf{T}}_{mb} = \tilde{\mathbf{T}}_{md}\mathbf{T}_{db}$. We limited the horizontal and vertical velocity to 0.6 m/s in the PX4 flight controller for smoother control.

4.4 Waypoint Navigation

CT3 was *waypoint navigation*.

Task Description: The goal is the navigate along a sequence of seven waypoints specified in the \mathcal{F}_w frame. First, the ground station sends a `launch` command, where CAPTOR is meant to ascend to an altitude of approximately 1.5 m. Then, the ground station sends a `test` command along with the seven desired waypoints $\{\tilde{\mathbf{T}}_{wd}^{(i)}\}_{i=1}^7$. The drone is required to reach all of the waypoints in order within 60 s. A waypoint with desired position $\tilde{\mathbf{p}}_w^{dw}$ is “reached” if the Vicon system measures \mathbf{p}_w^{dw} to be within 0.35 m of $\tilde{\mathbf{p}}_w^{dw}$ at any point during the test. After CAPTOR travels along the waypoints, the ground station will send a `land` command.

Functional Analysis: CT3 builds on the functions from CT2 and adds the following functions:

- **World Frame Conversion:** The drone must be able to read Vicon data from the ground station to determine \mathbf{T}_{wm} (the transformation between the local map frame and the world frame). This transformation is required because the given landmarks are expressed in the world frame \mathcal{F}_w .
- **Waypoint Intake:** The drone must be able to intake a sequence of waypoints from the ground station and use them to perform the `test` command.
- **Waypoint Navigation:** The drone must be able to fly to the desired poses, detecting when it has reached a pose, continuing to the subsequent waypoint, and hovering when it has reached the final waypoint. This must be performed within the required time limit ($\leq 60 \text{ s}$) and with the required position accuracy ($\pm 0.35 \text{ m}$).

Solution: To meet these requirements, we were able to reuse the majority of our implementation from CT2. We only had to modify the global planner to add a `test` service, which set a flag `test_ready` that specified it was ready for testing, and a subscriber to the desired waypoints in the world frame $\{\tilde{\mathbf{T}}_{wd}^{(i)}\}_{i=1}^7$ from the ground station. In the waypoint subscriber, we first check if `test_ready` is if the drone is flying. If either of these conditions were not met, a warning is printed to the terminal. Otherwise, we transform the waypoints to be expressed in the map frame $\tilde{\mathbf{T}}_{md}^{(i)} = \mathbf{T}_{mw}\tilde{\mathbf{T}}_{wd}^{(i)}$ using the latest transform \mathbf{T}_{mw} from the

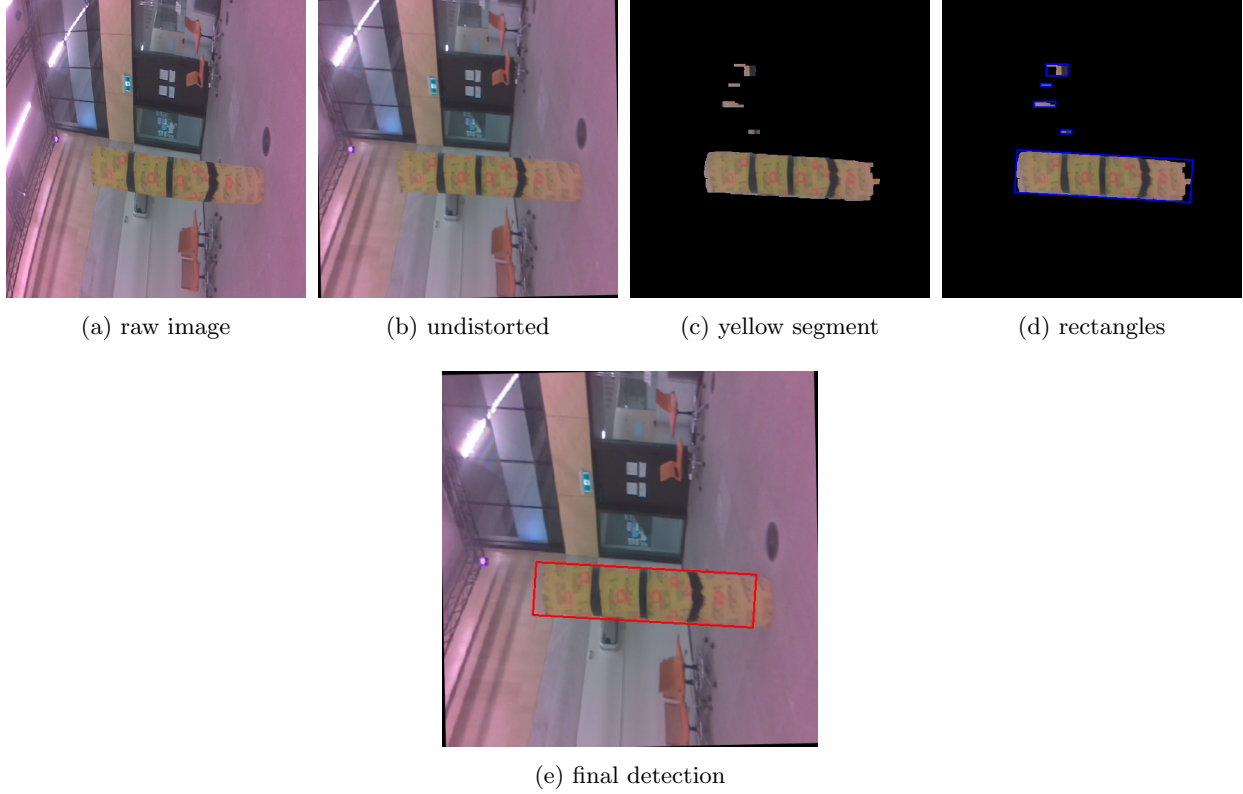


Figure 7: The sequence of image transformations used in obstacle detection, from figs. 7a to 7e.

TF tree, and pushed onto the waypoint queue. Other than this addition, we kept everything the same from CT2.

4.5 Online Obstacle Avoidance

CT4 was *online obstacle avoidance*.

Task Description: The goal of CT4 was to navigate to seven waypoints, similar to CT3, with four yellow pole obstacles placed in the testing area. The drone must fly counterclockwise around the poles with green lettering and clockwise around the poles with red lettering. The time limit for reaching all the waypoints is 75 s.

Functional Analysis: CT4 builds on the functions from CT2 and CT3, and adds the following functions:

1. **Percieve:** The drone must be able to detect the red and green obstacles.
2. **Discern:** The drone must be able to differentiate red from green obstacles.
3. **Plan:** The drone must be able to plan a path to each waypoint that avoids the obstacles in the correct direction.
4. **Acting:** The drone must be able to follow its proposed plan without colliding with obstacles and staying within the CT4 time constraint.

Solution: To meet these requirements, we added three new nodes to the onboard system: the *IMX219 publisher* to read and publish images from the IMX219 camera, the *Detector* to detect the obstacles, and the *Mapper* to build an occupancy map from the detections. Furthermore, we modified the Local Planner

to use the occupancy map to avoid the obstacles, and we changed the Global Planner to perform a different launch sequence to build the map before calling `test`. At a high level, our approach was to use the images from the IMX219 camera, detect an oriented bounding box of poles in the image using the Detector, use the known diameter of the obstacles to determine their 3D location, build an occupancy map based on those detections using the Mapper, and then navigate around the poles using the occupancy map using the A* algorithm on the occupancy grid. Below we describe the components of this system in detail.

IMX219 Publisher: The IMX219 publisher node reads data from the IMX219 color camera and publishes it to a ROS topic. We used the open-source library JetCam [18] to interface with the IMX219 camera. We used a checkerboard to calibrate our camera using the OpenCV functions `findChessboardCorners` and `calibrateCamera` to find the camera’s intrinsic parameter matrix \mathbf{K} and coefficients used in plum-bob distortion model \mathfrak{D} .

The IMX219 publisher node reads images from the camera at 10 Hz and a resolution of 540×540 for computational efficiency. Next, it undistorts them using OpenCV’s `initUndistortRectifyMap` and `remap` — see figs. 7a and 7b for the result of this undistortion — and publishes that message to the `imx219/image` topic.

Detector: The obstacle detector node takes images from `imx219/image` and detects the yellow cylindrical obstacles shown in fig. 7. The output of this node is a set of points $\{\mathbf{p}_c^{o(k)c}\}_{k=1}^K$ in the camera frame \mathcal{F}_c that represent the center of the obstacles. Each point has an associated color `green` or `red`, depending on the color of the letters on the obstacle. Figure 7e shows the detection of a red obstacle from log-replay data, indicated by the color of the bounding box. For each incoming image, the Detector works by the following steps:

1. Convert the image to the HSV color space.
2. Perform histogram equalization on the saturation channel, increasing the “spread” of saturation values. Histogram equalization is useful so that most of the pixels from the obstacles, which are the most saturated part of the image, will have a saturation ≥ 240 after histogram equalization, independent of the lighting.
3. Threshold the HSV image to find the yellow poles, using a lower HSV value of (10, 240, 0) and an upper HSV value of (80, 255, 255), which selects the most saturated yellow and green pixels at any brightness. See fig. 7c for an example of the resulting masked pixels.
4. For each contour in the resulting mask, we
 - (a) Go to the next contour if the contour area is less than 1000 px^2 , excluding spurious detections that are too small.
 - (b) Fit a minimum area rectangle to the contour. Figure 7d shows all the minimum area rectangles from the image, including those with an area less than 1000 px^2 .
 - (c) If any rectangles aren’t oriented approximately vertically ($\geq 10^\circ$ degrees from 0° or 90°), or the aspect ratio w/h of the rectangle is ≤ 2 , continue with the next contour and discard this one. This requirement leverages the prior that the poles are approximately oriented horizontally and are wider than they are tall, as seen from the IMX219 image.
 - (d) Go to the next contour if the minimum area rectangle has a corner too close to the top or bottom of the image (we used a 64 pixel threshold on both sides). This requirement prevents us from detecting a pole that is only half in our field of view, which would cause inaccuracies in subsequent depth estimation.
 - (e) At this step, we count the bounding box as a detection. Figure 7e shows an example detection. To determine if the obstacle has green or red lettering, we calculate the percentage of pixels within the bounding box with a hue value in the range [40, 80] (green hues). If this is above a tuned threshold value (we use 3%), we record the color as green. Otherwise, we record the color as red.
 - (f) The center of the detected bounding box is turned into a point \mathbf{p}_c^{oc} in the sensor frame by back-projecting the box using its known width of 0.3 m.
 - (g) Each detected point and color are published to the topic `detector/points`.

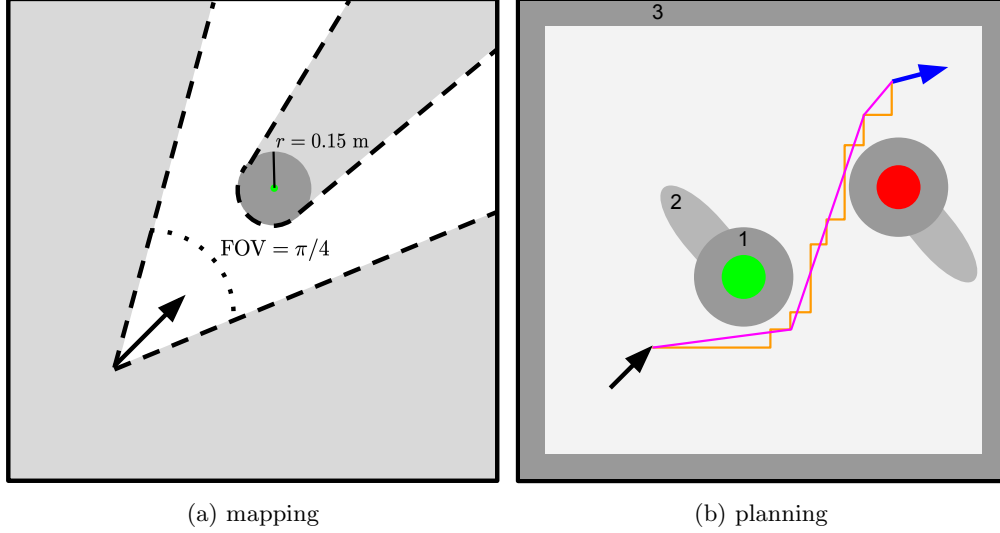


Figure 8: Schematics of the mapping and planning process. Figure 8a shows the logits update regions for pole detection: the green dot is the detection point, the white dashed region updates towards less evidence of occupancy and the dark grey circle updates towards more evidence of occupancy. Figure 8b demonstrates the planning process using the occupancy map: the green and red circles denote the occupancy maps, the black arrow is the start point and the blue arrow is the finish point. The darker the grey, the higher the cost used in A* planning. The orange path indicates the path returned by A* on a grid, and the purple path is the final smoothed path.

The back-projection procedure is as follows:

1. Calculate the depth $d = h_p f_y / (h)$, where $h_p = 0.3$ m is the cylinder diameter, h is the height of the oriented bounding box in pixels, and f_y is the focal length expressed in vertical pixels $f_y = \mathbf{K}_{2,2}$.
2. Calculate the 2D pixel coordinates of the center of the bounding box $\mathbf{c} = [u_c, v_c, 1]^T$, and compute $\mathbf{p}_c^{oc} = d\mathbf{K}^{-1}\mathbf{c}$.

Mapper: The mapper node takes detection points and their colors from `detector/points` and maintains and outputs two occupancy grids, one for red and one for green obstacles. It stores two 2D arrays of logits, `logits_green` and `logits_red` for the green and red obstacles, respectively. Both maps are square with a side length of 9.8 m (to match the width of MY580) and a resolution of 0.2 m, with \mathcal{F}_m located in the center of each map. The node receives detection points with associated colors and updates the maps for each. We describe the procedure for one point \mathbf{p}_c^{oc} :

1. Use the TF tree to find the transformation \mathbf{T}_{mc} , and transform the detection point into the map frame $\mathbf{p}_m^{om} = \mathbf{T}_{mc}\mathbf{p}_c^{oc}$.
2. Next, use \mathbf{T}_{mc} and \mathbf{p}_m^{om} to construct the occupancy update pattern shown in fig. 8a. Namely, in a circle of radius 0.15 m around the location of \mathbf{p}_m^{om} we add $\alpha = 1.0$ to `logits_green` if the detection is green or to `logits_red` if the detection is red. Further, for a 45° field of view (FOV) in the camera's direction, not including the occupied or occluded regions, we add $\beta = -0.1$ to both logit maps. We cap the logit maps at a minimum of -4.0 and a maximum of 20.0 to prevent overconfident predictions.
3. Convert the logit maps to probabilities and publish these as ROS `OccupancyGrid` messages to the topics `mapper/red` and `mapper/green`. We also threshold the occupancy maps at a probability value of 0.5 and print the disjunction (logical or) of the resulting binary masks to the terminal in ASCII format, where “#” indicates an occupancy probability greater than 0.5 and “.” denotes otherwise. Figure 3d shows this printed map, which helps the drone pilot assess the quality of the map in real-time and abort the flight if CAPTOR makes erroneous detections.

We tuned α and β based on log replay to ensure the occupancy updates were accurate and well-calibrated (i.e., not overconfident or underconfident).

Local Planner: The Local Planner was modified to use the red and green occupancy maps in its planning procedure. It stores a local copy of `map_red` and `map_green` by subscribing to `mapper/red` and `mapper/green`. The Local Planner also stores its current local path `local_path`, which we initialize to `None`. Given the current pose of the drone \mathbf{T}_{md} and the desired pose $\tilde{\mathbf{T}}_{md}$, the Local Planner computes a position command $\tilde{\mathbf{T}}_{mb}$ to send to the drone according to the following steps:

1. Initialize a 2D cost map `costs` of ones, the same shape as `map_green`, to be used by an A* planning algorithm. Thus, the nominal cost to travel from cell to cell (without allowing for diagonals) is 1.
2. Turn the current occupancy maps from probabilities to binary masks `mask_green` and `mask_red` using a threshold of 0.5. The positive regions of this mask are to `inf` in `costs` (indicating plans should never enter these regions). Figure 8b shows these regions with red and green circles.
3. Dilate (enlarge) the positive regions of the masks using OpenCV `dilate` by 0.5 m, and set these dilated regions of the mask to a large cost (we use 100) in `costs`. These dilated *buffer* regions are labeled (1) in fig. 8b. The buffers incentivize the A* planner to take wide paths around obstacles but do not make it impossible to come close to an obstacle if required.
4. Use the OpenCV function `dilate` to directionally dilate the occupancy masks perpendicular to the direction the drone is facing. For the green obstacles, we dilate to the left, and for the red obstacles, we dilate to the right, according to the direction the drone faces. Figure 8b indicates these *directional buffer* regions with the label (2). The dilated regions of the mask have their cost increased in `costs` (we add 50). These directional buffers incentivize the A* planner to turn counterclockwise around the green obstacles and clockwise around the red obstacles.
5. Set the cost of regions within 0.5 meters of the walls to a large value (we use 100) to incentivize the A* planner to stay away from the edges of the map where there are walls.
6. If the current `local_path` is `None`, the current `local_path` is empty (i.e., the drone has reached all of the poses along the current local path), \mathbf{T}_{md} is sufficiently close to the last pose on `local_path` (checked with `pose_is_close`), or the current `local_path` results in a collision (by checking if any point along `local_path` is in a cell in `cost` with infinite cost), then we compute and save a new `local_path` according to the following procedure:
 - (a) Convert \mathbf{T}_{md} and $\tilde{\mathbf{T}}_{md}$ into grid indices, and pass these indices along with `costs` into an A* planner that operates on a grid. We use the blazingly fast `pyastar2d` [11], which returns a list of 2D points (tuples) representing the path it has found, or `None` if no path exists:
 - i. If no path exists, the Local Planner returns the previously sent $\tilde{\mathbf{T}}_{mb}$, forcing the drone to maintain its current position. By staying still, the Detector and Mapper should have more time to update the occupancy map, such that the Local Planner can find a path.
 - ii. If `pyastar2d` finds a path, we smooth the path. Figure 8b illustrates the original path in orange and the smoothed path in magenta. The smoothing algorithm is: (i) Start at the first waypoint $i = 0$, and consider the subsequent point $j = 1$. (ii) If the line between the point at index i and j is collision-free, then increment $j = j + 1$. Otherwise, append point at index i to the smoothed path, set $i = j$, $j = i + 1$. Repeat from step (ii) until j reaches the end of the path, then append the point at j to the smoothed path. We check for collisions on the line connecting points at indices i and j by querying `costs` at all cells along that line and checking if any costs greater than one are found (i.e., we don't shorten the path to pass through high-cost regions). This smoothing procedure prevents the drone from traveling on a "staircase" along the grid cells found by A* (the orange path in fig. 8b).
 - (b) We convert the resulting list of grid indices to a list of poses in the map frame, where the altitude z linearly interpolates between the z value of \mathbf{T}_{md} and $\tilde{\mathbf{T}}_{md}$ and the yaw angle of CAPTOR is set such that the IMX219 camera points in the direction the drone is flying so it can see oncoming obstacles. Further, for smoother drone rotation, we add poses that interpolate any rotations at

a yaw resolution of 5° . We set the final pose on the path to $\mathbf{T}_{m\bar{d}}$ to fix the discretization error caused by planning on a grid.

7. Otherwise, if we don't need to re-plan, we use the stored `local_path`. If \mathbf{T}_{md} is sufficiently close to the first pose on `local_path`, then we remove the first pose in `local_path`.
8. The first pose on `local_path` — after transformation $\mathbf{T}_{mb} = \mathbf{T}_{md}\mathbf{T}_{db}$ — is sent as a position command to the PX4.

Global Planner: We modified the Global Planner launch sequence for CT4 to build the occupancy maps before calling `test`. On launch, the Global Planner controls the drone to hover at 1.5 m and then yaw slowly 360° , such that the IMX219 camera sees the entire surroundings. At this point, the pilot can proceed to `test` if the ASCII occupancy map in the terminal includes all of the obstacles in the scene.

5 Results and Evaluation

5.1 Stationkeeping

For stationkeeping, CAPTOR was able to meet the requirements by staying within the specified tolerances ($\pm 5^\circ$ for yaw, $1.5\text{ m} \pm 0.1\text{ m}$ for altitude, and $\pm 0.15\text{ m}$ for horizontal translation) for 30 s. Thus, our score for CT2 was 100%.

A critical lesson from CT2 was that the most significant uncertainty in the pose estimate from the T265 is in the depth direction. The results in fig. 9 illustrate this: during this test, the y axis of the world frame was aligned with the depth direction of the T265, and the y component of the drift Δy is far larger than Δx or Δz . To mitigate this error, it helped to place objects like chairs in front of the T265 so there were more features at a smaller depth. This learning carried over to CT3.

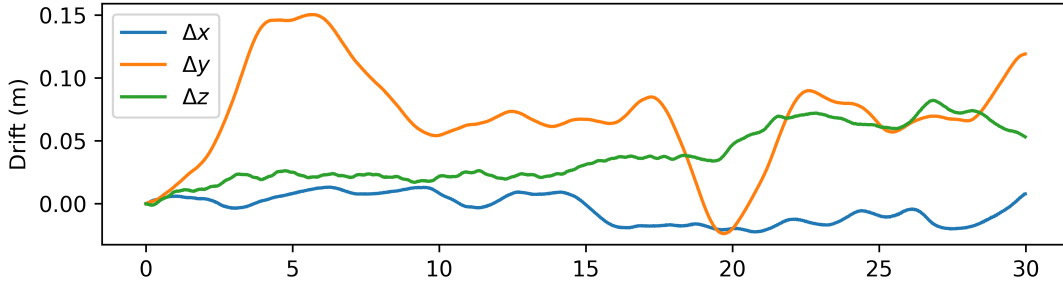


Figure 9: Translational deviation during CT2 without Vicon. $\Delta x, \Delta y, \Delta z$ are the drift in the x, y, z values of \mathbf{p}_w^{dw} from the beginning of `test`. During this test, the depth direction of the T265 was aligned with the y axis of the world frame.

5.2 Waypoint Navigation

For waypoint navigation with Vicon, CAPTOR was able to meet the requirements by hitting all of the waypoints within 60 s as shown in fig. 10b, so our score for this portion of CT3 was 100%.

For waypoint navigation without Vicon, we noticed a significant drift in the pose estimate \mathbf{T}_{mb} from the T265. To combat this, we investigated a few potential causes and solutions:

- We noticed that from a birds-eye view during practice trials, the actual drone trajectory appeared slightly rotated relative to the desired trajectory, as shown in fig. 10a. This rotation indicated a miscalibration in the transformation \mathbf{T}_{db} between the Vicon dots frame and the base frame. We attributed this to small rotations of the Vicon dots holder about their single screw anchor point. To combat this, we requested a new Vicon dots holder with two anchor points to prevent unintended rotation.

- Once the T265 and bridge were active, we would walk the drone around the room to build up a map of keyframes in the T265 SLAM system before launching the drone. We also placed more features in the MY580 testing arena, such as chairs. This reduced drift, so we used it when evaluating CT3.
- We found that flying the drone more slowly reduced the drift, so we completed CT3 with a maximum horizontal velocity of 0.4 m/s, which was the minimum speed that would also satisfy the time requirement. We hypothesize that this reduces IMU error and noise in the T265 and the PX4, improving localization performance.
- We tried flying the drone to the corners of a cube with side length 0.3 m at each waypoint to increase the chance of hitting the waypoint. We found that this did not appreciably improve performance, and it caused us to violate the time limits, so we did not use this solution when evaluating CT3.
- Based on the hypothesis from CT2 that the worst drift was along the depth direction for the T265, we tried flying the drone such that the T265 faced perpendicular to the current flight direction. We found that this did not improve localization performance and worsened it in some cases (e.g., if the T265 were to face toward the netting in MY580). Thus, when evaluating CT3, we used a constant orientation of the drone for the entire test.

After implementing these solutions, we were able to reach all the waypoints in under 60 s as shown in fig. 10b, so our score for this portion of CT3 was 100%. However, this was not a consistent result, and drift was still an issue we dealt with during CT4.

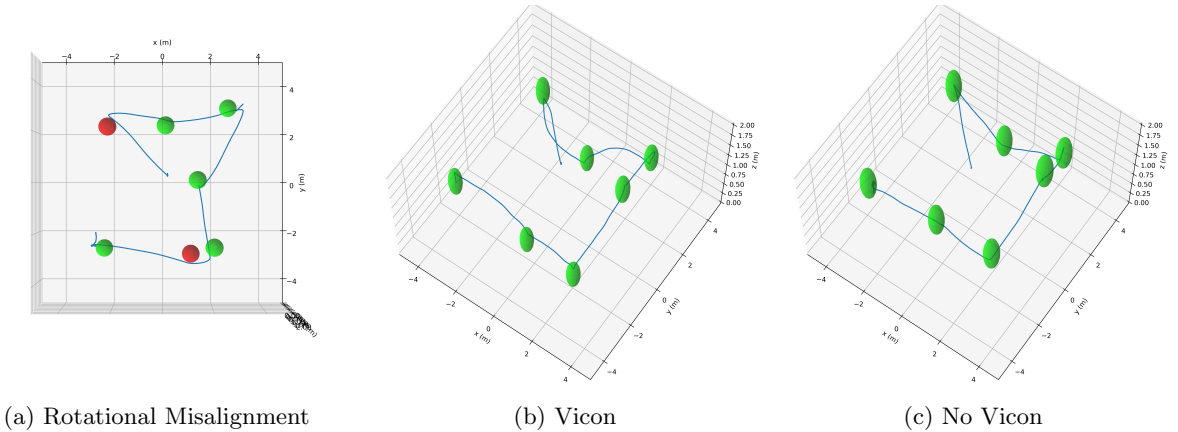


Figure 10: Figure 10a is a practice CT3 run where the rigid transformation of the Vicon dots relative to the PX4 \mathbf{T}_{bd} was miscalibrated due to a rotated Vicon mount. Figures 10b and 10c present our evaluation results for CT3, both with and without Vicon in the world frame \mathcal{F}_w . The spheres are centered on the waypoints with a radius of 0.35 m, which are colored green if the drone reaches them and red otherwise.

5.3 Online Obstacle Avoidance

CT4 followed our planned approach, with the most significant issues being bug fixes. Figure 11 shows our results with and without Vicon positioning. Our biggest challenge with CT4 was a faulty power distribution board, which caused erratic flight patterns. We thought there was an issue with our software and spent much time looking in the wrong place. Only when our motors burnt out did we realize that it was not a software issue. After replacing the power distribution board, ESCs, and motors, we completed CT4.

With Vicon, we were able to reach all waypoints (see fig. 11a) but flew for 84 s, exceeding the 70 s time limit. This resulted in a score of 98%.

Without Vicon, again, the biggest issue was the drift in the estimate from the T265. Figure 11b shows our best run, where we reached 6/7 waypoints in 98 s. This resulted in a score of 80%.

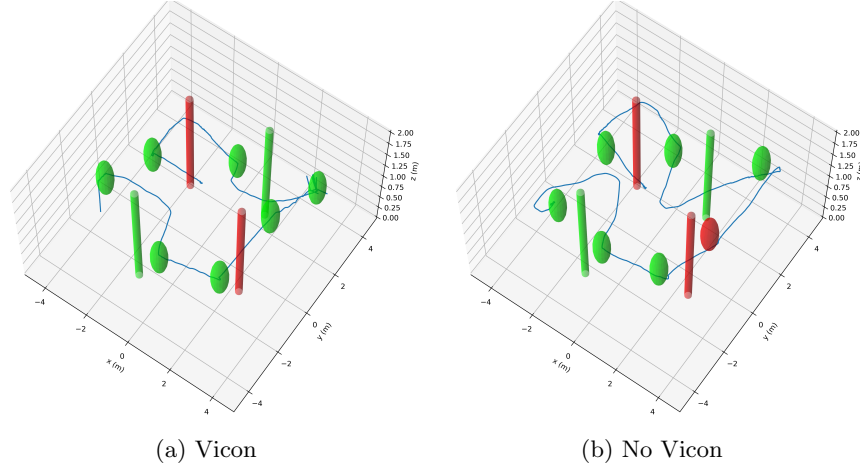


Figure 11: Results from CT4, both with and without Vicon in the world frame \mathcal{F}_w . The cylinders are at the ground truth obstacle locations, colored based on the color of the lettering on the real cylinders.

6 Lessons Learned

Below, we discuss the lessons we learned throughout our design process.

Avoid crashing: The cost of crashing the drone is very high; rebuilding takes a lot of time and energy that we would otherwise spend refining the design. For this reason, our first lesson learned is that flight safety should be a top priority.

Most of our crashes were from operator error during manual control (e.g., after taking control of the drone to avoid hitting an obstacle). Investing the effort to learn how to fly the drone manually would have saved us lots of time later in the development process.

To combat our lack of manual flight skill, midway through the project, we set all available manual control modes on the controller to “position hold mode”. This prevented us from accidentally switching to “altitude hold mode,” which is very hard to control manually. We think this change prevented many possible future crashes.

Hardware is hard: The issues that were the most difficult to fix were due to faulty hardware or hardware acting unexpectedly. In section 5.2, we discussed the undesirable rotation of the Vicon dots, and in section 5.3, we discussed the faulty power distribution board that resulted in erratic flight patterns, but there were other hardware-related issues.

An example is the T265 intermittently losing its pose estimate, likely due to high vibrations (as discussed in [this GitHub issue \[19\]](#)), causing the drone to fly in a random direction suddenly. Again, we were trying to find a problem with our code that would have caused this issue.

In the future, when working with hardware, we should start debugging by checking the “inputs” to the system (e.g., the data from the hardware), both manually and with automatic assertions that ensure the data is reasonable (and throw warnings if it is not).

Use your priors: Throughout this challenging design process, we learned that leveraging the “given” information, or priors, is vital to speed up development. For example, developing a general monocular object detection framework for CT4 would have been extremely difficult and lengthy. Instead, we used all the prior information we had about the obstacles we were trying to detect — aspect ratio, hue, saturation, diameter, and orientation — which greatly simplified the task.

Simulation is vital: Despite hardware-related issues, this project reaffirmed the importance of simulation. The amount of testing and debugging we could get done in simulation and with log replay was crucial to

our success. The reason *most of our real-life issues were in hardware* was *because* we worked out all of the software-related with simulation and log replay.

Shareable software: Building well-documented software that is easy to install and share with others is vital for cross-team development. The more barriers to entry into the software stack, the harder it is for team members to take the initiative and contribute to development.

References

- [1] 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)*, vol. 3, 2004, 2149–2154 vol.3. DOI: [10.1109/IROS.2004.1389727](https://doi.org/10.1109/IROS.2004.1389727).
- [2] C. L. Dym, *Engineering design: A project-based introduction*. John Wiley & Sons, 2013.
- [3] L. Meier, D. Honegger, and M. Pollefeys, “Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 6235–6240. DOI: [10.1109/ICRA.2015.7140074](https://doi.org/10.1109/ICRA.2015.7140074).
- [4] L. Meier, *Px4 drone autopilot*, 2023. [Online]. Available: <https://github.com/PX4/PX4-Autopilot>.
- [5] M. Juhasz, *Intel realsense gazebo/ros*, 2020. [Online]. Available: https://github.com/nilseuropa/realsense%5C_ros%5C_gazebo.
- [6] E. M. Benavides, *Advanced engineering design: An integrated approach*. Elsevier, 2011.
- [7] Qengineering, *Jetson-nano-ubuntu-20-image*, Mar. 2023. [Online]. Available: <https://github.com/Qengineering/Jetson-Nano-Ubuntu-20-image>.
- [8] Stanford Artificial Intelligence Laboratory et al., *Robotic operating system*, version ROS Noetic Ninjemys, 2020. [Online]. Available: <http://wiki.ros.org/noetic>.
- [9] G. Bradski, “The OpenCV Library,” *Dr. Dobbs’s Journal of Software Tools*, 2000.
- [10] C. R. Harris, K. J. Millman, S. J. van der Walt, et al., “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2). [Online]. Available: <https://doi.org/10.1038/s41586-020-2649-2>.
- [11] H. J. Weideman, *Pyastar2d*, 2022. [Online]. Available: <https://github.com/hjweide/pyastar2d>.
- [12] V. Ermakov, *Mavros*, 2018. [Online]. Available: <http://wiki.ros.org/mavros>.
- [13] T. D. Barfoot, *State Estimation for Robotics*, 2nd. USA: Cambridge University Press, 2022, ISBN: 1107159393.
- [14] S. Waslander, *Me 597: Autonomous mobile robotics section 8 – mapping i*, 2020. [Online]. Available: http://wavelab.uwaterloo.ca/sharedata/ME597/ME597_Lecture_Slides/ME597-6-MappingI.pdf.
- [15] S. Waslander, *Lecture 20: Extended & unscented kalman filters*, Mar. 2023.
- [16] D. Scaramuzza and Z. Zhang, *Visual-inertial odometry of aerial robots*, 2019. arXiv: [1906.03289 \[cs.RO\]](https://arxiv.org/abs/1906.03289).
- [17] J. Qian, *Rob498-flight*, 2023. [Online]. Available: <https://github.com/utiasSTARS/ROB498-flight>.
- [18] A. Kulkarni, *Jetcam*, 2020. [Online]. Available: <https://github.com/NVIDIA-AI-IOT/jetcam>.
- [19] Y. Rumyantsev, *T265 is losing position under vibrations*, 2019. [Online]. Available: <https://github.com/IntelRealSense/librealsense/issues/4176>.