École Polytechnique Fédérale de Lausanne

Aerial Robots Manual

Last update: March 12, 2018

Contents

TP 0: Introduction

During these practicals, you will learn how to develop algorithms for autonomous drone missions using the PX4 autopilot. We will use the Gazebo simulator for visualization and evaluation of the missions but in principle, your code is able to run on a physical drone.
By the end of these practicals, your drone will be able to complete a challenging mission. At first, you will navigate through multiple waypoints using your own implementation of a position controller. Once the waypoints are cleared, your drone will have to land on a static platform with unknown position and height using a sonar. Finally, you will have visually track the position of a moving truck and land on its cargo bed.
The aim of the following introduction is to explain the most important tools and concepts you need to get started with drone development.

0.1 PX4 autopilot

The PX4 autopilot is an open source project originally developed at ETH Zurich that has since gained wide adoption among professionals, researchers, and hobbyists alike. PX4 is growing rapidly and many drone manufacturers such as Parrot and Intel are releasing PX4 compatible commercial drones. Broadly speaking, PX4 can be categorized into two main components.
The first component, the *flight stack*, is responsible for integrating various sensor measurements to estimate the state of the drone, and ultimately to control its flight. Please refer to Fig. 0.1 for an overview of the PX4 flight stack.

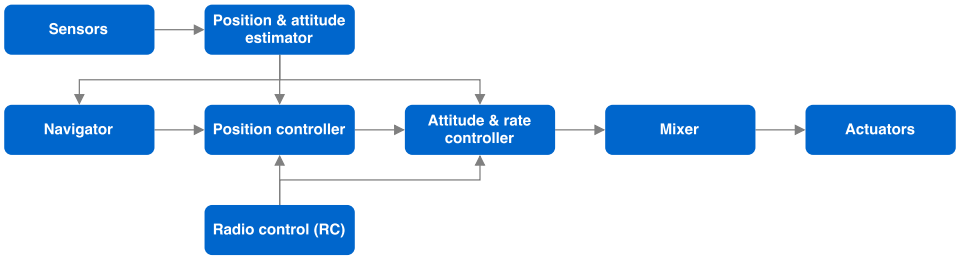

Figure 0.1: Overview of the PX4 flight stack. An **estimator** takes sensor inputs, e.g. from the IMU, to compute the vehicle state. A **controller** takes a setpoint and an estimated state, e.g. the desired and current position. Its goal is to adjust the estimated state such that it matches the setpoint. A **mixer** takes force commands, e.g. turn right, and translates them into individual motor commands.

The second component, the *middleware*, is a general robotics framework which provides drivers for various hardware components and implementations of communication protocols to send and receive messages in real-time.
For these practicals, you will not modify the existing flight stack, but work on external applications called `dronecourse` that will run along the flight stack.
For further information about PX4, please refer to the developer guide at `dev.px4.io/en/`.

0.2 Gazebo simulator

Gazebo is an open source project and the de facto standard simulator of the robotics community. Gazebo includes many features such as dynamics simulation using various physics engines, high-quality rendering of environments, and realistic modeling of sensors and noise. The simulation can be interfaced with plugins through the Gazebo API and many pre-built robot models are included. Fig. 0.2 contains a screenshot of the main Gazebo window with a simulated quadcopter.

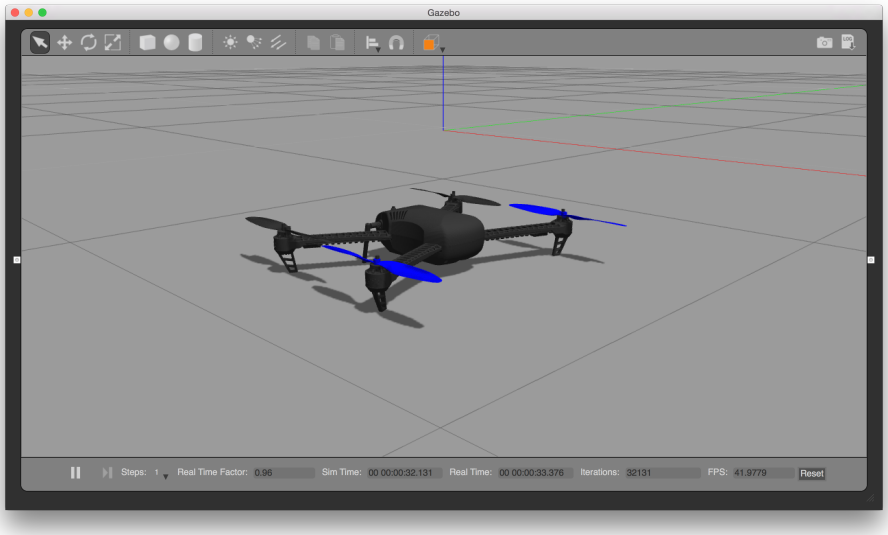

Figure 0.2: Screenshot of the Gazebo simulator. The robot model is a 3DRobotics IRIS quadcopter which you will be using in the practicals.

For further information about Gazebo, please refer to the tutorials at gazebosim.org/tutorials.

0.3 Documentation

You can generate the documentation using Doxygen as follows:

```
host> make dronecourse_doxygen
```

This command will create a subfolder `Documentation/html/` which contains an `index.html` file that you can open with a browser of your choice.
Much of the documentation is not relevant for our purposes but you can use it to learn more about specific parts of PX4. For instance, to learn more about `uorb`, you can navigate to the *Namespaces* tab, find *uORB* near the bottom of the list, and select the *Manager* class. Similarly, you can find the documentation of the `Matrix` class we will use extensively throughout these practicals.

0.4 Getting started

In the following sections, we will cover how to start the simulation and how to communicate with the drone using the PX4 console. The most important use cases of the PX4 console are the use of various built-in modules for displaying and setting drone parameters, as well as inspecting messages sent across the autopilot.

Starting and quitting the simulation Open up a terminal (in Ubuntu, you can use the keyboard shortcut `Ctrl + Alt + T`) and navigate to the `PX4` folder using the `cd` command. To start the drone course simulation environment, issue the following command:

```
host> make dronecourse_gazebo
```

This command will compile all necessary PX4 components and start the simulator. After a few seconds, the drone will arm itself automatically, take off from the ground, and hold its position in mid-air.
Remember that you will have to re-run this command whenever you make changes to your code! In order to quit both the simulation and PX4, simply use the keyboard shortcut `Ctrl + C` or issue the `shutdown` command inside the same terminal window.

Using the PX4 console and its modules In the same terminal window you used to start the simulation, you will see various PX4 log messages. If you press `return` inside the terminal window, you will be greeted with the PX4 console which allows you to send commands to the drone via modules. Use the `help` module in order to get a list of all the other built-in modules.

```
pxh> help
```

For our purposes, we will mainly be using the built-in `param` module and the application we are developing ourselves called `dronecourse`.
Simply typing the module's name will print more information about its purpose and usage. Just like in any regular UNIX shell, you can use the arrow up and down keys to cycle through previous commands.

Displaying and changing PX4 parameters PX4 uses a variety of onboard parameters that are stored on the drone but are easily accessed and edited via the PX4 console's `param` module. The parameters can represent all kinds of information such as the drone's maximum operating velocity, controller gains, or various threshold values. To display all parameters that are currently stored on the drone, you can issue the following command:

```
pxh> param show
```

However, if you are only interested in the parameters related to the local position estimator (`lpe`) application, you can issue:

```
pxh> param show LPE_*
```

Here, the `*` symbol acts a wildcard. To change a paramter permanently, you cann issue the following command:

```
pxh> param set PARAM VALUE
```

Here, `PARAM` is the parameter you would like to change and `VALUE` its desired new value.

Listing uORB topics and inspecting individual messages PX4 uses the uORB message bus to enable communication between different modules of the autopilot. For instance, the position and attitude estimator sends the drone's state to the position controller via uORB messages. uORB follows the publish-subscribe principle, meaning that modules can publish messages on a specific topic to be read by other modules. Similarly, modules can subscribe to a uORB messages simply by its topic name and message type. The `uorb` module is asynchronous, meaning that it reacts immediately as soon as new data is available.
To show a list of all uORB messages that are currently published on the drone, issue the `uorb` command:

```
pxh> uorb top
```

To inspect messages that are sent on a specific topic, you can use the `listener` module. For instance, you can take a look at the current estimated drone position by issuing:

```
pxh> listener vehicle_local_position
```

The previous command will print the drone's current local position and various other related information to the console.

Question 1:
Using the `uorb` and `listener` modules, can you find out at which 3D position (local and global) the drone is hovering after you start the simulation? At which frequencies are the local and global positions being published?

0.5 Hello sky

So far, you have only been running modules that are already provided by the PX4 autopilot. Now it is time to write your very first onboard application yourself.
Follow the tutorial at dev.px4.io/en/tutorials/tutorial_hello_sky.html.
You can ignore the sections about file setup and uploading the code to the physical autopilot hardware.
To verify that your first onboard application is working, you can issue the following command to start the simulation (followed by the name of the application once you are inside the PX4 shell):

```
host> make dronecourse_hellosky
```

0.6 Unit tests

For each practical, we provide you with unit tests as a sanity check so you can test your implementation before you start developing the final task. The test are located in the `src/dronecourse/tests` directory and you can run them by typing the following command:

```
host> make dronecourse test_all
```

TP 1: Waypoint navigation

In this practical, you will learn how to write a position and trajectory controller in order to navigate through multiple waypoints. Fig. 1.1 provides a visual description of the end goal of this exercise.

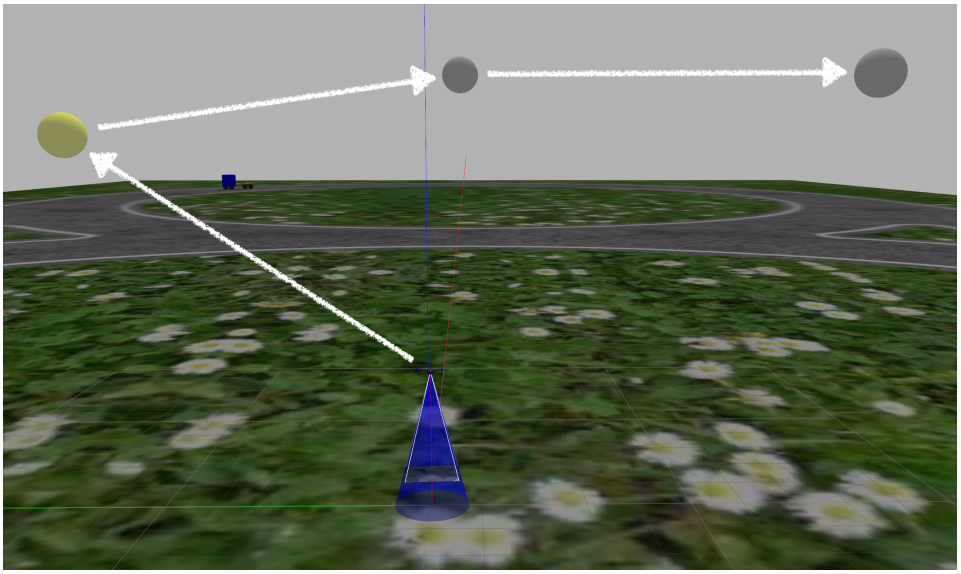

Figure 1.1: Waypoint navigation. Your goal is to navigate the drone through all three waypoints, one after the other. The next waypoint to be reached is shown in yellow. As soon as you fly close enough to the center of the waypoint, it turns green and the next waypoint will be highlighted in yellow.

At first, we will focus on the position controller, whose only responsibility is to move the drone to a specified setpoint position provided via the PX4 console. Once the position controller is functional, we can turn to the trajectory controller, whose task it is to navigate through multiple waypoints in succession.

1.1 Position controller

The main task of the position controller is to take a 3D setpoint position as input and output a 3D velocity command that moves the drone in the direction of the setpoint. As is shown in Fig. 1.2, the downstream controllers then transform this 3D velocity command into an attitude command and ultimately to motor commands.
To calculate the desired velocity command, we need to access the current position estimate which is published as a uORB topic.

1.1.1 Implementation

In the following sections, you will learn how to implement a position controller using the PX4 middleware and flight stack. First, we will cover how to access the local position estimate which is published in form of uORB messages. To achieve this, we have to subscribe to the corresponding uORB topic, check if new

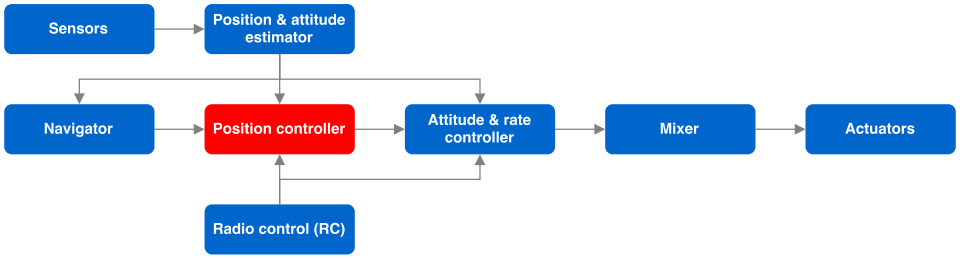

Figure 1.2: Flight stack with highlighted **position controller**. We will be given a setpoint position from the **navigator** and an estimate of our current position from the **position and attitude estimator**. All communication between system components happens via the uORB message bus.

data is available, and ultimately write the values into local variables. As soon as we have the local position estimate, we can calculate the target vector from the position estimate and the setpoint position.
Since we might want to change the controller's behavior while the simulation is running, we will implement the controller gain as an onboard parameter. Finally, we have to send the velocity command off to PX4.

For your implementation of the position controller, you will be working on the following files:

- `src/dronecourse/controllers/PositionCtrl.hpp`
- `src/dronecourse/controllers/PositionCtrl.cpp`

All places where you have to write your own code are marked with a `// TODO` comment.

This section will guide you through the implementation step by step but as the semester progresses, you will get more and more liberty in the implementation.

Subscribing to a topic and reading its messages

You have been given a class skeleton of the position controller called `class PositionCtrl`. Inside the position controller, `void update()` is called periodically to fetch both the current position estimate and selected onboard parameters, i.e. with the functions `void update_subscriptions()` and `void update_parameters()`, respectively.

You will be using the following functions for your implementation:

- `int orb_subscribe(const struct orb_metadata *meta)`
  Subscribe to a topic. Returns a `handle` to the subscription.
- `int orb_copy(const struct orb_metadata *meta, int handle, void *buffer)`
  Fetch data from a topic. Writes the data into the `buffer`.
- `int orb_check(int handle, bool *updated)`
  Check whether a topic has been published to since last `orb_copy`.

Since we want to know the drone's estimated position, we need to subscribe to the `vehicle_local_position` topic. A subscription is nothing but a `handle` that allows us later to receive messages on the topic.
In the constructor of our position controller, we can subscribe to the estimated position as follows:

```
_local_pos_sub = orb_subscribe(ORB_ID(vehicle_local_position));
```

Here, `_local_pos_sub` is a member variable that is already defined in the header of our position controller. The `ORB_ID` macro generates the `orb_metadata` struct from a topic name and returns a reference to it.
Now that we are subscribed, we can check if new local position messages are available on the topic. In `update_subscriptions()`, we can achieve this with the following lines:

```
bool updated;
orb_check(_local_pos_sub, &updated);
```

The variable `updated` will be set to `true` if a new message is available and to `false` otherwise. As soon as a new message is ready, we can copy its content into a local variable using

```
vehicle_local_position_s local_pos;
orb_copy(ORB_ID(vehicle_local_position), _local_pos_sub, &local_pos);
```

This will set all fields of our local position struct to the updated values.

> All the fields of the `vehicle_local_position` message are defined in the following file:
>
> - `msg/vehicle_local_position.msg`
>
> For now, we are only interested in the fields `x`, `y`, and `z`.

Now that we have the estimated local position, we can set the corresponding member variable which is already defined for us in the header file. We will represent the local position internally as three-dimensional vector of floating point numbers where each element can be accessed using parentheses. The values can be extracted from the struct as follows:

```
// access to the x coordinate
_current_pos(0) = local_pos.x;
```

Now the current position variable will be updated whenever a local position message is available.

> PX4 provides its own logging commands that you can use to debug your implementation:
>
> - `PX4_INFO("Info message with integer %i", i)`
> - `PX4_WARN("Warning message with double %f", d)`
>
> Note that these macros take a format string just like the `printf` function of the standard library.

To verify that your implementation is working, you can print values to the PX4 console using its built-in logging functionality.
You are now ready re-run the simulation. Once the drone hovers in mid-air, issue the following command to start the position controller:

```
pxh> dronecourse pos 0 10 -10
```

This command will start our application and send the drone to the specified coordinates. Note that -10 is 10 meters above ground since our z axis points downwards. Since we have not finished our implementation, the drone will continue to hover but the console should output the drone's current location. Since printing the drone's current location spams the console, remove the corresponding lines from your code.

Calculating velocity command

Now that we have access to the current position estimate, we need to calculate the target vector that points from our location to the setpoint position. For simplicity, we will implement a proportional controller to move the drone from its current position to the setpoint.

> The `Matrix` class provides the following shortcuts for performing calculations:
>
> - You can perform vector additions and subtractions with the operators `+` and `-`.
> - You can multiply or divide a vector by a scalar with the operators `*` and `/`.

Firstly, you need to calculate the current position error which is the vector from the drone's current position to the setpoint position.
Secondly, you need to calculate the desired velocity command, which consists of the position error and proportional gain. For now, you can use 0.3 as a hard-coded gain for the controller but we will replace this value with an onboard parameter in the next exercise.
Finally, you need to send the velocity command off to the downstream controllers. This functionality is already implemented in the base class of the position controller.

```
_target_vector = ... // to complete
matrix::Vector3f velocity_command = ... // to complete
send_velocity_command(velocity_command);
```

Your position controller is now ready to be tested. Launch the simulation again and wait for the drone to hover in mid-air. Then launch the position control application as before and send the drone to a position and observe its behavior.

Making gain an onboard parameter

To make it easy to change the gain of the proportional controller during runtime we will implement the gain as a onboard parameter. As for the uORB subscription, we will initialize a handle to the parameter, allowing us to read its value.

> To find a parameter by name and access its value, you can use the following functions:
>
> - `param_t param_find(const char *name)`
>   Finds a parameter by name. Returns `PARAM_INVALID` if it is not found.
> - `int param_get(param_t param, void *val)`
>   Writes the parameter's value into `val`.

Find the appropriate `// TODO` items in the position controller and implement the dynamic update of the `POS_GAIN` parameter. Don't forget to check your parameter for validity and print an appropriate warning

message. Once you have the parameter, use its value to make the proportional gain of the position controller dynamic.
Launch the simulation and send the drone to a position as before. Now you can change the gain of your position controller through the console with the `param` module. Try different parameter values and observe the drone's behavior.

Detecting when target is reached

To integrate your controller into a complete mission with waypoint navigation, we have to detect when a desired position is reached. Since we will probably not arrive at our desired setpoint position exactly, we need to define an acceptance radius. As soon as the drone enters this acceptance radius, it will signal the controller that the desired position is reached. To be able to change the behavior of the drone at runtime, we will implement the acceptance radius as an onboard parameter. We can use the same logic we used to add the proportional gain for the position controller.

For your implementation of this exercise, you will be editing the following file:

- `src/dronecourse/controllers/params.c`

All places where you have to write your own code are marked with a `// TODO` comment.

We will call the parameter for the acceptance radius `POS_ACCEPT_RAD`. While the parameter already exists for the gain, here you will have to add it yourself. Respect the format of the commentary since it can be used by PX4 to label your parameter correctly.
Add a handle for the acceptance radius and a corresponding member variable just like for the proportional gain. Now use the acceptance radius member variable to implement the logic for the `is_goal_reached()` function. You might find the vector `norm()` function useful to implement the decision if the goal is reached. Again, you can use PX4's logging functionality to test your implementation.
Restart the simulation and define a setpoint position as before using the `dronecourse` console module. Again, you can delete any log messages after you've tested your implementation to not spam the console.

1.2 Waypoint navigation

The aim of this exercise is to use the position controller that you have developed in the previous exercise for waypoint navigation. As you can see in Fig. 1.3, you will be working on a higher level of control as in the previous exercise.

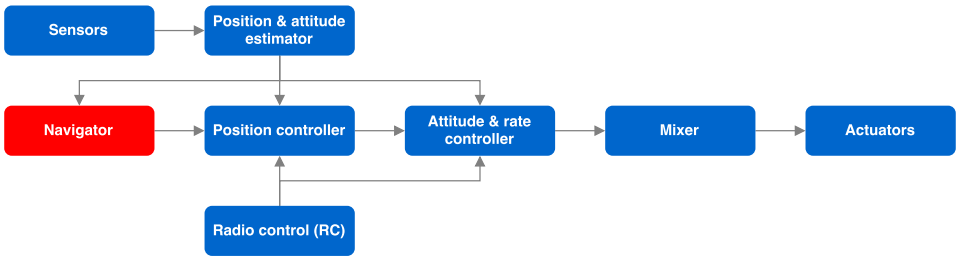

Figure 1.3: Flight stack with highlighted **navigator**. We will be given a set of waypoints, i.e. setpoint positions, which we have to pass on to the **position controller** from the previous exercise at the appropriate times.

For your implementation of the trajectory controller, you will be working on the following files:

- `src/dronecourse/controllers/TrajectoryCtrl.hpp`
- `src/dronecourse/controllers/TrajectoryCtrl.cpp`

All places where you have to write your own code are marked with a `// TODO` comment.

The transparent spheres you can see in the simulator are the waypoints you will have to fly through in order to complete this exercise successfully. The drone must be inside a two meter radius of the center of the waypoint to activate it but generally, the closer you are to its center, the better. Initially, the first waypoint you need to pass through is yellow, whereas all other ones are grey. Once the drone has activated the first waypoint, it changes its color from yellow to green and the second waypoint changes from grey to yellow. The drone must activate all waypoints in the given order.

1.2.1 Implementation

The trajectory controller inherits all the functionalities of the position controller. This means you can call the parent's function `set_position_command()` to set the next desired position of the drone and `update()` to compute the velocity command.

You can use the following functions from the `waypoints.hpp` header in your implementation:

- `int waypoint_copy(int index, matrix::Vector3f *waypoint)`
  Writes the waypoint at `index` into `waypoint`. Returns `INVALID_WAYPOINT` on failure.
- `int waypoint_count()`
  Returns the number of available waypoints

Once your controller is ready, you can start the waypoint navigation mission by issuing:

```
pxh> dronecourse waypoint_navigation
```

If your implementation is correct, the drone will pass through all waypoints autonomously. Congratulations, you have implemented your first autonomous drone mission with PX4!

To change the waypoint coordinates, you can edit the following world file:

- `Tools/sitl_gazebo/worlds/dronecourse.world`

1.3 Conclusion

After completing this practical, you should have a working position and trajectory controller. For the remainder of the practicals, you should be comfortable with all of the following topics:

- Subscribe to uORB topics and receive messages
- Print messages and warnings to the PX4 console
- Create and use your own onboard parameters

TP 2: Sonar landing

In this practical, you will learn how to use a sonar to detect and land on a static platform whose exact position and height are unknown. Fig. 2.1 provides a visual description of the end goal of this exercise.

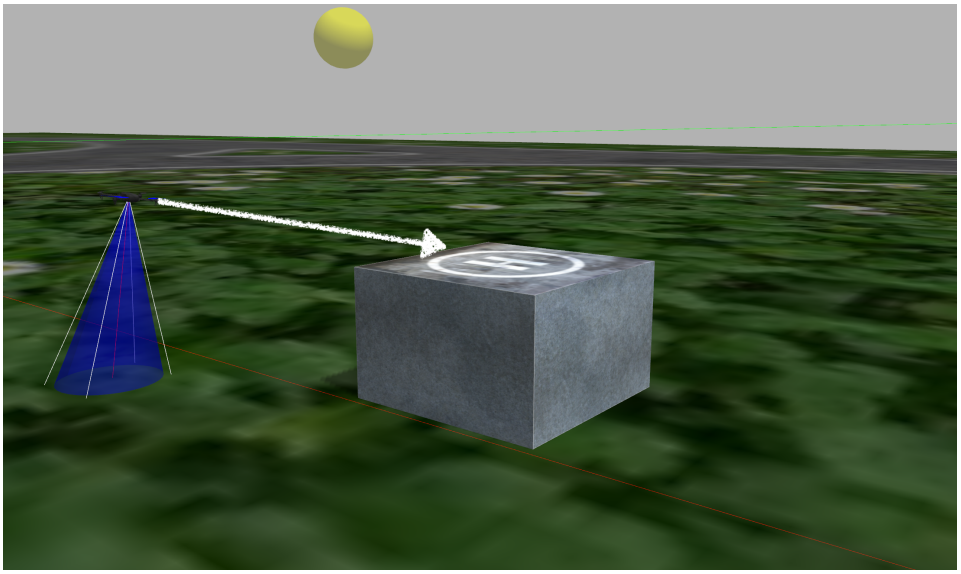

Figure 2.1: Sonar landing. Your goal is to land on a static platform using only a sonar sensor. The exact position of the platform is unknown and has to be determined by searching a region of interest on the opposite side of the dronecourse arena.

The following section will introduce you to the basics of a sonar sensor and how to access its sensor readings. You will then write a search algorithm to detect the platform, find its center, and land on it, using only the sonar measurements.

2.1 Sonar landing controller

Sonar sensors use sound propagation to estimate the distance to objects around them. This is achieved by creating a sound pulse, often called a *ping*, and listening for reflections of the sound signal.
Our particular sonar is located at the bottom of the drone and faces downward. It is visualized with a blue cone under the drone. The sonar measures the distance from the top of the cone to the closest obstacle inside that cone, projected on the center line of the cone as shown in Fig. 2.2.

2.1.1 Implementation

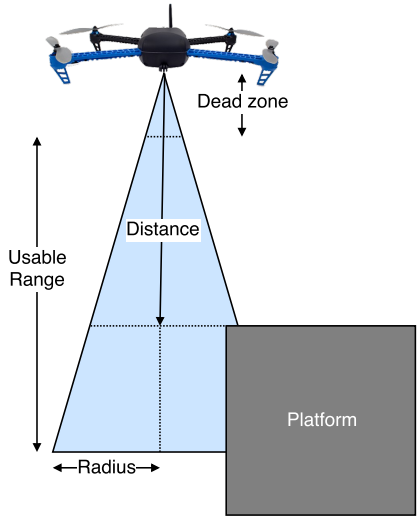

Figure 2.2: Schematic of a sonar attached to the bottom of the drone. We can use our sonar to sense the **distance** to the platform given that it lies in the **usable range** of the sensor. Notice that the sonar has a **dead zone**, meaning it cannot estimate distances that are in too close proximity or too far away. The radius of the sonar at the bottom end of its usable range is $r = 134$cm.

For your implementation of the sonar landing controller, you will be working on the following files:

- `src/dronecourse/controllers/SonarLandingCtrl.hpp`
- `src/dronecourse/controllers/SonarLandingCtrl.cpp`

All places where you have to write your own code are marked with a `// TODO` comment.

Reading the sonar measurements

We will start simple by reading and printing the sonar data to the PX4 console. The sonar is already enabled and publishing the distance to the ground as soon as the simulation is started. You can read the sonar measurements by subscribing to the corresponding topic.

All fields of the `distance_sensor` message are defined in the following file:

- `msg/distance_sensor.msg`

Start by adding a subscription to the sonar data in the constructor of the sonar landing controller. The subscriber handle has already been declared for you in the header file. Once you are subscribed to the topic, update the `_current_distance` variable similar to what you have already done in the position controller. Print the current distance to the PX4 console as you did for the position controller and start the simulation. You can start the sonar landing controller by issuing the following command.

```
pxh> dronecourse sonar_landing
```

Verify that the current distance to the ground is printed to the PX4 console correctly.

Question 2:
Use the current distance measurement from the sonar topic to find out what the operating range of the sonar is. What is the minimum and maximum distance at which the drone can detect obstacles such as the ground?
You can use the `dronecourse pos` command to change the drone's altitude and restart the sonar landing controller by issuing `dronecourse sonar_landing`.

Once you found the operating range of the sonar you can delete the print statements.

Detecting the platform

In this section, we will use the sonar measurements to detect the platform which is located close to the center of the second roundabout.
More specifically, the platform is guaranteed to be in a $3m^2$ square region centered around the coordinates (0, 100). The platform itself is a cuboid with a square landing pad of $2m^2$ on its top, guaranteed to be at least $0.8m$ and at most $1.3m$ high. The exact location and height that is chosen randomly when the simulation starts.
Now that you know the platform specifications, the next step is to implement the platform detection algorithm which means to update the `_platform_detected` variable to `true` whenever the platform is under the drone. For this task, you will need to use the current distance measured by the sonar that we have introduced in the previous section. You may need to add subscription to other topics such as the current position of the drone to complete this task.
For testing purposes, you may want to manually set the platform location and height. While the simulation is running, open a second terminal window, move to the firmware folder and issue the following command to find out how to interact with the platform.

```
host> ./Tools/dronecourse_platform.sh -h
```

Start the simulation and the sonar landing controller and move the platform under the drone to check that your algorithm correctly detects it.
You should now have a means to detect the platform whenever the drone hovers above it.

Finding the platform at an unknown location

In this section, you should use the platform detection capability to find the platform at an unknown location. In order to land on the platform safely, we need to estimate its center location as precisely as possible.
You can implement your platform search algorithm in the `bool update_search()` function which should return `true` whenever the drone is above the platform. To find the platform, you should recall what you did for waypoint navigation but using your own waypoints in order to create a scanning pattern. You can find two example patterns in Fig. 2.3 but feel free to implement your own.
You may also need to implement the scanning patterns at different granularities in order to get a good estimate of the center of the platform. By the end of this exercise, your drone should hover above the center of the platform.

Landing on the platform

In this section, our goal is to safely land on the platform by gradually reducing the drone's altitude.
You need to implement the `void update_landing()` function which should return `true` as soon as the drone has landed.

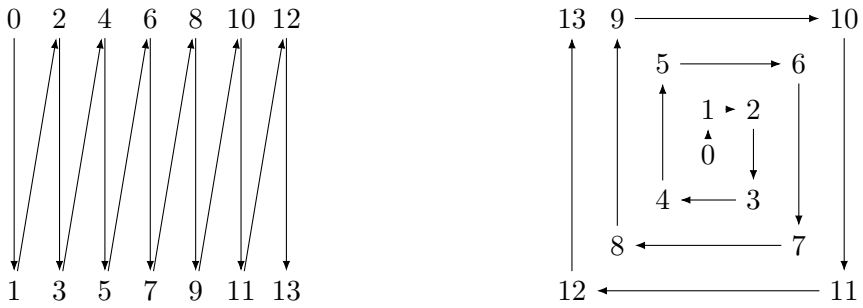

Figure 2.3: Two examples of scanning patterns. Left: a saw tooth pattern. Right: a Ulam spiral pattern.

To signal a successful landing, you should disarm the drone as soon as it is in contact with the platform. Recall that you can not use the sonar to detect when the drone is in contact with the ground because of its dead zone for short ranges. Fortunately, PX4 already provides a land detector application which sends corresponding message in case a landing event is detected.

All the fields of the `vehicle_land_detected` message are defined in the following file:

- `msg/vehicle_land_detected.msg`

Here, we are only interested in the boolean field called `_ground_detected`.

Subscribe to the corresponding uORB topic and update the `_landed` variable which is provided in the header to `true`.

2.2 Conclusion

After completing this practical, you should have a working implementation of a sonar landing controller. This means your drone should autonomously search for the platform, estimate the location of its center, and then safely land on it.

TP 3: Target detection

In this practical, you will learn how to locate a moving target in the arena with an onboard camera that is mounted on a gimbal. For simplicity, the computer vision part of the target detection is provided for you in form of a uORB message containing the truck's distance from the camera and its location in the image frame in form of pixel coordinates. Your task in this practical is to convert these image coordinates to local coordinates, i.e. the same coordinate system that the position controller uses to navigate the drone as visualized in Fig. 3.1. You will need to take into account both the drone's attitude (position and orientation) as well as the gimbal orientation to determine the target location correctly.

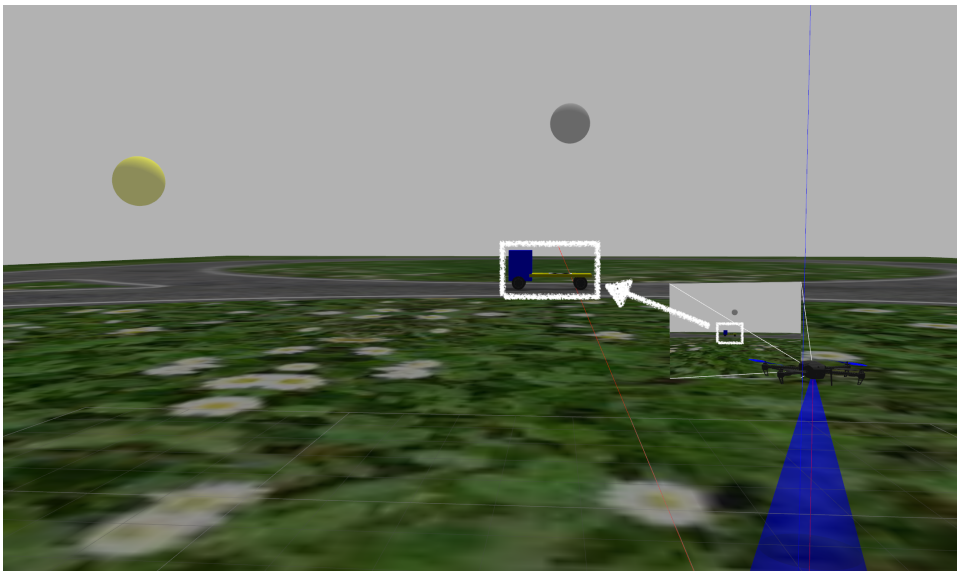

Figure 3.1: Target detection. Your goal is to transform the detection from the drone's camera in pixel coordinates to local NED coodinates.

3.1 Target detector

In the following sections, we will introduce the theory to perform the transformations between different reference frames, as well as some reference frame conventions used by PX4. Once we have derived all necessary frame transformations, we will implement them in PX4.

3.1.1 Theory

The first step in the transformation from image coordinates to the local frame is the transformation from image coordinates to the frame. We assume that our camera is a pinhole camera and use the model described below.

Transforming from image to camera frame

The main concepts of the pinhole camera model are illustrated in Fig. 3.2. The pinhole camera is an idealized model since it assumes that all light rays of the optical system intersect at a single point and it does not take into account the effects of lens distortions or blurring of unfocused objects. However, these effects can be compensated such that the model provides an accurate description of the relationships between a 3D scene and its 2D projection onto an image plane.

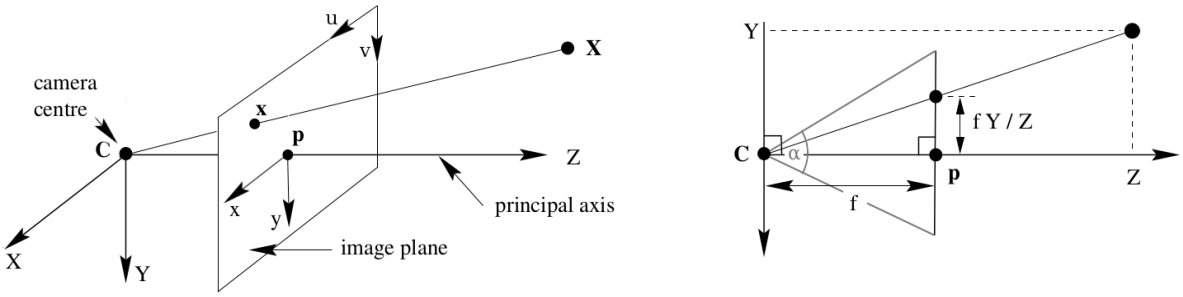

Figure 3.2: The pinhole camera model. Left: perspective projection at an angle. Right: orthographic projection from the side. Notice that the image plane is located in front of the camera projection center **C**. This model avoids the inversion of the image which is located behind the center in a regular camera.

The pinhole camera model assumes the rays from objects will all converge in the camera projection center **C**. The image coordinates u and v originate from the top left corner of the image and can be interpreted as pixel coordinates for digital images. Similarly, the centered image coordinates x and y are located in the center of the image plane at point **p**, i.e. at the intersection with the principal axis.

Question 3:
Using the image width w_{im} and height h_{im}, express the centered image coordinates x and y in terms of the image coordinates u and v.

The focal length f corresponds to the distance from the camera projection center **C** to the intersection with image plane. In a physical camera, the focal length corresponds to the distance between the lens and the actual image sensor. The field of view α is defined as the angle between the outermost light rays that reach the sensor. Fig. 3.2 visualizes the vertical field of view α for the pinhole camera model.

Question 4:
Using the orthographic projection in Fig. 3.2, find the focal length f as a function of the camera's vertical field of view α and the image height h_{im}.

The detection unit does not only provide the image coordinates u and v but also the distance d from the camera projection center **C** to the target.

Question 5:
Find the 3D camera coordinates X, Y and Z for an object at the 2D image coordinates x and y using the distance d.

Now you should have the equations to calculate the 3D position of an object from the 2D image coordinates u, v and the distance d given the image width w_{im}, height h_{im} and the field of view α.

Transforming from camera to gimbal frame

The PX4 autopilot uses the convention of the north-east-down (NED) coordinate system which differs from the camera frame as shown in Fig. 3.3. To comply with this convention, we need to transform our target location from the camera frame into the gimbal frame which is expressed as NED coordinates of the forward looking camera.

(a) Camera frame (b) Gimbal frame

Figure 3.3: Camera and gimbal reference frames. The standard camera frame convention is used extensively in the computer vision literature whereas the NED convention originates from aviation.

Any orientation of an object in 3D space can be described by three rotations known as Euler angles. In aeronautics, we use the roll-pitch-yaw (RPY) notation, i.e. a roll angle Φ (around X), pitch angle Θ (around Y) and yaw angle Ψ (around Z).
By Tait-Bryan Euler angle convention, we first rotate our object around its Z axis by Ψ, then around its Y axis (which has already been rotated) by Θ and then around its X axis (which has already been rotated twice) by Φ, as shown in Fig. 3.4. Note that the rotation order is very important since it affects the object's final orientation.

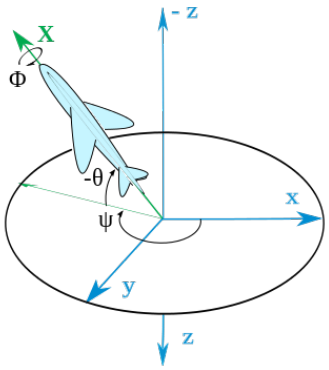

Figure 3.4: Tait-Bryan Euler angles. Tait-Bryan angles are expressed by three rotations around its principal axis following the or Z-Y-X convention. Rotations are applied in the order yaw, pitch, and roll according to the right-hand rule. Note that we are dealing with intrinsic rotations, meaning that every rotation changes the orientation of the coordinate system. This is often denoted by adding single quotes to the rotation axes as in Z-Y'-X".

Question 6:
Construct a rotation matrix $\mathbf{R}_C^G$ that converts a vector expressed in the camera frame to a vector expressed in the gimbal frame, as shown in Fig. 3.3. Alternatively, you may use Tait-Bryan Euler angles to perform the change of axis.

Transforming from gimbal to body frame

The onboard camera is mounted on a two-axis gimbal system that allows it to rotate and tilt freely. For simplicity, you can assume that the camera is located at the center of the drone's body despite the gimbal. The gimbal can rotate the camera around its Z axis (yaw) and then rotate around its Y axis (pitch). These rotation axes coincide with the drone's body coordinate frame.
Since we know the camera's pitch and yaw angle, we can transform the target location from the gimbal frame to the drone's body frame. The origin of this coordinate system is the center of the drone, with its X axis pointing forward (north), its Y axis pointing toward the right (east), and its Z axis pointing down.

Question 7:
Which set of roll, pitch and yaw angles would allow you to transform from the gimbal frame to the drone's body frame?

Transforming from body to local frame

The final transformation takes the target position from the drone-centric body frame to the fixed local frame. The target position in the local frame has its origin at the drone's takeoff point and the same orientation as the drone's body frame.

Question 8:
Which set of roll, pitch and yaw angles would allow you to transform from the drone's body frame to the local frame?

Uncertainty of detection measurements

Unfortunately, the target detection algorithm that was provided to us is far from perfect and the measurements it provides are noisy. More specifically, the simulated target detection camera has zero-mean Gaussian noise added on the pixel coordinates u and v as well as on the distance d. Knowing the uncertainty of our measurements is important since we need to know how much we can trust them to reliably track our target later on. Luckily, we are given the variances of the noise σ_u^2, σ_v^2 and σ_d^2 which we can use as a measure of uncertainty.
To know the uncertainty of our estimated position in the local frame, we have to propagate the variances from the image frame to the local frame. The uncertainties in the local frame are especially important for the Kalman filter which we will implement in the next practical.
Here we will look at some basic rules of error propagation. For a random variable x and a random variable y, the variance of their product $f = xy$ can be expressed as

$$\sigma_f^2 = \sigma_x^2 y^2 + \sigma_y^2 x^2 + 2xy\sigma_{xy}, \tag{3.1}$$

where σ_{xy} is the covariance. In case that the variables are independent, we have $\sigma_{xy} = 0$.

Question 9:
Using the equation above, find the variance of x_{if}, y_{if} and z_{if}.

Note, the variance of the scale s can be found as

$$\sigma_s^2 = \frac{\sigma_d^2}{l^2} + \frac{d^2}{l^6}(u^2\sigma_u^2 + v^2\sigma_v^2), \tag{3.2}$$

where $l = \sqrt{u^2 + v^2 + f^2}$. For simplicity, you can assume that s, u and v are independent. Furthermore, the focal length f is an exact value (i.e., $\sigma_f = 0$).
For a vector $\vec{x} = \begin{pmatrix} x & y & z \end{pmatrix}^T$, we express its uncertainty with the covariance matrix:

$$\mathbf{\Sigma}_{\vec{x}} = \begin{pmatrix} \sigma_x^2 & \sigma_{xy}^2 & \sigma_{xz}^2 \\ \sigma_{xy}^2 & \sigma_y^2 & \sigma_{yz}^2 \\ \sigma_{xz}^2 & \sigma_{yz}^2 & \sigma_z^2 \end{pmatrix} \tag{3.3}$$

Question 10:
Assuming the variables to be independent, find the covariance matrix for the target position in the image frame.

If we multiply our vector with a matrix $\vec{y} = \mathbf{A}\vec{x}$, we find a new covariance matrix as

$$\mathbf{\Sigma}_{\vec{y}} = \mathbf{A}\mathbf{\Sigma}_{\vec{x}}\mathbf{A}^T, \tag{3.4}$$

assuming the values of $\mathbf{A}$ to be exact.
Using these formulas and assuming the estimated drone attitude to be exact, we are able to propagate the uncertainties from the image frame to the local frame.

3.1.2 Implementation

In the following section, we will implement the frame transformations we derived in the theoretical part.

For your implementation of the target detector, you will be working on the following files:

- `src/dronecourse/target_detection/TargetDetector.hpp`
- `src/dronecourse/target_detection/TargetDetector.cpp`

All places where you have to write your own code are marked with a `// TODO` comment.

Transforming from image to local frame

In this section, we will subsequently implement all necessary frame transformations to convert the detection location from the image to the local frame. To reiterate, the order of the transformations is: image frame, centered image frame, camera frame, gimbal frame, body frame, and finally local frame.
First, we need to subscribe to the detection uORB topic `target_position_image` which contains the target location in image coordinates, the distance to the target in meters, and the respective variances of the measurements. The uORB message also contains the pitch and yaw angle of the gimbal which correspond

to the orientation of the camera with respect to the drone. In the header file, create the handle for this topic as a member variable of the target detection class and initialize the handle in the constructor.
In the `update()` function which is called periodically, check if we have received a new target position message. If we have received a new measurement, copy the message's content into the local variable `target_pos`.
The first step is to calculate the camera's focal length using the image width and horizontal field of view. The corresponding variables are already defined in the header file but they need to be set in the class constructor.
The next step is to convert the image coordinates to centered image coordinates using the image width and height. Together with the scale, focal length, and centered image coordinates, we can project the 2D detection in image coordinates back into 3D camera coordinates.
We will express the subsequent rotations as rotation matrices, otherwise known as direction cosine matrices (DCM). The first rotation will take us from the camera frame to the gimbal frame which is expressed in NED coordinates. The second rotation takes us from the gimbal to the drone's body frame using the known pitch and yaw angles of the gimbal which we can extract from the uORB message. Note that the angles describe the rotation from the drone's body frame to the gimbal frame, while you want to do the opposite.

The following functionalities of the matrix class will be useful for your implementation:

- `matrix::Dcm<float> rotation_matrix(matrix::Euler<float>(roll, pitch, yaw));`
  Create a rotation matrix from a Tait-Bryan Euler angle. Note that the constructor of the rotation matrix sets the transformation from frame 2 to frame 1 where the rotation from frame 1 to frame 2 is described by a 3-2-1 intrinsic Tait-Bryan rotation sequence.

- `matrix::Dcm<float> inversed_rotation(other_matrix.transpose());`
  The transpose of a rotation matrix performs the inverse rotation.

Note that you may alternatively construct the elements of the rotation matrix directly based on the lecture slides.

In order to transform from the drone's body frame to the local frame we need to know the vehicle's attitude and position. Create subscriptions to the `vehicle_attitude` and `vehicle_local_position` topics and update the corresponding member variables which are already defined for you in the header. The attitude message contains a field `q` which describes the drone's orientation in 3D space as a quaternion.

Quaternions are a compact way of representing rotations or orientations in 3D space as a 4D vector. To convert a quaternion representation to a rotation matrix, you can use the following constructor:

- `matrix::Dcm<float> rotation_matrix(q.inversed());`

Note that the rotation matrix has to be initialized with an inversed quaternion.

Knowing the drone's attitude, we can calculate the total rotation from camera to the drone centered local frame. In order to complete the transformation to the local frame, we only need to account for the position offset of the drone from its takeoff location which represents the origin of the local coordinate frame.
The last step before we can publish the transformed coordinates of the target is to propagate the uncertainties from the image frame into the local frame. For this purpose, you should create a covariance matrix using the measurement variances from the target detection message.
Using the total rotation matrix and the drone's local position, you can convert the target position and the covariance matrix to the local frame. The code to populate the target position message and send it via uORB is already provided for you.

Starting the target detector

The target detector is implemented as a separate module on the PX4 autopilot. You can start the module by issuing the following command:

```
pxh> target_detection start
```

Instead of manually starting the application each time you run the simulation, you can add the command to the following init script:

```
host> cat posix-configs/SITL/init/lpe/dronecourse
```

Just add a line that you would type into the console to the script. You can also add comments using `#`. The init script can be used throughout the course to automate commands.
Sending the target position over uORB will not only allow other modules to receive it. A separate model sends this message over the mavlink protocol to the ground station where you can observe the target position. This helps you to verify and debug your code.

3.1.3 Debugging

The following sections contain useful pointers on how to debug your code in order to implement the frame transformations.

Controlling the drone

You can control the gimbal on which the camera is mounted over the console:

```
pxh> dronecourse gimbal <pitch> <yaw>
```

Furthermore you can control the drone using the position controller you have implemented before:

```
pxh> dronecourse pos <x> <y> <z>
```

Modifying camera noise

Note that the camera sends image coordinates with added noise. To reduce or eliminate this error for testing, open the file `Tools/sitl_gazebo/models/dronecourse/iris_dronecourse.sdf` and change the tags `noise_xy` and `noise_z` which represent the standard deviations of the noise.

Modifying the truck trajectory

To modify the truck's trajectory, open the file `Tools/sitl_gazebo/models/truck/truck.sdf`. You can set `<speed_min>`, `<speed_max>` to control the truck's speed. (Set both to 0 for stationary truck.)
`<speed_bias_rate_std>` controls how much the truck accelerates and decelerates. `<pos_bias_max>` controls how much the truck deviates from a perfect trajectory and `<pos_bias_rate_std>` controls how fast it deviates. Set this values low for a more regularly moving truck. If the tag `<rand_trajectory>` is set to 0, the truck will stay on the outer trajectory and not enter the central part. This can make debugging easier since the truck will more often be on straight street segments. The trucks starting position can be set in the file `Tools/sitl_gazebo/worlds` with the `<pose>` tag:

```
<uri>model://truck</uri>
<pose frame=''>0 -50 0 0 0 0</pose> <!-- x y z roll pitch yaw -->
</include>
```

Note that the simulator uses a different coordinate systems (rotated by 90 degrees).

3.2 Conclusion

By the end of this practical, you should be able to convert the detection of the truck from pixel to local coordinates.

TP 4: Target Tracking

In this practical, you will implement a target tracking algorithm to estimate the position and velocity of a moving target.

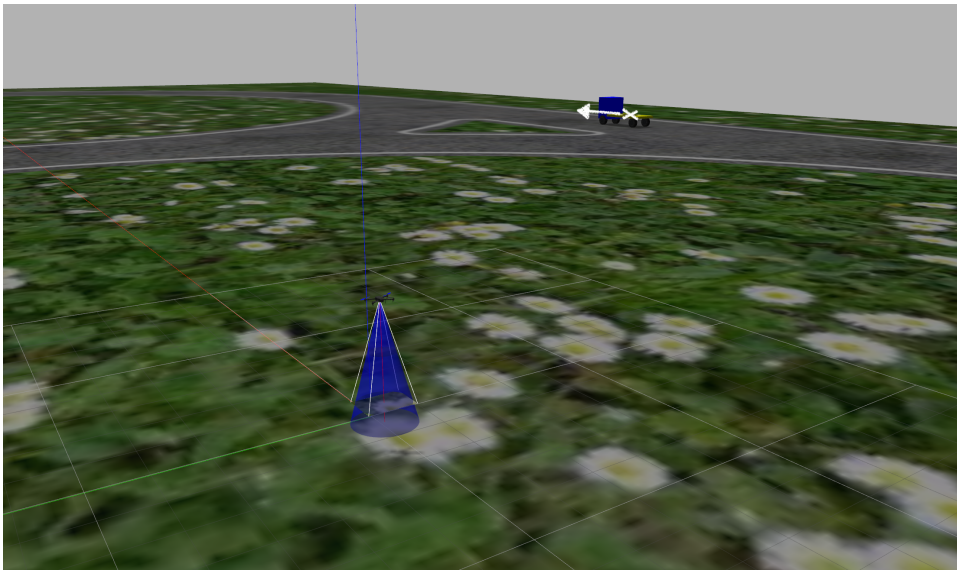

Figure 4.1: Target tracking. Your goal is to estimate the position and velocity of a moving truck using a Kalman filter that we will design in this practical.

4.1 Theory

We will use a Kalman filter with the state containing not only the target position, but also its velocity. First of all, this allows us to low pass filter the target position as well as to estimate its velocity. The Kalman filter features a prediction step which allows us to predict position and velocity between measurements. Since we only receive updates with 2Hz, this allows us to better approach the target. Another feature of the Kalman filter is that it will give us the uncertainty of our estimation and prediction in form of the covariance matrix. This is useful to decide whether (and maybe also how fast) we want to move towards the target. If the uncertainty is too high, it might no be wise to fly towards the target since the estimated position might be incorrect.
Here we go over the most important elements of the Kalman filter, but you should already know the principles. The state of our filter consists of the position and the velocity of the truck in the local frame

$$\mathrm{x} = \begin{pmatrix} px \\ py \\ pz \\ vx \\ vy \\ vz \end{pmatrix} \tag{4.1}$$

The dynamic equation relates the state to the derivative of the state.

$$\dot{\mathrm{x}} = \mathbf{F}\mathrm{x} + \mathbf{G}\mathrm{u}, \tag{4.2}$$

where $\mathbf{F}$ is the dynamic matrix (6x6) and $\mathbf{G}$ is the shaping matrix of the white noise $\mathbf{u}$.
The measurement model is described by the following equation:

$$\mathrm{z} = \begin{pmatrix} x_m \\ y_m \\ z_m \end{pmatrix} = \mathbf{H}\mathrm{x} + \mathrm{v}, \tag{4.3}$$

where z is the current measurement, $\mathbf{H}$ is the measurement (or design) matrix (6x3) and v is the measurement noise.

Question 11:
Define the dynamic matrix $\mathbf{F}$ and the measurement matrix $\mathbf{H}$ for our implementation assuming that the truck is moving with constant velocity.

Prediction

The prediction step which is performed periodically can than be expressed as

$$\tilde{\mathrm{x}}_{k+1} = \mathbf{\Phi}\hat{\mathrm{x}}_k, \tag{4.4}$$

where $\tilde{\mathrm{x}}_k$ is the new *predicted* state and $\hat{\mathrm{x}}_{k+1}$ is the old *estimated* state, and $\mathbf{\Phi}$ is the state transition matrix (6x6) (We discuss later how to calculate it).
It is not only important to have a prediction of the state, but also to estimate its certainty, which is given by the predicted covariance matrix (6x6) $\tilde{\mathbf{P}}_k$. The predicted covariance matrix is updated with every state update as

$$\tilde{\mathbf{P}}_{k+1} = \mathbf{\Phi}\mathbf{P}_k\mathbf{\Phi}^T + \mathbf{Q}, \tag{4.5}$$

where $\mathbf{P}_k$ is the old state covariance matrix and $\mathbf{Q}$ is the is the covariance matrix of the system noise (6x6) (We discuss later how to calculate it).
If we do not have a measurement during this iteration, we can set the estimated state and its covariance to our prediction:

$$\hat{\mathrm{x}}_{k+1} = \tilde{\mathrm{x}}_{k+1} \tag{4.6}$$

and

$$\mathbf{P}_{k+1} = \tilde{\mathbf{P}}_{k+1} \tag{4.7}$$

This completes the periodically executed prediction step of the Kalman filter.
The system noise covariance matrix $\mathbf{Q}$ can be found using two auxiliary matrices $\mathbf{A}$ and $\mathbf{B}$ as follows:

$$\mathbf{A} = \begin{pmatrix} -\mathbf{F} & \mathbf{G}\mathbf{G}^T \\ \mathbf{0} & \mathbf{F}^T \end{pmatrix} \Delta t, \tag{4.8}$$

where $\mathbf{G}$ is matrix containing the standard deviation of the model σ_a as diagonal elements:

$$\mathbf{G} = \begin{pmatrix} \sigma_{px} & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_{py} & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_{pz} & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_{vx} & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_{vy} & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_{vz} \end{pmatrix}, \tag{4.9}$$

which you will determine experimentally. Note that we set the position noise σ_{px}, σ_{py}, and σ_{pz} to zero since the uncertainty in the velocity (which our model assumes to be constant) will propagate to the position. Using $\mathbf{A}$, you can calculate $\mathbf{B}$ using the matrix exponential

$$\mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix} = \mathrm{e}^{\mathbf{A}}. \tag{4.10}$$

This allows you to calculate the state transition matrix as

$$\mathbf{\Phi} = (\mathbf{B}_{22})^T \tag{4.11}$$

and the system noise covariance matrix as

$$\mathbf{Q} = \mathbf{\Phi}\mathbf{B}_{12} \tag{4.12}$$

Correction (or Update)

Each time we receive a measurement, we want to correct our state prediction as well as the predicted of the state covariance. Note that what was before denoted with index $k+1$ is now denoted with index k. For the correction, we first calculate the Kalman gain $\mathbf{K}_k$ which tells us how much we trust in our measurement:

$$\mathbf{K}_k = \tilde{\mathbf{P}}_k \mathbf{H}^T (\mathbf{H}\tilde{\mathbf{P}}_k \mathbf{H}^T + \mathbf{R}_k)^{-1}, \tag{4.13}$$

where $\mathbf{R_k}$ is the covariance matrix of our measurement (3x3) that we calculated in the target detection chapter.
Now that we have our gain, we can update our state estimation

$$\hat{\mathrm{x}}_k = \tilde{\mathrm{x}}_k + \mathbf{K}_k(\mathrm{z}_k - \mathbf{H}\tilde{\mathrm{x}}_k) \tag{4.14}$$

At last, we have to update the estimation of the uncertainty by updating the estimated covariance matrix

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\tilde{\mathbf{P}}_k \tag{4.15}$$

4.2 Implementation

For your implementation of the target tracker, you will be working on the following file:

- `src/dronecourse/target_tracking/Kalman.hpp`
- `src/dronecourse/target_tracking/TargetTracker.hpp`
- `src/dronecourse/target_tracking/TargetTracker.cpp`

All places where you have to write your own code are marked with a `TODO` comment.

The `class TargetTracker` is the main class of the application and has an object of `class KalmanFilter` as member. The function `void update()` will be called periodically. It will first execute a prediction of the Kalman filter. Next, it checks for new measurements (`target_position_ned` uORB messages) from the TargetDetector. If there is a new update it will perform a correction (update) of the Kalman filter.
To keep our Kalman filter implementation flexible in terms of number of state variables and measurement variables, we will use C++ templates. You do not need to fully understand templates for this course. The template parameters of the Kalman class are `M`, the number of state variables and `N`, the number of measurement variables. You can create a Kalman object by providing it with a value for M and N, e.g.,

```
KalmanFilter<6,3> my_kalman;
```

which will create a kalman filter with 6 state variables and 3 measurement variables. Note that for templated classes, all functions have to be written in the `.hpp` file, hence the file `Kalman.cpp` does not exist. Within the `class KalmanFilter`, you can use `M` and `N` like constant integer variables.
To implement our Kalman filter, you can use the `class matrix::Matrix`. It is also a templated class taking as template parameters the type (i.e., `int`, `float` or `double`), the number of rows and the number of columns.

Here are some useful functions that will help you with your implementation of the Kalman filter:

- `matrix::Matrix<float, 6, 3> m; // Create a new matrix with 6 rows and 3 columns.`
- `m(1, 2) = 4.0f; // Set element at row 1 and column 2 to 4.0.`
- `m.setIdentity(); // Set the matrix to the identity matrix.`
- `m.setZero(); // Set all matrix elements to 0.`
- `m.setOne(); // Set all matrix elements to 1.`
- `matrix::Matrix<float, 3, 6> m_t = m.transpose(); // Transpose a matrix.`
- `matrix::SquareMatrix<float, 6> mmt; // Create a 6x6 square matrix.`
- `mmt = m * m_t; // Matrix multiplication.`
- `matrix::Matrix<float, 6, 6> mmt_inv = matrix::inv(mmt); // Matrix inversion.`
- `matrix::Vector<float, 6> diag = mmt.diag(); // Matrix diagonal extraction.`
- `mmt.print(); // Print a matrix to the console.`
- `B = matrix::expm(A); // Compute the matrix exponential`

4.2.1 Kalman Filter implementation

Initialization of the Kalman Filter

As a first step, you will implement the initialization of the Kalman filter in `init()`. Initialize the member variables `_h`, `_h_t`, `_dt`, `_f` and `_x`. (To know their meaning, look at the comments next to their definition.). Then fill the covariance matrix `_p` of the state with the initial values. The diagonal elements correspond to the variance (standard deviation squared!) of the initial state.

Next, we implement the function `void set_system_noise(const matrix::Vector<float, M> &w)`. This function should set the state transition matrix `_phi` and the covariance matrix of the system noise `_q`. (See in the theoretical part how to do it.) Now our Kalman filter is initialized.

Prediction step

In our implementation we do not distinguish between the predicted state $\tilde{\mathrm{x}}_k$ and $\hat{\mathrm{x}}_k$. Both are represented by the member variable `_x`. Similarly, we do not distinguish between the covariance of the prediction $\tilde{\mathbf{P}}_k$ and the covariance of the estimation $\mathbf{P}_k$ which are both stored in the member variable `_p`.
Implement the function `predict()` which updates `_x` and `_p` using the prediction formula from theory.

Correction step

We now implement the correction step in the function

```
void correct(const matrix::Vector<float,N>& z,
             const matrix::Vector<float,N>& v)
```

which has the parameters `z` which is the measurement and `r` which is the covariance matrix $\mathbf{R}_k$ of the measurement. First, calculate the Kalman gain $\mathbf{K}_k$ and store it in a local variable. Then you can update our state estimation $\hat{\mathrm{x}}_k$ and store it in `_x`. At last but not least, update the estimation covariance matrix $\mathbf{P}_k$ and store it in `_p`. This concludes the update step.

Return state estimation and variance

The two functions for you to implement are

```
matrix::Vector<float,M> get_state_variances() const
const matrix::Vector<float,M>& get_state_estimate() const
```

which return the state estimate and its variance. We return the variance of the state variables which are the diagonal elements of covariance matrix rather than the entire covariance matrix.

4.2.2 Target tracker implementation

Now we will implement the `class TargetTracker` which controls the Kalman filter. First, create a member variable of type `class KalmanFilter` in the definition of `class TargetTracker`. Use `M` and `N` as template parameters.

Setting up the onboard parameters

To be able to easily tune our filter, we will set up the system (or model) noise w as onboard parameters, which each element of the vector being a separate onboard parameter.

> You can create the onboard parameters in the following file:
>
> - `src/dronecourse/target_tracking/params.c`
>
> All places where you have to write your own code are marked with a `// TODO` comment. Note that maximum length for paramter names is 16 characters.

In the definition of `class TargetTracker`, create an array of parameter handles as a member variable. Having the handles as an array will make it easier to read out every values. In the constructor, initialize all elements of the handle array with the corresponding parameter name. After this, check if all the handles have been initialized correctly and print a warning in case of an invalid handle. You can use a for loop for to avoid copy pasting code. Next, create the member variable `_w` of type `matrix::Vector<...>` which will contain the values of the system noise.
Now we fill the function `void update_parameters()` in charge of updating the member variables for the system noise. Since changing the system noise implies recalculating the state transition matrix `_phi` and the covariance matrix of the system noise `_q`, we only want to update it when the values change. To do so, read the parameter values into a temporary variable and compare them to the previous values store in `_w`. If the values have changed, update `_w` and call the `void set_system_noise(const matrix::Vector<float, M> &w)` function of your Kalman filter. Note that you should never compare floating point variables using `==` since this values are not exact. Instead check if the difference between the values is larger than the resolution of your type:

```
float a;
float b;
if (fabsf(a - b) > FLT_EPSILON) {
  /* a and b are different, do something */
}
```

Setting up the Kalman Filter

First, we create the variables `M` and `N` containing the number of state variables and the number of measurement variables. Since these values have to be defined at compile time, we define them as `static const` and set their values directly:

```
static const int MY_INT = 5;
```

Next, we create the arguments used for `void init(...)` as local variables in the constructor, except `_w` which is a member variable and `dt` which is an argument of the constructor. Then call the `void init(...)` function of your Kalman member variable to initialize the filter with the arguments.

Update function

First, call `void update_parameters()` to update the system noise. Next, perform a prediction of the Kalman filter.
We now have to subscribe to the `target_position_ned` topic to receive target position estimates from our `TargetDetector`. Create the handle in the class definition and subscribe in the constructor. In the update function check if we have received a new message and copy the message into a local variable if it is the case. Create a vector containing the estimated position and the covariance matrix from the message. Using this vector and the covariance matrix, perform a correction of the Kalman filter. This completes our filtering. We perform a prediction at every iteration and perform a correction as we receive new data.

Publishing the filtered estimate

The last step is to publish the filtered position and velocity estimate as a uORB message on the topic `target_position_ned_filtered`. To do so, create a handle for the publication and initialize it to `nullptr` in the constructor. To keep the code easy to read, we will publish our new estimate in the function

`publish_filtered_target_position(...)`. Fill this function where you create a local message variable, fill it, and publish it. Note: the topic `target_position_ned_filtered` has the same message definition as the topic `target_position_ned`. Hence, the type `struct target_position_ned_filtered_s` does not exist. Use the type `struct target_position_ned_s` instead. You can set the field `target_id` to 0.
Call your function at the end of the update function to publish your new estimate after each update. This completes the implementation of the TargetTracker.

4.3 Debugging and parameter tuning

Once you manage to compile your code, you have to tune your filter by changing the standard deviations of the system and measurement noise. You have to start the target detection and the target tracking via the console:

```
pxh> target_detection start
```

```
phx> target_tracking start
```

The standard deviation of the system noise has to be determined experimentally. Note that the system noise greatly influences the variances of your estimation. Keep in mind that you will probably use these variances as a criterion whether you will fly towards the estimated position of the truck.

TP 5: Target Following

In this practical, we will first implement an automatic control of the gimbal system onto which the camera is mounted. This allow the camera to always point towards our target in order not to lose its sight once the drone starts moving towards it.

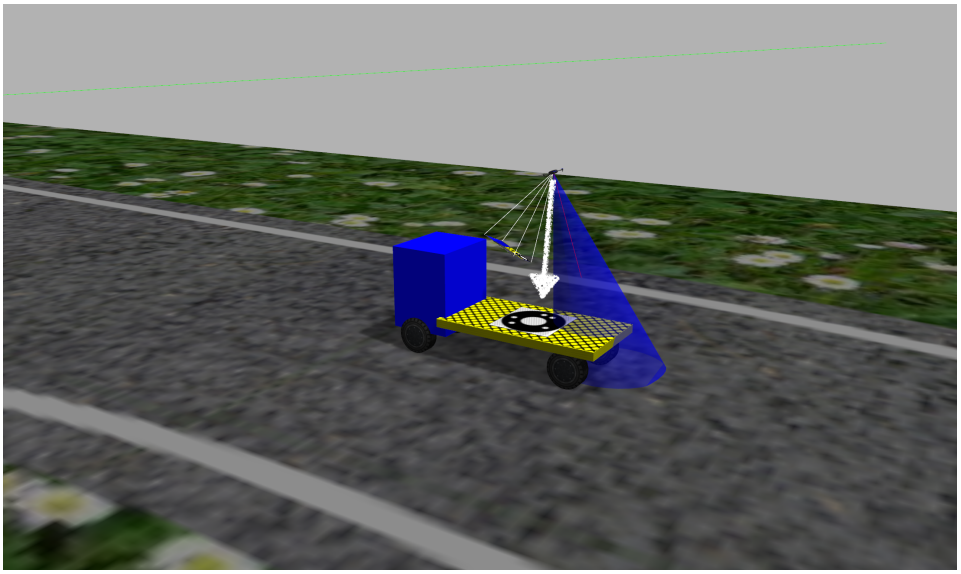

Figure 5.1: Target following. Your goal is to follow the truck and ultimately land on the cargo bed.

5.1 Automatic gimbal control

The gimbal system on the drone has two axes: the pitch and the yaw axis. Sending a pitch and yaw command to the gimbal will cause the gimbal to rotate around its yaw axis followed by the pitch axis. Finding these two angles is not trivial. Since the yaw rotation is around the drone's Z axis (down), this axis will remain stationary with the yaw rotation. This means that we find the pitch angle directly from the angle between the drone's Z axis and the target direction (i.e., the vector from the drone to the target). The pitch can be expressed as

$$\theta = \frac{\pi}{2} - \angle(\vec{Z}, \vec{T}), \tag{5.1}$$

where $\vec{Z}$ is the drone's Z axis and $\vec{T}$ is the target direction.
To calculate the yaw angle, we introduce an auxiliary plane that is formed by the drone's Z axis and the target direction. We are now looking for the yaw angle that rotates the drone's X axis (forward) into this plane. This is equal to the angle that aligns the drone's Y axis (east) with the normal vector of the auxiliary plane. So calculate the normal vector $\vec{n}$ of our auxiliary plane from the $\vec{Z}$ and $\vec{T}$ and find the yaw angle as

$$\psi = \angle(\vec{Y}, \vec{n}). \tag{5.2}$$

5.1.1 Implementation

For your implementation of the gimbal controller, you will be working on the following files:

- `src/dronecourse/controllers/GimbalCtrl.hpp`
- `src/dronecourse/controllers/GimbalCtrl.cpp`

All places where you have to write your own code are marked with a `// TODO` comment.

The gimbal can be controlled manually where the pitch and yaw angle are set from outside. This can be done in code with `set_command(...)` or by the user via the console by typing

```
pxh> dronecourse gimbal <pitch> <yaw>
```

This part of the course has already been written for you. Here you will only implemented the automatic mode, where the gimbal automatically follows the target. Switching to automatic mode can be done in code by `setAutomatic()` and via the console with

```
pxh> dronecourse gimbal auto
```

Note that setting a manual command by code or via the console overwrites the automatic mode.
First, set up uORB handles and subscribe them for the topics `target_position_ned_filtered` which gives us the filtered target position estimation from our target tracking module, `vehicle_attitude` which contains the drone's current attitude, and `vehicle_local_position` which contains the drone's current position in local frame.
Since our update should only be executed when we are in automatic mode, we work inside the `if` statement of the `void update()` function. To make our code more compact, we will not store the values of our uORB subscriptions in member variables. We create local variables in the `void update()` function for the filtered target position estimation and the drone's attitude and local position. Note, since we need the values of this variables at every execution of `void update()`, can copy their values into the local variables without checking if they have been updated.
Next, find the target direction $\vec{T}$, i.e., the vector from the drone to the target in the local frame. Then, transform the drone's Y and Z axis to the local frame. The next step is to find the normal vector of the plane containing the target direction and the drone's Z axis using vector geometry. Now, we can calculate the desired pitch and yaw angle. At last but not least, publish the desired angles in a uORB message of type `gimbal_command`. This concludes the implementation of the gimbal's automatic mode.

5.2 Following controller

For your implementation of the following controller, you will be working on the following files:

- `src/dronecourse/controllers/TargetFollower.hpp`
- `src/dronecourse/controllers/TargetFollower.cpp`

All places where you have to write your own code are marked with a `// TODO` comment.

The following controller which controls the drone to follow the target core of this tutorial. You will have complete freedom of how to implement it. As always, the `update()` function will be called periodically. Like in the position controller, you will have to implement the function `is_goal_reached()`. Starts by making

sure that the drone follows the target at low altitude stably. A low altitude will allow a smooth landing without much correction, keeping the drone in a stable attitude. You can subscribe to uORB message and define your own onboard parameters which will help you to tune your controller. Here are some tips that might be useful.
First of all, use all information of our `target_position_ned_filtered` message. Only following the estimated position will lead to a constant oscillation around the target position. Using the velocity estimate will allow you to smoothly follow the target. Furthermore, keep in mind that a target position estimate is always sent, even if we have no idea where the target is. The variance of the state variables, also contained in `target_position_ned_filtered` will allow you to now how sure we are about the position and velocity estimates.
Since the position and velocity estimates are unusably bad if there has not been a correction for a while, you should implement a strategy to look for the target. (This might be as easy as gain altitude and wait for the target to pass by).
You could use a finite state machine that has states such as search and follow, or more states if you want to implement a more sophisticated strategy. When looking for the target, it might be interesting to control the gimbal in manual mode and switching to automatic mode once the target has been found.
The position and velocity estimates may become unusable at very low altitude because the target is too close to the camera. You could use the sonar to check that the truck is still under your drone for the last meters of the landing and abort landing if the truck is not there.
To start the following controller, you can use the command below.

```
pxh> dronecourse target_following
```

In case you want to run the entire integrated mission, i.e. waypoint navigation, sonar landing, and target following, you can run the mission command.

```
pxh> dronecourse mission
```

This will switch the state of the system to the subsequent task as soon as the current task is achieved.