Crash Course Paparazzi 2022
MAVLab TU Delft

Roland Meertens, Christophe de Wagter, Guido de Croon,
Tom van Dijk, Matteo Barbera

February 2022

Contents

Chapter 1

Installing and Running Paparazzi

This chapter will guide you through the installation procedure of Ubuntu and the Paparazzi autopilot. In the following chapters you will perform your first flight using Paparazzi, make the drone autonomous and implement your own algorithms.

1.1 Installing Ubuntu

For this course you are required to install Ubuntu. If you are already running Ubuntu 18.04, skip to the next section. Otherwise, follow the steps here to install Ubuntu in a dual-boot setup with your existing Windows installation. This allows you to keep using your existing Windows installation next to Ubuntu. While it is possible to install Ubuntu as a virtual machine, this frequently causes problems with networking, USB and GPU support. Therefore we do not support virtual machines during the course.

The dual-boot installation of Ubuntu requires you to format parts of your drive, therefore there is a small risk of losing important files. Make sure you have backed up your files before starting the installation!

Follow this tutorial to install Ubuntu next to an existing Windows installation: `https://itsfoss.com/install-ubuntu-1404-dual-boot-mode-windows-8-81-uefi/`. We recommend you install Ubuntu 18.04 Bionic Beaver `https://www.ubuntu.com/download/desktop`. The rest of the course manuals assume that you are running Ubuntu 18.04.

Some useful tips and common issues:

- It may be useful to look up which function key (or combination of keys) your laptop manufacturer uses to enter the BIOS during start-up
- Sometimes the system will not let you shrink a partition's volume, even though free space is available. This is caused by immovable files placed towards the end of the volume (see `https://en.wikipedia.org/wiki/File_system_fragmentation`). Windows or third-party defragmentation utilities can help fix this problem. **NOTE** that these tools could leave your system unable to boot, so a system backup is strongly advised!
- If the Secure Boot option is grayed out in the BIOS, set the supervisor password in the BIOS
- After installing Ubuntu, if your computer boots directly to Windows, check the UEFI settings in your BIOS (usually accessible by pressing F10 or another FN key right after powering on the

laptop) and change the boot order from "Windows Boot Manager" to "ubuntu" or "*name of your hard disk*"

- Some BIOS let you add a delay at start-up to give you time to select which OS you would like to boot from. This can usually be changed in the BIOS Setup.
- **I**f you cannot manage to boot from the USB with the Ubuntu installation image it may be necessary to switch from UEFI Native to UEFI Legacy (With CSM) in the BIOS setup. The USB should then become visible in the Boot Device Options at startup (usually F9 during power-on)

1.2 Installing Paparazzi

Once Ubuntu is running, the next step is to install Paparazzi. We have prepared a Github branch here that contains the Paparazzi autopilot and example files for the course. The rest of this manual will show you how to install Paparazzi.

Most steps require you to use Ubuntu's Command Line Interface (CLI), also known as the *terminal*. You can open a terminal window by pressing Ctrl+Alt+T. Any commands that you need to run are listed as follows:

```
command 1
command 2
```

You need to type or copy these commands *line-by-line* to the terminal, pressing enter after each line. In other words: do not copy *all* commands at once, this can sometimes produce unexpected results. Note that pasting into the terminal uses Ctrl+*Shift*+V, not Ctrl+V.

Use the following steps to download the course branch and install Paparazzi:

1. Install Paparazzi UAV by opening a terminal (Ctrl+Alt+T) and running the one-liner found here: `http://wiki.paparazziuav.org/wiki/Installation`. Copy and paste (Ctrl+Shift+V) the entire one-liner into the terminal to prevent typing errors, then press enter to run it. After running the one-liner, you should see the Paparazzi center. Close this and return to the terminal.
2. During the course we will use git to keep track of your code changes and to collaborate with your teammates. First, go to `https://github.com/` and set up an account if you do not have one already. Then, in the terminal, register your git credentials using

   ```
   git config --global user.name your_name
   git config --global user.email your_email@host.com
   ```

   where you should replace `your_name` and `your_email@host.com` with the username and email you have registered at github.com.
3. Navigate to the paparazzi folder and add the tudelft remote.

   ```
   cd ~/paparazzi
   git remote add tudelft https://github.com/tudelft/paparazzi
   git fetch tudelft mavlabCourse2022
   ```

4. Checkout the mavlabCourse2022 branch using:

   ```
   git checkout mavlabCourse2022
   ```

5. Initialize, sync and update Paparazzi's submodules using:

```
git submodule init
git submodule sync
git submodule update
```

6. Build Paparazzi by using:

```
make clean
make
```

7. Select the conf and control panel files that were prepared for the course by running:

```
python start.py
```

   and select as Conf: userconf/tudelft/course_conf.xml

   and as Controlpanel: userconf/tudelft/course_control_panel.xml.
   Click 'Set active' and close the dialog.

8. Next to Paparazzi you will need some additional tools. Install ffmpeg, vlc, cmake, jstest-gtk and java using:

```
sudo apt install ffmpeg vlc cmake jstest-gtk default-jre
```

9. Install the Gazebo simulator. Gazebo version 9 is recommended for Ubuntu 18.04, earlier versions of Ubuntu may require Gazebo version 8. Install Gazebo 9 using:

```
sudo apt install gazebo9 libgazebo9-dev
```

10. Install eclipse to easily navigate through the paparazzi source code.
    (a) Download the eclipse-installer from `https://www.eclipse.org/downloads/download.php?file=/oomph/epp/2018-12/R/eclipse-inst-linux64.tar.gz`
    (b) Navigate to the Downloads directory, extract the .tar.gz file and run eclipse-inst.
    (c) Select the C/C++ version
    (d) It is recommended to use the default installation directory
    (e) After eclipse is installed start eclipse
    (f) Navigate to "File - New - Makefile Project with Existing Code
    (g) Name the project 'paparazzi', select the paparazzi installation location (`~/paparazzi`) and keep the default options

11. Build OpenCV for the Bebop.
    (a) Navigate to paparazzi/sw/ext/opencv_bebop:

```
cd ~/paparazzi/sw/ext/opencv_bebop
```

    (b) Install the required OpenCV libraries

```
sudo apt install libjpeg-turbo8-dev libpng-dev libtiff-dev zlib1g-dev
    libdc1394-22-dev
```

    (c) Make openCV using:

```
make
```

1.3 Running your first simulation

To test your Paparazzi installation, you will perform a short test-flight in simulation. Navigate to the Paparazzi folder and launch the Paparazzi Center using

```
cd ~/paparazzi
./paparazzi
```

In the top left drop-down box labeled ‘A/C’, select the ‘bebop_orange_avoid’ aircraft. In the top middle under ‘Target’, select ‘nps’, the Paparazzi simulator. Click ‘Clean’, then ‘Build’ to compile the example code. Under ‘Session’, select ‘Simulation - Gazebo’ and click ‘Execute’. You should now see the Ground Control Station and the Gazebo simulator. Click ‘Stop/Remove All Processes’ in the Paparazzi Center to end the simulation.

Chapter 2

Manual Flight with Paparazzi

In this part you will learn how to start Paparazzi, compile and upload your autopilot and perform your first manual flight.

2.1 Safety rules

To keep the practical safe and to prevent damage to the materials, there are some safety rules. Last year we had no incidents and managed to break zero drones, let's keep it that way :). **Carefully read and follow the rules below!**

CyberZoo rules

- No access after 18:00.
- Persons inside the flying area:
  - Try to avoid having people inside the flight area when drones are flying.
  - If there are people inside the flight area, notify them when you start flying.
  - If you are inside the flight area while other drones are flying, we recommend you to wear safety glasses. These hang on the poles next to the entrances. Pay constant attention to the drones that are flying in the zoo, as unexpected errors can cause them to make abrupt movements.
- When leaving the CyberZoo:
  - Clear up the desks.
  - Remove all obstacles and other materials from the CyberZoo.
  - Shut down Motive on the Optitrack PC.
  - Turn off the lights and open the curtain on the left side of the CyberZoo (as seen from the desks).
- Do not borrow any materials from the desks behind the CyberZoo (around the press). Do not use the desks, power sockets or ethernet cables there either.

Handling the drone

- If you damage or break anything, report this to the TA's so we can fix or replace it.
- Demonstrate your flying skills – including nose-in flight – in simulation to one of the TA's before flying the real drone. You should be able the safely land the drone when it does not behave as expected. Your team should have at least one safety pilot.
- Prevent common problems:
  - The example flight plan contains safety rules ('exceptions') that land the drone in case of problems. Do not disable these.
  - GPS (Optitrack) loss. The drone needs position and velocity feedback for navigation, without this feedback the drone cannot maintain its position. This feedback is provided through Optitrack (more in this in Chapter 3). The Optitrack system can fail when you did not initialize your drone's rigid body correctly, when you forget to plug in the ethernet cable, when your drone gets confused with another team's drone, etc. Therefore:
    - Before taking off, **always** verify that the drone's position and heading in the Ground Control Station are correct.
    - During flight, be ready to take over if the GPS fix is lost.
    - The flight plan should automatically trigger a landing when the GPS fix is lost.
  - WiFi loss. The GCS communicates with the Bebop over WiFi. Without a WiFi connection, you are no longer able to control the drone. Therefore:
    - Do not remove the datalink loss exception from the flight plan. This exception will land the drone automatically when the connection is lost.
  - Connecting to the wrong drone. Before uploading your code, make sure that you are connected to the right drone. Other teams will not appreciate it if you upload new code to their drone while in flight – this will lead to a crash. Also make sure you are connected to the right drone before taking off, letting another team's drone take off instead can cause dangerous situations.
  - Joystick issues. Make sure the joystick is calibrated and working correctly before flying. Before takeoff, check in the GCS that the mode switch is working.
  - Bad code. Bad code can cause a variety of problems, from missing an obstacle and crashing into it, to segmentation faults that kill the autopilot altogether, often with the motors still running. Therefore:
    - Test your code in simulation before testing it on the real drone.
    - When possible, test your vision code on a dataset you collected beforehand or with an in-hand test on the real drone, before testing it in-flight.
    - If your drone crashed because of unexpected behavior, try to find out what went wrong before flying again. Do not keep testing the same code in the hopes that it will work the next time. Logging (part 5) will help you find out what went wrong.
- Regarding batteries:
  - LiPo batteries, as used in the Bebop, become permanently damaged when the battery voltage drops too low. This will result in a lower battery capacity and shorter flight time. The Bebop battery has a voltage of 12.4 V when full, and 11.1 V when empty. **You should land when the battery level drops to 11.1 V or below**. The Ground Control Station will give an audible warning when this happens ('BAT LOW'), make sure to turn your

sound on. If you hear 'BAT CRITICAL', you are too late. The drone also consumes power while it is stationary (e.g. during debugging), make sure to check the battery voltage from time to time.

If you suspect your battery is damaged (BAT LOW at takeoff, exceptionally short flight times or physical damage), contact one of the TA's.

– LiPo batteries are a fire hazard, **do not leave the batteries unattended while charging!**

2.2 Pre-flight preparation

Before you begin your preparation, charge your battery if you have not already done so. Charging can take a significant amount of time, so you should do this beforehand or during your preparation.

Start Paparazzi Center

Start Paparazzi: go to your terminal and navigate to the paparazzi directory, then run `paparazzi`:

```
cd ~/paparazzi
./paparazzi
```

The Paparazzi Center performs the following functions: it manages aircraft, airframes and flightplans, it is used to compile and upload your code and it is used to start flight- or simulation sessions.

The top left corner shows the current aircraft (A/C). Use the drop-down menu to select 'bebop_orange_avoid'. Below the A/C there are fields to select your airframe and flight plan and other options. Leave these as they are for now; later in the course you may want to modify your flight plan or airframe.

The buttons in the top center (Target, Flash mode, Clean, Build, Upload) are used to compile and upload your code.

Finally, the top right allows you to execute a 'Session': a collection of programs that help you perform your flight. Examples of these programs are the communications server and datalink, the Ground Control Station and your joystick.

Saving a custom session is useful in case you are, for example, using a different joystick, flying with a different flight plan than the default, or if you always want to launch an additional process like the Log Plotter or the Messages (found in Tools).

Connect and calibrate your joystick

Before you can fly, you need to connect and calibrate your joystick. Connect the joystick to your computer[1].

Joystick calibration should be repeated every time you connect a joystick to your computer, as the center positions may vary slightly per device and the trim sliders may have shifted. It is also a good test to verify that your joystick is working correctly.

SM600 joystick example

Ensure that the trim sliders to the right and below the two sticks are in the center position. Check that the bottom mode selector next to the name (SM600) is in the second position, G3-G4.5. Before plugging in the USB of the joystick make sure that all sticks are centered, including the throttle.

[1] On some systems, a bug in Ubuntu 18.04 can cause the screen to rotate when the joystick is used. If this happens to you, open a terminal and enter the following commands to (permanently) fix this: `xrandr -o normal`, followed by `gsettings set org.gnome.settings-daemon.peripherals.touchscreen orientation-lock true`

Open a terminal (Ctrl+Alt+T), type jstest-gtk and press enter to start the joystick calibration program. Find your joystick in the list. If your device is not `/dev/input/js0`, write down the number after 'js', you will need this later. Double-click on your joystick to start the test and calibration program.

Click the 'Calibration' button, then 'Start Calibration'. Move both sticks in circles through their extreme positions, then return them to the center position (the throttle should also be centered). Switch the autopilot mode switch on the top left between the two positions. Move the tuning knob on the top right fully clockwise and counterclockwise. With the control sticks centered, click on the 'Ok' and 'Close' buttons. It's important to check that the throttle (Axis 2) can reach the minimum value after calibration, as the drone will not arm otherwise.

Other joysticks

A similar process using jstest-gtk needs to be repeated if another joystick or radio is being used. However, by default the Paparazzi Center will load the presets for the SM600 joystick. Other radios may use a different number of input channels, or the mapping of the channels will be different, and therefore another preset must be selected.

In order to select another joystick preset, start a session in the Paparazzi Center. If you plan to fly in simulation, select Simulation: Gazebo + Joystick, or Flight UDP when flying with a real drone. The joystick textbox should now have appeared, visible amongst the other programs of the session at the top of the Paparazzi Center. Replace the XML filename at the end of the textbox with the filename of the joystick you are planning to use. A number of XMLs have already been created for a variety of joysticks, you can check the available ones in the `conf/joystick` directory. To avoid having to repeat this process every time the simulation is launched, click Session → Save at the top of the window. The newly save session will now appear in the dropdown menu next to the Execute button.

Creating joystick XML files

In case the joystick you are planning to use does not have a corresponding XML it may be necessary to create a new, custom joystick XML. There is a comprehensive tutorial explaining all there is to know on how to do this in the `doc/sphinx/source/tutorials/intermediate/create_joystick.rst`.

If you succeed in creating a new joystick XML for a device you think others may have at home or want to use, do not hesitate to create a pull request to the mavlabCourse2022 branch to share your work!

2.3 Flying in simulation

Before you fly with the real drone, you will first practice in the simulator. This allows you to practice without the risk of breaking your drone. It also allows you to practice somewhere else than the CyberZoo as long as you have access to a joystick.

The simulation is started as follows:

1. In the Paparazzi Center, go to the Target drop-down menu and select the 'nps' target. NPS is the *New Paparazzi Simulator*; by setting nps as the target your autopilot is compiled to run on your own pc instead of on the Bebop.
2. Click 'Clean', then 'Build'. Paparazzi will now compile your autopilot.
3. In the Session drop-down menu, select 'Simulation - Gazebo + Joystick' (*not* 'Simulation'). Press execute to start the simulation.

   If your joystick device was not `/dev/input/js0` the following error will appear in the Paparazzi center: `Invalid_argument("index out of bounds")`. Fix this as follows: the command in the

'Joystick' textbox ends with `-d 0`; replace the 0 with the number of your joystick device (e.g. if your joystick was `/dev/input/js1`, change the command to `... -d 1`). Click the 'Stop' button next to the Joystick textbox, then click 'Redo'. In the top menu of the Paparazzi Center, click 'Session → Save' and overwrite the 'Simulation - Gazebo + Joystick' session. Paparazzi will now remember your joystick number. You may have to repeat these steps for the 'Flight UDP' session.

Sometimes a 'Speech dispatcher' error may appear. As long as you can still hear the messages from the GCS, this error can be ignored.

After these steps, two new windows will have appeared: 'GCS' and 'Gazebo'. GCS is the *Ground Control Station*, from here you can control and monitor your drone. The top left of this screen shows the status of your drone. Pay extra attention to the following fields:

- Bat: this shows the battery level of your drone. Stop flying when the voltage drops to 11.1 or below, otherwise you will damage the battery. This is of course not a problem in the simulator.
- Link: indicates that the drone and GCS are connected. This should be green.
- Status:
  - Autopilot mode (ATT or NAV): this is the current mode of the autopilot. In ATT mode you are in control of the drone, in NAV mode the drone will follow the flight plan. Check that the mode switch on the top left of the joystick toggles the autopilot mode, then set it to ATT.
  - RC/Joystick status (OK): this indicates that your joystick is operating correctly.
  - GPS status (3D): this indicates that the drone has a position fix. This field should be green and at '3D' for autonomous flight.
- Throttle (red, 0%): this shows the current throttle level. The indicator is red when the motors are killed and orange/green when the motors are armed (running).

The right part of the GCS window shows a map of your drone and flight plan (top), and the flight plan itself (bottom). The autonomous flight exercise will show how to use the flight plan. The GCS map may not be centered on the drone when you launch it. Click the map and press F to recenter it

The Gazebo window provides a view of your simulation. Navigate around the CyberZoo by panning (click-and-drag the left mouse button), zooming (scroll) and rotating (click-and-drag the middle mouse button). You can reduce the clutter by going to the 'Layers' tab (top left) and unchecking the checkboxes. (Note: the CyberZoo and environment will remain visible to the drone's camera).

2.4 Flying in simulation

Now that you are familiar with the GCS and Gazebo, it is time for your first flight. Open the GCS and make sure that the autopilot is in ATT mode (use the top left switch of the joystick). Then, move the throttle/yaw stick (left) fully to the bottom-right position until the throttle indicator turns orange: you have now started the motors.

Should you be using a different joystick (a stick joystick for example), the arming procedure may be different. For stick joystick you must move the throttle to zero and twist the stick counter-clockwise. This way you will be sending the same signal to the drone (no throttle, max yaw to the left), which by default arms/disarms the motors.

Switch to the Gazebo window. Slowly advance the throttle until the drone takes off. You can now use the simulator to practice your manual flying skills. It is very important to test if you can fly a drone nose-in (with its nose pointing towards you). When you can do this perfectly you can consider yourself good enough to serve as a safety pilot during this course.

When you are finished, you can stop the simulation by going to the Paparazzi Center (not the GCS) and clicking the 'Stop/Remove All Processes' button.

2.5 Flying the real drone

Once you have mastered flying in the simulator, it is time to transition to the real drone. If this is your first flight, first demonstrate your flying skills in the simulator to one of the TA's. Then:

1. Place the Bebop inside the CyberZoo. Connect the battery and turn the drone on using the power button on the rear side. When the light stops blinking, press the power button four times. (This allows custom code to be uploaded).
2. Return to your computer and connect to your Bebop's WiFi access point. (Note: disable and re-enable the WiFi in Ubuntu if the Bebop takes too long to appear in the list of available networks.) Make sure that you have connected to the right Bebop, it is a good idea to write down its number somewhere.
3. In the Paparazzi Center, select the 'ap' target and click 'Clean' and 'Build'. Check that you are connected to the right Bebop, then click the 'Upload' button. You should get a message that the autopilot has been started successfully.
4. Select the 'Flight UDP' session and click 'Execute'.

After these steps, the GCS window should appear. Ensure that the battery level is sufficient, the Link indicator is green, the RC status indicator is 'OK' and that the autopilot is in ATT mode. Since you will fly manually, it is ok if the GPS fix indicator is red. If everything is in order, start the engines by moving the yaw/throttle stick to the bottom-right. Slowly increase the throttle to take off, then enter a stable hover. Congratulations, you are now flying with Paparazzi!

Chapter 3

Autonomous Flight with Paparazzi

In this exercise you will use the provided aircraft and flight plan to avoid orange obstacles inside the CyberZoo.

If necessary, repeat the pre-flight preparation steps from Chapter 2: charge your batteries and make sure your joystick is connected and calibrated.

3.1 Preface: autopilot overview

Before you start flying autonomously, it is good to have a basic understanding of what is going on inside the autopilot. Broadly speaking, the drone is controlled by two nested control loops: stabilization and guidance.

Stabilization – also called inner-loop control – controls the attitude of the drone. Since quadrotors are inherently unstable, the stabilization controller is typically always running, even during manual flight. During the course you will not interact with the stabilization controller unless you provide attitude setpoints.

Guidance – also called outer-loop control – controls the position, velocities and accelerations of the drone. The guidance controller sends attitude setpoints and throttle commands to the stabilization controller. Later in the course you can provide your own setpoints to the guidance controller to navigate the drone around obstacles.

Detailed information on the control loops can be found on the Paparazzi wiki.

In this example, the guidance setpoints are provided by the *flight plan*. The flight plan acts as a finite state machine and will typically only change its setpoints in response to certain events, such as the drone arriving at a target waypoint. In later parts of the course, you will send guidance setpoints directly from your own code instead of from the flight plan.

Apart from the control loops, the autopilot uses estimators (AHRS, INS) to keep track of the drone's state. The estimators will not be considered in this course.

3.2 Simulation

It is important to test your autonomous flight plan in simulation before testing it on your real drone. This allows you to find and fix mistakes in your code before they cause damage to your drone. The simulator also allows you to prepare and test your flight plan without coming to the CyberZoo, charging batteries, etc.

Before starting the simulator, make sure that you are no longer connected to your real drone. Compile the autopilot by selecting the 'nps' target and clicking 'Clean' and 'Build'. Select the 'Simulation - Gazebo + Joystick' session and click 'Execute'.

For this exercise, it is convenient to place the GCS and Gazebo windows side-by-side. The bottom-right of the GCS shows the flight plan. The flight plan consists of 'blocks' of instructions, such as 'Start Engine' and 'Takeoff'. You can look inside a block by clicking the small triangle next to it. The currently active block is highlighted in green.

Use the flight plan to start the autonomous flight:

1. In the GCS, ensure the autopilot is in NAV mode.
2. Double-click the 'Start Engine' block to start the engines. The throttle indicator should turn orange.
3. Double-click 'Takeoff'. The drone will now take off and hover at its current position.
4. Start the orange avoider by double-clicking the 'START' block. The drone will now move randomly through the CyberZoo and will yaw to avoid the orange poles.

If something goes wrong during autonomous flight, you may have to take over manually. This is an excellent opportunity to practice this! Make sure that your throttle is about halfway up, then set the autopilot mode to ATT. You are now in full control of the drone. Try to land the drone in a controlled manner. Select the 'Holding Point' block before switching the autopilot back to NAV mode.

When you are finished, close the simulation using the 'Stop/Remove All Processes' button.

3.3 Real drone

OptiTrack setup

Autonomous flight becomes significantly easier if the drone has an accurate estimate of its position and velocity. Inside the CyberZoo, an OptiTrack motion capture system is used to provide these measurements. Before your flight, you will have to add your drone in the motion tracking software. This is performed as follows:

1. Place the drone in the center of the CyberZoo with the front of the drone pointing towards the right as seen from the desks. Turn on the Bebop and press the on button four times when the light stops flashing.
2. On the PC labeled 'OPTITRACK PRIME17W PC':
   (a) Start Motive if it is not already running.
   (b) In the top menu, click 'Layout → Capture'.
   (c) In the perspective view, select your drone's markers by dragging a box around them. Right click and choose 'Rigid body → Create from selected markers'.
   (d) Under properties (bottom left), read the value of your 'User ID'.
3. On your laptop:
   (a) Plug in the ethernet cable lying on the desk. (Note: you will no longer have access to eduroam)

(b) Upload the autopilot to the Bebop: in the Paparazzi Center, select the 'ap' target and click 'Clean', 'Build', then 'Upload'.

(c) In the Paparazzi Center, start the 'Flight UDP' session. Find the NatNet command textbox:

```
.../paparazzi/sw/ground_segment/misc/natnet2ivy  -ac 9999 42
```

This program gets your drone's position from Motive and transmits it to the Bebop. Replace the number 9999 with your User ID number from Motive. 'Stop' and 'Redo' the program. You should get a message that your rigid body is now being tracked.
You may want to save your session with this User ID now (Session → Save). Be careful: your User ID may be different the next time you are in the CyberZoo!

4. *Before *every* flight*: in the GCS, verify that:
   - You have a 3D fix (green indicator).
   - The position of your drone is correct.
   - The heading of the drone is correct.

Additionally, if you are using a new or modified flight plan then it might be a good idea to check the locations of the waypoints beforehand: ask a colleague to carry the drone to the waypoints in your flight plan and verify that they are inside the CyberZoo, not too close to the nets or walls and safely reachable for the drone.

Autonomous flight

At this point you have uploaded your autopilot to the Bebop and verified that it has a 3D fix. You are now ready to start your first autonomous flight. The steps are the same as during the simulation:

1. In the GCS, ensure the autopilot is in NAV mode.
2. Double-click the 'Start Engine' block to start the engines. The throttle indicator should turn orange and the drone's rotors should activate.
3. Double-click 'Takeoff'. The drone will now take off and hover at its current position.

You can start the orange avoider by double-clicking the 'START' block. Be ready to take over as the drone may not always see the obstacles as well as during the simulation! Also remember to watch the battery voltage as it might drop very fast.

You can use VLC to view the drone's camera feed. Start VLC, then click 'Media → Open File...' and open `paparazzi/sw/tools/rtp_viewer/rtp\_5000.sdp`. The Bebop's camera is rotated by 90 degrees. You can correct this in VLC by going to 'Tools → Effects and filters → Video Effects → Geometry', checking 'Transform' and selecting 'Rotate by 270 degrees' below it.

When you are finished, double-click on the 'Land here' block in the GCS to land your drone.

3.4 Next steps

Edit your flightplan

Now the power of Paparazzi is in your hands: you can now create a totally autonomous drone with the power of a flightplan. Read this page to understand what you can do: `http://wiki.paparazziuav.org/wiki/Flight_Plans`. Make sure you understand:

- What a waypoint is, and what a sector is.
- What exception, and while can do.

- What the vertical controls are that you can use (alt, climb, throttle).
- What the navigation modes are that you can use (attitude, heading, go, path).
- How you can set a variable and call a function.

Try to create a simple flightplan that performs something of your choice.

Create a safety rule

Last week we discussed several problems that can make your drone crash, such as an empty battery, losing GPS or coming too close to the wall of the arena. When flying your drone you want your drone to do something as soon as these dangerous situations occur:

- If your battery is empty, you want to perform a normal landing
- If your GPS is lost, you want to perform a normal landing
- If you fly to the wall of the arena you want to stay at the last safe point you found.

The Paparazzi flight plan allows you to create exceptions: when the check of that exception becomes true the drone will execute a certain block. Look at the flight plan file

$\boxed{\text{flight\_plans/tudelft/course\_orangeavoid\_cyberzoo.xml}}$ to see how these exceptions are implemented.

Chapter 4

Developing your own Module with Paparazzi

This final part is set up a bit differently than the previous ones: instead of telling you step-by-step what to do, this document will give you an overview of the concepts involved in writing your own module using the colored object detector and orange avoider as examples. You do not need to read this entire document (although you are recommended to do so), use the table of contents to quickly find the information you need.

The simplest way to start writing your own code is by modifying the example modules: the colored object detector and orange avoider. The rest of this document will often refer to these modules as examples and describe *why* they are written the way they are. Once you understand how these example modules work, you can create new or additional modules if you need them.

Throughout this chapter there will also be guided practical exercise where you will modify a simplified copy of the `orange_avoid` module. The module name is `mav_course_exercise`, and all relevant files will have the same name as is the convention in Paparazzi, which will be explained in this chapter. The code for this exercise can thus be found in the folder `paparazzi/sw/modules/mav_course_exercise`.

4.1 Paparazzi project structure

Before diving in, it is important to be familiar with the structure of the Paparazzi project, such that you may be able to quickly locate the files that you are looking for. Paparazzi is a complex software, which must manage everything that a drone needs to perform in order to fly, from the high level tasks like image processing, to the low level task of sending electrical signals to the physical motors. Since it can be flashed to all types of MAVs, Paparazzi needs to be quite modular in order to account for different drone configurations, from flying wings to unconventional rotorcrafts. All the source code for all that tasks related to flying are located in the `paparazzi/sw/airborne` folder (sw: software). Most of the code you will be developing during the course will be placed in this folder; you will adapt or develop your own module (`paparazzi/sw/airborne/modules/`).

The source code on its own, however, is not enough. Paparazzi would not know what to make of all the files, what to compile and link, or what to include on the drone you would like to fly. To achieve this, the configuration of modules, airframes,control panels, etc. can be found in the conf folder. Most of the files found here are XMLs that are used by Paparazzi to auto generate the code that will be uploaded to the drone. The folders relevant to this course are the `paparazzi/conf/airframes`, `paparazzi/conf/modules`, and `paparazzi/conf/userconf`. The airframes are files that specify which modules you want to include during compilation, settings for the physical actuators of the drone, motor mixing laws, and more. The modules folder contains all the source code for all modules, and the userconf folder contains the settings for the default PaparazziCenter panel, as well as the airframes that will appear in the dropdown menu of the of the PaparazziCenter.

4.2 Module code overview

The example aircraft use the cv_colorfilter and orange_avoider modules to avoid obstacles. In this part, we will take a closer look at how these modules are written and use this as an example for your own module.

4.2.1 Module files

Modules consist of the following files: a module .xml file, source and header files. The .xml file contains a description of the module that tells Paparazzi which files to include during compilation. These xml files can be found in `paparazzi/conf/modules/`. Take for instance the orange_avoider.xml:

```xml
<module name="orange_avoider">
  <doc>
    ...
  </doc>
  <depends>...</depends>
  <header>
    <file name="orange_avoider.h"/>
  </header>
  <init fun="orange_avoider_init()"/>
  <periodic fun="orange_avoider_periodic()" freq="4"/>
  <makefile >
    <file name="orange_avoider.c"/>
  </makefile>
</module>
```

The xml file starts with a 'module' element that sets the name of the module (orange_avoider). Optionally, this element can contain a 'dir' attribute as well, to specify the location of the source files relative to `sw/airborne/modules/`. In this case the directory is not provided and the module name is used (i.e. `sw/airborne/modules/orange_avoider/`).

After a documentation and dependency section, the xml contains a 'header' element in which the header files of the module are listed. Typically, you will only see one header file here that provides an easy-to-use access point for other modules.

The header element is followed by an 'init' and 'periodic' element. These specify what functions in your module code should be called by the autopilot, and in case of the periodic function it also specifies its frequency in Hz. Section 4.2.2 will describe these functions in more detail.

At the end of the xml file is the 'makefile' element. This section describes how your source files should be compiled. Simple modules such as the orange avoider only list one or more source files. More complicated modules such as `cv_opencvdemo` can specify additional compiler flags (to link OpenCV, for example) and can have different makefile sections depending on whether the autopilot is compiled for use on the drone (`target="ap"`) or in simulation (`target="nps"`).

More information on module .xml files can be found here.

The source and header files of your module can be found in `sw/airborne/modules/<your module dir>/`. In the next sections we take a closer look at the content of these files.

When creating your own module, you can use the `create_module` script in the `paparazzi` folder to create stubs for your module xml, the source and header files. Alternatively, you can copy paste an already existing module and manually create the module xml.

MAV Exercise: Setting up a module XML

A module XML for the `mav_course_exercise` module has already been created for you, but is incomplete. Navigate to the file, which can be found together with all other module XMLs in `paparazzi/conf/modules/`, and complete the code using the information from Section 4.2.1. If you are unsure about what needs to be added to the file, take a look at `orange_avoid.xml` and use it as a baseline.

To test whether the module was set up correctly, there are still two steps that need to be taken. First, the module must be added to an airframe, so that it will be compiled and set up to run in parallel to all other modules. An airframe has already been set up for this purpose, called `bebop_mav_course_exercise.xml`, found, together with all other airframes used in this course, in the folder `paparazzi/conf/airframes/tudelft/`. Simply add the following line:

```
<module name="mav_course_exercise"/>
```

somewhere next to where the other modules are declared to ensure that it will be included during compilation. By default, a new airframe will not automatically appear in the 'A/C' dropdown menu in the Paparazzi Center. So the second step that must be taken is to add this new airframe to your desired `conf.xml`. As explained in Section 1.2, the conf used for this course is `userconf/tudelft/course_conf.xml`. Open this file, and add a new aircraft by copy-pasting the `bebop_orange_avoid` entry (the `course_orangeavoid_cyberzoo_guided.xml` flight plan will not work with the `mav_course` module by default) and replacing the name, ac_id, and airframe field. If everything worked as expected, the next time Paparazzi is launched you should be able to compile the new aircraft!

4.2.2 Module functions

The `orange_avoider.c` file has roughly the following structure:

```
#include ...

#define ...

void orange_avoider_init() {
    ...
}

void orange_avoider_periodic() {
    ...
}

...
```

Note that the module does not have a 'main' function. This is because the module is not a stand-alone program. Instead, the autopilot will regularly call other functions that are part of your module, such as the `orange_avoider_init` and `orange_avoider_periodic` functions in this example. Which functions are called is defined by the module xml file described earlier.

The Paparazzi wiki lists the types of functions you can register in the module xml: `init`, `periodic`, `event` and `datalink`, of which init and periodic are the most relevant for this course.

The *init* function is called once at startup. You can use this function to initialize important variables of your module, or memory intensive structures such as large arrays, or for instance to subscribe to new video frames (Section 4.2.3).

Once the autopilot is fully initialized, it will enter an infinite loop[1] in which it will continuously read new sensor data, feed this to the guidance and stabilization controllers, and send new commands to the Bebop's motors[2]. From this loop, the autopilot can also call your module's *periodic* function at a frequency specified in the module xml. Within this function, you can for instance get the drone's state and use this to calculate new setpoints for the guidance controller. In the `orange_avoider_periodic` example, the module reads the amount of orange in the front camera image and uses this to set its next waypoint positions.

Because the periodic function is called from within the autopilot's control loop, you should take care that the function does not take too much time to run. The autopilot runs at 512 Hz, which means that it has slightly less than 2 ms to run your module code, the code of the other modules and the control loops and estimators. If your periodic function takes too long, the autopilot will run at a lower frequency than intended, which can lead to instability. In practice you have to make things pretty bad before this becomes a problem, but you should be careful when using large or nested loops in your periodic function, and video processing is best performed in the video callback function (section 4.2.3) as this callback runs in a separate thread.

Together with the periodic function, the module xml can specify a *start* and *stop* function. These are called when the module is started or stopped, respectively. The `autorun` attribute in the module xml's `periodic` element controls whether your module is started automatically or manually; you can manually start and stop modules from the GCS by going to 'Settings → System → Modules', selecting START or STOP and clicking the green checkmark. You can find an example of start and stop functions functions in `sw/airborne/modules/loggers/file_logger.c`, where they are used to open and close the log file.

MAV Exercise: Changing the autopilot behavior

Take a look at the source code for the `mav_course_exercise` module. If you are still unfamiliar with the Paparazzi project structure, a module's source code can be found in `paparazzi/sw/modules/<module_name>/` folder. Currently, the exercise module will behave exactly like `orange_avoid`, except that as soon as the first obstacle is found the drone will stop and hover. Try to change this behavior. For example, try to make the drone turn 20 degrees to the left or right instead, and attempt to continue. Take a look at `orange_avoid.c` for inspiration, and at Section 4.2.5 for additional information.

4.2.3 Handling video

Video handling in Paparazzi works similar to the module functions described above: the autopilot will call a function in your module when a new video frame becomes available. Consider the following extract from the colorfilter module, `sw/airborne/modules/computer_vision/colorfilter.c`:

```
#include "modules/computer_vision/colorfilter.h" // Includes computer_vision/cv.h!

...

struct image_t *colorfilter_func(struct image_t *img) {
   ...
   return img; // Return modified image for further processing
}

void colorfilter_init(void) {
   listener = cv_add_to_device(&COLORFILTER_CAMERA, colorfilter_func, COLORFILTER_FPS);
}
```

[1]`sw/airborne/firmwares/rotorcraft/main.c` line 68.
[2]`sw/airborne/firmwares/rotorcraft/main_ap.c handle_periodic_tasks` (line 202).

This code fragment contains two functions: a video callback function `colorfilter_func` and an init function `colorfilter_init`. As in the previous section, the init function is specified by the module xml, but this is not possible for video callbacks. Instead, the init function uses `cv_add_to_device` (from `modules/computer_vision/cv.h`) to register the video callback function. The `cv_add_to_device` function requires a video device pointer (here `&COLORFILTER_CAMERA`, which is set to `front_camera` in the airframe file), the name of the callback function (`colorfilter_func`) and the FPS at which the function should be called (`COLORFILTER_FPS`).

After registering, the video callback is called when new frames become available. The video callback function takes one argument: a `struct image_t` pointer to the current camera frame. You can perform further processing of this image inside your video callback function. At the end of the function you need to return a `struct image_t` pointer. This pointer is then used as input for the next module that subscribed to this camera. Typically this pointer is the same as the input pointer.

The video callback function takes a pointer to the latest camera frame as argument. This camera frame is an `image_t` struct, as defined in `sw/airborne/modules/computer_vision/lib/vision/image.h` (line 43). This struct gives you access to the raw pixel values in the `buf` field. Images in Paparazzi use a YUV color space, where Y is the luminance ('brightness') of a pixel and U and V specify the color. Pixel colors are stored in a UYVY format which works as follows: for each pixel, two bytes of color information are used: one byte for either the U or V value, then one byte for the Y value. In other words, each pixel is described by either a (U, Y) or (V, Y) byte pair. This means that *you cannot get both the U and V value for a single pixel*! Instead, you need to read the missing U or V value from a neighboring pixel, which is a reasonable approximation in most natural images.

While it is possible to work directly with the UYVY values (as, for instance, in `image.c`'s `image_yuv422_colorfilt` (line 154)), it is probably easier to transform this image to an OpenCV `Mat` which allows you to use OpenCV to further process the image. An example of this can be found in `sw/airborne/modules/computer_vision/opencv_example.cpp`:

```
#include "opencv_image_functions.h"
#include <opencv2/...>
using namespace cv;

int opencv_example(char *img, int width, int height) {
   // Transform image buffer img into an OpenCV YUV422 Mat
   Mat M(height, width, CV_8UC2, img);
   // Convert to OpenCV BGR
   Mat image;
   cvtColor(M, image, CV_YUV2BGR_Y422);

   /* Do OpenCV stuff here */
   ...

   // Convert back to YUV422, and put it in place of the original image
   colorrgb_opencv_to_yuv422(image, img, width, height);
   return 0;
}
```

The function first creates a `Mat` object from the image buffer, then uses OpenCV's `cvtColor` to convert it from Paparazzi's UYVY format to OpenCV's BGR. The image can then be processed as normally by OpenCV. At the end of the function, the image is transformed back to Paparazzi's UYVY format using `colorrgb_opencv_to_yuv422`. Of course, this step is only necessary if you want to pass a modified image to the next module that subscribed to this camera.

Thread safety

Unlike the module's periodic function, the video callback functions run in a separate thread from the autopilot. This means that the video callback runs *in parallel* to the autopilot loop, and as a result the processing time in your callback function can exceed 2 ms without slowing down the rest of the autopilot.

Threading, however, makes it tricky to send information from the video callback to the module's periodic function, as both functions could be running in parallel and reading/writing the same variable at the same time. Threading problems are notoriously hard to debug as they often depend on the relative timing between threads, which is more-or-less random. To prevent threading issues, variables accessed by the video thread should be protected by a *mutex*. Before accessing shared variables, a thread will attempt to lock this mutex. A mutex can only be locked once, other threads that try to lock the same mutex will wait until the mutex is unlocked again. Proper use of a mutex ensures that shared variables are accessed by only one thread at a time. A full-on introduction of mutexes is far beyond the scope of this course, instead you are recommended to follow the example from `sw/airborne/modules/computer_vision/cv_detect_color_object.c`:

```
#include "pthread.h"

static pthread_mutex_t mutex;
struct color_object_t global_filter; // Variable shared by video and ap thread
                                     // Note: in the real code this is an array,
                                     // here just a variable to simplify this example.

...

// Video callback, executed in video thread
static struct image_t *object_detector(...) {
   /* Calculate object detection results */
   ...

   pthread_mutex_lock(&mutex);
   global_filter = ... // Store results in global_filter
   pthread_mutex_unlock(&mutex);
   ...
}

...

// Module init function
void color_object_detector_init(void) {
   ...
   pthread_mutex_init(&mutex, NULL);
   ...
}

// Module periodic function
// Executed in autopilot thread
void color_object_detector_periodic(void) {
   static struct color_object_t local_filter;
   pthread_mutex_lock(&mutex);
   local_filter = global_filter; // Copy results from global_filter for processing
   pthread_mutex_unlock(&mutex);

   /* Do stuff with local_filter */
   ...
}
```

The example works as follows: this module is accessed by two threads that need to communicate with each other: a video thread that calls `object_detector` to perform computationally intensive calculations whenever a new video frame is available, and the main autopilot thread that periodically calls `color_object_detector_periodic` to further process the results of `object_detector`.

The threads communicate through the `global_filter` struct, which holds the object detection results that should be processed in the periodic function. Access to this struct is protected by the `mutex` object. The mutex object is initialized in the module's init function using `pthread_mutex_init(...)`.

When a new video frame arrives, this is processed in `object_detector`. After processing, it will store the results in `global_filter`. To prevent threading problems, the function will first lock the mutex using `pthread_mutex_lock` before it writes to `global_filter`. After writing, it immediately releases the mutex using `pthread_mutex_unlock`.

A similar mechanism is used in `color_object_detector_periodic` to read the `global_filter` struct: the function first locks the mutex, then copies global_filter to a local variable and immediately releases the lock. Note that the periodic function does not operate directly on the `global_filter` variable. The reason for this is that the video thread will hang as long as the periodic function has locked the mutex. After copying the results to a local variable, the `global_filter` is no longer necessary and can be overwritten again, which means that the mutex can be unlocked *before* further processing occurs and therefore the video thread does not need to wait as long.

In summary:

1. The video callback operates on a different thread than the rest of the autopilot. Variables that are shared between video callbacks and the rest of the code should be protected by a mutex.
2. Add a `pthread_mutex_t` object to your module and initialize this in your module's init function using `pthread_mutex_init`.
3. Before reading/writing a shared variable, lock the mutex using `pthread_mutex_lock`.
4. As soon as possible, release the mutex using `pthread_mutex_unlock`. Make a local copy of the shared variable(s) if you need more time to process them.

4.2.4 ABI: publish/subscribe messaging between modules

The simplest way to communicate *between different modules* is to share global variables between them. In more complex cases, however, the ABI messaging system provides an alternative to global variables. The ABI system allows modules to publish or subscribe to messages. Compared to global variables, this has the advantage that the modules do not need to know about each other – only about the message type – which simplifies code maintenance and makes it easier to swap out different modules, for instance different IMU or sonar drivers.

An example of ABI messaging can be found in the colored object detector (`sw/airborne/modules/computer_vision/cv_detect_color_object.c`) and the orange avoider (`sw/airborne/modules/orange_avoider/orange_avoider.c`). In this example, the colored object detector acts as publisher and the orange avoider as subscriber. The ABI message used in this example (`VISUAL_DETECTION`) is defined in `conf/abi.xml`, the sender ID (`COLOR_OBJECT_DETECTION1_ID`) in `sw/airborne/subsystems/abi_sender_ids.h`:

`conf/abi.xml`:

```
<protocol>
  <msg_class name="airborne">
    ...
    <message name="VISUAL_DETECTION" id="27">
      <field name="pixel_x"      type="int16_t">Center pixel X</field>
      <field name="pixel_y"      type="int16_t">Center pixel Y</field>
      <field name="pixel_width"  type="int16_t">Width in pixels</field>
      <field name="pixel_height" type="int16_t">Height in pixels</field>
      <field name="quality"      type="int32_t">Detection quality</field>
      <field name="extra"        type="int16_t">Extra field for options ...</field>
    </message>
    ...
  </msg_class>
</protocol>
```

`sw/airborne/subsystems/abi_sender_ids.h`:

```
...

/*
 * VISUAL_DETECTION communication (message 27)
*/
#ifndef COLOR_OBJECT_DETECTION1_ID
#define COLOR_OBJECT_DETECTION1_ID 1
#endif

#ifndef COLOR_OBJECT_DETECTION2_ID
#define COLOR_OBJECT_DETECTION2_ID 2
#endif

...
```

`sw/airborne/modules/computer_vision/cv_detect_color_object.c`:

```
#include "subsystems/abi.h"

...

void color_object_detector_periodic(void) {
   ...
   AbiSendMsgVISUAL_DETECTION(COLOR_OBJECT_DETECTION1_ID,
         local_filters[0].x_c, local_filters[0].y_c,
         0, 0, local_filters[0].color_count, 0);
   ...
}
```

`sw/airborne/modules/orange_avoider/orange_avoider.c`

```
#include "subsystems/abi.h"

...

#ifndef ORANGE_AVOIDER_VISUAL_DETECTION_ID
#define ORANGE_AVOIDER_VISUAL_DETECTION_ID ABI_BROADCAST
#endif
static abi_event color_detection_ev;
static void color_detection_cb(uint8_t sender_id, ...) { ... }

void orange_avoider_init(void) {
   ...
   AbiBindMsgVISUAL_DETECTION(ORANGE_AVOIDER_VISUAL_DETECTION_ID,
         &color_detection_ev, color_detection_cb);
}
```

To set up communication over ABI, you should first define your message in `conf/abi.xml`. Look at the existing messages to see how you should formulate your own, then make sure you set the `id` to a value that is not in use yet. After changing the xml file, you need to run `make clean` and `make` in the paparazzi folder to generate the corresponding .c and .h files. You should also modify `sw/airborne/subsystems/abi_sender_ids.h` to add one or more sender ID numbers, these are used to identify the source of the messages. Sender ID's should be unique within a message type.

Publishing messages is pretty straightforward: include the `subsystems/abi.h` header, then use `AbiSendMsg<YOUR_MESSAGE>(sender_id, ...)` to publish a message. The first argument of this function is the sender ID, the other arguments are the fields of your message as defined in `conf/abi.xml`.

Subscribing to messages is similar to subscribing to video frames. Include the `subsystems/abi.h` header to get access to the subscribe functions. Then, use `AbiBindMsg<YOUR_MESSAGE>(sender_id, ev, cb)` to subscribe to this message class. AbiBindMsg takes the following arguments: a `sender_id`, which is used to filter messages by their source ID. Set this sender_id to `ABI_BROADCAST` to accept messages from all senders. The second argument is a pointer to an `abi_event` that you declare somewhere in your code (here `color_detection_ev`). Finally, the third argument is a pointer to your callback function which gets called whenever a new message is published. The callback function takes as arguments the `sender_id`, followed by the fields defined for this message class.

ABI callbacks run in the thread that called `AbiSendMsg`, which in practice should be the autopilot thread (do *not* call AbiSendMsg from the video thread!). This means that you do not need to use mutexes to get/set global variables from the callback, but also that you should take care that your callback does not take too much time.

MAV Exercise: Add divergence module

In order to practice setting up an ABI message, we will add a check that will cause the drone to identify an obstacle if the optical flow divergence of the front camera becomes too large. A module that computes the divergence already exists, called `cv_opticflow`. Let's start by including it in our airframe by adding the following code block:

```
<module name="cv_opticflow">
   <define name="OPTICFLOW_CAMERA" value="front_camera"/>
   <define name="MAX_HORIZON" value="10"/>
   <define name="OPTICFLOW_MAX_TRACK_CORNERS" value="10"/>
   <define name="OPTICFLOW_SUBPIXEL_FACTOR" value="200"/>
   <define name="OPTICFLOW_CORNER_METHOD" value="0"/>
   <define name="OPTICFLOW_DEROTATION_CORRECTION_FACTOR_X" value="0.8"/>
   <define name="OPTICFLOW_DEROTATION_CORRECTION_FACTOR_Y" value="0.85"/>
   <define name="MAX_TRACK_CORNERS" value="10"/>
</module>
```

This code block should be added with the other `<module` *name*`>`. For many modules the order in which they are included in the airframe file does not matter, but due to certain dependencies it may be necessary to include the `cv_opticflow` module between the `video_thread` and `video_rtp_stream` for the debug output to be visible. There is a possibility that the debug output may interfere with the `orange_avoider` module.
Unlike the exercise module, the `cv_opticflow` module has a number of variables that can help tune the output, and which must be defined in the airframe. For now simply use these values; you will be able to play around with them once everything works.

If you check the module xml, you will see that this module comprises of a number of source files, with the main one being `opticflow_module.c`. This file collects the optic flow result and sends out the data in an ABI message, and it is this message that you will need to subscribe to. Try to include the necessary variables and functions needed to subscribe to the OPTICAL_FLOW message in `mav_course_exercise.c`, by using the information described in Section 4.2.4 and using the already existing VISUAL_DETECTION message as reference. The divergence value that is of interest in this case is the last message argument, the `div_size`.

Once the module `mav_course_exercise` is set up to receive the optic flow results, it is a simple matter of adding and if-statement in the navigation part of the code to check whether the divergence value is higher than a threshold, for instance 0.3, in which case the navigation state can be set to OBSTACLE_FOUND. This divergence value may not reliably find obstacles and that's ok, don't spend too much time tuning it at this stage. Try however to print out the divergence values to make sure the modules are communicating correctly.

4.2.5 Getting the drone's state

At some point in your module, you may want to get the current state of the drone. An example can be found in `sw/airborne/modules/orange_avoider/orange_avoider.c`, where the drone's position and heading are used to update the location of the next waypoint:

```
#include "state.h"

...

static uint8_t calculateForwards(struct EnuCoor_i *new_coor, float distanceMeters)
{
  float heading  = stateGetNedToBodyEulers_f()->psi;

  // Now determine where to place the waypoint you want to go to
  new_coor->x = stateGetPositionEnu_i()->x +
      POS_BFP_OF_REAL(sinf(heading) * (distanceMeters));
  new_coor->y = stateGetPositionEnu_i()->y +
      POS_BFP_OF_REAL(cosf(heading) * (distanceMeters));
  VERBOSE_PRINT("Calculated %f m forward position. x: %f  y: %f "
                "based on pos(%f, %f) and heading(%f)\n", distanceMeters,
                POS_FLOAT_OF_BFP(new_coor->x), POS_FLOAT_OF_BFP(new_coor->y),
                stateGetPositionEnu_f()->x, stateGetPositionEnu_f()->y,
                DegOfRad(heading));
  return false;
}

...
```

To get the drone's current state, include the `state.h` header. Then, the state can be accessed using the `stateGet...` functions. Scroll through the `state.h` header to get an idea of the values you can access.

State.h uses the following naming conventions: functions ending in `_f` return the drone's state in a floating-point format (recommended). These are the easiest to work with, but can be more computationally intensive on small microcontrollers. Functions ending in `_i` return the current state in a Binary Fixed-Point (BFP) format. The `_BFP_OF_REAL` and `_FLOAT_OF_BFP` macros in `sw/airborne/math/pprz_algebra_int.h` allow you to convert data between these formats.

The drone's state can be expressed in different coordinate frames. For this course, the North-East-Down (`Ned`) and East-North-Up (`Enu`) frames are likely the most relevant. Positions in these frames are expressed relative to the flight plan origin, which for the example flight plans lies in the center of the Cyberzoo. Angular rates and Euler angles are expressed in the body frame, which follows a Front-Right-Down convention.

4.2.6 Navigation and guidance commands

To avoid obstacles, your module will at some point need to control where the drone is going. Depending on the level of control you want there are multiple ways to achieve this.

NAV mode

The flight plan lets you describe an entire flight or mission and lets you give high-level commands to the drone, such as waypoint sequences to follow. The flight plan by itself does not give you fine-grained enough control to avoid obstacles, and will therefore not be discussed in further detail here.

You can, however, interact with the flight plan and waypoints from within your own module, which is the method used in `sw/airborne/modules/orange_avoider/orange_avoider.c`. Consider the following flight plan and code extracts:

`conf/flight_plans/tudelft/course_orangeavoid_cyberzoo.xml`:

```
<flight_plan ...>
   ...
   <waypoints>
```

```
      ...
      <waypoint name="GOAL" x="1.9" y="1.0"/>
      ...
   </waypoints>
   ...
   <blocks>
      ...
      <block name="START" ...>
         <call_once fun="NavSetWaypointHere(WP_GOAL)"/>
         <stay wp="GOAL"/>
      </block>
   </blocks>
</flight_plan>
```

`sw/airborne/modules/orange_avoider/orange_avoider.c`:

```
#include "firmwares/rotorcraft/navigation.h"

...

uint8_t moveWaypointForward(uint8_t waypoint, float distanceMeters);
uint8_t moveWaypoint(uint8_t waypoint, struct EnuCoor_i *new_coor);

...

void orange_avoider_periodic(void) {
   /* State machine logic left out for this example */
   moveWaypointForward(WP_GOAL, moveDistance);
   ...
}

uint8_t moveWaypoint(uint8_t waypoint, struct EnuCoor_i *new_coor)
{
  waypoint_set_xy_i(waypoint, new_coor->x, new_coor->y);
  return false;
}

uint8_t moveWaypointForward(uint8_t waypoint, float distanceMeters)
{
  struct EnuCoor_i new_coor;
  calculateForwards(&new_coor, distanceMeters);
  moveWaypoint(waypoint, &new_coor);
  return false;
}
```

In this example, the flight plan does not seem to do much: in the `START` block, it first places the `GOAL` waypoint at the drone's current position, then stays at this waypoint. Indeed, without the orange avoider module the drone would just stay in place. The avoidance behavior in this case comes from the orange_avoider's periodic function. In this function it moves the GOAL waypoint to a new location. Because the drone is commanded to stay at this waypoint, it will move to follow it.

Using the flight plan and waypoints works, but has its limitations. While you can control where the drone is going, it is not possible to control the drone's velocities. Furthermore, this method assumes that the drone's position is fully known, which may not always be the case such as when flying without Optitrack.

GUIDED mode

As an alternative, the drone can be controlled in GUIDED mode. In this mode, your module can directly send setpoints to the guidance controller, including setpoints for the drone's velocity. An example is provided in `sw/airborne/modules/orange_avoider/orange_avoider_guided.c` and the corresponding flight plan:

`conf/flight_plans/tudelft/course_orangeavoid_cyberzoo_guided.xml`:

```
<flight_plan ...>
   <header>
      ...
      inline void setNav(void) {
         autopilot_mode_auto2 = AP_MODE_NAV;
         autopilot_static_set_mode(AP_MODE_NAV);
      }
      inline void setGuided(void) {
         autopilot_mode_auto2 = AP_MODE_GUIDED;
         autopilot_static_set_mode(AP_MODE_GUIDED);
      }
   </header>
   ...
   <blocks>
      ...
      <block name="START" ...>
         <call_once fun="setGuided()"/>
         <stay wp="STDBY"/>
      </block>
      <block name="STOP">
         <call_once fun="NavSetWaypointHere(WP_STDBY)"/>
         <call_once fun="setNav()"/>
         <stay wp="STBDY"/>
      </block>
      ...
      <block name="Land here" ...>
         <call_once fun="setNav()"/>
         ...
      </block>
   </blocks>
</flight_plan>
```

`sw/airborne/modules/orange_avoider/orange_avoider_guided.c`:

```
#include "firmwares/rotorcraft/guidance/guidance_h.h"

...

void orange_avoider_guided_periodic(void) {
   if (guidance_h.mode != GUIDANCE_H_MODE_GUIDED) {
      return;
   }

   /* State machine logic left out for this example */
   guidance_h_set_guided_body_vel(speed_sp, 0);
   ...
}
```

In this example, the module uses `guidance_h_set_guided_body_vel` to send a velocity setpoint to the guidance controller. However, when the drone is in NAV mode, it will ignore these commands and follow the navigation commands from the flight plan instead.

To switch the autopilot to GUIDED mode, two extra functions were added in the flight plan header: `setNav` and `setGuided`. Calling these functions sets the autopilot mode to NAV or GUIDED, respectively. This is exactly what happens in the `START` block: the drone is commanded to stay at the STDBY waypoint, but just before that the autopilot is switched to GUIDED mode by calling `setGuided`. As a result, the drone will ignore the stay command and follow the module's setpoints instead. In the `STOP` block, the autopilot is switched back to NAV mode and stays at the STDBY waypoint.

You should take special care that the drone is in the correct mode at the start of each block. For instance, forgetting to switch the drone back from GUIDED mode to NAV in the `Land here` block would cause it to ignore the landing command, even when the landing was triggered by a safety exception! It is therefore best practice to call the `setNav` or `setGuided` function at the start of each block to ensure that the autopilot is in the correct mode.

MODULE mode

For completeness, it should be mentioned that your module can also *replace* the entire guidance or even stabilization controller. This can be achieved by switching the autopilot to MODULE mode. In module mode, the autopilot will call the `guidance_h_module_run` and `guidance_v_module_run` functions during its main loop, your module should provide these functions.

Two examples of MODULE mode controllers are provided in `sw/airborne/modules/ctrl/ctrl_module_innerloop_demo.` and `_outerloop_demo.c`. In MODULE mode *you* are responsible for the drone's stability, therefore we recommend you to stick to NAV or GUIDED mode during the course.

4.2.7 Defines and settings

Your module will most likely contain tunable parameters, such as the minimum and maximum YUV values for which a pixel is considered an obstacle. While you can write these numbers directly in your code, this will make it difficult to tune them later. Paparazzi provides two systems to simplify parameter tuning: defines and settings.

Defines allow you to set constant values from the airframe file. See, for example, the following abstract of the `bebop_course_orangeavoid.xml` airframe:

```xml
<airframe ...>
   <firmware ...>
      <target name="ap" board="bebop">
         <define name="COLOR_OBJECT_DETECTOR_LUM_MIN1" value="40"/>
         ...
      </target>
      ...
      <define name="ARRIVED_AT_WAYPOINT" value="0.5"/>
      ...
      <module name="cv_detect_color_object">
         <define name="COLOR_OBJECT_DETECTOR_CAMERA1" value="front_camera"/>
         ...
      </module>
   </firmware>
   ...
   <section name="GUIDANCE_H" prefix="GUIDANCE_H_">
      <define name="CLIMB_VSPEED" value="1.0"/>
   </section>
   ...
</airframe>
```

As you can see, defines can be set at multiple places in the airframe file. The behavior is mostly the same in these cases, with the following exceptions:

- Defines placed in the `<target>` elements are only set when the autopilot is built for that target, i.e. `"ap"` for the real drone and `"nps"` for the simulator. This allows you to, for instance, use different color filter settings on the real and simulated drone.
- Placing a define inside a `<module>` element has no special effect! The define is also visible in other modules, so be sure to use a unique name. Typically, defines are prefixed with the name of the module (e.g. `COLOR_OBJECT_DETECTOR_`) to make them unique. The only reason these defines are placed inside the module element is to improve readability.

- `<section>` elements allow you to specify a `prefix`, this prefix is placed in front of all define names inside this section. In the example, the `CLIMB_VSPEED` define is available in the code as `GUIDANCE_H_CLIMB_VSPEED`.

During compilation, these defines are turned into preprocessor macros and can be referred to directly from your code.

Airframe defines allow you to set constant parameters at compile-time, but in some cases it would be easier if you could change these values during the flight. This is possible with the 'settings' mechanism. Settings are defined in the module xml file. Take for example `conf/modules/cv_detect_color_orange.xml`:

```
<module name="cv_detect_color_object" ...>
   ...
   <settings>
      <dl_settings name="ColorObjectDetector">
         <dl_setting var="cod_lum_min1" min="0" step="1" max="255" shortname="y_min1"/>
         ...
      </dl_settings>
   </settings>
</module>
```

Settings listed in the module xml can be tuned from the Ground Control Station by going to the 'Settings' tab and then selecting the tab belonging to your module, as defined in the `dl_settings` element (here `ColorObjectDetector`). To read the current value of a parameter from the drone, click its value (the number) in the GCS. Te set a value on the drone, adjust the slider, *then click the green checkmark* to upload this new value to the drone. Click the value number again to make sure the setting was updated.

Use the `dl_setting` element in your module xml to add a setting to your module. The `var` attribute specifies the variable this setting should be written to; this variable should be globally accessible. The `min`, `step` and `max` attributes let you specify a range of possible values for this setting. Finally, using `shortname` you can control the name under which this setting is listed in the GCS.

It is possible to combine the define and settings mechanisms, where the define provides a default value that can be adjusted later using settings. This often uses the following pattern:

```
#ifndef MY_DEFINE
#define MY_DEFINE 0
#endif
int my_setting = MY_DEFINE;
```

In this example, `MY_DEFINE` provides the initial value of `my_setting`. MY_DEFINE can be set from the airframe file, but if it is not defined there this code will give it a default value of 0. The actual parameter is stored in `my_setting`, for which a `<dl_setting>` element is included in the module's xml file.

MAV Exercise: Setup GCS variables

In the previous parts of the exercise you set up a heading change in case of OBSTACLE_FOUND, and a divergence threshold to detect obstacles. These types of variables often require some tuning, and it is very time consuming to have to compile and upload your code every time you want to tweak them. Using the information in this section, make the changes necessary to be able to change the values of these two variables online, through the GCS.

4.3 Logging

While writing your module, you will likely want feedback on what is happening in your code while the module is running. Especially when things go wrong (and they will), logging is essential in tracking down and fixing problems. In this section you will find a brief overview of approaches to logging.

4.3.1 Printing to terminal

Perhaps one of the easiest approaches to logging is to make your module print messages to a terminal. While running the simulations of the earlier parts, you may already have noticed that the example modules print information to the terminal in the Paparazzi Center:

```
[orange_avoider->calculateForwards()] Calculated 1.000000 m forward position.
  x: -0.164062  y: 0.968750 based on pos(-0.007860, -0.017677) and heading(-9.120324)
[orange_avoider->moveWaypoint()] Moving waypoint 5 to x:-0.164062 y:0.968750
```

You can add your own debug messages as follows: first, include the `<stdio.h>` header to get access to the `printf` function. Then, at any point in your code, use `printf` to print your debug message. Printf requires a format string to know how to print the variables you specify, follow the link to find out more.

When running in simulation the printf messages will automatically appear in the Paparazzi Center. You can also read these messages on the real drone using the following steps:

1. Open a terminal window (Ctrl+Alt+T)
2. Make sure you are connected to the drone's WiFi access point, the open a telnet connection using `telnet 192.168.42.1`.
3. Navigate to the Paparazzi folder using `cd data/ftp/internal_000/paparazzi`.
4. You need to restart the autopilot for the messages to show up. Run `killall -9 ap.elf`, then `./ap.elf` to restart the autopilot.
5. When you are finished and the drone is landed, kill the autopilot by pressing Ctrl+C twice, then disconnect using Ctrl+D.

4.3.2 Telemetry

Paparazzi also provides a telemetry system to track and plot variables in real-time. Adding your own telemetry message requires you to change multiple files, which will be summarized here.

First of all you need to define your telemetry message. For this, you will need to edit `conf/messages.xml`, but if you are running a clean installation of Paparazzi this file will not exist. Copy this file from `sw/ext/pprzlink/message_definitions/v1.0/` and paste it in the `conf/` folder. This will override the default messages provided by pprzlink and allow you to add your own. The `messages.xml` file contains a collection of message types, their id numbers and their fields. The file looks as follows:

```
<protocol>
   <msg_class name="telemetry" id="1">
      ...
      <message name="DRAGSPEED" id="38">
         <description>...</description>
         <field name="u_est" type="float" unit="m/s"/>
         ...
      </message>
      ...
   </msg_class>
</protocol>
```

The `message` element describes a message class. Each message class is given a unique `id` number between 0 and 255. Most id numbers are already taken, you can find out which ones are free by running `./sw/tools/find_free_msg_id.out` from the paparazzi folder or by looking for comments such as `<!-- 45 is free -->`. The message itself is defined by a collection of `fields`, for which you need to specify the `name`, data `type` and optionally a `unit`.

Paparazzi needs to generate c code to send and receive your message class. After changing the messages.xml file, you will need to remake Paparazzi. Close paparazzi if it is still running, open a terminal (Ctrl+Alt+T), go to the paparazzi folder `cd paparazzi` and run `make clean && make`.

After adding your message class, you need to inform Paparazzi that it should actually send this message. For this you need to edit the drone's telemetry xml, which in the example aircraft is `conf/telemetry/default_rotorcraft.xml`:

```
<telemetry>
   <process name="Main">
      <mode name="default" ...>
         ...
         <message name="DRAGSPEED" period="0.02"/>
      </mode>
      ...
   </process>
</telemetry>
```

Now, Paparazzi will know that it has to send the message, but it does not know *how* yet.

For this, you will need to edit your module. We take the dragspeed module (`sw/airborne/modules/dragspeed/dragspeed.c`) as an example:

```
#include "subsystems/datalink/telemetry.h"

...

void dragspeed_init(void) {
   ...
   // Register callbacks
   register_periodic_telemetry(DefaultPeriodic, PPRZ_MSG_ID_DRAGSPEED, send_dragspeed);
   ...
}

...

static void send_dragspeed(struct transport_tx *trans, struct link_device *dev) {
   // Calculate INS velocity in body frame
   ...
   // Send telemetry message
   pprz_msg_send_DRAGSPEED(trans, dev, AC_ID, &dragspeed.vel.x, &dragspeed.vel.y,
         &vel_ins_body.x, &vel_ins_body.y);
}
```

In the module's init function, it registers a callback function to send the DRAGSPEED telemetry message. The callback is registered using `register_periodic_telemetry`, which takes as arguments a periodic_telemetry pointer (keep this one at `DefaultPeriodic`), the ID number of your message class (`PPRZ_MSG_ID_<YOUR MESSAGE NAME>`), and the callback function.

The callback function, here `send_dragspeed`, will actually send the telemetry message. The function takes two arguments: a `struct transport_tx *trans` and a `struct link_device *dev`. You do not need to use or understand these structs, you only need to pass them along to the `pprz_msg_send_<YOUR MESSAGE NAME>` function. Call this function with `trans, dev, AC_ID` as the first three arguments, the following arguments are pointers to the fields of your message.

To receive your telemetry messages, go to the Paparazzi Center, make sure your session is running, then click 'Tools → Messages'. To plot your telemetry values over time, click 'Tools → Real-time Plotter', then drag the telemetry message field from the 'Messages' window to the 'Plotter' window. Use the sliders in the plotter window to adjust the update rate and timespan of the plotter.

4.3.3 Video streaming

When developing computer vision modules, it can be helpful to view the camera images of the drone, possibly with annotations by your module. The `video_rtp_stream` module lets you stream video back to your pc; this module has already been added to the example airframes.

To receive the video stream, start VLC and open `sw/tools/rtpviewer/rtp5000.sdp` to view `VIEWVIDEO_CAMERA` (the front camera) and `sw/tools/rtpviewer/rtp6000.sdp` to view `VIEWVIDEO_CAMERA2` (the bottom camera). Note that there might be a delay on the video; this delay is caused by the video streaming and is not visible to the modules.

4.3.4 CSV file logging

Printing to terminal, telemetry and video streaming give you live feedback of your drone's behavior, but for some purposes it might be useful to analyze the results in more detail after the flight. To help you with this, we have included a CSV file logger module (`logger_file`) that will periodically write the values of important variables to a csv file that you can analyze later using python or matlab.

The csv logger is already included in the example airframes. To start logging, go to the 'Settings' tab in the GCS, then to 'System → Modules'. Set `file_logger_periodic` to START and press the green checkmark to start recording. After your flight, press STOP, then the green checkmark to stop recording and save the csv file.

You can change the variables that are being logged by editing `sw/airborne/modules/loggers/file_logger.c`:

```c
...

static void file_logger_write_header(FILE *file) {
  fprintf(file, "time,");
  fprintf(file, "pos_x,pos_y,pos_z,");
  fprintf(file, "vel_x,vel_y,vel_z,");
  fprintf(file, "att_phi,att_theta,att_psi,");
  fprintf(file, "rate_p,rate_q,rate_r,");
  fprintf(file, "cmd_thrust,cmd_roll,cmd_pitch,cmd_yaw\n");
}

static void file_logger_write_row(FILE *file) {
  struct NedCoor_f *pos = stateGetPositionNed_f();
  struct NedCoor_f *vel = stateGetSpeedNed_f();
  struct FloatEulers *att = stateGetNedToBodyEulers_f();
  struct FloatRates *rates = stateGetBodyRates_f();

  fprintf(file, "%f,", get_sys_time_float());
  fprintf(file, "%f,%f,%f,", pos->x, pos->y, pos->z);
  fprintf(file, "%f,%f,%f,", vel->x, vel->y, vel->z);
  fprintf(file, "%f,%f,%f,", att->phi, att->theta, att->psi);
  fprintf(file, "%f,%f,%f,", rates->p, rates->q, rates->r);
  fprintf(file, "%d,%d,%d,%d\n",
      stabilization_cmd[COMMAND_THRUST], stabilization_cmd[COMMAND_ROLL],
      stabilization_cmd[COMMAND_PITCH], stabilization_cmd[COMMAND_YAW]);
}

...
```

The function file_logger_write_header writes the column names in the first line of the csv file. Then, for each periodic function call, file_logger_write_row writes a new row to the csv file. When modifying the file logger, make sure that the fprintf's in write_row match the column names of write_header. Also, do not forget to end the final fprintf statements with a ‘\n’ to end the line!

When logging from the simulator, you can find the csv files in /tmp/paparazzi/log/<timestamp>.csv. Section 4.3.6 will show how to download the csv files from the real drone.

4.3.5 Video capture

To help you develop vision algorithms, the video_capture module lets you record camera images from your drone (both in simulation and on the real drone). The module is already included in the example airframe files. To start and stop recording, use the buttons in the top-left of the GCS.

Video frames recorded in the simulator can be found in /tmp/paparazzi/images/<timestamp>/. the images are named with a timestamp in microseconds since the autopilot was started, this timestamp matches the one recorded by the csv file logger. Section 4.3.6 will show how to download the images from the real drone.

(Note: the image folder name is determined at the start of the autopilot, not the start of recording. If you record multiple videos during one flight, they will be stored in the same folder.)

4.3.6 Downloading log files through FTP

After recording your log and/or video files, you can download them from the real drone over FTP using the following steps:

1. Make sure you are connected to the drone's WiFi access point.
2. Open Ubuntu's file browser.
3. On the left, click ‘Connect to Server’.
4. Enter ftp://192.168.42.1 as the server address.
5. Click ‘Connect As → Anonymous’ then click ‘Connect’.
6. You can now access the drone's file system. The log files are stored in internal_000/log/, the video frames in internal_000/images/. Use the file browser to copy them to your pc. (Note: depending on your version of Ubuntu, you may need to use ‘Right click → Copy’ instead of Ctrl+C to copy the files).

4.4 Testing

During development, you will regularly want to test your code to check that everything works as expected. While it is tempting to test everything on the real drone, this is often time-consuming and risky. Instead, we recommend you to do a large part of your testing off-board. A large part of development can be done on datasets that you collected beforehand (see the previous section), and closed-loop tests can be performed in simulation before the code is finally tested on the real drone.

4.4.1 Offline development with Python and OpenCV

While it is possible to develop your vision algorithms directly in Paparazzi, the frequent recompiling and lack of interactivity of c makes this a time-consuming process. Instead, we suggest that you prototype your vision algorithms in Python. Python code is in general a bit easier to write, but the biggest advantage of Python is that you can run it interactively (IPython). This makes it significantly easier to try out stuff and play around with the functions that OpenCV has to offer.

In this section we show how to install and use Python (3), OpenCV and Spyder. Spyder is a python editor that is aimed specifically at scientific computing and has excellent IPython support. As for the actual development of your vision code: that is something we leave up to you.

To start the installation, use apt to install the most important components. Open a terminal (Ctrl+Alt+T) and enter:

```
sudo apt install python3 pip3 virtualenv spyder3
```

Next, you need to install OpenCV for python. It is possible to do a system-wide installation of OpenCV, however the Bebop uses an older version of openCV (3.4.5) that we would like to match in python. For obvious reasons, installing an outdated version of a library on your entire system is not ideal. Instead, we will set up a virtual environment (a stand-alone version of python and its libraries) that we will install opencv to.

1. Open a terminal (Ctrl+Alt+T)
2. Create a new folder to store your virtual environment:
   ```
   mkdir -p ~/envs/mavlabcourse
   cd ~/envs/mavlabcourse
   ```
3. Create the virtual environment:
   ```
   virtualenv -p python3 env
   ```

Now we need to install opencv to this virtual environment. Use to following steps to install opencv (and other packages):

4. Activate the virtual environment:
   ```
   source env/bin/activate
   ```
   You should now see `(env)` before the terminal prompt.
5. Install opencv 3.4.5:
   ```
   pip3 install opencv-python==3.4.5.20
   ```
6. To simplify plotting and drawing, we will also install matplotlib (pyplot):
   ```
   pip3 install matplotlib
   ```
7. When you are finished installing packages, run `deactivate` to deactivate the virtual environment. The `(env)` should disappear.
8. Set up Spyder to use the virtual environment:
   - Launch Spyder:
     ```
     spyder3
     ```
   - Set up the python interpreter. In the menu, click 'Tools → Preferences'. Navigate to 'Python interpreter'. Select 'Use the following Python interpreter', then click the file button and select the `python3` executable in your environment's `bin` folder (`~/envs/mavlabcourse/env/bin/python3`).
   - Click 'Apply', 'Ok', then close Spyder.

9. To verify that your virtual environment is working correctly, restart Spyder and enter the following lines in the IPython console:

   ```python
   import cv2
   cv2.__version__
   ```

   This should return the following result: '3.4.5'.

As for the actual development, we recommend you to get familiar with OpenCV (yes, the documentation is awful...). The 'OpenCV-Python Tutorials' are a good place to start if you have not worked with OpenCV before.

Before the course, we collected some example datasets on the Bebop. You can download these here: `https://surfdrive.surf.nl/files/index.php/s/68S6moWEIqCT14K`. Use `cv2.imread` to open the images and start experimenting.

4.4.2 Testing in simulation

After developing and implementing your vision code as a Paparazzi module, you will want to test its behavior in a closed loop with the rest of the autopilot. Testing in simulation has several advantages over testing on the real drone:

- You do not risk breaking the drone if you make a mistake.
- You do not need to wait to charge batteries.
- You do not need to come to the Cyberzoo.
- You do not need to share your virtual drone between your teammates.

All the example code has been tested in simulation before it was used on the real drone, we recommend you to follow a similar workflow as it will save you a lot of time.

Paparazzi uses Gazebo as a simulation platform. Gazebo allows you to create virtual worlds and robots from xml files, and provides a physics and rendering engine and a collection of virtual sensors.

In the mavlabCourse2021 branch of Paparazzi, we have provided you with a collection of Gazebo worlds and models that are relevant for the course. These can be found in `conf/simulator/gazebo/` and in `sw/ext/tudelft_gazebo_models/`. Three Cyberzoo world files have already been provided in the `sw/ext/tudelft_gazebo_models/world/` folder:

- `cyberzoo2019_orange_poles.world` - The world file used by the example airframes, consisting of the Cyberzoo with a handful of orange poles to avoid.
- `cyberzoo2019_orange_poles_panels.world` - Same as above, but with black metal panels added as additional obstacles.
- `cyberzoo2019_orange_poles_panels.world` - Cyberzoo with orange poles and metal panels, with traffic mats added on the ground to make 'green detectors' harder to use...

The placement of the obstacles should roughly match that of the obstacles in the real dataset.

The Gazebo world that is used by Paparazzi is specified in the flight plan, e.g.
`conf/flight_plans/tudelft/course_orangeavoid_cyberzoo.xml`:

```xml
<flight_plan ...>
   <header>
      ...
      #define NPS_GAZEBO_WORLD "cyberzoo_orange_poles.world"
   </header>
   ...
</flight_plan>
```

You can change this define to load a different world file. Paparazzi will search for world files in the two folders mentioned above.

Check out the SDFormat spec for more information on the content of the world (and model) files. If you want to create your own world, it is probably easiest to copy one of the existing Cyberzoo worlds and edit that. *Save your modified files in the `conf/simulator/gazebo/` folder!* That way your changes are tracked in your Paparazzi branch. The `sw/ext/tudelft_gazebo_models` folder is a git submodule, which makes tracking and sharing changes there a real headache.

The simulation is of course not perfect, you will quickly discover that the camera colors and calibration in the sim may differ from those on the real drone. This is an important reason not to hard-code your tuning parameters (also see subsection 4.2.7) as this means that your module works either in the sim or on the real drone, but not both! Instead, you can use the airframe xml to supply a different set of parameters depending on whether you are compiling the autopilot for the sim or the real drone. See for example `conf/airframes/tudelft/bebop_course_orangeavoid.xml`:

```
<airframe name="bebop_avoider">
  ...
  <firmware name="rotorcraft">
    <target name="ap" ...>
      ...
      <define name="COLOR_OBJECT_DETECTOR_LUM_MIN1" value="40"/>
      <define name="COLOR_OBJECT_DETECTOR_LUM_MAX1" value="145"/>
      ...
    </target>
    <target name="nps" ...>
      ...
      <define name="COLOR_OBJECT_DETECTOR_LUM_MIN1" value="41"/>
      <define name="COLOR_OBJECT_DETECTOR_LUM_MAX1" value="183"/>
      ...
    </target>
    ...
  </firmware>
  ...
</airframe>
```

This airframe provides two compilation targets: `ap` – the autopilot for the real drone, and `nps` – the simulator. Placing your defines inside the `target` elements allows you to change your parameters depending on the compilation target, which in turn should allow you to keep running *the same code* in the simulator and on the real drone.

4.4.3 Testing on the real drone

After testing your code in simulation, you will need to test the same code on the real drone. The previous parts of the course manual have already shown you how to fly your drone with Paparazzi. Be sure to use the logging capabilities described in section 4.3 to help you track down problems. Keep in mind to *follow the safety rules* (part 2), especially when flying experimental code. Also, do not forget to start your loggers before flying.

4.5 Collaboration using git

Refer to the lecture slides (`https://brightspace.tudelft.nl/d2l/le/content/133606/viewContent/1231699/View`) for more information on using git.

4.6 Where to find more information

For a *high-level overview* of Paparazzi and descriptions of file types such as the *flight plan xml* or *module xml*'s, refer to the Paparazzi wiki at `https://wiki.paparazziuav.org`.

For more in-depth descriptions of *existing modules* and their defines, check out the auto-generated documentation at `http://docs.paparazziuav.org/latest/`. The content of these pages is generated directly from the module .xml files, which can be found in the `paparazzi/conf/modules` directory.

If you are experiencing *issues with Paparazzi that are not caused by your own code*, check out the issue pages on Github (`https://github.com/paparazzi/paparazzi/issues`, `https://github.com/tudelft/paparazzi/issues`) to see if someone else is running into the same problem and if there is a temporary solution or fix. **If you want to create a new issue, please do so on the tudelft fork of Paparazzi** (`https://github.com/tudelft/paparazzi/issues`).

If you really want to know *what goes on inside the autopilot*, you can't get more detailed information than the source code itself! All of the autopilot code can be found in the `paparazzi/sw/airborne/` directory. The following files and directories are most relevant for the course:

- `paparazzi/sw/airborne/`
  - `firmwares/rotorcraft/` - *This folder contains the autopilot code that is shared between all types of rotorcraft. This includes guidance and stabilization code.*
    - `guidance/` - *Here you can find the guidance controllers. The controller is split into a horizontal part (guidance_h) and a vertical part (guidance_v).*
    - `stabilization/` - *This folder contains the stabilization controllers. The example airframe uses the stabilization_indi_simple controller.*
    - `autopilot_guided.h` - *This header allows you to set position and velocity setpoints while flying in GUIDED mode.*
    - `main.c` - *If you were wondering if Paparazzi also has a 'main' function, here it is. After initializing the autopilot, the main function will loop indefinitely while performing its periodic tasks.*
    - `navigation.h` - *These functions are commonly called from the flight plan.*
  - `math/` - *Here you can find math functions for matrix and vector operations in Paparazzi. These are documented in paparazzi/doc/pprz_algebra/headfile.pdf. Pprz_algebra_float.h might be a good starting point for general-purpose floating point calculations.*
  - `modules/` - *Here you find the code for modules that other people have written. Be sure to check out how these modules are written if you're looking for inspiration.*
    - `computer_vision/` - *This folder contains the computer vision code. Use the functions in cv.h to register your video callback.*
  - `state.h` - *Use this header to read the current state of the UAV. Use the stateGet... functions.*

The autopilot code is spread over a large number of files. Use Eclipse to navigate these files effectively. Hovering over a variable or function will show its definition and hopefully some descriptive comments. Ctrl+clicking on a variable will take you to its definition or declaration. Ctrl+clicking will also allow you to quickly open header files, then use Ctrl+Tab to open the corresponding source file. If 'Mark Occurrences' (Shift+Alt+O) is turned on, placing your text cursor on the variable name will highlight all occurrences in the file you have opened. To see how a variable is used across files, right click it and choose 'References → Project'. Finally, 'Search' (Ctrl+H) allows you to search arbitrary text in all files. Set ‘Scope’ to ‘Selected resources’ to limit your search to the folder you have selected in the Project Explorer.

If your question is not answered by these resources or if you don’t know where to look, feel free to ask the TA’s for advice.