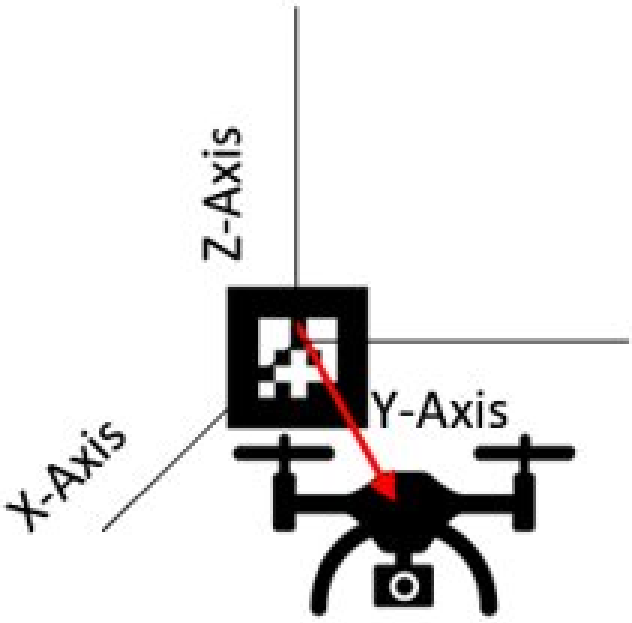
Report and setup manual of the UAV

Project Visual servoing UAV

2021-2022-S1

23/01/2022



Supervisors:

Hanieh Esmaeeli

Mark Reiling

Students:

Guido Smit(S486872)

Rob Meester (S481100)

Hugo Kunneman (S477191)

Inhoud

1. Introduction

The Saxion Mechatronics lectorate has a division specializing in drone development. They are looking into the development of autonomous working drones which can for example be used for inspection of objects, areas, or other events. Some of these working situations can be dangerous, expensive, or very work intensive if done by humans and it might be more efficient if they are done by autonomous drones.

For this reason, one of the projects that is being worked upon is an autonomous drone that can take over the process of inspecting the turbine blades of a windmill. The drone will be sent to a location near the blade of the windmill, around the area where a feature or crack in the blade has been found and it needs to inspect this feature more thoroughly to confirm or reject actual damages to the turbine blade. To do this, the drone needs to localize itself in comparison to the feature and whilst GPS is reasonably accurate, for this instance something more accurate is required.

This is where this project comes into play. As a step towards developing these autonomous inspection drones, this project investigates the possibility of a drone flying towards a feature, locking on to the feature and then autonomously hovering in front of this feature without the use of external localization equipment (like GPS or the Optitrack).

1.1 Concept

To accomplish this, a choice has been made to use vision software in combination with ROS, Pixhawk 4, RC-control and the on-board software and hardware which should help localize the drone. In this project it is required that the object in question is stationary which helps simplify an initial set-up. To help futureproof this system, a choice has also been made to make the parts of the system modular.

For this project that means the project has been split into 3 pieces. First, the vision system, which should find a feature, determine its location in respect to the camera and from this information, send x, y and z location data. The second system should be the vision controller. This controller accepts the x, y and z location data and uses it to determine the pose of the drone, compared to the feature and the frame that it is in. Because the x, y and z data could come from any vision program, in the future the current vision program could be swapped out for a different vision program, as long as it sends the same data type. It is also for this reason that data is send within conventional ROS objects like PoseStamped. If a proper PoseStamped object has been send from the vision system, the vision controller should be able to handle the data. Below you can see the architecture of the currently used nodes, where the "/ch_mov" node, which is the output of the vision program sends its values towards the "/offb_node", which is the vision controller node. This means that the programs which provide the input of the "/ch_mov" node could be replaced by a different vision program.

Lastly there is a RC-controller part. This controller should be able to communicate with the drone in order to initially get the drone towards a proper starting position (in front of a feature) and it should be able to switch the drone from normal flight mode into an offboard mode.

Lastly this project has been tested with the current versions:

Ubuntu: 18.04 Bionic

Ros: Melodic

2. Visual *part*

To let the drone fly at a steady position in front of a feature, for example a crack in a wind turbine, you need a program to determine the drone's position in front of the feature. This program must be able to read the location of the feature. This program will be described in this chapter.

2.1 Node parameters /input output

To determine the position of the drone in front of the crack in the wind turbine, we used a qr-code as a replacement for the crack. Because you know the dimensions of the qr-code and the field of view (FoV) of the camera, this can be used as an initial vision program for the drone.

This part contains three ROS nodes (python):

- *det_movement3.py*
- *distance_scanning2.py*
- *featute_scanning4.py*

The first node that the user will start is *feature_scanning4.py*. In this node, the camera will be used to scan the qr-code. Here multiple packages are used. Opencv to read the camera and pyzbar to detect the qr-code and determine its position on the frame. The output will be sent on */ch_scan*. The output is:

- frame width (im_w) [pixels]
- frame height (im_h) [pixels]
- Field of view (FoV) [deg]
- Offset of horizontal location from middle point (x_c) [pixels]
- Offset of vertical location from middle point (y_c) [pixels]
- heigth of qr code (h) [pixels]
- width of qr code (w) [pixels]

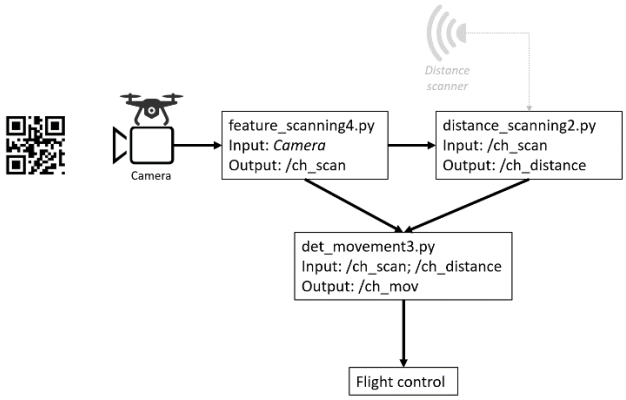
In *distance_scanning2.py* the channel */ch_scan* will be used to determine the distance from the camera to the qr-code. This is possible when the dimensions of the qr-code are known. The output will be sent by */ch_distance* and is:

- distance from camera to feature (z) [cm]

The code in this node is not integrated in *feature_scanning4.py*, because now the user can easily replace this node with a real distance scanner while there are no changes needed to the other nodes.

The last node is *det_movement3.py*. This node receives */ch_scan* and */ch_distance*. With this data the distance in all three dimensions of the qr-code is calculated. The results will be sent on */ch_mov* and can be used in the flight control. */ch_scan* contains:

- x offset (x) [cm] (left/right)
- y offset (y) [cm] (up/down)
- z offset (z) [cm] (forward/backward)



2.2 Instructions

1. Create a workspace for catkin

```
$ mkdir -p ~/visual_ws/src
$ cd ~/visual_ws/
$ catkin_make
```

http://wiki.ros.org/catkin/Tutorials/create_a_workspace

2. Create a package

```
$ cd ~/visual_ws/src
$ catkin_create_pkg code_qrcode std_msgs rospy roscpp
```

http://wiki.ros.org/ROS/Tutorials/CreatingPackage

3. Put the three ROS nodes in a folder called *script* in *~/visual_ws/src/code_qrcode*

4. Add code to the end of *CMakeLists.txt* in ~/visual_ws/src/code_qrcode. The text:

```
catkin_install_python(PROGRAMS script/feature_scanning4.py
script/det_movement3.py script/distance_scanning2.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

5. Make catkin workspace.

```
$ cd ..
$ catkin_make
```

6. Adjust *feature_scanning4.py* to the current situation.
   - Line 37: adjust to frame rate of camera
   - Line 45: adjust to camera input. src=0: intern webcam; src=1 or 2: extern webcam
   - Line 56: Set field of view (FoV) to value of camera

7. Adjust *distance scanning2.py* to the current situation.
   - Line 39: adjust value of the real size of the printed qr-code in cm

8. Install pyzbar for ROS in new terminal.

```
$ sudo apt-get install libzbar0
```

9. Close all terminals

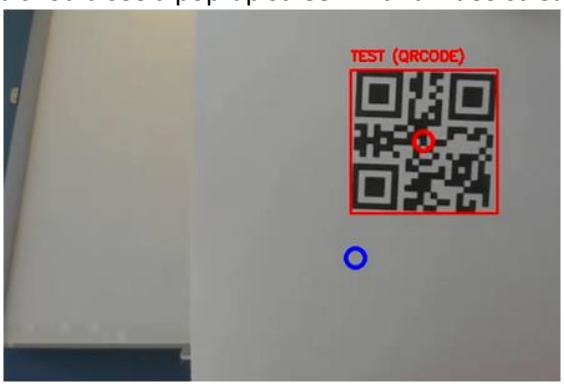
10. Start ROS in a new terminal

```
$ roscore
```

11. Open a new terminal and start *feature_scanning4.py.*

```
$ cd visual_ws/
$ source ./devel/setup.bash
$ catkin_make
$ rosrun code_qrcode feature_scanning4.py
```

Now you should see a pop-up screen with a video stream and a blue circle



12. Open a new terminal and start *distance_scanning2.py*

```
$ cd visual_ws/
$ source ./devel/setup.bash
$ catkin_make
$ rosrun code_qrcode distance_scanning2.py
```

13. Open a new terminal and start *det_movement3.py*

```
$ cd visual_ws/
$ source ./devel/setup.bash
$ catkin_make
$ rosrun code_qrcode det_movememt3.py
```

Now all the ROS nodes that are needed are running.

14. Hold the qr-code in front of the camera.

15. Listen to the channel by opening a new terminal

```
$ rostopic echo /ch_mov
```

2.3 Recommendations

A recommendation would be to replace the qr-code with an actual crack and a distance scanner. So there can be a vision-based recognition program. An option here would be to work together with Stenden. We already had setup contact with NHL Stenden (Maya Aghaei) where they are specialised in recognition in combination with machine learning. But a start with a simple vision code would be already a good start.

When you are not using a qr-code anymore there must be a distance scanner to determine the distance from the object. This can be a simple laser or a lidar.

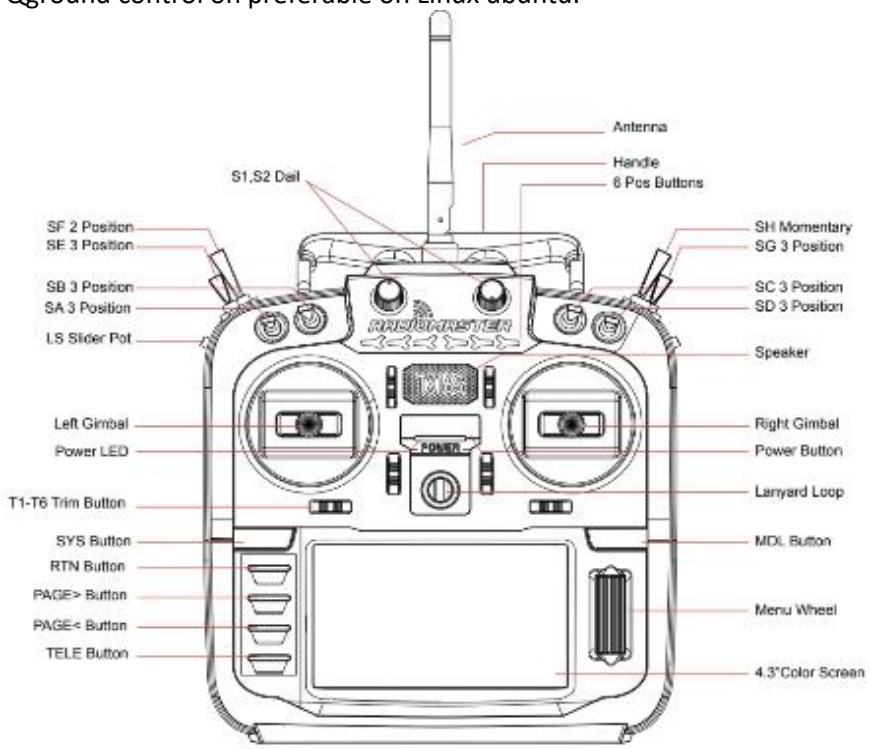
A recommendation could be to check the offsets determined by the qr-code, but because this offset can be relative it isn't really necessary to have the measurements precisely correct.

3. Controller settings and px4 parameters setup

To fully control the drone in this project. The drone needs to be prepared to handle all the control inputs of the user and software. This part is about the setup to fly the drone manually and the preparations to switch the drone into offboard mode.

Requirements

- install Qground control on preferable on Linux ubuntu.



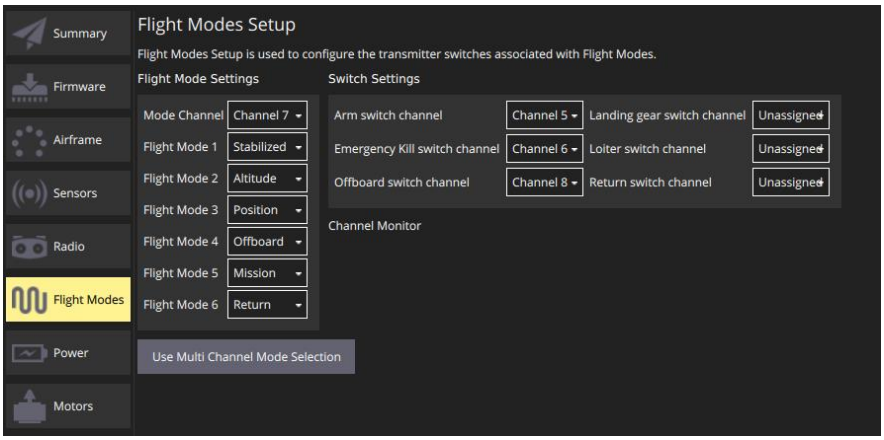
To connect the RC to the drone. See the Installation and setup of a multirotor with Pixhawk running PX4 firmware manual. If this RC is already connected and installed for this drone. the refed manual does not apply for u.

The functions of all the buttons of the controller for this project:

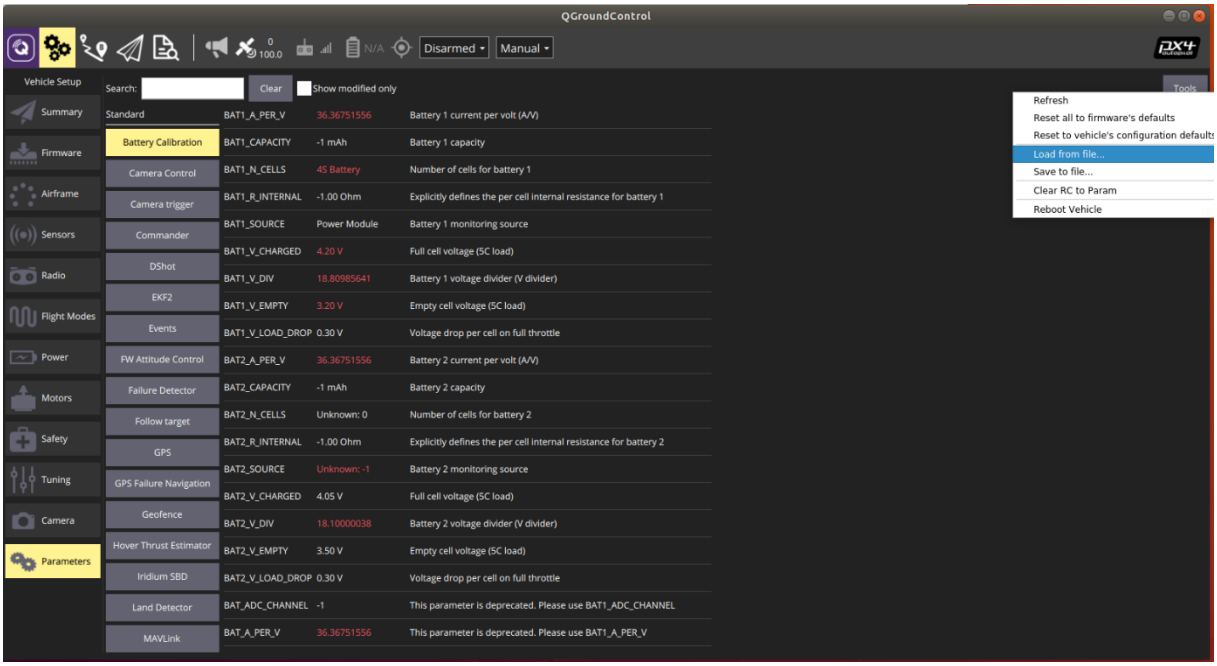
Button	Function
SF 2 position	Arm the drone
SH momentary	Kill switch (caution!! Before you let go of this button, first switch off the arm and offboard button. Before release the kill switch)
SG 3 position	Arm offboard switch
Button 4	Arm offboard switch

The standard procedure to switch a mode is to use the button 4. But because this is a test setup the SG 3 position button does the same thing and has a benefit that it is closer to the kill switch.

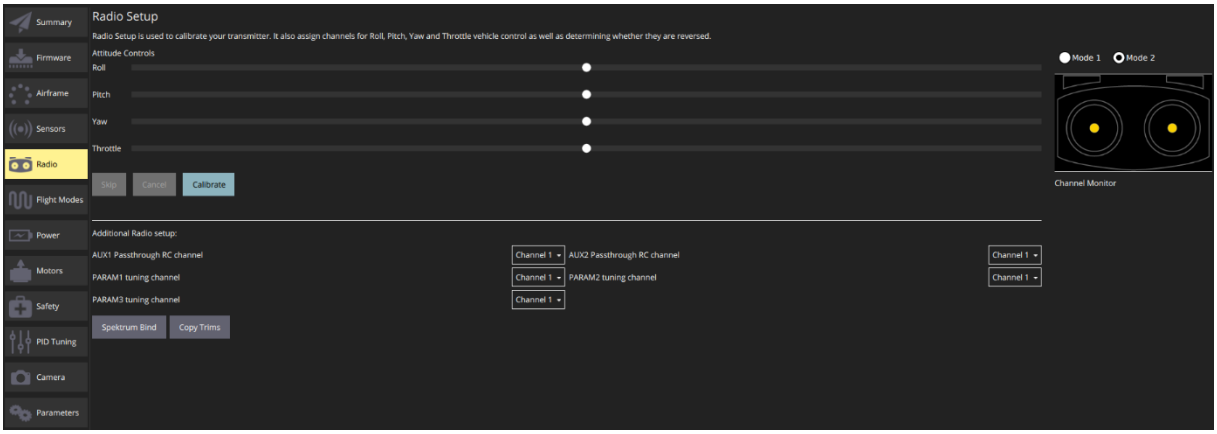
To chance the button functions for the RC. Go to QGroundcontrol and connect the drone with a normal USB cable. Not the long one! After loading go to settings and click **flight modes**. Chance all the settings until that it has the same as the picture below.



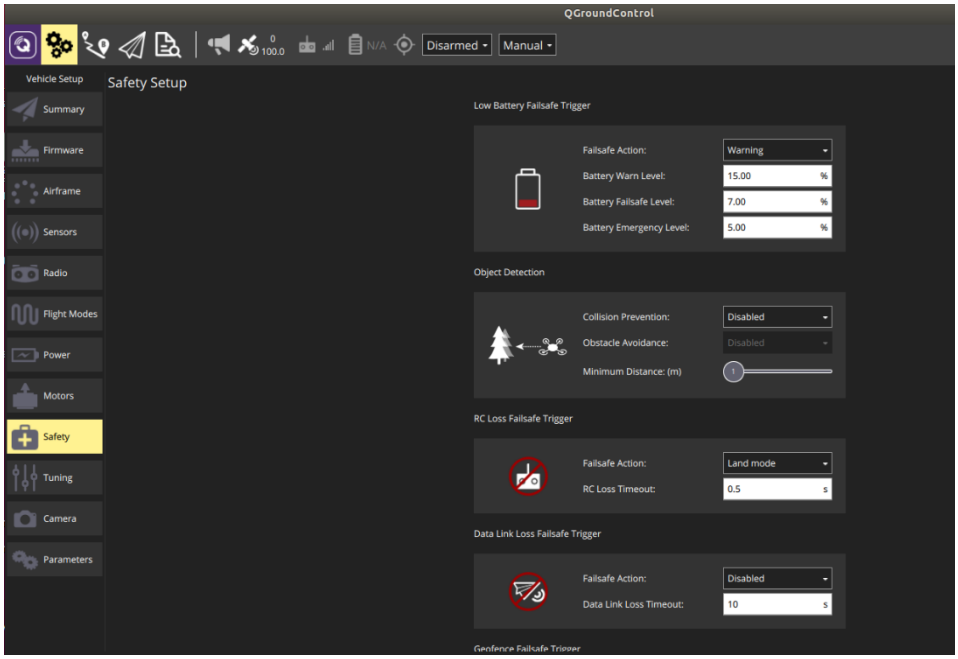
To chance the parameters, we used for or project. Go to **parameters** and click on **tools** right above the corner. Click on **load from file**. And select the **parameters Visual servoing UAV.params**. See picture below:

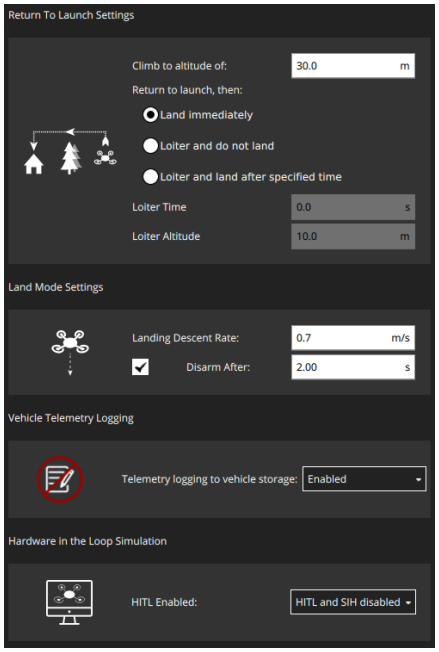


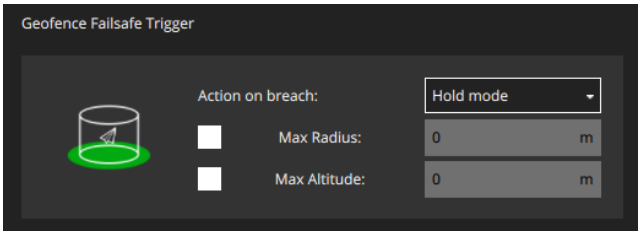
Now go to **radio** and check if the values are the same of the picture below:



Now go to **safety** and check if the values are the same of the picture below:







4. Vision based flight controller

Once the drone has manually been flown to the area where the feature is that it needs to detect, it initially needs to be able to switch to its offboard mode and start hovering in front of this feature. In order to accomplish this, data from the vision program is obtained and used to determine the position of the drone in front of the feature or the case of this project, a qr-code. This data in turn is used to update the local position of the drone. On top of that, for the drone to know where it is supposed to fly towards, a setpoint is also given to the drone.

4.1 Node parameters /input output

To accomplish the updating of the local position and giving the drone a setpoint to fly towards, another ROS node has been written called: “vision_controller_node.cpp”.

This node is subscribed to 2 channels. A “mavros/state” channel which comes from the drone and publishes a state object filled with values describing the current state of the drone, which is used to check for a connection and other diagnostic data. But the focus lies on the “ch_mov” channel, which is the output of the vision program and publishes a PoseStamped object which is filled with x, y and z-offset values with respect to the qr-code. These x, y and z-offset values are in turn used to update the local position of the drone. More detail in the working of this will come in the next section.

Next to the 2 subscribed channels, the node also publishes towards 2 channels. The “/mavros/vision_pose/pose” channel, which publishes a PoseStamped object filled with x, y, z and current time values which are used to update the local position of the drone. The second channel it publishes towards is the “/mavros/setpoint_raw/local” channel, which publishes a PositionTarget object filled with x, y, z and current time values, which is used in order to tell the drone towards which coordinate it should fly with respect to its local position. Both channels are publishing their data for the internal controller of the drone to use.

The 2 publishing channels will have objects that are initially pre-filled and are put in a while loop in such a way that they will always publish values. As such, this node will start publishing their values immediately when starting this node. For the “/mavros/vision_pose/pose” channel, this means it will initially be sending x,y and z values of 0 until values from the “ch_mov” are received and update the values for this channel. For the "/mavros/setpoint_raw/local" channel, this means that the drone will always have the goal to fly towards coordinate (0,0,0) with respect to its current local position.

4.2 General workings of the vision control

In this part of this report, a little more background information is given into the workings of the vision_controller_node to give the reader a better understanding of what it is that it is actually doing.

First off, the data from the “ch_mov” channel is dissected in their x, y and z values and put into a new PoseStamped object where the x, y and z values gotten from the ch_mov channel properly align with the x, y and z values in the frame the drone uses them. On top of that, for the “pose.pose.position.x” depth value of the new PoseStamped object, an “object_distance” value has been added in order for the local position coordinate (0,0,0) to be the “object_distance” meters away from the qr_code. Once this is done this new PoseStamped object gets the current ROS internal time added to its header and published on the “/mavros/vision_pose/pose” channel which is received by the drone in order to update the local position of the drone.

Whilst the node is publishing towards the "/mavros/vision_pose/pose" channel to update the local position of the drone, it is not the only factor which is used by the drone for it to determine its eventual local position. The drone internally uses a Kalman-filter which uses multiple inputs to make a prediction on what the local position of the drone is. These inputs can be given a certain weighing in how much each of these inputs should influence the prediction of the local position and this results in the actual local position which is known to the drone. In this case, the "/mavros/vision_pose/pose" channel influences the local position, but the drone will also use its internal sensors for example the accelerometers and currently known local position.

For this reason, it is recommended that this node starts running a while before switching to offboard mode and let the drone hover in front of the qr-code whilst detecting the qr-code for a few seconds. This will help the "mavros/vision_pose/pose" influence the drones' internal local position value to align with the sought after (0,0,0) position in front of the qr-code.

4.3 Instructions

Assuming that the instructions have been used to create the vision part of a new workspace, the next step to have this script working is that MAVROS needs to be added to this workspace.

1. Note that to have MAVROS installed, you will initially need to install wstool as well as catkin-tools which can be found with the following links:

(wstool) http://wiki.ros.org/wstool

(catkin tools) https://catkin-tools.readthedocs.io/en/latest/

Once this is done instead of building the workspace with "catkin_make", the workspace is now built with "catkin build".

2. One of the two following links can now be used as sources to get the proper version of MAVROS installed. Keep in mind that instead of the "`~/catkin_ws`", "`~/visual_ws/`" should be used in these manuals.

(from the header: "installation" onwards)

1. https://github.com/mavlink/mavros/tree/master/mavros#installation

(For this, mind that the ROS version used here should be compatible for other ROS versions)

2. https://docs.px4.io/master/en/ros/mavros_installation.html

3a. At this point you should be able to make a new package for the vision controller through

```
$ cd ~/visual_ws/src
$ catkin_create_pkg offboard_control std_msgs rospy roscpp
```

And put the ROS node "vision_controller_node.cpp" in a new folder you create called *src* in "*~/visual_ws/src/*offboard_control"

4. From this point on you should add the following at the end of the *CMakeLists.txt* in "*~/visual_ws/src/*offboard_control".

```
add_executable(offboard_control src/offboard_control.cpp)
target_link_libraries(offboard_control ${catkin_LIBRARIES})
```

5. Finally you should go back to the root of your workspace and build it again with catkin

```
$ cd ~/visual_ws
$ catkin build
```

6. Once this is done you can check if it works by trying to connect to the drone with usb connection and run:

```
$ roslaunch mavros px4.launch fcu_url:="/dev/ttyACM0:57600"
```

Note: your drone might have a different fcu_url and it should be adjusted accordingly

If succesful, one of the lines that should appear in the terminal contains:

CON: Got HEARTBEAT

If this is seen, a proper connection with the drone has been established.

The next step is to then open a second terminal window and run:

```
$ cd ~/visual_ws
$ rosrun offboard_control vision_controller_node
```

By itself this node won't show anything, but you can then open another terminal window and run:

```
$ cd ~/visual_ws
$ rostopic echo /mavros/vision_pose/pose
```

Which is a channel the script should be publishing data towards. If you edit a starting value for this data in the script, it should be reflected in the response from the echo.

4.4 Recommendations

The first and main recommendation for this node is that it is only used for a stationary object. The way this code works now is such, that when the object that is being tracked moves, the (0,0,0) local position and the rest of it frame moves with it due to the vision_pose inputs. This intervenes with the outputs of the drones' internal sensors and currently known local position and causes instability for the drone when these values go through the Kalman filter.

A solution for this might lie in using the offsets gotten from the vision code to update the setpoints instead of the vision_pose and use only the internal sensors to update the local_position. However, this could be prone to environmental effects and errors within the sensors.

The second recommendation would be to look at the Kalman filter and look into the weight of each input influencing the local position of the drone. With a good vision program, it could be beneficial to let the vision_pose influence the local position of the drone to a higher degree in comparison to the internal values and sensors.

5. What to do next.

From this point on quite a lot of avenues of research can be looked into in order to further develop this project.

- Improved vision, programs, (more accurate, detecting multiple features, switch from qr-code to feature tracking)
- Better hovering, (use a Kalman filter and adjusting the weight of local position influencing inputs) create a controller for the drone to hover in a more stable way (PID)
- Self-movement, (input values for set-points in order to let the drone follow a certain pattern (this does require that the drone has features to track))
- Environment that moves, (use internal sensors to determine the local position whilst the vision offsets are used as setpoints) OR (Use a vision program which tracks a stationary target to determine the local position whilst a second vision program tracks the moving object which is then used to send set-points).