TUHH
Hamburg
University of
Technology

Max Lange, Luca Mülke, Laurent Rodemerk, Tobias Kamrath,
Paul Gläser, Joao Clemente, Joe Kunert, Ruben Kuhlmann,
David Verastgui Stiglich, Yosef Fares, Ghislain Nkamdjin Njike,
Julius von Sulecki, Lukas Mackel, Christian Sühl

# Implementation of an Autonomous Sailing Algorithm for a Model Boat

Institute of Medical Technology
and Intelligent Systems
Building E | 21073 Hamburg
www.tuhh.de/mtec

Institute of Medical Technology and Intelligent Systems

A project paper written at the Institute of Medical Technology
and Intelligent Systems and submitted in partial fulfillment of the requirements for the degree Master of Science.

Author: Max Lange, Luca Mülke, Laurent Rodemerk, Tobias Kamrath,
Paul Gläser, Joao Clemente, Joe Kunert, Ruben Kuhlmann,
David Verastgui Stiglich, Yosef Fares, Ghislain Nkamdjin Njike,
Julius von Sulecki, Lukas Mackel, Christian Sühl

Title: Implementation of an Autonomous Sailing Algorithm for a Model Boat

Date: September 6, 2023

Supervisors: Konrad Reuther (BSc.)
Alexander Schlaefer (Dr.-Ing.)

Referees: Prof. Dr.-Ing. Alexander Schlaefer

# Contents

# List of Figures

# 1 Introduction

With the increasing digitization in today's globalized world, the use of robotics in various areas and disciplines of everyday life is a broad field of research. The medical part occupies a large area, where, for example, the so-called "Cyberknife" [7] complex surgical precise removals of tumors on the brain or other critical regions of the body can be performed. This requires a large number of different sensors, adapted to the respective situation or application of the robot. In 2008, the so-called "World Robotic Sailing Championship and International Robotics Sailing Conference" [5] (WRSC/IRSC) was formed for the first time to map robotics research to the topic of the discipline of sailing. A wide variety of universities and technical colleges took part in the conferences with their self-built model sailing boats, which were up to 4 m long. The conference offers the opportunity, in addition to the regatta idea, to discuss the scientific problems involved in sailing and the development of autonomous sailing boats at an international scientific level.[5] On the basis of this competition/conference, groups of students from technical courses at the TU Hamburg were asked to use an already existing constructed boat, designed by Professor Schlaefer (Medical Technology Institute of the TU Hamburg), as part of the "Intelligent Systems Project" Module, using techniques of image processing, computer science and control engineering, for the resumption of this competition. The student group from the last summer semester dealt with the basics of image recognition. As a student group of students in the "Master of Science Computer Science Engineering" course, we had the goal for the summer semester 2023 of implementing the autonomous sailing process on the already existing and prepared hardware. Our Teamtable is listed in 1.1. In this report we want to give an insight into our development and thus enable the further development of our components by future student groups. In the beginning, the hardware architecture and the composition of the project or the development process are examined in more detail and then the individual implemented components and algorithms are discussed. Finally, this paper completes with an evaluation of the results obtained and gives an outlook on the continuation of the project.

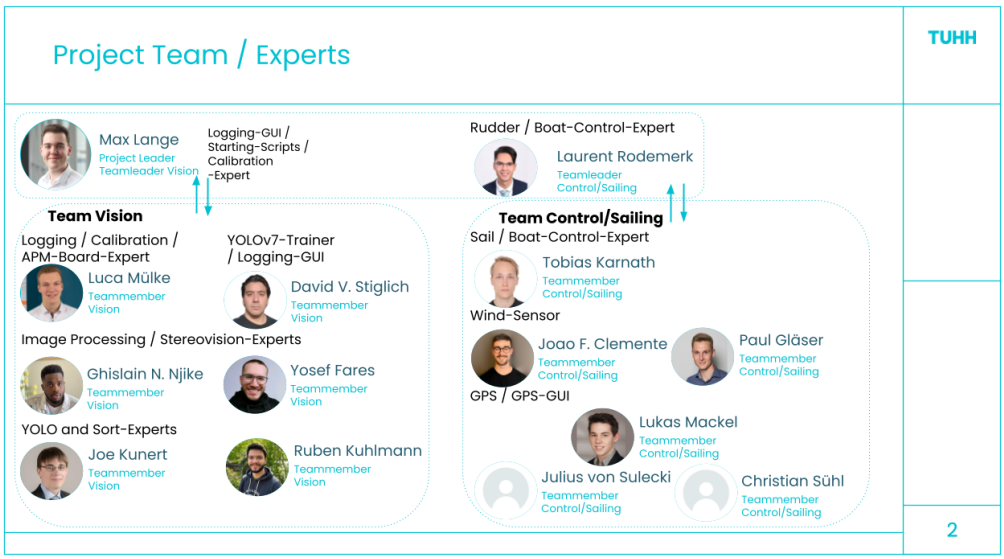


Fig. 1.1: Teamtable with Team Vision and Team Control SS23

## 1.1 Hardware Architecture

In order to be able to go into the system architecture of the sailing boat, the hardware architecture of the robotic system is explained in advance below. As seen in figure 1.2, the sailboat is equipped with: a 3D printed rudder, a seal, a stereo vision camera (mounted on a rotatable gimbel consisting of servos), a wind direction sensor, a GPS sensor and a wind speed sensor. The mechanical components (camera mounts, rudders, sails) are controlled by servos controlled by an Autopilot Mega (APM) board. The image processing or object recognition as well as the sailing process are taken over in the main part by a small, powerful computer (JETSON Nano from NVIDIA). This is specialized in use for the purpose of image processing. As can be seen in figure 1.2, the JETSON-Nano is connected to the APM board via a USB interface. Assignment of the Pins can be seen in figure 1.2.

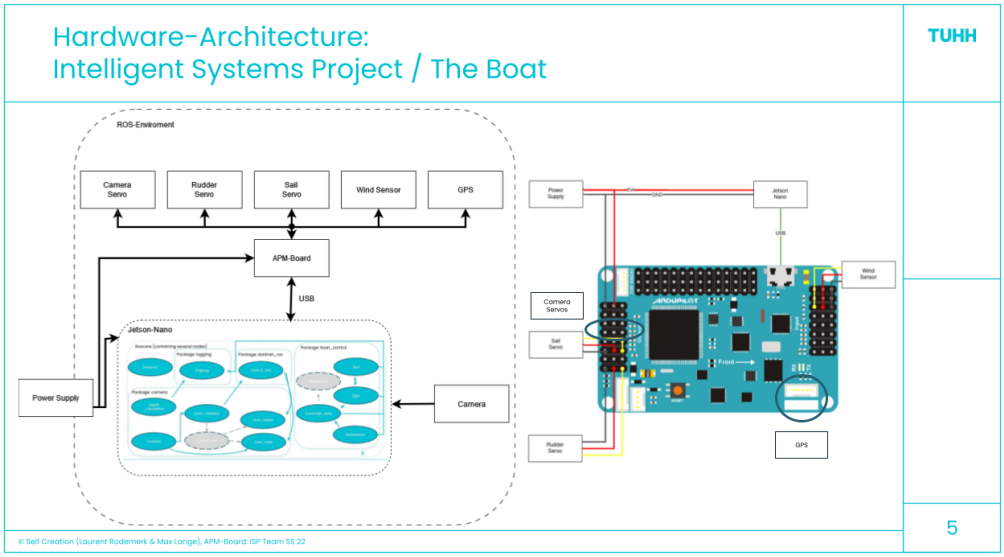


Fig. 1.2: Hardware Architecture based on figure by group of last year

## 1.2 System Architecture

We use the robot operating system (ROS) [9] to control the APM board and evaluate the various sensors. The operating system supports C++ and Python codes that can run in so-called nodes. Nodes consist of publishers and subscribers, with which the nodes can send data and messages back and forth in different ways or react to certain values. Centrally for the communication between the APM board and the JETSON Nano, we use the MavROS package [8]. Figure 1.3 shows the system architecture of the various ROS nodes. The architecture can be divided into: the camera package, the logging package, the boat_control package. The Boat_Control package takes over the function of autonomous sailing in that it controls the servos (in the controller node) and reacts to the respective inputs from the wind sensor and GPS. The camera package calculates depths of objects in front of the camera (stereo vision - not functional now) and the detection and tracking of objects (using Yolov3 or YOLOv7). The logging node stores data in JSON-like format of current APM values and messages about process steps of the individual nodes. There are also scripts for accomplishing the calibration (calibration_script can easily be executed as a Python script and then follow the instructions in the console) and quick_calibrate, which automatically describes the APM after calibration.

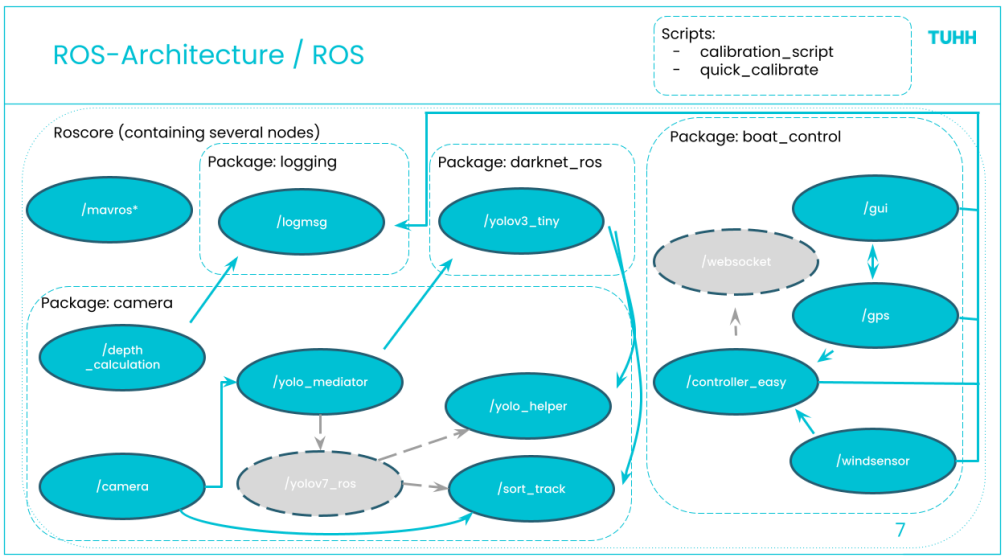


Fig. 1.3: System Architecture of different ROS-Nodes and their connection

# 2 Working Process and Timeline

Below is the timeline in figure 2.1 and 2.2 for a more detailed overview of the course of the project .

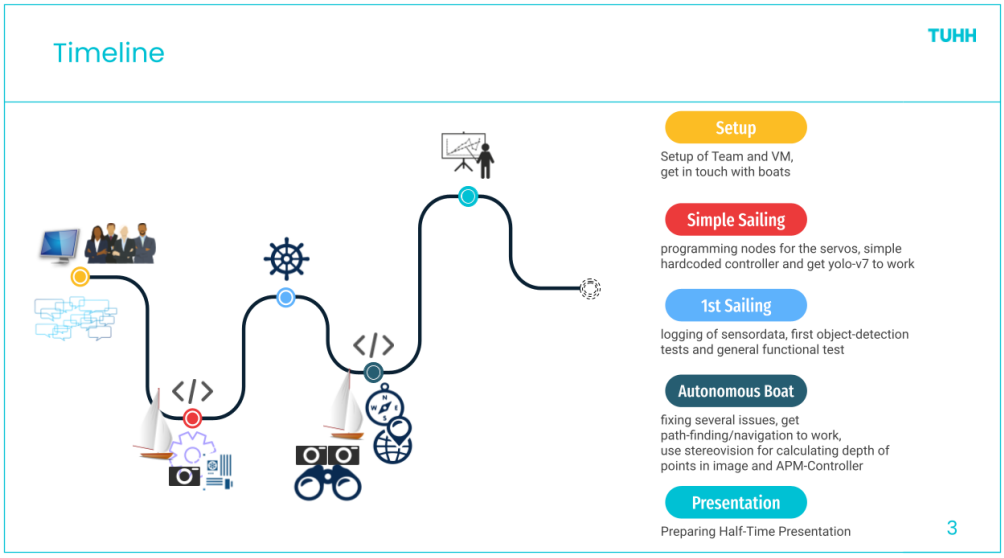


Fig. 2.1: System Architecture of different ROS-Nodes and their connection

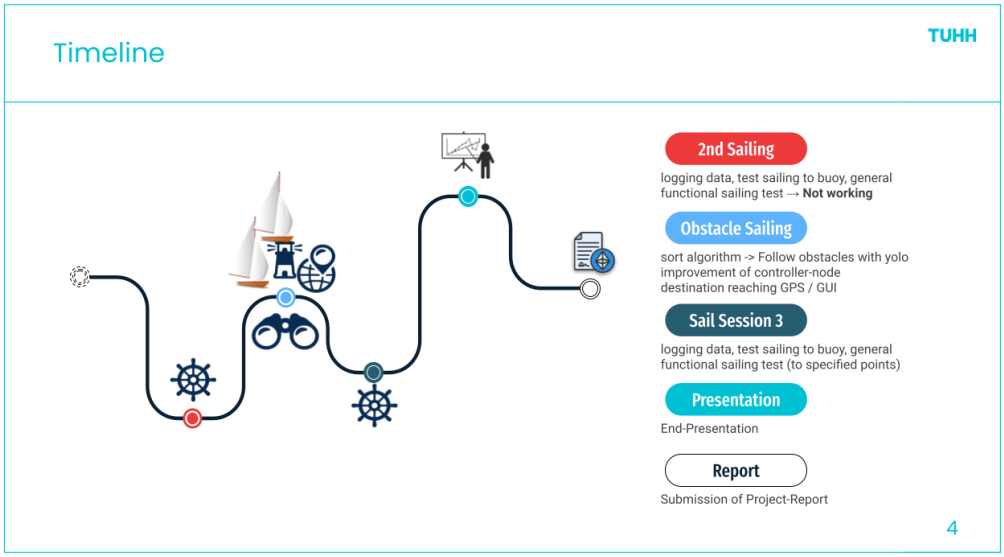


Fig. 2.2: System Architecture of different ROS-Nodes and their connection

# 3 System Components

## 3.1 Controller Node

In order to conduct all main components we introduced a controller node. This node evaluates incoming sensor data and determines which course to take or which manoeuvre to perform. Possible manoeuvres are as follows:

- **Start Sailing**:
  After booting the firmware, the controller node is in state *Start Sailing*. Here, the sail is set and for a period of 8 seconds the vessel sails straightforward in order to pick up speed.
- **Follow Course**:
  The vessel is sailing straight ahead towards a given GPS point.
- **Jibe**:
  A jibe is performed.
- **Tack**:
  A tack is performed.
- **Beating**:
  The vessel is cruising against the wind.
- **Do Nothing**:
  The controller does not make any changes. Sensor data is contradictory or misleading. This state is used for error handling.

### 3.1.1 Proposed Algorithm

The controller node's algorithm consists of two parts, namely evaluating sensor data resulting in a decision of the next manoeuvre, and the actual execution of that manoeuvre. The algorithm is executed whenever the GPS node or the wind sensor node publishes new data.

#### Manoeuvre Decision

Since wind is the dominant force in sailing, sensor data must be converted in respect to the wind first. As the wind sensor is fixed on top of the sail and therefore also moving with it, its data has to be converted in respect to the bow by using the relation

$$\theta_{boat} := \theta - \text{SAIL ANGLE}$$

where $\theta$ is the raw data given by the wind sensor, i.e. in respect to the sail, $\theta_{\text{boat}}$ the angle of the wind in respect to the bow and SAIL ANGLE the current position of the sail. Furthermore we gain the direction to the next GPS target in respect to the wind by

$$\phi_{\text{wind}} := \begin{cases} 360° - \theta_{boat} + \phi_{\text{boat}} & : \text{wind comes from port side} \\ \theta_{\text{boat}} - \phi_{\text{boat}} & : \text{otherwise} \end{cases}$$

where $\phi_{\text{wind}}$ is the angle to the next GPS target in respect to the wind direction, $\phi_{\text{boat}}$ the angle to the next GPS target in respect to the bow. Note that degrees are defined positive clockwise.

The manoeuvre is then determined through a case differentiation involving $\phi_{\text{wind}}$. The case differentiation is depicted in Fig. 3.1.

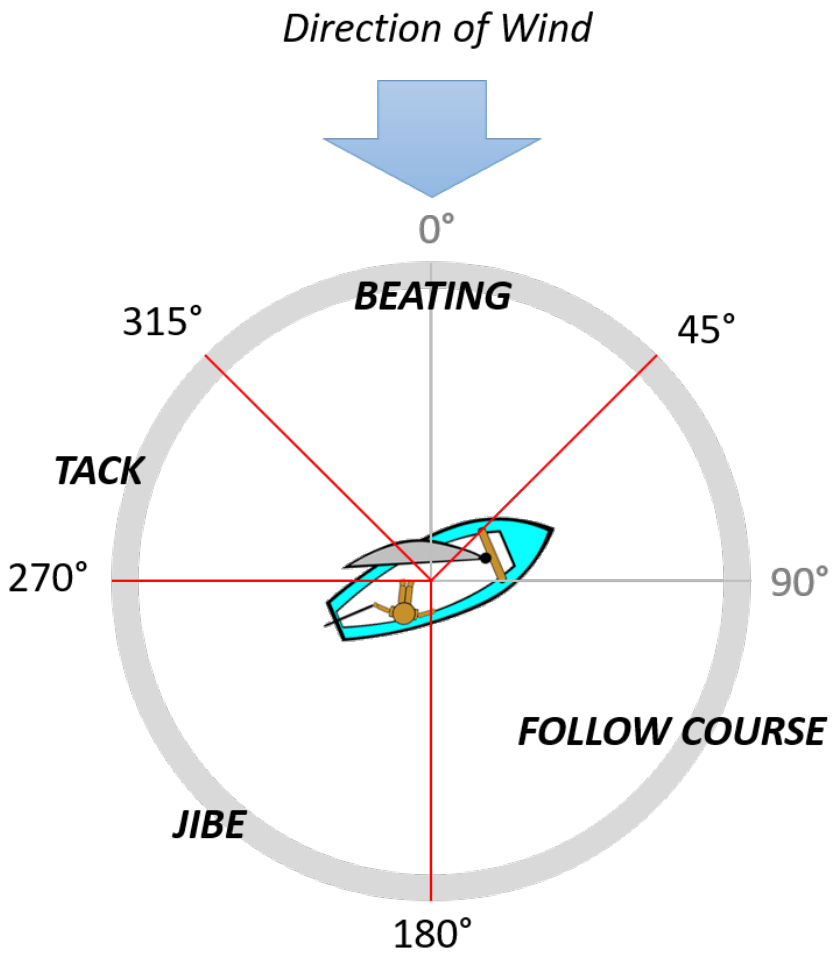


Fig. 3.1: Manoeuvre decision when wind comes from port side

### Execution of the Manoeuvre

Since direct control of the servos is already implemented on the APM, determining target angles for the sail and apparent wind is necessary. These target values are then written

to the APM.
In order to pick up speed in state *Start Sailing*, we set

$$\text{TARGET APPARENT} := \begin{cases} 270° & \text{: wind comes from port side} \\ 90° & \text{: otherwise} \end{cases}$$

and

$$\text{SAIL ANGLE} := \begin{cases} 60° & \text{: wind comes from port side} \\ -60° & \text{: otherwise} \end{cases}$$

respectively. Values are retrieved empirically and kept for a period of 8 sec.

While being in state *Follow Course* the target point can be reached by just adjusting rudder and sail. The goal is to keep the target_apparent constant as long as possible. This ensures that there is always good wind in the sail and the boat moves. For steering, the sail_angle is adjusted and then the apm controls the rudder to restore the original target_apparent. Only if the sail cannot be opened enoughed for steering, we have to adjust the target_apparent.
Current values can be updated according to the target's point angle $\phi_{\text{boat}}$ given by the GPS module. Hence,

$$\text{SAIL ANGLE} := \begin{cases} min(50°, \text{SAIL ANGLE} + \phi_{\text{boat}}) & \text{: wind comes from port side} \\ max(-50°, \text{SAIL ANGLE} - \phi_{\text{boat}}) & \text{: otherwise} \end{cases}$$

and if $\phi_{\text{boat}} + current\_sail\_angle \notin [-50°, 50°]$, i.e. a downwind course can be sailed, then

$$\text{TARGET APPARENT} := \begin{cases} 270° + 30° - (\phi_{\text{boat}} - (50°) - current\_sail\_angle) & \text{: wind comes from p.} \\ 90° - 30° - (\phi_{\text{boat}} + (50°) - current\_sail\_angle)) & \text{: otherwise} \end{cases}$$

Otherwise we get the sail as close as possible to 30°, i.e.

$$\text{TARGET APPARENT} := \begin{cases} 270° + 30° & \text{: wind comes from port side} \\ 90° - 30° & \text{: otherwise} \end{cases}$$

Note that ±60° are hardware boundaries of the sail's servo and ±50° is a boundary for the follow_course manoeuvre.

A *Jibe* is performed within three steps: Firstly bearing away by setting

$$\text{TARGET APPARENT} := \begin{cases} 245° & \text{: wind comes from port side} \\ 115° & \text{: otherwise} \end{cases}$$

and

$$\text{SAIL ANGLE} := \begin{cases} 65° & \text{: wind comes from port side} \\ -65° & \text{: otherwise} \end{cases}$$

After a waiting time of 2 sec we turn to the other side of the wind by heading downwind using

$$\text{TARGET APPARENT} := \begin{cases} 105° & : \text{wind came from port side initially} \\ 225° & : \text{otherwise} \end{cases}$$

and

$$\text{SAIL ANGLE} := \begin{cases} -30° & : \text{wind came from port side initially} \\ 30° & : \text{otherwise} \end{cases}$$

Last step is luffing which is done by the *Follow Course* manoeuvre.

As for the jibe a *Tack* is performed within four steps: First we steer into the wind by setting

$$\text{TARGET APPARENT} := \begin{cases} 345° & : \text{wind comes from port side} \\ 15° & : \text{otherwise} \end{cases}$$

and

$$\text{SAIL ANGLE} := \begin{cases} 15° & : \text{wind comes from port side} \\ -15° & : \text{otherwise} \end{cases}$$

respectively. After a period of 1 sec we turn using

$$\text{TARGET APPARENT} := \begin{cases} 45° & : \text{wind comes from port side} \\ 315° & : \text{otherwise} \end{cases}$$

and

$$\text{SAIL ANGLE} := \begin{cases} -15° & : \text{wind comes from port side} \\ 15° & : \text{otherwise} \end{cases}$$

Finally we pick up speed again after a waiting time of 1 sec by sailing a half-wind course. Hence,

$$\text{TARGET APPARENT} := \begin{cases} 90° & : \text{wind came from port side initially} \\ 270° & : \text{otherwise} \end{cases}$$

and

$$\text{SAIL ANGLE} := \begin{cases} -30° & : \text{wind came from port side initially} \\ 30° & : \text{otherwise} \end{cases}$$

After 3 sec further turning is done by *Follow Course*.

In order to beat, corresponding turning points have to predefined during mission planning. In state *Beating* we sail close-hauled by setting

$$\text{TARGET APPARENT} := \begin{cases} 345° & : \text{wind comes from port side} \\ 15° & : \text{otherwise} \end{cases}$$

and

$$\text{SAIL ANGLE} := \begin{cases} 30° & : \text{wind comes from port side} \\ -30° & : \text{otherwise} \end{cases}$$

until the GPS target point is in the range where a tack can be performed according to 3.1.1. This is repeated for all predefined turning points.

### 3.1.2 Evaluation

Due to the short testing times and various problems encountered during testing, a concrete statement about the performance of the controller node is difficult to make. The basic concept of the controller was implemented and worked in theory on campus. This means that the controller initiated the correct manoeuvres for corresponding values entered via the terminal and steered the sail accordingly. Having this fundamental basic structure is a big milestone for sailing autonomously.

However, on the water, it could not be determined that the boat was being steered correctly. The boat did not go straight or where it was supposed to go. In addition, the boat turned a lot on the spot and the sail changed position far too often. Once the boat got going, it was apparent that it could at least go straight in the right direction. However, other manoeuvres were attempted, but the desired result was not achieved. Neither a tack nor a jybe led to the desired result.

Possible reasons for the controller's performance could be:

Firstly, the controller was thinking too often, meaning that every time a new wind value or heading of the compass was received, a new manoeuvre was calculated. This resulted in multiple instructions being given every second, which ultimately caused an excessive number of sail movements. Additionally, every deflection or wobble of the wind sensor was reacted to, and noise had a great influence on the controller and the sailing. Two points could immediately improve the performance: a "slower" thinking of the controller, which has already been implemented with a sleep(), and a more reliable input signal of the wind, which has already been smoothed. In conclusion, the whole structure has to be more robust.

During the last sailing session, there was unfortunately very little wind. As a result, the boat drifted more than sailed, making it impossible to carry out intended maneuvers. Instead, the boat kept turning, causing the sail and rudder to move unnecessarily without significant progress. Due to light wind, frequent turning of the sail accelerated the wind sensor and often resulted in false readings, exacerbating the problem. Additionally, we lacked experience to set the angles (sail to wind and sail to boat) correctly during individual maneuvers. During the last part of the sailing session, the wind sensor was not calibrated correctly. Ideally, 0° should be in front, in the direction of the sail; however, in our case, this value was shifted by 200°, rendering the controller useless.

In summary we did not saw the full potential of what the controller is capable of. But that doesn't mean that everything is correct and the manoeuvres will match the intended behaviour. We have a basic structure for the controller to identify and execute the correct maneuver. However, further work is needed to refine the concrete implementation. Specifically, the angles for the individual maneuvers must be adjusted based on experience and testing, and the wind speed should be taken into account. There is also potential for improvement in the robustness of the manoeuvre choice and handling noisy input data.

### 3.1.3 Future Work

Currently, only data from the wind sensor and the GPS module is used for manoeuvre decision. But in order to achieve full autonomy visual feedback about the environment might also be considered. Therefore the controller node's algorithm might be extended by obstacle detection and the implementation of traffic rules in order to operate in environments with multiple vessels.

## 3.2 Vision

### 3.2.1 PyEnv

PyEnv lets you easily switch between multiple versions of Python. This is needed, because the boat has a lot of different dependencies. Therefore different Python versions are used. With PyEnv it is possible to install different package versions on one computer. This is especially useful when working with the camera and its systems. Since machine learning is such a fast developing field.
Before installing or upgrading a new python package, make sure to verify that you are using the correct PyEnv environment. Otherwise a new or updated python package, for example numpy could lead to fatal crashes. Which virtual python environment is used is declared in the shebang of the different subsystems. Important pyenv console commands are the following:

```
pyenv install
pyenv versions
pyenv global
pyenv help
```

PyEnv install is used to create new virtual environments. This is handy if you want to test new systems with a lot of python package dependencies. PyEnv versions shows all currently installed versions and the active version indicated by a star(*). Finally with PyEnv global you can change which version should be the default system python version.

### 3.2.2 YOLOv7

YOLO is an algorithm that provides real-time object detection. It is popular because of its speed and accuracy. As implied by its name this is the 7th version of said algorithm which provides better performance in the accuracy and numbers of detected objects when compared to the third version which was already implemented at the start of this project. Our implementation is based on the repository found at [2]. This repository makes use of the official YOLOv7 implementation which can be found at [1]

YOLO subscribes to /camera/rgb/image_raw and performs object detection on the received images. Once computation is finished the results are published to /yolov7/yolov7 as values and to /yolov7/yolov7/visualization as images this can be seen in 3.2. An example of an image can be seen in figure 3.3 where the performance of YOLOv7 is shown in the fact that even objects that are far away enjoy good and clear detection. (The discoloration is due to an early version of the YOLO helper 3.2.3 where the RGB-value change over the ros messages was not accounted for)

For our implementation, we make use of the YOLOv7-tiny model in order to mitigate performance issues.

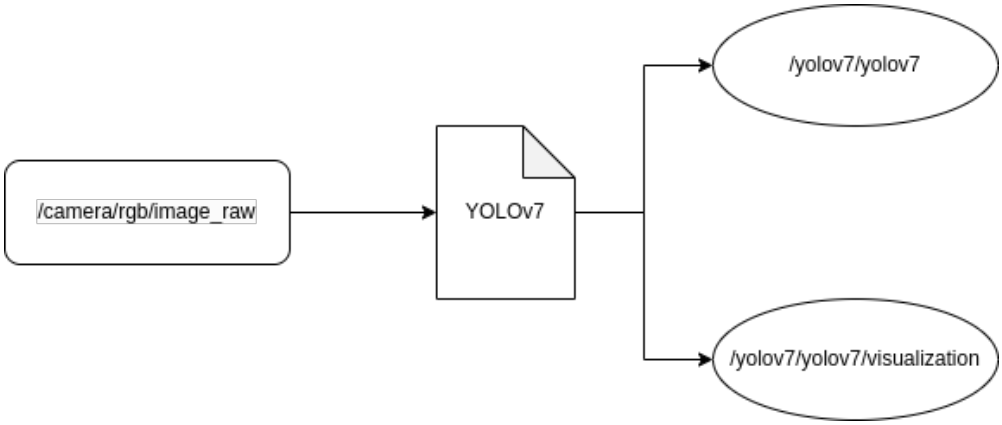


Fig. 3.2: Overview of YOLOv7 topics.

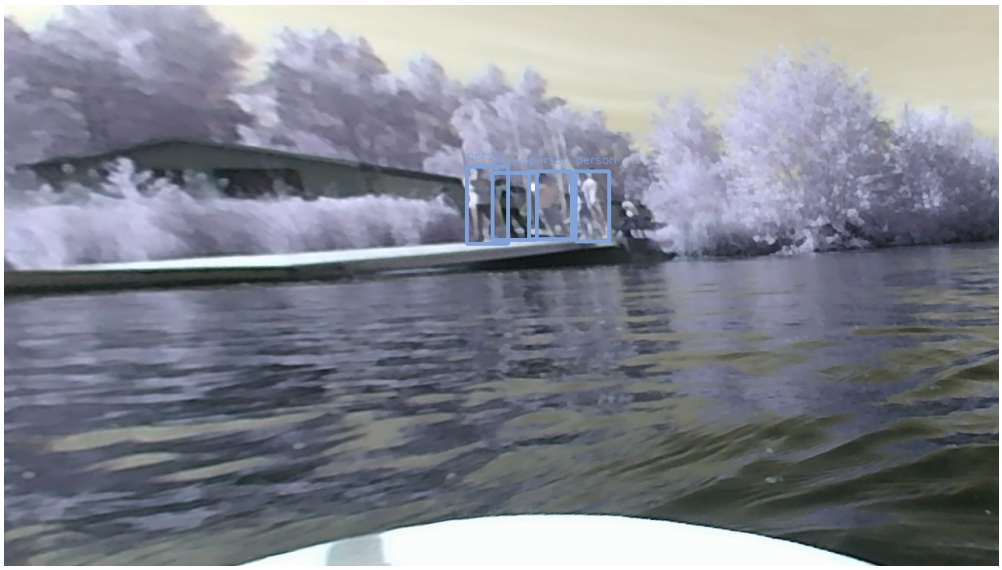

Fig. 3.3: Demonstration of object detection performance with YOLOv7 (2. sailing session).

**install:** For installation after the YOLOv7-ros package has been added to the ros src folder and system path one has to execute these commands one after another:

```
pyenv install 3.6.15
pyenv global 3.6.15
cd isp-2022/jetson_nano/catkin_ws/src/yolov7-ros
if done in vm change shebang of
detect_ros.py to correct pyenv folder
pip install numpy==1.19.4
pip install --upgrade
pip install -r requirements
```

It is important to use the files from the boat repository since modifications of the original repository were necessary to make it functional in our environment. Most notably the issue with cv_bridge to work with images over the ros messages. In many places, it is sufficient to replace this with numpify/msgify from the ros_numpy package. Furthermore, the launch file of the YOLOv7 package offers a comprehensive list of options such as input and output topics as well as the option to switch between CPU or GPU accelerated computation of objects. For GPU computation TensorFlow and CUDA has to be installed on the system.

To launch YOLO execute

```
roslaunch yolov7_ros yolov7.launch
```

**problems/future work:** The biggest problem for this YOLO implementation is the performance of the jetson-nano used to run the whole ROS environment. Specifically RAM usage is of concern since just the components used to provide YOLO with pictures and YOLO itself push the ram usage in our tests into making use of the swap quite heavily ( 500Mb-1Gb). These tests were performed with a connected screen so there might be some performance gain if no screen is connected/bothering the GPU, however since components not directly related to YOLO and sort_track (explained in 3.2.4) also need computing resources later sail sessions where done with the YOLOv3 installation to not risk crashes. As later explained sort and the yolov7_helper have been adapted such that usage of either YOLO version is possible. This problem needs further investigation into possible performance gains/hindrances as a more reliable detection of objects substantially increases the quality of things like object tracking or depth calculation.

Another problem is the detection of objects in two different pictures. Since the boat has a stereo camera it makes sense to actually detect the position of objects on both of the produced pictures for further processing. However, besides the performance demands, this also puts further demands on the YOLO implementation since a synchronization between the two pictures needs to take place. Currently whenever YOLO is done with computation on one picture it takes the newest picture produced as the next input. Since the camera however takes pictures much faster than YOLO can process them this behavior would lead to desynchronization in time between the two pictures. A start to the solution for this problem may be the fact that the amount of pictures distributed by the camera ros-node can be changed.

### 3.2.3 yolov7_helper:

The yolov7_helper is used to create logs of the output of YOLO under catkin_ws/src/yolov7_helper/log both the pictures and the values can be logged for both YOLO versions via their respective scripts. For this, the scripts simply subscribe to the relevant topics: pictures: /darknet_ros/detection_image /yolov7/yolov7/visualization values: /darknet_ros/bounding_boxes /yolov7/yolov7

**install:** This program works right out of the box

**problems/future work:** The logging of the values can be done in the general logging node and once sort_track is executed it performs its own logging of images.

### 3.2.4 Object Tracking Sort:

An important part of object detection is to keep track of the detected object. This means, that not only the position of our object is important but also if the detected object is the same object as previously detected. In both of our used Yolo versions this is not possible. We only get information on which object was detected and where it was detected. But to be able to follow the object with our camera we need to be able to give IDs to our detected objects as seen in figure 3.5. These IDs seen on top of the detected objects are created by the sort algorithm and are not lost if the object was not detected in a short period of time. This is needed because the positions of our detected objects are always changing due to our own boat moving but also because the detected object itself has a high chance of movement.
The sort algorithm is better described in [4]. It is designed for object tracking where only past and current frames are available and the method produces object identities instantly. On the boat are currently two different versions of this principle installed. The sort algorithm itself and an improved version called deep sort. The inner work of deep sort is explained in these papers [11] and [12]. It is recommended to use deep sort, since it has better results.
Usage: The interactions of the different ROS Topics are shown in Figure 3.4.

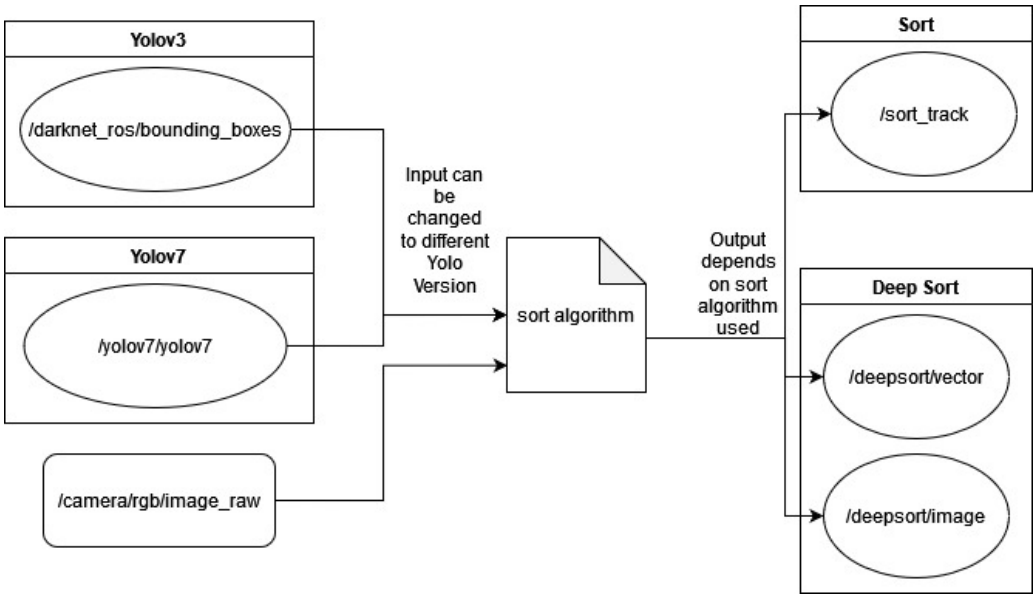


Fig. 3.4: Used topics of Sort Algorithm

To run sort use:

```
roslaunch sort_track sort.launch
```

To run deep sort use:

```
roslaunch sort_track sort_deep.launch
```

To change the Yolo Version a flag inside the sort algorithm code has to be changed. It is located here: jetson_nano/catkin_ws/src/sort_track/sort_track/src/[track_deep.py] or [track.py]


Fig. 3.5: Object tracking with yolov3 and deep sort

Output: The normal sort algorithm creates a list of detected objects and publishes it under the topic /sort_track. It also saves the images and the movement of the detected objects in a log file. This is saved at /home/nano/isp-2022/jetson_nano/catkin_ws/src/yolov7_helper/. The deep sort algorithm has two different output topics. The first one publishes the detected objects marked with an identifier. The other topic is a vector which tracks the movement of the objects. Both of these informations are also saved on the Jetson Nano itself at /home/nano/isp-2022/jetson_nano/catkin_ws/src/yolov7_helper/.

Installation: For the installation the sort_track folder from our git is needed. After that some python packages have to be installed. The following commands should be everything you need.

```
git pull
pyenv install 3.8.16
pyenv global 3.8.16
pip install numba
pip install numpy==1.21.2
pip install scikit-image
python -m pip install cython
python -m pip install scikit-learn==0.22.2.post1
pip install filterpy
pip install tensorflow==1.15.0
catkin_make
```

Future work: The next step is to design a camera controller. This controller has to keep the selected object in the middle of the picture. For this the vectors calculated in the sort algorithm can be used. The controller will need a lot of fine-tuning since the object detection takes some time and therefore our data is currently ~1 second delayed. It is also important to not change the camera to drastically, else it could lead to the loss of the tracked object.

## 3.3 Stereovision

### 3.3.1 Objectives and Importance

The main objective of the Stereo-vision team was the process of resizing the image to a standardized dimension enhances the speed of subsequent image processing tasks, such as object detection and depth calculation and the implementation of nodes that can enable to compute the **depth** of the detected object.
In this project the **depth calculation** is essential for effective obstacle avoidance. By calculating the depth of detected objects, the boat can make informed decisions about altering its course or adjusting its trajectory to circumvent potential collisions.

### 3.3.2 Implementation

The *bild_skalieren* node subscribes to an incoming image topic from a camera sensor and employs the *OpenCV* library to resize the received images to a standardized dimension of 720x480 pixels. The resized images are then republished to a new image topic. This resizing step aims to optimize computational efficiency, memory utilization, and processing speed during subsequent tasks such as object detection and depth calculation.

The Stereovision node starts by importing the necessary ROS packages and message types, establishing a fundamental connection to the ROS ecosystem. It initializes key parameters such as the focal length, baseline, and image width, which are crucial for accurate depth calculation. These parameters are tailored to the specifics of the boat's camera setup, ensuring precise distance estimations.
The core functionality of the code revolves around the *coords_disp_from_subscriber* function. This function takes the $x$ and $y$ coordinates of the detected object's center and computes its depth using stereo vision principles. It accounts for potential shifts of the object's position to the image center if it's too close to the image edges. This mechanism ensures that depth calculations remain accurate even when objects are detected near the borders of the image frame.
The *object_callback* function, triggered by the Darknet ROS package's bounding box messages, extracts object position and size information. This information is crucial for calculating depth accurately. The code evaluates whether the detected object is positioned too close to the image edges and, if so, shifts the object's position to the center to prevent depth miscalculations caused by skewed object positioning.
The important steps in calculating image depth are as follows

1. **Object Detection Condition**: The code begins by checking if a valid object detection has occurred based on the *object_size* parameter.

2. **Disparity Computation in $x$ and $y$ Directions**: If an object is detected, the disparities in the horizontal ($x$) and vertical ($y$) directions are calculated. These disparities are computed using the object's size and its $x - y$ coordinates in the image, along with the camera's focal length.

3. **Average Disparity Calculation**: The calculated disparities in $x$ and $y$ are then averaged to derive an average disparity value.

4. **Distance Estimation**: The average disparity is used in conjunction with the stereo camera's baseline to estimate the distance to the detected object from the camera. The calculation involves applying triangulation formulas.

5. **Log Distance**: The calculated distance is logged using ROS messaging, providing real-time feedback on the estimated depth to the object.

Given that the bounding box message only generates two $x$ and $y$ coordinates, the image depth was calculated using just one camera (the camera on the right).
However, we have implemented the Stereovision-final node that introduces the concept of using four coordinates, representing the corresponding positions of an object in both left and right cameras of the stereo setup. This refined approach more accurately models the stereo geometry, which is vital for calculating precise depth information.

The *coords_disp_from_subscriber* function now takes four coordinates as inputs: *x_left*, *y_left*, *x_right*, and *y_right*. These coordinates represent the positions of the detected object's center in both the left and right camera images. The function calculates the disparity in both x and y directions, leveraging the stereo camera setup's characteristics, which is pivotal for accurate depth estimation.
In the *object_callback* function, the coordinates of the object's center for both the left and right cameras are directly set. Additionally, the size of the object is defined. These parameters form the necessary inputs for the *coords_disp_from_subscriber* function.

### 3.3.3 Future work

The code used to calculate the depth of the image with four coordinates is already implemented but has just been tested with random coordinates. The missing element is the generation of the 4 coordinates by the bounding box message corresponding to the 2 cameras.

While the updated code that incorporates four coordinates for precise depth calculation based on stereo vision principles appears promising, it faces a significant **constraint** due to the limitations of YOLO object detection computation on the boat's hardware. Despite its potential benefits, the computation demands of YOLO, especially for processing multiple coordinates and object sizes, can prove to be too slow for real-time execution on the given hardware. Real-time simultaneous processing of both images for a rendering of the coordinates, proved to be impossible by the YOLO team.

## 3.4 GPS

The GPS node is responsible for publishing the currently required angle to reach the next target position. This angle is relative to the current heading of the boat and takes into account the relative position of the target in relation to the position of the boat.
The node requires the current position, heading and target which is received via three subscribers. Therefore the three functions *update_target*, *update_position* and *update_heading* are used that are callbacks on the respective ROS topics and use the values given to update node internal variables.
Every time an input value is updated the internal function *calculate_angle* is called that calculates the angle. First the absolute angle between the current position and target is calculated. This angle is the transformed to be relative to the current heading.

Finally the angle is shifted to a range between -179° and 180° and published to the *mavrossetpoint_angle* topic for the control node to use.
Everything is logged in detail using print statements and the logging topic. An example output from the running gps node with updating position is shown here:

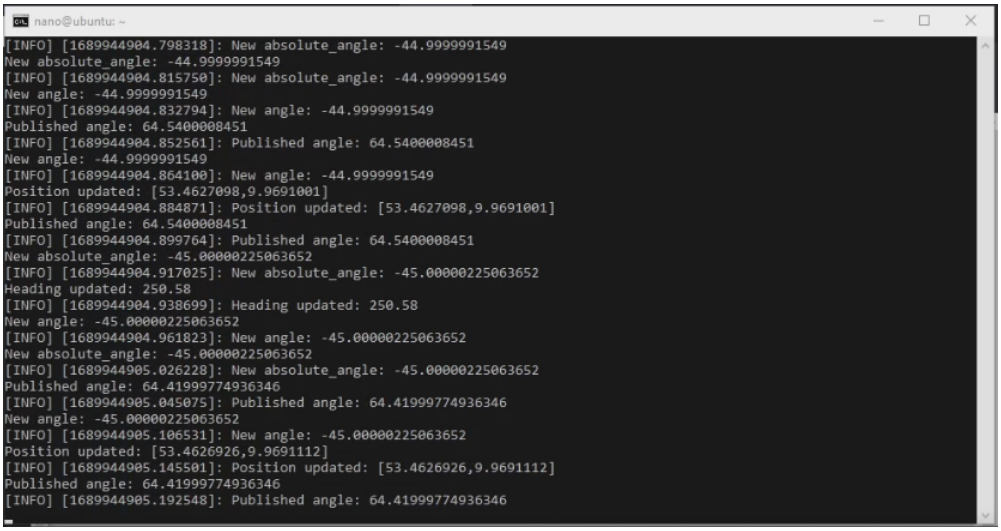


Fig. 3.6: GPS node output

This node can be started independently by executing *rosrun boat_control gps* in the scrips folder or automatically by the *startscript*.

## 3.5 GUI

The GUI is based on a flask web server running on the jetson nano.

A map is shows, coordinates can be added via clicking on the map or via inserting them in the form below.

This node can be started independently by executing r̈osrun boat_control guiïn the scrips folder or automatically by the start script. The web page can be accessed by a web browser on a device which is in the same local network as the jetson nano. Latter needs a internet connection. The web page runs on the IP-Address of the jetson nano on port 5000. If IP-Addesss:port have been typed in the browser the map might not load properly. A workaround is to right click on the webpage and select **Element Untersuchen** (Or comparable) and close the opened tab right afterwards.

The GUI nodes main component is the web page from which different functions are called. If clicked on the map on the web page the point on the map is directly added as a target to the lists of targets. A target might also be added via the form with the fields *latitude* and *longitude*. After pressing the *add target* button the target appears in the list with all targets at the bottom. The button *reset route* deletes all targets from the list of targets, if pressing *start route* the function *start_route()* is called and the boat begins going to the first target in the list of targets.


Fig. 3.7: GUI web page

Step by step guide to start and use the GUI:

1. Modify (replace the XXX by 103 or 112 most likely) the ip address in "../src/boat_control/gui.py" (line 139)

2. Do the same modification to the ip address in "../src/boat_control/templates/gui.html" (lines 42,47,50,54)

3. Start the GUI node in the "scripts" folder with "rosrun boat_control gui"

4. Access "192.168.0.XXX:5000" (XXX being 103 or 112 most likely) in a web browser on a computer connected to the mtecsail wifi. Scroll down to see the input fields and buttons.

5. Enter coordinates in the "latitude" and "longitude" field as numbers with "." as decimal point. For example 53.462052 and 10.024129.

6. Add the target using the "add target" button. The website will reload and you should see the added coordinates at the bottom of the page. Add as many more coordinates as needed by repeating step "5." and "6."

7. Finally start the route by clicking the "start route" button. (If the GPS module is active at this point the current position will be added as the final target to return to.)

8. See the output and log for details, what the node is currently doing and the current distance to the next target.

9. Use the "reset route" button to discard your current route. Afterwards new targets can be added and a new route started.

## 3.6 Wind Sensor

The wind_sensor node allows to publish the data related to the wind_sensor as a Float32 message onto the 'wind_data' topic. After initializing the node it searches for the calibration data ('WIND_MIN' and 'WIND_MAX') contained in the JSON file 'calibration_data.json'.

```
FILENAME = "calibration_data.json"
...
\__location__ = os.path.realpath(
os.path.join(os.getcwd(), os.path.dirname(__file__)))
file_path = os.path.join(__location__, FILENAME)
with open(file_path, 'r') as f:
    calib_data = json.loads(f.read())
```

If the values exist, it will update them in the global variables, instead of using the predefined range of 0 to 1024, as we can see in the following code section:

```
WIND_MIN = 0
WIND_MAX = 1024
...
if calib_data["WIND_MIN"] is not None:
    WIND_MIN = calib_data["WIND_MIN"]
rospy.loginfo("Wind_min: " + str(WIND_MIN))

if calib_data["WIND_MAX"] is not None:
    WIND_MAX = calib_data["WIND_MAX"]
rospy.loginfo("Wind_max: " + str(WIND_MAX))
```

In the script main loop, the 'wind_data' is read from the 6 RC channel and then converted to degrees for clearer interpretation, using the following formula:

```
wind_converted =(wind_speed-900)*(360/(WIND_MAX-WIND_MIN)) - 60
if wind_converted < 0:
    wind_converted = 360 + wind_converted
    rospy.loginfo(str(wind_converted) + " degrees")
```

The adjustment of -60 degrees was applied to solve for the sensor's calibration and the value always falls between the range of 0 to 360 degrees.

After this data filtering, the 'wind_converted' is published on the 'wind_data' topic with a sleeping rate to control the loop, waiting for 1 second between iterations.

```
self.pub.publish(wind_converted)
        rospy.sleep(1)
```

### 3.6.1 Arduino Nano

In order to better interface with the 'wind_sensor' and process its data, it was also implemented code in Arduino. The Arduino-Hardware is placed between the windsensor and the APM-Board.
After setting up the Arduino Board's configuration, the 'loop' function is the main execution of the program:

- measure(): counts the rotations and calculates the wind_speed;
- it then waits for the pulse on 'pinDir' (pin of wind direction) to estimate wind direction;
- calcSpeed(): is used to calibrate and scale the wind data. The speed value is constrained within a range and mapped to a PWM value;
- calcDir(): is used to average the wind direction data by taking into account 5 of the stored values and returning its mean.

To update the code on the Arduino, the following steps have to be performed:

1. **Download** and install Arduino Software (e.g. Arduino IDE 2.2.1)
2. **Update** the code inside the editor
3. **Reflash** the Arduiono:
   a) Connect the Arduino via USB-cable to your PC/Laptop
   b) Perform setup:
      - Tools → Port: select connected Port
      - Tools → Processor: "ATmega328P (Old Bootloader)"
   c) Click the Button "Verify" and afterwards "Update"

## 3.7 ArduPilot Mega (APM-Board)

The APM-Board is a microcontroller board, which has a direct interface to the hardware, such as sensors and servos. It is connected to the Jetson Nano via USB and communicates with the MAVLink protocol. The software on the APM board is mainly used for control tasks, which need low latency. This would be more difficult on the Jetson Nano, as the communication over MAVLink takes some time.

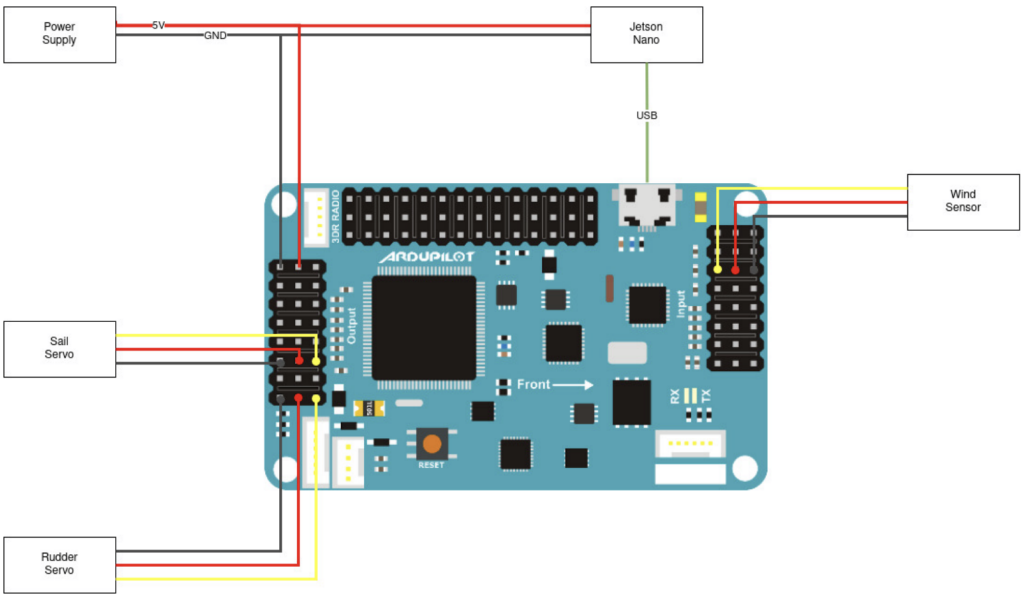


Fig. 3.8: APM with connections to peripheral devices

The USB connection between the Jetson and the APM is used for communication and power, however another power supply needs to be connected to power the servos. The wiring should be done like in figure 3.8.

The APM has several internal sensors available, which include 3-axis gyroscope, compass and accelerometer. In the software many of the sensors are condensed to the "ahrs" object, which contains some state variables of the boat. It is important to note that the APM board is mounted in a backward position in the boat, which leads to a 180 degree offset in some angles.

The main folder of the APM software can be found in the apm_boat/APMboat directory. The software implements a modified version of the ArduPilot project. The "Rover" class is the main class used for new functionality concerning the boat. The Scheduler, which periodically starts the main functions, is implemented in the "APMboat.cpp" file. Reading sensor data and moving servos accordingly is implemented in the "BoatControl.cpp" file. Variables, which should be communicated to the Jetson Nano over MAVLink can be found in the "Parameters.h" file. The object, which holds these variables in the Rover class is the "g" object. Default values for those variables can be set in the "config.h" file, but they can also be set after startup through the MAVLink connection (e.g. in the calibration).

The control of the servos is one of the features implemented on the APM. There are different control modes, which can be set over MAVLink.

In control mode 1 there are two target angels, which can be set from the Jetson Nano. The target angles are between hull and sail (SAIL_ANGLE) and between sail and wind (TARGET_APPARENT). The sail position is set once, when the target angle changes. Much of the controlling work is done with the rudder to keep the angel of the boat to the wind as constant as possible. This computation includes both target angles. Very small changes in the wind direction will be ignored, to stabilize the course of the boat, as the wind sensor direction will naturally vary even if the wind is relatively constant. The short term controlling to reach these angles is done on the APM, while a long term course can be reached with modifying the target angles from the Jetson. The goal of this control mode is to keep the boat moving all of the time, as the rudder is not useful when the boat is stationary. This might lead to the course being not the shortest possible, but avoids not being able to control the direction of the boat at all.

In control mode 2 the target is to control the compass course of the boat. This leads to the target course being the shortest distance to the next waypoint possible. However this way might not be feasible, when the wind direction is not suitable for the computed course. When this this is the case the speed of the boat might drop, which leads to the rudder not being useful and with that not being able to control the direction of the boat at all.

Control mode 0 is used for manual controlling using the remote. The current control mode is switched to 0, if an input on the remote is detected. This means it is always possible to override the automatic control with the remote and switch back to automatic control afterwards.

Most of the previous testing was done in control mode 1, the current control code on

the Jetson Nano also requires control mode 1.

If there are strong gusts of wind the boat might be pushed down on it's side. In this position the sail isn't useful and the rudder is partially out of the water. To counteract this behavior, the APM checks the roll of the boat and in case of extreme roll (>70 degrees), the sail will be set to it's maximum position of the particular side. This leads to the wind having less of an effect on the boat, which can then go back to the original position more easily. After the boat is back in the normal operating range the sail position is reset.

New values to the servos aren't directly written to PWM, but are first processed by a ramp (in file "ServoControl.cpp"). The ramp uses the current movement of the servo and external parameters to calculate the next step. This leads to acceleration and deceleration to a maximum speed. The maximum step length can be configured over MAVLink (see section 3.8). A higher maximum step length also leads to faster acceleration and deceleration.

The control of the camera gimbal is implemented in the "GimbalConntrol.cpp" file.

Uploading new code to the APM-Board is done through these shell scripts:

- initJetson.sh: Creates initial folders and files on the Jetson Nano, which are used by other scripts (Only needs to be run once)
- makeHex.sh: Build hex file
- uploadHexToBoatJetson.sh: Moves Hex file to Jetson Nano, from where it is uploaded to the APM using avrdude
- cleanHex.sh: Deletes the Hex file (Doesn't need to be run for uploading, mainly for debugging)

Messages from the APM (e.g. for debugging) can be sent with "gcs_send_text_fmt(PSTR())" and viewed through mission planner (ArduPilot project) or mavros.

## 3.8 Calibration

The calibration is used to determine servo and sensor values, which correspond to certain positions. The calibration is part of the "boat_control" package. It consists of the executables "calibrate" and "quick_calibrate" , which can be started through the rosrun command.
For the first start of a new boat "calibrate" should be run. In the script the maximum, middle and minimum values of the servos are obtained, and the middle position of the windsensor is established. The angles used for calibration were 65 degrees for the sail and 40 degrees for the rudder. These may be varied, but should be the same on each side of the boat. The calibrated servo and sensor values are saved to a json file: "calibration_data.json" and can later be used. "Calibrate" only needs to be run again, if the values change(e.g. changes to servos).

Some other important values can be manually set in "config.json".

**Config values:**

- "TARGET_MODE": The mode of the APM control loop(1 is control to two angles: Target Apparent and Sail Angle; 2 is control to compass course)
- "SERVO_INCREMENT": Used for servo ramp, controls how fast the servos accelerate, decelerate and move
- "TARGET_APPARENT": Target angle between windsensor and sail (Only for initialization, should later be used in other scripts)(The unit of angels is tenth of a degree to avoid float values)(Angels should be positive clockwise, when looking from above)
- "SAIL_ANGLE": Target angle between sail and hull (Only for initialization, should later be used in other scripts)(The unit of angels is tenth of a degree to avoid float values)(Angels should be positive clockwise, when looking from above)
- "R_SERVO_UPRIGHT": Direction of the rudder servo (On most new boats 1, for old boats 0)

The values from the two json files can now be used in "quick_calibrate", which is used to write the values to the APM-Board through MAVLink. Quick calibrate checks if values were set correctly and might retry to set the values.

For both scripts, roscore and mavros need to be running before starting the calibration. The calibration can be run with the following commands:

```
rosrun boat_control calibrate
```

```
rosrun boat_control quick_calibrate
```

## 3.9 Logging

The "logging" package handles logging of ROS messages, raw camera images and data on the APM. Logging is handled by the LogROSProcess and the LogAPMProcess. Log data can be found in the "loggingNode/log" directory.

Logging of APM values is done periodically, the connection is established with help of the dronekit package. Data includes the gps position, angles in space and several other values.
Logging of ROS messages is used on the Jetson. Every other node can send string messages to the "/logmsg" topic, which are then condensed into one file by the logging package.
Logging of raw camera images is done by subscribing to the "/camera/frame_both" topic, which is published by the camera node.
For reconstruction of the state of the boat, every log record includes a timestamp, either directly in the log text or in the file name (for camera images).

For logging ROS messages, roscore needs to be running. Additionally for logging of camera images the camera node needs to run. All other nodes can be started after the logging. If mavros should run simultaneously, logging should be the first to be started. It may be necessary to run "pip3 install ." in the loggingNode directory after changes to the package. Logging can be started with the following command:

```
rosrun loggingNode logging
```

## 3.10 Communication

The USB connection between APM and Jetson is used for serial communication over the MAVlink protocol. On the APM side the values, which should be shared are in the "Parameters.h" file and in the corresponding object "g" in the Rover class. MAVLink communication on the side of the Jetson Nano is handled by the "mavros" package, which provides a ROS interface for communication over MAVLink. The ROS topics for mavros all start with "/mavros". Part of this functionality is wrapped in the "apm" node in the boat_control package. It provides functions for getting and setting values on the APM without need for additional configuration.

Another suggested means of communication was the use of **MAVProxy**. The purpose of using MAVProxy was to serve as a vital intermediary for facilitating efficient communication between the APM board and the Jetson Nano. MAVProxy streamlines the MAVLink communication process, enabling real-time reception and interpretation of servo data from the APM board, encompassing critical parameters such as rudder and sail positions. This data serves as a fundamental component for monitoring and controlling the boat's control surfaces, ensuring safe and stable operation, and contributing to mission planning and execution on the Jetson Nano. It's worth noting that while the implementation was tested, the desired results were not fully achieved, indicating the potential need for further development and troubleshooting.

## 3.11 Credentials

Laptop User name: projekt Password: mteclabor2022
Laptop VM User name: developer Password: password
Jetson Ubuntu 20.04 User name: nano Password: jetson Mac-Address: 28:D0:EA:88:52:DE
WiFi SSID: TP-Link_9C4A Password: 94588637
WiFi Admin Panel IP: 192.168.0.1 Password: mtecsail2022

# 4 Start Script and Future Work

## 4.1 Start Script

The boot can be started via a start script (bash script) in the "jetson-nano" folder.

## 4.2 GUI Interface for Improved communication with the Boat: Websocket

In order to accelerate the world detection and the sailing process in terms of its performance, an attempt was made in the last few weeks of the project to replace the HTTP Flask server with a web socket node (using the Python library SocketIO), which is connected to a Node-js React-WebApp-frontend connected to the boat. A node was created that connects to the socketIO ([10] and [6]) server when the boat is started (backend of the frontend). The frontend is based on a main page (with Open Street Map via the leaflet [3] package), which can add waypoints with a click and should communicate the coordinates via the web socket in the future and a possibility via the web socket to generate generic points on the map to draw and to visualize lines (courses). These basic building blocks (visualization of lines and points) were implemented by us in preparation. However, due to the project completion date, the work has not been continued. An insight into the frontend can be seen in figure 4.1. The required nodes packages can be created using the command "npm install" in the "gui-frontend" repository folder. The Backend also needs packages that are installed in the "Backend" folder using "npm install" must also be installed. Both applications are started in two different terminals using "npm start" for the frontend in the "gui-frontend" folder and "node index.js" in the "Backend" folder.


Fig. 4.1: View of the Main-Home-Page

# 5 Conclusion

In summary, during this project we created a central controller node managing the autonomous sailing process based on GPS-calculation. The waypoints for this sailing process can be set via the also-created GUI. We calculate the path to be taken by the boat with the angle to our list of waypoints and also with respect to data from the wind sensor.

Furthermore, we enabled the boat to perform object detection and tracking. For this YOLOv7 and (Deep) Sort were implemented as well as multiple possibilities to log the resulting data.

Lastly running infrastructure in terms of start scripts and screen processes were also created for the sailing and the vision parts of this project to ease the startup of the boat and subsequent tests.

**Knowledge gained** The project gave all of us the opportunity to learn a lot in numerous fields surrounding the boat. The skill to read and get into an existing project that was already quite complex was needed right at the beginning. Hand in hand with that came the knowledge of how to handle a framework like ROS(PUB/SUB communication between nodes) and the complexity involved when multiple embedded devices form a single embedded system, especially if these different hardware components need to communicate with one another.

The project also taught us a lot about the basics of sailing and vision processing, especially regarding certain algorithms that can be used and general concepts to be employed. As well as navigation calculation and HTML implementation. Further lessons to be learned were the high impact of Version-Controlling and later into the project the resulting high importance of testing which became quite apparent to us.

Finally, due to the big team, we learned a lot about general forms of project work and the fact that huge teams equal a huge organizational effort.

**Future work** Further work to be done on the boat may include:

- The implementation of a sailing controller on the APM board itself for fluently moving sail and rudder without added delay between the Jetson and the APM.
- The implementation for further usage of a WebSocket to control the boat (see algorithm steps (autonomer Zwilling))
- The implementation of advanced pathfinding in the control-node, including obstacle avoidance based on the implemented object detection/tracking and depth calculation.
- The communication between camera and APM controller.
- Enabling a gimbal algorithm to keep tracked objects inside of the camera's view, resulting in better recognition and positioning of objects in a map.

- Efforts to improve performance of the Jetson Nano.

# Bibliography


[1] Official yolov7. https://github.com/WongKinYiu/yolov7, accessed: 2023-09-01

[2] Ros package for official yolov7. https://github.com/lukazso/yolov7-ros/tree/master, accessed: 2023-09-01

[3] Agafonkin, V.: Leaflet (2023), https://leafletjs.com/

[4] Bewley, A., Ge, Z., Ott, L., Ramos, F., Upcroft, B.: Simple online and realtime tracking. In: 2016 IEEE International Conference on Image Processing (ICIP). pp. 3464–3468 (2016)

[5] Österreichische Gesellschaft für innovative Computerwissenschaften: World robotic sailing championship (2010), https://www.roboticsailing.org/

[6] Grinberg, M.: Python-socketio (2021), https://python-socketio.readthedocs.io/en/latest/

[7] Incorporated, A.: Cyberknife® system technology (2023), https://cyberknife.com/cyberknife-technology/

[8] Robotics, O.: Mavros (2018), http://wiki.ros.org/mavros

[9] Robotics, O.: Ros - robot operating system (2021), https://www.ros.org/

[10] socket.io: Socketio (2021), https://socket.io/

[11] Wojke, N., Bewley, A.: Deep cosine metric learning for person re-identification. In: 2018 IEEE Winter Conference on Applications of Computer Vision (WACV). pp. 748–756. IEEE (2018)

[12] Wojke, N., Bewley, A., Paulus, D.: Simple online and realtime tracking with a deep association metric. In: 2017 IEEE International Conference on Image Processing (ICIP). pp. 3645–3649. IEEE (2017)