SLAP

April 1, 2025

1 Introduction

Self Logging Auto Pilot (SLAP) is my AQA A Level NEA completed in 2025. The aim of this project is to develop a prototype real-time control system using low cost componants. featuring elements of robotics, electronics and software. The project has been an oppotunity to write a larger software system and gain experience of many programming techniques to overcome the challenges that the project presented.

The choice of this project was driven by my interests in sailing and real world control systems. The project shares many aspects of aerospace engineering which I aim pursue in future. The project has been a successful learning experience and has included a number of difficult problems which took many hours of work to overcome.

The approach to this project has been one of [iterative development]. I started the development work by creating experiments in python to test and understand the various parts of the system individually. After this early work, I created Iteration0, this being the first iteration of SLAP. Each time realised the code needed restructuring I created new iteration. In the final iteration (Iteration2), I found the structure worked well enough to complete the project.

One particular problem in this project is that it has not been possible to fully test the auto pilot on a sailing boat due to the season. To solve this problem this project also includes the computer modelling of boat steering dynamics. This allowed me to test my control system without the need of the physical boat. The other advantage of the building the simulator was that I developed a better understanding of the problem at hand. The development of SLAP is both about meeting the users needs but also about developing a real time closed loop control system.

Throughout the development I used the Git source code repository to manage versions of my code and enable me to roll back my code when problems spiraled. I used a combonation of manual testing and unit testing to ensure the functionality and I used a task list to manage my progress.

2 Analysis

2.1 Background

Long distance sailing is restricted by the necessity to course correct during travel. This requires a person to be actively steering the vessel for excessive periods of time which prevents the crew from preforming other important tasks this problem is exacerbated when sailing single-handed. Maintaining a boats course is complex, due to many factors affecting the direction of travel. This could be automated using a computer system of which corrects a boats course to a predetermined destination. This allows longer distance travels to be more accessible.

Auto pilot systems have been used in marine navigation since the early 20th century, with the first patent for a marine auto pilot being filed in 1916. These early systems were mechanical and used a gyroscope to maintain a steady course. The development of electronic systems in the 1960s and 70s led to more sophisticated auto pilots that could maintain a course using electronic compass readings. Modern auto pilots range from simple mechanical systems to complex computer-controlled systems that can interface with GPS and other navigation equipment.

The primary purpose of an auto pilot is to maintain a vessel's course without constant human intervention. This is particularly important for small sailing boats where crew numbers are limited. On longer passages, maintaining a constant course through manual steering is physically demanding and can lead to fatigue. This becomes especially challenging when sailing single-handed, where the sailor needs to manage multiple tasks such as navigation, sail trim, and monitoring weather conditions while also controlling the boat's direction.

Today's market offers various auto pilot solutions. More expensive systems integrate with boat's navigation systems and can cost thousands of pounds. These systems often include features like GPS tracking, wind monitoring, and the ability to follow complex routes. Simple mechanical systems like wind vanes and basic electronic auto tillers provide a more affordable solution for small boat owners, though with limited functionality.

2.2 Problem Statement

This project sets out to build an Auto Tiller using ready available componants and a smart-phone. The specific problem is how a boat's heading can be controlled given the boat's steering dynamics and any distruptions to its heading from the sea, wind and tide conditions. How can such a system be developed as the development cannot be made at sea? The project also looks to find what functions can be added given the oppotunities a smart-phone presents.

I have contacted two sailors who would typically use an auto pilot on their boat perticularly for long single handed passages. To understand their needs I created a questionaire and I focused on understanding requirements by asking them to describe how and when they would use an auto pilot system. The result of the questionaire have shaped this project by attempting to achieve and complete the set out objectives.

2.3 Reaserching the problem

There are many different forms of auto pilot used on sailing boats. These range from electonic systems linked to the boats navigation systems through to mechanical wind vane systems. The mechanism auto pilots control the boat also varies as some driver the tiller / rudder directly and some have an auxillary powered rudder and some use a system known as trim tabs which function like ailerons on a aircraft wing. Most of these commercial systems are expensive to purchase however the basic parts for an auto tiller can be obtained inexpensively.

The common feature of all self steering and auto pilots is [closed loop control]. This means that the system in some way measures the difference between desired and actual boat heading. This difference must be fed back to the boats rudder in such a way that the difference is reduced. How this difference is fed to the rudders varies however most basic systems use an algorithm such as [PID] although some modern system use machine learning.

As a part of gathering requirements I investigated the specification and operating instructions for a Raymarine ST1000 auto tiller. This was very helpful in undertanding the system requirements

and operation.

2.3.1 RayMarine ST1000

2.3.2 Summary of functions

Navigation

Start / Stop -User Action- - Press start or stop button

System Response - engage / disengage auto pilot control

Adjust -User Action- - Press one of the increment buttons

System Response - increment current target angle by buttons value

Config Mode -User Action- - Press change mode button

System Response - Cycles through the 4 modes

Calibration -User Action- - Cycle through options to correct menu - Use increment buttons to change value

System Response

- Save changed setting and maintain calibration

2.4 Identification of Users

There will only be one intended user for SLAP who is the boat's skipper or helmsmen. Their role is to input the desired heading into the computer. In this regard identifing users for SLAP is somewhat simple.

2.4.1 Identification of users' needs and acceptable limitations

To identify the users needs I developed a questionaire which I made available online using Google Forms. I first contacted two sailors and explained the project and then asked them to complete the questionaire. The idea behind the questionaire was to develop a [user story] which is a simple description of the user, their goal in using the system and a step by step description of how the user interacts with the system and what the system does in response to these interactions.

The questionaire and its responses are included in appendix A.

2.4.2 General User Story

Using the material from the questionaire, the following general user story has been created.

"Sam the sailor sets of on his boat alone. He motors out to open water and points his boat into the wind. He then raises his sails and bears away to start moving. He sets his course in the direction he wants to travel and he adjusts his sails to achieve a balance between the main sail and the jib such that the boat holds a steady course without significant force on the rudder. Sam relaxes and steers the boat for some time. Having satisfied himself that everything is ship shape he decides he would like to cook himself dinner. Sam has already connected the SLAP tiller actuator to his trim tabs on the rudder. He switches on the power to slap and using his mobile phone he connects to the wifi hotspot provided by SLAP. He opens a browser on his phone and enters SLAP into the address bar. He is immediatly taken the to the SLAP home page. After a few moments, SLAP obtains a GPS signal and sam can see his current heading shown on the compass rose in slap. Sam decides to log his journey and first presses the log button on SLAP which begins reading the SLAP sensors and recording them in log. Sam now presses start (Auto Pilot) and SLAP engages. SLAP now measures the boat's heading using the GPS and compares this to the heading recorded when he pressed start. SLAP now adjusts the tiller actuator in response to any error in the course. Sam waits to check the auto pilot is functioning and heads below to make his dinner. While making his curry, Sam keeps his mobile phone in view and occasionly checks the compass rose where he can see the current course, the tiller angle and the desired course. Sam can feel the boat has slowed a little because the wind has shifted slightly, Sam uses his judgment as a sailor and presses the '+1' degree button on slap three times to trim the boat to the wind. Sam takes his dinner up to the cockpit where he enjoys his meal while SLAP looks after his boat.

Sam sails through the night taken the occasional uncomfortable 15 minute nap while SLAP steers the boat. In the morning he arrives near to his destination. Sam presses stop on SLAP's interface and SLAP disengages and stops recording. Sam hands the steers the boat under the power of his engine into harbour. Sam ties up the boat and makes some coffee. Sam opens his mobile phone and connects to the harbour wifi. Sam opens SLAP and navigates to the 'Trips' table. He sees the latest trip in the list and presses upload. SLAP connects to the internet and uploads the sailing log to an online mapping service. When complete Sam presses view on SLAP and is presented with a map view showing the course of his boat as recorded by SLAP throughout the night. Sam is a happy sailor."

The story above provides a helpful guide to the users requirements. This story is broken down into use cases in the design section of this report. These use cases can then define a substantial part of the testing.

The user needs to be able to effectively use the software whilst under the sometimes challenging conditions found whilst making a passage so large interactive elements are important. The user is expected to be competent in basic marine navigation as the system should not be completely relied upon.

2.5 Modelling the Problem

As a part of the analysis, research was necessary to identify a suitable control algorithm.

Closed Loop Control - The rocket problem - PID Control example - Informing the design - Disturbances

2.6 Scope of Information

The analysis below considers the information content within the project. It covers the input and output data and their characteristcs. This analysis informs the design section.

2.6.1 Data Sources and Outputs

Incoming data would be from various sensors around the boat to read information about the environment and boat's condition. These could include optional data such as wind direction, strength , magnetic compass (magnetometer), movements (accelerometer) depending on which sensors have been added to the SLAP system. In all cases these sensors output numerical values which are processed and stores in SLAP's log database table. Information about the users desired heading will be inputted into the system in degrees. The system needs to be easy to activate and deactivate in case of emergencies. The other inputs to the system are the configuration parameters. These are used to tune the boat control system to produce the best response for the perticular boat's characteristics.

The primary output of the system is a signal to drive the tiller actuator which changes the boat's direction. This numeric value within the software must interface with the tiller actuator motor itself. The secondary outputs of the system are the information readouts presented on the SLAP user interfaces. These include the headings, the tiller position and the sensor readings. The final output from the system is a trip log showing the waypoints measured throughout the journey on a map.

Potential outputs may be visual representations of the systems performance including the difference between the desired course and actual course.
After a course has been completed, data will be inserted into a log table to then be used as reference for future use, adjustments, and performance assessments. A score could then be outputted based on the performance assessment.

2.6.2 Data Volumes

All incoming data will be read in regular intervals of approximately 2 minutes whilst the system is active. As an estimate the system would take data from a given number of sensors every second. We assume all sensors are within two bytes, however this may become larger due to the encoding

of the information. As an estimate for each byte of raw data we may need ten bytes of storage. A passage of 4 days (96 hours) would then result in a 3MB log file.

2.6.3 Data Dictionary

Primary Sensor Data

GPS will be used to get positional data about the boat and its location relative to the desired destination using its longitude and latitude. GPS will also be used to determine approximate speed and direction using the standard NMEA format. Speed and direction will be stored as an integer value

A Motion Sensor will be used to report information about the boat's movement and by extension the seas condition. It will have an integer value for each axis.

Log files will include many different data types, such as time (integer), distance (integer)

Data Collection

Current Time will be stored for each read of each sensor.

A motion sensor, a mangnetometer, a temperature and pressure sensor will be intergrated into the system

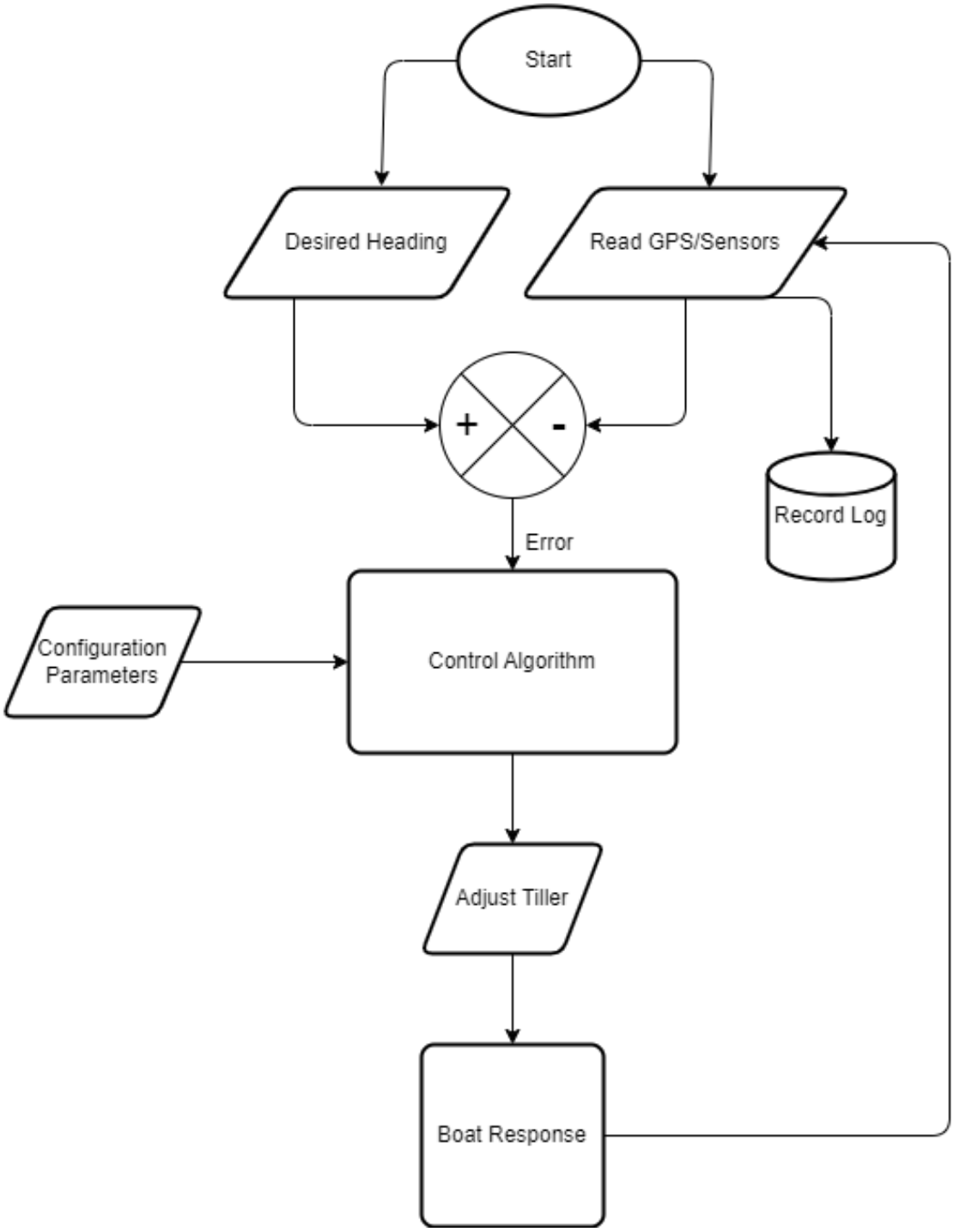
The destination will be collected from the user and stored as a longitude and latitude coordinate (integers).

Actual course will be stored as a change in longitude and latitude (integers).

Desired course will be stored as a change in longitude and latitude (integers) from start position to desired destination.

Actuator value will be stored as an integer value.

Data Flow Diagram



The diagram shows an analysis of the necessary data flows in the system. It shows how the user's desired heading is compared with the GPS sensor heading and the difference is fed into the control algorithm. The configuration parameters are used by the control algorithm to set the tiller position. The boat now responds and changes its course. Which is indirectly fed back to the gps (as it reads the new heading).

Entity Relationship Diagram

Analysis of the requirements results in the design of the following entity relationship diagram.

The database consists of four main tables:

Config Table: - Contains parameters used to tune the control algorithm - Each parameter has a name, value and description - Parameters include PID control values and other tuning constants - Multiple config sets can be stored for different conditions

Sensors Table: - Maintains a registry of all sensors connected to the system - Each sensor has a unique ID, name, type and calibration data - Sensor types include GPS, compass, wind speed/direction, accelerometer etc. - Status field tracks if sensor is active/inactive

Trips Table: - Records details of each sailing trip/passage - Contains start/end times, start/end locations - Stores target heading and actual course taken - Links to config settings used for that trip - Performance metrics and statistics for the trip

Readings Table: - Stores all sensor readings during trips - Each reading links to a specific trip and sensor - Includes timestamp and raw sensor value - One reading record per sensor per measurement interval - Used for analysis and performance optimization

The relationships show that each Trip uses one Config set, each Reading belongs to one Trip and one Sensor, and multiple Readings can exist for each Trip-Sensor combination. This structure allows comprehensive logging and analysis of the autopilot's performance over time.

ERD DIAGRAM HERE

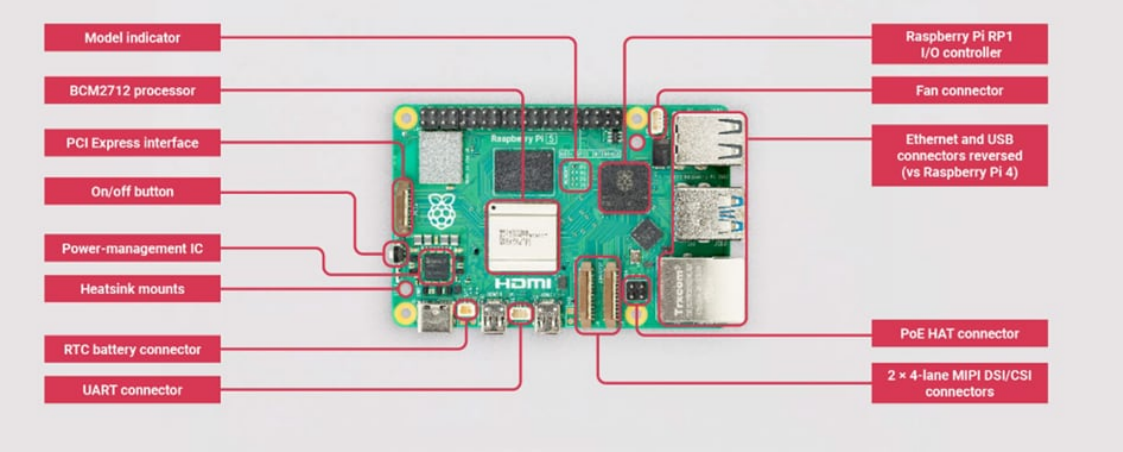
2.7 Proposed Solution

The proposed solution, SLAP, is a prototype sailing autopilot system built around a Raspberry Pi computing module. The hardware components include a GPS sensor for position and heading data, a magnetic compass for precise directional information, enviromental sensors and a servo motor system to control the tiller. The Raspberry Pi runs a Python-based control program that implements a PID control algorithm to maintain the desired heading. The software architecture includes modules for sensor data, heading calculation, control logic implementation, and data logging to an SQLite database. The system features a web-based user interface allowing sailors to set course parameters and monitor performance in real-time. All sensor readings, trip data, and system configurations are logged to enable review. A modular design will allow for easy expansion with additional sensors and future enhancements.

2.7.1 List of Hardware

2.7.2 Rasberry Pi

The rasberry Pi 5 Compute Module is chosen as a development platform. It allows easy electronic interfacing, wifi connectivity and is powerful enough to full Python programs.



Rasberry Pi 5

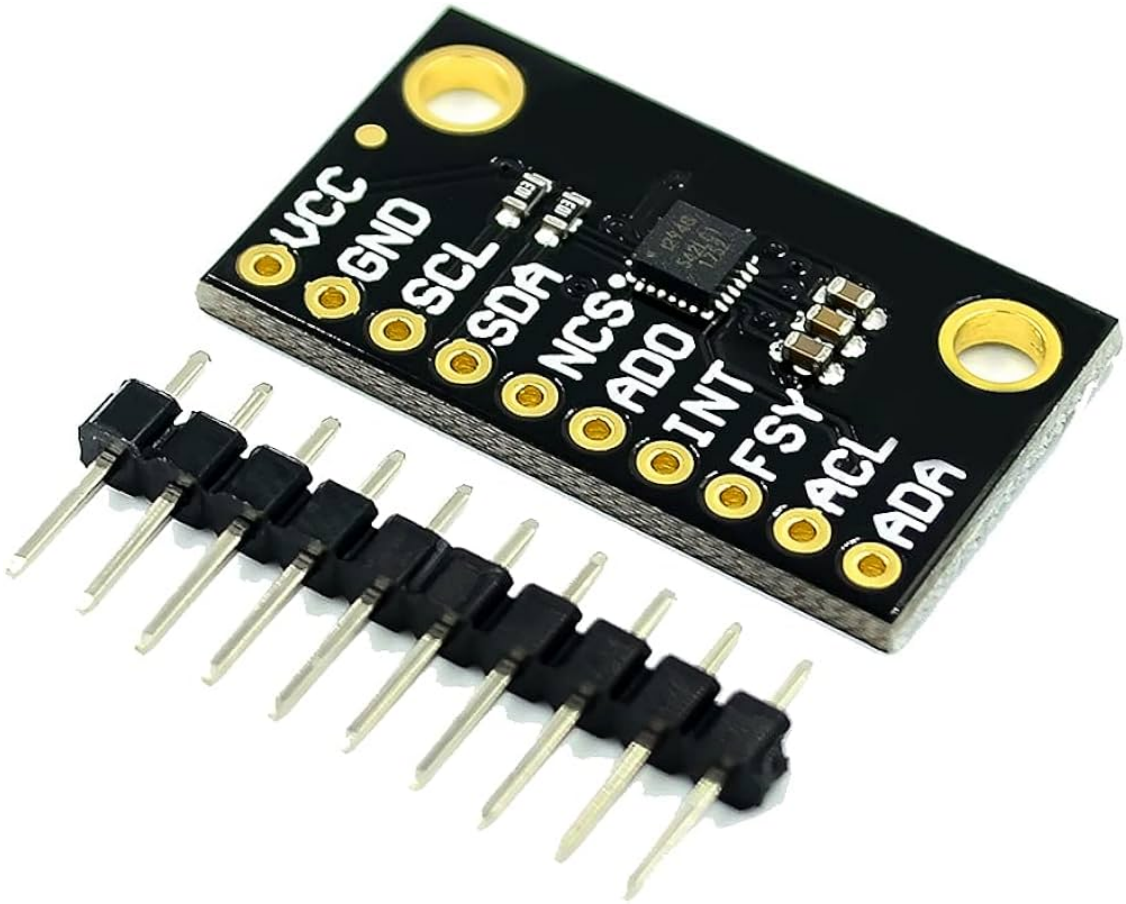
2.7.3 8120MG Digital Servo Motor

The 8120MG Digital Servo Motor is a high-torque servo, providing 20kg/cm of torque. It features a 270-degree rotation range. The servo accepts standard PWM control signals. The servo is suitable for use in the SLAP to adjust a tiller position.

8120MG Digital Servo Motor

2.7.4 ICM204948

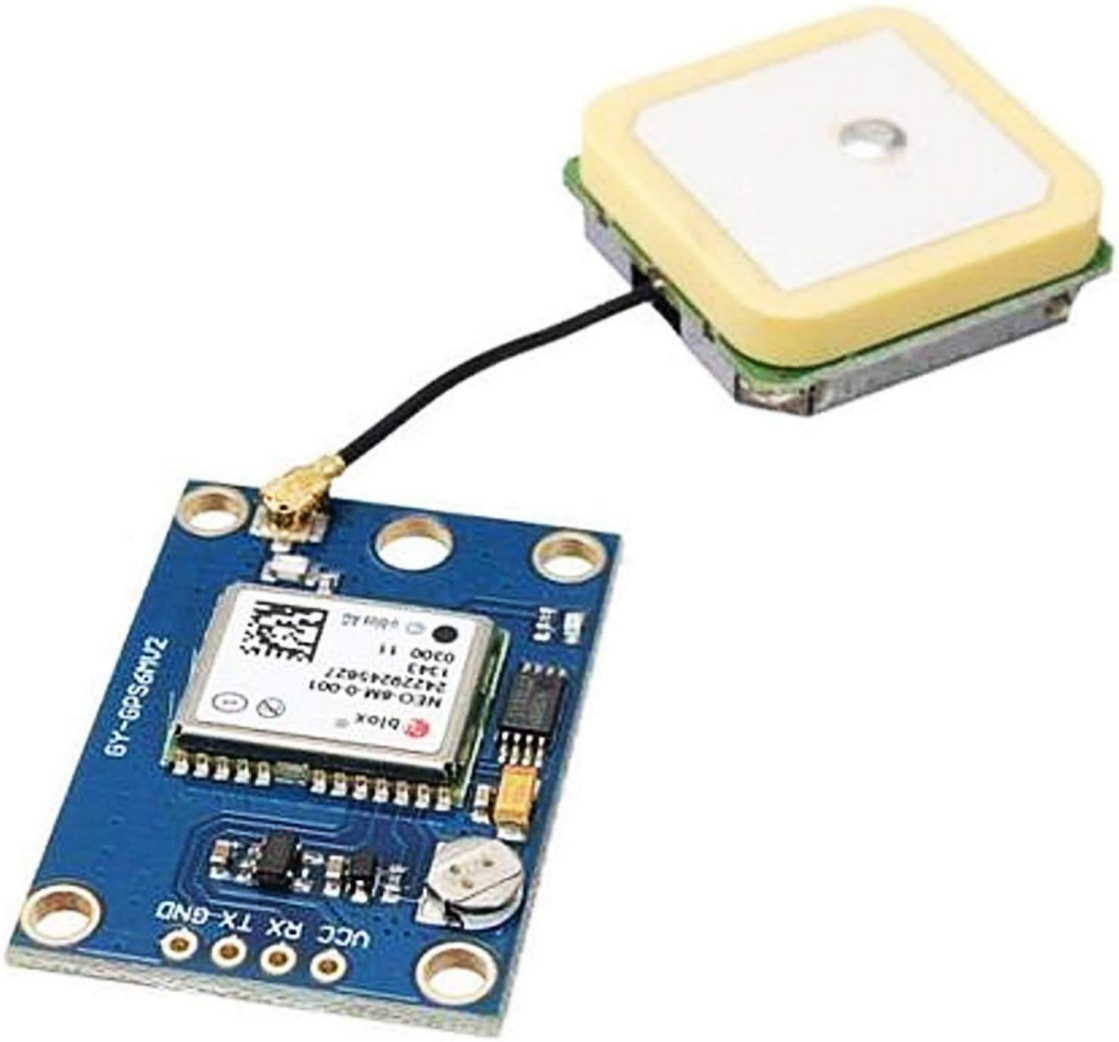
The ICM20948 is a 9-axis motion tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, and 3-axis magnetometer. It provides accurate motion sensing and heading data through I2C/SPI digital interfaces. The module is suitable for use in SLAP to provide precise orientation and movement data for improved heading control.



ICM204948

2.7.5 NEO6m GPS

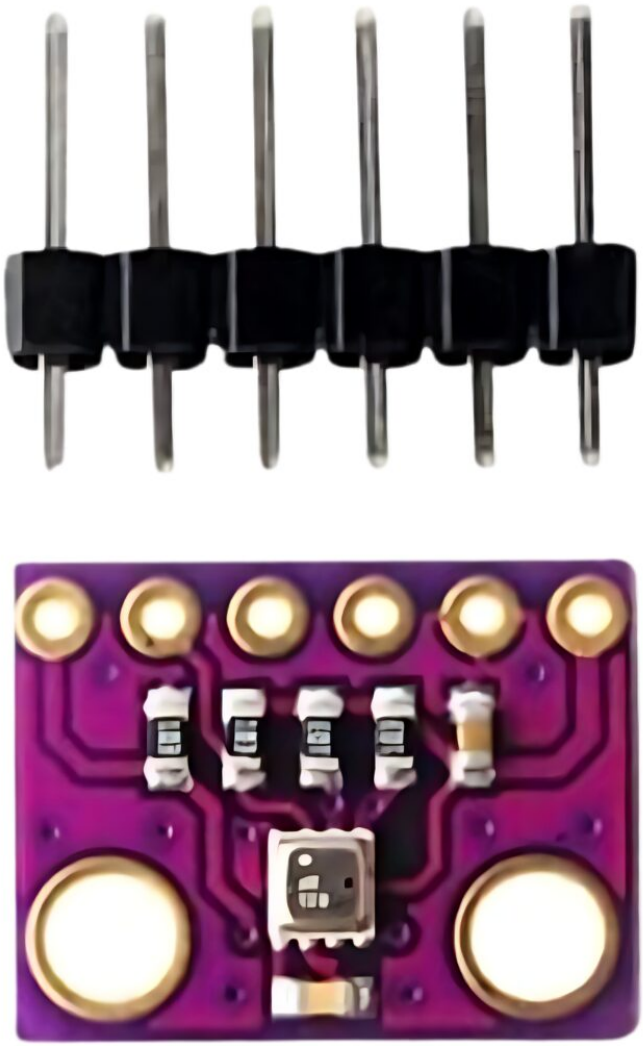
The NEO-6M GPS module provides accurate positioning and navigation data. It reads GPS satellite data and outputs standard NMEA data through an asynchronous serial interface. The module is suitable for uses in SLAP to provide position tracking and heading information.



NEO6m GPS

2.7.6 BMP280

The BMP280 is an environmental sensor that measures barometric pressure and temperature. It communicates via I2C synchronious serial interfaces. In SLAP, it provides atmospheric data that could be used for weather monitoring and logging during sailing trips.



BMP280

2.8 Objectives and Requirements

The primary objective of SLAP is to provide a reliable auto-pilot system for small sailing vessels that can maintain a steady course while allowing the sailor to attend to other tasks. The system is to be contructed from available low cost components including a tiller actuator (servo motor) to interface with standard tiller actuator hardware and provide a user-friendly interface accessible via mobile devices.

The system must implement a suitable control algorithm that can automatically adjust the tiller position based on GPS heading data to maintain the desired course. The control parameters need to be configurable to accommodate different vessel characteristics, sailing conditions and must respond to enviromental forces which change the boats heading

SLAP requires a sensor system including GPS for position and heading data, and optionally addi-

tional sensors like wind direction, compass, and accelerometer. The sensor data must be logged at regular intervals for course plotting.

To enable system development and enable testing, the system must have a **simulate** mode for boat dynamics and enviromental disturbances to the boats heading. Without this, the system is not at a prototype level where testing on a real boat can be undertaken.

The user interface must be designed with large, clear controls. Key features include course display, manual course adjustment controls, and system status indicators. The interface should be web-based and accessible via WiFi connection to mobile devices.

A comprehensive data logging system is required to store trip details, sensor readings, and system performance metrics in a structured database. The logs should be uploadable to an online mapping service for review. The database needs to support multiple trips, sensor configurations, and control parameter sets.

2.9 Specific Objective

The following specific objectives have been derived from the problem statement, the user questionaire, the user story and the analysis of the Raymarine ST1000.

1. Maintain a steady course within ±5 degrees using a closed-loop control algorithm
2. Provide manual parameter tuning interface allowing configuration of the control system
3. Maintain autonomous control for a minimum duration of 10 minutes
4. Provide a simulation feature to enable development of a prototype to be tested on real boat
5. Implement a responsive web interface with large touch controls and information display
6. Log GPS positions (waypoints) for boat trips
7. Store trip data including start/end times in a trips table
8. Process sensor data and update tiller position at minimum 1Hz to maintain desired heading

3 Design

3.1 Overall System Design

The SLAP (Self Logging Auto Pilot) system is designed to provide automated steering control for small sailing vessels while maintaining detailed logs of each journey. The system consists of several interconnected modules that work together to meet the system requirements and project objectives.

At its core, the system uses a PID (Proportional-Integral-Derivative) control algorithm that continuously monitors the boat's heading and makes adjustments to maintain the desired course. This is accomplished via a GPS sensor for postition and heading. Additional sensors to detect sea and enviromental conditions. The sensor data is processed in real-time to calculate necessary rudder adjustments.

The user interface is implemented as a web application, allowing sailors to interact with the system through any mobile device with a web browser. This interface provides visualisation of key data like current heading whilst also allowing manual control inputs when needed. Communications between the web app and the smart phone is usig the RP5 WiFi HotSpot and a HTTP based web API

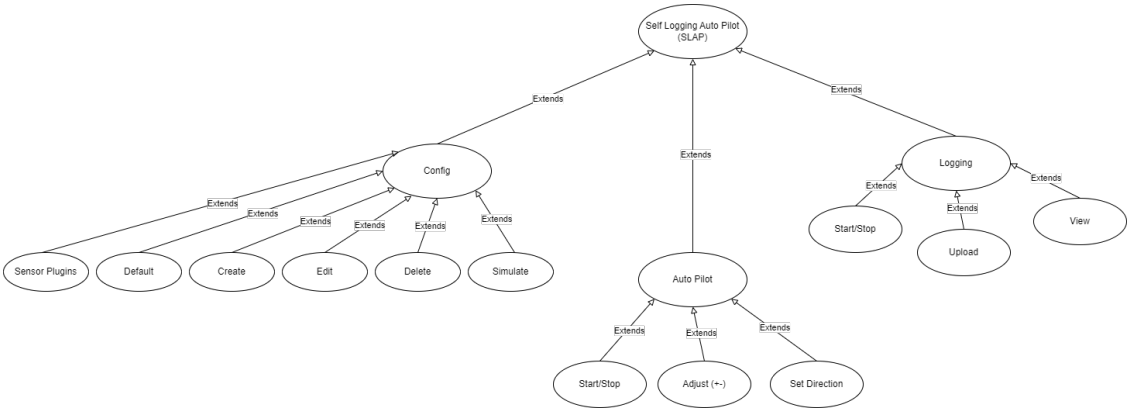
Data logging is a key feature of the system, with detailed records of each journey stored in a database. The logs capture all sensor readings, including GPS position, motion sensor data, and

heading. The logged data can be uploaded to a cloud based mapping application.

The system is designed to be reliable, it operates autonomously while still giving the sailor full control when needed.

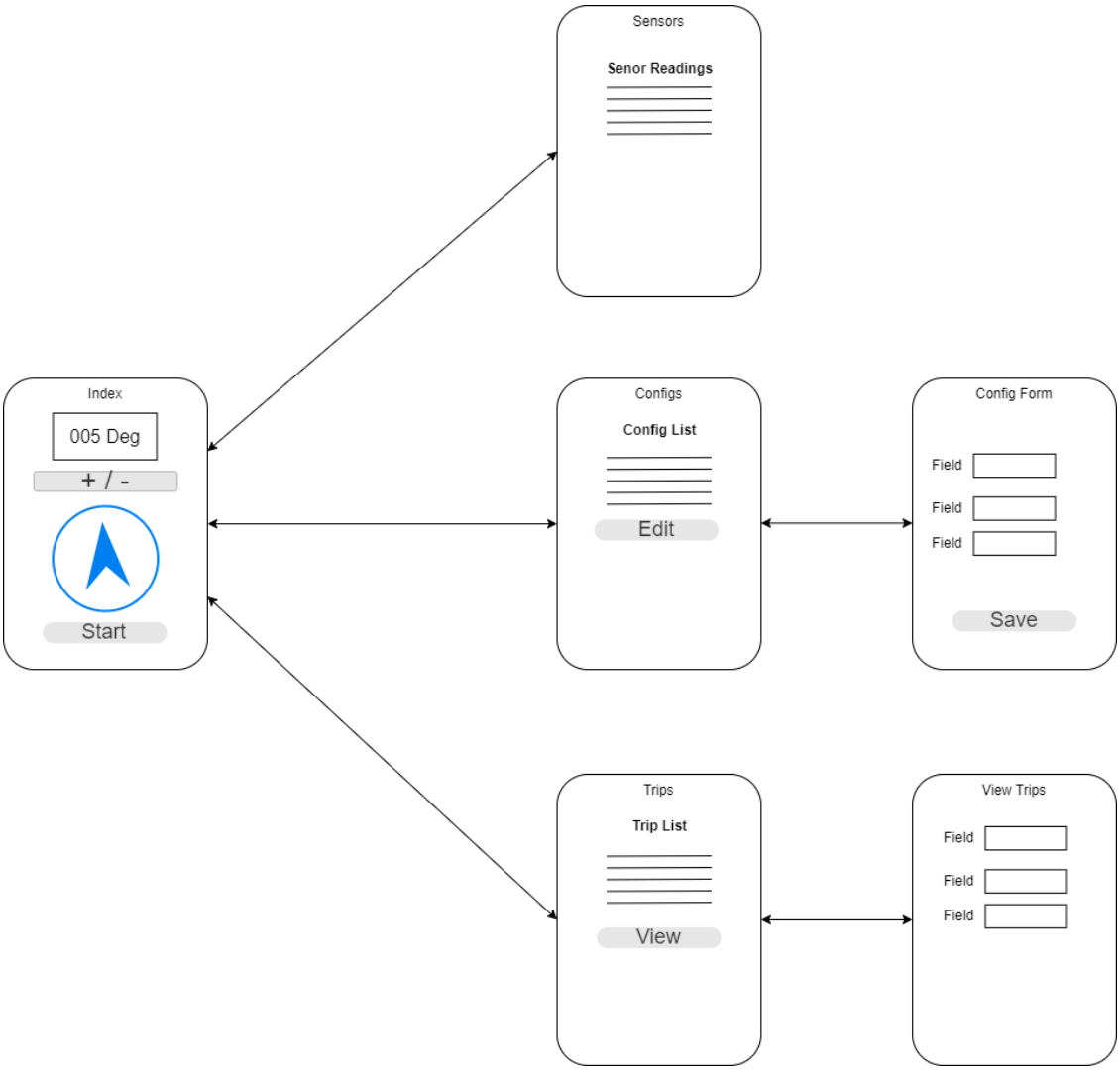
3.2 Use Cases

Based on the user story and system objectives the necessary use cases to meet the overall requirements have been designed. These use cases are shown in the diagram below:



3.3 UI Navigation and Design

Bases on the user story and use cases above, the following user interface structure was designed



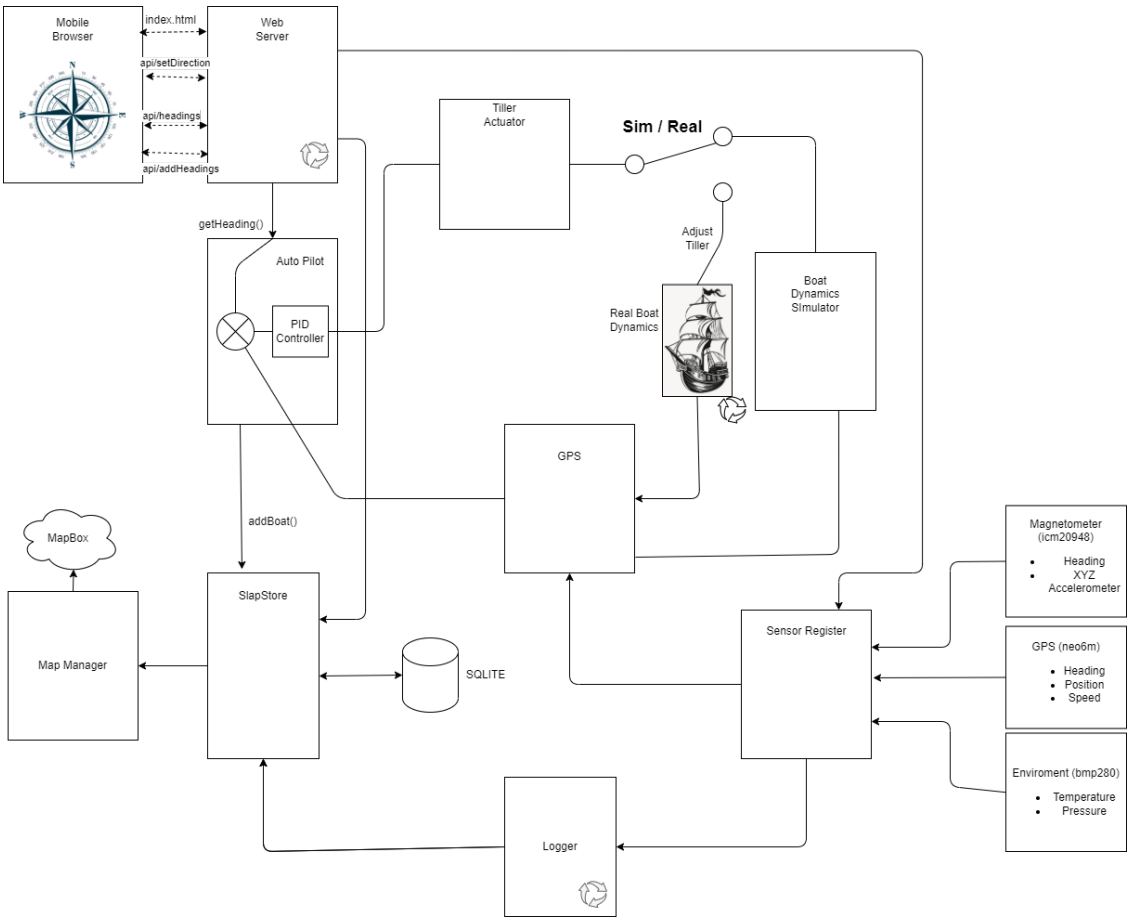
3.3.1 UI Design

Index A central value gives the angle of the target heading in degrees between 0 and 359. Central at the bottom is a set button which begins the Auto Pilot which will try to keep the boat on course. Another button will start the logging. The stop button will stop the Auto Pilot control loop. The simulate button will engage the boat sim. Manual adjustments can be made with the buttons at the top, allowing for both fine and broad adjustments. The menu button in the top left will give you access to the other pages.

3.4 Modular Design

The system architecture follows a modular design, each module is designed to provide a clear set of functions. The webserver module, running MicroPython on an Rasberry Pi or similar microcontroller, manages the file system and handles communication between the user interface and control systems. A dedicated PID module executes the main feedback control loop, receiving setpoint adjustments from the webserver when manual changes are made.

Modular Design Diagram:



3.5 Module Descriptions and Interconnections

The SLAP system consists of modules that work together to provide automated steering control. Each module has specific responsibilities and functions. I have tried to make sure that related responsibilities have not been split between modules [reference]. The modules are described below. The code in the Technical Solution section is organised along the lines of these modules,

3.5.1 Auto Pilot Module

- **PID Controller**: The central control module that implements the proportional-integral-derivative control algorithm
- **Input**: Current heading, desired heading, sensor data
- **Output**: Rudder position commands
- **Interfaces**:
  - Receives sensor data from Sensor register
  - Sends control commands to Tiller Actuator
  - Receives target heading updates from Web Interface

3.5.2 Boat Sim Module

- **Components**: Virtual Boat Dynamics, Simulated geographic position
- **Responsibilities**: For testing purposes by mimicing the behaviour of a boat

- **Interfaces**:
  - Recieves rudder angle
  - Calculates new heading and position
  - Provides data to GPS (in sim mode)

3.5.3 Sensor Register Module

- **Components**: GPS, Motion Sensors, Eviroment Sensors
- **Responsibilities**: A register of all the hardware transducer modules and the sensors that these contain
- **Interfaces**:
  - Sends sensor data to Auto Pilot Controller
  - Keeps the GPS module updated
  - Provides data to Logging Module
  - Communicates with Web Interface for display

3.5.4 GPS Module

- **Components**: Register of GPS values
- **Responsibilities**: To provide information about the boats position and heading
- **Interfaces**:
  - Collects data from either a GPS Hardware Module, the Magnetometer or the boat simulator
  - Supplies modules which require heading and positional data

3.5.5 Tiller Actuator Module

- **Components**: Servo Motor Controller
- **Responsibilities**: Converts the tiller angle into signals for the servo motor
- **Interfaces**:
  - Receives tiller angle signal from PID Controller
  - Provides feedback to Web Interface

3.5.6 Web Interface Module

- **Components**: Flask Server, Javascripts, HTML Template for each page, style sheets
- **Responsibilities**: User interaction for system control and monitoring
- **Interfaces**:
  - Sends manual control inputs to Auto Pilot
  - Receives data from all sensors
  - Displays system status

3.5.7 Logging Module

- **Components**: Log Time Keeper
- **Responsibilities**: Collecting data and writing to database
- **Interfaces**:
  - Receives data from all sensors
  - Talks to SlapStore database service to store logs

3.5.8 SlapStore Module

- **Components**: Database connection, Table Definitions, Storage Service Layer
- **Responsibilities**: Reading and writing to the database, keeping the database details hidden from the rest of the program
- **Interfaces**:
  - Connection to SQLite database
  - Storage services for all parts of the system#

3.5.9 Mapping Module

- **Components**: Map Manager, Cloud interfacing
- **Responsibilities**: Read log data and write to the cloud mapping service
- **Interfaces**:
  - Receives data from SlapStore service layer
  - Talks to MapBox to upload trip logs

3.5.10 Real-Time Behaviour

It was not possible to make the system function properly with a single loop where each componant is run in sequence. Therefore the system makes use of a number of execution threads. Each componant which needs to execute in its own loop is given a dedicated thread. The choice of which modules required a thread was based on the real world functions required.

The threaded modules are: - Boat sim - All sensors - Logger - Web server

3.5.11 Communication Flow

1. Sensor Modules continuously collects data
2. The auto Pilot receives heading data and calculates control commands using the PID module
3. Actuator Module recieves the control signal from the auto pilot and controls the servo motor
4. Logging Module records all sensor data in a regular time interval
5. Web Interface recieves user interactions and displays system status
6. Web Server, recieves interactions from the interface and sends them to the system
7. The Map Manager takes the trip log and sends it to the cloud

3.6 Data and Code Design

3.6.1 Data Dictionary

The following data items are used throughout the system:

Data Item	Description	Format	Example
Heading	The current compass direction the boat is pointing	Integer (0-359 degrees)	180
Target Heading	The desired compass direction for the boat	Integer (0-359 degrees)	175

Data Item	Description	Format	Example
Heading Error	Difference between current and target heading	Integer (degrees)	-5
Position	Current location of the boat	Tuple (latitude, longitude)	(50.7192, -1.8808)
Rudder Position	Current rudder angle	Integer (-45 to +45 degrees)	-10
Servo Output	Signal sent to rudder servo	Integer (-100 to +100)	25
Log Time	Timestamp for data logging	DateTime	2024-01-20 14:30:00
Trip ID	Unique identifier for each journey	Integer	1234

3.6.2 Primary Sensor Data

The system uses the following sensors: - BNO055 9-axis IMU sensor for: - Heading data (compass) - Roll and pitch angles - Acceleration - NEO-6M GPS module for: - Position (latitude/longitude) - Ground speed - Course over ground - DS18B20 temperature sensor for: - Water temperature monitoring - INA219 current/voltage sensor for: - Battery voltage monitoring - Current draw measurement

3.7 Data Stucture Definitions

3.7.1 Database Design

The diagram below represents the database entities in the system, their relationships and the multiplicities.

erd images/image_1_erd.drawio.png DIAGRAM

The database consists of four main tables:

Config Table: - Contains parameters used to tune the control algorithm - Each parameter has a name, value and description - Parameters include PID control values and other tuning constants - Multiple config sets can be stored for different conditions

Sensors Table: - Maintains a registry of all sensors connected to the system - Each sensor has a unique Identifier, name, type and calibration data - Sensor types include GPS, compass, wind speed/direction, accelerometer etc. - Status field tracks if sensor is active/inactive

Trips Table: - Records details of each sailing trip/passage - Contains start/end times, start/end locations - Stores target heading and actual course taken - Links to config settings used for that trip

Readings Table: - Stores all sensor readings during trips - Each reading links to a specific trip and sensor - Includes timestamp and raw sensor value - One reading record per sensor per measurement interval

3.7.2 Data Validation

Data entered into input fields such as editing configs must be validated to ensure they are of the of the correct format and type. Being an active with multiple hardware sensors, most of the data in the system is recieved automatically. The limited user input data is validated in the user interface. The following user interfaces capture data and therefore be validated as indicated below:

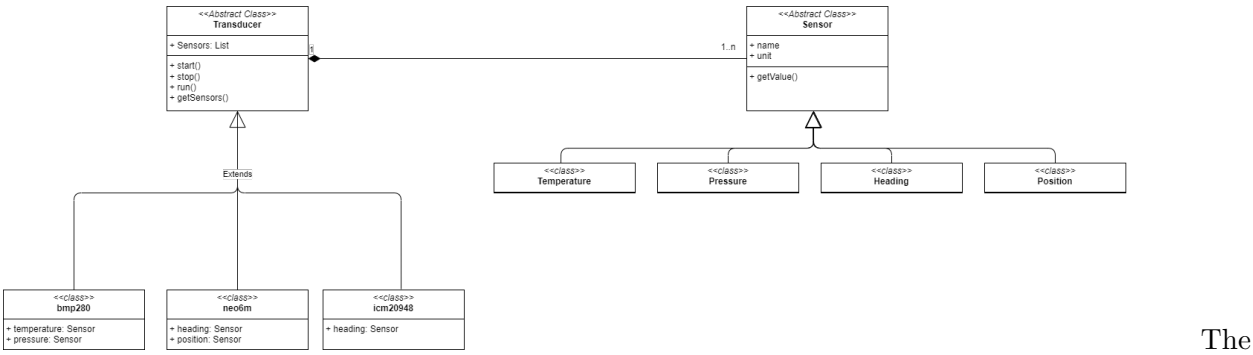
Form	Input Field	Data Type	Valid Range/Format	Description
Index	Set Heading	Integer	0 to 359 degrees	Desired boat heading
Edit Config	Config Name	Text	15 characters max	Name of configuration
Edit Config	P Value	Float	Positive decimal	Proportional control value
Edit Config	I Value	Float	Positive decimal	Integral control value
Edit Config	D Value	Float	Positive decimal	Derivative control value

Other user input data does require validation as it is contrained by the input method. For example + / - 1 or 10 degree buttons. These inputs do not allow invalid data input

3.7.3 Object Oriented Programming

The design of the code for using hardware modules each with a number sensors is complex. The following design was arrived at after a number of iteration. It is explained here due to its complexity. To achieve a clean code solution a class inheritance pattern is utilised. Also a sensor [builder pattern] is used.

The diagram below shows the object oriented class structures used:



The structure uses two abstract base classes. This allows to work with specific Tranducer Modules and their associated Sensors without needing to know their specific details.

The `Transducer` abstract class represents hardware devices. Note that each hardware device can contain a number of actual sensors.

The Transducer class has:

- A list of sensors
- Thread management so that the readings from the hardware can be collected in real-time

The `Sensor` abstract class represents the sensors in each hardware (Transducer) module, including:

- basic sensor properties (name, units);
- and, getValue method to retrieve sensor reading;

which are used by other parts of the system to hide the specifics of each sensor.

Specific sensor class which inherit from `Sensor` and represent real sensors:

- `heading`: The direction of travel in degrees
- `position`: The longitude and latitude
- `temperature`: The enviroment's temperature
- `pressure`: Air pressure

This object oriented class structure allows the rest of the system to handle sensor information without having to handle their specific types. It allows a register of sensors to be created.

The implementations of the Transducer classes contain the code necessary to interface the hardware with the RPi5. The implementation of the Sensor classes defines the nature of all the sensors in the system.

3.8 Algorithms

There are some key algorithms within the system which required some design. The control algorithm was first investigated as part of the Analysis section.

3.8.1 Boat Simulator

The boat simulator allows SLAP to run without the need of a real boat, this is necessary for test and development purposes, and understanding the boat dynamics and creating modelling code is a key part of the project. Without the simulator objective of a prototype system ready for testing on a real boat cannot be met.

There are two element to the boat simulator.The first is the mathmatical model which represents the boats response to the tiller. This is characterised by calculating the new heading using all the previous information about the boats heading. The dynamics of this turning behaviour are that the boat begins to turn quickly, as the boat turns its rate of turn slows until a steady rate is reached. This is a first order differential response. i.e. the rate of change of turn is proportional to the difference (differential) between the current and eventual rate of turn.

The second element of the boat modeling is the disturbances to the boats heading due to the enviroment. The boat simulator would not be realistic without this as the control algorithm would simply and quickly achieve the correct heading. The boat simulator model therefore randomly generates disturbances which affects the boats heading and forces the Auto Pilot control algorithm to react.

Mathmatics

In order to control the movement and direction of the boat we must first be able to model how the rudders angle affects the boats direction.

We understand that the rudders angle and boat's direction do not respect a linear relationship, if we represent the rudder's angle with θ, and the turning radius of the boat as R we can see that: $R = \frac{1}{K|\theta|}$ where K is some constant .

This inversely proptional relationship is proven as we know as the rudders angle increases, the turning radius decreases. The rate of which the boats heading changes, also known as the angular velocity, ω, can be represented as the differential of the angle and time: $\frac{dA}{dt}$ or as the velocity over the radius: $\frac{v}{R}$.

Therefore $\omega = \frac{v}{(K|\theta|)^{-1}}$, and so:

$$\omega = vK|\theta|$$

We can simply intergrate this to find the change in angle over time to give :

$$\Delta\omega = vK|\theta|t$$

However we understand that the boat does not turn to the target radius immediately, there is always some lag between turning the rudder and the target radius being reached.

The turn rate follows an exponential decay until it reaches the target, the standard decay equation is $N_{t+1} = N_t - e^{-\lambda t}$

Where lambda is our decay constant and t is our discrete time value.

This can be implemented with our boat applicable values:

$$\omega(t) = \omega_\infty \left(1 - e^{-t/\tau}\right)$$

This algorithm is implemented in code as descibed in the Technical Solution section(boatSim.py)

Disturbances

The disturbance algorithm randomly picks a start time by to the current time. Its duration is determined by repeating this process but using the start time. When a disturbance is inflicted on the simulator its rate of turn is increased or decreased by a random magnitude. All random values are determined within a realistic range to mimic the affect of sea conditions on a boat.

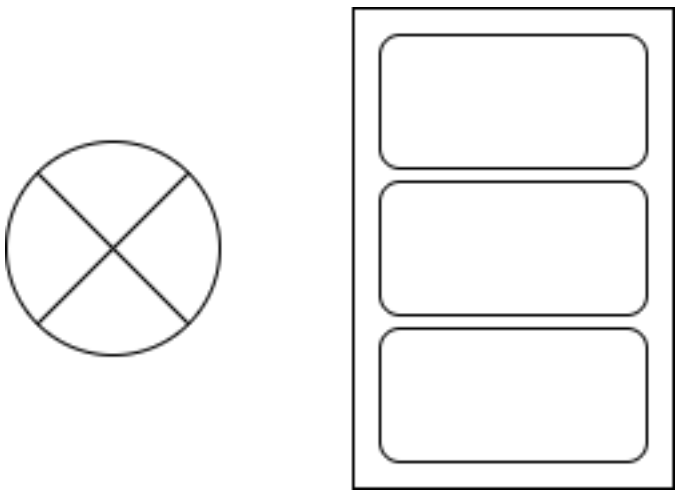
The following code illustrates the disturbance algorithms

[1]:
```
TIMECONSTANT = 0.333333
MAX_DISTURBANCE_DURATION = 5000 # ms
MIN_DISTURBANCE_DURATION = 2000 # ms
MAX_DISTURBANCE_MAGNITUDE = 20 # degrees per secon


# Create a disturbance based on a random start time, duration and magnitude
def createDisturbance(self):
        currentTimeMilli = int(round(time.time() * 1000))
```

```
        disturbance =  random.randint(-MAX_DISTURBANCE_MAGNITUDE,␣
↪MAX_DISTURBANCE_MAGNITUDE)

        startTime = currentTimeMilli + random.randint(␣
↪MAX_DISTURBANCE_DURATION, 2 * MAX_DISTURBANCE_DURATION)

        endTime = startTime + random.
↪randint(MIN_DISTURBANCE_DURATION,MAX_DISTURBANCE_DURATION)

        output = {'endTime': endTime,
                'disturbance': disturbance,
                'startTime': startTime}

        return output

# Return the disturbance if it is currently active, otherwise create a new one
def disturbance(self):
        currentTimeMilli = int(round(time.time() * 1000))
        if (currentTimeMilli < self.nextDisturbance['startTime']):
            return 0
        elif currentTimeMilli > self.nextDisturbance['startTime'] and␣
↪currentTimeMilli < self.nextDisturbance['endTime']:
            return self.nextDisturbance['disturbance']
        else:
            self.nextDisturbance = self.createDisturbance()
            return 0
```

3.8.2 PID Control Algorithm

The PID control algorithm allows SLAP to adjust the tiller angle to correct its course usiong the difference between its actual heading and the desired heading. During the Analysis it was discovered that a control algorithm known as PID was well suited to this project. The figure below shows the PID algorithm in diagrammitic form:



Mathmatics

The PID (Proportional, Integral, Derivative) control algorithm is a feedback control system that calculates an error value as the difference between a desired setpoint and a measured process variable. The algorithm applies three types of control:

1. Proportional (P): Responds proportionally to the current error
2. Integral (I): Accumulates past errors to eliminate steady-state error
3. Derivative (D): Predicts future error based on its rate of change

The output is calculated as: $output = K_p e_t + K_i \int e_t dt + K_d \frac{de}{dt}$

Where: - K_p, K_i, K_d are tuning parameters - e_t is the error at time t - The integral term sums past errors - The derivative term predicts future errors

This algorithm is implemented in SLAP to maintain the boat's heading by continuously adjusting the rudder angle based on the difference between the desired and actual heading.

The PID method shown in the code below and was used in the Analysis section to investigate suitable control algorithms. The full PID algorithm used in the SLAP is detailed in the Technical Solution section.

[2]:
```
# %load slap/src/pid/experimentForReport/pidModule.py

class PidController:

    def __init__(self,KP,KI,KD):
        self.kp = KP
        self.ki = KI
        self.kd = KD
        self.accumlatedError = 0
        self.lastPos = 0
        self.elapsed = 0

    def pid(self, pos, target, dt):

        # PROPORTIONAL-------
        error = target - pos
        proportional = error

        # Intergral----------
        intergal = ((error * dt) + self.accumlatedError)
        self.accumlatedError = intergal


        # Differential-------
        dpos = self.lastPos - pos
        differential = (dpos / dt)

        self.lastPos = pos
```

```
        return self.kp * proportional + self.ki * intergal + self.kd *␣
↪differential
```

3.8.3 Working with Compass Headings

There are complications in the arithmatic relating to compass headings. This is due to the fact a compass functions on a 360 degree basis. For example the difference between a heading of 5 degrees and 355 degrees should be -10 degrees and 355 degrees + 10 degrees must equal 5 degrees not 365 degrees.

When calculating which way to adjust the tiller as a part of the control algorithm, this arthmetic complication is important otherwise the boat will attempt to correct its course the wrong way around. The algorithms for these calculations are shown in the code snippets below.

The basic approach to dealing with compass headings is to use modulo 360 arithmetic. Secondly when calculating differences for control we must find the smallest change in heading necessary.

[3]:
```
# Get the error between the target and heading in its shortest path
def getHeadingError(self, target, heading):
    if target - heading <= 180:
        error = target - heading
    else:
        error = (target - heading) - 360
    return error

# Convert a heading to a value between 0 and 359
def compassify(input):
    print("Compassing: ", input)

    return input% 360
```

3.9 Security

In designing the security for this project, the first things to consider are the risks and the threats.

The main risk identified is the boat steering incorrectly. The main threat is the system preforming incorrectly. To reduce this risk, the system must be tested rigorously.

The second thread identified is inappropiate access to the system. If the system were connected to the internet, during operation, the risk would be significant. However, on a boat with SLAP connected to a smart phone via a WiFi HotSpot the oppotunity for an unorthorised user to access the system is very small. This is because access is controlled by the WiFi security on the RPi5 HotSpot.

3.10 Testing Strategy

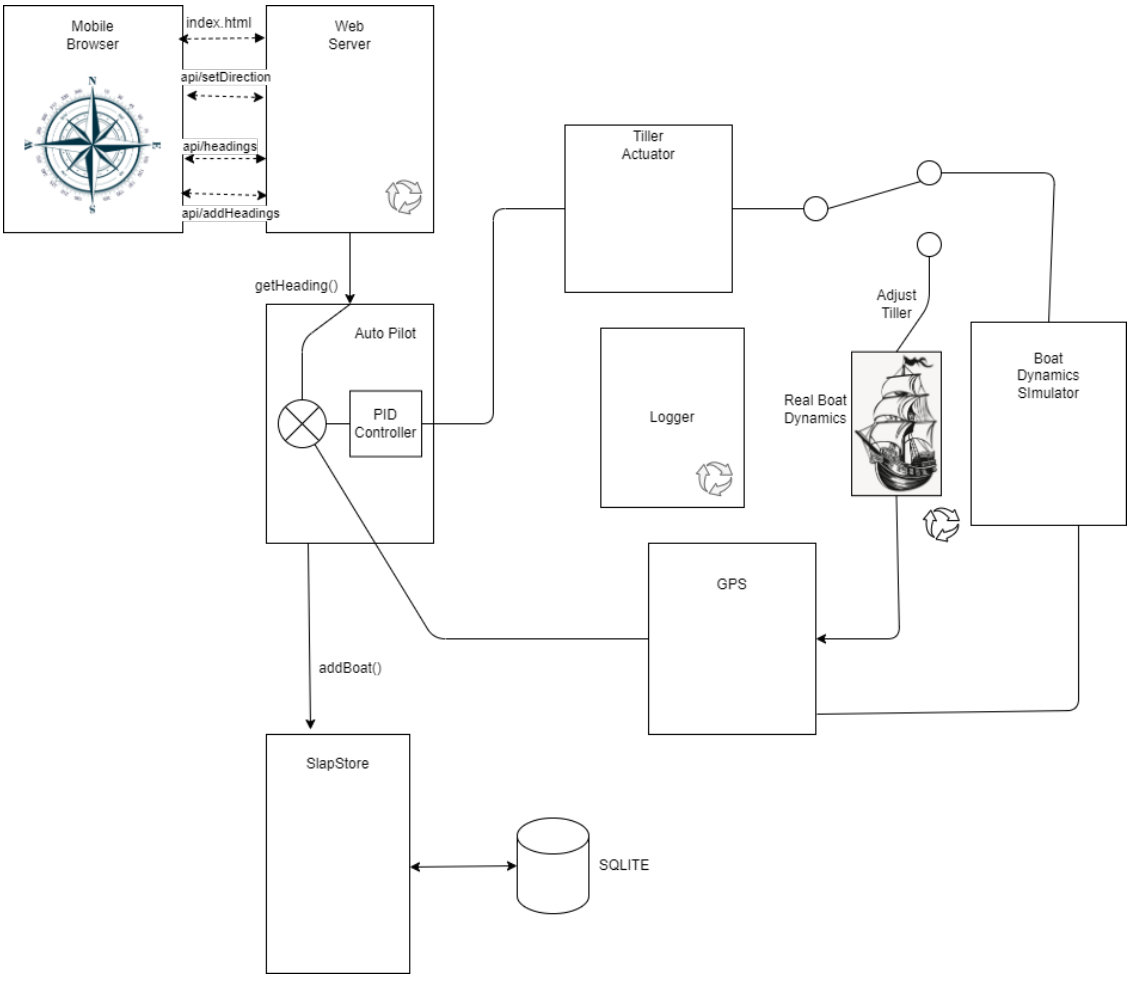
section on testing based on manual and unit testing. Tests necessary for each use case in the system Tests necessary for the project objectives (evaluation?)

4 Technical Solution

4.1 Introduction

In this section the SLAP technical solution is explained. Sail Logging Auto Pilot (SLAP) is a real time control system which uses a GPS heading as the basis for steering a sailing boat. The steering of the boat is achieved using an actuator (servo motor) to adjust the tiller position. The skipper of the boat activates the auto pilot and the system uses an active control loop to maintain the boats heading.

The design of the system is fully explained in the design section. This section details the implementation of this design (Technical Solution). The code for SLAP is carefully structured inline with the modular design. To aid understanding the diagram below is repeated from the Design section as it provides a helpful index to the code. The diagram has a close correspondance to the code structure. This technical solution section explains the system's architecture and each of the modules.



4.1.1 Modes of Operation

SLAP has two modes of operation:

In **Simulator Mode** the boat's dynamics (how the boat's heading changes over time in response

to the tiller action) is mathmatically modelled.

In **Run Mode** the tiller actuator affects the boats heading according to the actual characteristics of the boat which is reflected in the GPS readout.

The diagram above illustrates how the system switches between these modes. The controller, in turn, links the tiller actuator to the GPS via a PID controller system. A database is used to store the system's configuration and to act as a logger recording the detail of each trip.

4.2 Overview

The following sections describe each module in outline, full details of their operation and coding are provided below.

4.2.1 Web Browser

This is the user interface which is the web browser on the user's smartphone. The user connects to the local WiFi HotSpot on the boat, provided by the Rasberry Pi where the SLAP system is running.

4.2.2 Web Server

In early the experiments the Web Server was coded using python function for HTTP handling. This was complex and required a lot of code. In the final iteration, the Web Server code is built using the Flask Python library. This allows HTTP EndPoints to be more easily coded. The Web Server includes has two primary functions. The first is to serve HTML pages and the second is to provide HTTP API EndPoints.

4.2.3 Controller

This is the central coordinating module in the system. It recieves heading data in real time from the GPS, which is forwarded into the PID controller. The output of the PID then drives the tiller actuator module. The controller includes the Proportional, Integral and Differential (PID) feedback system. This PID controller attempts to adjust the boat's heading to minimise the difference between the actual heading and the target heading as set by the skipper.

4.2.4 GPS

The GPS module is a real time system providing a continous update of the boats heading. The GPS module is connected to a hardware GPS board that recieves saterlite data. In 'Simulator Mode' the GPS module returns a heading provided by the boat dynamics simulator module, rather than the actual GPS. The GPS module gets data from actual GPS hardware via the sensor register.

4.2.5 Tiller Actuator

This module recieves tiller angle settings from the PID controller and activates a connected servo motor to drive the tiller. In Simulator Mode this module drives the boat dynamics simulator. The tiller servo motor uses a PWM (Pulse Width Modulated) signal to set the motors position.

4.2.6 Boat Dynamics Simulator

This module is used in Simulator Mode to represent the boat's behavour. It is based on the heading changing in response to the tiller action. The mathmatics of this model is based on a first order differential equation and is detailed in the Design section.

4.2.7 SLAP Store

This module is an service interface which interacts with the database. The store is responsible for persisting the system configuration and for logging details of the boat's track (a set of waypoints). The SlapStore module provides an API to the other system modules to hide all database details such as database connections and SQL complexity.

4.2.8 Sensor Register

The Sensor Register is a module which provides access to a list of sensors within the system. The register is filled with objects of the class Tranducer, each containing their sensors. The Sensor Register provides an API to access all sensor details within the system.

The Sensor Register and its related classes use object oriented code patterns including abstract classes and inheritance. This pattern is explained in the Design section

4.2.9 Map Manager

This module is responsible for uploading Trip logs to the MapBox cloud based mapping system. It takes data from the logging module, converts it to JSON, connects to MapBox via a web interface, authenticates with the service and makes an upload

4.2.10 Logger

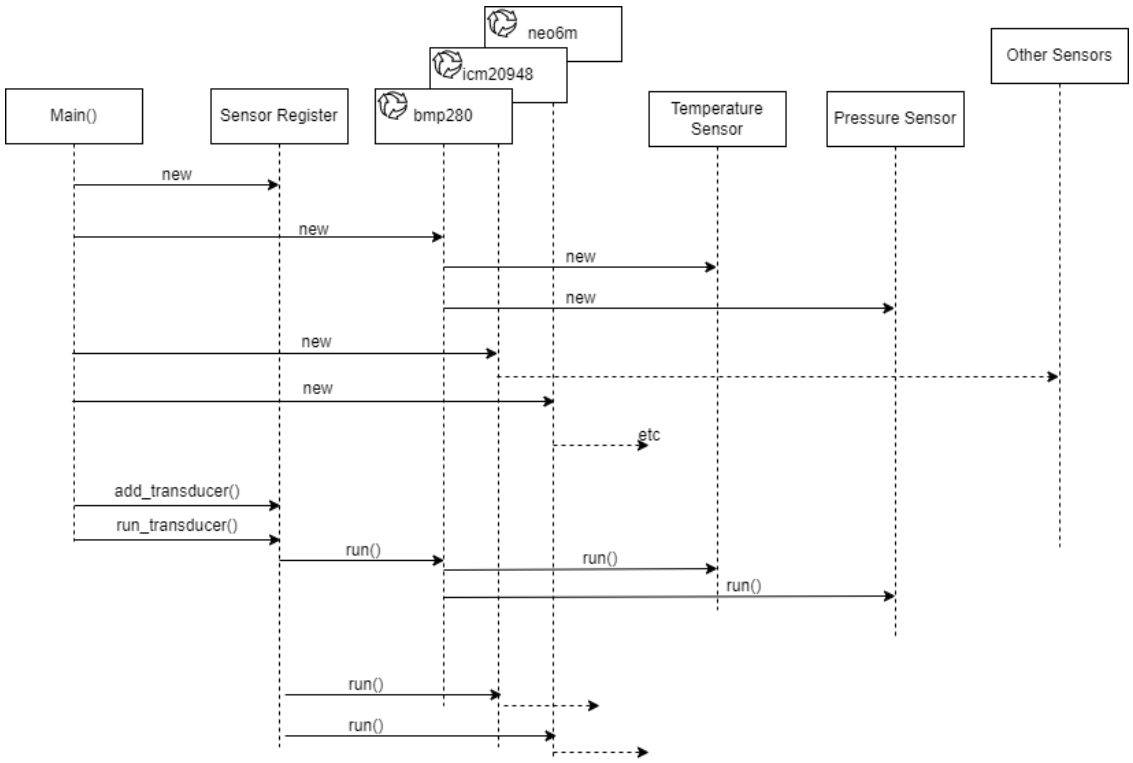
The logger module is responsible for writing the latest system data to the database at regular intervals. It runs in its own thread and interacts with the SlapStore Service Layer to write logs to the database.

4.3 Threading and Module Interaction

The following diagrams represent the interaction between the key modules in the system. It also illustrates which modules run in their own dedicated threads. These diagrams read alongside the system architecture diagram is key to understanding the code and its operation.

4.3.1 Sensor Register

The diagram below shows how the main.py firstly creates the Sensor Register, then adds the transducers which in turn creates and their sensors. When the sensor register is run, it asks runs each Transducer in a new thread. These threads continously obtains values from the Transducers hardware and sets the values on each of its sensors. This is shown in the sequence diagram below. Not all of the objects are shown simplicity.



4.3.2 Main Modules

The main.py contructs each of the SLAP modules. These being TillerActuator, AutoPilot, Gps, BoatSim, MapManager, Logger and SlapStore. These modules are then wired together as seen in the System Architecture diagram. The main finally creats and starts the Web Server in a seperate thread.

4.4 Code Packages

The following sections detail the package of code which form the each modules including their operation and the code itself.

4.4.1 File Structure:

The following file structure is organised inline with the modules described above.

```
iteration2
├── control
│   ├── __pycache__
│   ├── __init__.py
│   ├── autoPilot.py
│   ├── boatSim.py
│   ├── pidModule.py
│   ├── simpleModel.py
│
├── services
│   ├── __pycache__
│   ├── slapStore.py
│   ├── slapStoreTest.py
│
├── transducers
│   ├── __pycache__
│   ├── gps.py
│   ├── tillerActuator.py
│
├── utils
│   ├── __pycache__
│   ├── nmea
│   ├── __init__.py
│   ├── liveCompass.py
│
├── web
│   ├── __pycache__
│   ├── static
│   ├── templates
│   ├── app.py
│
└── main.py
```

4.4.2 Main.py

This code is the entry point for the program's execution. It starts by instantiating each of the main classes in the system in turn. When a class is created the necessary dependancies are passed in. These dependancies are the solid arrows shown in the system diagram (figure 1). Instantiating the main classes and passing them into the other classes ensures only a single instance of each is

created.

Being a real time system SLAP has a number of program threads running simultanously as explained above. main.py as shown below is responsible for starting the various execution threads within the system. These are the Web Server and the GPS.

[ ]:
```
# %load slap/src/iteration2/main.py
#                                Project Entry Point                                #
from services.mapManager import MapManager
from services.slapStore import SlapStore
from control.autoPilot import AutoPilot
from control.boatSim import BoatSim
from web.app import WebServer
from transducers.gps import Gps
from transducers.tillerActuator import TillerActuator
from services.logger import Logger
import threading
import time
import atexit
from transducers.sensorRegister import SensorRegister
from transducers.bmp280Transducer import Bmp280Transducer
from transducers.neo6mGps import Neo6mGps
from transducers.icm20948Magnetometer import ICM20948Magnetometer

class Main():

    def __init__(self):

        # Produce an instance of the sensor register and add transducers to it
        self.sensor_register = SensorRegister()
        bmp280 = Bmp280Transducer()
        gps = Neo6mGps()
        magnetometer = ICM20948Magnetometer()
        self.sensor_register.add_transducer(bmp280)
        self.sensor_register.add_transducer(gps)
        self.sensor_register.add_transducer(magnetometer)
        # Start the transducers in separate threads
        self.sensor_register.run_transducers()

        # Init an instance of the SLAP modules
        self.tiller_actuator = TillerActuator()
        self.auto_pilot = AutoPilot()
        self.gps = Gps()
        self.boat_sim = BoatSim(self.sensor_register, self.gps)
        self.map_manager = MapManager()
        self.logger = Logger(self.gps, self.map_manager, self.sensor_register)
        self.store = SlapStore("slap.db")
```

```
        # Link classes together
        # Ensuring only one common instance of each module is used
        self.gps.setAutoPilot(self.auto_pilot)
        self.auto_pilot.setTillerActuator(self.tiller_actuator)
        self.tiller_actuator.setBoatSim(self.boat_sim)
        self.logger.setStore(self.store)


    def main(self):

        # Start the boat in real mode in a new thread
        self.boat_sim.stopSim()

        # Add each sensor to the database
        for sensor in self.sensor_register.getSensors():
            self.store.addSensor(sensor)

        # Ensure the controller stops when the application exits
        atexit.register(self.boat_sim.stopSim)

        # Create Flask web server
        webserver = WebServer(self.auto_pilot, self.logger, self.
 ↪sensor_register, self.boat_sim, self.gps)
        app = webserver.create_server(self.store)

        # Start the web server in a new thread
        app.run(debug=True, use_reloader=False, port=5000, host='0.0.0.0')


# This is where the application starts executing
if __name__ == '__main__':
    main = Main()
    main.main()
```

4.5 Transducers, Sensors and the Sensor Register

4.5.1 Sensor Register

The Sensor Register as explained above holds a list of the Transducers, each with their respective Sensors. The register is responsible for starting all the Transducer threads.

[ ]:
```
# %load slap/src/iteration2/transducers/sensorRegister.py
from transducers.transducer import Transducer

class SensorRegister:
    def __init__(self):
```

```
        self.transducers = []

    #Defines service methods for the sensor register

    def add_transducer(self, transducer: Transducer):
        self.transducers.append(transducer)

    def get_transducers(self):
        return self.transducers

    #Starts Trandsucers in separate threads
    def run_transducers(self):
        for transducer in self.transducers:
            transducer.start()

    #Stops transducer Threads
    def stop_transducers(self):
        for transducer in self.transducers:
            transducer.stop()

    def getSensorReadings(self):
        sensors = []
        for transducer in self.transducers:
            for sensor in transducer.getSensors():
                output = {
                    "identifier": sensor.getIdentifier(),
                    "name": sensor.getName(),
                    "value": sensor.getData(),
                    "units": sensor.getUnits()
                }
                sensors.append(output)
        return sensors

    def getSensor(self, name):
        for transducer in self.transducers:
            for sensor in transducer.getSensors():
                if sensor.getName() == name:
                    return sensor
        raise Exception("No sensor found with name: " + name)

    def getSensors(self):
        sensors = []
        for transducer in self.transducers:
            for sensor in transducer.getSensors():
                sensors.append(sensor)
        return sensors
```

Transducers This package contains the Transducer code. There are three Transducers, the bmp280, ICM20948, and the NOE6M GPS.

The Transducer classes create the Sensors in each hardware module, and contain the code to interface the hardware with the Raspberry Pi.

The interfacing hardware code for each module was obtained for internet examples.

bmp280 Transducer

The bmp280 contains a temperature and pressure sensor.

[ ]:
```
# %load slap/src/iteration2/transducers/bmp280Transducer.py
try:
    from smbus2 import SMBus
    from bmp280 import BMP280
    IS_RPI = True
    print("Running on a Raspberry Pi")
except (ImportError, ModuleNotFoundError, RuntimeError):
    print("Not running on a Raspberry Pi")
    IS_RPI = False

from transducers.transducer import Transducer
import time
import threading
from transducers.sensor import Sensor

class Bmp280Transducer(Transducer):
    def __init__(self):
        super().__init__()  # This calls the parent class's __init__

        # Create the Sensors provided by this hardware
        self.temperature = Sensor(self, "temperature", "Temperature", "°C")
        self.pressure = Sensor(self, "pressure", "Pressure", "hPa")
        self.sensors = [self.temperature, self.pressure]
        self.running = False
        # Checks if code is running on a Raspberry Pi
        # If so, initializes the BMP280 hardware
        if IS_RPI:
            self.bus = SMBus(1)
            self.bmp280 = BMP280(i2c_dev=self.bus)


    def run(self):
        # Main thread loop that continuously reads pressure
        while self.running:
            try:
                if IS_RPI:
                    self.pressure.setData(str(self.bmp280.get_pressure()))
```

```
                    self.temperature.setData(str(self.bmp280.get_temperature()))
                else:
                    self.pressure.setData("1013.25")  # Standard atmospheric␣
↪pressure in hPa
                    self.temperature.setData("20")
                #print(f"{self.name}, Value: {self.getData()}")
                time.sleep(0.1)  # Read pressure every 100ms
            except:
                print("Error reading pressure sensor")
                time.sleep(1)  # Wait longer on error
```

icm20948 Transducer

The icm20948 provides a three axis magnetometer and a three axis accelerometer. In SLAP we currently only utilise the magnetometer to calculate a heading

[ ]:
```
# %load slap/src/iteration2/transducers/icm20948Magnetometer.py
import math
import time
from transducers.transducer import Transducer
from transducers.sensor import Sensor
try:
    from icm20948 import ICM20948
    IS_RPI = True
except Exception as e:
    print(f"Error loading magnetometer: {e}")
    IS_RPI = False



class ICM20948Magnetometer(Transducer):
    def __init__(self):
        super().__init__()
        self.heading = Sensor(self, "magheading", "MagHeading", "°")
        self.sensors = [self.heading]
        self.running = False

        if IS_RPI:
            self.imu = ICM20948()

    def run(self):
        if IS_RPI:
            X = 0
            Y = 1
            Z = 2
            AXES = Y, Z
            amin = list(self.imu.read_magnetometer_data())
            amax = list(self.imu.read_magnetometer_data())
```

```
        while self.running:
            if IS_RPI:
                # Read the current, uncalibrated, X, Y & Z magnetic values from␣
↪the magnetometer and save as a list
                mag = list(self.imu.read_magnetometer_data())

                # Step through each uncalibrated X, Y & Z magnetic value and␣
↪calibrate them the best we can
                for i in range(3):
                    v = mag[i]
                    # If our current reading (mag) is less than our stored␣
↪minimum reading (amin), then save a new minimum reading
                    # ie save a new lowest possible value for our calibration␣
↪of this axis
                    if v < amin[i]:
                        amin[i] = v
                    # If our current reading (mag) is greater than our stored␣
↪maximum reading (amax), then save a new maximum reading
                    # ie save a new highest possible value for our calibration␣
↪of this axis
                    if v > amax[i]:
                        amax[i] = v

                    # Calibrate value by removing any offset when compared to␣
↪the lowest reading seen for this axes
                    mag[i] -= amin[i]

                    # Scale value based on the highest range of values seen for␣
↪this axes
                    # Creates a calibrated value between 0 and 1 representing␣
↪magnetic value
                    try:
                        mag[i] /= amax[i] - amin[i]
                    except ZeroDivisionError:
                        pass
                    # Shift magnetic values to between -0.5 and 0.5 to enable␣
↪the trig to work
                    mag[i] -= 0.5

                # Convert from Gauss values in the appropriate 2 axis to a␣
↪heading in Radians using trig
                # Note this does not compensate for tilt
                heading = math.atan2(
                        mag[AXES[0]],
                        mag[AXES[1]])
```

```
                # If heading is negative, convert to positive, 2 x pi is a full␣
↪circle in Radians
                if heading < 0:
                    heading += 2 * math.pi

                # Convert heading from Radians to Degrees
                heading = math.degrees(heading)
                # Round heading to nearest full degree
                self.heading.setData(heading)
                print("mag Heading: ", heading)

                time.sleep(0.1)
            else:
                self.heading.setData(0)
                time.sleep(1)
```

Neo6M GPS Transducer

This Transducer is a GPS reciever. It connects to GPS satelites and provides a range of information. In SLAP we only create sensors for heading, the longitude and latitude position.

[ ]:
```
# %load slap/src/iteration2/transducers/neo6mGps.py
from transducers.transducer import Transducer
from transducers.sensor import Sensor
import math
try:
    import serial
    import time
    import string
    import pynmea2
    IS_RPI = True
except Exception as e:
    print(e)
    IS_RPI = False


# This class is a transducer for the NEO-6M GPS module.
# It inherits from the Transducer class and provides methods to read GPS data.
# The class has two sensors: one for heading and one for position.
# It uses the pynmea2 library to decode NMEA 0183 protocol sentences.
class Neo6mGps(Transducer):
    def __init__(self):
        super().__init__()  # This calls the parent class's __init__
        self.heading = Sensor(self, "gpsHeading", "Heading", "°")
        self.position = Sensor(self, "gpsPosition", "Position", "lon, lat")
        self.sensors = [self.heading, self.position]
        self.running = False
```

```
        self.lng = 50
        self.lat = -3
        if IS_RPI:
            self.port = "/dev/ttyAMA0"
            self.ser = serial.Serial(self.port, baudrate=9600, timeout=0.5)

    def getLongitude(self):
        return self.lng

    def getLatitude(self):
        return self.lat

    def run(self):
        while self.running:
            try:
                # Check if the code is running on a Raspberry Pi
                # If so, read data from the GPS module
                if IS_RPI:
                    #print("Running NEO6M")
                    newdata = self.ser.readline()
                    newdata = newdata.decode('utf-8')

                    # Returns longitude and latitude in decimal degrees
                    if "GLL" in newdata:
                        newmsg=pynmea2.parse(newdata)
                        self.lat=newmsg.latitude
                        self.lng=newmsg.longitude
                        pos = f"{self.lng},{self.lat}"
                        self.position.setData(pos)

                    # Returns heading in degrees
                    if "HCHDG" in newdata:
                        newmsg=pynmea2.parse(newdata)
                        heading=newmsg.heading
                        self.heading.setData(heading)

                else:
                    self.position.setData("50.604531, -3.408637")
                    self.heading.setData("0")
            except:
                print("Error reading GPS data")
```

4.6 Main Modules

The following sections show the main modules shown in the architecture diagram

4.6.1 GPS Module

The GPS module provides heading and positional data to the rest of the system. In Real Mode the GPS module is supplied by the position and heading data from the appropiate sensors in the sensor register. In Sim Mode the Boat Simulator will pass in the simulated position and heading data to the GPS module.

[ ]:
```
# %load slap/src/iteration2/transducers/gps.py
from utils.nmea.nmeaEncoder import Encoder
from control.autoPilot import AutoPilot
class Gps:
    def __init__(self):
        # Import the instance of the auto pilot

        self.longitude = "0.0"
        self.latitude = "0.0"
        self.heading = 0.0
        self.encoder = Encoder()

    def update(self, heading, longitude, latitude, time):
        #print("heading in gps.py is: ", heading)
        self.auto_pilot.update(heading, time)
        self.longitude = longitude
        self.latitude = latitude
        self.heading = heading

    def setAutoPilot(self, auto_pilot: AutoPilot):
        self.auto_pilot = auto_pilot

    def getPos(self):
        pos = {'lon': self.longitude,
               'lat': self.latitude}
        return pos

    def getHeading(self):
        return self.heading
```

4.6.2 Boat Simulator Module

boatSim.py implements the boat dynamics algorithm which models the boat's heading which responds to changes in the boat's tiller position. This is a threaded module, meaning the boat's position and heading are continously updated. The Boat Simulator algorithm calculates the heading. A Geo Positional library is used to calculate the longitude and latitude for the boat's position based on the boat's current position and simulated speed.

The boats simulator also incudes modeling for boat heading disturbances. The details of this algorithm are in the Design section

```
[ ]: # %load slap/src/iteration2/control/boatSim.py
from geopy.distance import geodesic
from geopy.point import Point
from transducers.sensorRegister import SensorRegister
import sys
import os
from threading import Thread
import time
import math
import random

# Used for the decay equation
TIMECONSTANT = 0.333333
MAX_DISTURBANCE_DURATION = 5000 # ms
MIN_DISTURBANCE_DURATION = 2000 # ms
MAX_DISTURBANCE_MAGNITUDE = 20 # degrees per second
class BoatSim:

    def __init__(self, sensor_register: SensorRegister, gps):
        # Imports the heading variable
        self.gps = gps
        self.heading = 0.0 # Degrees
        self.running = False
        self.speed_over_ground = 50 #knots
        self.pos = Point(50.604531, -3.408637)
        self.sensor_register = sensor_register
        self.simThread = Thread(target=self.simulatedLoop, daemon=True)
        self.readSensorThread = Thread(target=self.readSensorLoop, daemon=True)


    def startSim(self):
        # Starts the control system on a new thread
        self.running = True
        if self.readSensorThread.is_alive():
            self.readSensorThread.join()
        self.simThread = Thread(target=self.simulatedLoop, daemon=True)
        self.simThread.start()
        self.rudderAngle = 0

    def stopSim(self):
        # Stops the system
        self.running = False
        if self.simThread.is_alive():
            self.simThread.join()

        # Create a new thread instance for reading sensors
        self.readSensorThread = Thread(target=self.readSensorLoop, daemon=True)
```

```
        self.readSensorThread.start()
        self.rudderAngle = 0

    def simulatedLoop(self):
        while self.running:
            # Gets current time and creates a disturbance
            self.currentTimeMilli = int(round(time.time() * 1000))
            self.nextDisturbance = self.createDisturbance()
            self.previousTime = 0
            while self.running:

                # Preforms one iteration of the boats movements and ensures its␣
↪a usable value
                self.currentTimeMilli = int(round(time.time() * 1000))

                if self.previousTime != 0:
                    dt = (self.currentTimeMilli - self.previousTime) / 10**3
                else:
                    dt = 0

                # Calculate new long/lat based on distance travelled and␣
↪current heading
                newPos = self.getNewPosition(self.pos, self.speed_over_ground,␣
↪self.heading, dt)
                disturbance = self.disturbance()

                # Calculates new yaw rate based on rudder angle and disturbance
                yawRate = (1 / TIMECONSTANT) * self.rudderAngle + disturbance
                #  (t) =  (0) +   * [t/T + (exp(-t/T) - 1)]

                # Calculates new heading based on yaw rate and time since last␣
↪update
                newHead = (self.heading + yawRate * (dt / TIMECONSTANT + (math.
↪exp(-dt / TIMECONSTANT) - 1))) % 360

                # Updates the GPS with the new heading and position
                self.gps.update(newHead, str(newPos.longitude), str(newPos.
↪latitude), self.currentTimeMilli)
                self.heading = newHead
                self.pos = newPos
                self.previousTime = self.currentTimeMilli


    def readSensorLoop(self):
        # This is our Run Mode Loop
        # Starts reading the sensors and passing the data to the GPS
```

```
        while not self.running:
            self.currentTimeMilli = int(round(time.time() * 1000))
            heading = self.sensor_register.getSensor("MagHeading").getData()
            position = self.sensor_register.getSensor("Position")
            posModule = position.getTransducer()
            longitude = posModule.getLongitude()
            latitude = posModule.getLatitude()
            self.gps.update(heading, longitude, latitude, self.currentTimeMilli)

    def disturbance(self):
        currentTimeMilli = int(round(time.time() * 1000))

        # Checks if disturbance is active and returns the disturbance value
        # If not, creates a new disturbance
        if (currentTimeMilli < self.nextDisturbance['startTime']):
            print("time until disturbance",self.nextDisturbance['startTime'] -␣
↪currentTimeMilli)
            return 0
        elif currentTimeMilli > self.nextDisturbance['startTime'] and␣
↪currentTimeMilli < self.nextDisturbance['endTime']:
            print("During")
            print("time left: " , (self.nextDisturbance['endTime'] -␣
↪currentTimeMilli))
            return self.nextDisturbance['disturbance']
        else:
            print("After")
            self.nextDisturbance = self.createDisturbance()
            return 0


    def setRudderAngle(self,angle):
        self.previousTime = self.currentTimeMilli
        self.rudderAngle = angle

    def createDisturbance(self):
        # Creates a disturbance with a random magnitude and duration
        # The disturbance is a random value between -MAX_DISTURBANCE_MAGNITUDE␣
↪and MAX_DISTURBANCE_MAGNITUDE
        currentTimeMilli = int(round(time.time() * 1000))
        disturbance =  random.randint(-MAX_DISTURBANCE_MAGNITUDE,␣
↪MAX_DISTURBANCE_MAGNITUDE)
        startTime = currentTimeMilli + random.randint(␣
↪MAX_DISTURBANCE_DURATION, 2 * MAX_DISTURBANCE_DURATION)
        endTime = startTime + random.
↪randint(MIN_DISTURBANCE_DURATION,MAX_DISTURBANCE_DURATION)
        output = {'endTime': endTime,
```

```
                'disturbance': disturbance,
                'startTime': startTime}
        print(output)
        return output


    def getNewPosition(self, start_position: Point, speed_kt, heading_deg,␣
↪time_secs):
        # Calculate the distance travelled in km based on speed and time
        distance_km = (speed_kt * 1.852) * (time_secs / (60 * 60))  # Convert␣
↪knots to km/h and compute distance
        #print(distance_km)
        return geodesic(kilometers=distance_km).destination(start_position,␣
↪heading_deg)
```

4.6.3 Tiller Actuator Module

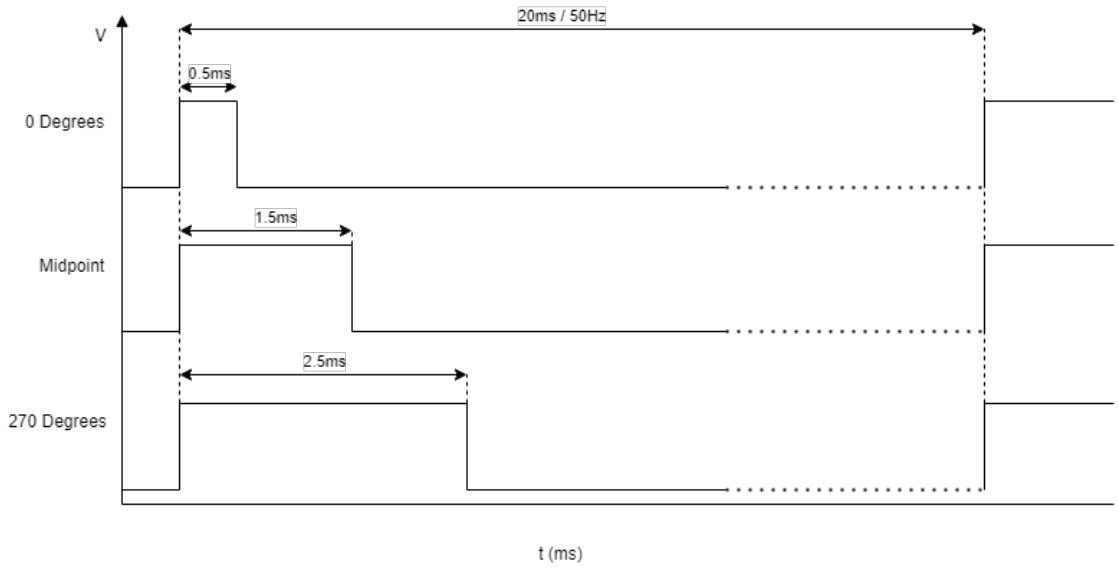
The Tiller Actuator is used to turn the variables in the system into usable motor control signals for the servo motor to the physically turn the rudder. However in Sim Mode tillerActuator.py sets the angle of the simulated rudder, which in turn affects the heading of the boat simulation code.

This module contains the code to interface the RPi5 with a servo motor. This is achieved using the hardware PWM function in the RPi5. This proved necessary since when implemented using a software PWM the other threads in the system causes the PWM timing to jitter.

Servo Motor Implementation

The tiller is driven by a servo motor. In this prototype project a small servo motor. The motor used is a 20kg 8120MG 270 degree servo. The motor operates over its full range of 0 to 270 degrees in response to a pulse width modulated (PWM) signal, where the pulse widths range from 0.5ms to 2.5ms. Therefore the midpoint is where the pulse width is 1.5ms. The frequency of the signal according to the motor specification is 50hz and therefore a 20ms period is used. The diagram below illustrates the required PWM signals.

The motors PWM pattern:



The following code implements the hardware interface and PWM functions as described.

[ ]:
```
# %load slap/src/iteration2/transducers/tillerActuator.py
from control.boatSim import BoatSim
import time
try:
    from rpi_hardware_pwm import HardwarePWM
    IS_RPI = True
    print("Running motor from Pi")
except (ImportError, ModuleNotFoundError, RuntimeError):
    print("Not running on a Raspberry Pi")
    IS_RPI = False


# Constants for the servo motor
RUDDER_COEFFICIENT = 25
SERVO_RANGE = 270

class TillerActuator():

    def __init__(self):
        # Create the PWM object for the servo motor
        # This is only used when running on a Raspberry Pi
        if IS_RPI:
            # Set up the PWM channel for the servo motor
            self.p = HardwarePWM(pwm_channel=2, hz=50,chip=2)
            self.p.start(100)
            self.cycle = 0
        else:
            self.cycle = 0
```

```
    def setBoatSim(self, boat_sim: BoatSim):
        self.boat_sim = boat_sim

    def setTurnMag(self, turn_mag):
        # Set the turn magnitude for the boat simulation if in Sim Mode
        angle = RUDDER_COEFFICIENT * turn_mag
        self.boat_sim.setRudderAngle(angle)
        if IS_RPI:
            # Sets the servo angle based on the turn magnitude
            self.setServo(turn_mag)
        return angle

    def setServo(self, turn_mag: float):
        # The PWM signals function between a range of 0.5ms to 2.5ms
        # The centre point therefore being 1.5ms
        milli = 1.5 + float(turn_mag)
        self.cycle = (milli / 20) * 100
        print("Cycle is: ", self.cycle)
        self.p.change_duty_cycle(self.cycle)
```

4.6.4 Auto Pilot Module

autoPilot.py is the main boat controller. It reads the GPS to find out the current heading and the target heading which are fed to the PID algorithm. The PID algorithm uses the PID constants and the equation to calculate a new Tiller Actuator position. This is fed to the actuator to control the rudder angle. It also contains start / stop functions to start the system or stop it. It contains set and get functions for the headings.

[ ]:
```
# %load slap/src/iteration2/control/autoPilot.py
import control.pidModule
from services.slapStore import SlapStore, Config
from control.pidModule import PidController
from transducers.tillerActuator import TillerActuator
from threading import Thread
from utils.nmea.nmeaDecoder import Decoder
from utils.headings import angularDiff
import time
import math
import random


# If the difference between the target and actual heading is greater that
# the limit of control then fully extend the tiller
LIMIT_OF_CONTROL = 30
```

```
class AutoPilot():

    def __init__(self):
        # Imports all the needed instances for the controller
        # Creates all needed variable for the controller
        self.thread = None
        self.data_store = SlapStore("slap.db")
        self.proportional = 0
        self.integral = 0
        self.differential = 0
        self.pid_controller = PidController( self.proportional, self.integral,␣
 ↪self.differential, LIMIT_OF_CONTROL)

        self.target_heading = 0
        self.decoder = Decoder()
        self.actual_heading = 0.0
        self.tiller_angle = 0.0
        self.config = None
        self.running = False

    def start(self):
        self.pid_controller.reset()
        self.running = True

    def stop(self):
        self.running = False

    def setHeading(self,input):
        # set Heading to the input
        self.target_heading = input
        return self.target_heading

    def getPilotValues(self):
        # Returns a dictionary of both heading values
        headings = {
            'target': self.target_heading,
            'tiller': self.tiller_angle
        }
        return headings

    def setTillerActuator(self,tiller_actuator):
        self.tiller_actuator = tiller_actuator

    def update(self, heading, time):
        # Preforms one iteration of the control
        if self.running:
```

```
            self.actual_heading = heading
            # Preforms one iteration of the PID controller
            diff = angularDiff(self.actual_heading, self.target_heading)

            if abs(diff) <= LIMIT_OF_CONTROL:
                turn_mag = self.pid_controller.pid(self.actual_heading, self.
↪target_heading, time)
            else:
                # If we go outside the control range we must reset the PID␣
↪controller
                self.pid_controller.reset()
                if diff > 0:
                    turn_mag = 1
                elif diff < 0:
                    turn_mag = -1

            # Sets the new rudder angle to the output
            self.tiller_angle = self.tiller_actuator.setTurnMag(turn_mag)
        else:
            self.tiller_angle = 0


    # This function is used to calculate the error between the target and␣
↪actual heading
    # It takes into account the wrap around at 360 degrees
    def getHeadingError(self, target, heading):
        if target - heading <= 180:
            error = target - heading
        else:
            error = (target - heading) - 360
        return error

    def setPidValues(self, config: Config):
        self.config = config
        self.pid_controller.setGains(config)

    def getCurrentConfig(self):
        return self.config
```

4.6.5 PID Module

pidModule.py is the PID control algorithm with takes the current heading, target heading and the gains for each aspect of the calculation (proportional, integral and differential). The PID module works by taking the error (distance between the target heading and current heading).

Adding that to the integral of the error so as the current heading sits apart from the target heading the area under the error-time graph will increase, therefore the output to increase.

This is then added to the differential, where the differential of the error is taken, also known as the rate of change of heading, this being the gradient of the heading-time graph. The use of differential allows us to measure the speed that we are heading towards the target heading, meaning we can dampen the speed as we get nearer the target.

The sum of all these values gives us the function to take the current and target heading and get an output which can adjust the rudder to get us closer to the target heading.

[ ]:
```
# %load slap/src/iteration2/control/pidModule.py
from utils.headings import angularDiff
from services.slapStore import Config
class PidController:

    def __init__(self,KP,KI,KD, LIMIT_OF_CONTROL):
        # Imports the gains to be used
        self.kp = KP
        self.ki = KI
        self.kd = KD
        self.accumlatedError = 0
        self.lastPos = 0
        self.elapsed = 0
        self.time = 0
        self.previous_time = 0
        self.LIMIT_OF_CONTROL = LIMIT_OF_CONTROL


    def pid(self, pos: int, target: int, time: int):
        self.time = time
        intergal = 0
        differential = 0

        # PROPORTIONAL-------
        error = angularDiff(pos, target)
        proportional = error

        if self.previous_time != 0:
            dt = (self.time - self.previous_time) / 10**3
        else:
            dt = 0


        if dt != 0.0:
            # If this is the first run through we cannout calculate dt, so we␣
↪don't find D or I


            # Intergral----------
            intergal = ((error * dt) + self.accumlatedError)
```

```
            self.accumlatedError = intergal


            # Differential-------
            if self.lastPos != None:

                dpos = self.lastPos - pos
                differential = (dpos / dt)
                # Stores the previous position
                # to use in the differential equation next time it is run
            else:
                differential = 0
            self.lastPos = pos

        self.previous_time = self.time

        # Returns the addition of all these values adjusted using the gains
        return (self.kp * proportional + self.ki * intergal + self.kd *␣
↪differential) / self.LIMIT_OF_CONTROL

    def reset(self):
        # Resets the PID controller to its default values
        self.accumlatedError = 0
        self.elapsed = 0
        self.lastPos = None
        self.previous_time = 0

    def setGains(self, config: Config):
        print("setting gains")
        self.kp = config.proportional
        self.ki = config.integral
        self.kd = config.differential
        self.reset()
```

4.6.6 Web Server Module

This package contains the components that run the web server and allow it to interact with the rest of the system: This module contains all the HTML files to be served to the client, the javascript and all images used.

The web server (app.py) has two main functions:

1. Serving HTML files: These HTML files are the user interfaces which are presented on the user's smartphone.
    - The Main Navigation page (/)
    - The List Configs page (/configs)
        - Edit Config page (/configs/)

- The List Trips page (/trips)
  - View Trip page (/trips/)
- The List Sensors page (/sensors)

2. Providing HTTP methods: These are the API calls used by the browser to interact with the system they are:
   - Starting the pilot (/api/startPilot)
   - Stopping the pilot (/api/stopPilot)
   - Uploading trip data to Mapbox (/api/uploadTrip/)
   - Getting current heading (/api/heading)
   - Setting target heading (/api/setHeading)
   - Getting sensor readings (/api/readings)
   - Getting list of configurations (/api/configs)
   - Getting list of trips (/api/trips)
   - Getting specific trip details (/api/trip/)
   - Getting list of sensors (/api/sensors)
   - Getting specific sensor details (/api/sensor/)
   - Starting trip logging (/api/startLogging)
   - Stopping trip logging (/api/stopLogging)

[2]:
```
# %load slap/src/iteration2/web/app.py
import os
from flask import Flask, request, render_template, jsonify, g, redirect, url_for
from flask.wrappers import Response
from services.slapStore import SlapStore, Config, Trip
from control.autoPilot import AutoPilot
from utils.headings import compassify
from services.logger import Logger
from transducers.sensorRegister import SensorRegister
from control.boatSim import BoatSim
from transducers.gps import Gps
import threading
import queue
import time
import json
from datetime import datetime

class WebServer:
    # Initialises the web server with required components
```

```
    def __init__(self, auto_pilot: AutoPilot, logger: Logger, sensor_register:␣
↪SensorRegister, boat_sim: BoatSim, gps: Gps):
        # Importing the Auto Pilot instance
        self.auto_pilot = auto_pilot
        self.logger = logger
        self.sensor_register = sensor_register
        self.logging = False
        self.current_trip = None
        self.boat_sim = boat_sim
        self.gps = gps

    # Creates and configures the Flask web server
    def create_server(self, store: SlapStore):
        # Creating instances of Flask and the Database service
        app = Flask(__name__)
        self.store = store
        # Load boat data

        # Loads all configurations from the store
        def load_configs():
            f = store.listConfigs()
            print({'configs': f})
            return {'configs': f}

        # Loads all trips from the store
        def load_trips():
            trips = store.listTrips()
            print({'trips': trips})
            return {'trips': trips}

        # Template Routes

        # Renders the home page
        @app.route("/")
        def home():
            # Default Route
            return render_template('index.html')

        # Displays sensor readings on a dedicated page
        @app.route('/sensorsReadings')
        def sensorsReadings():
            data = self.sensor_register.getSensorReadings()
            print(data)
            return render_template('sensorsDisplay.html', sensorReadings = data)

        # Shows all configurations
        @app.route('/configs')
```

```
        def index():
            data = load_configs()
            return render_template('configs.html', configs = data["configs"])

        # Shows all trips
        @app.route('/trips')
        def trips():
            data = load_trips()
            return render_template('trips.html', trips=data["trips"])

        # Displays details of a specific trip
        @app.route('/view_trip/<int:tripId>', methods=['GET'])
        def view_trip(tripId):
            # Get trip details
            trip = self.store.getTrip(tripId)
            if not trip:
                return redirect(url_for('trips'))
            return render_template('view_trip.html', trip=trip)

        # Handles editing of configurations
        @app.route('/edit/<int:configId>', methods=['GET'])
        def edit(configId):
            if configId == 0:
                config = {'configId': 0, 'name': 'New Config', 'proportional':␣
↪'0', 'integral': '0', 'differential': '0'}
                return render_template('edit.html', config=config)
            else:
                data = load_configs()
                config = next((proportional for proportional in data['configs']␣
↪if proportional['configId'] == configId), None)
                if config:
                    return render_template('edit.html', config=config)
                return redirect(url_for('index'))

        # Saves configuration changes
        @app.route('/save', methods=['POST'])
        def save():
            config_configId = int(request.form['configId'])
            print(config_configId)
            if config_configId == 0:
                config = Config(0, str(request.form['name']),float(request.
↪form['proportional']),float(request.form['integral']),float(request.
↪form['differential']))
                self.store.newConfig(config)
            else:
                updated_config = {
                    'configId': config_configId,
```

```
                    'name': request.form['name'],
                    'proportional': request.form['proportional'],
                    'integral': request.form['integral'],
                    'differential': request.form['differential']
                }
                updated_config = Config(updated_config['configId'],␣
↪updated_config['name'], updated_config['proportional'],␣
↪updated_config['integral'], updated_config['differential'])
                self.store.updateConfig(updated_config)
            return redirect(url_for('index'))

        # Selects a configuration as active
        @app.route('/select/<int:configId>', methods=['POST'])
        def select(configId):
            config = self.store.getConfig(configId)
            self.auto_pilot.setPidValues(config)
            self.store.setDefault(configId)
            return jsonify({'message': f'Selected config with ID: {configId}'})

        # Deletes a configuration
        @app.route('/delete/<int:configId>', methods=['POST'])
        def deleteConfig(configId):
            self.store.deleteConfig(configId)
            return redirect(url_for('index'))

        # API Routes

        # Sets the target heading for the autopilot
        @app.route('/api/setDirection', methods=['PUT'])
        def setDirection():
            try:
                heading = request.get_data().decode('utf-8')
                if int(heading) < 0 or int(heading) > 360:
                    response_data = {"angle": "Enter a value between 0 and 360"}
                    return jsonify(response_data), 200
                heading = self.auto_pilot.setHeading(int(heading))
                response_data = {"angle": str(heading)}
                return jsonify(response_data), 200
            except Exception as e:
                print(f"Error processing setDirection request: {str(e)}")
                return jsonify({"error": str(e)}), 400

        # Returns current heading information
        @app.route('/api/headings', methods=['GET'])
        def get_headings():
            heading = self.gps.getHeading()
            pilot_values = self.auto_pilot.getPilotValues()
```

```
        headings = {
            'target': pilot_values['target'],
            'tiller': pilot_values['tiller'],
            'actual': heading
        }
        return jsonify(headings)

    # Adjusts the current heading by a specified amount
    @app.route('/api/addDirection', methods=['PUT'])
    def addDirection():
        try:
            change = request.get_data().decode('utf-8')
            headings = self.auto_pilot.getPilotValues()
            heading = headings['target']
            heading = heading + int(change)
            heading = compassify(heading)
            heading = self.auto_pilot.setHeading(heading)
            response_data = {"angle": str(heading)}
            return jsonify(response_data), 200
        except Exception as e:
            print(f"Error processing request: {str(e)}")
            return jsonify({"error": str(e)}), 400

    # Toggles the logging system
    @app.route('/api/toggleLogging')
    def toggleLogging():
        try:
            print("toggleLogging")
            self.current_trip = None
            if self.logger.running:
                self.logger.stop()
            else:
                config = self.store.getCurrentConfig()
                self.auto_pilot.setPidValues(config)
                self.logger.start(config)
                self.current_trip = config
            status = {
                'status': self.logger.running,
                'tripName': "LogName"
            }
            return jsonify(status), 200
        except Exception as e:
            print(f"Error processing request to toggle logging: {str(e)}")
            return jsonify({"error": str(e)}), 400

    # Returns the current status of all system components
    @app.route('/api/systemStatus')
```

```
        def systemStatus():
            if self.current_trip is None:
                name = "No Trip"
                running = self.logger.running
                pilotRunning = self.auto_pilot.running
                simRunning = self.boat_sim.running
            else:
                name = self.current_trip.name
                running = self.logger.running
                pilotRunning = self.auto_pilot.running
                simRunning = self.boat_sim.running

            status = {
                'status': running,
                'tripName': name,
                'pilotRunning': pilotRunning,
                'simRunning': simRunning
            }
            return jsonify(status), 200

        # Returns current sensor readings
        @app.route('/api/sensorReadings')
        def sensorReadings():
            readings = self.sensor_register.getSensorReadings()
            return jsonify(readings), 200

        # Toggles the boat simulation
        @app.route('/api/toggleSimulation', methods=['GET'])
        def toggleSimulation():
            try:
                if self.boat_sim.running:
                    self.boat_sim.stopSim()
                    message = "Simulation stopped"
                else:
                    self.boat_sim.startSim()
                    message = "Simulation started"
                return jsonify({"message": message}), 200
            except Exception as e:
                print(f"Error processing request to toggle simulation:␣
↪{str(e)}")
                return jsonify({"error": str(e)}), 400

        # Starts the autopilot
        @app.route('/api/startPilot')
        def startPilot():
            self.auto_pilot.start()
            return jsonify({'message': 'Pilot started'})
```

```
        # Stops the autopilot
        @app.route('/api/stopPilot')
        def stopPilot():
            self.auto_pilot.stop()
            return jsonify({'message': 'Pilot stopped'})

        # Uploads a trip to Mapbox
        @app.route('/api/uploadTrip/<int:tripId>', methods=['GET'])
        def uploadTrip(tripId):
            trip = self.store.getTrip(tripId)
            readings = self.store.getPosLogs(trip)
            self.logger.map_manager.uploadToMapbox(f"Slap Trip ID: {trip.
↪tripId}", readings)
            return jsonify({'message': 'Trip uploaded'})

        return app
```

4.6.7 HTML Templates

The HTML templates create the user interfaces on a web browser in the smart phone. These web pages are rendered when the templates are requested. SLAP uses advanced web techniques to create interactive pages without refreshing the entire page for each update. The templates include interactive elements, including input fields and buttons. They also include elements which are automatically updated by javascript code which fetches information from the server. Therefore SLAP can be described as a dynamic web application.

The HTTP Templates in SLAP are repeated below:

- The Main Navigation page (/)
  - The List Configs page (/configs)
    - Edit Config page (/configs/)
  - The List Trips page (/trips)
    - View Trip page (/trips/)
  - The List Sensors page (/sensors)

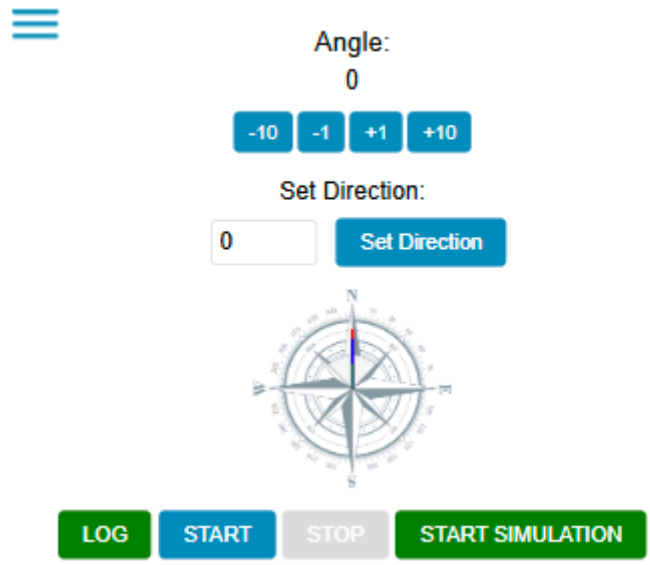
Main User Interface Page

The Main page has the compass rose, with the three colour coded pointers:

- Red: The Target Heading
- Blue: The Current Heading
- Green: The Rudder Angle

This page also include the four heading adjustment buttons and the set direction input field. Below the buttons are:

- Toggle Logging: Starts and stops logging function creating a new Trip Log each time
- Start / Stop Auto Pilot: Starts and stops the automated control of the tiller
- Toggle Simulation: Engages and disengages the boat simulator for the selected config.



The code below is the HTML template for this page:

[5]:
```
# %load slap/src/iteration2/web/templates/index.html
<!DOCTYPE html>
<html lang="en">

<script src="{{ url_for('static', filename='js/angle.js') }}"></script>
<script src="{{ url_for('static', filename='js/burger.js') }}"></script>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>SLAP - Self Logging Auto Pilot</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/styles.
 ↪css') }}">
</head>

<body>

    <div class="burger-menu">
        <div class="burger-icon" onclick="toggleMenu()">
            <span></span>
            <span></span>
            <span></span>
            <span></span>
        </div>
```

```
        <div class="menu-overlay" id="menuOverlay">
            <div class="menu-content">
                <a href="configs">Configs</a>
                <a href="trips">Trips</a>
                <a href="sensorsReadings">Sensors</a>
            </div>
        </div>
    </div>
    <div class="container">

        <div class="angle-control">
            <div class="angle-display">
                Angle: <div id="angle">0</div>
            </div>
            <div class="angle-buttons">
                <button onclick="changeValue(-10)">-10</button>
                <button onclick="changeValue(-1)">-1</button>
                <button onclick="changeValue(1)">+1</button>
                <button onclick="changeValue(10)">+10</button>
            </div>
        </div>

        <div class="direction-control">
            <label for="directionSet">Set Direction:</label>
            <input type="number" id="directionSet" name="directionSet">
            <button onclick="setValue(parseInt(document.
↪getElementById('directionSet').value))">Set Direction</button>
        </div>

        <div class="compass-container">
            <svg width="400" height="400" xmlns="http://www.w3.org/2000/svg"␣
↪viewBox="0 0 200 200">
                <image href="{{ url_for('static', filename='images/compass-rose.
↪png') }}" x="0" y="0" width="200" height="200" opacity="0.5"/>
                <line id="target"
                    x1="100" y1="100"
                    x2="100" y2="40"
                    style="stroke:rgb(255, 0, 0);stroke-width:3"
                    transform="rotate(0, 50, 50)" />
                <line id="actual"
                    x1="100" y1="100"
                    x2="100" y2="50"
                    style="stroke:rgb(0, 0, 255);stroke-width:3"
                    transform="rotate(0, 50, 50)" />
                <line id="tiller"
                    x1="100" y1="100"
                    x2="100" y2="75"
```

```
                    style="stroke:rgb(0, 173, 0);stroke-width:2"
                    transform="rotate(0, 50, 50)" />
            </svg>
        </div>

        <div class="logging-control">
            <button id="loggingButton" onclick="toggleLogging()">LOG</button>
            <button id="startPilotButton" onclick="startPilot()">START</button>
            <button id="stopPilotButton" onclick="stopPilot()">STOP</button>
            <button id="simulateButton" onclick="toggleSimulation()">START␣
 ↪SIMULATION</button>
        </div>
    </div>
</body>

</html>
```

List Configs Page

The List Config Page contains...

[Screenshot]

JavaScript This JavaScript is responsible for the function in the web browser that interacts with the server.

1. Its interactions with the web server in response to the user's interactions
2. It has a polling loop to continously retrieve updates from the server on the current heading and update the user interface.

[ ]:
```
// %load ../../../src/iteration2/web/static/js/angle.js
// Functions for reacting to button presses


function changeValue(c) {
    const url = '/api/addDirection';
    console.log("Inside changeValue", c);

    fetch(url, {
        method: "PUT",
        headers: {
            "Content-Type": "text/plain"  // Changed from text/html
        },
        body: c.toString()
    })
        .then(response => {
            if (!response.ok) {
                throw new Error(`HTTP error! status: ${response.status}`);
```

```
            }
            return response.json();
        })
        .then(data => {
            console.log("Received response:", data);
            document.getElementById("angle").textContent = data.angle;
            updateLine(data.angle)
        })
        .catch(error => {
            console.error("Error:", error);
            document.getElementById("angle").textContent =
                "Error: " + error.message;
        });
}

function setValue(c) {
    const url = '/api/setDirection';
    console.log("Inside changeValue", c);

    fetch(url, {
        method: "PUT",
        headers: {
            "Content-Type": "text/plain"  // Changed from text/html
        },
        body: c.toString()  // Send the raw value instead of JSON.stringify
    })
        .then(response => {
            if (!response.ok) {
                throw new Error(`HTTP error! status: ${response.status}`);
            }
            return response.json();
        })
        .then(data => {
            console.log("Received response:", data);
            document.getElementById("angle").textContent = data.angle;
        })
        .catch(error => {
            console.error("Error:", error);
            document.getElementById("angle").textContent =
                "Error: " + error.message;
        });
    }
            // Update the linear compass for the actual heading
function updateHeading(a){
    console.log("update Heading: ", a)
    line = document.getElementById("actual")
    line.setAttribute("transform", `rotate(${a}, 100, 100)`);
```

```
}
            // update the line on the compass for the target heading
function updateTarget(a){
    console.log("update Target: ", a)
    line = document.getElementById("target")
    line.setAttribute("transform", `rotate(${a}, 100, 100)`);

}

// Update the compass by getting the values from the server
async function updateCompass() {
    const response = await fetch('/api/headings');
    console.log(response)
    const readings = await response.json();
    updateTarget(readings.target)
    updateHeading(readings.actual)

}

// Initial load
updateCompass();

// Update readings
setInterval(updateCompass, 200);
```

Services This package contains the database and storage. The file called slapStore contains all the service functions to interact with the database. This module contains the entities within the database model as shown in figure 2. It also contains a collection of service methods that access the database. The design pattern of this package is known as a Service Fascade. The purpse of this package is to provide the methods the program requires whilst hiding the complexity and implementation of the database from the rest of the program.

[ ]:
```
# %load ../../../src/iteration2/services/slapStore.py
import sqlite3


# --- All Classes possible to put into the database ---
class Boat():

    def __init__(self, id: int,  name: str, model: str, proportional: int,␣
 ↪integral: int, differential: int):
        # Contains and assigns the values for the Boat type
        self.name = name
        self.boatId = id
        self.model = model
        self.proportional = proportional
```

```
        self.integral = integral
        self.differential = differential

class Sensor():

    def __init__(self, id: int, boat: int, name: str, model: str):
        # Contains and assigns the values for the Sensor type
        self.sensorId = id
        self.boatId = boat
        self.sensorName = name
        self.sensorModel = model

class Trip():
    def __init__(self, trip_id: int, boat_id: int, time_started: str,␣
 ↪time_ended: str, date_started: str, date_ended: str, distance_travelled:␣
 ↪float):
        # Contains and assigns the values for the Trip type
        self.tripId = trip_id
        self.boatId = boat_id
        self.timeStarted = time_started
        self.timeEnded = time_ended
        self.dateStarted = date_started
        self.dateEnded = date_ended
        self.distanceTravelled = distance_travelled

class Reading():
     # Contains and assigns the values for the Reading type
    def __init__(self, sensor_id: int, trip_id: int, data: float, timestamp:␣
 ↪str):
        self.sensorId = sensor_id
        self.tripId = trip_id
        self.data = data
        self.timeStamp = timestamp


class SlapStore():

    def __init__(self):
        # Set up database and creates all necessary tables
        print("Creating Database")
        self.connection = sqlite3.connect("slap.db")
        self.cursor = self.connection.cursor()

        # Boat table
        self.cursor.execute('''
            CREATE TABLE IF NOT EXISTS BOATS (
                boatId INTEGER PRIMARY KEY,
```

```
                boatName TEXT NOT NULL,
                boatModel TEXT NOT NULL,
                proportional INTEGER NOT NULL,
                integral INTEGER NOT NULL,
                differential INTEGER NOT NULL
            )
        ''')

        # Sensor table
        self.cursor.execute('''
            CREATE TABLE IF NOT EXISTS Sensor (
                sensorId INTEGER PRIMARY KEY,
                boatId INTEGER NOT NULL,
                sensorName TEXT NOT NULL,
                sensorType TEXT NOT NULL,
                dataType TEXT NOT NULL,
                FOREIGN KEY (boatId) REFERENCES BOATS(boatId) ON DELETE CASCADE
            )
        ''')

        # Trip table
        self.cursor.execute('''
            CREATE TABLE IF NOT EXISTS Trip (
                tripId INTEGER,
                boatId INTEGER NOT NULL,
                timeStarted TEXT NOT NULL,
                timeEnded TEXT NOT NULL,
                dateStarted DATE NOT NULL,
                dateEnded DATE NOT NULL,
                distanceTravelled FLOAT NOT NULL,
                PRIMARY KEY (tripId, boatId),
                FOREIGN KEY (boatId) REFERENCES BOATS(boatId) ON DELETE CASCADE
            )
        ''')

        # Reading table
        self.cursor.execute('''
            CREATE TABLE IF NOT EXISTS Reading (
                sensorId INTEGER NOT NULL,
                tripId INTEGER NOT NULL,
                data FLOAT NOT NULL,
                timeStamp TIME NOT NULL,
                FOREIGN KEY (sensorId) REFERENCES Sensor(sensorId) ON DELETE␣
↪CASCADE,
                FOREIGN KEY (tripId) REFERENCES Trip(tripId) ON DELETE CASCADE
            )
        ''')
```

```
        self.connection.commit()

#         ---All service functions for the database---

    def addBoat(self, boat: Boat):
        # Inserts a boat into the database
        self.cursor.execute(f"INSERT INTO BOATS (boatId, boatName, boatModel,␣
 ↪proportional, integral, differential) VALUES ('{boat.boatId}','{boat.
 ↪name}','{boat.model}','{boat.proportional}','{boat.integral}','{boat.
 ↪differential}')")
        self.connection.commit()

    def getGains(self,id: int):
        # Returns all PID gains stored in the Boat instance
        gains = {}
        # Retrieves all data and stores it as a dictionary
        self.cursor.execute(f"SELECT proportional, integral, differential FROM␣
 ↪BOATS WHERE boatId = '{id}'")
        columns = [desc[0] for desc in self.cursor.description]
        for row in self.cursor.fetchall():
            row_dict = dict(zip(columns, row))
            gains = row_dict
        return gains

    def getBoat(self,name: str):
        # Returns all information on a boat using its name as the identifier
        self.cursor.execute(f"SELECT * FROM BOATS WHERE boatName == '{name}'")
        row = self.cursor.fetchone()
        return row

    def addSensor(self, sensor: Sensor):
        # Inserts a Sensor into the database
        self.cursor.execute(f"INSERT INTO Sensor (sensorId, boatId, sensorName,␣
 ↪sensorType, dataType) VALUES ('{sensor.sensorId}', '{sensor.boatId}',␣
 ↪'{sensor.sensorName}', '{sensor.sensorModel}', 'N/A')")
        self.connection.commit()

    def getSensor(self, sensor_name: str):
        # Returns all information on a sensor using its name
        self.cursor.execute(f"SELECT * FROM Sensor WHERE sensorName ==␣
 ↪'{sensor_name}'")
        row = self.cursor.fetchone()
        return row

    def addTrip(self, trip: Trip):
        # Inserts a Trip into the database
```

```
        self.cursor.execute(f"INSERT INTO Trip (tripId, boatId, timeStarted,␣
↪timeEnded, dateStarted, dateEnded, distanceTravelled) VALUES ('{trip.
↪tripId}', '{trip.boatId}', '{trip.timeStarted}', '{trip.timeEnded}', '{trip.
↪dateStarted}', '{trip.dateEnded}', '{trip.distanceTravelled}')")
        self.connection.commit()

    def getTrip(self, trip_id: int, boat_id: int):
        # returns all information on a trip using the tripId and BoatId as the␣
↪identifier
        self.cursor.execute(f"SELECT * FROM Trip WHERE tripId == {trip_id} AND␣
↪boatId == {boat_id}")
        row = self.cursor.fetchone()
        return row

    def addReading(self, reading: Reading):
        # Inserts a Reading into the database
        self.cursor.execute(f"INSERT INTO Reading (sensorId, tripId, data,␣
↪timeStamp) VALUES ('{reading.sensorId}', '{reading.tripId}', '{reading.
↪data}', '{reading.timeStamp}')")
        self.connection.commit()

    def getReading(self, sensor_id: int, trip_id: int):
        # Returns all Reading information using the sensorId and TripId as␣
↪identifiers
        self.cursor.execute(f"SELECT * FROM Reading WHERE sensorId ==␣
↪{sensor_id} AND tripId == {trip_id}")
        row = self.cursor.fetchone()
        return row

    def getAllReadings(self):
        # Prints all readings from any sensor
        self.cursor.execute(f"SELECT * FROM Reading")
        for row in self.cursor.fetchall():
            print(row)

    def dropAllTables(self):
        # Deletes all tables
        self.cursor.execute(f"DROP TABLE *")
```

Database The database is used to store information about all the trips a boat has preformed including information about the boad, sensors and readings.

The Boat table includes all the attributes of the boat, such as its name, model and the PID tuning information.

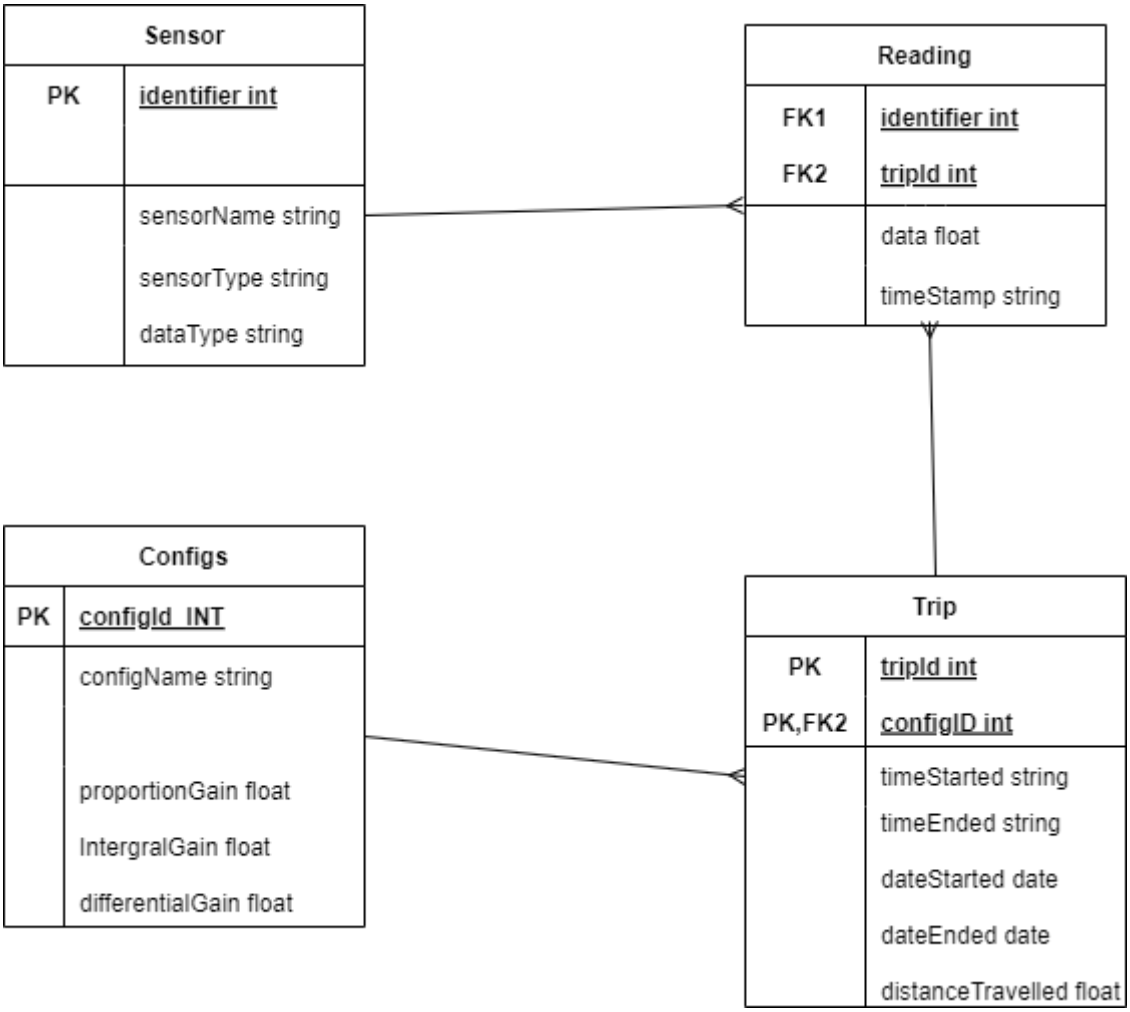
The Sensor table stores the information about a specific sensor. Such as its name, model, its datatype and its associated boat

The Reading table stores a single reading from a sensor. it only stores the data value and timestamp.

The Trip table stores all the information about a single trip, including the associated boat, its start and end data, time and the distance travelled.

Below is the entity relationship diagram for the SLAP database.

Figure 2



4.7 Testing

4.7.1 Introduction

Testing on the project source code consists of manual testing, where the user interfaces are tested to verify their expected functions.

The tests are organised into functional modules according to the system design. The functional modules and the use cases are shown in the diagram, figure xyz. For each of the use case tests (manual) the underlying code is tested with automated unit tests.

4.8 Index of Testing

The following index sets out all of the testing in the project. The tables below index both the manual and associated unit tests. Following the index, evidence for testing is given where appropriate.

Config The following tests verify the systems configuration functions. The configuration system allows different control parameters to be arranged for different sea conditions. see section xyz for functional details

Test Number	Use Case	Summary	Type	Result
1	Default	The system behaviour when no config is present	Manual	[]
1.1		Function to correctly create default config	Unittest	[]
2	Create	User enters config values for a new control configuration	Manual	[]
2.1		Function to correctly create custom config	unittest	[]
3	Edit	User changes a configuration's values	Manual	[]
3.1		Function to correctly edit existing config	unittest	[]
3.2		User enters invalid values to edit page	Manual	[]
4	Delete	User deletes a configuration	Manual	[]
4.1		Function to correctly delete existing config	unittest	[]
5	Simulate	User enables simulator mode for the selected config	Manual	[]
6	Add Plugin	Adds a sensor definition and plugin code	Manual	[]

Auto Pilot

Test Number	Use Case	Summary	Type	Result
1	Start/Stop	User starts or stops the autopilot	manual	[]
1.1		Function to correctly start/stop autopilot	unittest	[]
2	Adjust (+-)	User adjusts the autopilot settings	manual	[]
2.1		Function to correctly adjust target angle	unittest	[]
2.2		Function to correctly adjust servo motor for rudder angle	unittest	[]
3	Set Direction	User sets the direction for the autopilot	manual	[]
3.1	Set Direction	User enters invalid direction	manual	[]
3.1		Function to correctly set target direction	unittest	[]

Logging

Test Number	Use Case	Summary	Result
1	Start/Stop	User starts or stops the logging	[]
1.1		Function to correctly start/stop logging	unittest
2	Upload	User uploads the log data	[]
2.1		Function to correctly upload log data	unittest
3	View	User views the log data	[]

4.8.1 Evidence

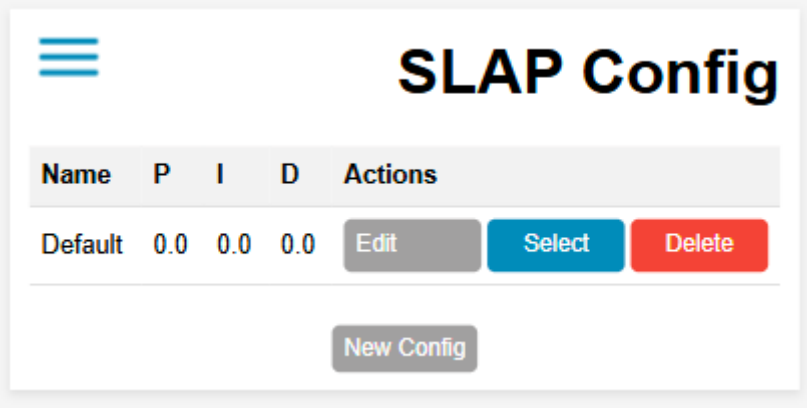
Config Test 1: Default

Description	Expected Outcome	Test Type
When the system is first started, no user config has been create and so the database will be empty. The system detects this, it creates a default config which is added to the database	Created default config	Manual

Procedure:

1. Clear database
2. Start application
3. Verify a default config has been added

Result: The screenshot shows a default config has been created



Unit Test The unit test code below verifies the underlying methods for this functionality

[5]:
```
# %load C:
↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_defaultConfig.
↪py
def test_getCurrentConfig_creates_default_when_none_exists(self):
        """Test that getCurrentConfig creates a default config when none␣
↪exists"""
        # Get current config (should create default)
        config = self.store.getCurrentConfig()

        # Verify default config was created
        self.assertEqual(config.name, 'Default')

        # Verify config was saved to database
        self.store.cursor.execute("SELECT * FROM CONFIGS WHERE isDefault =␣
↪True")
```

```
        saved_config = self.store.cursor.fetchone()
        self.assertIsNotNone(saved_config)
        self.assertEqual(saved_config['name'], 'Default')
        self.assertEqual(saved_config['proportional'], 0)
        self.assertEqual(saved_config['integral'], 0)
        self.assertEqual(saved_config['differential'], 0)
```

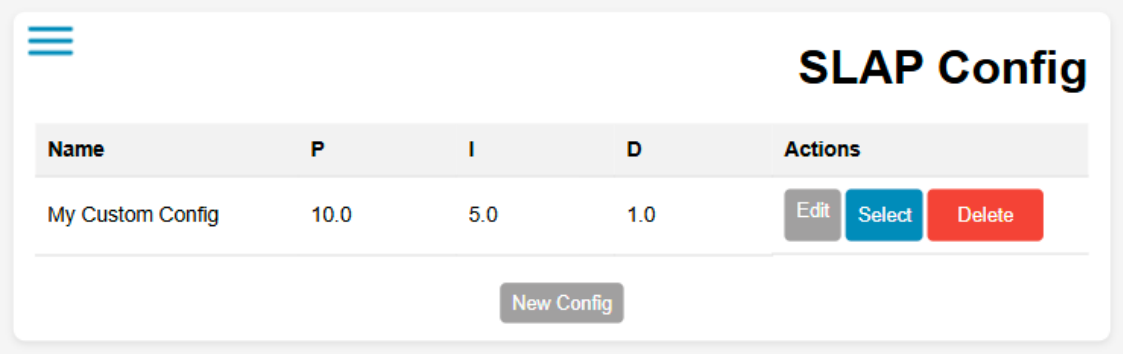
Test 2: Create

Description	Expected Outcome	Test Type
Slap provides the facility to create custom configs, the user can create a config and save it to the database to be selected for later use	Normal	Config is saved to database

Procedure:

1. Visit config page
2. Press create Config
3. Enter all needed values in the form
4. Press save
5. View saved config in database

Result: The screenshot shows a custom config has been created



Unit Test The unit test code below verifies the underlying methods for this functionality

[7]:
```
# %load C:
↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_createConfig.
↪py
from services.slapStore import SlapStore, Config

def test_createConfig_creates_new_config(self):
        """Test that createConfig creates a config"""
```

```
        print("--------------------------------")
        print("")
        print("UNIT TEST: Config_Create")
        print("\nTesting creation of new config...")

        # Get current config (should create default)
        config = Config(0, "My Custom Config", 10, 5, 1)

        config = self.store.newConfig(config)

        self.assertEqual(config.name, 'My Custom Config')

        # Verify config was saved to database
        self.store.cursor.execute(f"SELECT * FROM CONFIGS WHERE configId =␣
↪'{config.configId}'")
        saved_config = self.store.cursor.fetchone()
        self.assertIsNotNone(saved_config)

        print("Verifying saved values match input values...")
        self.assertEqual(saved_config['name'], 'My Custom Config')
        self.assertEqual(saved_config['proportional'], 10)
        self.assertEqual(saved_config['integral'], 5)
        self.assertEqual(saved_config['differential'], 1)
        print("All values verified successfully")
```

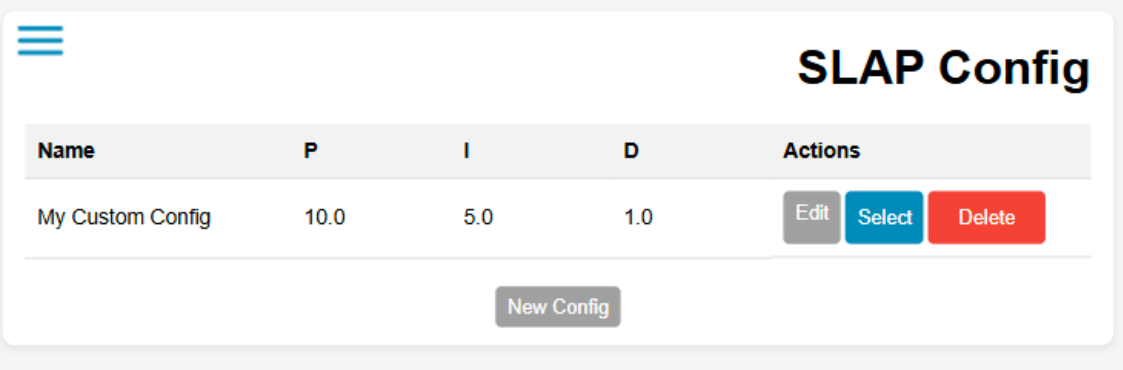
```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[7], line 2
      1 # %load C:
 ↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_createConfig
 ↪py
----> 2 from services.slapStore import SlapStore, Config
      4 def test_createConfig_creates_new_config(self):
      5         """Test that createConfig creates a config"""

ModuleNotFoundError: No module named 'services'
```

Test 3: Edit

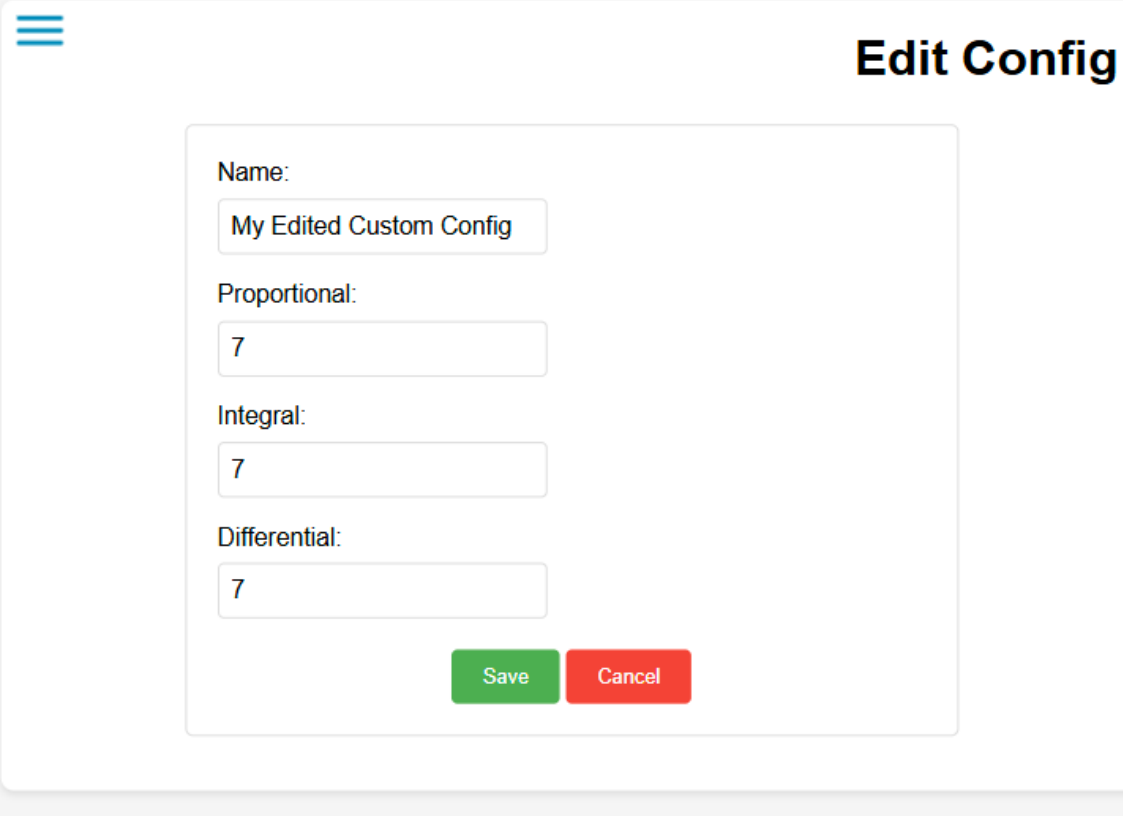
Description	Data Type	Expected Outcome	Test Type
User changes a configuration's values	Normal	Edited config is saved to database	Manual

Procedure:

1. Visit config page

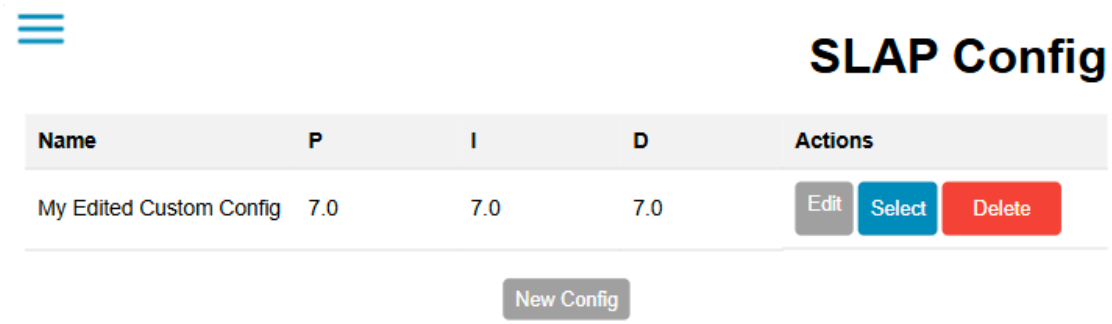


2. Press edit Config



3. Adjust needed values in the form
4. Press save
5. View saved config in database

Result

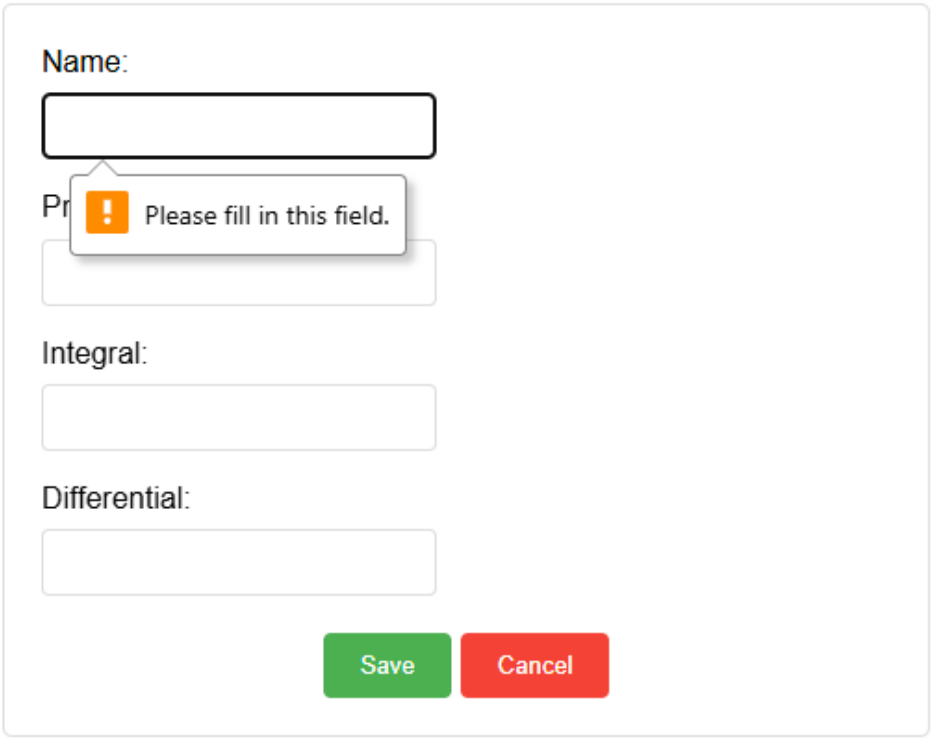


Description	Data Type	Expected Outcome	Test Type
User changes a configuration's values	Errornous	Prompted of incorrect input	Manual

Procedure:

1. Visit config page
2. Press edit Config
3. Eneter invalid values into the form
4. Press save
5. View prompt to reenter values

Result



Unit Test The unit test code below verifies the underlying methods for this functionality

[ ]:
```
# %load C:
↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_editConfig.
↪py
from services.slapStore import SlapStore, Config

def test_editConfig_updates_existing_config(self):
        """Test that editConfig updates an existing config"""
        print("--------------------------------")
        print("")
        print("UNIT TEST: Config_Edit")
        print("\nTesting editing of existing config...")

        # Create initial config
        initial_config = Config(0, "Test Config", 1, 2, 3)
        initial_config = self.store.newConfig(initial_config)

        # Edit the config
        edited_config = Config(initial_config.configId, "Edited Config", 10,␣
↪20, 30)
        self.store.updateConfig(edited_config)
```

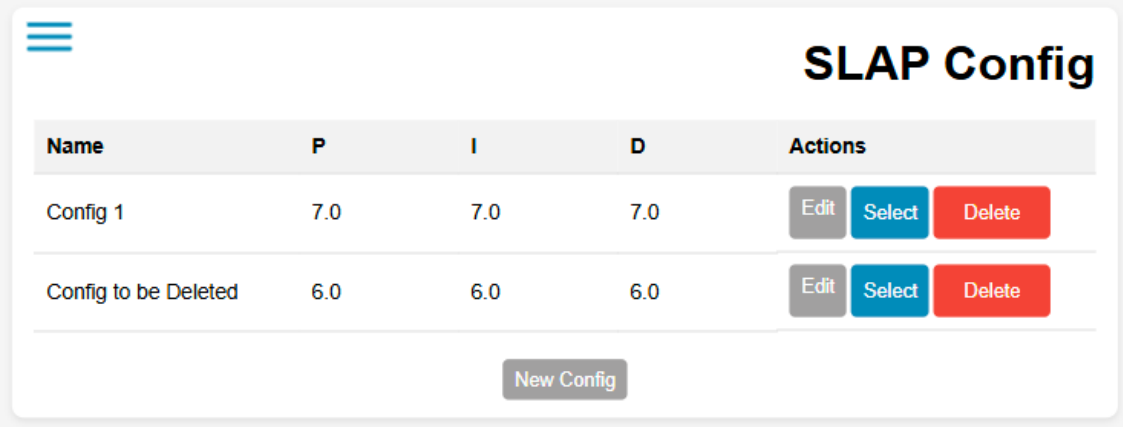
```
        # Verify config was updated in database
        self.store.cursor.execute(f"SELECT * FROM CONFIGS WHERE configId =␣
↪'{initial_config.configId}'")
        saved_config = self.store.cursor.fetchone()
        self.assertIsNotNone(saved_config)

        print("Verifying saved values match edited values...")
        self.assertEqual(saved_config['name'], 'Edited Config')
        self.assertEqual(saved_config['proportional'], 10)
        self.assertEqual(saved_config['integral'], 20)
        self.assertEqual(saved_config['differential'], 30)
        print("All values verified successfully")
```

Test 4: Delete

Description	Data Type	Expected Outcome	Test Type
User deletes a configuration	Normal	Config is removed from database	Manual

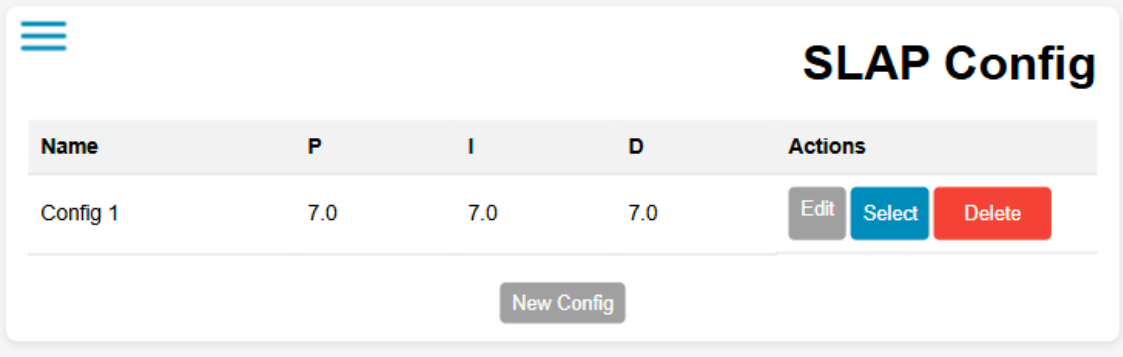
Procedure:

1. Visit config page



2. Press delete on a config row
3. View configs in database to verify deletion

Result



Unit Test The unit test code below verifies the underlying methods for this functionality

```
[ ]: # %load C:
      ↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_deleteConfig.
      ↪py
    from services.slapStore import SlapStore, Config

    def test_deleteConfig_removes_existing_config(self):
            """Test that deleteConfig removes an existing config"""
            print("--------------------------------")
            print("")
            print("UNIT TEST: Config_Delete")
            print("\nTesting deletion of existing config...")

            # Create initial config
            initial_config = Config(0, "Test Config", 1, 2, 3)
            initial_config = self.store.newConfig(initial_config)

            # Delete the config
            self.store.deleteConfig(initial_config.configId)

            # Verify config was deleted from database
            self.store.cursor.execute(f"SELECT * FROM CONFIGS WHERE configId =␣
      ↪'{initial_config.configId}'")
            deleted_config = self.store.cursor.fetchone()

            print("Verifying config was deleted...")
            self.assertIsNone(deleted_config)
            print("Config deletion verified successfully")
```

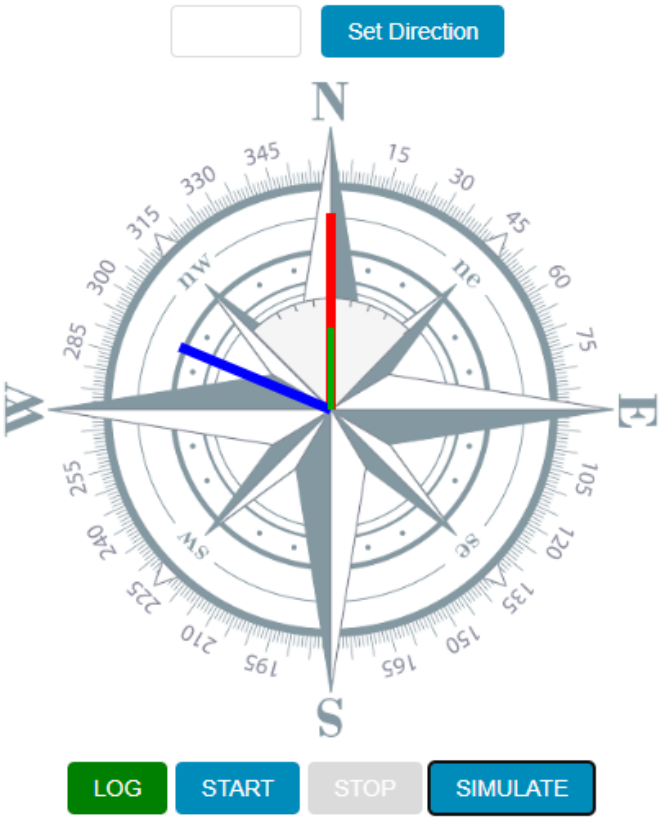
Test 5: Simulate

Description	Data Type	Expected Outcome	Test Type
User enables simulator mode for the selected config	Normal	Simulator is started	Manual

Procedure:

1. Press simulate
2. Verify simulator starts and see the needle wandering

Result:



Test 6: Add Plugin

Description	Data Type	Expected Outcome	Test Type
Adds a sensor definition and plugin code	Normal	Plugin is saved to database	Manual

[Screenshot of config in database]

Auto Pilot Test 1: Start/Stop

Description	Data Type	Expected Outcome	Test Type
User starts or stops the autopilot	Normal	Autopilot starts or stops	Manual

Procedure:

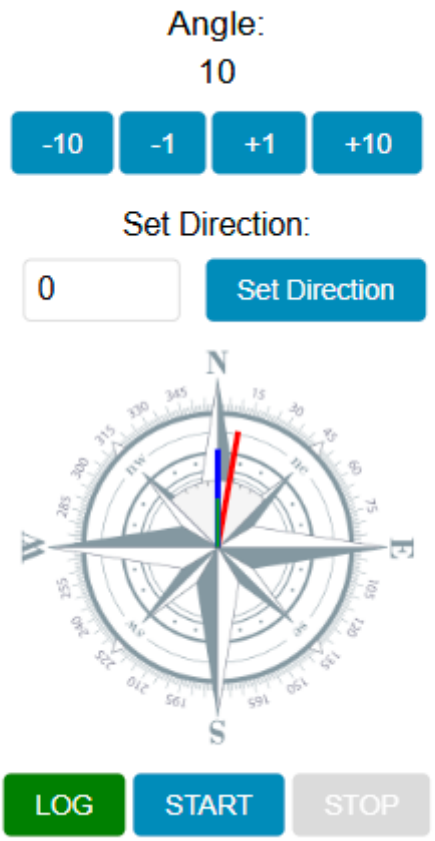
1. Press start/stop button
2. Verify autopilot status changes

Test 2: Adjust (+-)

Description	Data Type	Expected Outcome	Test Type
User adjusts the autopilot settings	Normal	Settings are adjusted	Manual

Procedure:

2. Press +/- buttons to adjust settings
3. Verify target heading changes correctly



Unit Test The unit test code below verifies the underlying methods for this functionality

[ ]:
```python
# %load slap/src/iteration2/tests/test_adjustAutoPilot.py
from control.autoPilot import AutoPilot

def test_adjust_target_angle(self):
    """Test that the +/-10 buttons correctly adjust the target angle"""
    # Create test autopilot with initial target angle of 0
    auto_pilot = AutoPilot()
    auto_pilot.setHeading(0)

    # Test +10 button
    heading = auto_pilot.setHeading(auto_pilot.getHeadings()['target'] + 10)
    assert heading == 10

    # Test -10 button
    heading = auto_pilot.setHeading(auto_pilot.getHeadings()['target'] - 10)
    assert heading == 0
```

Test 3: Set Direction

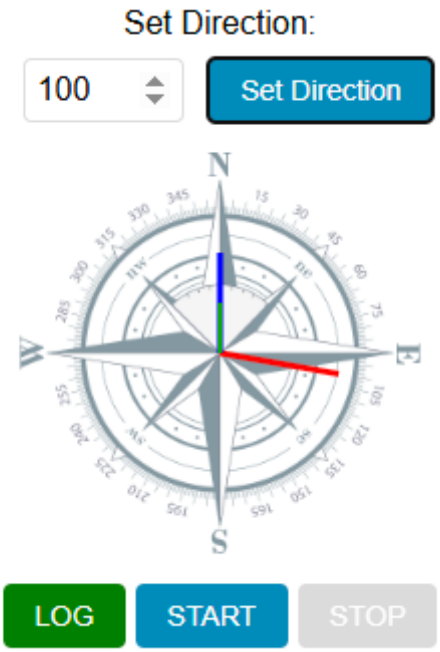
Description	Data Type	Expected Outcome	Test Type
User sets the direction for the autopilot	Normal	Direction is set	Manual

Procedure:

2. Enter desired heading in degrees (0-359)

3. Press set button
4. Verify target heading updates to entered value

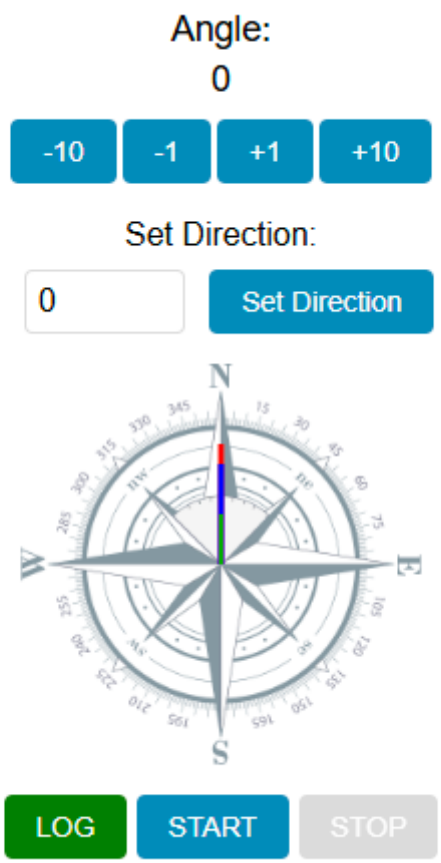
Result:



Description	Data Type	Expected Outcome	Test Type
User sets the direction for the autopilot	Errornous	Direction is set	Manual

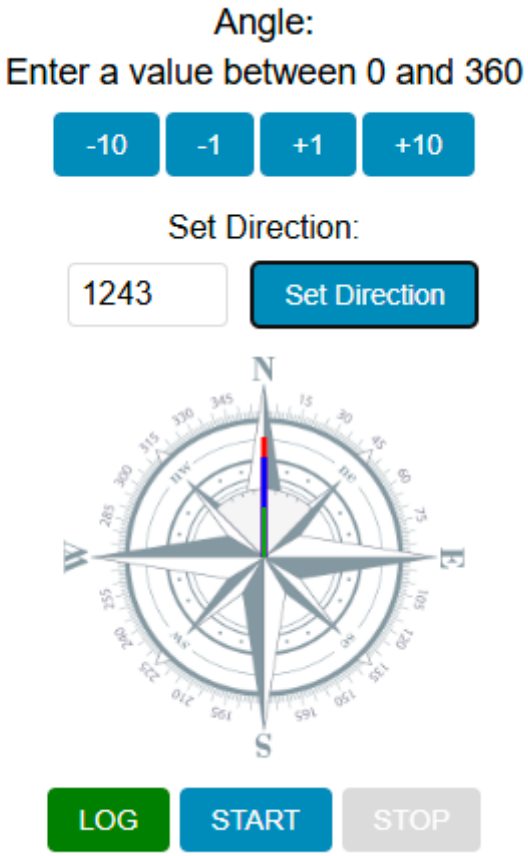
Procedure:

2. Enter errornous heading in degrees (<0 or >360)
3. Press set button



4. Verify that the system returns an error, prompting the user to enter another value

Result:



Unit Test The unit test code below verifies the underlying methods for this functionality

[ ]:
```
# %load C:
 ↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_setAutoPilot.
 ↪py
from control.autoPilot import AutoPilot
from web.app import WebServer
from services.logger import Logger
from services.mapManager import MapManager
from transducers.gps import Gps
from services.slapStore import SlapStore
import unittest
import json

def test_set_direction(self):
    print("\nTesting setDirection endpoint...")

    # Create test client and make request
    self.auto_pilot = AutoPilot()
    self.logger = Logger(Gps(), MapManager())
```

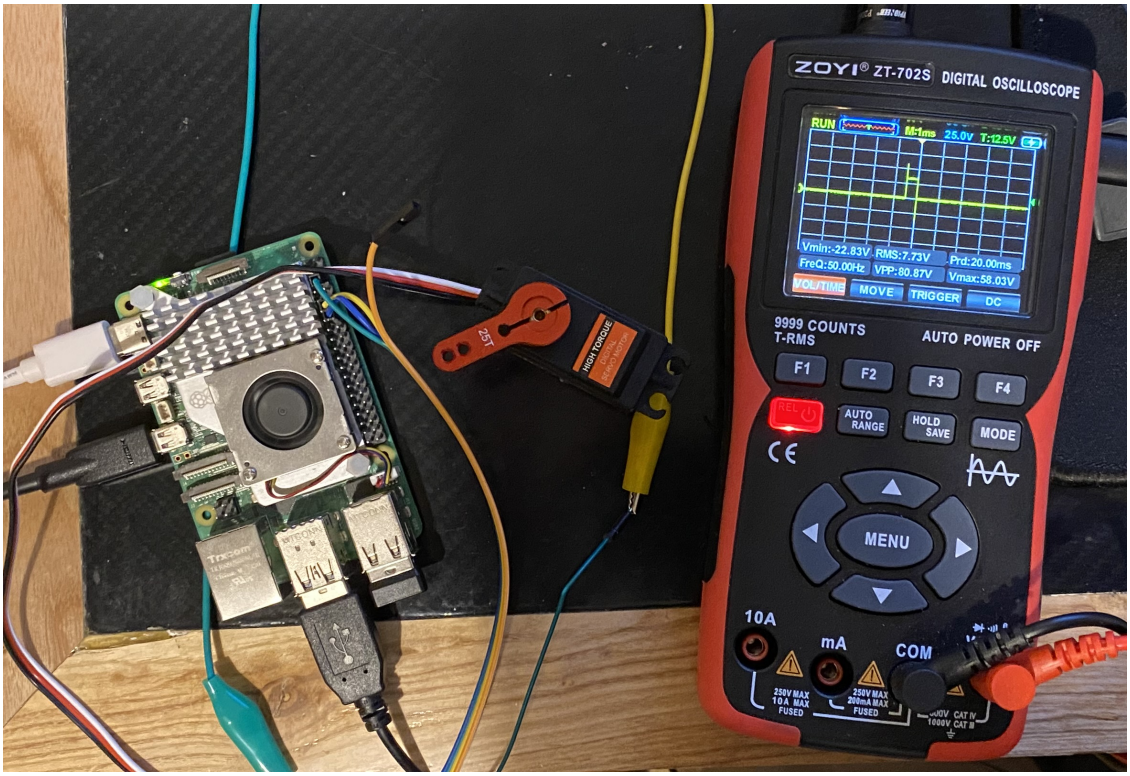
```
    self.web_server = WebServer(self.auto_pilot, self.logger)
    app = self.web_server.create_server(self.store)
    self.client = app.test_client()
    response = self.client.put('/api/setDirection', data='180')
    self.assertEqual(response.status_code, 200)
    data = json.loads(response.data)
    self.assertEqual(data['angle'], '180')
    self.assertEqual(self.auto_pilot.getHeadings()['target'], 180)

    response = self.client.put('/api/setDirection', data='90')
    self.assertEqual(response.status_code, 200)
    data = json.loads(response.data)
    self.assertEqual(data['angle'], '90')
    self.assertEqual(self.auto_pilot.getHeadings()['target'], 90)

    print("Valid heading tests passed")
```

Test 3: Set Direction (Hardware)

Description	Data Type	Expected Outcome	Test Type
User sets the direction for the autopilot	Normal	Direction is set	Manual

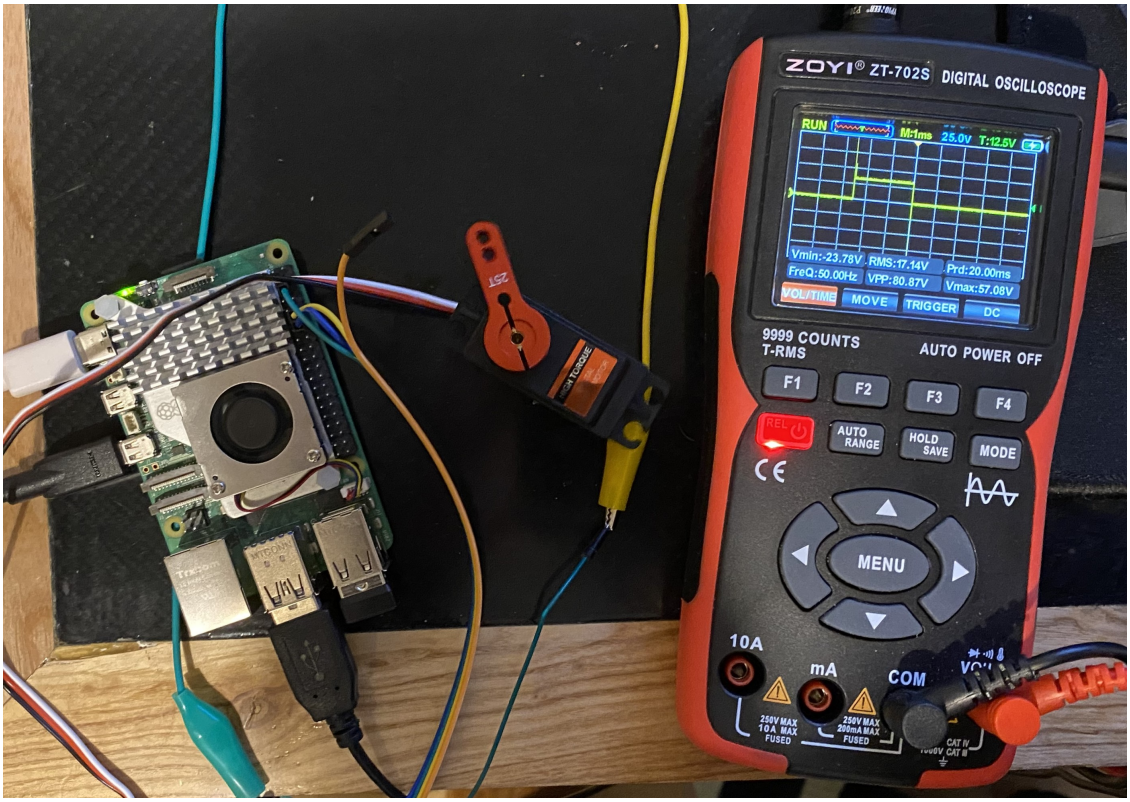
Procedure:

1. Enter Heading so tiller turns to left
2. Check PWM response and motor position



The picture shows the motor position for the tiller to the left in response to the PWM interval of 0.5ms as seen on the oscilliscope trace.

3. Change Heading so tiller turns to right
4. Check PWM response and motor position



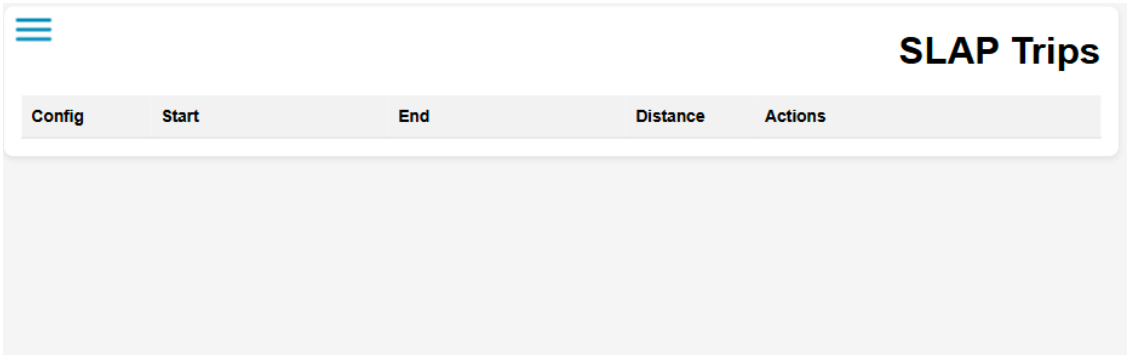
The picture shows the motor position for the tiller to the right in response to the PWM interval of 2.5ms as seen on the oscilliscope trace.

Logging Test 1: Start/Stop

Description	Data Type	Expected Outcome	Test Type
User starts or stops the logging	Normal	Logging starts or stops	Manual

Procedure:

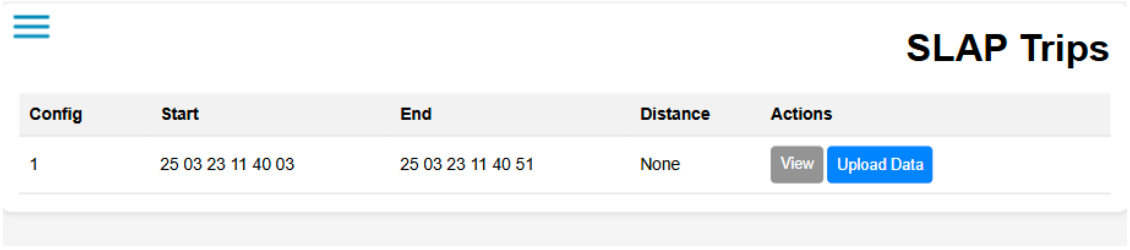
1. Verify no trip is present



2. Press start/stop button and check UI updates

3. Check trip is created/closed appropriately

Result:



Unit Test The unit test code below verifies the underlying methods for this functionality

```
[ ]: # %load slap/src/iteration2/tests/test_startStopLogging.py
from services.logger import Logger
from transducers.gps import Gps
from services.mapManager import MapManager
from services.slapStore import Config

def test_startStopLogging_creates_new_trip(self):
    """Test that starting and stopping logging creates a new trip"""
    # Create test components
    gps = Gps()
    map_manager = MapManager()
    logger = Logger(gps, map_manager)
    logger.setStore(self.store)
```

```
    # Start logging
    logger.start(Config(0, 'default', 0, 0, 0))
    assert logger.running == True

    # Stop logging
    logger.stop()
    assert logger.running == False

    # Verify a new trip was created in the database
    self.store.cursor.execute("SELECT COUNT(*) FROM Trip")
    trip_count = self.store.cursor.fetchone()[0]
    assert trip_count == 1
```

Test 2: Upload

Description	Data Type	Expected Outcome	Test Type
User uploads the log data	Normal	Log data is uploaded	Manual

Procedure:

2. Press upload button
3. Verify data is sent to server
4. Verify data appears in cloud storage

Result:

Test 3: View

Description	Data Type	Expected Outcome	Test Type
User views the log data	Normal	Log data is displayed	Manual

Procedure:

2. Press view button
3. Display map view to show trip

[Screenshot of log data]

5 Evaluation

5.1 Introduction

This section is an evaluation of the project. The first section is based on user feedback, followed by an evaluation of each project objective against the system's preformance.

5.2 General User Feedback

"I have used francis' SLAP program which I accessed via the internet on my phone. I understood that this is the same as if I was using the system on my boat. Francis gave me a short tutorial explaining how to use the system and how to engage the simulator mode."

" I found the user interface easy to understand, perticularly because the functions of the system are as I would expect from other auto pilots. I used the simulator mode and then engaged the auto pilot imagining that I was the boat. I could see that the control system maintained the boat heading closely against the desired heading when the pilot is engaged. I was able to use the adjustment buttons to alter course as if I were avoiding obsticals. I was able to engage and disengage as and when needed. For example when I am tacking the boat."

"I found the compass display on the phone to be a useful addition to what you would normal see on an auto pilot. The ability to create trip logs is a nice feature not seen on most auto pilots. I thought it was very impressive that the logs could be uploaded to the internet to view at home."

"Overall I found the project to be impressive, and I am quite keen to have a go with the actuator really controlling the trim tabs on my boat to adjust the steering. An unexpected feature of francis' system is the low power consumption. From my experience the problem with commercial auto pilots is that it uses too much power as they must drive the rudders directly. With this system connected to the trim tabs the system can turn the rudder with little force requiring less power improving its feasiblility to engage the auto pilot for many hours."

Hand steering the boat onto the desired course and trimming the sails ready to balance the boat to engage the autopilot. Once using the autopilot I would be keeping an eye on the course from time to time. If the wind changed, I would need to alter the sails and the pilot to stay on course. When finishing I want a quick way of disconnecting it.

Get the boat sailing on required course with sails trimmed accordingly. Switch on autopilot to continue on desired course. Might need to interact with autopilot to avoid obstacles ahead eg other boats, buoys etc. Once clear of an obstruction set autopilot back on desired course. If I need to change direction, eg rounding a headland then I would adjust the course again using the controls on the autopilot. If I have to tack, I would put the autopilot on standby and once settled on the new tack, give control back to the autopilot. Once arrived at my destination I would switch the autopilot off and put it away.

5.3 Evaluation against objectives

Objective	Summary	Evaluation	Method	Status
1	Maintain steady course using closed loop control	The PID controller successfully maintains course within ±5 degrees in simulator testing	PID control system implemented and tested in simulator	Complete
2	Allow manual tuning during travel	Users can adjust heading via UI buttons and compass display	UI controls for course adjustment implemented and validated through user testing	Complete
3	Steer boat for 10+ minutes	System can run continuously for extended periods in simulator	Extended simulator testing sessions conducted	Complete
4	Accessible UI for marine conditions	Large buttons, clear compass display, high contrast design	User testing and feedback from experienced sailors	Complete
5	Log system performance	All course data and adjustments logged to database	Database logging implemented and verified	Complete
6	Database storage for passages	MongoDB database stores all trip data with unique IDs	Database schema designed and implemented	Complete
7	Plot passage on map display	Mapbox integration allows viewing saved routes	Map display and route plotting tested	Complete
8	Real-time course adjustment	PID controller makes continuous adjustments based on sensor data	Real-time control loop implemented and tested	Complete

6 Objectives and Requirements

General and Specific Objective

- Maintain a steady course using a closed loop control system
- Allow for manual tuning during travel, allowing for adjustment of angle to be made based on a compass bearing

- Steer the boat for at least 10 minutes allowing for other tasks to be preformed
- The user interface must be accessible for all users no matter the conditions on sea including angle adjustment and course viewing
- Performance of the system must be logged allowing for review
- A database should be used to store log data with entries for each individual passage made
- To be able to use the user interface to plot a passage from a map display
- The system must be able to automatically adjust its course in real time based on data input

6.1 Future Improvements