KSU AUV Software Documentation 2025

Ryan Slovensky

June 30, 2025

1 Introduction

This document contains information on how the code developed for the 2025 RoboSub competition functions. For the sake of clarity this document has been divided into two parts, one part containing information on how the code functions at a higher level, including documentation on which processes interact with one another and why. The second part is focused on documenting functions and classes within the main packages within the developed code.

2 Standard Operation

2.1 Background

The developed code consists of several 'packages' which perform certain tasks. The primary packages include:

- Flask Handler
- Hardware Interface
- Movement Package
- Camera Package
- AI Package
- Virtual Hardware Interface

Note that in a standard run only one of the Hardware Interface packages will be run.

The virtual variant is designed for testing within the developed Unity simulation, allowing for the Orin to pilot the sub within it. The regular Hardware Interface allows for the Orin to pilot the physical submersible within the real-world. The other packages remain that same across both real-world and simulation runs.

2.1.1 Flask Handler

The FlaskHandler package uses the Python Flask library to setup a web application that is hosted locally on the Orin. Other processes running on the Orin are able to access the application, as such several databases are hosted on this application for the other packages to GET and POST data, allowing for communication between the different packages. The databases created include the following:

- Inputs
- Outputs
- Sonar
- Batteries
- IMU
- Sensors
- Debug

These databases are accessed by navigating to "http://127.0.0.1:5000/(database name)". Attempting to navigate to a database URL within a browser will result in a blank page, as the pages aren't made to visually display the database. The databases when accessed return a json dictionary, with the key being the column header and the value being the most recent recorded value for that column header.

The databases themselves and how they are structured are detailed below.

Inputs: The Inputs database stores the values of data that will be sent to the submersible to change its behavior. In Autonomous mode this database is used to store the values that the AI Package has decided should be sent to the components of the Submersible to change what it is doing. Manual mode sees these values instead consist of values gained from the flight controller inputs. The database itself consists of the following fields:

- id: Used to differentiate input values sent at different times.
- X: Value that impacts the sub's position on the X-axis.
- Y: Value that impacts the sub's position on the Y-axis.
- Z: Value that impacts the sub's position on the Z-axis.
- Roll: Value that impacts the sub's Roll.
- Pitch: Value that impacts the sub's Pitch.
- Yaw: Value that impacts the sub's Yaw.
- Torp1: Value used to determine if the torpedo should be fired.
- Torp2: Value used to determine if the torpedo should be fired.
- Claw: Value used to determine if the Claw should be activated.

Outputs: The Outputs database stores the values of data that has acted on by the submersible. The database essentially acts as a log for what the sub has done at a given time. The database itself consists of the following fields:

- id: Used to differentiate input values sent at different times.
- M1: Value that is sent to motor 1.
- M2: Value that is sent to motor 2.
- M3: Value that is sent to motor 3.
- M4: Value that is sent to motor 4.
- M5: Value that is sent to motor 5.
- M6: Value that is sent to motor 6.
- M7: Value that is sent to motor 7.
- M8: Value that is sent to motor 8.
- Torp1: Value used to determine if the torpedo should be fired.
- Torp2: Value used to determine if the torpedo should be fired.
- Claw: Value used to determine if the Claw should be activated.

Sonar: The Sonar database stores data gained from the Sonar component. The database stores information on where obstacles are in relation to the sub. Specifically it records the angle at which an obstacle is, and the approximate distance the obstacle is in relation to the sub. The database itself consists of the following fields:

- id: Used to differentiate input values sent at different times.
- Distance: Used to store how far away a detected obstacle is in relation to the sub.
- Angle: Stores the angle where an obstacle is in relation to the sub.

Batteries: The Batteries database stores information on the batteries that power the submersible. This database is mainly for trouble shooting and safety, allowing for determining if a problem exists with the batteries while the sub is running. The database itself consists of the following fields:

- id: Used to differentiate input values sent at different times.
- Voltage1: Records the voltage of battery 1.
- Voltage2: Records the voltage of battery 2.
- Voltage3: Records the voltage of battery 3.
- Current1: Records the current of battery 1.
- Current2: Records the current of battery 2.
- Current3: Records the current of battery 3.
- Error: Records if an error has occurred during operation.

IMU: The IMU database stores information from the IMU component. The stored data consists of acceleration and orientation of the sub at a given point in time. The database itself consists of the following fields:

- id: Used to differentiate input values sent at different times.
- AccelX: Records the acceleration of the sub on the X-axis at a given time.
- AccelY: Records the acceleration of the sub on the Y-axis at a given time.
- AccelZ: Records the acceleration of the sub on the Z-axis at a given time.
- GyroX: Records the orientation of the sub in relation to the X-axis.
- GyroY: Records the orientation of the sub in relation to the Y-axis.
- GyroZ: Records the orientation of the sub in relation to the Z-axis.

Sensors: The Sensors database stores information from the rest of the sensor components on the sub. The database itself consists of the following fields:

- id: Used to differentiate input values sent at different times.
- Temp: Records the temperature of the electrical housing.
- Humidity: Records the humidity of the electrical housing.
- Pressure: Records the pressure within the electrical housing.
- Depth: Records how deep the sub is.
- Heading: Records the direction that the sub is currently traveling.

Debug: The Debug database records errors that have occurred during runtime. The database itself consists of the following fields:

- id: Used to differentiate input values sent at different times.
- Package: Records the package where the error has taken place.
- ErrorType: Records the type of error that has occurred.
- Message: Records the error message of the given error.

2.1.2 Camera Package:

The Camera package accesses two camera. One is a ZED 2i camera, which acts as the front-facing camera for the sub. The other is an Anker Power Conference C200 camera, which is used as the anchor camera to allow the sub to observe what is below it.

The feed from bother cameras is stream to the web application setup by the FlaskHandler process. The feeds themselves being accessed using "http://127.0.0.1:5000/video_0" and "http://127.0.0.1:5000/video_1". 'video_0' contains the feed from the ZED camera while 'video_1' contains the feed from the anchor camera.

To avoid unnecessary overhead the Camera Package has been split among several files. The file 'CameraPackage.py' only contains data and functions that other packages may need, while the bulk of the Camera Package's processing occurs in the files within "src/modules/CameraPackageSupport". The files within this folder are detailed below.

WebCamService: This file allows for other processes to easily access the cameras by creating a class 'WebCam'. The 'WebCam' class contains the following attributes:

- ip: stores the IP address of the WebCam.
- camera_number: stores the number used to identify the different cameras.
- capture: stores the latest captured image from the camera.

The 'WebCam' class also has two functions:

- get_frame(self, capture): This function captures an image from a given video stream that is passed to the function. The captured image is then converted to jpeg format and then returned as bytes.
- crop_frame(self, frame): This function crops a frame that is passed in the frame argument. This function is mostly used for the ZED camera given that it returns two images, one from each lens.

routes: This file is used to provide the web application with images from the cameras. To do this the file contains two functions that both utilize the WebCam class mentioned previously:

- gen(webcam): This function uses a provided WebCam object to access the cameras and continuously provide an image of what the camera's see. The function handles the ZED camera differently due to having to convert from the ZED camera image format to OpenCV's format. To allow for the camera feed to be provided continuously the 'yield' function is used, which provides images in jpeg format.
- monitoring(): This function is called when "http://127.0.0.1:5000/stream" is accessed. The function simply provides an image of what the camera currently sees.
- get_blueprint(): This function returns a Flask blueprint, essentially a programmer-defined webpage, for the main app module.

camera_1 and camera_2: These files do the same thing, with camera_1 being for the ZED camera, and camera_2 being for the anchor camera. Both files use a pre-defined flask blueprint which is accessed by navigating to "http://127.0.0.1:5000/video_0" and "http://127.0.0.1:5000/video_1". Navigating to either of these pages will call the function video_0 or video_1, which will create a WebCam object and then call the gen(webcam) function from 'routes.py'. This will provide the web application pages with a stream of what each camera currently sees.

2.1.3 Hardware Interface

The Hardware Interface allows for the Orin, the brain of the sub, to communicate with the physical hardware that makes up the sub. The interface main purpose is to allow for communication with sensors and motors.

The Hardware Interface is currently divided among two different files, 'HardwareInterface.py' and 'MPU6050.py'. 'HardwareInterface.py' is the primary package for the hardware interface, 'MPU6050.py' merely providing functionality for the MPU sensor on the sub.

HardwareInterface.py: This file contains the HardwareInterface class. The class itself contains the following attributes:

- bus: A variable used to handle communications through the System Management Bus.
- addresses: A dictionary that holds the addresses of each component that interacts with the hardware interface. The address being where on the PCB board a given component is connected.
- url: Holds the url where the hardware interface can be accessed on the web application.
- mpu: Used to hold the MPU object that is created to allow for accessing the MPU and the data it generates.

The HardwareInterface class also contains several functions detailed below:

- read_i2c_word(self, register): This function reads data from two i2c registers, the register argument referring to the first register to read from. The second register to read from is the one following the first register. This function allows for getting data from the connected components.
- convert_temp(self, data): This function converts provided temperature data into Celsius. The data argument is a list of two bytes which contains the raw temperature data from the temperature sensor inside the electrical housing.
- convert_humi(self, data): This functions converts provided humidity data into a percentage. The data argument is a list of two bytes which contains the raw humidity data from inside the electrical housing.
- log_data(self, type_of_data, data): This function logs gathered data to the main.log file. The type_of_data argument is a string that indicates the type of data being logged, the types including Debug, Info, Warning, Error, and Critical. The data argument is a dictionary that contains the collected data to be logged.
- print_data(self, data, data_type): This function sends the provided data, and the provided type of said data to the debugger.
- handle_error(self, error): This functions logs an error passed as an argument.

- get_data(self): This function attempts to retrieve data from the '/sensors' page on the web application. The retrieved data is returned as a dictionary. Should the function be unable to retrieve data then an error will be logged.
- send_data(self, data): This function attempts to send the web application data to be stored at the '/sensors' page on the web application. The data argument contains the data to be sent to the web application. The return value of the function is the response from the server as a string.
- write_ESCs(self, data): This function writes data passed to the function as the 'data' argument to the electronic speed controllers. Should the attempt to write data fail then an error will be logged.
- write_Arm(self, data): This function writes a provided dictionary, provided by the data argument, to the robotic arm attached to the sub. Should an issue with writing to the arm occur then an error is logged.
- read_Temp(self): This function reads data from the temperature sensor attached to the sub. Once the raw data is retrieved it is then sent to the convert_temp function, with the resulting value then being returned. Should the attempt to read the temperature data fail then an error is logged.
- read_BatteryMonitor(self): This function reads data from the battery monitor attached to the sub. The function returns the voltage of the battery as a float value. Should the function be unable to read data from the battery monitor then an error is logged.
- read_Depth(self): This function will use an attach component to determine the current depth of the sub at a given time. It is currently unfinished.
- read_Hydrophones(self): This function will use the attached hydrophones to help the sub navigate in the pool. It is currently unfinished.
- read_MPU6050(self): This function reads data from the MPU unit attached to the sub. The function returns a dictonary containing the data. Should the attempted read fail then an error is logged.
- test_ESCs(self): This function tests the ESCs by sending them values for a period of four seconds and then sending different values. By paying attention to the motors while this function is being run we can determine if the ESCs can be written to.
- test_BatteryMonitor(self): This function tests communication with the battery monitor. It is currently unfinished.
- test_Arm(self): This function tests communication with the robotic arm. It is currently unfinished.
- test_Depth(self): This functions tests communication with the depth sensor. It is currently unfinished.
- test_Hydrophones(self): This function tests communication with the hydrophones. It is currently unfinished.

- test_MPU6050(self): This function tests communication with the MPU. It is currently unfinished.
- test_suite(self): This function prompts the user for which tests they would like to run. The chosen tests are then called.

MPU6050.py: This file contains code associated with the MPU component. The code is in this file instead of the HardwareInterface.py file to avoid unnecessary overhead. Other processes that interact with the Hardware Interface don't need access to the MPU directly, only its data. As such separating these functions so only the HardwareInterface.py file has access to them has been done. The MPU6050.py file is composed of a MPU6050 class definition. The class has the following attributes:

- GRAVITY_MS2: This attribute defines the acceleration of gravity in meters per second.
- address: This attribute defines the address of the MPU.
- bus: This attribute defines which System Management Bus the MPU uses.
- ACCEL_SCALE_MODIFIER_2G:
- ACCEL_SCALE_MODIFIER_4G:
- ACCEL_SCALE_MODIFIER_8G:
- ACCEL_SCALE_MODIFIER_16G:
- GYRO_SCALE_MODIFIER_250DEG:
- GYRO_SCALE_MODIFIER_500DEG:
- GYRO_SCALE_MODIFIER_1000DEG:
- GYRO_SCALE_MODIFIER_2000DEG:
- ACCEL_RANGE_2G:
- ACCEL_RANGE_4G:
- ACCEL_RANGE_8G:
- ACCEL_RANGE_16G:
- GYRO_RANGE_250DEG:
- GYRO_RANGE_500DEG:
- GYRO_RANGE_1000DEG:
- GYRO_RANGE_2000DEG:

The class also contains other attributes which contain the address of MPU registers on the PCB board. The registers consist of power information, testing information, acceleration, temperature, orientation, and the settings for acceleration and orientation. These attributes include:

- PWR$_MGMT_1 = 0x6BPWR_MGMT_2 = 0x6C$
- SELF$_TEST_X = 0x0DSELF_TEST_Y = 0x0E$

- SELF$_{T}EST_{Z} = 0x0FSELF_{T}EST_{A} = 0x10$
- ACCEL$_{X}OUT0 = 0x3BACCEL_{X}OUT1 = 0x3C$
- ACCEL$_{Y}OUT0 = 0x3DACCEL_{Y}OUT1 = 0x3E$
- ACCEL$_{Z}OUT0 = 0x3FACCEL_{Z}OUT1 = 0x40$
- TEMP$_{O}UT0 = 0x41TEMP_{O}UT1 = 0x42$
- GYRO$_{X}OUT0 = 0x43GYRO_{X}OUT1 = 0x44$
- GYRO$_{Y}OUT0 = 0x45GYRO_{Y}OUT1 = 0x46$
- GYRO$_{Z}OUT0 = 0x47GYRO_{Z}OUT1 = 0x48$
- ACCEL$_{C}ONFIG = 0x1CGYRO_{C}ONFIG = 0x1B$ The MPU class also contains several functions including:
  - init(self, address): This functions takes a provided address which is then set as the primary address for the MPU. The function also wakes up the MPU by sending data to the PWR_MGMT_1 register.
  - read$_{i}2c_{w}ord(self, register) : Thisfunctiontakestheregisterargumentandreturnsthecontentsofthatregi$
    $ThisfunctionreturnsthetemperaturetheMPUsensesinCelcius.ThisisdonebyreadingtheTEMP_{O}UT0re$
- set_accel_range(self, accel_range): This function changes the range of acceleration values that the MPU can record. The argument accel_range is written to the ACCEL_CONFIG register.
- read$_{a}ccel_{r}ange(self, raw = False) : Thisfunctionreturnsanumbercorrespondingtotherangetheacceleromete$
  $1. -1indicatingthatanerrorhasoccurred.Notethatthesereturnvaluesonlyoccurifthe argumentrawissettofalse$
  $False) : ThisfunctionreturnstheaccelerationalongtheX, Y, andZaxisinm/s^2ifg =$
  $False, andintermsofGifg = True.Thedataisreadusingtheread\_i2c\_wordfunctiontoreadtheACCEL_{X}OUT0,$
- set$_{g}yro_{r}ange(self, gyro_{r}ange) : ThisfunctionchangestherangeofgyroscopevaluesthattheMPUcanrecord.Th$
  $False) : Thisfunctionreturnsanumbercorrespondingtotherangethegyrometerissetto.Thereturnvaluescanbe$
  $1. -1indicatingthatanerrorhasoccurred.Notethatthesereturnvaluesonlyoccurifthe argumentrawissettofalse$
- get$_{g}yro_{d}ata(self, g = False) : ThisfunctionreturnstheorientationindegreesalongtheX, Y, andZaxis.Thedata$

2.1.4 Movement Package

The Movement Package is concerned with allowing the sub to move in an intelligent way. This is done by translating inputs received either from the AI or a controller into data that when sent to the motors will produce the desired results. In order to do this a class MovementPackage has been defined in MovementPackage.py. The class has the following attributes:

- ip: Defines the IP address that the MovementPackage object posts data to.
- port: Defines the port that the MovementPackage object posts data to.
- debug: References the universal debugger code used in all packages.

- horizontalMotors: A list that contains values associated with motors 1 through 4.
- verticalMotors: A list that contains values associated with motros 5 through 8.
- horizontalInputs: A list that contains input values for the X-axis, Y-axis, and Roll.
- verticalInputs: A list that contains input values for the Z-axis, Pitch, and Yaw.
- input_data: A list that contains input values to be transformed into output values to move the sub.
- horizontalMapping: A numpy array that contains mappings for the X-axis, Y-axis, and Yaw.
- verticalMapping: A numpy array that contains mappings for the Z-axis, Pitch, and Roll.
- output_data: A dictionary that contains values translated from input into a form that the sub can use to change its movement.
- deadzone: A number that defines the controller deadzone to prevent small values from being picked up by the package.

The MovementPackage class also contains several functions which are detailed below:

- mapping(self, x):
- get_data(self): A function that retrieves input data from the flask web application. The retrieved data is then stored in the input_data attribute.
- split_data(self): A function that separates input data into the horizontalInputs and verticalInputs attributes.
- calculate_motor_speeds(self): A function that ensures that input values are above the deadzone attribute's value. If above the deadzone value then the verticalInput attribute's value is transformed.
- send_data(self): This function sends the transformed input data to the outputs database on the flask web application.
- print_data(self): This function logs the input_data, horizontal_Inputs, vertical_Inputs, and output_data to the log file.
- handle_error(self, error): This function prints the error that has occurred to the terminal.
- test_get_inputs(self): This function runs the get_data function.
- test_calculate_motor_speeds(self): This function provides test input which is sent to the split_data, calculate_motor_speeds, and print_data functions to ensure that the functions handle the data correctly.

- test_send_data(self): This function sends data to the web application using the send_data function. The sent data is also logged using the print_data function allowing for the user to check if data was properly sent to the web application.
- run(self): This function controls the standard operation of the Movement-Package object. Within a While True loop the get_data, split_data, calculate_motor_speeds, and send_data functions are run in this order. Should debug mode be enabled then the print_data function is run last for each loop.

The Movement Package also uses another file PID.py which provided a Proportional Integral Derivative (PID). The PID allows for the sub to more intelligently reach its destination, which we call the set-point. The issue is by default the sub cannot reach the set-point exactly due to gravity dragging the sub down and any current pushing the sub away from the set-point. As such to have the sub remain at a set-point we need to have the motors move the sub. The issue is that the motors when moving the sub will typically overshoot the set-point due to momentum, this continuing as the sub sees that it needs to reach the set-point essentially ping-ponging across the set-point constantly. To prevent this from occurring we can use a PID to calculate the force we want the motors to generate to get to that set-point while accounting for the sub's momentum from activating the motors. We can predict how much force we need from the motors by taking the derivative of the velocity of the sub, allowing us to predict the location of the sub and then turning off the motor when we have enough momentum to reach the set-point. We then take the integral of how far we are from our set-point historically, the calculated integral increasing or decreasing the force we request the motor to produce to get to the set-point.

PID.py The PID.py file is mostly composed of a class definition for a PID object. The object contains the following attributes:

- Kp: The value for the proportional gain Kp
- Ki: The value for the integral gain Ki
- Kd: The value for the derivative gain Kd
- setpoint: The initial setpoint that the PID will try to achieve
- sample_time: The time in seconds which the controller should wait before generating a new output value. The PID works best when it is constantly called (eg. during a loop), but with a sample time set so that the time difference between each update is (close to) constant. If set to None, the PID will compute a new output value every time it is called.
- output_limits: The initial output limits to use, given as an iterable with 2 elements, for example: (lower, upper). The output will never go below the lower limit or above the upper limit. Either of the limits can also be set to None to have no limit in that direction. Setting output limits also avoids integral windup, since the integral term will never be allowed to grow outside of the limits.

- auto_mode: Whether the controller should be enabled (auto mode) or not (manual mode).
- proportional_on_measurement: Whether the proportional term should be calculated on the input directly rather than on the error (which is the traditional way). Using proportional-on-measurement avoids overshoot for some types of systems.
- differential_on_measurement: Whether the differential term should be calculated on the input directly rather than on the error (which is the traditional way).
- error_map: Function to transform the error value in another constrained value.
- time_fn: The function to use for getting the current time, or None to use the default. This should be a function taking no arguments and returning a number representing the current time. The default is to use time.monotonic() if available, otherwise time.time().
- starting_output: The starting point for the PID's output. If you start controlling a system that is already at the setpoint, you can set this to your best guess at what output the PID should give when first calling it to avoid the PID outputting zero and moving the system away from the setpoint.

The PID class also contains the following functions:

- __call__(self, input_, dt): This function updates the PID controller. Call the PID controller with *input_* and calculate and return a control output if sample_time seconds has passed since the last update. If no new output is calculated, return the previous output instead (or None if no value has been calculated yet). The parameter dt if set will use its value for timesetp instead of real time.
- components(self): This function returns the values of P, I, and D from the last computation inside of a tuple. It is used to visualize what the controller doing.
- tunings(self): This function returns the tunings used by the controlled as a tuple. The tuple including the Kp, Ki, and Kd values.
- tunings(self, tunings): This function sets the Kp, Ki, and Kd values equal to the values provided by the tunings parameter.
- auto_mode(self): Returns whether the controller is enabled or not. If in auto mode then the controller is enabled.
- auto_mode(self, enabled): This function allows for the auto mode to be enabled or disabled depending on the value of the enabled parameter.
- set_auto_mode(self, enabled, last_output): Enable or disable the PID controller, optionally setting the last output value. This is useful if some system has been manually controlled and if the PID should take over. In

that case, disable the PID by setting auto mode to False and later when the PID should be turned back on, pass the last output variable (the control variable) and it will be set as the starting I-term when the PID is set to auto mode.

- output_limits(self): Returns the current output limits as a tuple, the first value being the lower limit and the second value being the upper limit.
- output_limits(self, limits): Sets the _min_output and _max_output to the values provided by the limits parameter.
- reset(self): Reset the PID controller internals. This sets each term to 0 as well as clearing the integral, the last output and the last input (derivative calculation).

2.1.5 Sonar Pacakge

The Sonar Package is separate from the Hardware Interface package due to concerns of unnecessary overhead of including its code in the package and its need to constantly be operating to be useful.

The sonar used for the package is the Ping360, which has an effective range of 60 meters (175 feet). The current (2025) AUV submersible powers the sonar using 16.8 Volts, with amperage varying from .67 to .69.

This code operates by sending out a signal at each angle, note that this is innately in gradians (gradian * .9 = angle) for the ping360. Using the transmitAngle(gradian) we can control the gradian at which the signal is sent out. This is done for all 400 gradians to get a 360-degree view of the surroundings.

The transmitAngle(gradian) method returns a pingmessage which we can extract data from, we can then perform operations to get the signal intensity from it and the distance at which the intensity occurred.

The SonarPackage.py file defines a Sonar class which contains the following attributes:

- p: This attribute defines a Ping360 object from the Blue Robotics's Ping360 library.
- url: This defines the URL where the sonar data will be posted.
- logData: This defines the data to be logged.
- debug_handler: This holds a reference to the DebugHandler object that is used to log data from the sonar.

The Sonar class also contains the following functions:

- __init__(self, ip, port) This function sets up the sonar by running the connect_sonar and initialize_sonar_settings functions. It also uses the provided ip and port parameters to connect to the web application where its data will be posted.
- connect_sonar(self): This function starts a serial connection between the sonar and the Orin.

- initialize_sonar_settings(self): This function initializes the sonar settings. This includes settings for transmission frequency, sample period, number of samples, gain settings, and transmission duration.
- calculate_sample_distance(self, ping_message, v_sound): This function calculates the distance that each sample covers. This is done by multiplying the sample's period, v_sound (sound in water), and the constant 12.5e-9.
- filter_data_within_range(self, data, dist_per_sample, lower_limit): This function filters collected data within a certain radius around the sonar due to a 'deadzone' of .75 meters around the sonar.
- detect_highest_intensity(self, data): This function identifes the highest intensity reading within the data collected from a gradian. The highest intensity indicating that an object is likely located at that point.
- process_scan(self, gradian): This function processes a sonar scan that occurred at a provided gradian. This consists of transmitting a signal at that gradian, calculating the distance that each sample covers, retrieving the data from the buffer, filtering data within the deadzone, and then returning the point where an obstacle has occurred.
- log_and_send_data(self, gradian, highest_value, highest_index, dist_per_sample): This function logs instance where an obstacle likely is by checking if the intensity of the signal passes a pre-defined threshold. If so then the angle of the obstacle and its distance from the sub are sent to the web application using the send_data function.
- send_data(self): This function sends data stored in the logData attribute to the web application. run(self): This function operates the sonar. Within a While True loop a for loop iterates through all gradians gathering data, logging it, and then sending it to the web application.

2.1.6 AI Package

2.1.7 Debug Handler

To track errors and data during runtime the Debug Handler package was created. This package is imported into the majority of packages to provide logging functionality. The DebugHandler.py file itself contains the class definition for a DebugHandler object which includes the following attributes:

- Package: Contains a string value that describes the package that this DebugHandler object is associated with.
- ip: Contains the IP address where debug data will be posted.
- port: Contains the port number where debug data will be posted.
- MessageType: Used to store the type of message that will be posted to the web application.
- Message: Used to store the message that will be posted to the web application.

The DebugHandler class also contains the following functions:

- __init__(self, Package, ip, port): Initializes the Package, ip, and port attributes with the provided values. The MessageType and Message attributes are set to None.
- set_data(self, MessageType, Message): Sets the MessageType and Message attributes to the provided values. Then calls the send_data function.
- send_data(self): This function converts the Package, MessageType, and Message attributes into a dictionary which is then posted to the web application.