Extended abstract for USV obstacle avoidance:

NS

General idea: detect objects, decide if they’re dangers to you, take appropriate action

LiDAR: **Li**ght **D**etection **A**nd **R**anging (or Light Imaging Detection And Ranging). Lidars are time of flight sensors which use light as their ranging method. The one used for this project (SlamTec’s RPLIDAR A1M8-R6) is not three dimensional. It only measures the range of items visible on the plane of its rotation. If an object does not intersect the plane, it is invisible to the sensor. The sensor consists of 6 parts. The transmitter, the receiver, the housing, the motor, the pcb and a converter.   
The transmitter and receiver are placed on a cylinder above the rest of the sensor, so they are unobstructed. The transmitter sends a beam of light at a specific time. Should the light beam encounter something in its flightpath, it might bounced back with sufficient energy remaining so that the receiver picks it up. To get a 360 degree view, both the transmitter and receiver are rotated together. To move this cylindrical housing, a motor is present at the bottom of the sensor. It powers a gear through the sensor’s body, that extrudes to the top, which is connected to a belt. This thin belt is held across the circumference of the larger cylindrical housing of the receiver and transmitter.   
Next to the motor, on the underside of the sensor lies the PCB. It controls the sensor and interprets its data. It uses non-standard cabling, so a converter is used so that the sensor can be connected using a a Micro-USB cable.

Libraries used:   
rplidar by Roboticia.  
rplidar was used as it was made specifically for this model of Lidar. Other libraries such as pylidar, Pylidar3 etc were made with other sensors in mind, and thus had trouble with their usage.  
Array:  
Used to create non-tuple2D arrays.  
Sys:  
-  
random:  
to generate fake data to test the code.  
math:  
needed to calculate cosines and square roots in some parts of the code  
time:  
currently used for testing, can evolve into usage for velocity calculations.

Get\_scan():   
GetScan is a function that uses the rplidar library’s iter\_measurements. Iterates through 2 scans and returns the second cscan. A scan in this context is a complete rotation of the sensor, which returns data on any angle where it detected something. 2nd scan is used instead of the first as the first scan is of reduced size and does not detect as many objects. Lidar.stop() clears out the buffer of the sensor (so function can be called multiple times consecutively with minimal time losses)

Arr is a 2D array of the first dimension being an object id (1, 2, 3, …., x-2, x-1, x) with X being a number between 70-130, as those are the numbers of measurements made per scan. 2nd dimension is a tuple of 3 doubles/floats: Quality (nicknamed Strength). The quality of the returning signal.

A screen shot of a computer

Description automatically generated with low confidence

Process\_scan(Matrix, RatioVariable, FlagNum, StrengthThres):

Process\_scan is the first stage of data processing. It receives an array of tuples (typically the output of the get\_scan() function), along with 3 parameters.

RatioVariable – double.

FlagNum – integer.

QualityThres – double

The working principle of the function is as follows:

Process\_scan receives as input an array. It iterates through that array, comparing the data of the first cell with the one after it. should the two cells have satisfactory values (reasoning and methods of checking that below), they are flagged. This flag indicates that these two points are “connected”. If a number of flags is reached, then the cluster of measurements is labeled a “spot” and after some calculations, it is added to the list that process\_scan outputs.

The calculations that lead to two measurements being flagged are the following.

First we measure the difference between the angles of the two points. Should it exceed a hardcoded threshold (currently sitting at 5, as sensor has not been shown to exceed 4 degree jumps), it continues with the measurements.

The difference of two consecutive points is calculated. That difference is divided by the largest distance (to obtain a larger ratio and counteract sensor irregularities). Should that difference be below the function parameter RatioVariable, it increases a flag variable by one.

When a ratio is not smaller than the ratiovariable, the programme checks for the length of the flag variable. Should it be larger than FlagNum, the other function parameter, the points are treated as an object. This is done for the whole length of the passed array, with the flag being zeroed each time a ratio fails the RatioVariable check.

Should all of the above checks be met, the programme now treats the numerous measurements as 1 object. A spot. This spot needs to have some key variables derived from so following steps are easier to deal with, logically and computationally.

Each spot is passed into the output array of *Spotslist[][]*, first dimension is object IDs. The second dimension of the array is a tuple of numerous values. These are:

[0] – MQ – Mean Quality – average quality of the flagged measurements.

[1] - MA – Mean Angle – Angle value of the middle measurement.

[2] – MD – Mean Distance – average distance of the flagged measurements.

[3] – ML – Length – an approximation of the true length of the detected spot.

[4] – SD – Shortest Distance – the shortest distance between the spot and the sensor.

[5] – Sda – Shortest Distance Angle – the angle of the spot’s shortest distance.

Lastly, we have Compare\_Scans(Scan1, Scan2, QualityThres, AngleThres, DistanceThres, LengthThres)

This function takes two spotslists, and tries to figure out which spot shows continuity, which one appears to both scans. It compares the variables of each scan1 spot, with the variables of each scan2 spot. It then outputs the spots that are within thresholds.