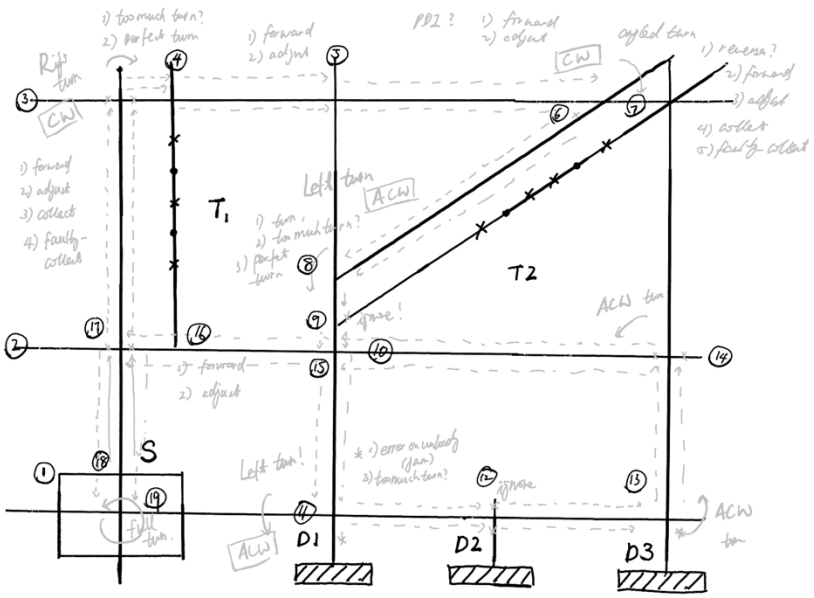
Software Sub-team

Overview

Overall, the design of the software is driven by three basic principals: modular, dynamic and stable. Modularity meaning the software will be containing many functions breaking general tasks to many specific sub-tasks. Dynamic meaning the robot should know its exact location and when needed will handle specific cases individually (explained in details below). Finally, stable is the sensitivity of individual functions (in particular the plant detection function) to environmental noise, also the possibility of memory leakage, infinite while loops, etc. should be eliminated.

Figure below shows the course taken by the robot. The course has been broken down at each node. Table below shows the functions that maybe called at each stretch of the course.
The purpose of memory of the exact location (node) of the robot is to handle specific cases if needed. Given the short time constraint, it is likely some final changes will be made physically to the robot which may affect the robot at specific stretches (i.e. from 6 – 7), changing the general functions' parameters will affect the robot's performance at other non-problematic stretches. Hence, by breaking the functions down to specific tasks and by making the robot remember exactly where it is will speed up the process of troubleshooting and error handling.



Nodes	Functions (probably) required
1	F
2	PF, C
3	CW, F
4	F
5	F
6	CW
7	CW, PF, C
8	ACW, F
9	F
10	F
11	E, ACW
12	F
13	E, ACW
14	F
15	F
16	F
17	ACW, F
18	F
19	ACW

Week 1

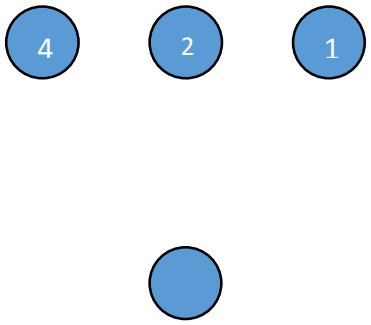
During week 1, the software sub-team investigated the following:

i. How motor behaves when a moment is added to one side of the robot (simulating weight of the picking mechanism)
ii. Rudimentary line following algorithm (robot following straight lines)
iii. Timing of the codes run on workstation and microprocessor
iv. Overall software layout and communication between functions
v. Pin assignment with the electrical team

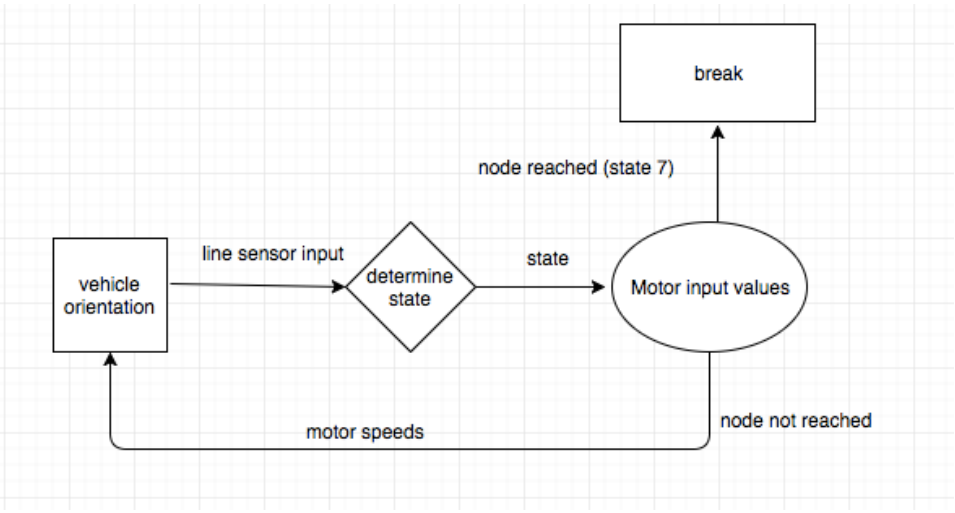
Line Following Algorithm

The initial Design for the arrangement of the line sensors is to have 3 horizontal line sensors which are the only sensors used when doing the line following. If an outside line sensor is activated along with the middle one the direction the vehicle is slowly drifting in so the motor speeds can be readjusted in order to straighten the path of the robot. 3-bit number which represents the state of these 3 lines sensors is imported and can manually set motor speeds for each state in order to counteract any deviation in path.
The blue circles on the right represents the configuration of the 4 light sensors, and each of the top 3 circles (which are the line sensors used for the line following) can be converted from its binary number to its decimal number. Below is a basic breakdown of how the robot should deal with each state that it's in.

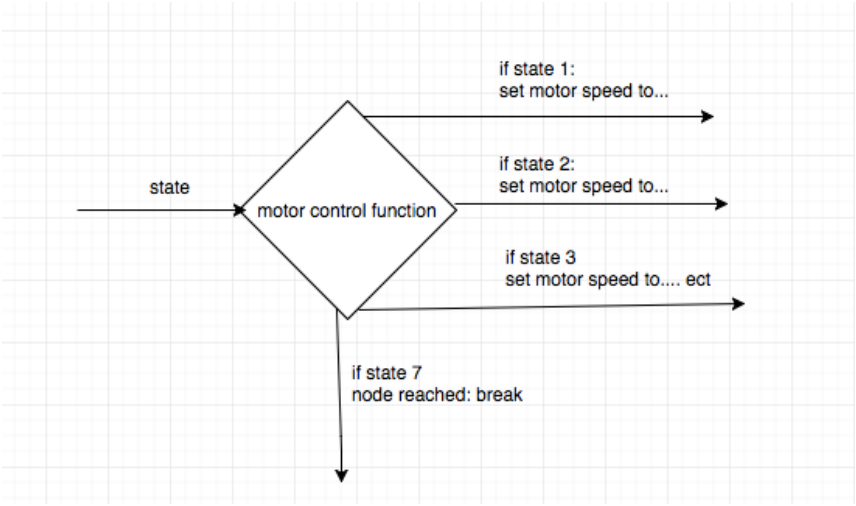
0	Robot lost path- recovery function
1	Sharp turn right
2	On track- continue straight
3	Slight turn right
4	Sharp turn left
5	45 degree node
6	Slight turn left
7	90 degree node reached



The general idea with this arrangement of light sensors was that if the outer sensors are close enough to the middle sensor then deviations from path can be detected early and be able to readjust the angles before they get too large and hopefully reduce the effect of oscillation. Here is the state diagram of the line following function (The general form of the function has been created)

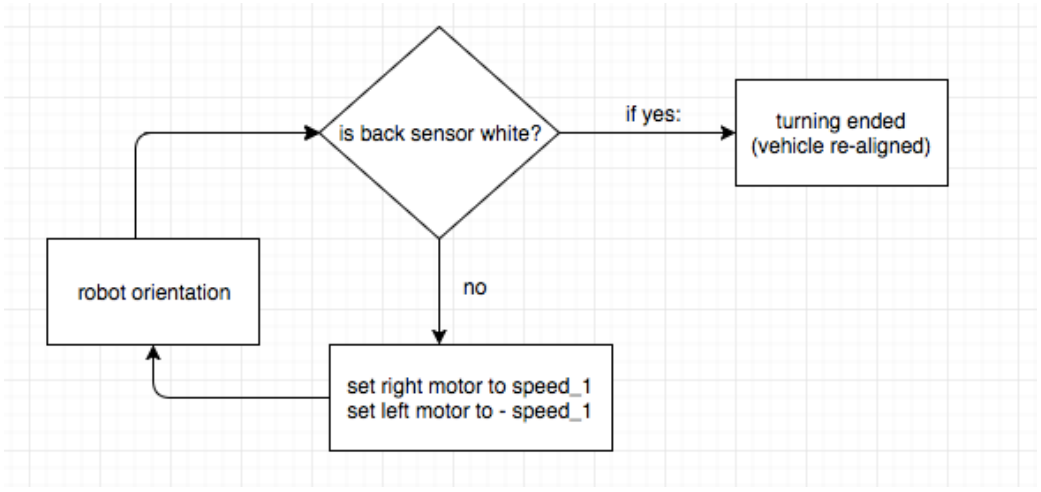


The line sensors input to a function which outputs to the motors, setting their speeds to correct the vehicles path and then this runs on a loop until the node has been reached



And this is the very basic function which uses if statements to check the state and then motor speeds are input manually.

Now based off of the previous line following algorithm it seems that the back light sensor is rendered useless. However, the team plans on using the back sensor for turning: the robot will only attempt turning at a node so the robot will continually turn until the back sensor will align itself well with the line which is 90 degrees to the original direction. Once this is complete the robot can simply break back into the line following algorithm and continue moving forward. Below is the state diagram for the turning function:



Setting the motor speeds opposite values hopefully turns the robot about a point in between the two motors, so if we place the 3 front line following light sensors at this pivot point the robot should hopefully still be roughly aligned to the new direction. Two things to note which was not displayed in the state diagram: at the start of turning the black sensors will not be read as it might just read the current white line it is on, and also once the turning is the robot will take a few steps forward so it doesn't just immediately exit the function since it's technically on a node.

Plant Detection

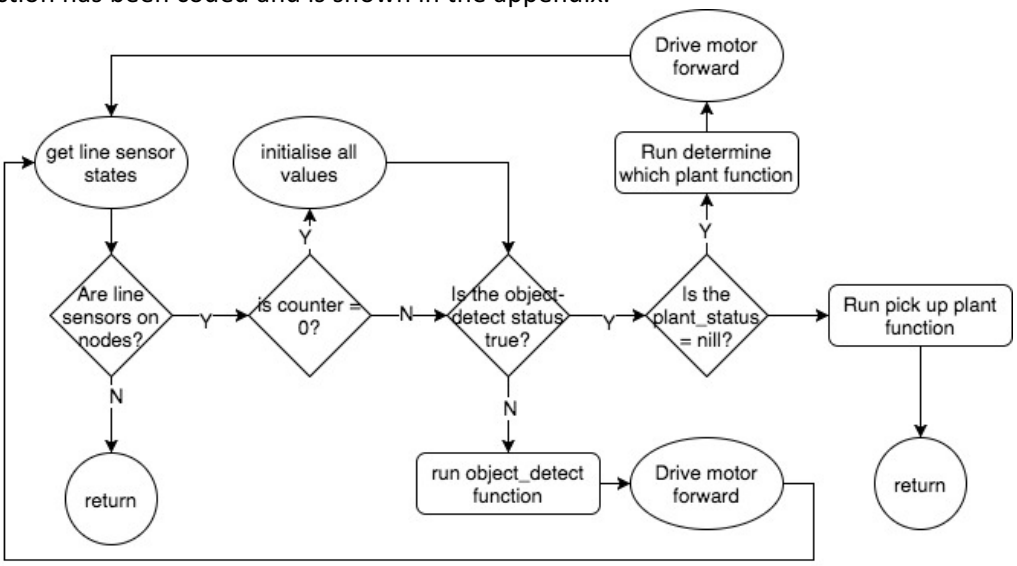
Algorithm Overall Design Principal:
Following from the main principles laid out from the overview section, the algorithm consists of four separate functions addressing different aspects of the plant detection stage (modular). Different thresholds can be determined through experiments and can be altered easily (dynamic). Regression is used to ensure singular readings do not affect the overall plant detecting decisions (stable).

Algorithm Overview:
There are four functions that make up the overall plant detection algorithm. The *plant_forward(), object_detect(), determine_plant()* and *plant_analysis(). Plant_forward()* is the main function that is called within the main loop, *object_detect()* determines whether there is a plant (of any kind) viewed by the sensors. *Determine_plant()* records all the plant's profile before finally calling *plant_analysis()* which determines the exact type of plant viewed by the robot. Each function is explained in detail below.

Plant_forward()
The block diagram below shows the overall logic within the *plant_forward()* function starting from "get line sensor states" (top left block). The objective of this function is to:

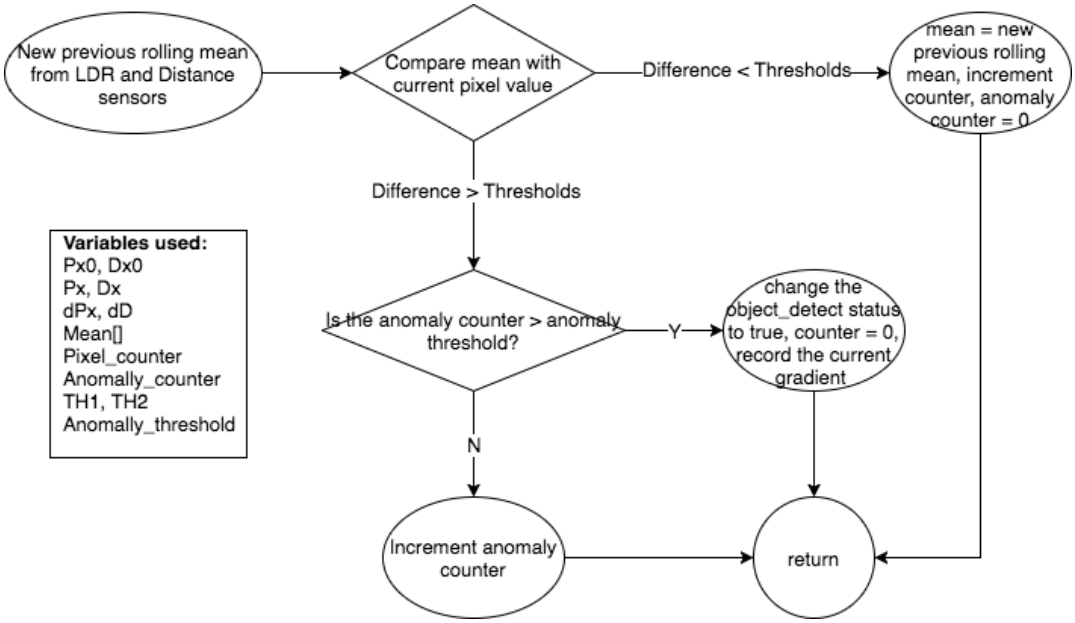
1. Keep on running the *object_detect()* function and the motors when there is no pixels belonging to any parts of the plant detected.
2. Once it is certain that part of a plant has been detected (object-detect status is true), the function calls for *determine_plant()* continuously until the exact type of plant (undersized/normal cabbage/cauliflower).
3. Once the exact type of plant has been determined, the function calls for the subroutine that activates plant harvesting mechanisms.

This function has been coded and is shown in the appendix.



Object_detect()
See the figure below for the block diagram. As the function is called, it computes the difference between the current LDR and distance sensors' readings with the previous rolling mean. If the absolute difference is smaller than the set threshold, the function includes the current readings into the rolling mean. If the absolute difference is greater than the set threshold, the function increments a value that indicates a "suspicious" pixel has been detected. This anomaly counter is used to ensure that faulty readings do not produce faulty decisions, "suspicious" pixels are not included in the rolling mean calculation. Once the number of suspicious pixels have reached a set value, the *object_detected_status*

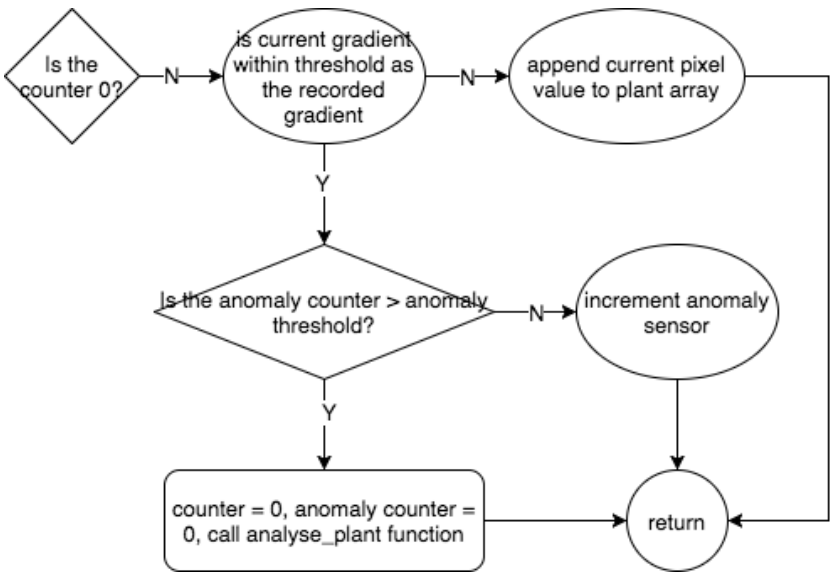
is set true which indicates to the previous *plant_forward()* function that part of the plant has been detected. Finally the function records the transition between white background and the coloured plant (see the next function for explanation).



Determine_plant()

This is the penultimate function called before the actual type of plant is determined, its block diagram is shown underneath. The sole purpose of this function is to record all the pixels of the plant in order to build a profile (array of integers) for it. In order to determine whether the end of the plant has been reached, the gradient between the beginning of the plant recorded in the function above and the current gradient is compared. **N.B** This method is purely theoretical and relies on the symmetry of the plant, there will be other plans if this method is proven inadequate.

After the entire plant profile is recorded, the final regression algorithm is run (not formalised yet) and the type of plant is written into a memory address to be accessed by *plant_forward()* function.



Bill of Materials

Function name	Estimated/Actual Lines of code	Current status
Get_sensor_input()	10	Code complete
Get_linesensor_state()	20	Code complete
Set_motor_speed(arg1, arg2)	10	Code complete
Move_forward()	40	Code complete
Clockwise_turn(arg1)	30	Diagram complete
AntiClockwise_turn(arg1)	30	Diagram complete
Turn_45_right(arg1)	30	Incomplete
Turn_45_left(arg1)	30	Incomplete
Plant_forward()	30	Code complete
Object_detect()	30	Diagram complete
Determine_plant()	40	Diagram complete
Plant_analysis()	60	Incomplete
Pick_up_plant()	50	Incomplete
Eject_plant()	100	Incomplete
U_turn(arg1)	30	Incomplete
Initialise()	40	Code complete
*Extra lines for debug and console output	200	Incomplete

TOTAL: 780

Appendix: plant_forward()

```
int Px = 0, Px0 = 0, Dx = 0, Dx0 = 0; dPx = 0, dDx = 0; //Initialise LDR and distance sensor readings
int th1 = 5, th2 = 5; anomaly_threshold = 5; //Specify thresholds
int N = 0; //Initialise counters
int anomaly_counter = 0; //Initialise counter for number of odd pixels
int mean[2] = {0,0}; //Initialise mean LDR and distance readings
enum Plant_state { "cabbage", "cauliflower", "nada" };
Plant_state plant = "nada"; //Initialise the state of plant detected
bool Object_status = false;
int plant_array[];

/*
This is the main function which from the LDR and Distance sensor inputs, determines the action of the robot,
namely following the lines and keep recording data, or initiate pick-up sequence
*/
plant_forward(I){
        // I is the line-sensor state which will trigger the termination of the plant_fotward function
        while (Sensor_state()!=I)
                // Update pixel and distance readings from sensors
                Px = get_sensor(READ_PORT1);
                Dx = get_sensor(READ_PORT2);
                // Handling initial values
                if (N=0){
                        mean[0] = Px0 = Px;
                        mean[1] = Dx0 = Dx;
                }
                //If no object detected then drive forward
                if (Object_Status == false){
                        Object_detect();
                        drive_forward(arg1, arg2);
                }
                //If a plant is detected then drive at some slower speed
                else if (plant=="nada"){
                        Determine_plant();
                        drive_forward(arg3, arg4);
                }
                //Pick up the plant once a the type of plant has been determined
                else {
                        Pick_up_plant();
                }
                Px0 = Px;
                Dx0 = Dx;
        }
}
```