15-06-2021

ELEC50003 – ELEC50008: Computer Engineering Design Project 2 2020-2021

Mr. Adam Bouchaala *Mrs. Esther Perea*

Mars Rover Project

Tanguy Perron
01722386 || tlp19@ic.ac.uk

Riccardo El Hassanin
01729427 || re119@ic.ac.uk

Daniel Romano
01728878 || djr19@ic.ac.uk

Jordan Chin
01737233 || jyc119@ic.ac.uk

Bertil de Germay de Cirfontaine
01767839 || bd519@ic.ac.uk

Jason Tang
01720640 || jyt19@ic.ac.uk

Word count: 12790 Page count: 48

Table of Contents

Introduction

Our second year marked the separation of the cohort into two sub-streams, EIE and EEE, having different modules, teachers, and studying different contents. Combining the two groups again, we have accumulated a broad and general knowledge in both electrical, electronic, and information engineering. This allows us, working again as a team, to enter the Mars Rover project with trust and enthusiasm by combining our efforts towards a common goal. Moreover, the thought of combining both software and hardware insights to get a better understanding of the work accomplished during our laboratories so far (using the same tools or equivalent ones: FPGA, SMPS, Arduino) as well as getting insight on the other stream's laboratories seemed exciting and very much interesting.

Thus, this project seemed like a perfect opportunity to gain some working experiences by managing a network of six, including students specified in different domains, working in different countries and in different time zone. Another key aspect was the ability to apply the knowledge learned so far this year while working on a hands-on project. The latter putting forward team reflection and decision making, setting a radical turn in our studies compared to previous projects with strong guidelines. Indeed, having had specifications that were not too strict, allowed and encouraged us to take initiatives in design choices, and made us gain autonomy and confidence while working on a project that was dear to all of us as discussing Space exploration. While we have to admit that designing a project during the Covid-19 pandemic represents a challenge, our envy to succeed and our interest in the subject kept us focused until the end, by managing and sticking to a strict timetable. Now more than ever, communication along with workload repartition skills were determinant in the accomplishment of our Mars Rover.

The project aims to design and implement an autonomous Rover being effective in a remote location where the user cannot have direct access to it. This Mars Rover can be constructed by dividing the workload into six submodules that can interact with each other. The Rover receives movement commands from the user to be executed, while sending back status data regarding its internal state and environment. It also needs to conduct a constant scan of the surroundings to avoid contact with objects by initialising an obstacle avoidance protocol.

Using all those requirements, we will be able to store all the data sent back from the system's submodules and make it accessible to the user while using it to plot the Rover's movements on a map alongside the scan of the surroundings. All in all, the system is divided into six parts being, Command, Control, Drive, Vision, Energy, and Integration, which need to be implemented as one unity block to allow the data to flow between the submodules. This necessity also gave us an incentive to stay in close contact with each other during the whole project's time.

Command

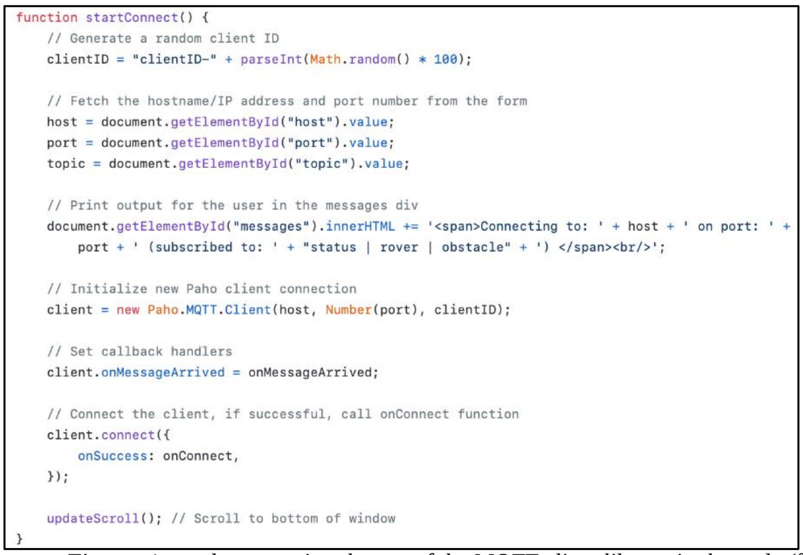
The Command submodule represents the front-end of our Mars Rover project. It establishes the connection between the user and all the functionality of the rover by setting connections and exchanging information with the other submodules. The interface created needs to let the client send instructions to the Drive submodule, but also be able to receive, analyse and present the data sent back from the back-end of our system. In order to set up a wireless connection compatible with long distance information exchanges, compatible with the theory of sending our rover on Mars, we used the MQTT protocol.

The associated client library works around subscribing and publishing messages. A client subscribed to a topic will receive all the information exchanged on said topic. While in the same way, the publish function allows clients to send data on selected topics over the channel. Using this library, we can therefore initialise the channels with the other subsystem whilst avoiding loss of data due to conflictual reception. Thus, having three submodules to exchange data with, one of them receiving data as well as sending feedback on execution of the instructions, we initialised four channels connected to the broker by using its IP address and Port following our MQTT client library that go through the control subsystem performing computation and analyses.

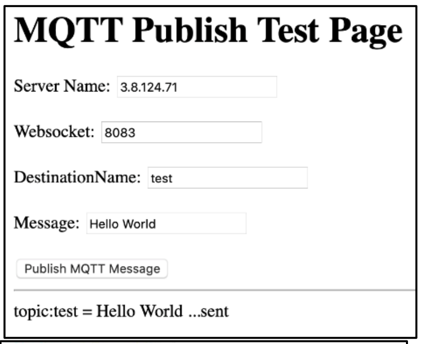
TOPIC	**PUBLISH or SUBSCRIBE**	**SUBMODULES connected**
command	Pub	Command-Drive
rover	Sub	Command-Drive
obstacle	Sub	Command-Vision
status	Sub	Command-Energy

Figure 1-1: table presenting the topics under which the data flows

Leaning on the theory of Information Processing and the hardware available, we set up this method using the Amazon Web Services (AWS). By creating an online server, we assure a connection to the rover over the Internet accessible to the client, later on accessible everywhere online.



```
function startConnect() {
    // Generate a random client ID
    clientID = "clientID-" + parseInt(Math.random() * 100);

    // Fetch the hostname/IP address and port number from the form
    host = document.getElementById("host").value;
    port = document.getElementById("port").value;
    topic = document.getElementById("topic").value;

    // Print output for the user in the messages div
    document.getElementById("messages").innerHTML += '<span>Connecting to: ' + host + ' on port: ' +
        port + ' (subscribed to: ' + "status | rover | obstacle" + ') </span><br/>';

    // Initialize new Paho client connection
    client = new Paho.MQTT.Client(host, Number(port), clientID);

    // Set callback handlers
    client.onMessageArrived = onMessageArrived;

    // Connect the client, if successful, call onConnect function
    client.connect({
        onSuccess: onConnect,
    });

    updateScroll(); // Scroll to bottom of window
}
```



MQTT Publish Test Page

Server Name: 3.8.124.71

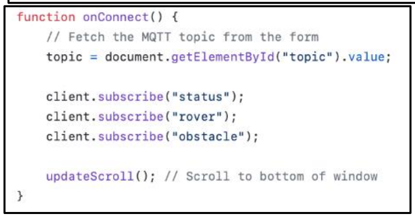
Websocket: 8083

DestinationName: test

Message: Hello World

Publish MQTT Message

topic:test = Hello World ...sent



```
function onConnect() {
    // Fetch the MQTT topic from the form
    topic = document.getElementById("topic").value;

    client.subscribe("status");
    client.subscribe("rover");
    client.subscribe("obstacle");

    updateScroll(); // Scroll to bottom of window
}
```

Figures 1-: code presenting the use of the MQTT client library in the code (figures 2-4) as well as a test page containing code to test the connections (figure 3)

1) Live routing

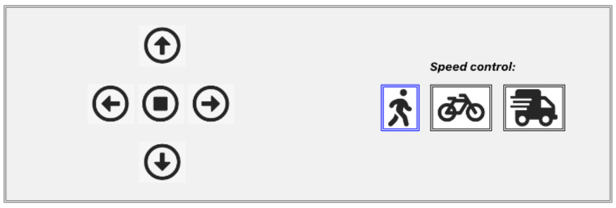
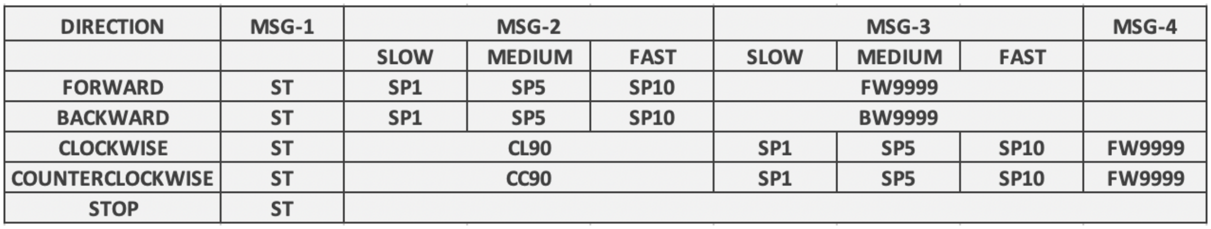


Figure 1-5: command panel of the Live routing mode

When thinking about the implementation of a command set for a robot, we directly think about live instructions that are executed in real-time by the Mars Rover. Thus, we implemented a first section dedicated to live routing. We offered the client to interact with a set of eight buttons to regulate two components, the direction and the speed. The user has the choice of making the rover go forward as well as backward, make it turn clockwise/counter-clockwise and stop. Along with the direction selection, three buttons are available on the side to let the user select among slow (10%), medium (50%), and fast (100%) speed, representing the scale of speed the rover can travel at compared to its maximum velocity.

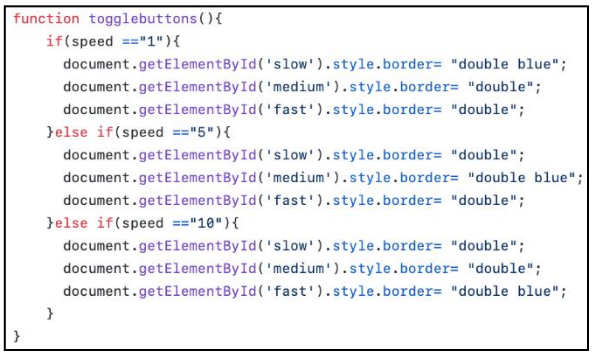
In order to implement those instructions in java-script, we created buttons for the direction ones which upon selection are executing the `send_mqtt_msg2()` function, publishing the command along with the last speed selected. To make it work and to properly generate a live control of the rover's movement, the rover executes at a given speed the instruction selected until the next user's input. On selection, a message "ST" is sent which stops the execution of the previous instruction and starts the new one. Finally, a FW9999 (or BW9999) is given, standing for going forward (or respectively backwards) for 9999mm, big enough to act like an infinite value, but not big enough to slow down the processing of the rover.



DIRECTION	MSG-1	MSG-2			MSG-3			MSG-4
		SLOW	MEDIUM	FAST	SLOW	MEDIUM	FAST	
FORWARD	ST	SP1	SP5	SP10	FW9999			
BACKWARD	ST	SP1	SP5	SP10	BW9999			
CLOCKWISE	ST	CL90			SP1	SP5	SP10	FW9999
COUNTERCLOCKWISE	ST	CC90			SP1	SP5	SP10	FW9999
STOP	ST							

Figure 1-6: table showing the messages sent when pressing one of the buttons

On a design level, this section of the website tries to convey a live command set to the user by involving dynamic buttons that react upon interaction with the user, along with a very intuitive designing approach. In that regard, we used the website, fontawesome.com, which provides a library of buttons that can be modified using CSS to be more expressive, alongside the function `togglebuttons()` that highlights the selected speed.

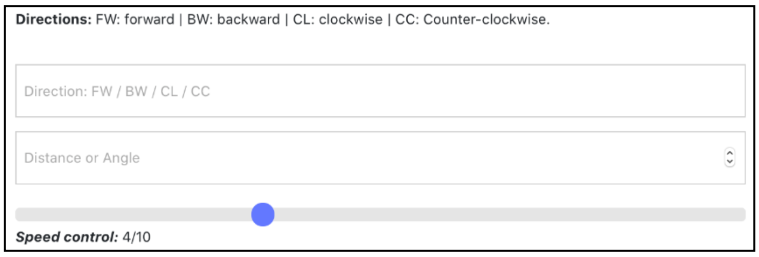


```
function togglebuttons(){
    if(speed =="1"){
      document.getElementById('slow').style.border= "double blue";
      document.getElementById('medium').style.border= "double";
      document.getElementById('fast').style.border= "double";
    }else if(speed =="5"){
      document.getElementById('slow').style.border= "double";
      document.getElementById('medium').style.border= "double blue";
      document.getElementById('fast').style.border= "double";
    }else if(speed =="10"){
      document.getElementById('slow').style.border= "double";
      document.getElementById('medium').style.border= "double";
      document.getElementById('fast').style.border= "double blue";
    }
}
```

Figure 1-7: code presenting the formatting function for buttons

Finally, as we expected it to be, the limits of this solution came to light quite soon after the testing phase. As the information travelling along the channel takes time to attain its destination, precision is not able to be achieved because of the communication delays. This impact is seen for all instructions, but especially turning instructions as a difference of a small angle translates to a growing error as the rover then moves forward or backwards. Therefore, we decided to tackle the issue in a way that would reduce at a minimum the discrepancy of precision by setting constant the setting the angles at a constant value of 90 degrees (cf. table shown above). We kept this option in the final design because it made sense to us to include a live routing as a solution, keeping however in mind that it was far from optimal and went on with our second solution.

2) Fixed routing


Directions: FW: forward | BW: backward | CL: clockwise | CC: Counter-clockwise.

Direction: FW / BW / CL / CC

Distance or Angle

Speed control: 4/10

Figure 1-8: command panel of the Fixed routing mode

A more realistic approach is to base our instructions on precise directions, angles, and distances, telling the Mars Rover to combine those characteristics in order to pinpoint the wanted destination. This appears as a good solution to guaranty both precision and anticipation, a command queue was set up to store the instructions in order. Just as for the Live Command-mode, Fixed routing is based on message publications on the "command" topic through the broker. The client will have control over the direction and the speed by inputting data in the required fields and sliding a throttle.

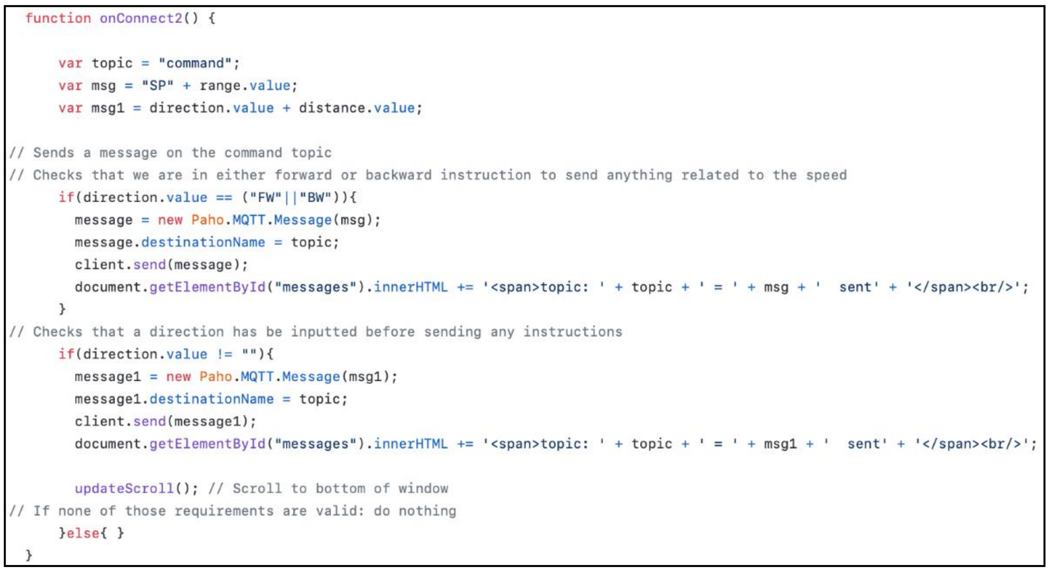
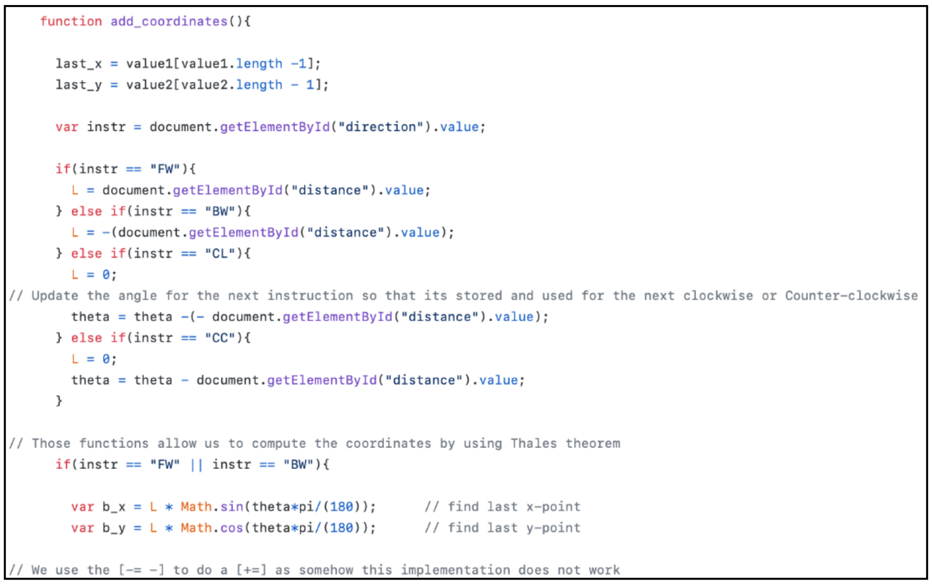

```
  function onConnect2() {

      var topic = "command";
      var msg = "SP" + range.value;
      var msg1 = direction.value + distance.value;

// Sends a message on the command topic
// Checks that we are in either forward or backward instruction to send anything related to the speed
      if(direction.value == ("FW"||"BW")){
        message = new Paho.MQTT.Message(msg);
        message.destinationName = topic;
        client.send(message);
        document.getElementById("messages").innerHTML += '<span>topic: ' + topic + ' = ' + msg + '  sent' + '</span><br/>';
      }
// Checks that a direction has be inputted before sending any instructions
      if(direction.value != ""){
        message1 = new Paho.MQTT.Message(msg1);
        message1.destinationName = topic;
        client.send(message1);
        document.getElementById("messages").innerHTML += '<span>topic: ' + topic + ' = ' + msg1 + '  sent' + '</span><br/>';

        updateScroll(); // Scroll to bottom of window
// If none of those requirements are valid: do nothing
      }else{ }
  }
```

Figure 1-9: code describing the use of a MQTT client library to set up the connection

The Fixed routing protocol is a bit less intuitive but gets closer to the real conditions a Mars Rover system should fulfil. The communication protocol, based on the client's precise input, is unaffected by delay nor by the user's reaction time. The client must enter two characteristics, a direction among the four presented in the previous division along with a distance or an angle depending on the instruction name typed in. Moreover, he also has the choice of making the speed fluctuate with the use of a slider going from one to ten, ten being the fastest, following unit steps, and being initialised to one. Once the details of the instruction are all selected, a click on the button “send Command to Rover” publishes the instruction to processed by Control and then transferred to the drive subsystem.

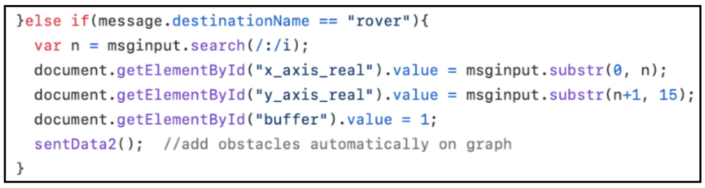
However, it is not the only functionality of this command-mode, a graph is present onto the website on which the ideal path of the rover is plotted upon transmission of an instruction. To do so, said instruction of the form, [direction + distance/angle], must be converted to coordinates before being plotted on the graph. Thus, we created the function, `add_coordinates()`, that uses the distance given during Forward and Backwards instructions alongside the angle of the previous instruction to compute cartesian coordinates. Once calculated, the array is expanding to store the Ideal Rover’s path with the result and update the graph.



```
    function add_coordinates(){

      last_x = value1[value1.length -1];
      last_y = value2[value2.length - 1];

      var instr = document.getElementById("direction").value;

      if(instr == "FW"){
        L = document.getElementById("distance").value;
      } else if(instr == "BW"){
        L = -(document.getElementById("distance").value);
      } else if(instr == "CL"){
        L = 0;
// Update the angle for the next instruction so that its stored and used for the next clockwise or Counter-clockwise
        theta = theta -(- document.getElementById("distance").value);
      } else if(instr == "CC"){
        L = 0;
        theta = theta - document.getElementById("distance").value;
      }

// Those functions allow us to compute the coordinates by using Thales theorem
      if(instr == "FW" || instr == "BW"){

        var b_x = L * Math.sin(theta*pi/(180));      // find last x-point
        var b_y = L * Math.cos(theta*pi/(180));      // find last y-point

// We use the [-= -] to do a [+=] as somehow this implementation does not work
```

Figure 1-10: code showing the implementation the coordinate conversion

Plotting such a path helps with the visualisation of the instruction inputted, especially when comparing with the trace of the actual Rover path that it is being sent back to us at the end of every instruction through the topic “rover”. When receiving such a message, it has to be parsed using the search function that looks at a separator’s position and divides the string in two at its location. Having the traces side by side on the graph allows us the analyse the discrepancies that occur during execution time.



```
}else if(message.destinationName == "rover"){
  var n = msginput.search(/:/i);
  document.getElementById("x_axis_real").value = msginput.substr(0, n);
  document.getElementById("y_axis_real").value = msginput.substr(n+1, 15);
  document.getElementById("buffer").value = 1;
  sentData2();  //add obstacles automatically on graph
}
```

Figure 1-11: code showing the reception/parsing of messages

Finally, even though the design is not as intuitive as the Live one was, this option allows us to have a precise control over the direction of the rover, being able to select the exact angle and the distance to travel.

3) Coordinate routing and graph implementation

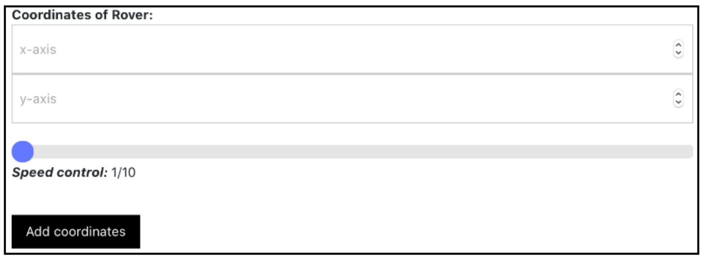


Figure 1-12: command panel of the Coordinates routing

Trying to merge the two previous solutions found, we proposed one last Command-Mode based on the exchanges of coordinates between the web server and the Rover. It has the benefit of being both accurate and intuitive as it only requires choosing a point on a map without dealing with angles nor distances. This method allows great precision as it makes possible clear pathing and gives the possibility to the return to the starting position of the rover using the instruction [CO0:0]. Nonetheless, once a coordinate instruction is sent to the Rover through control, it is also added to the graph and following the same pattern as for Fixed Routing, can be compared to the actual path of the Rover.

We implemented another functionality referred to as the switching of mode. This creates a link between Fixed and Coordinates routing by storing the angle but also the last position on the graph after a coordinate instruction is sent. To achieve this, a general function common to multiple submodules was implemented. It is called in the web app code `computeAngle()`. The angle, calculated using the current and last coordinates, is added to the sum of angles (I.e., orientation of the rover with respect to the y-axis). This helps us keep track of the direction to take in Fixed routing.

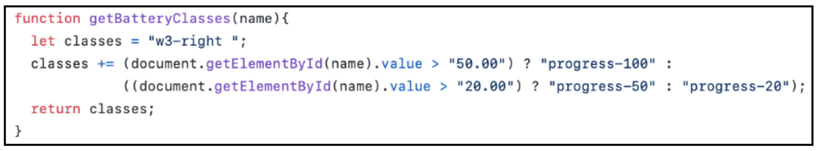
In addition to receiving information from the drive submodule, another kind of data is required for the graph. Whenever an instruction is sent to the rover, it will be executed while scanning the area for obstacles. If none are spotted it executes the command normally, on the other hand, if it recognises one, the Mars Rover will overwrite the current instruction with an obstacle avoidance protocol that will appear on the graph alongside the obstacle spotted. In this case, the control module will publish a message on the topic, “obstacle” with the first letter being the obstacle’s colour and the rest its coordinates. In such circumstances, the comparison of the Ideal and Actual path of the Rover on the graph is expressive as we can clearly see the deviation taken upon the encounter of an obstacle.

Thus, the implementation of coordinates seems very important as useful in order to provide a solution that could easily be used on an actual Mars Rover. In fact, every surface can be turned into a map and thus allow the Rover to reach easily any points on said map with simplicity to the user and efficiency.

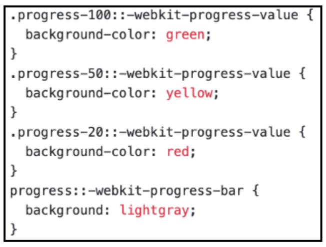
4) Battery communication

While the Battery submodule had an issue and we were required to stop working rather abruptly on the components, we decided to implement a connection between Energy and Command and simulate the Battery features. In a similar fashion of printing the data on the web page to make it both clear and understandable, we chose to rely on progress bars. The Energy submodule transmits two values in every message on the topic, “status”, the Battery level, and the state of health of the Battery. The latter one can only be updated at the end of every Battery level cycle (when the Battery level is equal to zero).

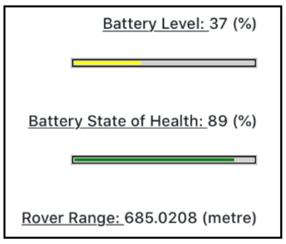
Thus, we made a program that decrements every second by ten percent the Battery level and when arriving at zero decrements as well by ten percent the Battery state. From those two inputs and the current speed of the Rover, the theoretical range that the Mars Rover could go with the current Battery level and state of health is computed through the function `roverRangeCalc()`. Moreover, using the CSS resources available to us, we modified the basic design to make it switch colour when changing ranges of values, [0-20] in red, [21-50] in yellow, and [51-100] in green.



```
function getBatteryClasses(name){
  let classes = "w3-right ";
  classes += (document.getElementById(name).value > "50.00") ? "progress-100" :
             ((document.getElementById(name).value > "20.00") ? "progress-50" : "progress-20");
  return classes;
}
```



```
.progress-100::-webkit-progress-value {
  background-color: green;
}
.progress-50::-webkit-progress-value {
  background-color: yellow;
}
.progress-20::-webkit-progress-value {
  background-color: red;
}
progress::-webkit-progress-bar {
  background: lightgray;
}
```



Battery Level: 37 (%)

Battery State of Health: 89 (%)

Rover Range: 685.0208 (metre)

Figures 1-: code for the progress-bar colour changing (figures 13-14) and the results outputted on the website (figure 15)

All in all, the implementation of the Command submodule is fundamentally based on the MQTT broker method that allow the exchanges of information around our project sub-divisions. By interacting with all the other sub-modules, the user gets an extensive vision of the Rover's environment and can thus modify its commands in order to optimise its control. The web server is designed in such a way to make it the simplest and as intuitive as possible so that the information is delivered clearly to the user. Finally, to make it neater and most importantly more accurate to an actual web server, we chose a personalised domain name though the website duckdns.org and added a personalised tab icon on the left edge:

mars-rover-returnsuccess3.duckdns.org:8000

Figure 1-16: photo of the website domain name and icon

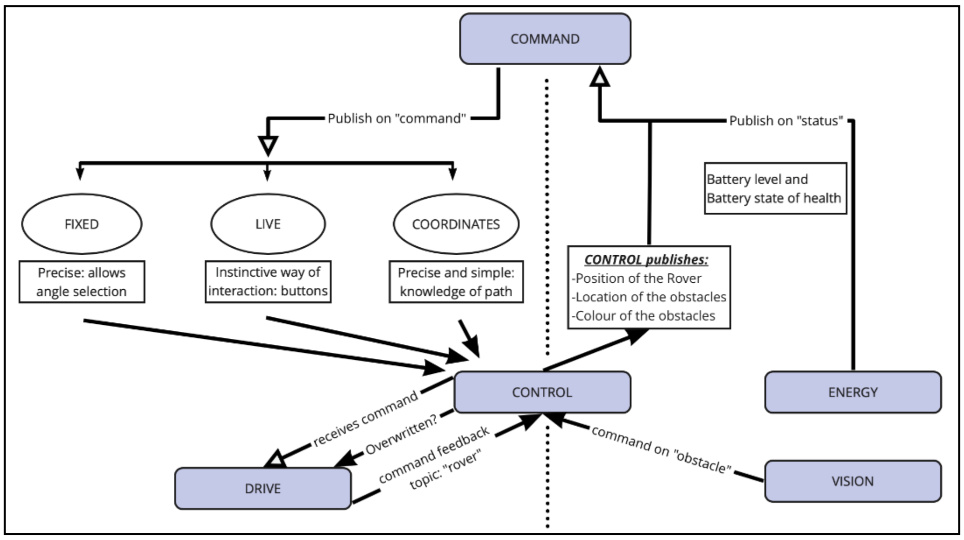

COMMAND
Publish on "command"
Publish on "status"
Battery level and Battery state of health
FIXED
LIVE
COORDINATES
Precise: allows angle selection
Instinctive way of interaction: buttons
Precise and simple: knowledge of path
CONTROL publishes:
-Position of the Rover
-Location of the obstacles
-Colour of the obstacles
CONTROL
ENERGY
receives command
Overwritten?
command feedback topic: "rover"
command on "obstacle"
DRIVE
VISION

Figure 1-17: functional diagram of the Command subsystem

Control

1) Introduction of the subsystem and hardware

The Control subsystem links the Command, Vision, Drive and theoretically Energy subsystems together. It handles the communications between them and defines the logic that drives them all together as one complete Rover. This effectively makes the Control subsystem the hub of all the other subsystems, directing what information flows, where, in what direction, and with what priority, adapting the format and quality of the data to each specific limitation of the counter-part subsystems.

The requirements of the Control subsystem are to communicate wirelessly with the Command subsystem to receive instruction to drive the Rover. It then needs to interpret and transmit the received instructions to the Drive subsystem that will execute them to move the Rover according to the input of the user. The Drive subsystem will then return the coordinates of the Rover to send back to Command in order to keep track of where the Rover has been. The Control subsystem also needs to listen for input from the Vision subsystem that will alert of obstacles that the Rover could encounter. It then needs to transform that alert into instructions to send to the Drive subsystem so that the Rover is clear of all threat of collision. It should also send information to the Command system about those threats so that the user is informed and shown where the obstacles are. Information from the Energy subsystem should also be transmitted to the Command subsystem so that the user has access to the battery status of the Rover.

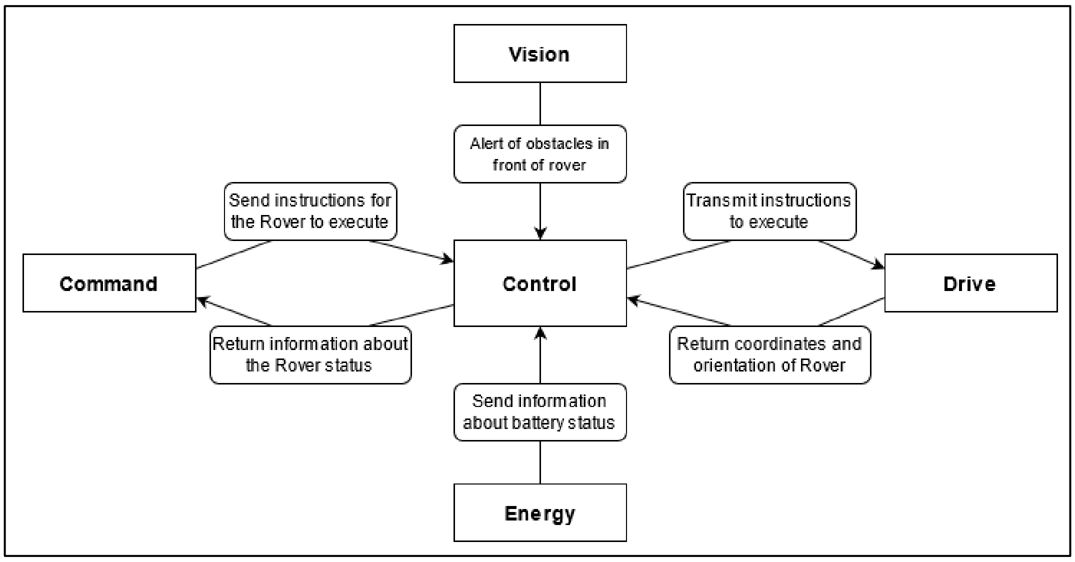

Vision
Alert of obstacles in front of rover
Send instructions for the Rover to execute
Transmit instructions to execute
Command
Control
Drive
Return information about the Rover status
Return coordinates and orientation of Rover
Send information about battery status
Energy

Figure 2-1: Graph of the dataflow between Control and the other subsystems

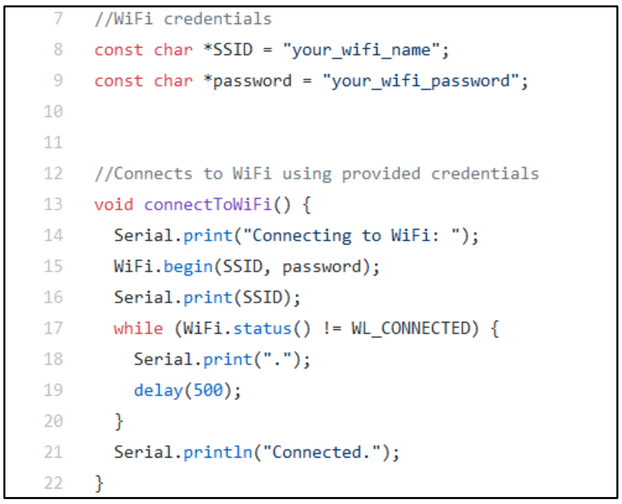
The hardware used for supporting this data flow and computation is an ESP32 microcontroller. The ESP32 is a low-cost System-on-a-Chip highly used in IoT devices development because of its power-efficient processor, Wi-Fi, and Bluetooth capabilities, as well as many ports for serial communications, including 3 UART and 4 SPI ports. The exact model used is the ESP-WROOM-32 module mounted on an ESP32-DevkitC-V4 break-out board.

2) Command to Control

The Wi-Fi capabilities of the ESP32 is ultimately what made it a standout choice over any other Arduino board. This allows the ESP32 to connect to a Wi-Fi network and send data wirelessly to a remote server. This is implemented in the communications between the Command and Control subsystems.

To talk to the remote Command web app, we use a network protocol called Message Queuing Telemetry Transport (MQTT). It is designed for IoT devices and its main advantage is that it is extremely lightweight, allowing for a small code footprint, and low network bandwidth requirements. It is a publish-subscribe messaging transport, which means that clients can publish to certain topics and that clients that subscribe to those topics will all receive the published messages. It is ideal for connecting multiple devices together but is still extremely useful for one-to-one communication as it is fast and reliable. As the Rover is supposed to be in a remote destination far away, using a small network bandwidth is a priority as it might be difficult to allow for bandwidth-intensive communications to happen.

To connect to a Wi-Fi network, the ESP32 board uses the standard WiFi.h Arduino library and functions. We first specify the WiFi SSID that we want to connect to, as well as its password, and then connect to it using the `connectToWiFi()` function as follows:



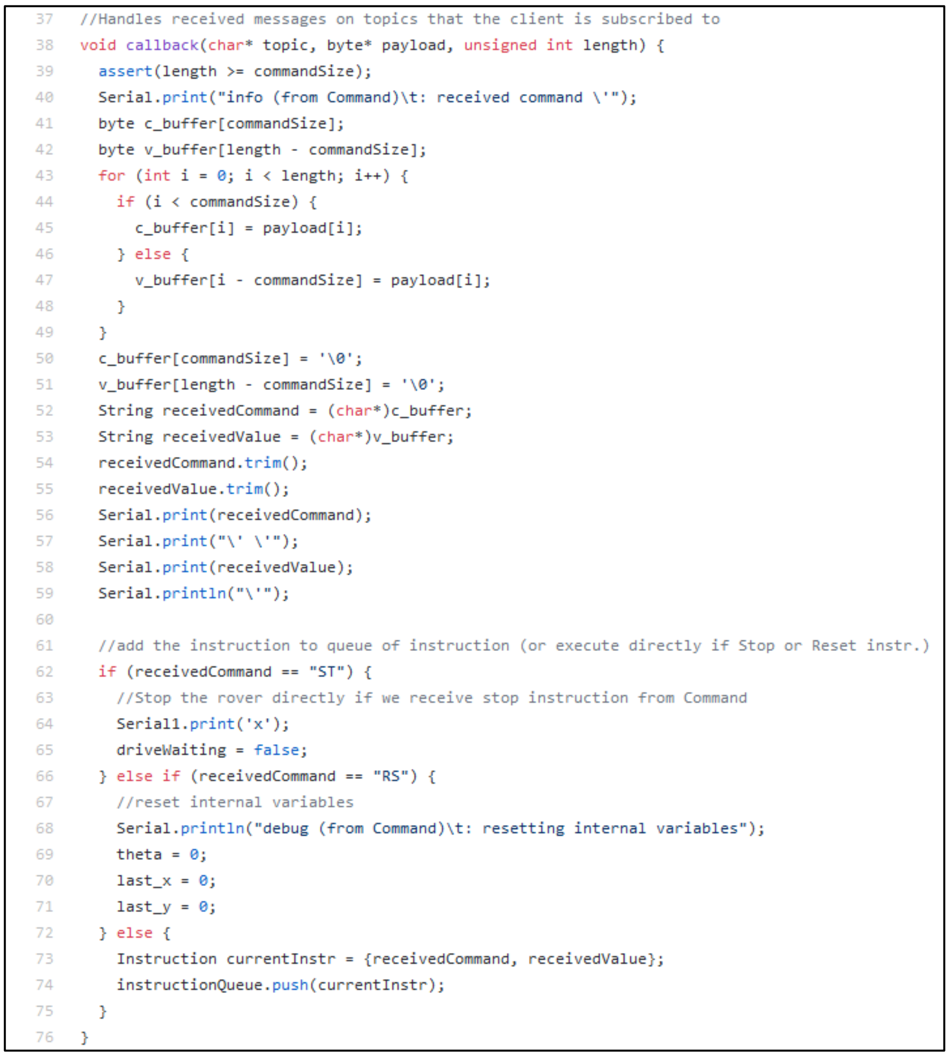
```
//WiFi credentials
const char *SSID = "your_wifi_name";
const char *password = "your_wifi_password";


//Connects to WiFi using provided credentials
void connectToWiFi() {
  Serial.print("Connecting to WiFi: ");
  WiFi.begin(SSID, password);
  Serial.print(SSID);
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(500);
  }
  Serial.println("Connected.");
}
```

Figure 2-1: function used to connect to WiFi

The ESP32 then uses an MQTT Client library called PubSubClient created by Nick O'Leary. This library allows for the setup of the MQTT connection to a remote private Broker hosted on Amazon Web Services (AWS) using its IP address and the port that the Broker listens on. The MQTT Broker is the hub of all MQTT communications. It handles subscriptions from clients and redirects incoming publish requests to all clients subscribed to the specified topic of the message.

A custom AWS EC2 instance hosts the MQTT Broker using the Eclipse Mosquitto program and the broker is set up to listen for incoming connections on ports 1883 (for Control's ESP32 publications) and port 8083 (for the Command's web-app publications using the web sockets protocol).

On the ESP32, the PubSubClient library also provides tools to subscribe to topics, publish messages, and receive messages whose topics the client is subscribed to. To connect to the Broker, we first need to instantiate a PubSubClient object based on a WiFiClient. Then we can set the IP Address and Port of the Broker that the client needs to connect to using the `PubSubClient::setServer()` function. We can then connect to the Broker using the `PubSubClient::connect()` function and supplying a unique client ID to identify the client. Once the client is connected, it can subscribe to the topics that it needs to receive messages from using the `PubSubClient::subscribe()`, in our case, this is the topic "command". The last step to allow our client to receive MQTT messages from the Command subsystem is to define and set up a callback function. This function gets called every time the MQTT Client receives a message transferred by the Broker. In our implementation, this is where all the processing of Command's messages happens:



```
//Handles received messages on topics that the client is subscribed to
void callback(char* topic, byte* payload, unsigned int length) {
  assert(length >= commandSize);
  Serial.print("info (from Command)\t: received command \'");
  byte c_buffer[commandSize];
  byte v_buffer[length - commandSize];
  for (int i = 0; i < length; i++) {
    if (i < commandSize) {
      c_buffer[i] = payload[i];
    } else {
      v_buffer[i - commandSize] = payload[i];
    }
  }
  c_buffer[commandSize] = '\0';
  v_buffer[length - commandSize] = '\0';
  String receivedCommand = (char*)c_buffer;
  String receivedValue = (char*)v_buffer;
  receivedCommand.trim();
  receivedValue.trim();
  Serial.print(receivedCommand);
  Serial.print("\' \'");
  Serial.print(receivedValue);
  Serial.println("\'");

  //add the instruction to queue of instruction (or execute directly if Stop or Reset instr.)
  if (receivedCommand == "ST") {
    //Stop the rover directly if we receive stop instruction from Command
    Serial1.print('x');
    driveWaiting = false;
  } else if (receivedCommand == "RS") {
    //reset internal variables
    Serial.println("debug (from Command)\t: resetting internal variables");
    theta = 0;
    last_x = 0;
    last_y = 0;
  } else {
    Instruction currentInstr = {receivedCommand, receivedValue};
    instructionQueue.push(currentInstr);
  }
}
```

Figure 2-2: implementation of the MQTT callback function

The function receives and decodes the incoming message byte per byte, i.e. character per character. It stores the first two characters in a character array called `c_buffer`, this is our instruction code. It then stores the rest of the incoming message in a character array called `v_buffer`, this is the value/argument of the instruction. Both character arrays are then appended with a null character and transferred to Arduino strings to be decoded. All supported instructions by the Control subsystem and the decoding performed by the callback function are as follows:

Command instr. code	**Drive instr. code**	**Argument**	**Meaning**	**Decoding of the callback function**
'FW'	'f'	Integer in millimetres	Make the Rover go forward by the value of the argument.	Add to the instruction queue

’BW’	‘b’	Integer in millimetres	Go backwards by the argument	Add to the instruction queue
‘CL’	‘r’	Integer in degrees	Turn clockwise by the value of the argument	Add to the instruction queue
‘CC’	‘l’	Integer in degrees	Turn counter-clockwise by the argument	Add to the instruction queue
‘CO’	Translated as: ‘r’ or ‘l’ Followed by: ‘f’	String in the format “X:Y” where X and Y are integers in millimetres	Make the Rover go to the coordinates in the argument	Add to the instruction queue
‘ST’	‘x’	n/a	Stop immediately the Rover	Tell the Rover to stop, jumping the instruction queue
‘RS’	n/a	n/a	Reset the internal values for the coordinates and orientation of the Rover	Reset the respective variables, jumping the instruction queue

Figure 2-4: table of all instructions supported by the Control subsystem and their decoding done by the callback function

As mentioned in the table, the ESP32 keeps a queue of all the instructions received over MQTT, that way the user does not need to worry about timing and can send all the instructions needed in a row, knowing that they will all get executed correctly in order. The instruction queue is implemented as a C++ queue using the standard `<queue>` library. Each element of the queue is of a custom-made structure called `Instruction` that stores both the instruction code (also called the `command` in the code) and the argument (also denoted as the `value` in code) as Arduino strings.

3) Vision to Control

The communication between the ESP32 of Control and the FPGA of the Vision subsystem is implemented using the Universal Asynchronous Receiver-Transmitter protocol (UART). As mentioned before, the ESP32 possesses three UART ports, one of which is used in the break-out board for the micro-USB connector. That leaves 2 UART ports available to be used by the board for communications with other subsystems. One of those is used for the Vision to Control connection.

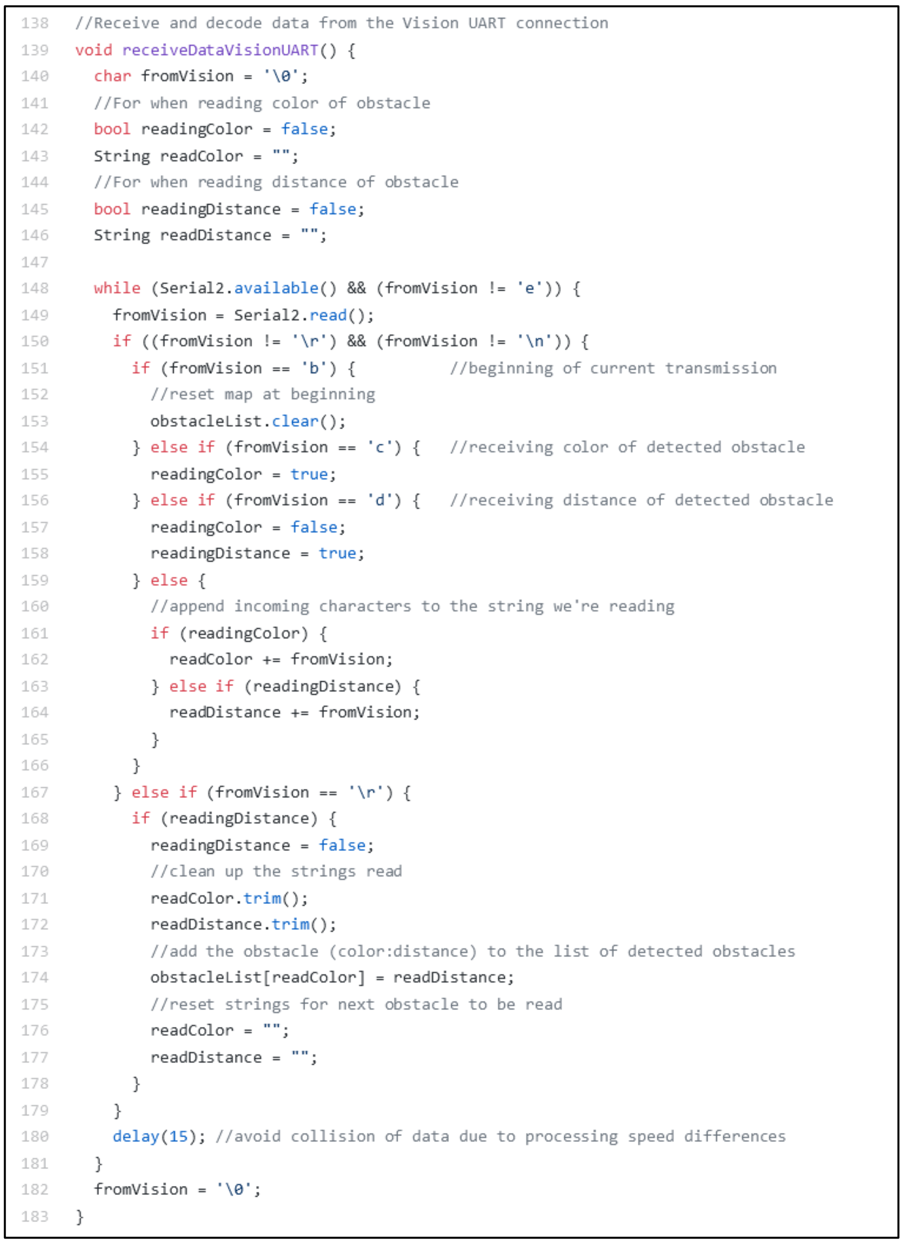
As a UART port can be instantiated with any GPIO pins on the break-out board, the ESP32 uses pin 22 as a receiver, and pin 23 as a transmitter. Those pins map respectively to pins 3 and 2 on the ESP32-to-FPGA adapter board, and therefore pins IO3 and IO2 on the FPGA itself. Ground pins are already connected by the adapter board and do not need to be manually connected. In code, the communication is instantiated in the Arduino setup function with `Serial2.begin()`, passing the baud rate (9600), the protocol used (SERIAL_8N1), and both the RX and TX pins as arguments.

Data only flows in one direction, from the FPGA to the ESP32, as the ESP32 does not need to send any information to the FPGA. On the other hand, the FPGA sends encoded information to the ESP32 about what the camera captures and processes. As UART is a protocol that transmits characters, the data is encoded into a character array as follows:

Beginning of transmission marker	**Obstacle data (optional)**					**Obst. Data…**	**End of transmission marker**
	Colour marker	**Colour of obstacle**	**Distance marker**	**Distance to obstacle**	**End of obstacle marker**	**…**	
‘b’	‘c’	‘O’/’P’/’G’/’B’/’A’	‘d’	‘F’/’C’	‘\r’	…	‘e’

Figure 2-5: format of Vision to Control UART communications (transmission happens from left to right)

The transmission happens from left to right, starting with a ‘b’ and finishing with an ‘e’. In between those two markers, zero or more “obstacle data” blocks can be included according to the number of obstacles detected by the camera. This sequence of characters is received, read, and decoded using the `receiveDataVisionUART()` function.



```cpp
//Receive and decode data from the Vision UART connection
void receiveDataVisionUART() {
  char fromVision = '\0';
  //For when reading color of obstacle
  bool readingColor = false;
  String readColor = "";
  //For when reading distance of obstacle
  bool readingDistance = false;
  String readDistance = "";

  while (Serial2.available() && (fromVision != 'e')) {
    fromVision = Serial2.read();
    if ((fromVision != '\r') && (fromVision != '\n')) {
      if (fromVision == 'b') {          //beginning of current transmission
        //reset map at beginning
        obstacleList.clear();
      } else if (fromVision == 'c') {   //receiving color of detected obstacle
        readingColor = true;
      } else if (fromVision == 'd') {   //receiving distance of detected obstacle
        readingColor = false;
        readingDistance = true;
      } else {
        //append incoming characters to the string we're reading
        if (readingColor) {
          readColor += fromVision;
        } else if (readingDistance) {
          readDistance += fromVision;
        }
      }
    } else if (fromVision == '\r') {
      if (readingDistance) {
        readingDistance = false;
        //clean up the strings read
        readColor.trim();
        readDistance.trim();
        //add the obstacle (color:distance) to the list of detected obstacles
        obstacleList[readColor] = readDistance;
        //reset strings for next obstacle to be read
        readColor = "";
        readDistance = "";
      }
    }
    delay(15); //avoid collision of data due to processing speed differences
  }
  fromVision = '\0';
}
```

Figure 2-6: receiveDataVisionUART() function

In a similar fashion to reading the MQTT messages from Command, this function reads incoming characters from the Vision-Control UART connection one at a time and depending on if the character is a marker, or following a specific marker, according to the table of the transmission format given above, it appends it to the right character array and then decodes it as an Arduino string. Both the colour and the distance contained in the obstacle data block are then added as entries to a map using the colour as the key. This map is reinitialized before each transmission and therefore keeps track of all obstacles detected and transmitted by Vision in real-time (transmission delays excluded). This list of obstacles is then analysed to check if any obstacles are detected as close to the Rover based on an arbitrary threshold set by the Vision subsystem (denoted by the character ‘C’ after the distance marker). If there are any, the flag `visionOverride` is set to true, signalling that the obstacle avoidance routine needs to be executed. The Rover gets immediately stopped with a "ST" instruction.

The obstacle avoidance routine is a pre-determined queue of instructions to be executed by the Rover to steer clear of the obstacle triggering the obstacle avoidance. It is only triggered if the Rover is going to collide with the detected obstacle when moving forward. If this condition is fulfilled, the `playingRoutine` flag is set to true, and the normal instruction queue gets halted while the alternate routine queue gets executed. After executing the routine, the normal instruction queue is resumed, starting with the instruction that was interrupted by the obstacle avoidance detection system. It is however not started back from the beginning, instead, we only continue the instruction for the distance that was left to be travelled, first by getting the distance already travelled from the Drive subsystem and subtracting it from the total distance of the instruction, as well as removing the distance that the Rover travelled autonomously when avoiding the obstacle.

The exact obstacle avoidance routine is as follows:

CL90	FW300	CC90	FW600	CC90	FW300	CL90

Figure 2-7: instructions of the obstacle avoidance routine

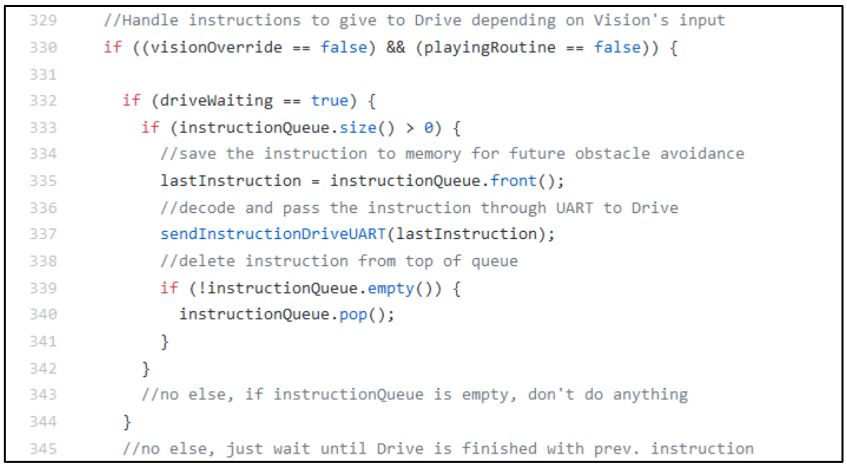
4) Control to Drive

The last UART port available to the ESP32 is used for communications with the Drive subsystem. This UART port uses pin GPIO16 as the receiver and pin GPIO17 as the transmitter. The equivalent adapter pins are respectively pins 9 and 8. The ground pin of the Arduino also needs to be connected to one of the ground pins on the ESP32 adapter board. This connection is instantiated as Serial1.

To send the instructions either contained in the instruction queue or in the avoidance routine queue to the Drive subsystem, they first need to be decoded and translated in a format that Drive can easily decode. For instructions such as "FW", "BW", "CL", "CC" and "ST" this means to simply change the instruction code to a single character to help Drive process it more efficiently, and then appending the argument (if any). However, for the "CO" instruction that has coordinates as an argument, it needs to be translated into two simpler instructions that the Drive subsystem can understand. This translation needs to be done in the Control subsystem for timing purposes that will be detailed later. Upon sending a coordinate instruction, the ESP32 uses the function `translateCoordinates()` that uses the current coordinates and orientation of the Rover and the target coordinates to compute the equivalent angle that the Rover would need to turn by and the distance that it would need to move forward by to reach the target coordinates. It then uses the function `insertFrontInstructionQueue()` to build the corresponding equivalent instruction and replace the initial "CO" instruction. After decoding and translating the instructions, the resulting character array is passed through the UART connection using `Serial1.print().`

To see the correspondence between Command and Drive instruction codes, refer to figure 4.

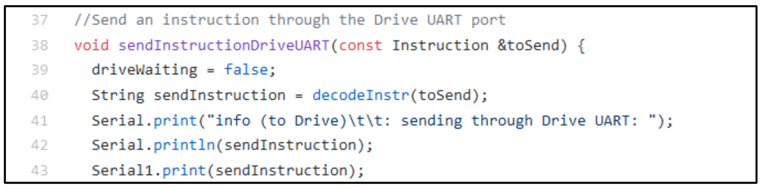
The ESP32 handles scheduling of instructions to ensure that all instructions get executed correctly. After sending an instruction through UART, the flag `driveWaiting` gets set to false, meaning that the Drive subsystem is busy moving the Rover. Once Drive is finished, the flag is set back to true, and the next instruction is sent through the UART. This ensures that no instructions clash with each other, as Drive only has one instruction at a time in memory. This flag is common for both the normal instruction queue and the avoidance routine queue, but can be ignored in the case of an instruction that requires immediate action, such as the stop instruction "ST". The scheduling of the instructions of the regular instruction queue is implemented in the main `loop()` as follows:



```
//Handle instructions to give to Drive depending on Vision's input
if ((visionOverride == false) && (playingRoutine == false)) {

  if (driveWaiting == true) {
    if (instructionQueue.size() > 0) {
      //save the instruction to memory for future obstacle avoidance
      lastInstruction = instructionQueue.front();
      //decode and pass the instruction through UART to Drive
      sendInstructionDriveUART(lastInstruction);
      //delete instruction from top of queue
      if (!instructionQueue.empty()) {
        instructionQueue.pop();
      }
    }
    //no else, if instructionQueue is empty, don't do anything
  }
  //no else, just wait until Drive is finished with prev. instruction
```

Figure 2-8: implementation of the scheduling of instruction based on the three general flags

Where the function `sendInstructionDriveUART()` is implemented as detailed below. The function `decodeInstr()` function implements the decoding and translation detailed earlier.



```
//Send an instruction through the Drive UART port
void sendInstructionDriveUART(const Instruction &toSend) {
  driveWaiting = false;
  String sendInstruction = decodeInstr(toSend);
  Serial.print("info (to Drive)\t\t: sending through Drive UART: ");
  Serial.println(sendInstruction);
  Serial1.print(sendInstruction);
```

Figure 2-9: sending instructions as strings through the UART connection to Drive

5) Drive to Control

To signal that the Rover is done executing an instruction, the Drive subsystem sends back the character ‘d’ through the UART Port to the ESP32. Upon receiving this marker, the `driveWaiting` flag is set to true, allowing the next instruction to be sent through the UART port to the Drive Arduino Nano Every. This ‘d’ marker is not the only data sent back by Drive to Control. Upon finishing a forward or backwards instruction, Drive also sends back the coordinates of the Rover, according to the X and Y-axis defined by the optical sensor when starting the Rover. Similarly, upon finishing a turn instruction (either clockwise or counter-clockwise), Drive sends back the orientation of the Rover with regards to the Y-axis (denoted as theta is the code). Moreover, when stopped using the “ST” immediate instruction, for example when obstacle avoidance is triggered, Drive returns the distance travelled since the last instruction. All those transmissions contain data formatted in a similar way to the transmissions made from Vision to Control, as follows:

Information transmitted	**Marker**	**Data**	**2nd Marker**	**Data**	**End of transmission marker**
End of instruction	‘d’				
Coordinates of Rover	‘x’	Integer in millimetres	‘y’	Integer in millimetres	‘\r’
Orientation of Rover	‘a’	Integer in degrees			‘\r’
Distance travelled	‘t’	Integer in millimetres			‘\r’

Figure 2-10: format of Drive to Control communication (transmission happens from left to right)

They are decoded using an equivalent function called `receiveDataDriveUART()` (see appendix A for detailed code implementation).

6) Energy to Control

As the two UART ports available for other subsystem connections are already used for the Vision-Control and Drive-Control communications, another protocol needs to be used for the Energy-Control communication. Serial Peripheral Interface (SPI) is a synchronous master-slave serial communication protocol that allows for communications at very high speeds. However, this specific advantage is not really at play here, SPI is simply chosen for this connection as it provides a low-cost interface between the two subsystems. However, this is only a theoretical connection.

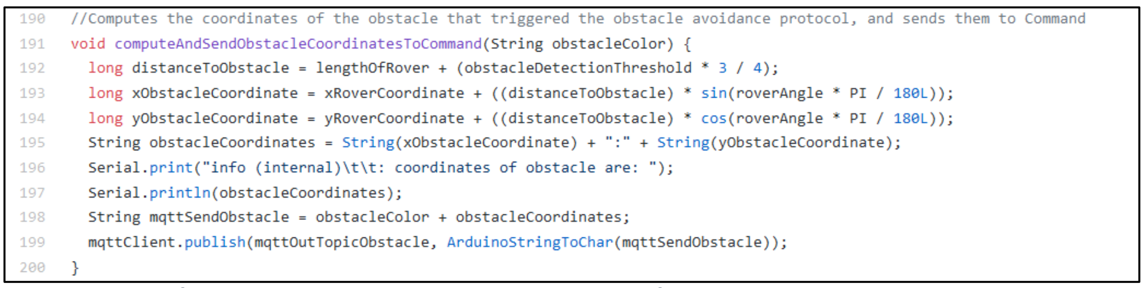
In practice, the Energy subsystem is never physically attached to the Rover. Therefore, a wired connection is not possible. To actually implement this connection, the Energy subsystem first sends the information through a UART serial connection between the Arduino and a computer. The computer then acts as a relay to collect the output of the Energy subsystem in a Python script that uses the `paho.mqtt.client` library to publish the information collected directly back to the Command web app on topic “status”.

7) Control to Command

The last objective of the Control subsystem is to send back information about the state of the Rover to the Command web app controller. For our current implementation, this excludes the Energy information as this is sent directly by the relevant subsystem to the web app. The remaining information to send is the one received from Drive and the information deduced from Vision’s input.

The Drive subsystem returns the measured coordinates of the Rover after each forward or backwards instruction. This therefore needs to be transmitted over to the Command controller to be plotted for the user to see and evaluate. The X and Y coordinates are respectively stored in `xRoverCoordinate` and `yRoverCoordinate` as long integers. However, a slight optimisation is to use the strings received from Drive in the `receiveDataDriveUART()` function directly as this saves a conversion from long to String. Both coordinates are first combined into an Arduino string in a format that can be decoded by the Command program, before being converted into a character array and sent to Command using the MQTT `PubSubClient::publish()` function to publish the coordinates of the Rover to the topic “rover”.

The `PubSubClient::publish()` function is also used to transmit the approximate coordinates of detected obstacles. However, as the Vision communication does not include the measured distance to the obstacle, an assumption is made about the position of the obstacle with regards to the detection threshold. Subsequently, the coordinates of the obstacle are computed using the coordinates of the Rover, as well as its orientation, and then sent to Command through the topic “obstacle” within the function below:



```
//Computes the coordinates of the obstacle that triggered the obstacle avoidance protocol, and sends them to Command
void computeAndSendObstacleCoordinatesToCommand(String obstacleColor) {
  long distanceToObstacle = lengthOfRover + (obstacleDetectionThreshold * 3 / 4);
  long xObstacleCoordinate = xRoverCoordinate + ((distanceToObstacle) * sin(roverAngle * PI / 180L));
  long yObstacleCoordinate = yRoverCoordinate + ((distanceToObstacle) * cos(roverAngle * PI / 180L));
  String obstacleCoordinates = String(xObstacleCoordinate) + ":" + String(yObstacleCoordinate);
  Serial.print("info (internal)\t\t: coordinates of obstacle are: ");
  Serial.println(obstacleCoordinates);
  String mqttSendObstacle = obstacleColor + obstacleCoordinates;
  mqttClient.publish(mqttOutTopicObstacle, ArduinoStringToChar(mqttSendObstacle));
}
```

Figure 2-11: function that computes the estimate coordinates of an obstacle and send them to Command over MQTT

Drive

1) Introduction and design process

The Drive subsystem is responsible for the movement of the Mars Rover and measurement of its distance travelled. Therefore, the main requirements of this submodule are to control the direction and speed of the Rover according to the instruction received from the Control subsystem. Moreover, the Rover has to be able to measure the path travelled in the X and Y directions in order to estimate its the exact location.

As seen in *Figure 3-1*, the hardware used to implement the functionality requirements is an Arduino Nano Every. The Arduino receives instructions serially from the ESP32 via UART communication. According to the instruction received, the Arduino is used to control the SMPS, the motor control IC, and the optical flow sensor. The SPMS working in close loop Buck can be controlled in order to provide a desired DC voltage, V_M, that is fed to the two H-bridge circuits built in the motor control IC. By varying the voltage provided to the motor, their speed can be controlled. Moreover, depending on different control signals from the Arduino (PWMR, PWRL, DIRR, DIRL), a motor can be turned on /off, and the direction of rotation can be controlled. In order to measure the distance travelled in X and Y directions, the microcontroller communicates with the optical flow sensor (using the SPI port).

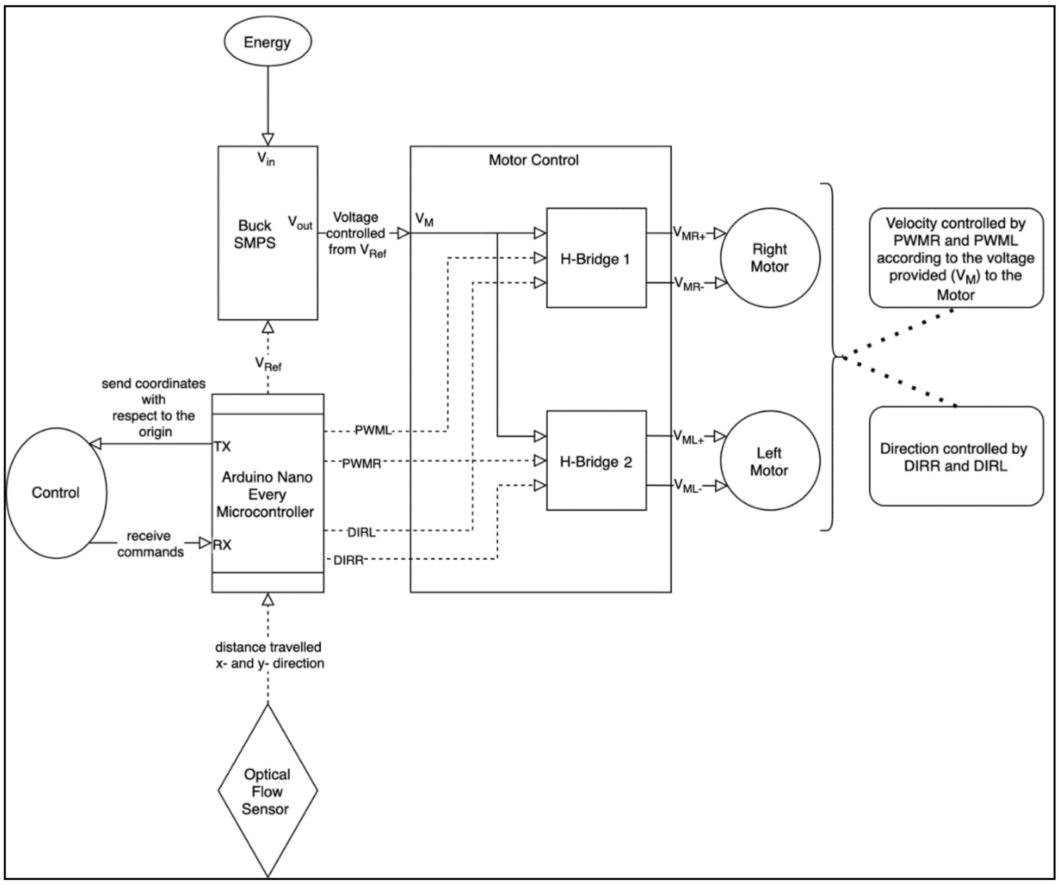

Energy
Vin
Buck SMPS
Vout
Voltage controlled from VRef
Motor Control
VM
H-Bridge 1
H-Bridge 2
VMR+
VMR-
VML+
VML-
Right Motor
Left Motor
Velocity controlled by PWMR and PWML according to the voltage provided (VM) to the Motor
Direction controlled by DIRR and DIRL
VRef
send coordinates with respect to the origin
TX
Control
receive commands
RX
Arduino Nano Every Microcontroller
PWML
PWMR
DIRL
DIRR
distance travelled x- and y- direction
Optical Flow Sensor

Figure 3-1: Structural diagram of the Drive subsystem

2) Speed control

The first step for the implementation of the Drive submodule is related to the ability of the Rover to change speed from a given instruction. As seen in *Figure 3-2*, the functional diagram shows the process of varying the velocity of the Rover between 10 different levels.

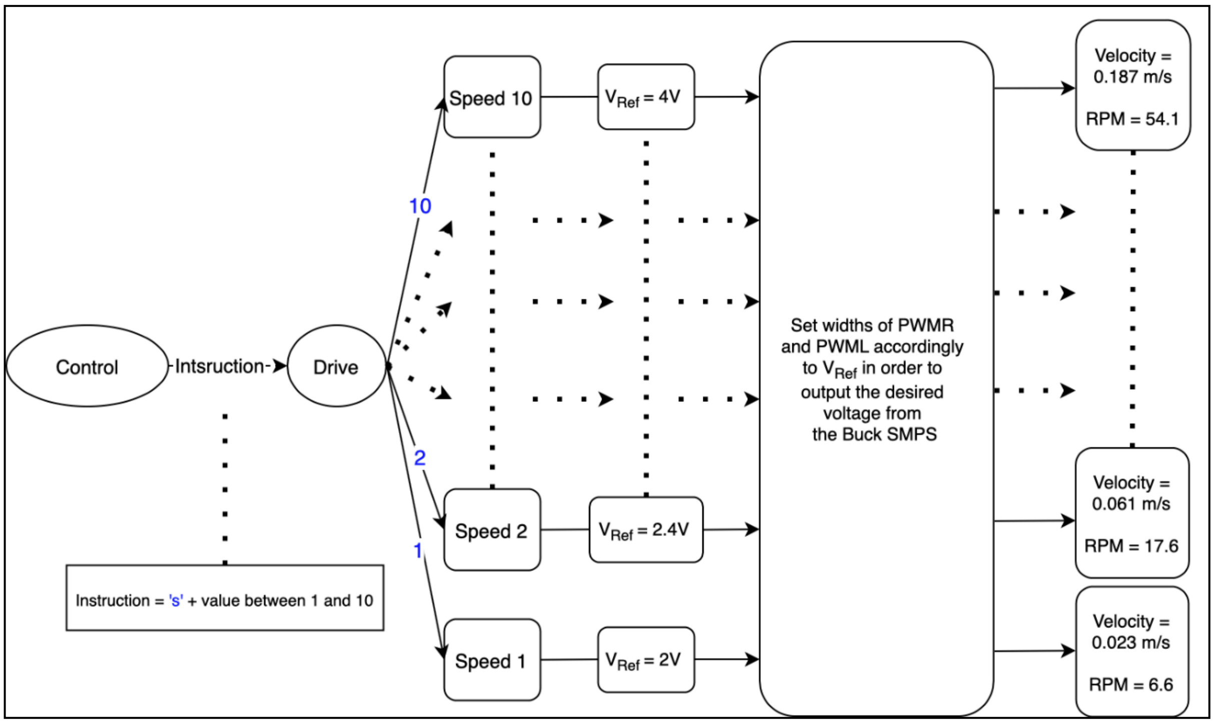


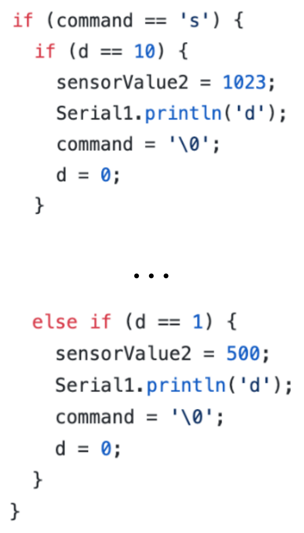
Figure 3-2: Speed Control functional diagram

The implementation of the speed control involves the variation of the reference voltage (V_{Ref},) of the Buck SMPS. Changing V_{Ref}, implies that the width of the pulse width modulated signal (PWM) is adjusted, thus the duty cycle also varies accordingly since they are proportional. Therefore, the DC voltage output from the Buck SMPS, that is fed to the motor control IC, is scaled according to V_{Ref}.

However, since the reference voltage is physically regulated with a potentiometer which has digital values between 0 and 1023, the corresponding the desired voltages had to be computed. Therefore, in order to implement all 10 desired speed levels, voltages values between 1 and 5 could be implemented. Nevertheless, when taking into account the load of the Rover, the threshold voltages were chosen to be 2V and 4V. These anolog voltages had to be converted to their digital significance between 500 and 1023 using the pre-given function:

$$digital = \frac{1023}{4.096} V_{ref}$$

As a result, the implementation in Arduino is as follows: when the character 's' is received, it checks what speed level corresponds with the instruction and sets the variable `sensorValue2` equal to a digital value that agrees with the desired V_{Ref}. Subsequently, the duty cycle and the width of the pulse width modulated signals are varied according to the new V_{Ref} and when the variables PWRR and PWRL (pins 5 and 6) are set "HIGH", the SPMS outputs a voltage V_M almost equal to the desired one. By setting the maximum voltage of the motors equal to V_{Ref} the rotations per minute (RPM) that can be achieved are limited and speed is controlled.



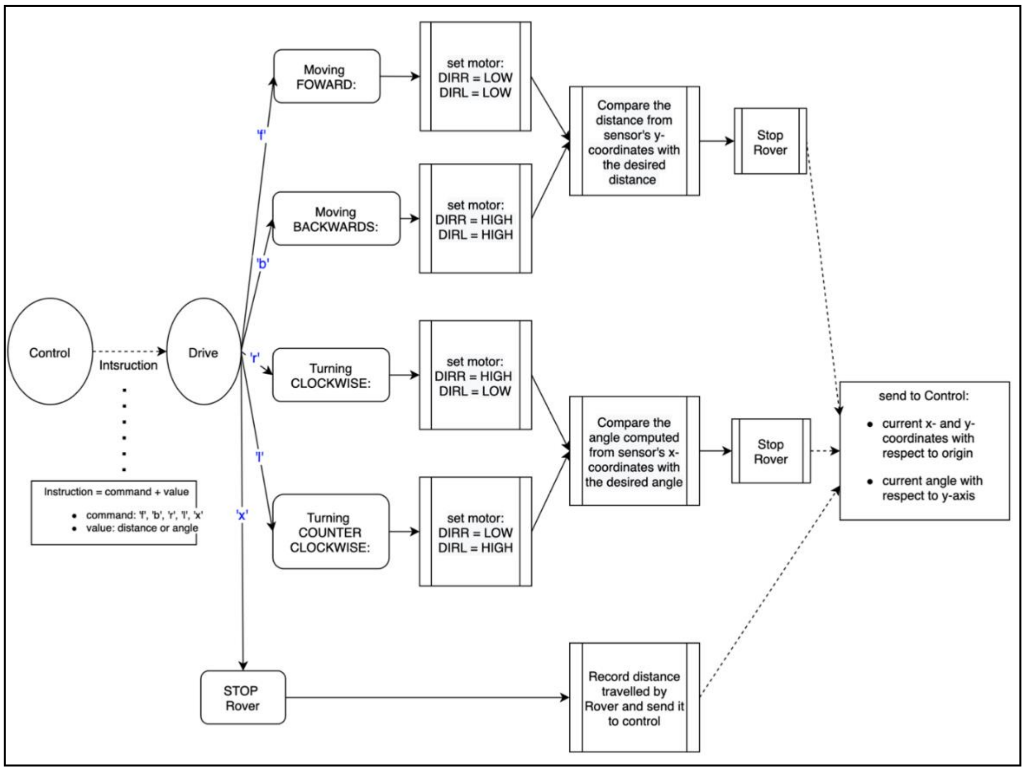
```
if (command == 's') {
  if (d == 10) {
    sensorValue2 = 1023;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }

          ...

  else if (d == 1) {
    sensorValue2 = 500;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
}
```

Figure 3-3: code snippet of the speed control algorithm

As seen in *Figure 3-3*, only the first and last if statements are shown which set `sensorValue2` equal to 1023 and 500 respectively. The remaining 8 cases follow the same format with a different digital value. In order to see full the speed control algorithm, refer to Appendix B.

3) Direction control

The second requirement of this subsystem involves creating an algorithm able to perform the movement instructions received from the Control submodule. The diagram in *Figure 3-4*, provides insight on the algorithm functionality.



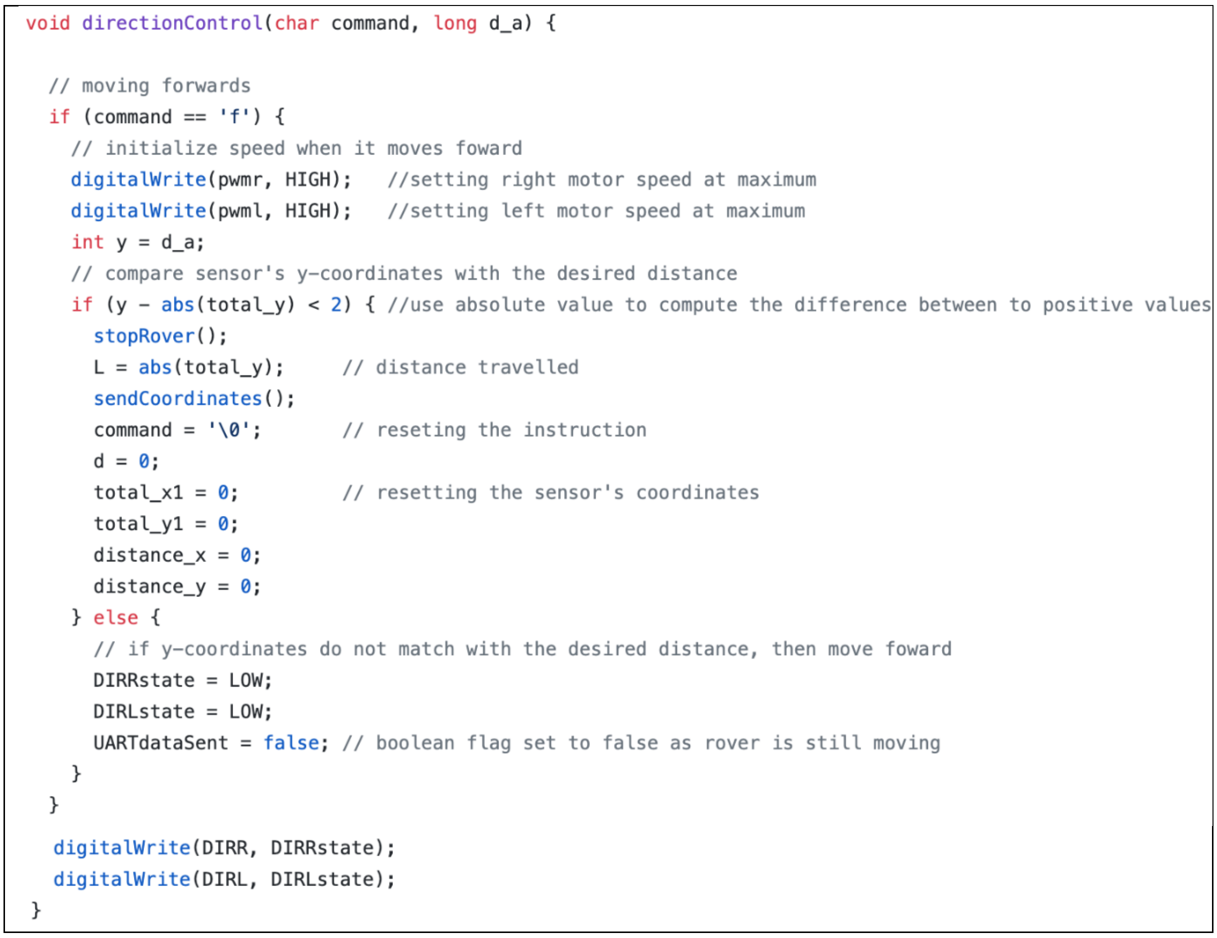
Control
Intsruction
Drive
Instruction = command + value
command: 'f', 'b', 'r', 'l', 'x'
value: distance or angle
'f'
'b'
'r'
'l'
'x'
Moving FOWARD:
Moving BACKWARDS:
Turning CLOCKWISE:
Turning COUNTER CLOCKWISE:
STOP Rover
set motor: DIRR = LOW DIRL = LOW
set motor: DIRR = HIGH DIRL = HIGH
set motor: DIRR = HIGH DIRL = LOW
set motor: DIRR = LOW DIRL = HIGH
Compare the distance from sensor's y-coordinates with the desired distance
Compare the angle computed from sensor's x-coordinates with the desired angle
Record distance travelled by Rover and send it to control
Stop Rover
Stop Rover
send to Control:
current x- and y-coordinates with respect to origin
current angle with respect to y-axis

Figure 3-4: Direction Control functional diagram

As seen from the functional diagram in *Figure 3-4*, and the code in *Figure 3-5*, the control system for direction is implemented with the function `void directionControl.` It uses two inputs: a character corresponding to a moving instruction (‘f’, ‘b’, ‘r’, ‘l’) and a long integer corresponding to a direction or an angle. These determine how to activate the motors directions.

As seen in *Figure 3-5*, when a forward instruction is desired, the character ‘f’ is received, and the microcontroller runs the first *if* statement of the function. At this stage, the function compares the input desired distance that the Rover is supposed to travel with the absolute value of the Y coordinates of the optical flow sensor. As long as their difference is more than 2 mm, the program sets the variables DIRRstate and DIRLstate both equal to “LOW”. These two Boolean variables indicate the orientation of the rotation of the wheels. Therefore, by writing to the corresponding pins (20, 21) with the same Boolean value, then the direction of rotation of the wheels is the same and the Rover moves forward.

Once the Rover reaches the desired destination, the previous condition is met and the program can stop the Rover as the instruction is completed, save the distance travelled (variable `L`) and send the coordinates back to the control submodule. Moreover, the instruction is cleared set to zero and the coordinates of the sensor are reset. This allows the Rover to receive the next instruction and being able to travel by a new distance or turn by a certain angle with respect to its previous orientation.



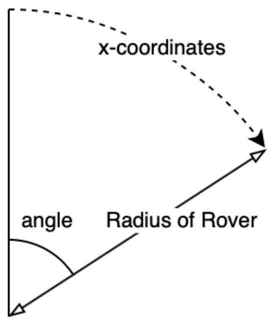
```
void directionControl(char command, long d_a) {

  // moving forwards
  if (command == 'f') {
    // initialize speed when it moves foward
    digitalWrite(pwmr, HIGH);   //setting right motor speed at maximum
    digitalWrite(pwml, HIGH);   //setting left motor speed at maximum
    int y = d_a;
    // compare sensor's y-coordinates with the desired distance
    if (y - abs(total_y) < 2) { //use absolute value to compute the difference between to positive values
      stopRover();
      L = abs(total_y);      // distance travelled
      sendCoordinates();
      command = '\0';        // reseting the instruction
      d = 0;
      total_x1 = 0;          // resetting the sensor's coordinates
      total_y1 = 0;
      distance_x = 0;
      distance_y = 0;
    } else {
      // if y-coordinates do not match with the desired distance, then move foward
      DIRRstate = LOW;
      DIRLstate = LOW;
      UARTdataSent = false; // boolean flag set to false as rover is still moving
    }
  }

  digitalWrite(DIRR, DIRRstate);
  digitalWrite(DIRL, DIRLstate);
}
```

Figure 3-5: code snipped of the forward direction control

The implementation of the backwards instruction follows the same logic with some minor changes. Firstly, the first if statement checks that the command corresponds to the character ‘b’ in order to perform the instruction. Secondly as long as the sensor’s Y coordinates are not equal to the desired distance, the variables DIRRstate and DIRLstate are both set to “HIGH” and the Rover should move. Once the Rover completes the instruction, the distance travelled is updated with a negative value since the it travelled in the opposite direction. Refer to Appendix C for the complete code implementation.

Turning method

On the other hand, when dealing with a turning instruction, the modifications to the previous code are quite different. Firstly, it is important to point out that the Rover rotates around itself, therefore the X coordinates of the sensor provide the arc travelled by the sensor. As seen in figure 3-6, by measuring the radius of the Rover, the angle by which the Rover turns around itself can be computed.



$$angle = \frac{arc}{R} \times \frac{180}{\pi} = \frac{x - coordinates}{R} \times \frac{180}{\pi}$$

Figure 3-6: arc and radius ratio

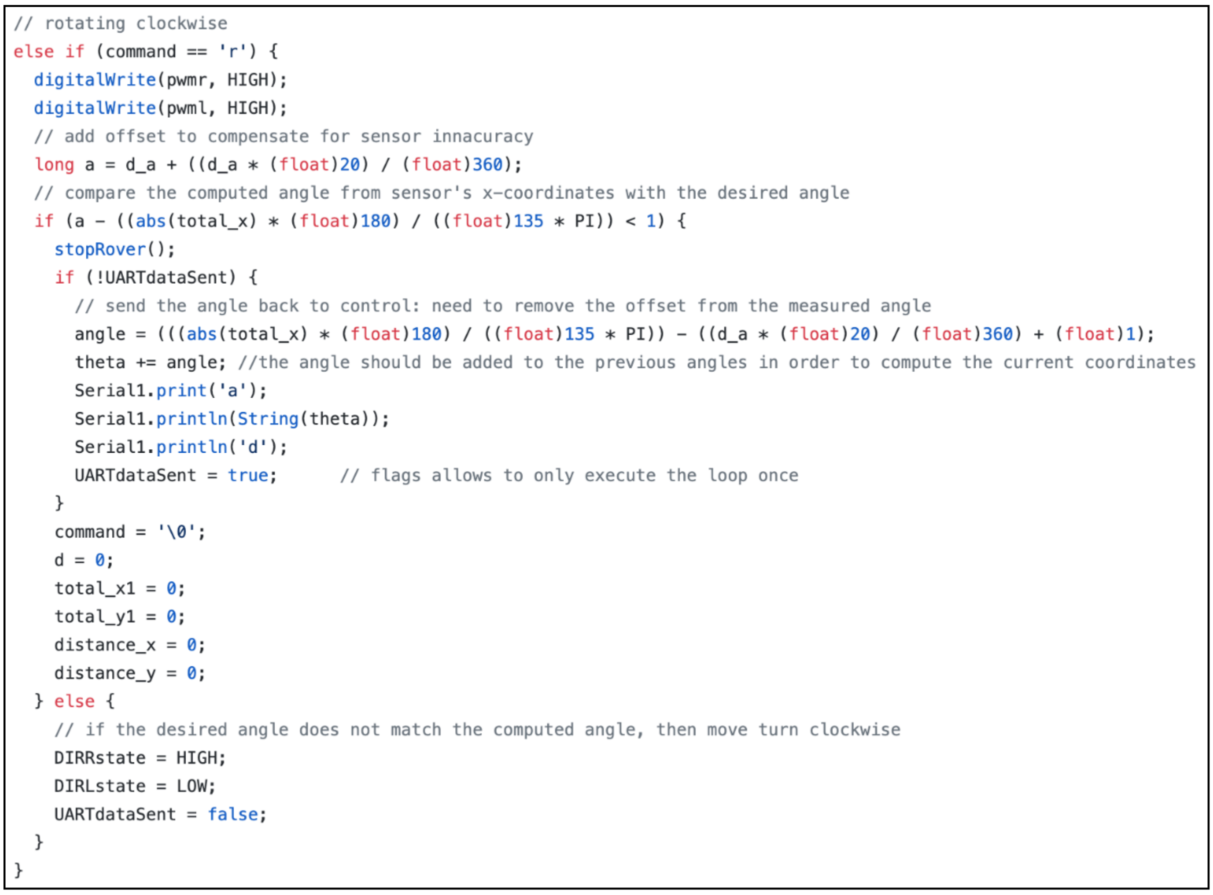
However, when performing short unit tests, we realized that an offset had to be added to the angle in order to account for inaccuracies of the optical sensor and obtain a more precise performance. Therefore, an offset proportional to the desired angle was derived:

$$offset = desired\ angle \times \frac{20}{360}$$

Figure 3-7 shows the implementation of the clockwise instruction. When character ‘r’ is received, the desired angle is adjusted by adding this offset. This has to be executed in order to perform the rotation by the desired angle. At this point the program starts comparing the new desired angle with the actual computed angle from the sensor’s coordinates and checks whether their difference is less than 1 degree. As long as this condition is not met, the variables DIRRstate and DIRLstate are set to logic “HIGH” and “LOW” respectively. By fixing the motors with opposite turning directions, the Rover is allowed to rotate in a clockwise manner.

Once the desired angle is reached, the program can stop the Rover as the instruction is completed. At this stage the measured angle needs removal of the previously added offset. This is of crucial importance because, while the Rover visually turns by the correct angle, the computed angle obtained from the sensor’s X coordinates also includes the offset which must be accounted for. Subsequently the corrected angle is added to the variable `theta`. This corresponds to the orientation of the Rover with regards to the y-axis. This is a key step to calculate the exact location of the Rover in a cartesian plane.

Lastly, once again, the instruction parameters must be cleared, and the coordinates of the sensor are reset to zero.



```
// rotating clockwise
else if (command == 'r') {
  digitalWrite(pwmr, HIGH);
  digitalWrite(pwml, HIGH);
  // add offset to compensate for sensor innacuracy
  long a = d_a + ((d_a * (float)20) / (float)360);
  // compare the computed angle from sensor's x-coordinates with the desired angle
  if (a - ((abs(total_x) * (float)180) / ((float)135 * PI)) < 1) {
    stopRover();
    if (!UARTdataSent) {
      // send the angle back to control: need to remove the offset from the measured angle
      angle = (((abs(total_x) * (float)180) / ((float)135 * PI)) - ((d_a * (float)20) / (float)360) + (float)1);
      theta += angle; //the angle should be added to the previous angles in order to compute the current coordinates
      Serial1.print('a');
      Serial1.println(String(theta));
      Serial1.println('d');
      UARTdataSent = true;      // flags allows to only execute the loop once
    }
    command = '\0';
    d = 0;
    total_x1 = 0;
    total_y1 = 0;
    distance_x = 0;
    distance_y = 0;
  } else {
    // if the desired angle does not match the computed angle, then move turn clockwise
    DIRRstate = HIGH;
    DIRLstate = LOW;
    UARTdataSent = false;
  }
}
```

Figure 3-7: code snipped of the clockwise turning method

The implementation of the counterclockwise instruction follows the same logic, where the main differences are due to the direction of rotation of the motors. The variables DIRRstate and DIRLstate are inverted, and the angle computed once the Rover has stopped is updated with a negative value, since it is moving in the opposite direction to the previous instruction. For the full code implementation refer to Appendix C.

Stop Rover

As previously seen in the direction control function, the Rover is stopped using the function `void stopRover()`, which writes the PWM pins for the right and left motors (pins 5 and 9 respectively) with Boolean logic 0. This fixes the Buck SMPS PWM to zero, hence setting its duty cycle to zero, and therefore no voltage is provided to the H-Bridges.

Secondly, *Figure 3-8* displays another function `void stopRoverOnCommand()` was also implemented in order to be able to stop moving when character ‘x’ is received from Control. The purpose of this function is to interrupt the previous instruction and stop the Rover when an obstacle is detected by the front camera. When this function is called, it transmits the distance travelled and the current coordinates of the Rover to be able to save its path. This allows the control system to run its obstacle avoidance algorithm and then resume the previous instruction coherently with the diverted path.

```
void stopRover() {
  digitalWrite(pwmr, 0);
  digitalWrite(pwml, 0);
}

void stopRoverOnCommand() {
  if (command == 'x') {
    Serial1.println('d');
    digitalWrite(pwmr, 0);
    digitalWrite(pwml, 0);
    command = '\0';

    L = abs(total_y);
    long distanceTravelled = L;
    sendCoordinatesOnCommand();
    Serial1.println(String(distanceTravelled));
    UARTdataSent = false;
  }
}
```

Figure 3-8: Stop Rover functions

4) Coordinates

Computation of coordinates from instructions

After performing an instruction involving the motion of the Rover, the coordinates with respect to the origin must be computed in order to map the distance travelled by the Rover.

As mentioned in the previous section, the current distance travelled (Y coordinates of the sensor), the computed angle with respect to the Rover's previous orientation (from the X coordinates of the sensor), and the sum of all previous angles are saved in variables L, `angle` and `theta` respectively. These parameters need to be saved because they are used to compute the current coordinates of the Rover, since the sensor is initialized at $x = 0$ and $y = 0$ after every movement instruction.

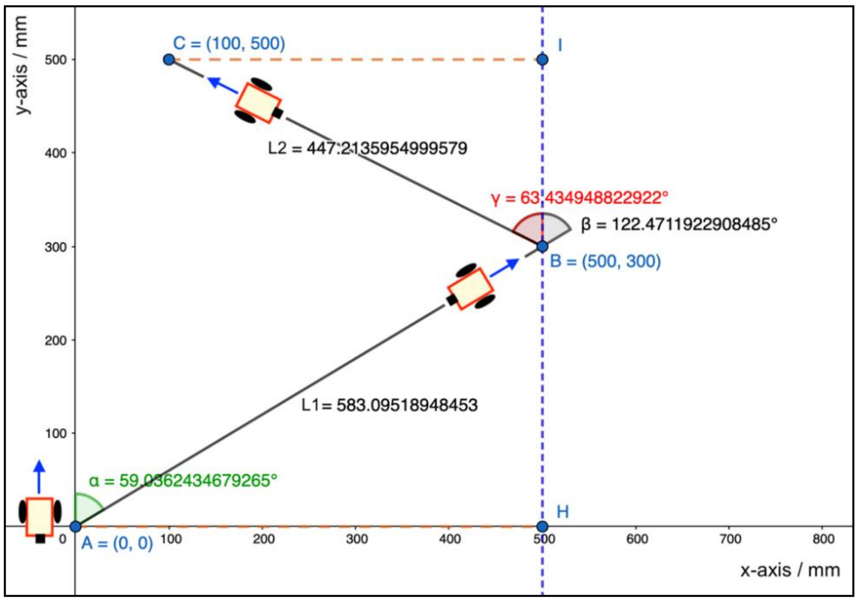


Figure 3-9: arbitrary coordinates diagram

The general mathematical formula can be obtained by taking three arbitrary points on a cartesian plane. As seen from *Figure 3-9*, when the Rover is placed at the origin, in order to reach point B, it has to turn clockwise by angle an α and go forward by a distance L1. Hence, to compute the coordinates of point B, simple trigonometry rules of a right-angle triangle can be applied to find the segments $\overrightarrow{AH}$ and $\overrightarrow{HB}$ and the X and Y coordinates:

$$\overrightarrow{AH} = L_1 \sin(\alpha) = x \qquad (1)$$

$$\overrightarrow{HB} = L_1 \cos(\alpha) = y \qquad (2)$$

The same method applies for point C. However, it has to be noted that the angle received to reach point C is β, which is measured counter clockwise form the orientation of the Rover at point B. Since the angle is measured in the opposite direction, β is negative. Then, to compute the segments $\overrightarrow{IC}$ and $\overrightarrow{BI}$, the angle with y-axis at point B, γ, has to be computed. The angle γ is the summation of the previous angles (α and $-\beta$). Therefore, the segments are given from:

$$\overrightarrow{IC} = L_2 \sin(\gamma) = L_2 \sin(\alpha - \beta) \qquad (3)$$

$$\overrightarrow{BI} = L_2 \cos(\gamma) = L_2 \cos(\alpha - \beta) \qquad (4)$$

Consequently, the X coordinates at point C, is the vector sum of $\overrightarrow{AH}$ and $\overrightarrow{IC}$, while the Y coordinates is the vector sum of $\overrightarrow{HB}$ and $\overrightarrow{BI}$. The formulae for the coordinates can then be generalized by computing the vector sum of the previous segments with the current ones.

$$x = L_1 \sin(\alpha) + L_2 \sin(\alpha - \beta) + \cdots \qquad (5)$$

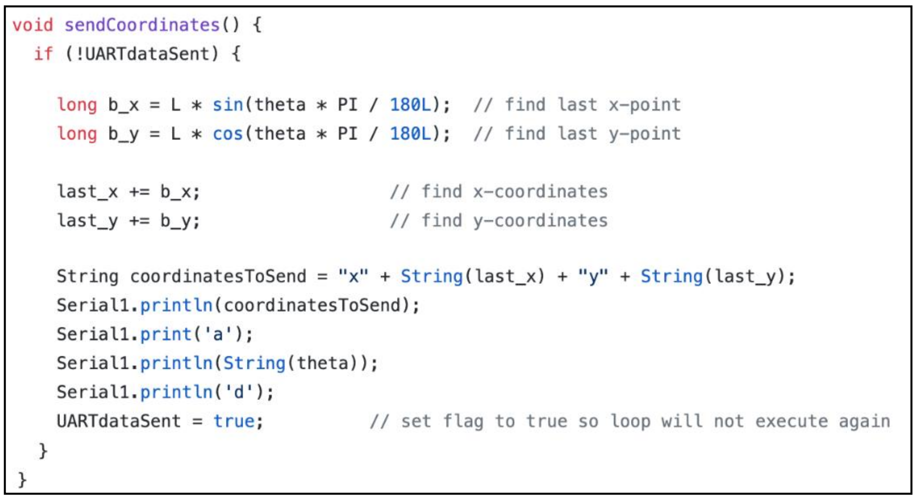
$$y = L_1 \cos(\alpha) + L_2 \cos(\alpha - \beta) + \cdots \qquad (6)$$

The above formulas can be simplified to just summation of the previous coordinates with the new segments calculated from the last travelled length, L, and angle with respect to the y-axis (summation of all the previous angles and current angle, θ):

$$x = last\ x + L \sin(\theta) \qquad (7)$$

$$y = last\ y + L \cos(\theta) \qquad (8)$$

As seen in *Figure 3-10*, the function `void sendCoordinates()` implements formulas (7) and (8) using the last distance in the Y direction recorded by the sensor, `L`, and the summation of all the angles, `theta`, in order to compute and send the X and Y coordinates back to control.



```
void sendCoordinates() {
  if (!UARTdataSent) {

    long b_x = L * sin(theta * PI / 180L);  // find last x-point
    long b_y = L * cos(theta * PI / 180L);  // find last y-point

    last_x += b_x;                  // find x-coordinates
    last_y += b_y;                  // find y-coordinates

    String coordinatesToSend = "x" + String(last_x) + "y" + String(last_y);
    Serial1.println(coordinatesToSend);
    Serial1.print('a');
    Serial1.println(String(theta));
    Serial1.println('d');
    UARTdataSent = true;          // set flag to true so loop will not execute again
  }
}
```

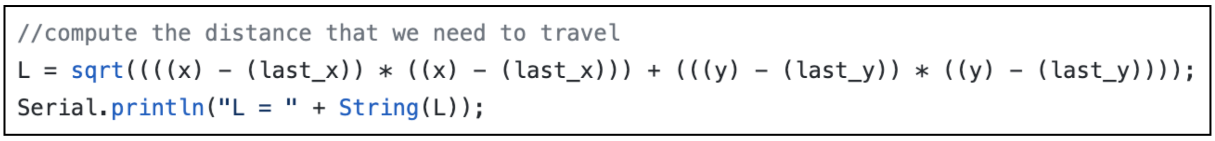
Figure 3-10: Conversion to coordinates

Computation of Instructions from Coordinates

From a set of coordinates, it is possible to compute the current distance travelled, L, and the turning angle with respect to the Rover's previous orientation.

First of all, in order to compute the segment of distance travelled between two arbitrary points in a Cartesian plane, Pythagoras theorem for a right-angle triangle can be applied, and directly implemented in the code (*Figure 3-11*).

$$L = \sqrt{(x - last\ x)^2 + (y - last\ y)^2} \qquad (8)$$



```
//compute the distance that we need to travel
L = sqrt((((x) - (last_x)) * ((x) - (last_x))) + (((y) - (last_y)) * ((y) - (last_y))));
Serial.println("L = " + String(L));
```

Figure 3-11: Computation of distance travelled

Subsequently, in order to find a formula for the turning angle it is possible to invert equation (7) and use it to find the angle with respect to the y-axis.

$$\theta = \arcsin\left(\frac{x - last\ x}{L}\right) \qquad (9)$$

Since θ is the summation of all the angles, ($\theta =$ sum of previous angles + current angle), the current angle can then be computed by:

$$current\ angle = \arcsin\left(\frac{x - last\ x}{L}\right) - sum\ of\ previous\ angles \qquad (10)$$

The implementation of this algorithm is in the function `void translateCoordinates(long x, long y)` in the Control submodule. The algorithm needs to take into account the limits of the argument of the arcsine function.

As seen in Figure 3-12, arcsine can return real values between 1 and -1. Ideally the argument, $\frac{x-last\ x}{L}$, should stay within the constraints. However, since the last X coordinates are sent by the Rover, there might be inaccuracies of measurements recorded by the sensor. This could lead to the nominator being bigger than numerator, resulting in an argument more than 1.

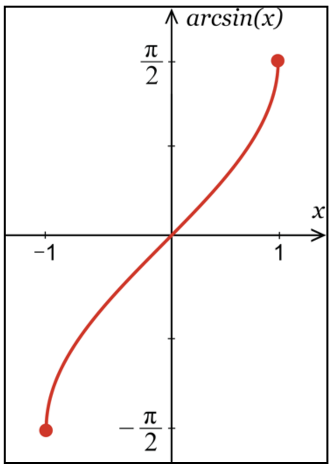
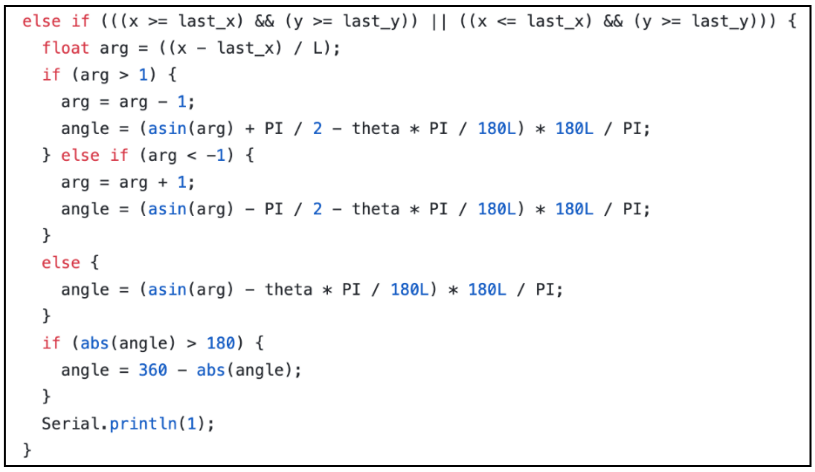


Figure 3-12: arcsine graph

In this case, as seen in *Figure 3-13*, the algorithm removes 1 from the argument and then adds 90 degrees after computing the final angle. As a result, the angle remains a real value that can be converted into an instruction in practice. The same principle applies for the case of argument equal to -1, where the algorithms firstly adds one and then subtracts 90 degrees.



```
else if (((x >= last_x) && (y >= last_y)) || ((x <= last_x) && (y >= last_y))) {
  float arg = ((x - last_x) / L);
  if (arg > 1) {
    arg = arg - 1;
    angle = (asin(arg) + PI / 2 - theta * PI / 180L) * 180L / PI;
  } else if (arg < -1) {
    arg = arg + 1;
    angle = (asin(arg) - PI / 2 - theta * PI / 180L) * 180L / PI;
  }
  else {
    angle = (asin(arg) - theta * PI / 180L) * 180L / PI;
  }
  if (abs(angle) > 180) {
    angle = 360 - abs(angle);
  }
  Serial.println(1);
}
```

Figure 3-13: Computation of turning angle

Figure 3-13 only shows the case where the given Y coordinates are higher than the previous ones. The implementation of the case where Y coordinate is less or equal to the previous Y is divided into two separates if statements. This is because if the X coordinate is more than the previous X, the output of the arcsine function is subtracted to 180 in order to move around the unit circle in the correct quadrant. While, if the X coordinate is less than the previous X, then a factor of 180 has to be subtracted (Appendix D).

Vision

1) Overview

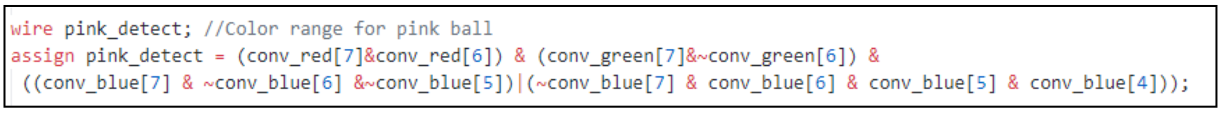
The purpose of vision is to detect the distance and the presence of different coloured balls. Our vision sensor aims to fulfil this purpose by selecting the pixels which correspond to the balls. This information is then passed to the ESP32 through a UART connection.

2) Verilog

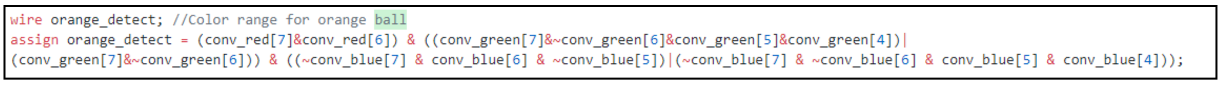
The vision processing is carried out in Verilog by comparing the RGB values seen in the FPGA's camera. As we know each ball corresponds to a colour, we can set a range of red, green, and blue values which match each ball. The values are given in the table below.

	Pink	**Orange**	**Green**	**Blue**
Red	255-192	255-192	131-70	134-75
Green	175-160	191-128	216-126	179-107
Blue	159-96	98-48	98-62	152-107

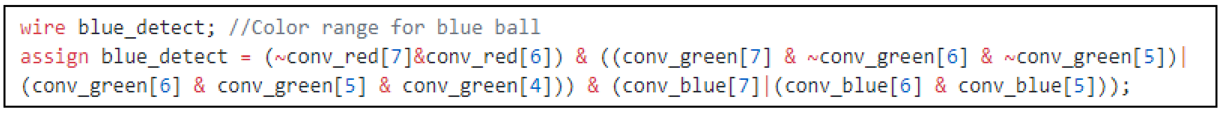
These ranges are implemented in binary. For example, the requirements on the red bit for pink are (red[7] & red[6]) which corresponds to the values 255 to 192. As all these colour ranges will have at least one colour range which is different from the rest, there will be no conflicts in colour decisions. The binary implementation is shown in the code snippets below.



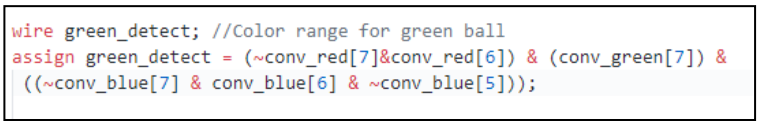
```
wire pink_detect; //Color range for pink ball
assign pink_detect = (conv_red[7]&conv_red[6]) & (conv_green[7]&~conv_green[6]) &
 ((conv_blue[7] & ~conv_blue[6] &~conv_blue[5])|(~conv_blue[7] & conv_blue[6] & conv_blue[5] & conv_blue[4]));
```



```
wire orange_detect; //Color range for orange ball
assign orange_detect = (conv_red[7]&conv_red[6]) & ((conv_green[7]&~conv_green[6]&conv_green[5]&conv_green[4])|
(conv_green[7]&~conv_green[6])) & ((~conv_blue[7] & conv_blue[6] & ~conv_blue[5])|(~conv_blue[7] & ~conv_blue[6] & conv_blue[5] & conv_blue[4]));
```



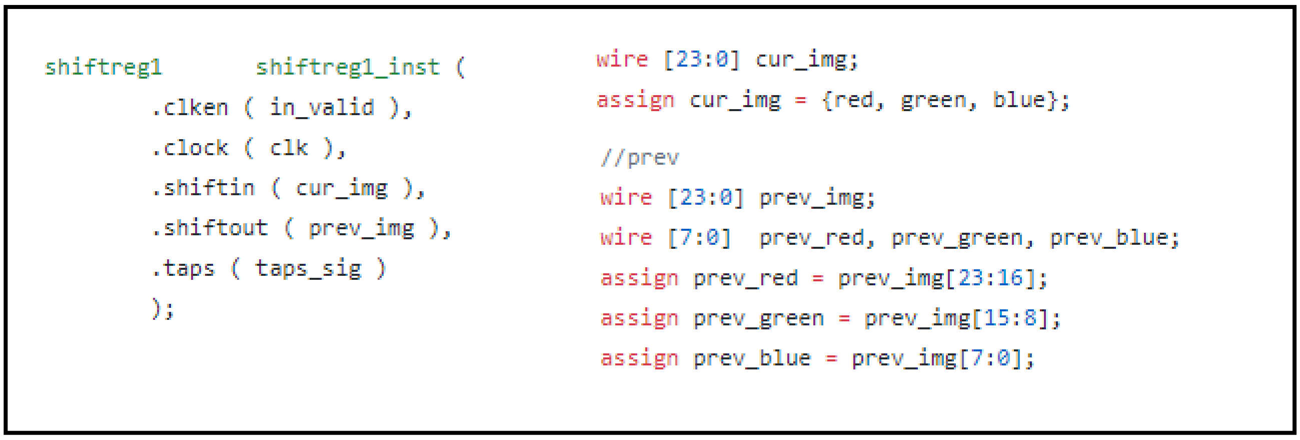
```
wire blue_detect; //Color range for blue ball
assign blue_detect = (~conv_red[7]&conv_red[6]) & ((conv_green[7] & ~conv_green[6] & ~conv_green[5])|
(conv_green[6] & conv_green[5] & conv_green[4])) & (conv_blue[7]|(conv_blue[6] & conv_blue[5]));
```



```
wire green_detect; //Color range for green ball
assign green_detect = (~conv_red[7]&conv_red[6]) & (conv_green[7]) &
 ((~conv_blue[7] & conv_blue[6] & ~conv_blue[5]));
```

Figure 4-1: Binary Ranges for each Ball

This approach will give us a basic colour detection system that can be used for the rover. Nonetheless, there are many glitches in detection, and non-ball objects in the frame will also be detected. We have implemented a blurring convolution filter to help remedy this problem. Alternative solutions would be to just defocus the camera, but we wanted to properly implement a gaussian blur. Our code uses shift registers to implement a cross-shaped convolution filter, which averages the current pixel with its surrounding pixels. A blurring filter serves the same purpose as a low-pass filter as it "smoothes out" large differences in color.

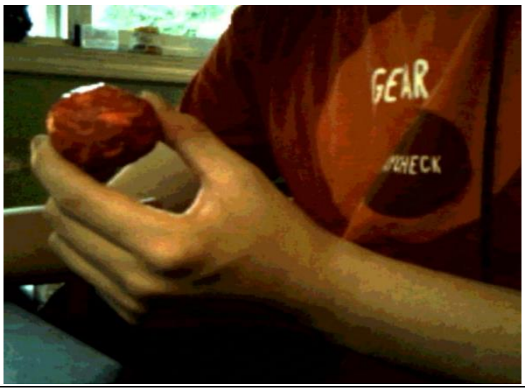
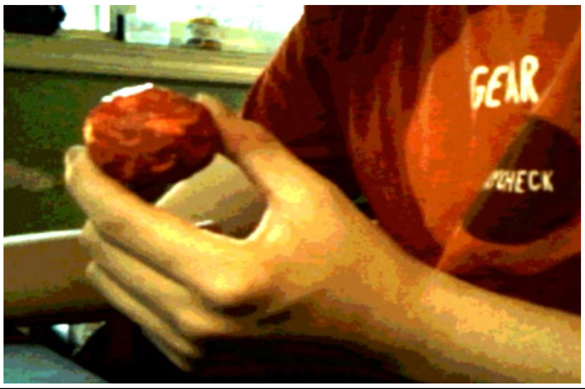


```
shiftreg1       shiftreg1_inst (
        .clken ( in_valid ),
        .clock ( clk ),
        .shiftin ( cur_img ),
        .shiftout ( prev_img ),
        .taps ( taps_sig )
        );
```

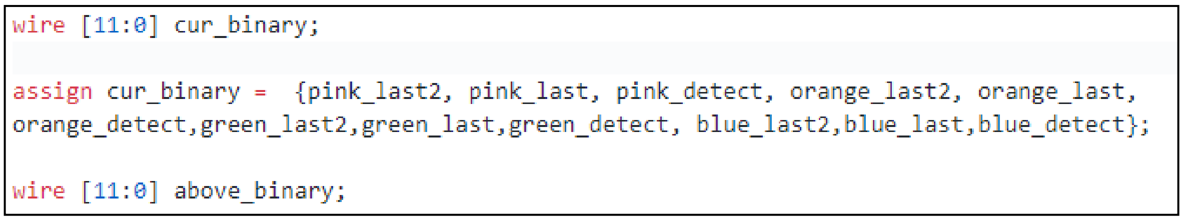
```
wire [23:0] cur_img;
assign cur_img = {red, green, blue};

//prev
wire [23:0] prev_img;
wire [7:0]  prev_red, prev_green, prev_blue;
assign prev_red = prev_img[23:16];
assign prev_green = prev_img[15:8];
assign prev_blue = prev_img[7:0];
```

Figure 4-2: Blurring convolution implementation for previous pixel

In the code snippet above, the previous pixel is stored in `cur_img` input and is received from the `prev_img` output of the shift register. It is then broken down into its component red, green, and blue parts. Each component is summed up for all pixels, and the mean is the resulting image.

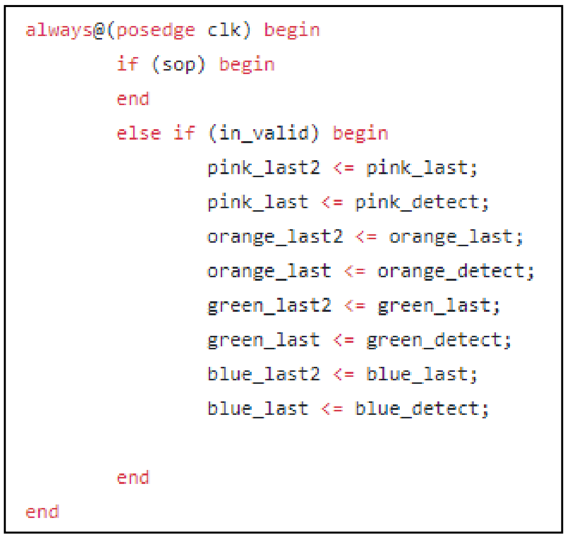
No Blurring Filter Implemented	Blurring filter implemented
	

The convolution filter greatly improves stray colors being detected and random artifacts, but a binary convolution filter is also implemented as well to further improve the performance. When the image is passed through the first convolution filter and detected by the various color ranges, the image is represented as separate bitmaps for each color. For example, the orange bitmap would have false where the orange ball is not detected and true for where it is. The binary convolution is a 2x3 filter which requires all six bits near the current bit to be true for this bit to be detected. This filter is effective in removing small patches of colors in the background. To improve performance and reduce the area on the chip, the past three binary values are stored in sequential registers similar to a ripple counter. These three values for each coloured ball are then passed to a shift register for use 639 shifts later corresponding to the row above. A total of 12 bits (three for each ball colour) are shifted each image cycle (Figures 4-3 and 4-4).



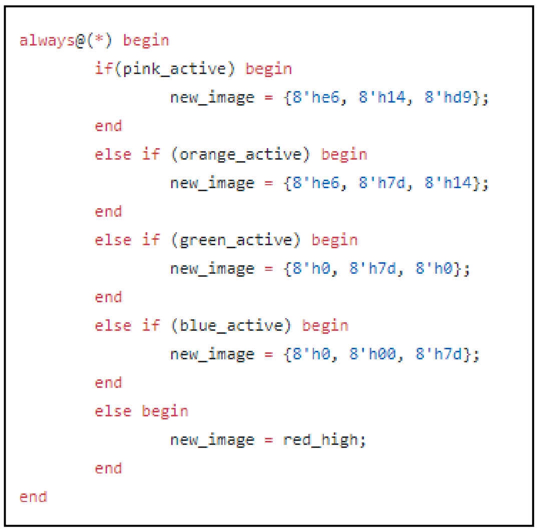
```
wire [11:0] cur_binary;

assign cur_binary =  {pink_last2, pink_last, pink_detect, orange_last2, orange_last,
orange_detect,green_last2,green_last,green_detect, blue_last2,blue_last,blue_detect};

wire [11:0] above_binary;
```

Figure 4-3: Binary convolution filter shift register



```
always@(posedge clk) begin
        if (sop) begin
        end
        else if (in_valid) begin
                pink_last2 <= pink_last;
                pink_last <= pink_detect;
                orange_last2 <= orange_last;
                orange_last <= orange_detect;
                green_last2 <= green_last;
                green_last <= green_detect;
                blue_last2 <= blue_last;
                blue_last <= blue_detect;

        end
end
```

Figure 4-4: Binary convolution sequential registers



```
always@(*) begin
        if(pink_active) begin
                new_image = {8'he6, 8'h14, 8'hd9};
        end
        else if (orange_active) begin
                new_image = {8'he6, 8'h7d, 8'h14};
        end
        else if (green_active) begin
                new_image = {8'h0, 8'h7d, 8'h0};
        end
        else if (blue_active) begin
                new_image = {8'h0, 8'h00, 8'h7d};
        end
        else begin
                new_image = red_high;
        end
end
```

Figure 4-5: Image color annotation for debugging

3) Debugging and Verification

Although only the most essential information is transferred through UART to the ESP32, it is very import to be able to debug and verify ball detection during the implementation phase. Through the FPGA VGA connection, we annotate the stream by colouring the detected pixels and drawing the bounding box for each colour (Figures 4-5 to 4-7). We also found it very useful to output the colour the FPGA sees, so the middle pixel (320,240) is marked on the stream, and the RGB values at that pixel are output in the NIOS-II terminal.

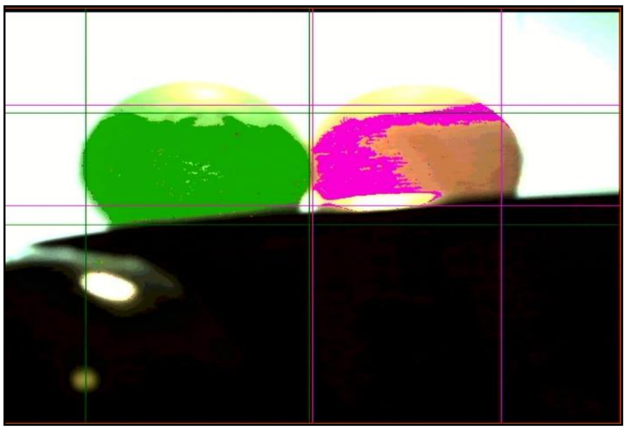


Figure 4-6: Annotated stream for green and pink ball

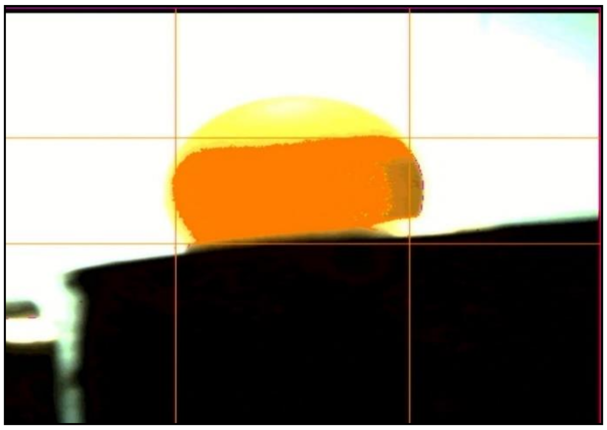
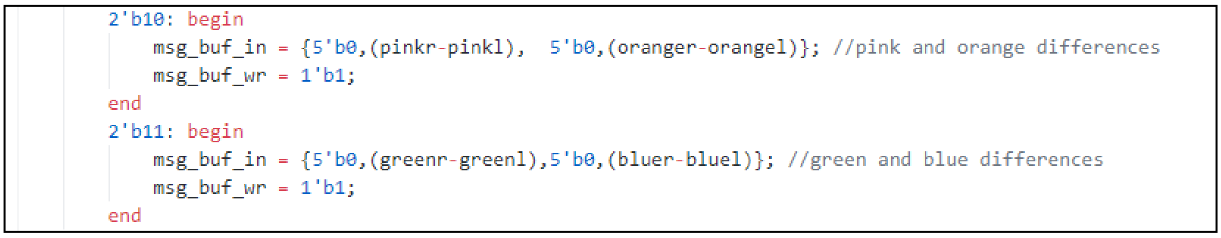


Figure 4-7: Annotated stream for orange ball

4) ESP32 and UART

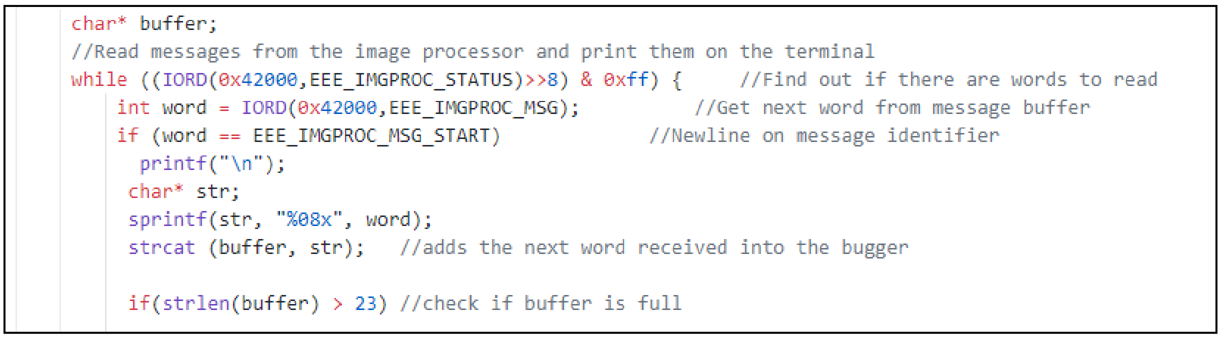
The communication between the FPGA and the rest of the rover is handled through UART to the ESP32. In Verilog, the difference in the leftmost pixel and rightmost pixel for each ball colour is output. This allows us to calculate the distance of the ball from the rover based on the apparent size of the ball.



```
2'b10: begin
    msg_buf_in = {5'b0,(pinkr-pinkl),  5'b0,(oranger-orangel)}; //pink and orange differences
    msg_buf_wr = 1'b1;
end
2'b11: begin
    msg_buf_in = {5'b0,(greenr-greenl),5'b0,(bluer-bluel)}; //green and blue differences
    msg_buf_wr = 1'b1;
end
```

Figure 4-8: Verilog output messages

The FIFO block is limited to 32-bit messages, so the differences need to be split into two messages, which is then captured by the NIOS-II processor. The NIOS-II stores all incoming messages in a 24-character (96 bit) string buffer, which when full, extracts each difference from its corresponding bit numbers and empties the buffer.



```
char* buffer;
//Read messages from the image processor and print them on the terminal
while ((IORD(0x42000,EEE_IMGPROC_STATUS)>>8) & 0xff) {     //Find out if there are words to read
    int word = IORD(0x42000,EEE_IMGPROC_MSG);          //Get next word from message buffer
    if (word == EEE_IMGPROC_MSG_START)              //Newline on message identifier
      printf("\n");
     char* str;
     sprintf(str, "%08x", word);
     strcat (buffer, str);   //adds the next word received into the bugger

     if(strlen(buffer) > 23) //check if buffer is full
```

Figure 4-9: NIOS-II String buffer for receiving messages

The differences are used to calculate the distance from the ball to the rover. If a ball is detected, the ESP32 will receive the colour of the ball, and whether it is close enough to trigger the obstacle avoidance routine.

Energy

The Energy subsystem is responsible for characterising the battery cells and the PV panels to maximise the capacity of the cell. We also must investigate the various battery and PV panel configuration and how each configuration affects the overall circuit. Once the design has been made, we can start to integrate it with the other subsystems as well as communicating the relevant data to the server.

1) Energy Design Specification

Section	Criteria	Description
Arduino	-Charge Model -Data Collection -SoC Calculation -SoH Management -Cell Safety -Communication -MPPT algorithm -Rover Range	-Charge the battery and implemented a voltage controller at 3.6V -Record the voltage and current of the cell every second -Estimate SoC using coulomb counting -Provide an equation to calculate SoH as well as balancing of the cells -I_{in}(max) = 500mA, V_{bat}(max) = 3.6V, V_{bat}(min) = 2. -Used the mqtt library in python to communicate -Rover range calculated from Rover speed and current
SMPS	-Buck or Boost Configuration	-Buck configuration to step down PV voltage and step-down motor voltage
Battery Pack	-Series or Parallel Configuration -Number of Cells	-Parallel Configuration -2 cells
PV Panel	-Implementation with System -Series or Parallel configuration	-Charging station -4 solar panels in parallel

2) Arduino – Charge Model

For the charge model, we decided to implement a CC/CV (Constant Current/Constant Voltage) charge model. This requires the battery to be charging at a constant current of the standard charging current provided by the data table at the appendix. The battery charges to a maximum value of 3.6V and we applied a voltage controller to keep the voltage constant while the current is gradually decreasing as shown in Figure 1. As a design choice, we stopped charging the battery once it has reached 50mA since it would take too long to reach the ideal value of 0mA. The CV implementation would increase the capacity of the cell since it reduces the trickle voltage when the cell is in the charging rest state.

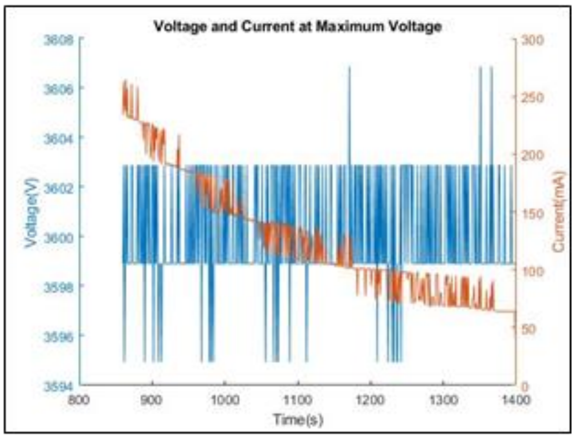


Fig. 5-1: CC-CV Curve

For our charge model, we ideally want to charge the cell until it reaches 0mA where the trickle voltage would theoretically disappear. This would further increase the capacity but because of a halt in battery usage, we were unable to test this. Another change we could have made to the CC/CV graph is to take the average current over 1000 samples in the fast loop. This would help smooth out the curve as it is currently oscillating due to the quantization error in the current sensing resistor seen in Fig. 5-1.

3) Arduino - State of Charge

The capacity of a cell is the amount of charge that can be stored in a cell and is measured in ampere hours (Ah). To obtain the maximum capacity of a cell, we discharge the cell and multiply the discharge current by the time taken to discharge in hours.

The state of charge is a property of a battery that is used to determine how much charge is present in the cell. There are multiple methods to calculate the state of charge but the method we implemented in our design is the coulomb counting method. To calculate the percentage discharge in a second, we multiply the current with time and divide it by the maximum capacitance. Subsequently, we multiply the result by 100 to get the percentage discharged. The state of charge is obtained by subtracting 100 from this value and this is the basis for the coulomb counting method where we keep integrating the current over time until we reach the end of the discharge cycle. [21]

The open circuit voltage (OCV) is another useful measurement which is used to calculate the SoC from an OCV-SoC graph. From this, we can start coulomb counting again if it is already partially discharged. To measure the OCV of a cell, we switch relay on and measure it using the measure pin and the graph below shows the OCV-SoC graph for both cells.

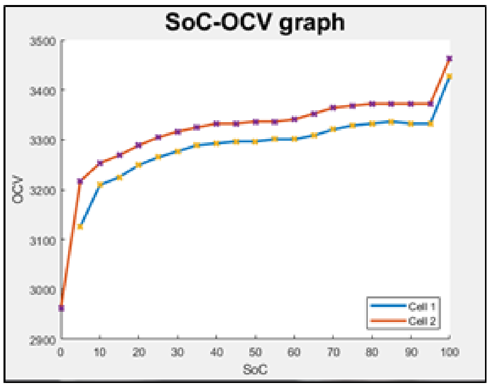
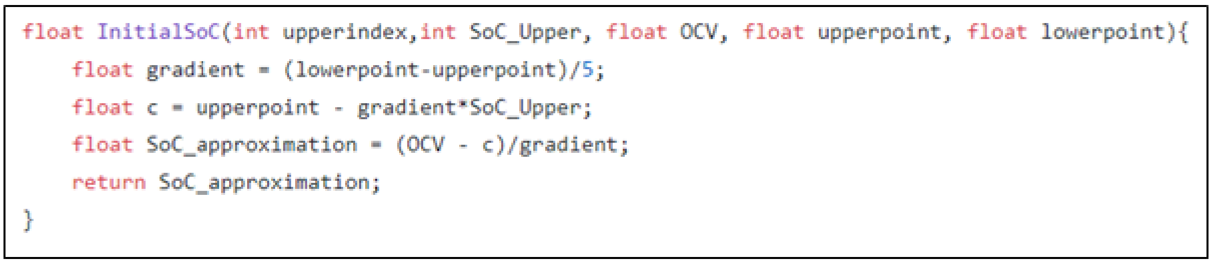


Fig. 5-2: SoC-OCV graph

From the OCV, we compare this to the OCV array we have from Figure 5-2 and estimate the two SoC points this OCV value lies between. Once we have these two points, we can use the function `InitialSoC` shown in Figure 5-3 where we draw a line between the two points and find the intersection to get the SoC.



```
float InitialSoC(int upperindex,int SoC_Upper, float OCV, float upperpoint, float lowerpoint){
    float gradient = (lowerpoint-upperpoint)/5;
    float c = upperpoint - gradient*SoC_Upper;
    float SoC_approximation = (OCV - c)/gradient;
    return SoC_approximation;
}
```

Fig. 5-3: Arduino Code to calculate SoC

4) Arduino – SoH Management

The state of health of the battery is characterized as Maximum Discharge Capacity / Nominal Capacity to give an indication of how the battery is performing compared to the datasheet in the appendix. After many discharge cycles, we expect the performance of the battery to deteriorate. To manage this and to reduce the amount of capacity that decreases from the cell after every cycle, we can charge the cell to its nominal voltage instead of 3.6V. This will come at a cost of a reduction of the capacity at every cycle, but the battery will perform at a consistent rate for a longer period. [22]

5) Arduino – Communication

To establish a connection the server, we first created an Arduino program which sends the SoH and SoC separated by a colon through the serial port e.g. (100:50). Since we are unable to use the batteries, we send the data to the server by decreasing the SoC by 10 every second. To demonstrate the effects of SoH, we decrease the SoH by 10 after every discharge cycle thus adjusting the range of the Rover. To calculate the Rover range, we used a simple equation shown at 1.

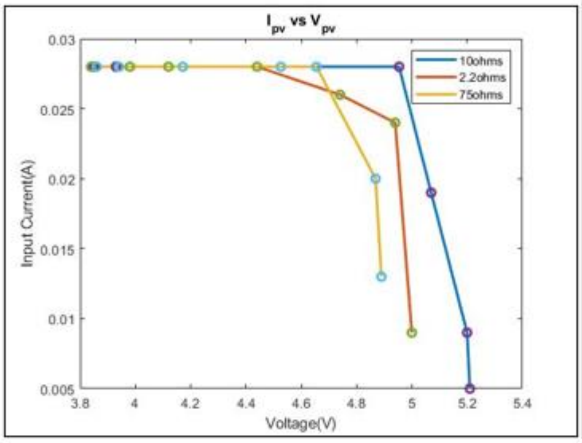
$$\text{Rover Range} = \text{SoC} * \text{Max Discharge CapacityCurrent} * 3600 * \text{speed of rover} \qquad (1)$$

This equation utilises the different speeds of the drive SMPS, the current consumed for each speed as well as the SoC of the battery pack.

6) PV panels – Characterizing

A characteristic of the solar panels is the $V_{pv} – I_{pv}$ graph measured at the solar panels. The image in Fig. 4 shows the different curves that are obtained when we vary the load at port B. Another factor that affects the graph is the irradiance which is the amount of sunlight received by the solar panels. If the irradiance increases, the current from the panel is at a higher maximum current but retains the same drop of current at the drop off voltage. The current is at a maximum for a large range of voltages and falls off for a small range.

Another important characteristic to observe is the $P_{pv} – V_{pv}$ graph where we measure the power at the solar panels as well as the voltage across it. As we increase the voltage, the power increases to a certain point then decreases drastically. We can obtain these readings by sweeping the duty cycle (V_{ref}) across the whole range to get an accurate reading and the results we obtained for 2 solar panels in parallel are shown in Figure 5-4.



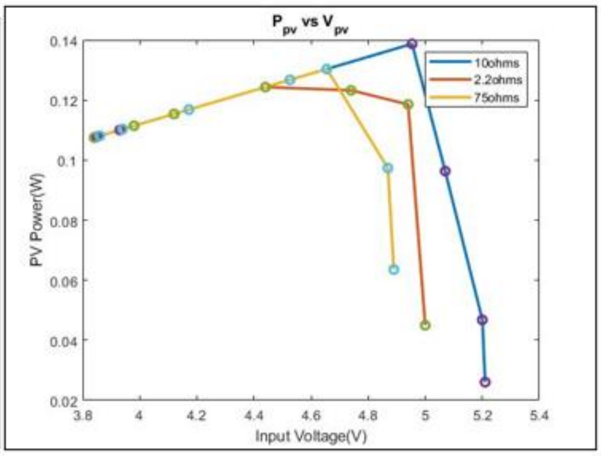


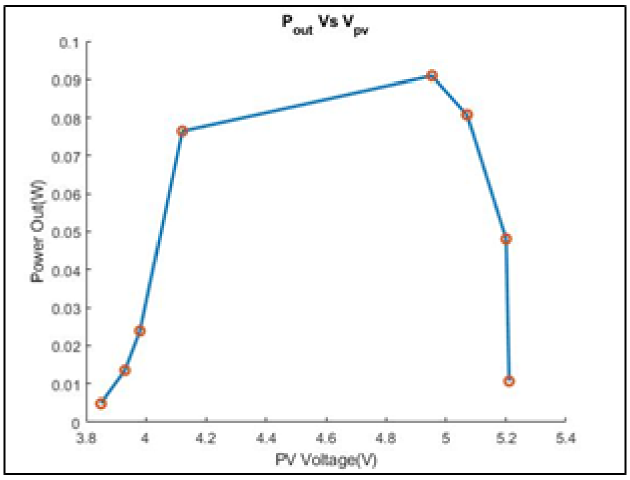
Fig. 5-4: PV panel graphs

Like the battery configuration, the two configurations that can be used to connect the solar panels are series and parallel connections. In my design we decided to connect 4 solar panels in parallel to increase the amount of current the drive SMPS can draw while retaining the same voltage. With 2 panels in parallel, we were able to obtain a maximum current of 90mA but with 4 panels, we measured a current of 140mA [12].

7) Arduino – MPPT Algorithm

To implement the MPPT algorithm, we use the power at the output instead of the solar panels since the current measuring resistor is at port B. The curve of output power against PV voltage is shown in Figure 5-5. We notice that the maximum output power of 0.09W can be obtained at around 4.9V so we want the panels to continuously operate at this range to maximize output current.

Fig. 5: Output Power vs Voltage



Maximum Power Point Tracking (MPPT) is an algorithm the PV panels use to adjust its impedance to operate at peak power. There are many factors which affect the performance of the PV panels such as solar irradiance, impedance of the load and the temperature of the surroundings. The MPPT algorithm we decided to implement is the perturbation and observe(P&O) algorithm which perturbs the initial operating voltage. We increment a small voltage interval and if the power increases, we keep the change and repeat the process. Otherwise, we decrement small voltage intervals and if the power increases, we keep the change and repeat the process. The flow chart in Figure 5-6 shows the process of how the algorithm works.[11]

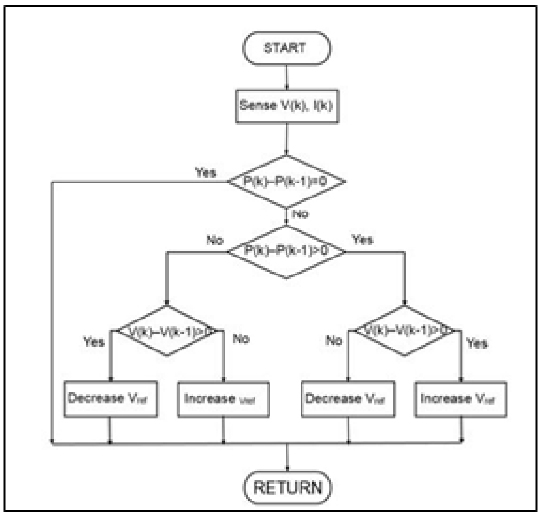


Fig. 5-6: P&O Flow Chart

However, there are also some disadvantages the P&O algorithm has shown which is that it is difficult to track the power under varying atmospheric condition [13]. It also takes some time to reach steady state and if we want to speed up the response, we must increase perturbation step size which comes at a cost of a loss of power due to inaccurate tracking [14].

8) Battery – Cell Configuration

In our design, we decided to implement 2 batteries in parallel with the goal of increasing the capacitance while retaining the same voltage. To connect it in parallel while keeping the battery safe, we must first consider the SoC for each of the cells using the SoC-OCV graph. Once we are aware of the SoC, we utilize active balancing which increases the charge of the cell with the lower SoC to the level of the other cell. Once they are approximately the same SoC, we apply passive balancing which causes current to flow from the cell with the lower voltage to the other cell [6]. Subsequently, we charge both the batteries to its maximum voltage and the advantage of the parallel configuration is that the battery balances itself when it is discharging.

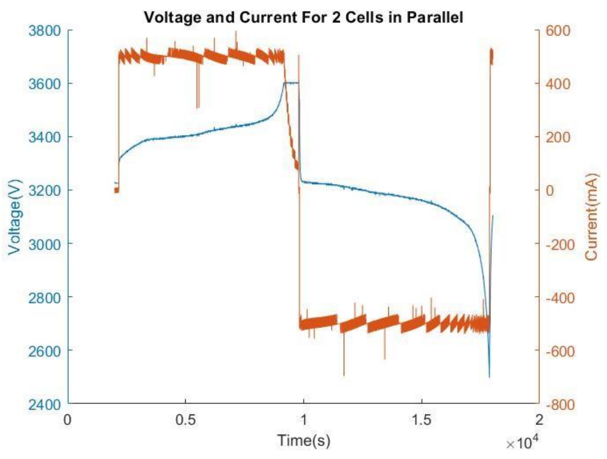


Fig. 5-7: Graph for 2 batteries in parallel

When testing the CC-CV graph for two cells in parallel, we deduce that the current to charge and discharge the cells are at 500mA instead of 250mA. In addition, the time taken to discharge the battery pack is the same as a single cell which proves an increase in capacitance since the current has doubled as seen in Fig. 5-7.

9) Final Circuit Diagram

The block diagram for the top-level design is shown in Fig. 5-8. This is a hypothetical implementation to the system where we use a charging station to charge the batteries using 1 SMPS. The other SMPS is used to discharge the batteries through the Rover. The energy SMPS will take in voltages at port A and step it down at port B to be supplied to the battery pack. The drive SMPS will take the energy supplied to the battery pack at port A and use it to power the Rover. The state machine for the energy SMPS is shown in Fig. 5-9 while the state machine for the drive SMPS is shown in Fig. 5-10.

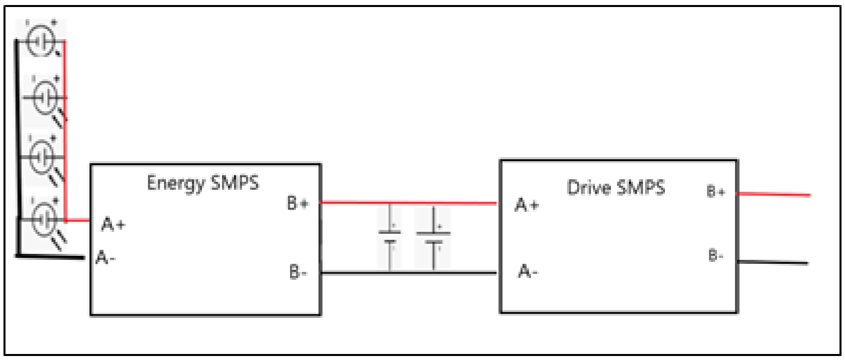
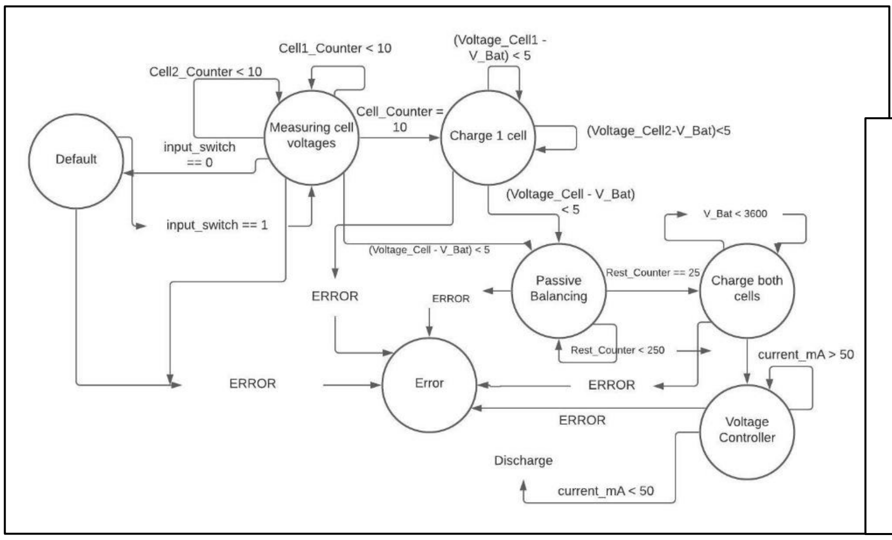


Fig. 5-8: Top Level Circuit Diagram

The detailed block diagram for the various pin arrangements is shown in the appendix F. R1 corresponds to relay for cell 1 and M1 corresponds to the measure pin for cell 1. The design implementation I have decided to implement are 4 solar panels in parallel at port A. The four panels in parallel would increase the current supplied to the SMPS so we can charge the battery at a faster rate.



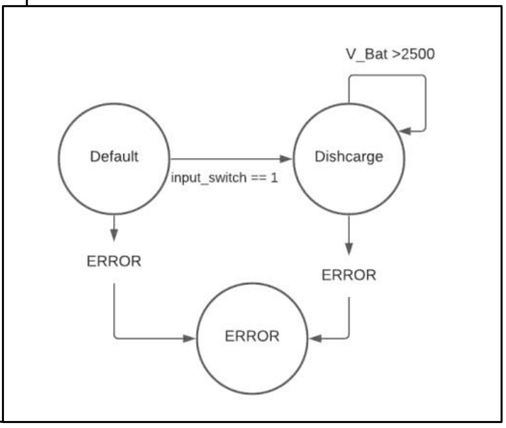


Fig. 5-9: State Machine for Energy SMPS

Fig. 5-10: State Machine for Drive SMPS

Integration

The first main step involved unit-testing the Drive subsystem. We sent simple commands to make the Rover turn by a specific number of degrees to check the X-coordinate, and to move a precise distance to check the Y-coordinate. Immediately it was clear that the sensor could take proper measurements only on certain surfaces. A flat, smooth, monotone surface without any distinctive marks allowed for good sensor value readings, and correct functionality of the Drive module. On the other hand, as shown in the diagram (Figure 6-1 of this part) when trying to run the Rover on a 'carpet like' surface, the sensor failed to perform properly. This is because the sensor never picked up the X,Y-coordinates values that correspond to the desired angle and distance which instruct the motors to stop.

Interestingly enough, even on the best surfaces, when testing rotations, we noticed that the angle by which the Rover turned was always a bit less than the desired value. This error also increased linearly with the size of the angle. Therefore, as shown in the Drive subsystem section, an offset obtained through trial and error was added which now allows the Rover to reach a desired angle with precision.

Another important aspect that impacted the Drive performance was drag. This could come again from an uneven 'carpet like' surface, but more significantly it came from some of the cables that were present for the connections.

We then proceeded to integrate with Control and Command, sending instructions to Drive from our website. As the UART and MQTT connections were successful, the only real disturbances in this part were the hotspot connection dropping or either forgetting to reset the drive sensor or needing to refresh the website. Having established a proper connection between Drive, Control, and Command we tested the three operating modes we implemented: Fixed Routing, Live Routing, Coordinate Routing. When none of the previous disturbances mentioned were too significant these tests were successful and recorded for the video demonstration. These tests were important because they allowed to check the versatility of the Rover, and also, by testing it in different conditions, we were able to identify special cases that required debugging that we had not initially considered in our algorithms. As an example, when running a Coordinate Routing test, we realised that the way we had initially implemented the arcsine in Drive, was not yet complete and could not accommodate some combinations of commands. Testing allowed to debug and correct the function.

After a positive experience in implementing UART for the Control-Drive connection, we decided to also use UART for communication between Vision and Control. After the integration of the Vision subsystem a few challenges occurred due to external environmental factors. The performance of the D8M camera is very susceptible to the lighting conditions of the environment in which it operates. For example, white surfaces tended to reflect a lot of green light, while wood often passed as orange light. This sometimes would result in an obstacle being detected even without having any ball in the scope of the camera. Therefore, when testing for example the orange, pink, and blue balls in the presence of a white wall or floor, it was useful to disable the transmission of the green colour data to Control. This way avoiding the mistake of detecting a white surface as a green ball. Overall, a non-white floor seemed to be the most ideal to minimize unwanted reflections.

A key part of Vision was being able to classify a ball as far or close by measuring how big the width appeared in the scope. This is because an object being classified as close is what triggers the obstacle avoidance algorithm. Different lighting conditions affected the precision of this measurement. Changing testing location, it was often necessary to recalibrate the threshold value in the code that defined an obstacle as close or far.

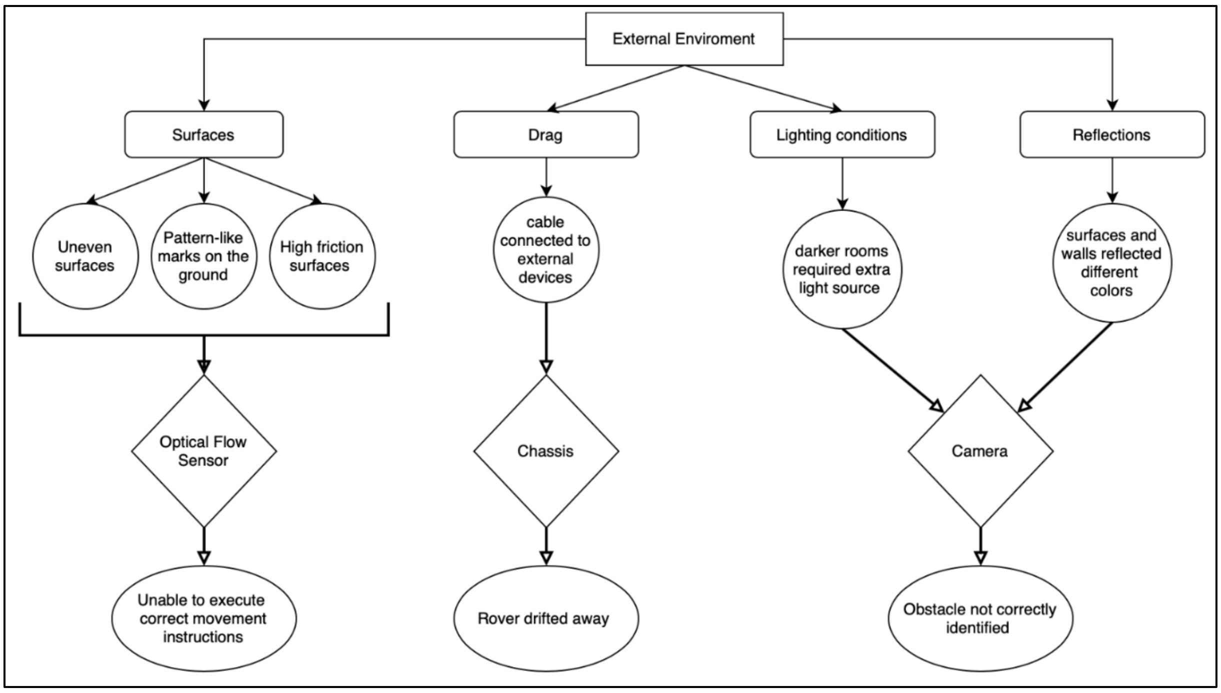

External Enviroment
Surfaces
Drag
Lighting conditions
Reflections
Uneven surfaces
Pattern-like marks on the ground
High friction surfaces
cable connected to external devices
darker rooms required extra light source
surfaces and walls reflected different colors
Optical Flow Sensor
Chassis
Camera
Unable to execute correct movement instructions
Rover drifted away
Obstacle not correctly identified

Figure 6-1: Functional diagram of the impact of the external environment

Conclusion and Improvements

Firstly, a possible improvement to the functionality of the drive submodule would be creating a PI controller in order to enhance the accuracy of the of direction control algorithm. The implemented method allows to continuously check the error between the desired and actual position of the Rover, scaled by a proportional gain of 1. However, a small error around 2mm or 1 degree is still obtained. These errors are not significant when performing initials instructions, however these small offsets are combined together as the number of instructions increases. Thus, an integral term would enable the PI controller to eliminate the steady state error which creates the offset.

Secondly, when performing the final testing, the results demonstrated that the Rover was experiencing a clear drag force on the left wheel especially when traveling long distances. An improvement would be to increase the voltage supplied to that motor in order to compensate for the drifting forces acting on the Rover.

Intellectual Property

Intellectual property is something created using one’s mind, and by properly applying for and using patent laws, unauthorized usage of your work can be stopped. There are many types of IP protection applicable to different circumstances. Copyright protection is used for original expressions such as novels, software, and many other form of art. Trademarks are used as a “badge of origin” and show where the item comes from. Registered designs are used to protect and claim ownership to a specific appearance and aesthetic of an object. Finally, patents are used to protect new inventions. Many of these forms of IP protection can be used on our rover.

Overall Rover

A registered design for the rover could be filed if we wanted to protect the look of the rover such as its two-wheel design, and front mounted camera placement. However, a design patent would not consider how these parts interact with each other and their utility to the rover.

If we wanted to sell our rover system, we can trademark a name (“Group 10” for example). This would show that all “Group 10” rovers originate from us, which is useful for quality control.

Vision

As software is not an invention, our vision algorithm cannot be covered by a patent. However, it would automatically be under the UK’s copyright protection. This protection can be invoked if it can be proven that we are the first authors of our program and can stop other people from using our code.

Drive

The software of the drive submodule subsystem only regards copyright protection. It is needed to mention that the pregiven code was given and must be cited.

Control

The software of the Control subsystem is not an invention and therefore only falls under copyright protection.

Command

The code for the website would fall under copyright protection, in addition the layout and design for the webpage could be protected as a registered design.

References

[1]N. O'Leary, "PubSubClient - Arduino Reference", Arduino.cc, 2020. [Online]. Available: https://www.arduino.cc/reference/en/libraries/pubsubclient/. [Accessed: 20- May- 2021].

[2]"queue - C++ Reference", Cplusplus.com. [Online]. Available: https://www.cplusplus.com/reference/queue/queue/. [Accessed: 02- Jun- 2021].

[3]"map - C++ Reference", Cplusplus.com. [Online]. Available: https://www.cplusplus.com/reference/map/map/. [Accessed: 07- Jun- 2021].

[4]W. Smith, "Arduino String to Char* (Example)", Coderwall, 2016. [Online]. Available: https://coderwall.com/p/zfmwsg/arduino-string-to-char. [Accessed: 08- Jun- 2021].

[5]“Image Kernels explained visually”, Explained
Visually, 2021. [Online]. Available: https://setosa.io/ev/image-kernels/. [Accessed 15 June 2021].

[6]"Series vs. Parallel Connections Explained", Relionbattery.com, 2019. [Online]. Available: https://relionbattery.com/blog/series-vs-parallel-connections-explained. [Accessed: 15- Jun- 2021].

[7]” Domain Name” - Duckdns.org. 2021. Duck DNS. [online] Available at: <https://www.duckdns.org>[Accessed 10 June 2021].

[8] “Button's appearances” - Fontawesome.com. 2021. Font Awesome. [online] Available at: <https://fontawesome.com> [Accessed 26 May 2021].

[9] “HTML/CSS tools” - Otto, M., 2021. Introduction. [online] Getbootstrap.com. Available at: <https://getbootstrap.com/docs/5.0/getting-started/introduction/>[Accessed 12 May 2021].[10] “HTML/CSS tools” - W3schools.com. 2021. HTML Tutorial. [online] Available at: <https://www.w3schools.com/html/default.asp>[Accessed 12 May 2021].

[11] "MPPT Algorithm", Uk.mathworks.com, 2021. [Online]. Available: https://uk.mathworks.com/solutions/power-electronics-control/mppt-algorithm.html. [Accessed: 15- Jun- 2021].

[12] A. Sendy, "How to wire solar panels in series vs. parallel", Solar Reviews, 2021. [Online]. Available: https://www.solarreviews.com/blog/do-you-wire-solar-panels-series-or-parallel. [Accessed: 15- Jun- 2021].

[13] S. RAJANI and V. PANDYA, Simulation and comparison of perturb and observe and incremental conductance MPPT algorithms for solar energy system connected to grid. Gandhinagar: Indian Academy of Sciences, 2015, p. 141.

[14] R. Alik, A. Jusoh and T. Sutikno, A Review on Perturb and Observe Maximum Power Point Tracking in Photovoltaic System. Johor, 2015, p. 747.

[15] L. Miller, "Arduino Data Logger (CSV) with Sensors and Python", Learn Robotics, 2021. [Online]. Available: https://www.learnrobotics.org/blog/arduino-data-logger-csv/?utm_source=youtube&utm_medium=description&utm_campaign=arduino_CSV_data_logger. [Accessed: 10- Jun- 2021].

[16] R. Cottis, "Coulomb Counting - an overview | ScienceDirect Topics", Sciencedirect.com, 2008. [Online]. Available: https://www.sciencedirect.com/topics/engineering/coulomb-counting. [Accessed: 22- May- 2021].

[17]"Serial - Arduino Reference", Arduino.cc, 2021. [Online]. Available: https://www.arduino.cc/reference/en/language/functions/communication/serial/. [Accessed: 15- Jun- 2021]

[18] ADNS-3080 Data Sheet. Avago Technologies, 2008, pp. 2-12.

[19] TB6612FNG Data Sheet. TOSHIBA Corporation, 2014, pp. 2-5.

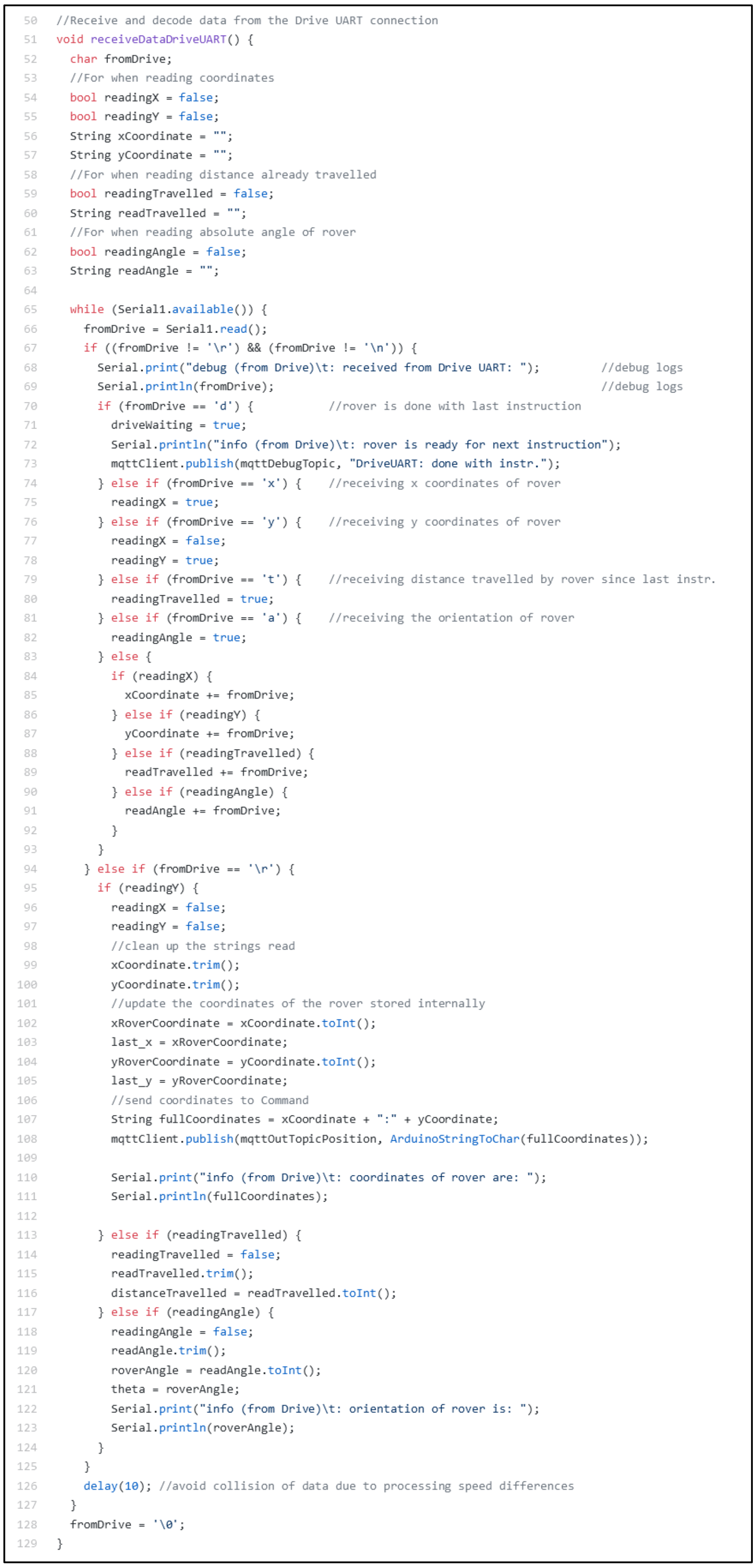
[20] Y. Zhu, sketch_motors_with_smps. Imperial College London, 2020.

[21] E. Stafl, Calculating the State of Charge of a Lithium Ion Battery System using a Battery Management System. 2020.

[22] "Battery State of Health Determination", Mpoweruk.com, 2021. [Online]. Available: https://www.mpoweruk.com/soh.htm. [Accessed: 06- Jun- 2021].

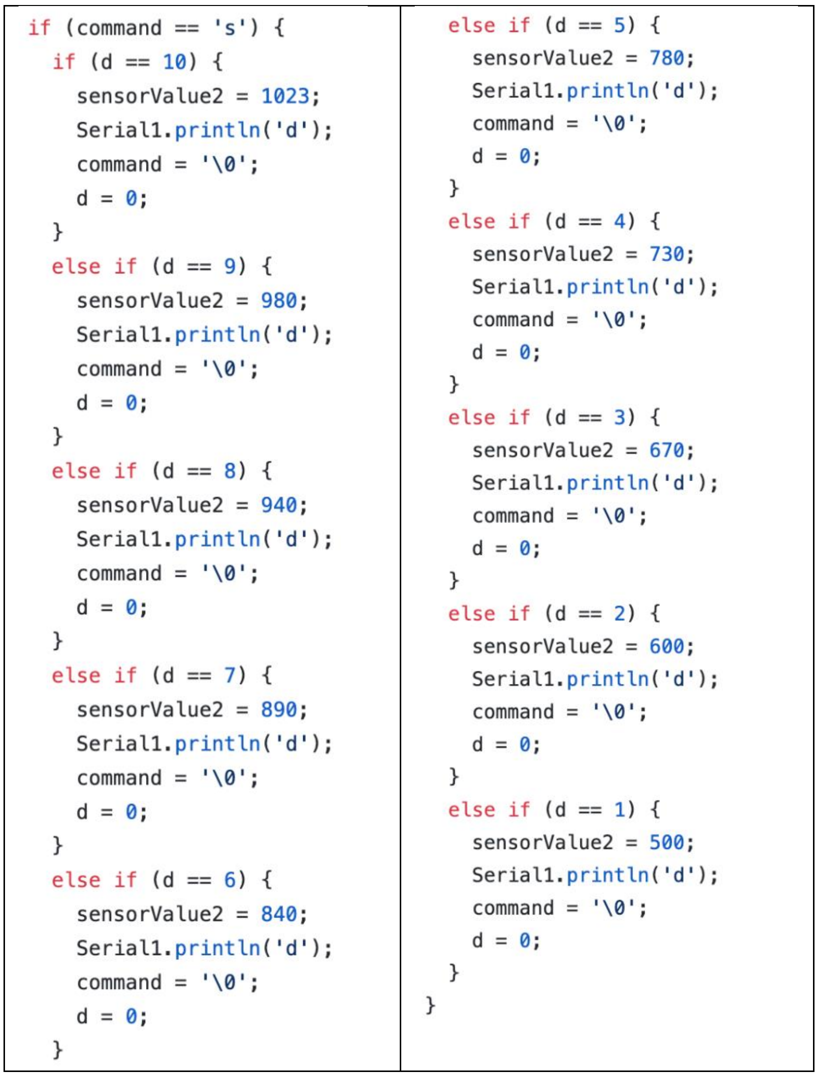
Appendices

Appendix A - Full implementation of the function receiveDataDriveUART()



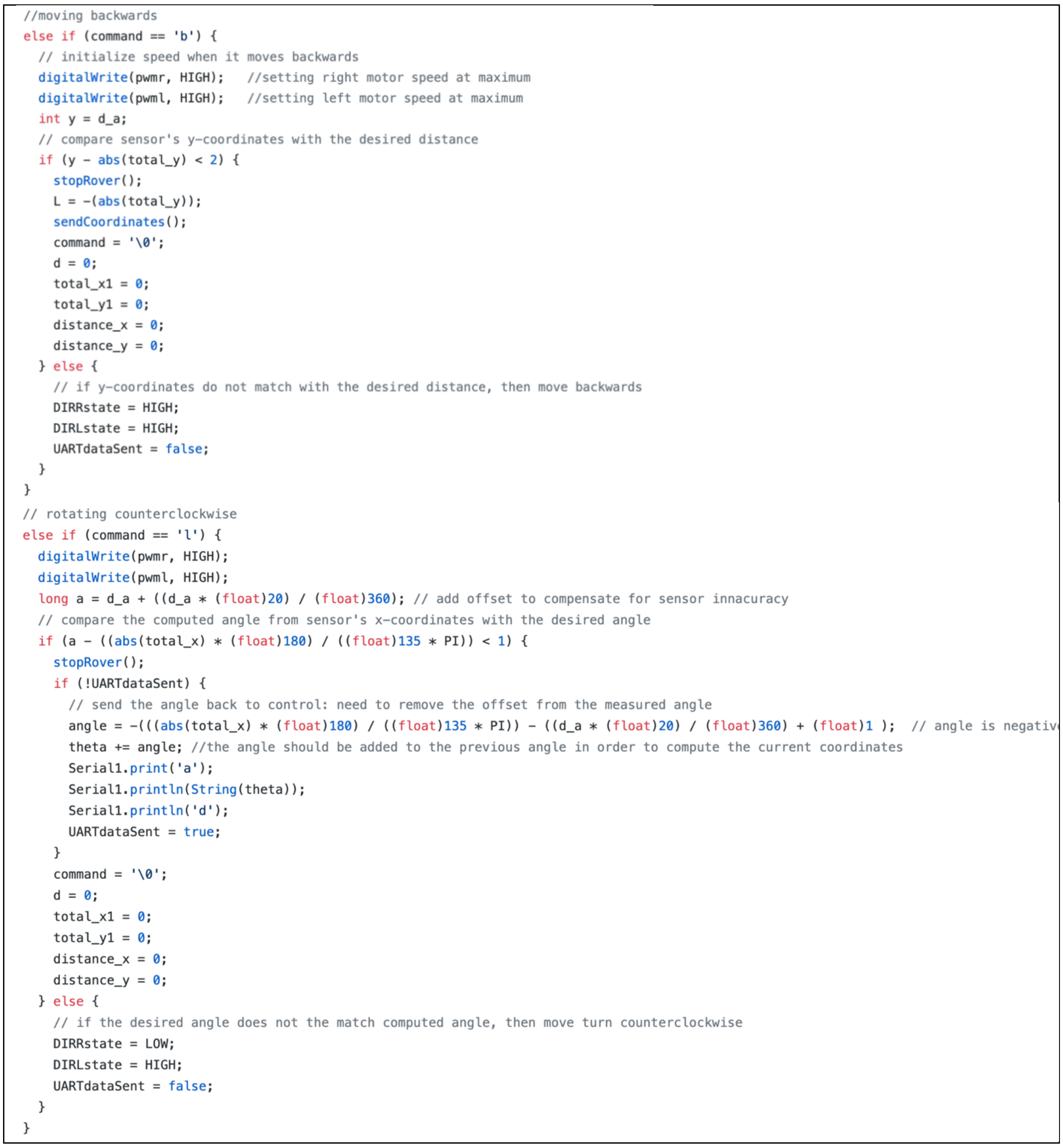
```
//Receive and decode data from the Drive UART connection
void receiveDataDriveUART() {
  char fromDrive;
  //For when reading coordinates
  bool readingX = false;
  bool readingY = false;
  String xCoordinate = "";
  String yCoordinate = "";
  //For when reading distance already travelled
  bool readingTravelled = false;
  String readTravelled = "";
  //For when reading absolute angle of rover
  bool readingAngle = false;
  String readAngle = "";

  while (Serial1.available()) {
    fromDrive = Serial1.read();
    if ((fromDrive != '\r') && (fromDrive != '\n')) {
      Serial.print("debug (from Drive)\t: received from Drive UART: ");         //debug logs
      Serial.println(fromDrive);                                                 //debug logs
      if (fromDrive == 'd') {           //rover is done with last instruction
        driveWaiting = true;
        Serial.println("info (from Drive)\t: rover is ready for next instruction");
        mqttClient.publish(mqttDebugTopic, "DriveUART: done with instr.");
      } else if (fromDrive == 'x') {    //receiving x coordinates of rover
        readingX = true;
      } else if (fromDrive == 'y') {    //receiving y coordinates of rover
        readingX = false;
        readingY = true;
      } else if (fromDrive == 't') {    //receiving distance travelled by rover since last instr.
        readingTravelled = true;
      } else if (fromDrive == 'a') {    //receiving the orientation of rover
        readingAngle = true;
      } else {
        if (readingX) {
          xCoordinate += fromDrive;
        } else if (readingY) {
          yCoordinate += fromDrive;
        } else if (readingTravelled) {
          readTravelled += fromDrive;
        } else if (readingAngle) {
          readAngle += fromDrive;
        }
      }
    } else if (fromDrive == '\r') {
      if (readingY) {
        readingX = false;
        readingY = false;
        //clean up the strings read
        xCoordinate.trim();
        yCoordinate.trim();
        //update the coordinates of the rover stored internally
        xRoverCoordinate = xCoordinate.toInt();
        last_x = xRoverCoordinate;
        yRoverCoordinate = yCoordinate.toInt();
        last_y = yRoverCoordinate;
        //send coordinates to Command
        String fullCoordinates = xCoordinate + ":" + yCoordinate;
        mqttClient.publish(mqttOutTopicPosition, ArduinoStringToChar(fullCoordinates));

        Serial.print("info (from Drive)\t: coordinates of rover are: ");
        Serial.println(fullCoordinates);

      } else if (readingTravelled) {
        readingTravelled = false;
        readTravelled.trim();
        distanceTravelled = readTravelled.toInt();
      } else if (readingAngle) {
        readingAngle = false;
        readAngle.trim();
        roverAngle = readAngle.toInt();
        theta = roverAngle;
        Serial.print("info (from Drive)\t: orientation of rover is: ");
        Serial.println(roverAngle);
      }
    }
    delay(10); //avoid collision of data due to processing speed differences
  }
  fromDrive = '\0';
}
```

Appendix B - Full speed control if-statements



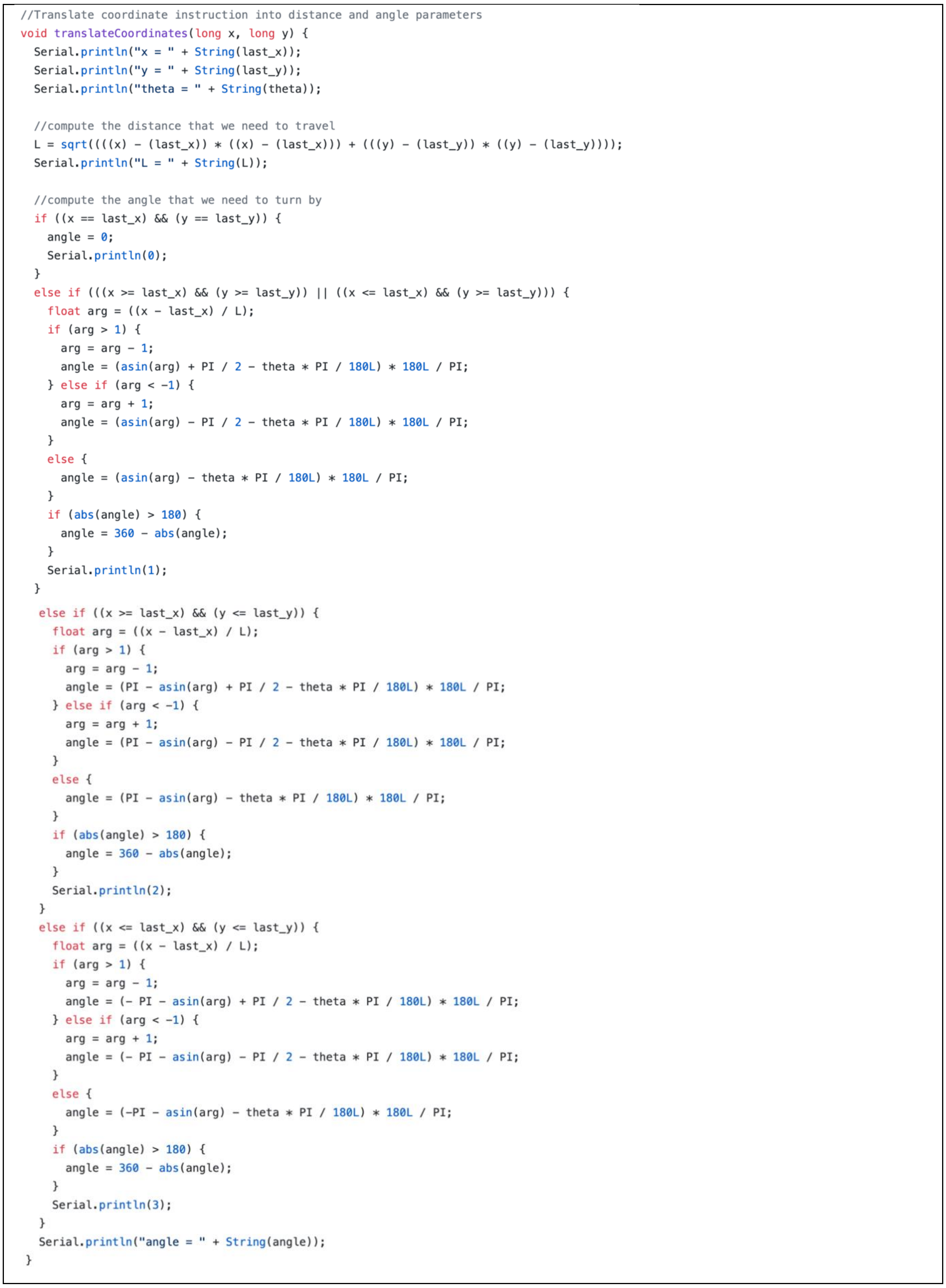
```
if (command == 's') {
  if (d == 10) {
    sensorValue2 = 1023;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
  else if (d == 9) {
    sensorValue2 = 980;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
  else if (d == 8) {
    sensorValue2 = 940;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
  else if (d == 7) {
    sensorValue2 = 890;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
  else if (d == 6) {
    sensorValue2 = 840;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
  else if (d == 5) {
    sensorValue2 = 780;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
  else if (d == 4) {
    sensorValue2 = 730;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
  else if (d == 3) {
    sensorValue2 = 670;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
  else if (d == 2) {
    sensorValue2 = 600;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
  else if (d == 1) {
    sensorValue2 = 500;
    Serial1.println('d');
    command = '\0';
    d = 0;
  }
}
```

Appendix C - Backwards and clockwise direction control implementation



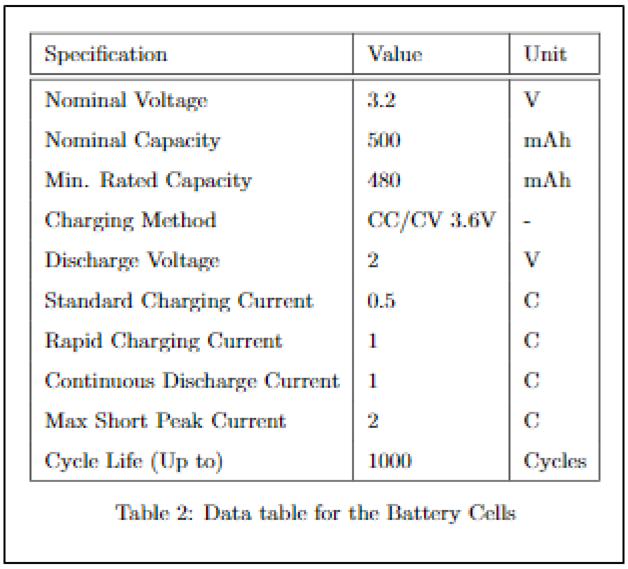
```
//moving backwards
else if (command == 'b') {
  // initialize speed when it moves backwards
  digitalWrite(pwmr, HIGH);   //setting right motor speed at maximum
  digitalWrite(pwml, HIGH);   //setting left motor speed at maximum
  int y = d_a;
  // compare sensor's y-coordinates with the desired distance
  if (y - abs(total_y) < 2) {
    stopRover();
    L = -(abs(total_y));
    sendCoordinates();
    command = '\0';
    d = 0;
    total_x1 = 0;
    total_y1 = 0;
    distance_x = 0;
    distance_y = 0;
  } else {
    // if y-coordinates do not match with the desired distance, then move backwards
    DIRRstate = HIGH;
    DIRLstate = HIGH;
    UARTdataSent = false;
  }
}
// rotating counterclockwise
else if (command == 'l') {
  digitalWrite(pwmr, HIGH);
  digitalWrite(pwml, HIGH);
  long a = d_a + ((d_a * (float)20) / (float)360); // add offset to compensate for sensor innacuracy
  // compare the computed angle from sensor's x-coordinates with the desired angle
  if (a - ((abs(total_x) * (float)180) / ((float)135 * PI)) < 1) {
    stopRover();
    if (!UARTdataSent) {
      // send the angle back to control: need to remove the offset from the measured angle
      angle = -(((abs(total_x) * (float)180) / ((float)135 * PI)) - ((d_a * (float)20) / (float)360) + (float)1 );  // angle is negativ
      theta += angle; //the angle should be added to the previous angle in order to compute the current coordinates
      Serial1.print('a');
      Serial1.println(String(theta));
      Serial1.println('d');
      UARTdataSent = true;
    }
    command = '\0';
    d = 0;
    total_x1 = 0;
    total_y1 = 0;
    distance_x = 0;
    distance_y = 0;
  } else {
    // if the desired angle does not the match computed angle, then move turn counterclockwise
    DIRRstate = LOW;
    DIRLstate = HIGH;
    UARTdataSent = false;
  }
}
```

Appendix D - Translation from coordinates to angle and distance



```
//Translate coordinate instruction into distance and angle parameters
void translateCoordinates(long x, long y) {
  Serial.println("x = " + String(last_x));
  Serial.println("y = " + String(last_y));
  Serial.println("theta = " + String(theta));

  //compute the distance that we need to travel
  L = sqrt((((x) - (last_x)) * ((x) - (last_x))) + (((y) - (last_y)) * ((y) - (last_y))));
  Serial.println("L = " + String(L));

  //compute the angle that we need to turn by
  if ((x == last_x) && (y == last_y)) {
    angle = 0;
    Serial.println(0);
  }
  else if (((x >= last_x) && (y >= last_y)) || ((x <= last_x) && (y >= last_y))) {
    float arg = ((x - last_x) / L);
    if (arg > 1) {
      arg = arg - 1;
      angle = (asin(arg) + PI / 2 - theta * PI / 180L) * 180L / PI;
    } else if (arg < -1) {
      arg = arg + 1;
      angle = (asin(arg) - PI / 2 - theta * PI / 180L) * 180L / PI;
    }
    else {
      angle = (asin(arg) - theta * PI / 180L) * 180L / PI;
    }
    if (abs(angle) > 180) {
      angle = 360 - abs(angle);
    }
    Serial.println(1);
  }
  else if ((x >= last_x) && (y <= last_y)) {
    float arg = ((x - last_x) / L);
    if (arg > 1) {
      arg = arg - 1;
      angle = (PI - asin(arg) + PI / 2 - theta * PI / 180L) * 180L / PI;
    } else if (arg < -1) {
      arg = arg + 1;
      angle = (PI - asin(arg) - PI / 2 - theta * PI / 180L) * 180L / PI;
    }
    else {
      angle = (PI - asin(arg) - theta * PI / 180L) * 180L / PI;
    }
    if (abs(angle) > 180) {
      angle = 360 - abs(angle);
    }
    Serial.println(2);
  }
  else if ((x <= last_x) && (y <= last_y)) {
    float arg = ((x - last_x) / L);
    if (arg > 1) {
      arg = arg - 1;
      angle = (- PI - asin(arg) + PI / 2 - theta * PI / 180L) * 180L / PI;
    } else if (arg < -1) {
      arg = arg + 1;
      angle = (- PI - asin(arg) - PI / 2 - theta * PI / 180L) * 180L / PI;
    }
    else {
      angle = (-PI - asin(arg) - theta * PI / 180L) * 180L / PI;
    }
    if (abs(angle) > 180) {
      angle = 360 - abs(angle);
    }
    Serial.println(3);
  }
  Serial.println("angle = " + String(angle));
}
```

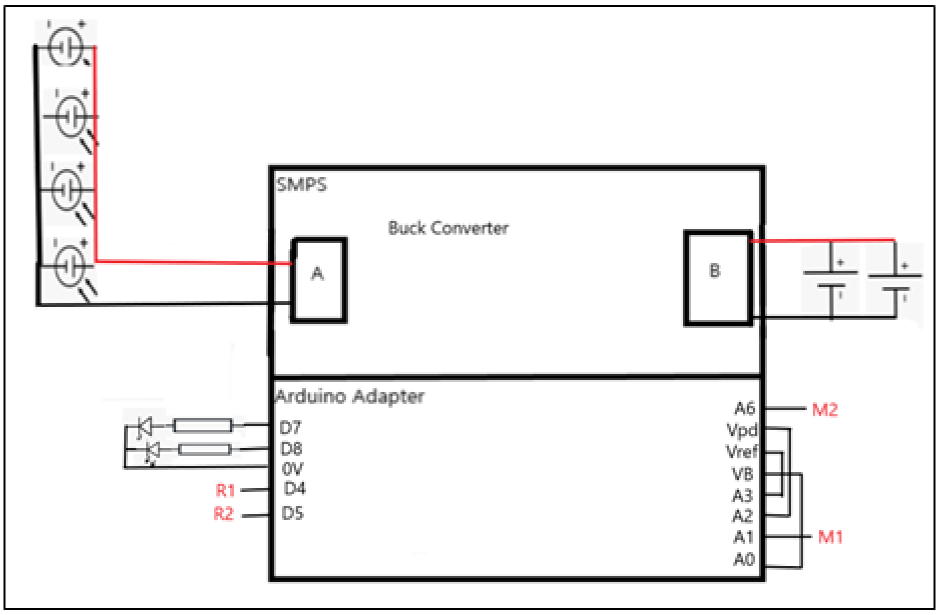
Appendix E - Battery datasheet



Specification	Value	Unit
Nominal Voltage	3.2	V
Nominal Capacity	500	mAh
Min. Rated Capacity	480	mAh
Charging Method	CC/CV 3.6V	-
Discharge Voltage	2	V
Standard Charging Current	0.5	C
Rapid Charging Current	1	C
Continuous Discharge Current	1	C
Max Short Peak Current	2	C
Cycle Life (Up to)	1000	Cycles

Table 2: Data table for the Battery Cells

Appendix F - Detailed circuit diagram of Energy SMPS


SMPS
Buck Converter
A
B
Arduino Adapter
D7
D8
0V
D4
D5
R1
R2
A6
Vpd
Vref
VB
A3
A2
A1
A0
M2
M1