The ARLISS Project
Design Document
CS 461

Steven Silvers, Paul Minner, Zhaolong Wu, Zachary DeVita
Capstone Group 27, Fall 2016

Abstract

This document defines the plan for how the various pieces of the ARLISS Project will be developed and implemented. Each piece that was discussed in the previous technology review will be gone through in detail in its own section.

CONTENTS

I. Introduction

In this document we have divided our project into twelve components, and a framework for each one of these components has been designed by the corresponding member from the list below. Each section will detail how a component will operate, and how it will interact with other related components. This document is intended to provide a structure, and the architectural layout for the entire project.

Project Framework
Zachary DeVita
CMOS Image Sensing
Zachary DeVita
Ultrasonic Radar Sensor
Zachary DeVita
Obstacle Avoidance
Steven Silvers
Control Board
Steven Silvers
Motorized Tracks
Steven Silvers
Parachute Deployment
Paul Minner
Getting Unstuck from Obstacles
Paul Minner
Finding and Touching the Finish Pole
Paul Minner
Navigation System and Algorithm(s)
Zhaolong Wu
Getting Rover Unstuck When It on Its Side
Zhaolong Wu
Payload Fairing
Zhaolong Wu

II. Framework

For our overall project design and framework, our team has decided to utilize the C++ programming language. We will be writing our individual portions of code using a C compiler in order to reduce the overall overhead and energy cost of the program, but we will eventually combine the portions into a C++ framework where we will be able to utilize classes and access modifiers. The team feels strongly that utilizing classes and access modifiers for a project of this magnitude would be a huge benefit, and, since C is a subset of C++, we can get the best of both programming languages. Using C++ compilers and C++ specific tools will also reduce time spent debugging and testing [**?**].

As far as the framework for the project, we will need a series of classes to pass control of the satellite between. This is due to the fact that our satellite's journey will be comprised of four primary stages where the controls for the satellite are vastly different from each other. All of these classes governing stages of control should have a private modifier so they can't be accessed by other classes.

The first stage of the satellite's expedition will be when it is encapsulated in its can-shaped housing. This stage will take place during the period of time that the rocket is on the ground prior to launch, and it will last until some period of time after the satellite has landed safely to the ground via the parachute. Though the satellite will remain in a static state during this period, it must be turned on and ready to receive instructions.

There will be a class designated for the control of the satellite for this period of time. The satellite will simply be listening for interrupts from other classes. There will be an interrupt that tells this class when to deploy its parachute. This interrupt will be provided by another class, and it will be based on some form of a timer. This event should occur shortly after the satellite has been ejected from the rocket. Another interrupt will tell the satellite to pass control to the class responsible for the next control stage of the expedition. Both of these classes with the interrupts should have a protected modifier so they are only visible to the class which has control of the satellite. This event should occur shortly after the satellite has been ejected from the rocket. This event should occur shortly after the satellite has landed safely on Earth's surface.

The second stage of the satellite's expedition will encompass the majority of the satellite's expedition, and there will be a class written specifically for the segment. This class will be responsible for the period of time that the rocket navigates from its landing zone until it has reached approximately 8-9 meters from the pole. The pole marks the final destination of the satellite, and the satellite must physically come in contact with the pole to finish the competition. The reason for the 8-9 meters is due to the fact that GPS is only accurate at 7.8 meters with a 95% confidence interval [**?**].

During this period there will be many other classes interacting with the class retaining control of the satellite. There should be a class which tells this control class if there is obstacles in the way and what direction to move to avoid them. That class should be a protected class as well, and that class should have a parent/child relationship with each class governing the sensors being used to identify obstacles.

There should also be a class which governs the GPS sensor. This class should help direct the satellite on a macro level, not taking into consideration any obstacles at the local level. This protected class should have its navigation instructions be overridden in the case of an obstacle in its path. This class will also send an interrupt to the control class when the satellite has reached 8-9 meters of the pole which will tell the control class to pass off control to the final governing the satellite's navigation.

The third stage of control will be reached only in the case where the satellite becomes stuck or on its side. The satellite design is being made specifically to preventing this type of situation from occurring, but there is still a chance of it happening. There will be a specific routine devised for this circumstance, and control of the satellite will be passed to this stage in this type of event. This class will have access to the CMOD, ultrasonic sensors via inheriting from these protected classes. When the satellite has recovered from the event, and some routine to avert the obstacle has been completed, then the control will be returned to the previous control stage.

The final stage of control will cover the period of time when the satellite reaches the 8-9 meter range from the pole, and it will last until contact with the pole has been made. The class responsible for the satellite's control over this period will no longer take input from the GPS. The system used previously for the obstacle avoidance system, i.e. the CMOS imaging sensor and the ultrasonic sensor, will be used here for navigation. Even if the ultrasonic sensor is operating at 40kHz, considering the target has a spherical shape, the sensor will be only accurate for up to 20.2 feet so the CMOS imaging sensor will have to suffice until the satellite is close to the pole [**?**]. The CMOS imaging sensor will need to switch modes in this stage to look for a pole shaped object and direct the satellite towards it. A separate protected class should be governing the sensor in this stage of navigation. Instead of looking for obstacles the sensor should be searching for an object which is similar to some stored image, allowing for an appropriate level of error.

III. CMOS Image Sensing

For the ARLISS project we are using a combination of a CMOS imaging sensor and at least one ultrasonic sensor in order to detect and avoid obstacles. The path of the satellite, on a macro level, is determined by the GPS sensor, but modifications to the path will need to be set locally using this system.

The idea behind the imaging sensor is that, when there is an abrupt change in color in an image, especially when referring to nature, it often shows that there is an abrupt change in the depth being viewed. For our purposes we want to use this observation to detect objects in the satellite's path, as well as, changes in terrain like a steep incline or a steep decline.

Manipulating raw camera frames can require a relatively large amount of space, as well as consume a large amount of energy. Energy is the most valuable resource for our project because we have a strict limit to the size and weight of the satellite so we have a very limited space for a battery to fit. There are several methods which we must use to minimize this cost, and there is one critical library in particular which we will use to help conserve this limited battery power.

As far as libraries go, we will be utilizing the cv.h library provided by OpenCV. This open-source library provides the capability of image and video I/O, image processing for individual video frames, as well as, built-in object recognition functionality [**?**]. All foreseeable video manipulating should be able to be done with this single library.

The plan for reducing energy consumption is simple; first we will take a frame of raw footage as input from the CMOS device, then we will convert it to grayscale, and from grayscale we can convert it to binary data. Manipulating this binary data will consume far less energy than performing complex algorithms on the original, raw video footage [**?**].

Using the OpenCv library we will be able to capture video and directly load each video frame from the sensor to an IplImage object. OpenCv also has a built-in function called cvCvtColor() which will convert each image to grayscale. There will then need to be an algorithm which will create a digital map of the image using a two-dimensional array, and which will assign values of 1 or 0 depending on the difference in color change between pixels [5]. When objects or abrupt changes in elevation are detected we can combine data from the ultrasonic sensor to determine the distance of the object, and modify the satellite's route to avoid the obstacle.

This functionality of importing the live video feed, converting it to frames, and applying functions to the data to convert it to binary will need to exist in its own class. This class will be responsible for the flow of the input video data from the sensor, and outputting the data as an array of binary values to a separate class. The output from this class will combined with the output from the class responsible for the ultrasonic sensor data, and this combination of data will be used elsewhere to help determine what sequence of events needs to occur in order to safely navigate the terrain.

IV. Ultrasonic Radar Sensor

For the obstacle detection system, data will be created using a complex algorithm which combines data from the CMOS imaging sensor as well as at least one ultrasonic sensor. As mentioned previously, the path of the satellite will be determined based on the terrain encountered and any obstacles detected in its path. The system will switch from the satellite's GPS sensor to the obstacle avoidance system when an obstacle is sensed, and then will temporarily divert from the direct path in order to drive navigate around it. The GPS will be used to determine the path on a macro level, but it can be modified locally using this system.

The reason for the combination of the CMOS sensor and the ultrasonic sensor is simple. Though a camera sensor, like the CMOS, is the absolute best at interpreting texture, shape, and is the natural choice for scene interpretation based on its accuracy, it has no way to tell the distance of an object [**?**]. Without a distance measuring sensor the satellite would not know the difference between an obstacle directly in front of it, and the horizon which is miles away.

The hardware being used will be a ping-response ultrasonic rangefinder. These sensors have two transducers, a speaker which sends out the burst of ultrasonic sound, and a microphone which listens for the sound as it is reflected off objects in its path. This hardware has two connections to the microcontroller, one for the echo pulse output and the other for the trigger pulse input which must be connected to digital I/O ports [**?**].

There will be one primary library for setting up the class for the ultrasonic radar sensor. The library NewPing.h already has built-in functions to read in input from the sensor. This library will allow us to specify the input pins where the sensor will be connected to the microcontroller, as well as set up a maximum distance of interest using a NewPing object. There is also a timer for the ping which allows us to set the frequency that each ping is sent out in milliseconds. For each ping that is sent out we must have a function which listens for the echo, and converts the time between the ping and echo into a distance [**?**].

There will be many distances recorded for each ping so they must be stored in a two-dimensional array which will be used as a map of the terrain. When a second echo is recorded then the two can be compared for accuracy. If the object in question does not show up in subsequent readings then the algorithm is reset. If the object continues to show up in subsequent readings, than this map of the terrain will be output to another class which will combine the map created by the CMOS sensor, and will use the data to determine if the satellite needs to divert from its current path.

Ultrasonic sensors provide not only accurate distance measurements, but they can also utilize this functionality to provide a method of determining angles of objects, or, more importantly, the slope of the ground. This will prove to be important when calculating the slope of the ground that is in the satellite's path. The ground should always show up in the readings. If, for instance, the ground was not being detected, it would mean that the robot was positioned in front of a steep decline. There will need to be some sort of range of acceptable slopes, and these constraints will need to be determined through testing.

V. Continuous Tracks

For the method of how we would travel along the ground, the team members of the ARLISS Project decided to go with the continuous track, or tank tread, method of traveling. The implementation of these tracks will be carried out by the Mechanical Engineering sub-team of the ARLISS Project.

Continuous tracks are widely popular for their ability to traverse rough terrain much better than traditional wheels, however they sacrifice speed in doing so. Our continuous track system will require at least four separate motors, front left, front right, rear left and rear right. Four motors are required by this design because using only two motors leads to a couple different problems, mainly that either the satellite will not be able to turn or its tracks will be significantly underpowered.

Tracks of various styles and sizes will be tested to see which one best suits our needs. We will do our best to simulate the texture of the Black Rock Desert in Nevada where the competition will be held to get the best and most useful test results possible. Tracks will be compared based on ease of traveling across flat ground, how the tracks handle changes in elevation and if there is any noticeable effect on power consumption between tracks. We will also look at various stress tests for the different tracks, such as how easy are they to break and how easy it is to get the track to fall off the satellite. These tests will help us ensure that the satellite will perform without fail at the competition.These tests will range from being as simple as trying to pry the tracks off of the satellite with our hands to advanced as setting up a course of very rough terrain that will try to either break or remove the track. determining the proper length of tread on the track is very important for the satellite. if the tread is too long it won't travel along the hard, flat desert ground as efficiently as it could be. Make the tread too short, and the satellite might not be able to climb out of tire ruts or other divots in its path.

While the satellite is traveling on the rocket and during its journey back down to the ground, it will need to be packed away in a container the size of a standard soda can. In order to achieve this, the continuous track system will need to be able to compact itself in order to conserve space used inside of the of the soda can container. the way that this will be accomplished is by having small servo motors than can contract the Continuous track motor systems and tread into the confined space. Once the satellite has landed the servo motors will then expand the motors and the treads to their standard position for the expedition across the desert. This will be controlled by the software class that handles getting the satellite out of the soda can container, as described in the Framework section of this document.

VI. Control Board

The control board acts as the physical central hub of our system, connecting together and managing the various I/O devices in our satellite system. The control board will be custom built by the Electrical and Computer Engineering sub-team of the ARLISS Project. Having a custom board built in house benefits us in a couple ways, it can make for the most efficient use of limited space and doesn't waste power or weight by not including I/O devices that we will not be using for our project, which would not be true if we used a pre-made third party board.

The current requirements for this control board include being able to handle I/O for our CMOS imaging sensor, sending it messages for when it needs to capture an image and the be able to move that data back for processing. The control board also needs to be able to support motor control functionality, individually telling our motors how much power to use and whether to turn forward or backward. The control board will also need to be able to interface with our parachute module, telling it when to deploy and hearing back from the module if it deployed properly. Lastly the control board will need to be able to interface with our on board GPS sensor. This is essential because the GPS sensor will handle deploying the parachute module as well as be the chief tool of navigation while on the ground trying to reach the final goal.

Our custom built control board will make use of the Atmel ATxmega128A4U chip. This chip is a low power eight bit AVR microcontroller featuring one hundred twenty eight kilobytes flash memory, with thirty four I/O pins and support for thirty four external interrupts[**?**]. The selling point of this chip is it has low power consumption, has enough I/O pins to suite our needs while also not going overboard. Finally, this chip is part of a family of chips, the mega128 family, that many people in our ARLISS project team are already familiar with as a similar chip from this family was used in the Electrical Engineering course Computer Organization and Assembly Language.

The detailed design and implementation is being handled by the Electrical Engineering team of the ARLISS Project. If designing and building our own board falls through and is no longer an option, the backup plan is to use the Raspberry PI Zero as our control board. The PI Zero will meet our basic needs for this project, but will still have some of the extra I/O devices and ports mentioned earlier that will not be needed for the ARLISS Project.

VII. Obstacle Avoidance

As detailed in the CMOS Image Sensing section, the data that we will be working with to detect and avoid obstacles will be a 2D array of binary values. A value of '1' will represent a change in the gray scale image, which will ultimately outline the terrain and give an easy to traverse dataset with which to avoid obstacles with. A value of '0' will represent no major change in shade of gray between neighboring pixels of the image, meaning that this will either be the interior of an object bounded by '1' values, or empty space in front of the camera.

The biggest concern and challenge facing our satellite project is the conservation of power and limiting power usage. Due to the space constraints of the competition our batteries will be smaller than desired for this kind of journey so we must conserve where we can. One way that we are doing this within the obstacle avoidance system is by not continuously taking in video feed from the camera. The setting where our system will be tested is for the most part very flat, meaning we can realistically take a new image to analyze after a short time interval without having to worry too much about a new obstacle appearing before we can avoid it. The exact time interval is not yet known, as it will take physically testing the satellite system to determine the correct value to use.

The way that the 2D array will be used to avoid obstacles is by examining the values by column from left to right to detect if an obstacle has appeared in our path. We expect there to almost always be a row of solid '1' values along the bottom of the 2D array that represents the horizon, where the ground meets the sky in the cameras eye. The algorithm for avoiding an object is quite simple, if a column or close to a column of '1' values appear towards the right side of the array, we stop forward motion, and rotate counterclockwise while periodically stopping to take a new picture to see if we have angled around the obstacle. once we have a picture showing no obstacles, we travel forward in this direction for approximately five seconds before going back to driving towards the target GPS coordinates. If the obstacle appears on the left side of the 2D array, we do the same thing except rotate clockwise.

Once we have the satellite built and the CMOS imaging implemented with the gray scale to 2D array conversion working, we will be able to test our satellite in various conditions and with different obstacles to see what kinds of obstacles we can drive over and ignore, and what obstacles must be driven around. During this testing period we will also tweak the timing at which how often a new picture will be taken and analyzed. The target for this testing is to find the time cycle at which we can take the fewest pictures while still avoiding all of the obstacles that we need to, and thus saving battery usage by the camera.

VIII. Parachute Deployment

For the parachute deployment protocol, we have decided to use GPS to determine when to deploy the parachute. This method is perfect for us because we already need a GPS unit for other uses, and the GPS unit will be more accurate than just deploying the parachute based on a timer, and take less time in the air than deploying the parachute immediately.

The parachute deployment system will be the first part of our software to run, so we will start the system once the satellite separates from the rocket. This will be accomplished by tripping a mechanical switch which sends a signal to the satellite. Once started, the system will take periodic altitude measurements every second until the altitude drops below 1,000 feet. Once below 1,000 feet for two measurements in a row, the software will send a signal to deploy the parachute. We chose 1,000 feet as the desired height because the average reserve parachute for a skydiver deploys in less than 400 feet [**?**]. Since our satellite only weights as much as a can of soda, 400 feet is a conservative estimate of the distance the parachute will take to deploy. This should ensure we dont waste too much time in the air, but also gives us a large margin of error so we dont accidentally hit the ground before our parachute deploys. The worst case accuracy for civilian GPS coordinates is 25 meters (75 feet) [**?**]. This accuracy is much worse than the navigation specification because of the high speed the satellite is traveling. In addition, altitude error is actually about 1.5 times the amount of horizontal error, which can be a

significant amount. In addition, if the GPS doesnt have an unobstructed view of the sky, the data cannot be trusted at all [**?**]. It is highly unlikely that our view will be obstructed, but it is possible a plane could fly overhead, or the rocket could briefly block our view of the sky. This is the reason we take two measurements before deploying the parachute. All of these factors effect the accuracy of our parachute deployment height, which is why we have been conservative. An extra 600 feet of distance to deploy the parachute is plenty of space, but still minimizes our time spent in the air. Once the software sends the signal to deploy the parachute, the parachute deployment system will end, and a new system will begin to check when the satellite is on the ground.

The system will be implemented within its own class which is responsible for deploying the parachute. An interrupt will be sent to the first stage class once the satellite is ejected from the rocket. Then, the parachute deployment class will launch its own function, which will loop continuously, checking the altitude from the GPS until the value of the altitude is less than 1,000 feet. Once this happens, the loop will execute one more time to check if the altitude is still below 1,000 feet. If it is, end the loop, send a signal to the first stage class to deploy the parachute, and hand control back to the first stage class. If the altitude isnt still below 1,000 feet, start the process over again. The first stage class will be responsible for actually deploying the parachute.

IX. Getting Unstuck From Obstacles

The system for getting unstuck from obstacles has two components. First, determining when the satellite is stuck, and second, getting the satellite unstuck. To determine when the satellite is stuck, the satellites speed will be monitored, and if the speed drops below a certain threshold for a specific amount of time, the system to get the satellite unstuck will be enabled [**?**]. To get the satellite unstuck, the system will attempt to move the satellite in different directions until the satellite frees itself.

The first component which checks if the satellite is stuck will be running constantly in the background while the satellite is navigating itself to its destination. It works by checking the coordinates from the GPS sensor every five seconds. After taking a new reading, the algorithm will determine the average speed the satellite has been moving in those five seconds. If the average speed is below the minimum threshold of 0.2 m/s, the satellite will switch from its normal navigation mode to getting unstuck mode [**?**]. This mode will work by attempting to move the satellite a different direction every five seconds, then checking the GPS coordinates like before to see if the average speed is greater than the minimum threshold of 0.2 m/s. If it is, then the satellite must have moved and the system can switch back to its normal navigation mode. The first direction the rover will try is directly backwards, and each time the rover fails to get unstuck, the algorithm will attempt to drive 45 degrees clockwise of the previous direction driven. If the algorithm travels 360 degrees, then it will drive 45 degrees counterclockwise. This cycle will continue until the satellite gets unstuck, or the satellite runs out of battery.

This system will be implemented in two separate pieces. The first piece, which detects if the satellite is stuck, will be running concurrently with the navigation system. The algorithm will be a continuous loop, which checks the coordinates from the GPS, calculates the average speed from the current coordinate and the previous coordinates, checks if the speed is less than the minimum threshold, and sleeps for five seconds. If the speed is less than the minimum threshold, an interrupt will be sent, transferring control from the navigation system to the getting unstuck system. This piece will be located in the class which contains the second stage of control, which is navigation. This second piece will be located in the class which contains the third stage of control, which deals with the satellite becoming stuck, or on its side. The algorithm will be a continuous loop which increments the direction by 45 degrees, attempts to move for five seconds, and checks if the satellite has moved by calculating its average velocity and comparing it to the minimum threshold. Once the satellite is determined to be unstuck, control will be transferred back to the navigation system.

X. Finding and Touching the Finish Pole

For the protocol to find and touch the finish pole, we have decided to use the existing CMOS imaging sensor to find the pole. Once the pole has been located, the satellite just needs to drive in the direction of the pole until it makes contact with it. The most complicated portion of this is the object detection algorithm. This algorithm will use the OpenCV library, due to its object recognition capabilities [**?**]. The object detection algorithm, however, will work differently than the obstacle avoidance algorithm.

Control of the satellite will be switched to this system once the satellite is within the error range for the GPS coordinates of the finish. Once in this mode, this system must do two things, locate the pole, and move towards the pole. To locate the pole, the satellite will slowly rotate, scanning its surroundings with the CMOS imaging sensor. An image will be taken every second, and each image will be scanned for a specific shape. That shape is the shape of the finish pole, which would resemble a long, skinny rectangle on a 2D image. This shape scanning ability is achievable using the OpenCV library [**?**].

Once the shape has been located, the algorithm will instruct the satellite to move forward. Images will still be taken every second, and the direction the satellite is driving will be adjusted based on how far the shape is from the center of the image. If the shape is on the right of the image, the satellite will direct itself more to the right, and if the shape is on the left, the satellite will direct itself more to the left. Once the satellite is very close to the target the shape may not be detectable, so the satellite will continue to move forward in the same direction. Once the satellite hits the pole, it may get deflected and drive in another direction, but we will have completed the competition, so that isnt important. The system can turn itself off after a certain amount of time has passed.

This system will be implemented as its own class. It is the final stage class, and is started once the satellite is within the error margin of the finish pole. There will be two functions in this class. The first will search for the pole, and the second will move towards the pole. The first function will be implemented as a continuous loop, which rotates the satellite ten degrees, gets an image from the CMOS sensor, and checks that image for the shape of the pole using OpenCV. Once the pole is located, the next function will be called. This function will also be implemented as a loop. Each iteration, it will move the satellite forward, get an image from the CMOS sensor, use OpenCV to find the location of the pole in the image, and determine how far to the left or right to navigate the satellite based on the location of the pole in the image. Eventually, the satellite should hit the finish pole, and we will have completed our mission.

XI. Navigation system design

Navigation system is the one of the most crucial parts of any autonomous vehicles. The team is planning to implementing the rover's navigating system with two major goals: to get the rover drive towards the target pole without getting stuck; Design or use some existing algorithms to let the rover remotely updates the best driving path, in order to achieve the optimized battery life. The team divided the navigation system implementation into hardware and software two pieces, in order to help us research more comprehensively.

On the hardware side, by far we have designed two layers to ensure that the CanSat to meet the functional requirements. The team first collaborated with the electrical engineering team and made the agreement on that we will use the FGPMMOPA6C MTK3339 GPS chipset. The MTK3339 ultimate GPS module has an excellent high-sensitivity receiver and a built in antenna. It is capable to get up to -165 dBm sensitivity, track 22 satellites on 66 channels with a 10 Hz update rate, with its ultra low power usage, this GPS module is unbeatable considering its the under 30 dollars price and ultra compact size(16mm * 16mm *5 mm, 4 grams). A GPS module essentially generates a set of way-points(path) based on the given target pole coordinates, in order to make the CanSat to have more precision control and a better motion awareness, we decided to add a 3 axis accelerometer onto the system. At this time being, we have not yet finalized the selection on which specific accelerometer we are going to use on the CanSat.

Software design is the most important piece of the Navigation system, as mentioned above, the CanSat will navigate itself to the target pole based on the given pole coordinates. In order to enable the autonomous driving feature, we have made a basic design which is based on an existing function called distance and angle calculation method.[**?**] With this method, the system essentially compares its current location with the target pole location, Calculate the shortest path between these two points, then travels through the shortest path. If the system senses any obstacles in the path, it will calculate the new path based on the new current location. This simple process is continued until the CanSat gets close to the target pole(roughly within 8 meters), the target pole mode will be turned on in this case. Lastly, if the CanSat has successfully touched the target pole, it gets stopped, and the whole system will be shut down, see **Fig. 1**. This distance and angle calculation method can be expressed in a set of mathematical equations, please see below:

$$A = Long_{pole} - long_{current}$$
$$B = Lat_{pole} - lat_{current}$$
$$\theta_{pole} = tan^{-}1(B/A)$$
$$Angle(\theta) = Pole_{angle} - Current_{angle}$$

Where, A is distance to be travelled in latitude and B is distance to be travelled in longitude

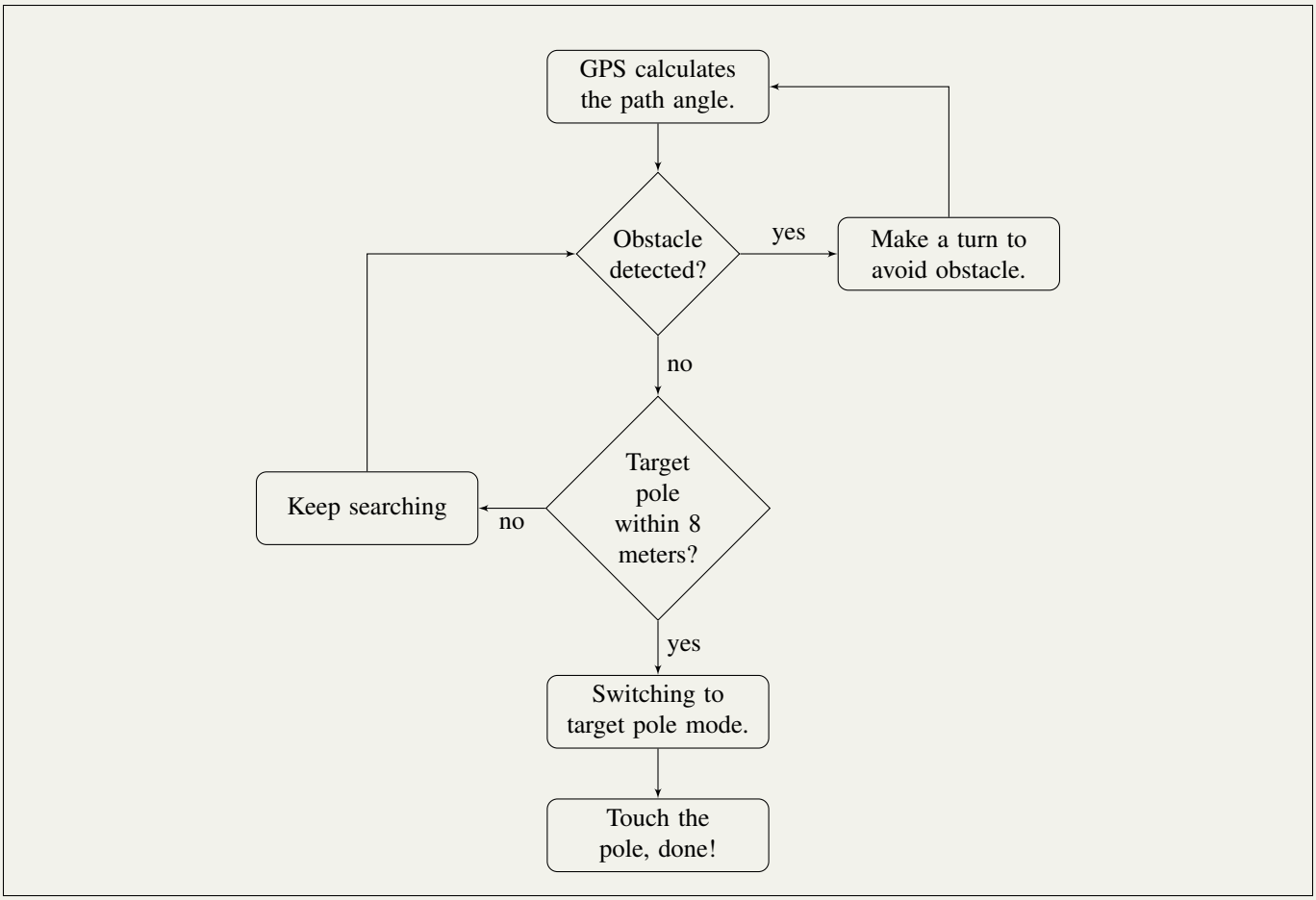

Fig. 1: the flowchart above shows the details of how the navigation system works.

Furthermore, for details about the obstacle avoidance algorithms, as we discussed in the section No.VII, we are essentially using the CMOS image sensor to constantly generating pictures of the front of the path, convert those pictures into a 2D array of binary values, where 1 means object and 0 means open space. [**?**] This 2D array contains a set of binary values can be expressed as this imaginary figure below, the center of this array has a bunch of 1 values which represent an object, more specifically, is an obstacle.

```
00000000000000000000000000000000
00000000000001111100000000000000
00000000011111111111000000000000
00000001111111111111100000000000
00000000011111111110000000000000
00000000001111111100000000000000
00000000000011111100000000000000
00000000000000000000000000000000
```

XII. Getting rover unstuck when it on its side

The main goal of this task is to get the rover recovered from falling sideways. We have designed a very efficient protocol that ensures the rover is able get unstuck under any kind of scenarios. This protocol can be simply broken down to three layers. The first layer is to determine whether the rover has fell sideways, second layer is called self-rescue program, and the third layer is to place a servo driven arm on the rover body. These three layers has to be executed in this sequence of order in order to make the protocol operates in a cost effective fashion.

To determine whether the rover is operating under normal the conditions or running into problems, we simply utilize its moving speed and heading data from comparing the current waypoint with the last waypoint. If suddenly, the accelerometer

captured an unexpected motion and the rover is constantly sitting on a same waypoint, or moving under a minimum threshold of 0.2m/s, if it is, the system will make the decision on that the rover fell sideways, send an instruction to let the system enable the self-rescue layer.

The self-rescue program layer is a script we are going to implement in the third stage of control. After the system has confirmed that the rover is on its sides, the microprocessor will send an instruction to the motor controller, triggering the motor generating the maximum torque to drive the treads moving opposite direction of its heading, three times. Normally, the motor we are going to use is able to generate torque up to 120 Nm, this type of sudden force is big enough to get the rover recovered consider the rover weighs 300 grams. If the program failed on the rover recovery, then the system will enable the third layer of the protocol.

The third layer is an add-on, though the team believed that the self-rescue program has enough power to recover the rover itself from land sideways. Mechanical engineering team has still made the decision on adding an extra servo on the rover in case for any need. This mechanical arm will be used in the fairing process as well as the third layer of the recovery protocol. If the self-rescue program failed to recover the rover, then the program calls that servo to move mechanical arm push perpendicularly the against the ground to make the recovery.

XIII. Payload fairing

The main purpose of the research done into this subsystem was to help determine the ways the payload can stay safe during its descent and landing. Fairing is essentially the first task the CanSat has to pass when it landed on the ground. The major goal of the fairing protocol is get the rover out then move away from the fairing, and parachute quickly, safely, and without stuck, jam, or having any residues.

Per competition requirement, it is known that the fairing will be built out of a soda can and will probably contain cushioning on the inside to protect the payload. So this limits the options that can be used to separate the fairing. We have developed a solution that is inspired by the SpaceX Falcon Heavy and Falcon 9 payload fairing design, or two half-shells fairing design.[**?**] We broke down this fairing protocol into two stages. First determine whether the CanSat is landed or not, if so then move onto the fairing separation stage so that the rover is able to move away as quick as possible.

To determine whether the CanSat is landed or not, we use the accelerometer to track its ground hitting motion. Speaking the motion, the CanSat create a big deceleration when it goes from descent into a complete stop, any accelerometer in the market is capable to capture this motion. As soon as the accelerometer captured this motion, a signal will be sent to the microprocessor, triggers the fairing separates into two halves. The separation of the fairing is the part that we still stuck on at this moment, we have came up with two feasible options. First, a small explosive or a compressed gas will have to be placed in the fairing in order to cause the separation. Second, as mentioned above in section **XII**, there will be an mechanical arm placed on the rover, this servo controlled arm is primarily used to get the rover unstuck when it falls sideways, we can also write a small script to let this arm to make a series of motions to cause the separation.

Additionally, we have to ensure that the rover moves away from the fairing the opposite direction, compare to its moving direction before landing. This action will ensure that the rover won't get jammed with the parachute and able to move onto the next stage as soon as possible. We simply write a script that let the GPS and accelerometer to determine it's landing direction, and set the rover's initial heading the opposite direction of its descent heading. Many teams from previous years competition have stuck on this stage, specifically their CanSat were jammed by the parachute, made theirs rovers aren't able to go anywhere.