1.1. Autonomous system

The autonomous system is developed with Robotic Operating System (ROS) and C/C++. It has three functions: GPS Navigation, Autonomous Navigation and Image Processing. The whole system has the ability of navigating to the provided GPS coordinate, avoiding obstacles and detecting a tennis ball elevated 10-50cm off the ground. The design goal of the system originally is to help UCL Rover Team to compete in the University Rover Challenge, in Utah, the US. The autonomous system of UCL Mars Rover is consisted of two sub-systems: the master system and the slave system.

- The master system is developed with Robotic Operating System (ROS) as its free and open source ecosystem a natural fit for the team to cut costs and avoid re-engineering common robotic software. it is embedded into NVIDIA Jetson TX2 AI computing device.
- The slave system is an Arduino Mega 2560 microcontroller flashed with C/C++. The codes of Master System and Slave System are in the USB stick attached.

In UCL Mars Rover, performances of some components are better than NASA's Curiosity Rover in on-board processor. The Curiosity Rover carries 200MHz BAE RAD750 CPUs with 256MB RAM, 2GB Flash and 256KB EEPROM, whereas UCL Mars Rover has 8GB RAM, one 2GHz quad-core ARM CPU and one 2GHz dual-core NVIDIA CPU.

Table 1: Rover Comparison

	CPU	RAM
NASA Curiosity Rover	200MHz BAE RAD750	256MB
UCL Rover	2GHz Dual-Core ARM	8GB

1.1.1. Autonomous System Initial Assumptions

In the autonomous task, the rover will autonomously traverse between markers (tennis balls) with GPS coordinate provided across moderately difficult terrain (Error! Reference source not found.) in the desert less than 2000m cumulative distance. However, the marker will not be close to the GPS coordinate, which requires the rover to detect the marker around the coordinate and drive towards it. During the cruise, the rover needs to avoid obstacles above a specific size to prevent it cannot move forward and GPS navigation to drive the rover to the marker.

Figure 1 Competition terrain

1.1.2. Master System

1.1.2.1. Hardware

The master system can be understood as the brain of the rover, as it will collect and process all the raw information from sensors and generate command to slave system that is responsible for controlling the action of the rover. It includes:

- **Jetson TX2 board:** The rover requires an on-board computer to process the raw information collected from sensors. The selection mainly focuses on processing abilities, compatibilities with other sensors and ROS. The board will take information from sensors and send commands to the slave system to move the rover.

- **Ublox NEO-M8N GPS + HMC5983 compass (XL) GPS unit:** Ublox GPS can output GNSS signal for rover's global localisation and autonomous navigation to a specific GPS coordinate.

- **ZED depth camera**: Traditionally, there are two ways to measure the distance for autonomous vehicles. One is Light Detection and Ranging (LiDAR) and another one is depth camera. Depth camera was chosen as it proved more cost efficient than LiDAR and has better integration with other components of the system. ZED depth camera is able to provide real-time coloured image, depth image, point-cloud image streams and odometry information and hence the obstacle avoidance can be achieved (**Error! Reference source not found.**) (34).

- **SparkFun Razor 9DoF Razor IMU (Inertia Measurement Unit).** The IMU is for rover's local localisation that features three 3-axis sensors – an accelerometer, gyroscope and magnetometer that give the rover the ability to sense linear acceleration, angular rotation velocity and magnetic field vectors.

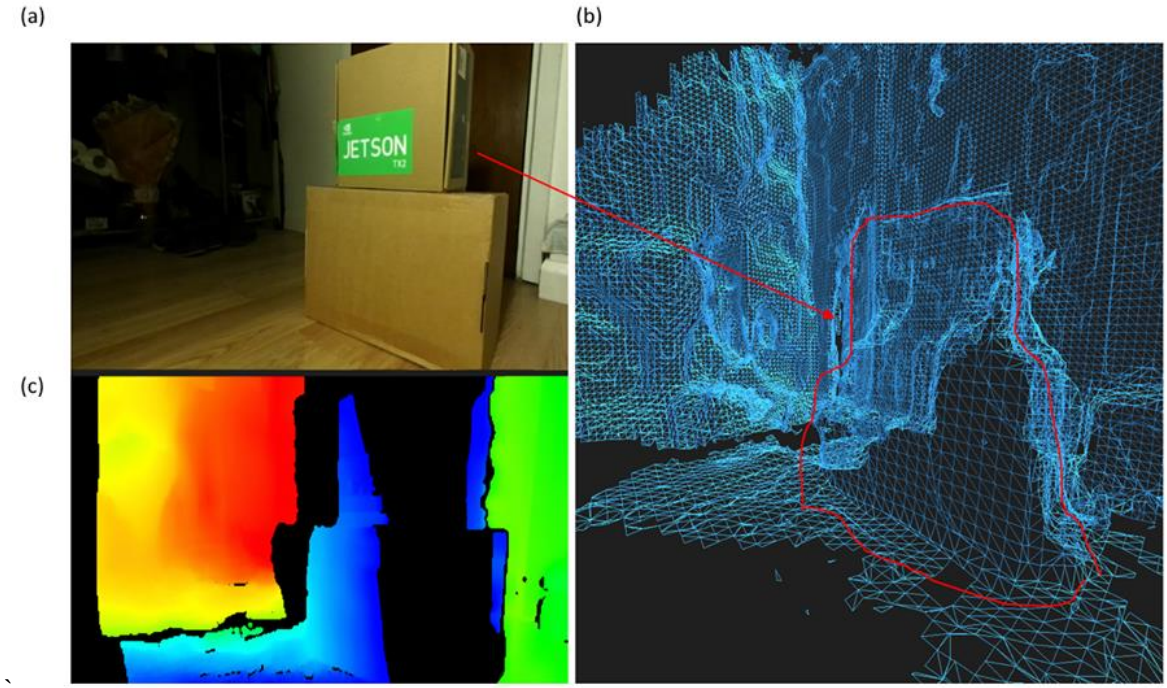


Figure 2: ZED Camera Raw Image, Depth Image and Fusion (a) Real image (b) Fusion mapping (c) Distance

1.1.2.2. Hardware Integration

GPS and IMU connected with Jetson board through GPIO expansion headers (specific connection layout and their electronic characteristics refer to Appendix D. Appendix E.). ZED camera connected to Jetson through USB serial connection.

1.1.3. Software Architecture

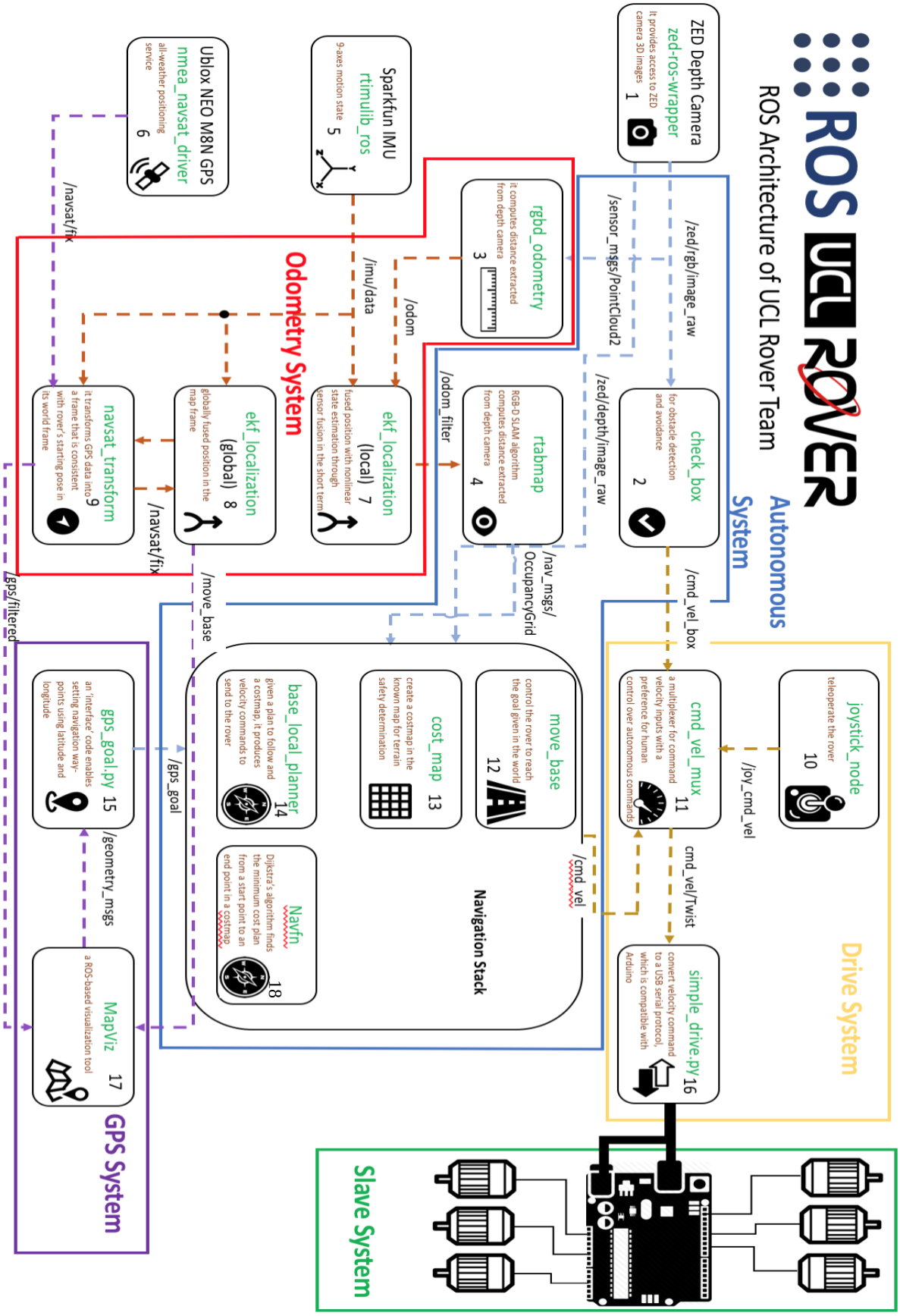


Figure 3: Software architecture used for the autonomy system in UCL Rover

1.1.4. Autonomous Navigation

The design of the sub-system uses a ZED depth camera's visual odometry function and SLAM (Simultaneous Localization and Mapping) to autonomously navigate the rover to a specified GPS coordinate and avoid static obstacles.

In this process, the ZED Camera outputs odometry data for processing. The processed visual odometry data will later generate an estimated localisation for the rover using an Extended Kalman Filter algorithm.In the next step, RTAB-Map algorithm (Real-Time Appearance-Based Mapping) creates a continuously growing point cloud of the world using simultaneous localization and mapping (SLAM) [3]. Inherent to the SLAM algorithm is pin pointing the own location in the map that is building as the rover moves. Using this map, RTAB-Map then creates an occupancy grid map [4] where the processed results will be reflected on the *navigation stack*[12,13,14,18] to avoid obstacles. When the rover is on terrain the system can detect static obstacles with the help of *checkbox*[2], an open source computer vision library developed for object detection and avoidance by OpenCV.

Simulation was generated to validate the functionality of ROS Autonomous System. The simulation outcome has the same characteristic with ZED outcomes in reality (Appendix B.) and the actual terrain in the simulation can be mapped by the ZED camera, hence the functionality of ROS architecture of Autonomous System has been validated successfully. However, due to the rover is distant to the terrain, the terrain mapped is slightly hazy. By this time, the rover can map and navigate in the reality.

1.1.4.1. Path Planning

The odometry, obstacle detection and tennis ball detection functions allow the rover to react dynamically to unknown circumstances when traversing. However, in order to navigate efficiently, the system needs a path planning algorithm that traces the most optimal trajectory to the goal location, in this case, *navigation stack*[12,13,14,18].

The navigation stack is consisted of three *ROS nodes*[12,13,14,18] and they are able to function collectively. The ROS navigation stack is a collection of components and it performs *global planning*[18] and *local planning*[14]. Global planning provides a fast-interpolated navigation function that can be used to create plans for the rover. The planner assumes a circular robot and operates on a cost map to find a minimum cost plan from a start point to an end point in a grid that is computed with Dijkstra's algorithm (Appendix C.) (36), whereas local planning provides implementations of the Trajectory Rollout and Dynamic Window methods to calculate the desired linear and angular velocities $(dx, dy, d\theta)$ of the rover (37) in a computing period.

```
1  TrajectoryPlannerROS:
2      acc_lim_x:  0.5 # maximum acceleration of x direction in m/s^-2
3      acc_lim_y:  0.5 # maximum acceleration of y direction in m/s^-2
4      acc_lim_theta:  1.00 # maximum angular acceleration in rad/m^-2
5
6      max_vel_x:  0.27 # maximum velocity of x direction in m/s
7      min_vel_x:  0.20 # minimum velocity of x direction in m/s
8      max_rotational_vel: 0.4 # maximum wheels velocity in rad/s^-2
9      max_vel_theta:  0.1 # maximum angular velocity in rad/s
10     min_vel_theta:  -0.1 # minimum angular velocity in rad/s
11
12     yaw_goal_tolerance: 1.39626 # the maximum yaw angle (80 degrees)
13     sim_time: 1.7 # set between 1 and 2. The higher the value, the smoother
14                      the path (though more samples would be required)
```

Figure 4: Important State of Motion Parameter Settings in the .yaml Configuration File

However, before using *move_base*[12], other parameters including wheel diameter, cruising cost, the distance to arrive the targeted position are required to configure by other. yaml files as well, which locate at the following files in the USB attached:

Table 2: List of Configuration. yaml Files

base_local_planner_params.yaml	costmap_common_params.yaml
global_costmap_params.yaml	local_costmap_params.yaml

1.1.4.2. Odometry System

The goal of Odometry System is to estimate the relative distance of the rover travelled from the start. The mechanism of Odometry System with *RTAB-Map*[3] has been explained in section 1.1.1.

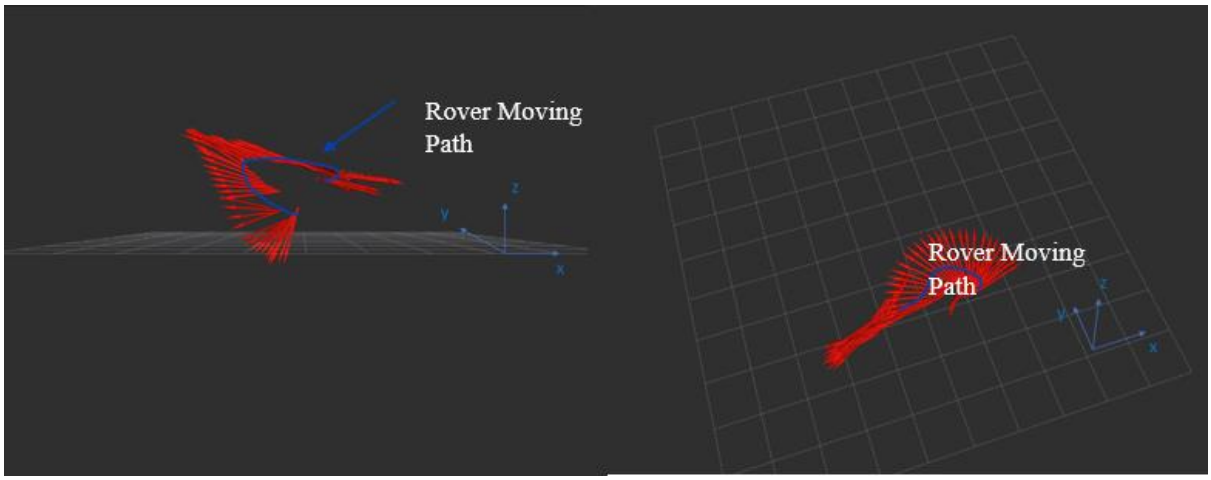


Figure 5: ZED's Built-in Visual Odometry Function (Red Arrows Point the Rover Direction)

As the Autonomous System is not functional without the Odometry System, so they cannot be explained separately. In the rover, visual odometry has been applied other than wheel odometry as it performs better than wheel odometry in the complex desert terrain as it is not affected by wheel slips (38). Except the visual odometry function from ZED camera (section 1.1.1), a local odometry from *IMU*[7,8,9] and a global odometry from *GPS*[9] have been applied to increase the estimation accuracy of the relative distance from the start.

1.1.4.3. GPS System

The task requires the rover can be navigated to a series of specified GPS coordinates. To use GPS navigation function, firstly, GPS coordinate needs to be transferred to ROS coordinate via WGS84 ellipsoid and geographiclib python library in *gps_goal.py*[15] node. This node will transfer the actual GPS coordinate (latitude, longitude, altitude) to float 64 (**Error! Reference source not found.**) and hence can be imported into ROS.

```
geometry msgs/PoseStamped # ROS topic subsribed
    std_msgs/Header header # ROS message subscribed
    geometry_msgs/Pose Pose # ROS message subscribed


sensor msgs/Pose pose # ROS message subscribed
  uint8 COVARIANCE_TYPE_UNKNOWN= 0
  uint8 COVARIANCE_TYPE_APPROXIMATED= 1
  unit8 COVARIANCE_TYPE_DIAGONAL_KNOWN= 2
  unit8 COVARIANCE_TYPE_KNOWN= 3
  std_msgs/Header header # Publish parameters above
  sensor_msgs/NavSatStatus status # Enable the node to read GPS coordinate input
  float64 latitude # Regulate calculation method of latitude in the system
  float64 longitude # Regulate calculation method of longitude in the system
  float64 altitude # Regulate calculation method of altitude in the system
  float64[9] position_covariance # Regulate calculation method of parameters above
  uint8 position_covariance_type # Send
```

Figure 6: ROS message of gps_goal.py

The GPS sensor is required to calibrate and configure in term of speed, localisation accuracy and signal before the usage and are achieved by the interface below:

Figure 7: GPS calibration interface

However, many tests suggested the transferring outcomes were inaccurate. After experiments, it has been found that before transferring GPS coordinate to ROS coordinate, the node is required to initialise. After initializing, publishing targeted GPS coordinate to the rover to navigation and the rover drive autonomously to the target position.

GPS Coordinate Input

```
> pose:
>   position:
>     x: 0.0125
>     y: -0.00147
>     z: 0.0
>   orientation:
>     x: 0.0
>     y: 0.0
>     z: 0.0
>     w: 0.0"
publishing and latching message. Press ctrl-C to terminate
^Cchenxi@chenxi-Alienware-15:~$ rostopic pub /gps_goal_fix sensor_msgs/NavSatFix
header:
>   seq: 0
>   stamp: {secs: 0, nsecs: 0}
>   frame_id: ''
> status: {status: 0, service: 0}
> latitude: -0.00149
> longitude: 0.0125
> altitude: 0.0
> position_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
> position_covariance_type: 0"
publishing and latching message. Press ctrl-C to terminate
```

Figure 8: GPS coordinate input interface

Once the target coordinate is set, the rover will drive towards to the position with path planning, which is shown in **(**Error! Reference source not found.**)**.

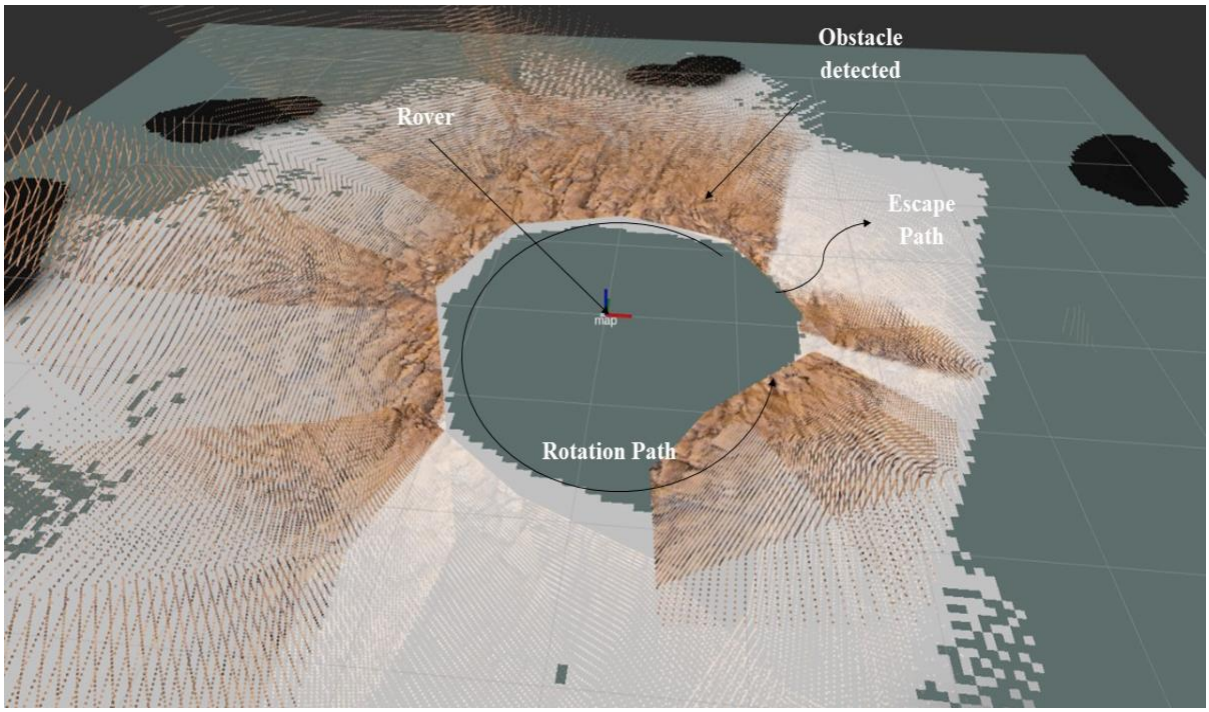


Figure 9: Path planning function simulation

On the way to the target point, if the rover detects there are obstacles above the threshold set, it will move around to detect the surrounding until it finds the way out. Once the rover has arrived at specific GPS coordinate, the system will output a command to indicate the rover has arrived.

1.1.5. Drive System

The Drive System is displayed as "red square 2" in the systematic function diagram. Two functional nodes in Drive System are *cmd_vel_mux*[11] and *simple_drive.py*[16]. *Joystick node*[10] is for teleoperation. The *cmd_vel_mux*[11] is a velocity multiplexer node. It sends speed command based on the signal processed from Autonomous System to *simple_drive*[16], which is a serial communication protocol node (set with the *serial_dev* and baudrate ROS parameters) between the Master and Slave System as the command output cannot be directly recognised by Slave System. This node could be eliminated if the microcontroller supports ROS.

1.1.6. Slave System

The slave system is developed on C/C++ language (code of *drive_firm* is in the USB attached) and has been deployed on Arduino mega2560 microcontroller through PlatformIO, an IoT platform. The Arduino board receives motor commands from the *simple_drive*[16] node over a USB serial connection and output voltages to digital PWM (pulse-width modulation) output to be received by motor controllers (ESC, Electronic Speed Controller) introduced in section **Error! Reference source not found.**. The code defines the input impulse of full-forward and full-reverse are 2000 and 1000 μs (Appendix F.). It sets the velocities for wheels of both side of the rover calculated by:

$$Left\ Side\ Wheels\ Linear\ Velocity = V_{linear} + \omega \times r$$

$$Right\ Side\ Wheels\ Linear\ Velocity = V_{linear} - \omega \times r$$

Where V_{linear} is linear velocity, ω is angular velocity and r is the radius of wheels

1.1.7. Simulation

To test the functionality of Master System of the rover, the simulator called RVIZ and Gazebo have been used. RVIZ simulator is more for having physical hardware and data visualisation. It is used to test the functionalities of the physical hardware (ZED in **Error! Reference source not found.**, Error! Reference source

not found.), GPS navigation, path planning and object detection. Gazebo has been used for the whole ROS architecture simulation, for it has a physics engine that allows to simulate the hardware and an environment.

In Gazebo simulation, a Mars-liked environment (39) has been applied and a husky rover (40) is used as a rover model as it utilises the same differential velocity motion mechanism with the UCL Mars Rover.

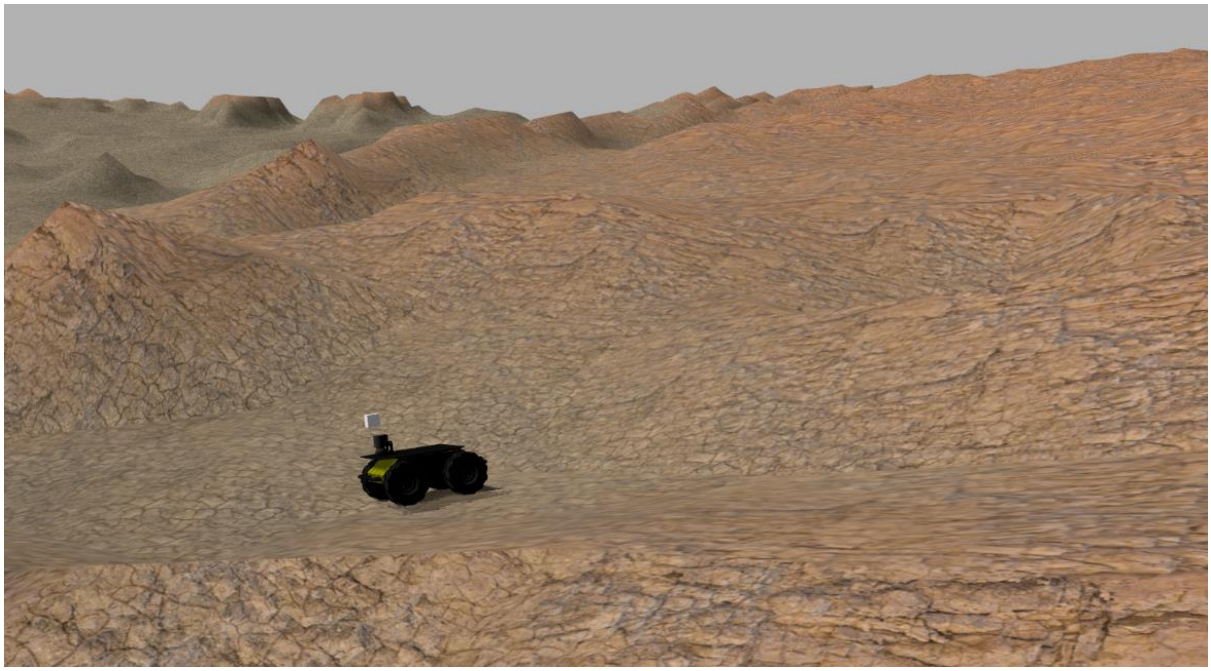


Figure 10: Gazebo simulation in a Mars like environment

Once the simulation and other necessary ROS packages for simulation have been launched in the simulator, a GUI plugin called *rqt_graph* (Error! Reference source not found.) is used to visualise the ROS computation graph for testing the correctness of ROS architecture functionality and integrity, where square 1 represents localisation and obstacle avoidance functions in section 1.1.1 and square 2 is Drive System from section 1.1.5. It can be seen that the communication has been achieved between each function node with no errors.

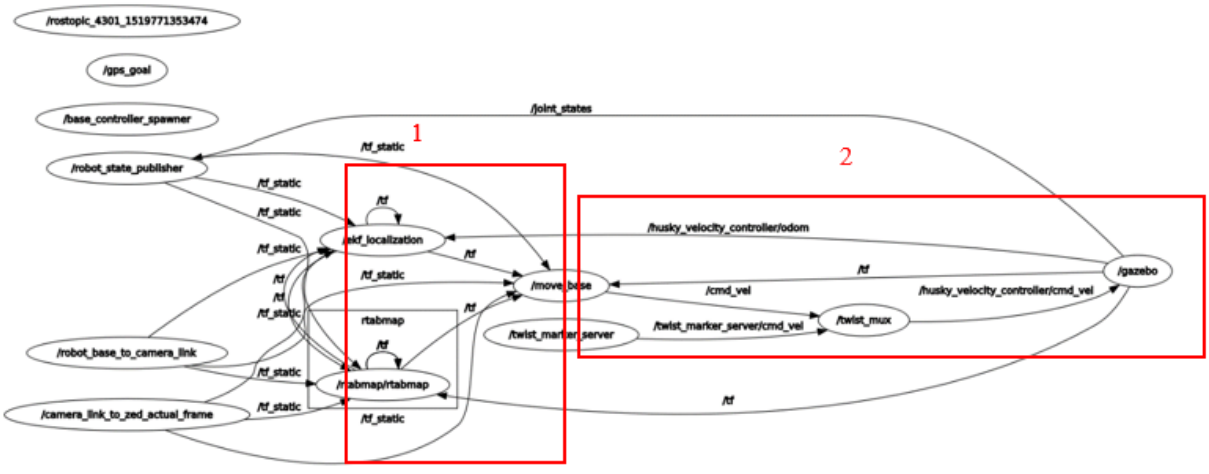


Figure 11: ROS computation and function graph from rqt_graph viewable interface

When the rover arrives at the specific GPS coordinate (Error! Reference source not found.), a ball assigned with the tennis ball colour characteristics will appear in the camera and be detected through OpenCV Hough Circle Transform [10] and send signals to control the rover towards it (Error! Reference source not found.).

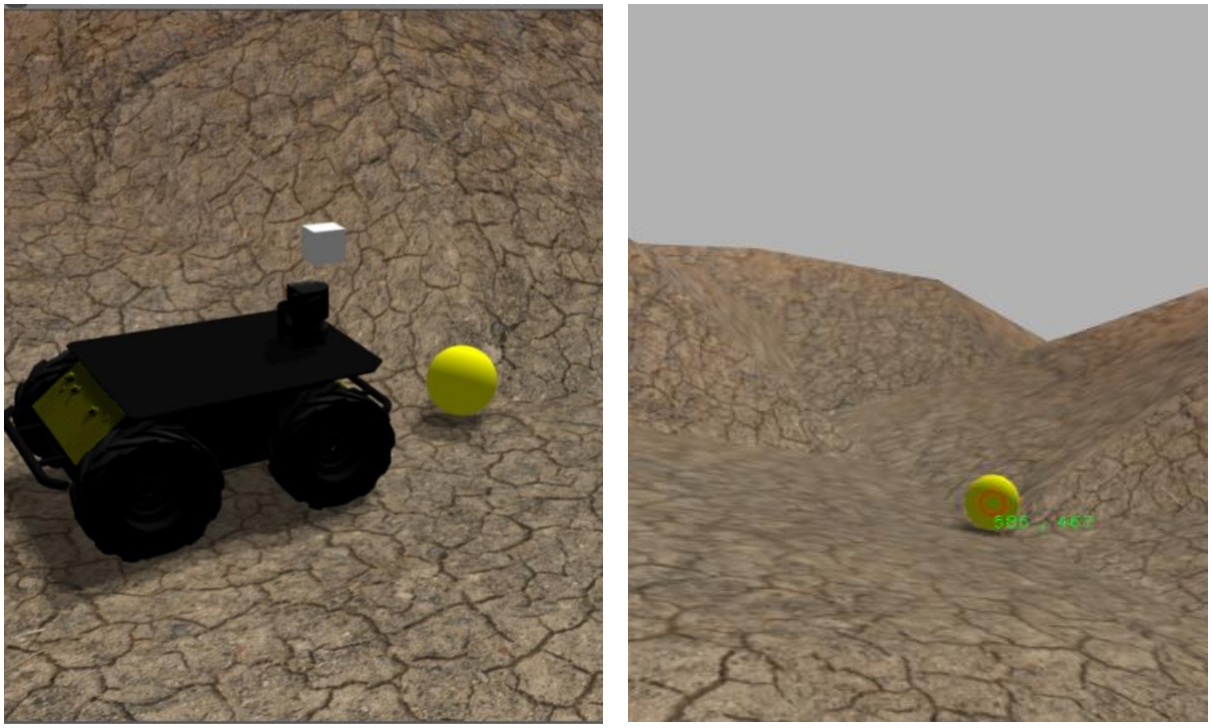


Figure 12: Process of tennis ball detection by the autonomous system

1.1.8. Future Improvement

The future improvement relates to long distance communication from the ground station to the on-board Jetson.

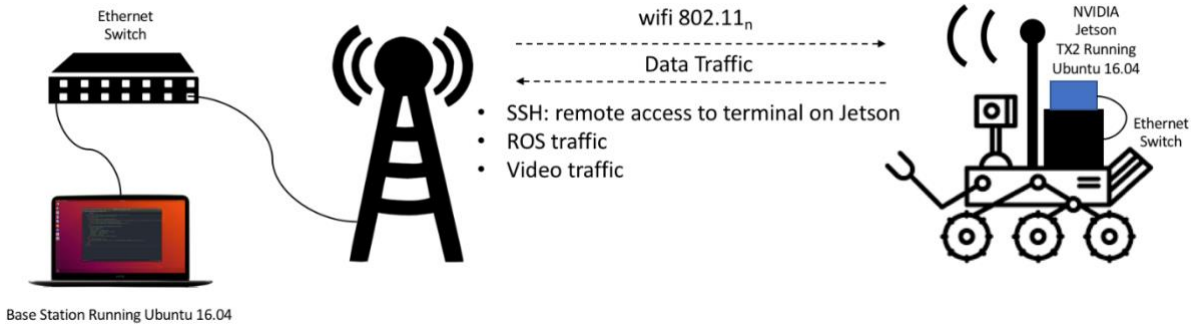


Figure 13: Telecommunication and remote-control improvement of the future rover

The wireless communication between the rover and ground station can be achieved by the configuration in (Error! Reference source not found.**)** with two ethernet switches one at the ground station and one on the rover with antenna installed. The data traffic can transfer live video stream and commands sent from ground station. Also, the rover requires on-field tests to maximise its performance.

Appendix A. ROS Terminologies

ROS Packages

Software in ROS is organised in ROS packages. A ROS package, which is a central concept to how files in ROS are organised can be analogised into a folder in the computer. It may contain ROS nodes, a ROS-independent library, a dataset or a third-party software.

ROS Nodes

A ROS node is a process that performs computations. A robot control system will comprise many nodes and each node is independent to each other, which means each node conducts separate function and not interfere to the others. This setting ensures once one node is fault, the fault is isolated to other nodes and it decreases the difficulties of debug. For example, in the UCL Mars Rover, *ekf_localisation*[7] node (see Odometry System, Figure 2.4) will only provide localising function and only *cmd_vel_mux*[11] (see Drive System, Figure 2.4) controls the wheel motors.

ROS Topics and ROS Messages

ROS topics are unidirectional and are named buses over which nodes communicate to each other. As every ROS node is independent in functionality and topological structure, ROS topics are the channel for nodes to send and collect data. By subscribing to the relevant topic, the data then can be sent and received by the interested nodes. There can be multiple data publishing nodes (publisher) and subscribing nodes (subscribers) to a topic, whereas this data stream is called ROS messages (integer, floating point, Boolean, etc.).

Figure 1: ROS Topic Bus

ROS node zed-ros-wrapper publishes ROS message that contains the data processed by ZED camera through ROS topic /zed/rgb/image_raw to another node check_box, which subscribes the same topic. The software environment of the Master System is using ROS Kinetic software built on Ubuntu 16.04 LTS.

The master system software architecture includes four sub-systems: Autonomous System, Odometry System, Drive System and GPS System. Each sub-system is made of many ROS nodes that are communicated with ROS messages through ROS topics subscribed.

sensors (depth camera, GPS and IMU) are required to switch on and launch in the software environment by their ROS drivers (*zed-ros-wrapper*[1] for ZED camera, *nmea_navsat_driver*[6] for Ublox GPS, *rtimulib_ros*[5] for

IMU). When the sensors are launched, they will transmit the raw data collected as ROS messages to other data processing nodes through respective ROS topics (/zed/rgb/image_raw, /zed/depth/image_raw, /sensor_msgs/PointCloud2 are ROS topics for ZED camera; /navsat/fix for GPS; /imu/data for IMU).

The ROS nodes in four sub-systems are responsible for functional implementation and operation. Their responsibilities are path planning and object detection for Autonomous System; Localisation including local and global localisation for Odometry System; GPS navigation for GPS System; Desired action output and control for Drive System.

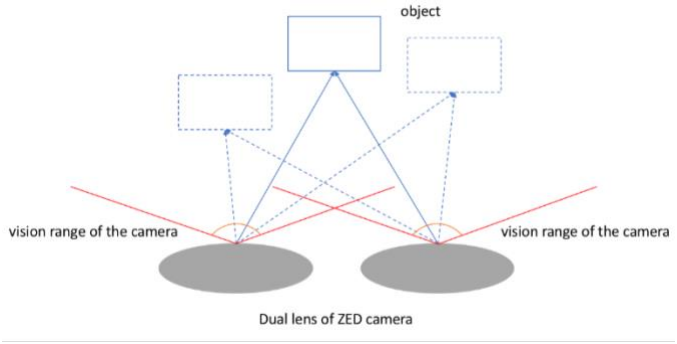
Appendix B. Master Hardware setup

JETSON

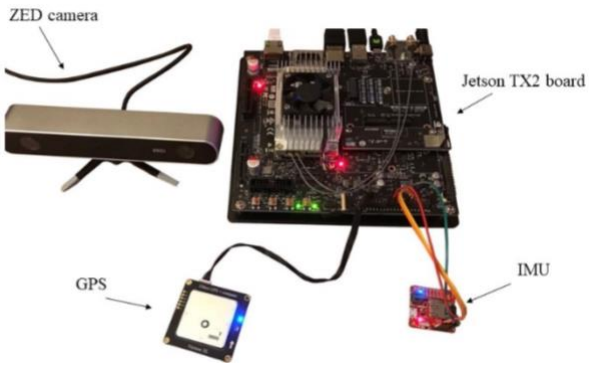
Jetson TX2 board (**Figure 2.2**) the development tool with its 8GB RAM, 32GB storage complete multi-channel PMIC in processing speed advantages and 400 pin high-speed I/O connector in hardware integration compatibility as well as strong supportive platform has been chosen as the master system carrier.

ZED

ZED camera uses the similar navigation mechanism of human vision. It uses a dual camera to create the depth maps. As the distances of an object in the front of the cameras are different, by producing synchronised right and left images as a single image with developed vision-based algorithm, the camera can measure the distance of objects in the front.



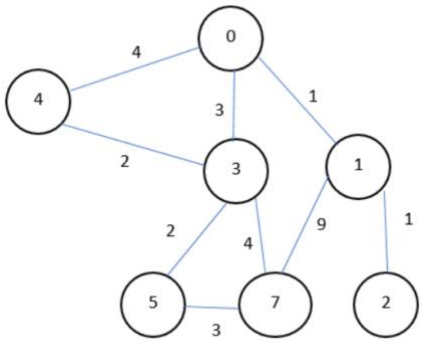
Hardware integration



Appendix C. Dijkstra's Algorithm

Dijkstra's algorithm works similar to BFS algorithm; however, it uses the cost of the path function g(n) to sort the queue. The evaluation function f(n)=g(n) represents the cost of moving from the cell being examined to its successor, and this cost gets added as the path evolves, meaning that Dijkstra will always expand first the paths with less cost.

Consider a network of cells, assuming we know the cost of moving from one cell to another as in the following figure.



Dijkstra algorithms would search for the goal as following:

Define environment as a grid matrix.

Define start and goal locations on the matrix

Add start cell to the queue.

While queue not empty

Get last cell on the queue and examine

If cell is goal cell

Return SUCCESS

Else

Compute cost f(n)=g(n) of paths to successors

Put cell in PARENTS LIST

If successor is in queue

Compare new cost to old cost

If new cost is lower

update cost and PARENTS LIST

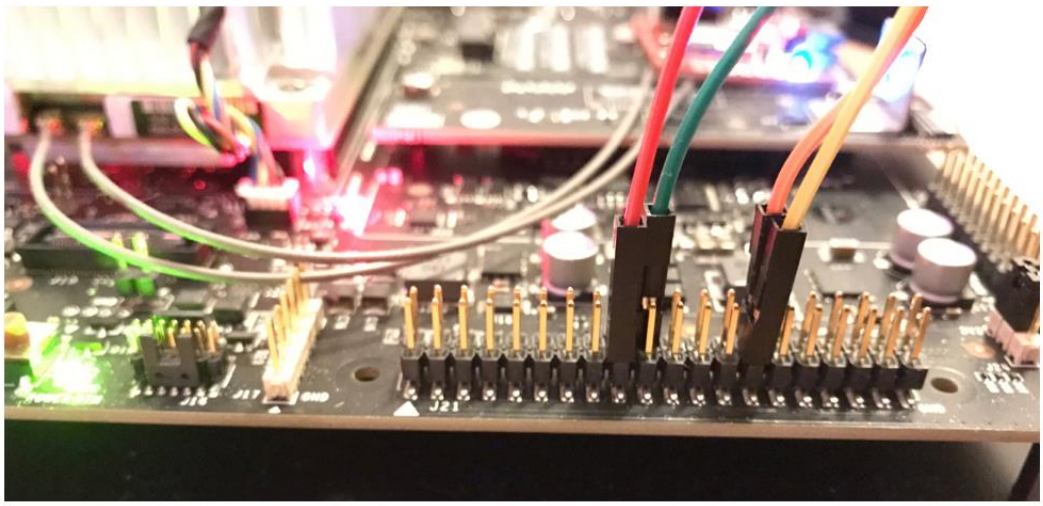
Else add successors to queue and sort by cost (lowest first)

Return FAILURE

Appendix D. Jetson TX2 GPIO Layout

Pin #	Signal Name	Associated Jetson Module Pin Name	Usage/Description	Type/ Direction	Signal Voltage Level at Header	GPIO Max Drive (I_{OL}/I_{OH}) or Power Pin Current Capability	Notes
1	CAN_WAKE	CAN_WAKE	CAN Wake	Output	3.3V	1mA / -1mA	2, 4
2	VDD_3V3_SYS	–	Main 3.3V Supply	Power	–	1A	1
3	CAN0_STBY	–	Unused	Unused	–	–	–
4	VDD_1V8	–	Main 1.8V Supply	Power	–	1A	1
5	CAN0_RX	CAN0_RX	CAN #0 Receive	Output	3.3V	1mA / -1mA	2, 4
6	AP2MDM_READY	GPIO15_AP2MDM_READY	AP to Modem Ready GPIO	Bidir	–	1mA / -1mA	2, 4
7	CAN0_TX	CAN0_TX	CAN #0 Transmit	Input	3.3V	1mA / -1mA	2, 4
8	VDD_5V0_IO_SYS	–	Main 5.0V Supply	Power	–	1A	1
9	CAN0_ERR	CAN0_ERR	CAN #0 Error	Output	3.3V	1mA / -1mA	2, 4
10	GND	–	Ground	Ground	–	–	–
11	GND	–	Ground	Ground	–	–	–
12	I2C_GP2_CLK	I2C_GP2_CLK	General I2C #2 Clock	Bidir/OD	1.8V	1mA / -1mA	2
13	CAN1_STBY	CAN1_STBY	CAN #1 Standby	Input	3.3V	1mA / -1mA	2, 4
14	I2C_GP2_DAT	I2C_GP2_DAT	General I2C #2 Data	Bidir/OD	1.8V	1mA / -1mA	2
15	CAN1_RX	CAN1_RX	CAN #1 Receive	Output	3.3V	1mA / -1mA	2, 4
16	WDT_TIME_OUT_L	WDT_TIME_OUT#	Watchdog Timer Output	Output		1mA / -1mA	2, 4
17	CAN1_TX	CAN1_TX	CAN #1 Transmit	Input	3.3V	1mA / -1mA	2, 4
18	I2C_GP3_CLK	I2C_GP3_CLK	General I2C #3 Clock	Bidir/OD	1.8V	1mA / -1mA	2
19	CAN1_ERR	CAN1_ERR	CAN #1 Error	Output	3.3V	1mA / -1mA	2, 4
20	I2C_GP3_DAT	I2C_GP3_DAT	General I2C #3 Data	Bidir/OD	1.8V	1mA / -1mA	2
21	GND	–	Ground	Ground	–	–	–
22	SLEEP	SLEEP#	Sleep Indicator	Output	1.8V	1mA / -1mA	2, 4
23	I2S1_CLK	I2S1_CLK	I2S #1 Clock	Bidir	1.8V	1mA / -1mA	2
24	I2S1_SDOUT	I2S1_SDOUT	I2S #1 Data Out	Bidir	1.8V	1mA / -1mA	2
25	I2S1_SDIN	I2S1_SDIN	I2S #1 Data In	Input	1.8V	1mA / -1mA	2, 4
26	I2S1_LRCLK	I2S1_LRCLK	I2S #1 Left/Right Clock	Bidir	1.8V	1mA / -1mA	2
27	DSPK_OUT_CLK	DSPK_OUT_CLK	Digital Speaker Out Clock	Output	1.8V	1mA	2, 4

Appendix E. Master System Hardware Components Connection Layout with Jetson



IMU connections

IMU Pin	Pin Description	Jetson Expansion Header Pin	Pin Description	Colour
1	Ground	20	Ground	Green
2	3.3V	17	3.3V	Red
3	SDA (I2C)	27	ID_SD (I2C Data)	Yellow
4	SCL (I2C)	28	ID_SC (I2C Clock)	Orange

GPS connections

GPS Pin	Pin Description	Jetson Expansion Header Pin	Pin Description
1	5V	4	5V
2	Rx	n/a	Not used
3	Tx	n/a	Not used
4	SCL (I2C)	5	SCL1
5	SDA (I2C)	3	SDA1
6	(Ground)	6	Ground

Arduino Mega Connections

Arduino Pin	Pin Description	Jetson Expansion Header Pin	Pin Description	Colour
A10	Ground	9	Ground	Green
A11	5V	2	5V	Red
	TX0	10	RXD0	Blue
	RX0	8	TXD0	Brown

Appendix F. Slave System PWM Settings

```
#define escReverse 1000 // UCL Mars Rover full-reverse input pulse
#define escForward 2000 // UCL Mars Rover full-forward input pulse

struct DATA {
    float linear;
    float rotation;
} received;

void setWheelVelocity(int left, int right) {
    leftOne.writeMicroseconds(map(left, -100, 100, escReverse, escForward));
    leftTwo.writeMicroseconds(map(left, -100, 100, escReverse, escForward));
    leftThree.writeMicroseconds(map(left, -100, 100, escReverse, escForward));
    rightOne.writeMicroseconds(map(right, -100, 100, escReverse, escForward));
    rightTwo.writeMicroseconds(map(right, -100, 100, escReverse, escForward));
    rightThree.writeMicroseconds(map(right, -100, 100, escReverse, escForward));
}
```