ORANGE SOIL ON THE MOON

QUT EGB320 2020 S2 Group 16

Abstract

This report details how a team of undergraduate students completed the challenge of finding and collecting orange soil samples from a mock lunar environment – with the focus of the report analyzing the vision subsystem

Isaac Tan

N9960210

Other team members: Jack Emanuel, Tom Bartlett, Drew Edwards

Table of Contents

Introduction

In the fifty years since mankind landed on the moon, there have been significant technological advancements. As such, the ideas of lunar habitation and interplanetary exploration are being considered more feasible. This report outlines a challenge proposed by the start-up company *Yanada320* - to land a lunar rover on the moon that finds and collects lunar soil samples while avoiding past rovers that have been stranded and left on the moon. The report analyses how a team of undergraduate University students attempted to solve this challenge, with the focus of the report detailing the vision and perception subsystem of an integrated TRL3 robot prototype. This subsystem applied various image processing transformations to a series of live frames from a camera module, segmenting distinct objects from the frame and applying mathematical relations to find relevant ranges and bearings of said objects. These ranges and bearings were supplied to the navigation subsystem which in turn determined the most appropriate course of action given the environmental parameters, further relaying this information to the mobility and sample collection subsystems to move the robot within the environment. These four integrated subsystems formed a cohesive robot that was ultimately able to find and detect the samples within the test environment and return them to a predetermined deposit point in the environment.

1.0 The Problem

Start-up company *Yanada320* aims to pitch a prototype of a lunar rover to potential sponsors in order to attain funding and ultimately land a rover on the moon to collect and test samples of soil for signs of water or other minerals. They have approached Group 16 of QUT's EGB320 unit to design and build this prototype. This prototype lunar rover is to be completed at a Technology Readiness Level (TRL) 3 and is required to detect distinct objects in a 2x2m mock lunar environment.

These objects include lunar samples, rocks, obstacles and the lander - represented by orange golf balls, blue 70x70mm cubes, green 150x150mm cubes and a yellow truncated octagonal base pyramid standing 45mm tall respectively. The environment is also enclosed by black walls 450mm tall.

The protype must collect as many of the orange spheres in the mock environment as possible within a five-minute timeframe while avoiding the obstacles and walls. Once a sample has been collected, it must be deposited at the lander situated in the centre of the environment. The environment contains two samples in the open, another obscured by an obstacle, and an additional two underneath rocks. In order to collect the two samples underneath the rocks, the prototype must be able to lift or remove these rocks, revealing the sample beneath them.

To solve this problem, the challenge was broken down into four subsystems - vision, navigation, mobility, and sample collection, each needing to be connected and integrated with one another to form a cohesive solution.

To market this protype as viable, it had to fit within a set of physical and financial constraints. The prototype was to be no larger than a 150x150x150mm cube, no heavier than 1.3kg and not exceed a budget of $185, with a further $15 allowed for prototyping and testing. Additionally, no component or system on the robot was to exceed 60V.

2.0 Relevant Theory

2.1 Colour Spaces

The various objects within the environment each had a distinct colour. In order to provide meaningful data to the navigation subsystem, each of these colours needed to be identified, and segmented. Three potential colour spaces were investigated and evaluated against what would be most beneficial for solving the problem.

2.1.1 RGB

The RGB, or red, green, blue colour space uses the addition of the three primary colours in varied proportions from 0 to 255 to create other colours (Photo Review, n.d.). Figure 2.1.1 shows how the three axes of colour are added, and when all three are added at full intensity, the colour white is created.

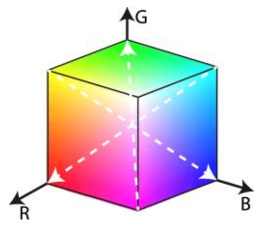


Figure 2.1.1 RGB model

2.1.2 HSV

HSV - short for hue, saturation, value uses a cylindrical model based upon RGB that integrates the saturation and brightness of a colour. The hue refers to the pure colour ranging from a degree of 0-360 - determined by the addition of the primary and secondary colours, similar to RGB. It should be noted that this degree from 0-360 is compressed from 0-179 in programming due to the 8-bit numbers used having an upper limit of only 256. The saturation relates the colour with its intensity, ranging from grey to vivid as a percentage. Finally, the value refers to the brightness or darkness of the colour, again as a percentage, where with no value the colour is black (MasterClass, 2020). Figure 2.1.2 shows this colour model in a conical form, where the hue ring can be seen against the outward axis of saturation and the vertical axis of value.

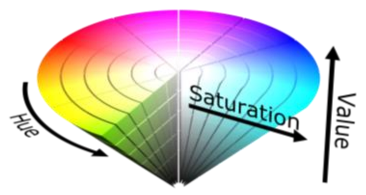


Figure 2.1.2 HSV model

2.1.3 CIELAB

CIELAB refers to a colour space certified by the International Commission on Illumination (CIE) that uses an intensity of lightness (L) from 0 to 100, a scale of green to red (a), and a scale of blue to yellow (b) to describe a colour. Figure 2.1.3 illustrates how these three values interact in a cylindrical model.

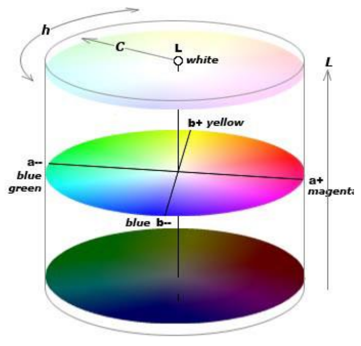


Figure 2.1.3 CIELAB model

2.2 OpenCV

OpenCV (Open Source Computer Vision) is a software library containing functions and algorithms related to computer vision and machine learning (OpenCV, n.d.). The library's pre-written and optimised functions allow for easier development of the vision subsystem and are used throughout the vision subsystem software.

2.3 Similar Projects

Past QUT students that undertook the EGB320 unit completed a similar task to the one posed by Yanada320. Students were to build a robot that detects an orange soccer ball and attempts to shoot it at a goal within a 2x2m soccer field. Students that completed this unit used hardware such as a Raspberry Pi with its camera module, the same hardware used by the EGB320 students this year for the lunar challenge.

Teams designing the vision system for this soccer robot also investigated the colour spaces previously mentioned, with most teams using HSV as it had better light variance acceptance. There were a few teams that opted for an omni-directional vision as opposed to the standard monocular vision system. An omni-directional camera allowed for 360^{o} vision around the robot, as opposed to the 62.2^{o} horizontal view that the Pi Camera gives. One team used a 3D printed mound-shaped mould and formed a reflective silver sheeting over-top. They then pointed the camera at this reflective mound which give the catadioptric vision system the 360^{o} view (Edwards, 2019). A similar design can be seen in Fig 2.3.1. This was investigated for use in the lunar challenge, however, it required significantly more time investment, calibration, and while it proved overall more beneficial for the navigation subsystem, made it harder to integrate.

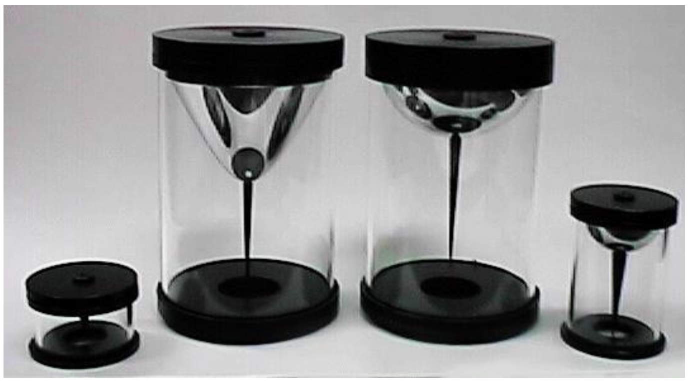


Figure 2.3.1 UPenn catadioptric camera modules
Image from: https://www.cis.upenn.edu/~kostas/omni.html

Many teams that completed soccer challenge used a three-motor omni-directional drive system to give a greater freedom of movement. This was investigated for the lunar challenge, however due to the 15^{o} incline of the lander, it was decided that omni-directional wheels were not optimal for this challenge. It was unlikely that the omni-directional wheels – designed to have little traction – would have enough traction to go up the incline of the lander.

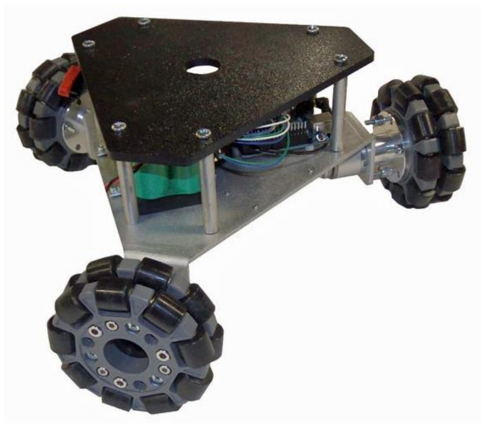


Figure 2.3.3 Three-wheel omni-drive system
Image from:
https://www.superdroidrobots.com/

3.0 The Solution

The team's final solution to the proposed problem was a differential drive system robot, that used the HSV colour space to capture and process images. This image processing compared a live captured image with a mask of previously calibrated HSV thresholds using a bitwise AND. The resulting segmented image data was then used to calculate the range and bearing of relevant objects using similar triangles.

This data was supplied to the navigation system which used a combination of a simple reactive system and potential fields to control motor movements and ultimately drive the robot to the targets while avoiding obstacles. Additionally, the prototype utilised a lever arm and gate attached to a servo motor to flip blue rocks over, revealing additional samples beneath them. To capture samples, the robot lifted the servo arm and gate and drove over the sample. A cut-out space in the base of the robot allowed the sample to be stored underneath the robot. To secure the sample, the gate was then lowered, enclosing the sample within the chassis. This cut-out containment section can be seen in Fig 3.0.1 along with the a front and side view of the final prototype.

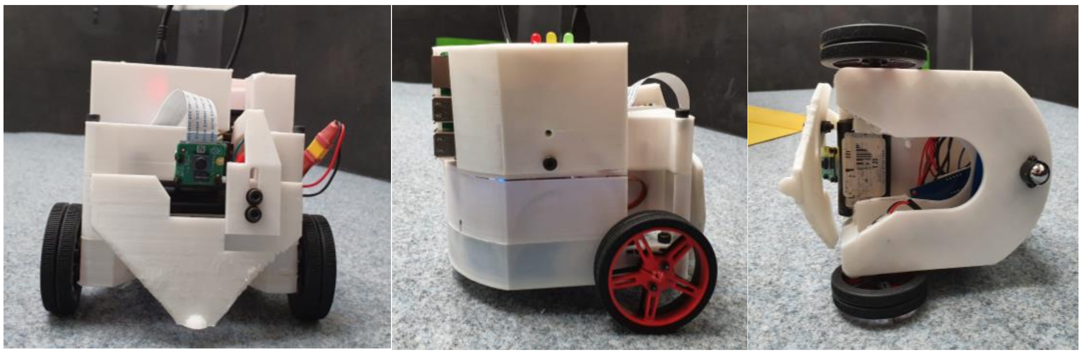


Figure 3.0.1 Final prototype design

A system architecture diagram of the integrated subsystems can be seen on the right in Fig 3.0.2.

This section of the report details how the vision system takes inputs from the mock lunar environment and the process undertaken to analyse the inputs to provide meaningful data to the other subsystems.

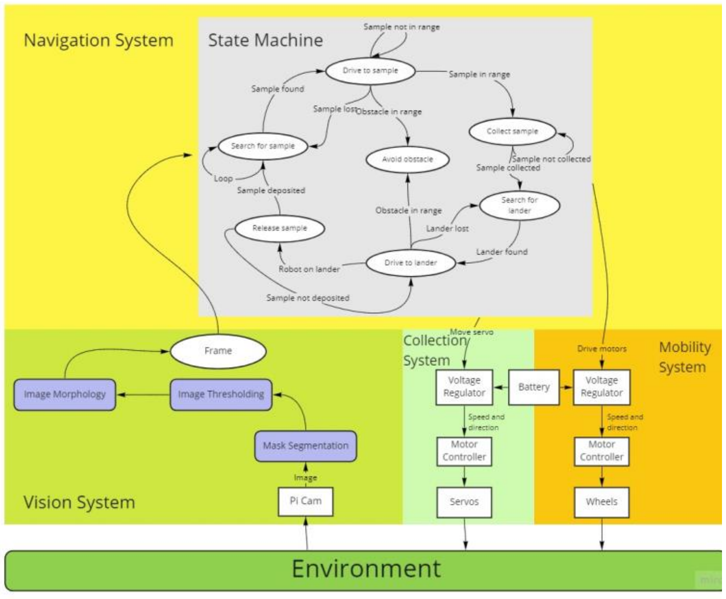


Figure 3.0.2 System architecture diagram

3.1 Vision and Perception

The main sensor used to perceive the environment was the Raspberry Pi camera Module v2. This captured an image of the environment directly in front of the robot with a range of 62°. In addition to this, a light dependant resistor was used in conjunction with a laser diode acting as a tripwire to detect when the robot had collected a sample.

During testing it was discovered that turning off the auto adjust setting on the camera and setting the brightness as a constant value for both the calibration and demonstration led to more consistent readings of HSV values. The brightness was adjusted in the experimentation phase of the project and it was deduced that having a lower brightness such as 0.4 allowed for a more consistent view of the yellow lander, however had limitations when sensing the blue rocks, green obstacles and the walls. Conversely, when the brightness was too high at a value of 0.6 the system was unable to detect the yellow lander. As such, the brightness was set at a value of 0.5. This can be seen in the Python code below in Fig 3.1.1.

```
cap.set(cv2.CAP_PROP_BRIGHTNESS, 0.5)
```

Figure 3.1.1 Python code used to set the brightness of the camera module

Extending from the software settings of the camera, the physical height and angle of the camera were vital to capturing the maximum amount of information from the environment. The initial position of the camera was mounted vertically at 130mm above the mock lunar surface. However, in testing it was identified that with the Raspberry Pi camera's field of view of 48.8° in the vertical axis, this height was not optimal. At this height, for the robot to see objects at range closer than 180mm, the camera had to be tilted downwards significantly such that it was pointed at the surface. However, in doing so the camera was no longer able to see objects in the environment at a further range. Fig 3.1.2 shows this blind spot in the first iteration of the prototype. Conversely to this problem, if the camera was mounted too low to the lunar surface, then the field of view would be unable to reach the top of the 150mm tall obstacles when they were in close range. It was identified that having the camera at half of the height of these obstacles at 75mm would allow for the most broad view of them. It was angled very slightly downwards to favour a close-range view, while still capturing the farther reaches of the environment. Fig 3.1.3 shows the final dimensioned location of the camera sensor on the robot. This height also provided enough room underneath the camera for the sample collection subsystem to effectively capture the sample. There were still some limitations with the close-range view of the orange sample and blue rock at this height, however due to these objects being smaller in dimensions compared to the obstacle, it was possible to alleviate these issues in software. These alleviations are discussed further in section *3.3 Calculating Range and Bearing.*

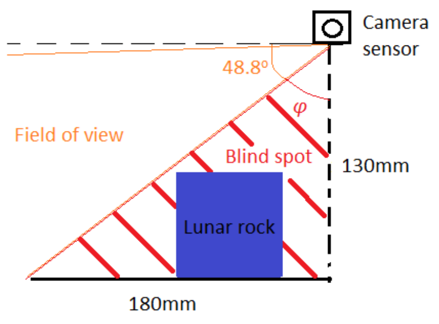


Figure 3.1.2 Blind spot of the initial camera configuration

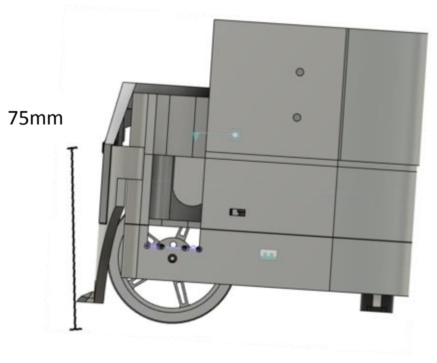


Figure 3.1.3 Final configuration of the camera sensor on the robot

3.2 Image Processing

Once the camera sensor had captured a frame of the environment, a series of image processing functions were applied to remove noise, mitigate false positives and false negatives, and ensure that only vital data from the frame was being analysed.

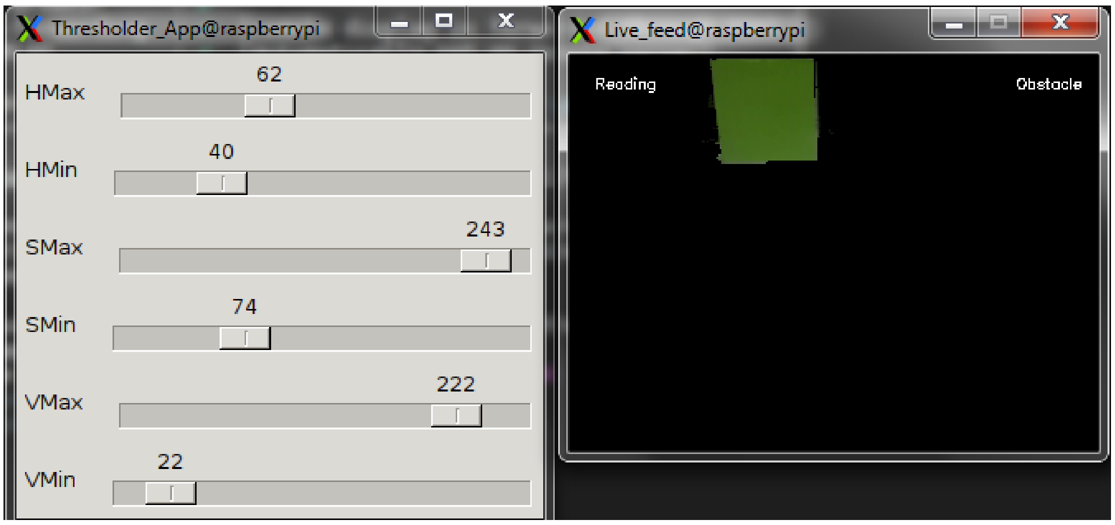
By default, when capturing images with the camera used, the images are in RGB. However, in initial testing, it was discovered that when using this colour space there were limitations with varying brightness and as such other colour spaces were investigated. The HSV colour space generally solved this issue of brightness, however, did have some limitations in detecting the walls. The CIELAB colour space was also researched as an alternative for detecting walls in the environment, however it was decided that the additional conversion of RGB to both HSV and CIELAB just to detect walls more accurately was not computationally viable. Therefore, HSV was the sole colour space used for the vision subsystem. The Python code used to convert the RGB frame to HSV can be seen below in Fig 3.2.1.

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)    #convert rgb to hsv
```

Figure 3.2.1 Python code used to convert the frame from RGB to HSV

3.2.1 Masks

A calibration program was created that uses a GUI with a live feed of the camera and slider bars to adjust a set of threshold HSV values. As the slider bars are adjusted, the live feed reacts and displays the resulting image after being compared with a bitwise AND to a mask of threshold ranges. This GUI can be seen in Figure 3.2.1 a) where it is being used to find the thresholds of an obstacle.



HMax 62
HMin 40
SMax 243
SMin 74
VMax 222
VMin 22

Figure 3.2.1 a) Mask calibration GUI

The calibration program saves the HSV minimum and maximum values to a local text file for each object. These text files are read at the initialisation of the main robot program, after which masks are created for each object. Figure 3.2.1 b) shows the unprocessed camera frame alongside each object mask compared with a bitwise AND to the frame.

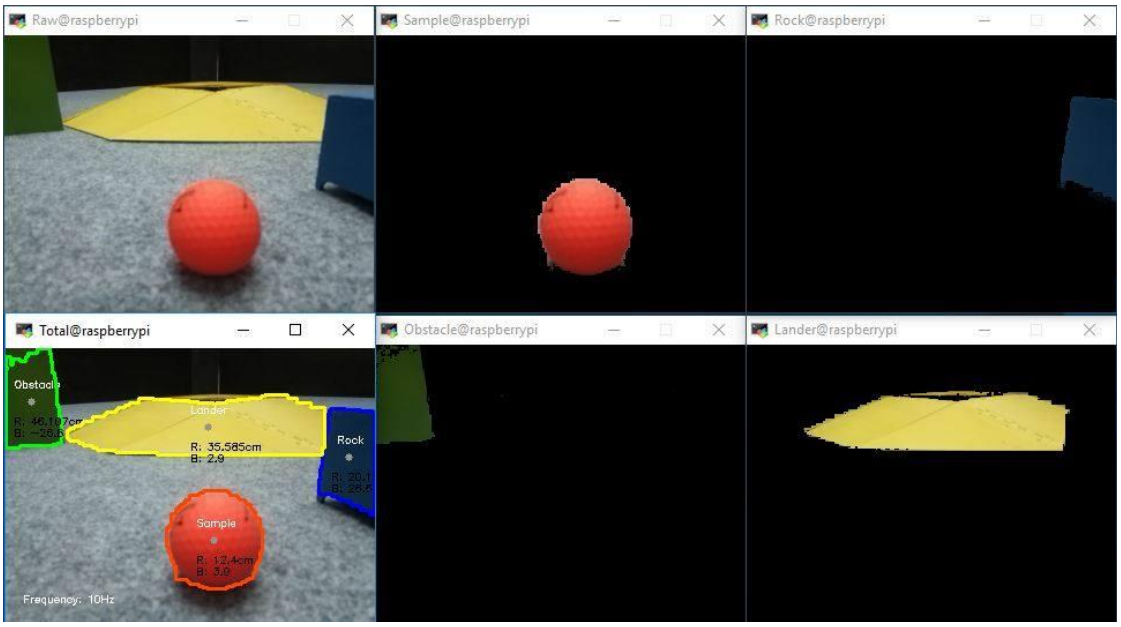


Figure 3.2.1 b) Video feed of segmented objects

In the experimental phase of prototyping it was realised that the hue of the orange sample wrapped around the bounds of the hue thresholds due to the nature of hue being radial. Given setting the threshold to allow hues between 0-179 would create multiple false positives, a bitwise OR was used in the case of the sample thresholds. This created a mask that had threshold hues ranging from a lower bound (around 170) to 179, as well as hues from 0 to an upper bound (around 10). Fig 3.2.1 c) illustrates the orange sample hue range. It should be noted that the degrees used in programming are half of the degrees labelled in the picture due to 8-bit numbers having an upper limit of 255.

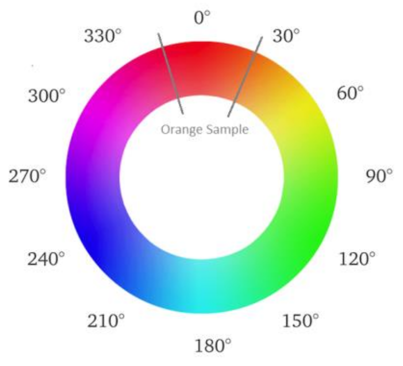


Figure 3.2.1 c) Colour wheel of orange sample hue.

3.2.2 Morphology

One of the first hurdles faced in the prototyping phase of the vision subsystem was the false positive of rocks. The carpet in the test environment was steel-grey in colour and when the frame was segmented with the rock HSV thresholds, very small areas of the carpet would return a false positive of a rock. To resolve this issue, image morphology was applied to the binary segmented image. Binary positive pixels that were not adjacent with a 5x5 pixel binary positive were eroded to return negative, after which the remaining positives were dilated, again by a 5x5 pixel area. The Python code that achieves this process can be seen below in Fig 3.2.2 b). This process of morphological opening resulted in a binary image with no noise, without the reduced size that eroding incurs. This process can be seen in Fig 3.2.2 a) where it is applied to the lander segmentation. Without morphological opening applied here, the vision system would have detected two separate areas and thus two landers due to the warped plane on the top of the lander leaving a gap. The resulting binary image with the top of the lander removed also allowed for more accurate calculation of distance, as the pixel height of the segmented object – the isolated side of the lander - was more easily relatable to the real-life dimensions of the lander. This calculation of distance is covered more in section *3.3 Calculating Range and Bearing.*

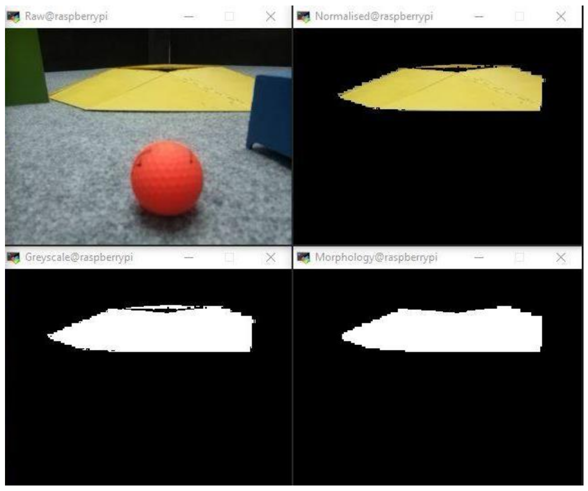


Figure 3.2.2 a) Morphological opening applied to the lander segmented image

```
kernel = np.ones((5,5),np.uint8)              #creates a 5x5 matrix of ones
erosion = cv2.erode(input_frame,kernel,iterations = 1)     #erodes with a 5x5 matrix
opened = cv2.dilate(erosion,kernel,iterations = 1)         #dilates with a 5x5 matrix
```

Figure 3.2.2 b) Python code used to apply morphological opening

3.2.3 Contours

Following morphological opening, the contours of each blob are found. The contours are the outlines of a shape with the same colour or intensity (OpenCV, n.d.) in this case, objects in frame that are binary positive. Finding the contours of the shape enable the calculation of the centroid of the object and thus the bearing of a segmented object, through the relation of the centroid to the robots known frame of reference. Additionally, the distance between the robot and object can be calculated based upon the height or width of the contour and its relation to the known constant size of the object. Furthermore, the contour data allows for calculations such as the ratio of the height to width of the object, which is useful when an object is extremely close to the camera and only partially visible in frame. The extraction of contours is done with the OpenCV *findContours* operation shown below in Fig 3.2.3 a).

```
cnts = cv2.findContours(opened, cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)      #grabs contours from cnts
```

Figure 3.2.3 a) Python code used to extract the contours from the frame

Subsequent to obtaining the contours, the moments of each contour can be extracted using the OpenCV *moments* function shown below in Fig 3.2.3 b). This function returns the moments of the contour up to the third order (OpenCV, n.d.). The moments are used to calculate the centroid of the contour. Fig 3.2.3 b) shows how the m_{10} moment is divided by the m_{00} moment or the area to calculate the centroid x coordinate. Likewise, the m_{01} moment is divided by m_{00} to give the centroid y coordinate.

```
for c in cnts:
       M = cv2.moments(c)  #Moments
       cX = int(M["m10"] / M["m00"])#Centre x-coord
       cY = int(M["m01"] / M["m00"])#Centre y-coord
```

Figure 3.2.3 b) Python code used to extract the moments from the contour and then calculate the centroid

Additionally, knowing the height or width of a contour enables the calculation of the distance between the camera and an object. The Python code in Fig 3.2.3 c) takes the coordinates of the extreme points and stores them into variables xmin, xmax, ymin and ymax. It then isolates these sets of coordinates to only include the x-coordinate for the extreme points on the x-axis, and only include the y-coordinate for the extreme points on the y-axis. The width and height of the contour are then stored in variables w and h respectively and converted to a floating point number as it is used for division later in section *3.3 Calculating Range and Bearing.*

```
for c in cnts:
        xmin = tuple(c[c[:,:,0].argmin()][0])      #Left most x-coord
        xmax = tuple(c[c[:,:,0].argmax()][0])      #Right most x-coord
        x1 = xmin[0]    #Take only the x-coordinate from the tuple
        x2 = xmax[0]    #Take only the x-coordinate from the tuple
        w = float(x2 - x1)  #Width of the contour
        ymin = tuple(c[c[:, :, 1].argmin()][0])     #Top most y-coord
        ymax = tuple(c[c[:, :, 1].argmax()][0])     #Bottom most y-coord
        y1 = ymin[1]    #Take only the y-coordinate from the tuple
        y2 = ymax[1]    #Take only the y-coordinate from the tuple
        h = float(y2 - y1)  #Height of the contour
```

Figure 3.2.3 c) Python code used to find the extreme points of the contour

3.3 Calculating Range and Bearing

Having obtained the centroid of a contour and thus the centroid of a segmented object, the bearing of the object to the robot's centre of frame can be calculated. The camera's known horizontal field of view is 62.2^{o} and as such it can be assumed that the central pixel on the frame captured by the camera is 0^{o} on the robot's frame of reference. Furthermore, it can be concluded that pixel 1 on the frame corresponds to -31.1^{o} and pixel 320 corresponds to +31.1^{o} from the centre of view given the resolution of the frame is 320x240 pixels. With this information, the following formula can be derived for calculating the object's bearing based upon it's centroid's x-coordinate in the frame.

$$Bearing = 31.1 \times \frac{x_{centroid}(pixels) - \left(\frac{Width_{frame}(pixels)}{2}\right)}{\frac{Width_{frame}(pixels)}{2}}$$

This formula's application is shown in Fig 3.3.1 where it is used to calculate the bearing of the sample from the robot's centre of view.

$$Bearing = 31.1 \times \frac{180 - \left(\frac{320}{2}\right)}{\frac{320)}{2}}$$

$$Bearing = 3.89^{o}$$

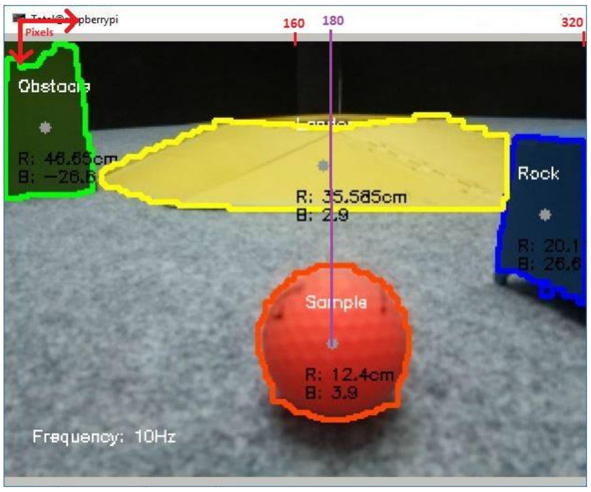
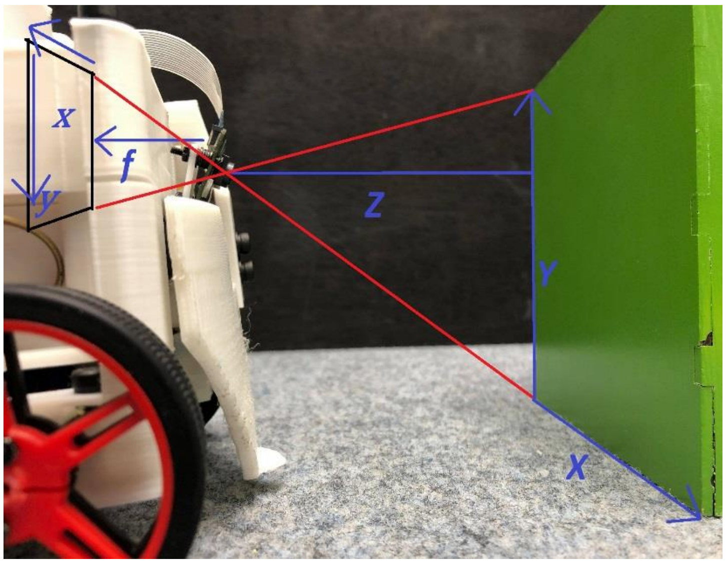


Figure 3.3.1 Bearing formula applied to the sample in frame

Similarly, an objects distance to the camera can be calculated using the relation between its pixel height or width in the frame, and its known real-life dimensions. This can only be done when the focal length of the camera is known. Fig 3.3.2 illustrates this relationship with the robot and an obstacle where the obstacle is projected through the camera onto an image plane.



$$\frac{Y}{Z} = \frac{y}{f}$$

$$\frac{X}{Z} = \frac{x}{f}$$

$$x = \frac{fX}{Z}$$

$$y = \frac{fY}{Z}$$

$$Z = \frac{fY}{y}$$

Figure 3.3.2 Relationship between real-life dimensions and image frame dimensions

Fig 3.3.2 is based on a Master Class by Professor Peter Corke and remodelled to use objects in this mock environment (Corke, n.d.).

It was decided that the height of objects would be used to calculate distance rather than the width. This was done as the lander was a wide object and was often only partially in view of the sensor. It also didn't have a consistent ratio of height to width, so this partial view was difficult to recalculate using software. This is discussed further in this section.

Examining the formula $Z = \frac{fY}{y}$ seen in Fig 3.3.2, the variable Z is the distance from the camera to the object, f is the focal length of the camera, Y is the real-dimension of the object, and the y is the height of the object in pixels on the image plane. With this information the formula can be rewritten as the following.

$$Distance(mm) = \frac{L_{focal}(mm) \times H_{object}(mm)}{H_{pixel}(pixels)}$$

However, this distance still needs to be related to the resolution of the image plane and the as such needs to be multiplied by the image plane resolution in pixels and divided by the height of the sensor in mm. The subsequently, the following distance equation is derived.

$$Distance(mm) = \frac{L_{focal}(mm) \times H_{object}(mm) \times H_{screen}(pixels)}{H_{pixel}(pixels) \times H_{sensor}(mm)}$$

In this equation, the focal length, height of the object, image plane resolution, and sensor height are all constant and known. As such, the only unknown variable is the height of the object in the image plane, which is obtained by the vision system when the contours are found. It should be noted that the height of the object varies between objects, where the height of the sample is 40mm, the rock 70mm, the obstacle 150mm and the lander 45mm.

In the case of objects being too close to the camera and as such partially out of view of the camera sensor, the distance formula will miscalculate the distance, as the height of the contour only includes part of the object. Examples of this scenario can be seen in Fig 3.3.3. To solve this issue, the ratio of the contour's height to width can be used to calculate how much of the object is out of view. Given all the objects excluding the yellow lander were either circular or cubical, their ratio of height to width was 1:1.

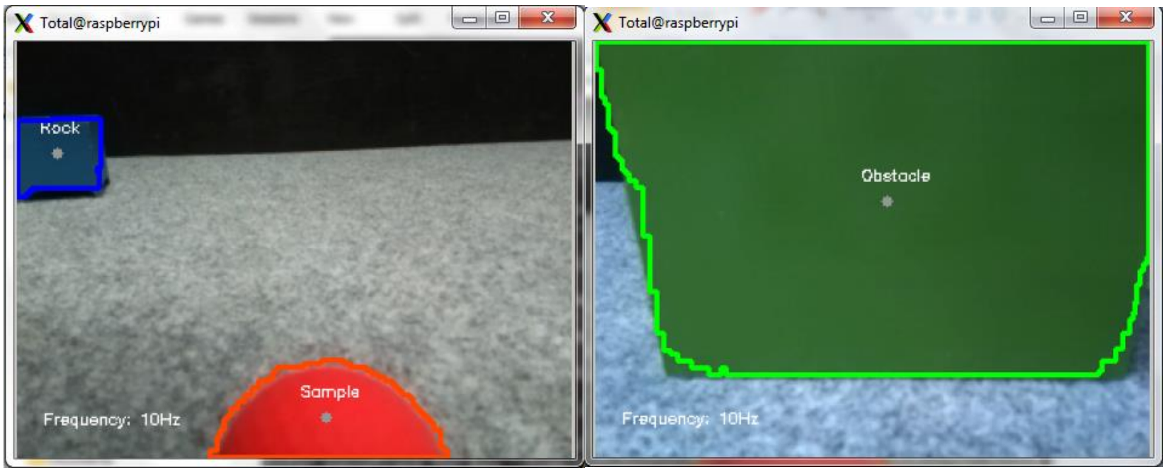


Figure 3.3.3 Objects that are partially out of frame due to being too close to the sensor

A variable in the software named *ratio* can be calculated as the height of the contour divided by the width. When this variable is less than 1, it is likely that the object is too close to the camera sensor. To ensure this is the case, the previously calculated extreme points can be used to check if the contour is at the bottom or top of the sensor's field of view. If both cases are met, the height can be revaluated as the previous height divided by the ratio variable. The Python code used to make these calculations can be seen below in Fig 3.3.4. Additionally, figure 3.3.5 visualises this concept - where it can be seen that the height of the contour is approximately half of the width. As such, the ratio equates to roughly ½. When the height of the contour is divided by this fraction, it doubles the original height of the contour and the actual height of the object is calculated.

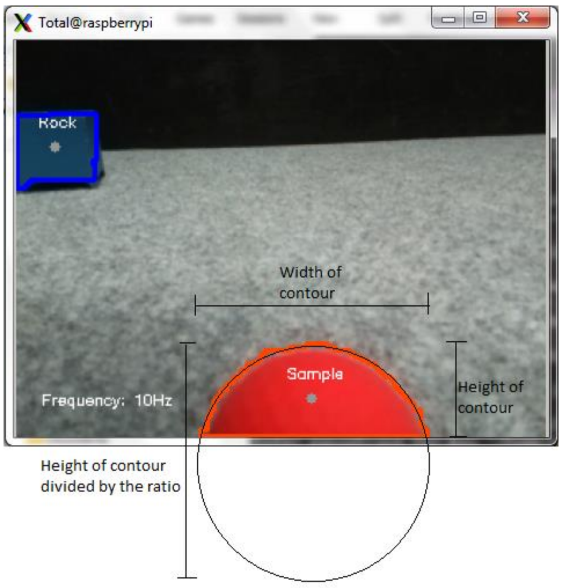


Figure 3.3.5 Python code used revaluate the height variable

```
ratio = h / w
if ((y2 > 235 or y1 < 5 ) and ratio < 0.9 and ratio != 0.0):
        h = h / ratio #Divide original height by the ratio of height to width
```
Figure 3.3.4 Python code used revaluate the height variable

Following the calculation of the range and bearing of each object in the sensor's frame, the data was stored into appropriate classes for the navigation subsystem to use. A class was created for each of the objects – samples, rocks, obstacles and lander. Each class stored an ID of the object, its distance, bearing, and centroid coordinates.

3.4 Sample Collection Tripwire

In addition to the camera sensor, the robot used a prototype sensor to perceive updates in the environment. A sensor was created to detect when the robot was in possession of a sample. It utilised a laser diode pointed at a light dependant resistor to act as a tripwire.

The Raspberry Pi and its System-on-Chip, the BCM2837B0 does not have analog read capabilities (Blum, 2016). Therefore, an RC circuit was created that utilises the time taken for a capacitor to charge to give this sensor an analog input. The circuit diagram for both the laser diode and the sensor can be seen in Fig 3.4.1.

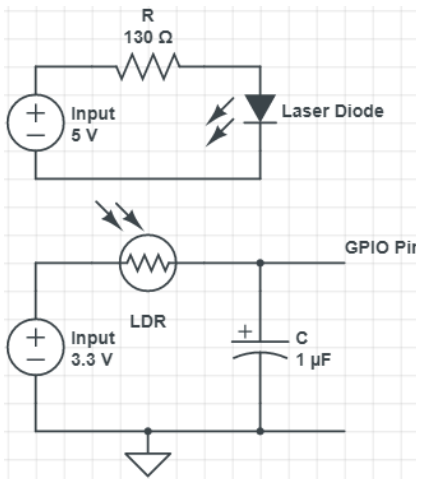


Figure 3.4.1 Circuit diagram of the laser diode output and LDR input

The basic premise for this sensor is as the light on the photoresistor decreases, it will take longer for current to get through the circuit and as a result take longer to charge the capacitor. Essentially, when the sample is captured by the robot and prevents the laser diode from emitting light directly on the photoresistor, the capacitor will take longer to charge. This time measurement corresponds to the analog reading of the sensor. A function was created in Python that first sets the output of the photoresistor as low, then counts the time taken for it to reach the threshold of high. This can be seen in Fig 3.4.2

```python
def laser():
    reading = 0                                     #Initialise reading as 0
    GPIO.setup(PHOTOCELL, GPIO.OUT)                 #Set Photoresistor pin as output
    GPIO.output(PHOTOCELL, GPIO.LOW)                #Set Photoresistor pin to 0
    time.sleep(0.1)
    GPIO.setup(PHOTOCELL, GPIO.IN)                  #Set Photoresistor pin as input
    while (GPIO.input(PHOTOCELL) == GPIO.LOW):      #While the input is low
        reading += 1                                #Increase the count
    return reading
```

Figure 3.4.2 Python function used to analog read the tripwire

This was used in the integrated code to update the status of the robot. If the tripwire was tripped, a sample had been collected and as such the next step for the robot was to return to the lander. The Python code used to update this status can be seen in Fig 3.4.3 below where it updates a variable named *captured*.

```python
if (laser() > LASER_THRESHOLD):
    global captured
    captured = 1
```

Figure 3.4.3 Python statement to update when the sample was collected

4.0 Results

The final product - an integration of mobility, sample collection, vision and navigation subsystems - was largely successful against the challenge requirements. The prototype was able to move off the lunar lander into the mock environment, and search the environment for targets. The prototype started off by locating the nearest blue rock and flipping it over to reveal an orange sample. It would then capture this sample and return it to the lander. Following the flip of a rock, a state was changed in the software where the robot would treat the blue rocks as obstacles rather than targets, avoiding them while navigating the environment looking for open air samples and returning them to the lander. The prototype collected two samples in the 5-minute period, capturing and returning a third at 6 minutes. This was done while avoiding both green obstacles, and the blue rocks following the rock flip sequence. The final TRL3 prototype can be seen in Fig 4.0.1.

The prototype was within budget for both the final design and the testing budget allocated, costing a total of $194. A copy of the budget can be found in the appendix. The electronics fit the requirements and did not exceed the voltage restriction of 60V. The robot was under the weight restriction of 1.3kg, weighing in at a total of 700g. It did not however fit into the size constraints. Additional wheels were added to the chassis to give it more traction and greater ability to drive up the lander. The addition of these wheels made the robot roughly 1cm too wide to fit in the 150mm CubeSat size restriction.

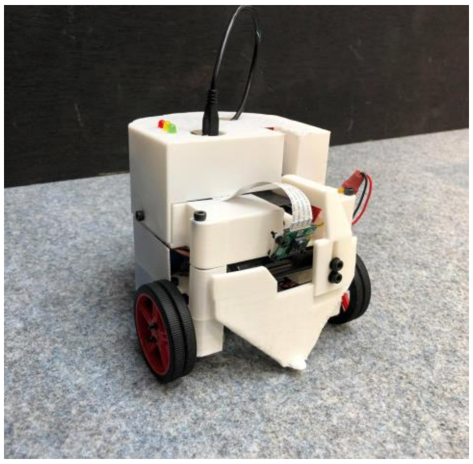


Figure 4.0.1 The final TRL 3 prototype

The result of the vision subsystem was a series of clearly segmented and labelled objects within the robot's field of view. These objects were all labelled over the top of a live video feed of the camera sensor. Through accurate thresholding with the threshold GUI and image morphology, there were no false positives of objects in the vision system. Through consistent settings of the camera brightness, the system was able to segment and distinguish all objects in the mock environment. Even when objects were partially out of view from the camera sensor, their range was still able to be calculated accurately.

Additionally, the stored objects in the classes were easily obtainable by the navigation subsystem to then determine the best path for the robot.

Further development of the system could prove critical to Yanada320's eventual goal of landing a robot on the moon to collect and test real soil samples. While proven effective for this prototype, the rudimentary system has much room for improvement as it mainly focuses on large blobs of colour in the mock environment. Image processing to identify more complex shapes and varied colour can still be implemented for a more realistic application.

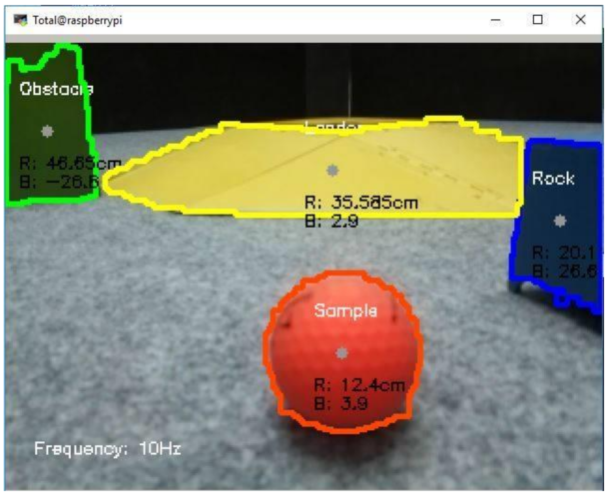


Figure 4.0.2 Screenshot of live video feed of vision system

There were minor issues with the speed at which the vision system ran. During the prototype demonstration, functions to display the image and object contours were commented out to increase speed of the integrated program. For the final integrated program, the vision system did not segment the walls as the navigation subsystem had no reactive planning for them. Not segmenting the walls did increase the run speed, and while the vision system ran mostly around 10Hz, it did sometimes dip to speeds of around 8Hz when there were multiple objects in the field of view. Additionally, due to numerous “sleep” commands in the navigation subsystem, there were pauses in the vision system. As such, the input frame would pause while the robot was navigating, and the vision system would not be able to update object class values. Similarly, the measurement of time used to obtain an analog reading of the light dependant resistor when detecting the sample caused slight delay. While the sensor was counting it held the system locked. Both delays led to lag throughout the entire integrated system.

To alleviate speed issues, processes such as multithreading could be investigated. This would enable the vision system to simultaneously segment samples, rocks, obstacles and the lander – as opposed to the current method of doing one after another, with each added object slowing the system.

Furthermore, changing the buffer size in the camera capture settings has potential to increase speed within the vision system. It was discovered after the prototype demonstration that the camera capture stores a buffer of frames, and when called, uses the last frame in the buffer. However, when the buffer is full the camera slows down and is unable to capture frames at the same rate.

While it did not seem necessary given the accuracy of the vision, camera capture settings like white balance, contrast, gain and saturation were not experimented with. Testing and analysis of changing these settings could lead to a more accurate segmentation process. This could potentially eliminate the need for morphological opening, which in turn could increase the speed of the system.

Given the opportunity to take on this challenge again, there would be some changes made. The process of multithreading as a potential solution to speed lags in the vision system would be studied for its viability. Additionally, the camera capture settings mentioned above would be tested to see image processing like morphology is even necessary in the mock environment where objects are mostly a distinct colour.

A lot of time was spent helping other members of the group with their subsystems, and as a result the vision system was still underdeveloped. While the whole integrated product was the most vital aspect of the challenge, it would have been advantageous to the vision system if less time was spent on other subsystems as a result of group members not committing to the challenge and as such their subsystems.

Conclusion

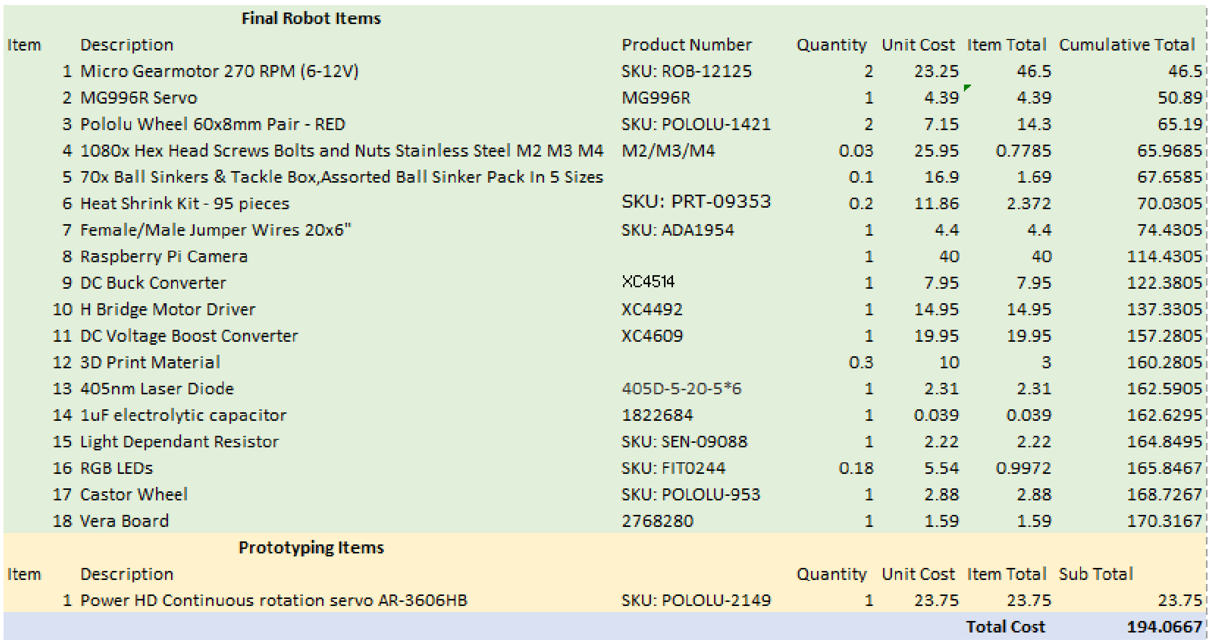
Yanada320 aims to eventually land a rover on the moon that collects soil samples and tests them for minerals and water. To prove that a mission like this is viable, the start-up company approached the QUT EGB320 undergraduate students to design and build a TRL3 prototype. This prototype was demonstrated to potential sponsors in a mock lunar environment. The aim of this prototype was to find and collect as many orange samples in a 5-minute time period as possible, while avoiding green obstacles that represent past lunar rovers. The students of EGB320 took on this challenge with each member of the four-member group developing a single subsystem within the design requirements, with the goal to integrate each subsystem cohesively.

This report detailed the vision subsystem of the prototype, demonstrating how sensors were used with a Raspberry Pi to give the other subsystems perception of the environment so that the robot could best navigate it. The process of capturing live images was detailed, along with the method of image processing. Through use of the HSV colour space and a threshold calibration GUI, different objects of interest in the environment were segmented to be analysed. In analysis, the range and bearing of each object was calculated and stored in software classes for the other subsystems to use. The vision subsystem was effective, but a little slow and unoptimized.

In demonstration, the prototype proved successful, but still had room for improvement. Two samples were identified, captured and returned in the 5-minute timeframe. The integrated solution was within budget, electronic and weight restrictions. However, it was just outside of the size restriction.

Overall, the TRL3 prototype validated future investment into Yanada320's mission. It proved that the rudimentary systems could be integrated with one another to form a cohesive unit. Further development into each subsystem has potential to constitute a robot that eventually lands on the moon to collect and test soil samples.

Appendix



		Final Robot Items				
Item	Description	Product Number	Quantity	Unit Cost	Item Total	Cumulative Total
1	Micro Gearmotor 270 RPM (6-12V)	SKU: ROB-12125	2	23.25	46.5	46.5
2	MG996R Servo	MG996R	1	4.39	4.39	50.89
3	Pololu Wheel 60x8mm Pair - RED	SKU: POLOLU-1421	2	7.15	14.3	65.19
4	1080x Hex Head Screws Bolts and Nuts Stainless Steel M2 M3 M4	M2/M3/M4	0.03	25.95	0.7785	65.9685
5	70x Ball Sinkers & Tackle Box,Assorted Ball Sinker Pack In 5 Sizes		0.1	16.9	1.69	67.6585
6	Heat Shrink Kit - 95 pieces	SKU: PRT-09353	0.2	11.86	2.372	70.0305
7	Female/Male Jumper Wires 20x6"	SKU: ADA1954	1	4.4	4.4	74.4305
8	Raspberry Pi Camera		1	40	40	114.4305
9	DC Buck Converter	XC4514	1	7.95	7.95	122.3805
10	H Bridge Motor Driver	XC4492	1	14.95	14.95	137.3305
11	DC Voltage Boost Converter	XC4609	1	19.95	19.95	157.2805
12	3D Print Material		0.3	10	3	160.2805
13	405nm Laser Diode	405D-5-20-5*6	1	2.31	2.31	162.5905
14	1uF electrolytic capacitor	1822684	1	0.039	0.039	162.6295
15	Light Dependant Resistor	SKU: SEN-09088	1	2.22	2.22	164.8495
16	RGB LEDs	SKU: FIT0244	0.18	5.54	0.9972	165.8467
17	Castor Wheel	SKU: POLOLU-953	1	2.88	2.88	168.7267
18	Vera Board	2768280	1	1.59	1.59	170.3167
	Prototyping Items					
Item	Description		Quantity	Unit Cost	Item Total	Sub Total
1	Power HD Continuous rotation servo AR-3606HB	SKU: POLOLU-2149	1	23.75	23.75	23.75
					Total Cost	**194.0667**

References

Blum, R., 2016. *Using Analog Sensors with the Raspberry Pi.* [Online]
Available at: https://www.informit.com/articles/article.aspx?p=2491680
[Accessed 28 10 2020].

Corke, P. P., n.d. *MasterClass: How Images are Formed.* [Online]
Available at: https://robotacademy.net.au/masterclass/how-images-are-formed/
[Accessed 28 10 2020].

Edwards, M., 2019. *EGB320 RoboCup SSL Kicking Technical Challenge.* [Online]
Available at: https://www.youtube.com/watch?v=06Vd30tHAdU&ab_channel=MargauxEdwards
[Accessed 28 10 2020].

MasterClass, 2020. *Hue, Saturation, Value: How to Use HSV Color Model in Photography.* [Online]
Available at: https://www.masterclass.com/articles/how-to-use-hsv-color-model-in-photography#what-is-hue
[Accessed 27 10 2020].

OpenCV, n.d. *About.* [Online]
Available at: https://opencv.org/about/
[Accessed 28 10 2020].

OpenCV, n.d. *Contours : Getting Started.* [Online]
Available at:
https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html#:~:text=Contours%20can%20be%20explained%20simply,better%20accuracy%2C%20use%20binary%20images.\
[Accessed 28 10 2020].

OpenCV, n.d. *Structural Analysis and Shape Descriptors.* [Online]
Available at:
https://docs.opencv.org/master/d3/dc0/group__imgproc__shape.html#ga556a180f43cab22649c23ada36a8a139
[Accessed 28 10 2020].

Photo Review, n.d. *Colour Spaces Explained.* [Online]
Available at: https://www.photoreview.com.au/tips/outputting/colour-spaces-explained/
[Accessed 27 10 2020].