Udacity: Search and Sample Return Report

Shane Reynolds

January 28, 2018

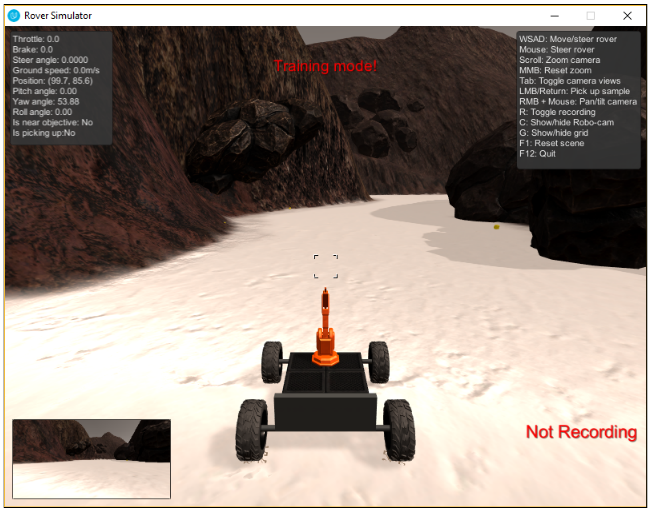
Contents

1 Introduction & Background

A simplistic and wide reaching definition of a robot is a machine which performs a task with some level of autonomy. In this context, robotic systems are appealing as they allow humans to avoid work that is considered dull, dirty and dangerous. This broad definition somewhat obfuscates the elements that make robotics work. Indeed, there is no clear consensus as to the mandatory subsystems which comprise a robotic system, however, there are some common features which can be observed across many existing robotics platforms, these include:

- **Perception systems**: systems which allow the robot to perceive the world around it
- **Decision making systems**: systems which allow the robot to decide a course of action given some information set
- **Actuation systems**: systems which allow the robot to physically interact with the world

This project serves as a short introduction to these three systems. Principally, it touches on elementary image processing concepts, and very briefly explores some basic decision making. The project is based on a simulated mobile robot operating in a simple terrain environment. The simulation is built in Unity and the main python script which gives the robot perception and decision making capabilities is called `perception.py` and `decision.py`, respectively. Finally, the interface between image processing, decision making algorithms, and the simulation is driven by SocketIO. A screen shot of the simulation can be seen in Figure 1 below.



Throttle: 0.0
Brake: 0.0
Steer angle: 0.0000
Ground speed: 0.0m/s
Position: (99.7, 85.6)
Pitch angle: 0.00
Yaw angle: 53.88
Roll angle: 0.00
Is near objective: No
Is picking up:No

Training mode!

Not Recording

Figure 1: A screenshot of the mobile robotic rover at a standstill in the simulated environment.

2 Methods and Implementation

2.1 Sensor Data

The robot perceives its world via sensors. The main sensor used in this project is the camera mounted to the front of the robot. Figure 2 shows an example of a single image captured from the rover's camera. The camera images are received approximately once every 27ms, or 36Hz.

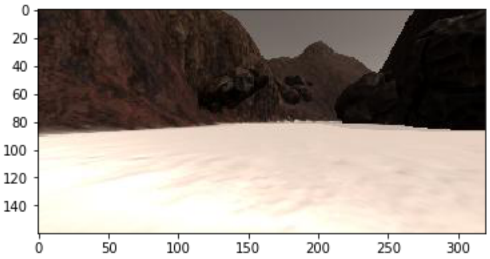


Figure 2: A single image of the simulated environment as shown from the robot's front mounted camera.

In addition to the camera data, the there are sensors which measure the rover's position and orientation in space. Position is given by simple Cartesian coordinates, (x, y), with reference to a fixed world frame. Orientation is given by roll, pitch, and yaw which are also with reference to the fixed world frame. Finally, throttle, brake, and steering angle of the rover are also provided. These values values are received at the same frequency as the images. The sensor data is used to help determine a course of action for the robot. This is achieved with two principal functions: `perception_step` and `decision_step`. Table 1 shows the different sensors data types, and a basic description of use of the captured data.

Table 1: A table which shows the different sensor data types, and their python format.

Sensor Data Type	**Variable**	**Description**
Image	`img`	The image which is captured by the rover's front mounted camera - it is an `np.array` of dimension $(160, 320, 3)$ and type `uint8`
Position	`pos`	Position of the rover with respect to a fixed world coordinate frame - it is a tuple of type `np.float`
Yaw	`yaw`	The yaw of the rover with respect to the world coordinate frame - it is of type `np.float` and is one part of the rover orientation description
Pitch	`pitch`	The pitch of the rover with respect to the world coordinate frame - it is of type `np.float` and is one part of the rover orientation description
Roll	`roll`	The roll of the rover with respect to the world coordinate frame - it is of type `np.float` and is one part of the rover orientation description
Current Velocity	`velocity`	The velocity of the rover - is of type `np.float` and is capped at $2\,\mathrm{m\,s^{-1}}$ in this project
Steering Angle	`steer`	The steering angle of the rover is determined by the angle of the front wheels with the centre line axis of the rover - is of type `np.float` and is bound between ±15deg
Throttle Value	`throttle`	Represents the value of locomotive force applied by the rover motors - is of type `np.float` and is binary in opearation in that throttle is either applied, or not
Brake Value	`brake`	Represents the applied force opposing motion (due to friction)

2.2 Image Processing

Image processing represents a large component of the project, and consists of multiple stages. It is encapsulated in the `perception_step` function, and is applied to each image captured by the rover's front mounted camera. The processing consists of three principal components: perspective transformation, segmentation, and translating the image to obtain a rover centric coordinate system. There is no fixed order in which the perspective transformation and segmentation steps need to occur, however, the transformation to a rover centric coordinate system can only be performed once the image has undergone a perspective transformation. The following subsections decribe each component in more detail.

2.2.1 Perspective Transformation

To make navigation easier to comprehend for observers of the rover behaviour, it is often useful to implement a perspective transforms. Certainly, the rover is capable of navigating via the front mounted RGB camera, however, it is often useful to transform this view into a top down perspective. The process for implementing such a transform involves the following four steps:

1. Define four (x, y) points in the source image (the image from the front mounted camera);
2. Identify the four (x, y) points, defined from the source image, in the destination image (the top down image);
3. Using the information from steps 1) and 2), create a transformation matrix which will transform the points from the source image to the destination image;
4. Apply the transformation matrix to captured image arrays to convert the images from the front mounted camera to a top down perspective.

Often is it useful to apply a grid of known size to both the source and destination images to help with the identification of points in each image. The grid applied in this instance was a 1m by 1m grid, and can be seen in Figure 3. The points used in both the source image and destination image to build the transformation matrix can be seen in Table 2, and the code snippet showing the set-up of the parameters used for the perspective transformation can be seen in Listing 1. Finally, once the transformation matrix is determined, then it can be applied to each received image to transform the perspective from the front mounted camera to the top down view - a demonstration of this can be seen in Figure 5. It is worth noting that it should be expected that the perspective transform will have regions of black present in transformed image - these black areas represent the rover's blind spots. The perspective transform function can be seen in Listing 2.

Table 2: The points determined from the destination and source images use to create the perspective transformation matrix.

Source	**Destination**
P1	P1
P1	P1
P1	P1
P1	P1

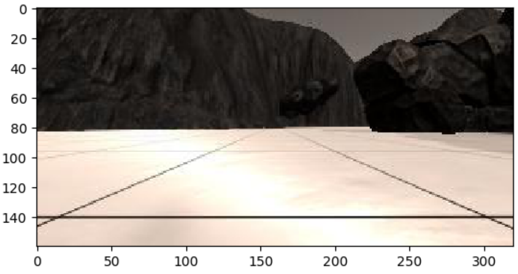


Figure 3: A 1x1 meter grid used to help establish points of reference.

Listing 1: Set up of the parameters which are used for the perspective transform.

```
img = Rover.img # takes the current camera image from the Rover and stores it in img
dst_size = 5 # what does this do
bottom_offset = 6 # what does this do

# Create the source array
source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]]) # specify the source array
destination = np.float32([[img.shape[1]/2 - dst_size, img.shape[0] - bottom_offset],
                          [img.shape[1]/2 + dst_size, img.shape[0] - bottom_offset],
                          [img.shape[1]/2 + dst_size, img.shape[0] - 2*dst_size - bottom_offset],
                          [img.shape[1]/2 - dst_size, img.shape[0] - 2*dst_size - bottom_offset],
                          ])
```

Listing 2: Code for the perspective transform function implementation.

```
def perspect_transform(img, src, dst):

    M = cv2.getPerspectiveTransform(src, dst)

    # Keep same size as input image
    warped = cv2.warpPerspective(img, M, (img.shape[1],img.shape[0]))

    return warped
```

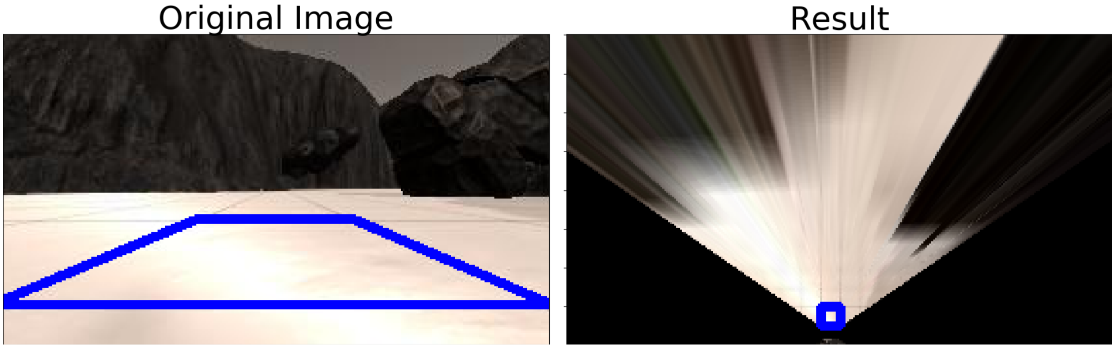
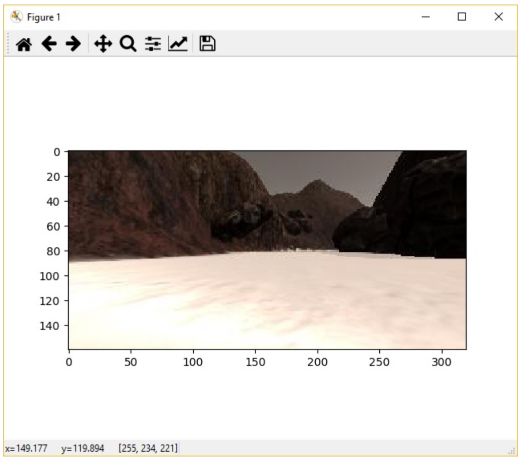


Figure 4: The original image from the rover's front mounted camera can be seen on the left, with the image after the perspective transform has been applied can be seen on the right.

2.2.2 Segmentation: Navigable Terrain & Obstacles

There are 3 different types of object that are of interest to the rover: navigable terrain, obstacles, and rock samples. A simple way to obtain the navigable terrain is to create a basic RGB filter. This works because it exploits the stark contrast between obstacles (which are dark), and navigable terrain (which is light). It must be noted that this technique would not generalise well, and would obviously perform best in environments with similar features to the simulation. Looking more closely at the filter, we see that we are able to extract both navigable terrain and obstacles with one instance of filtering since these two types of terrain are mutually exclusive - to put this simply: if the terrain is not navigable, then it must be an obstacle. The filtering itself is simplistic in its implementation, using an upper threshold for each of the R, G, and B values. To determine the cut-off threshold for each of the R, G, and B channels, most Operating Systems come with a basic image viewer which will provide information for each of the R, G, and B channels by pixel. A single frame of navigable terrain was loaded into an image viewer. Using this crude analysis values of 160 for each of the R, G, and B channels were determined as an appropriate threshold for navigable terrain. This process can be seen in Figure 5.



x=149.177 y=119.894 [255, 234, 221]

Figure 5: Using XXXX a crude analysis of the image was undertaken to determine the threshold values for R, G, and B values which represent navigable terrain.

Implementation of the colour threshold function can be seen in Listing 3. If a pixel in an image has R, G, and B values which are all greater than 160, the pixel is classified as navigable terrain. An example of the classification of navigable terrain can be seen in Figures 6 and 7.

Listing 3: Insert caption

```python
def color_thresh(img, rgb_thresh=(160, 160, 160)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "True"
    # where threshold was met
    above_thresh = (img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,1] > rgb_thresh[1]) \
                & (img[:,:,2] > rgb_thresh[2])
    # Index the array of zeros with the boolean array and set to 1
    color_select[above_thresh] = 1
    # Return the binary image
    return color_select
```

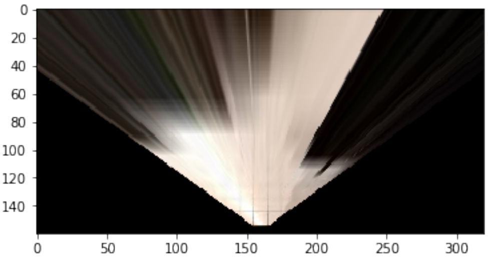


Figure 6: The perspective transformed image prior to the segmentation filter application.

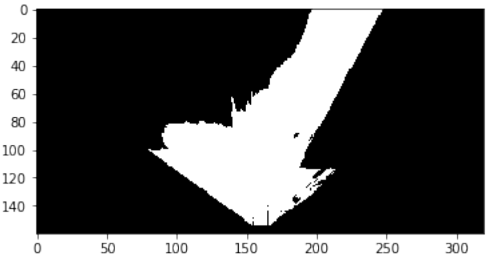


Figure 7: The perspective transformed image after the segmentation filter is applied - note that this is shown in grayscale.

Whilst this method can be applied successfully to determine navigable terrain, incorrect results maybe obtained when segmenting obstacles. This is caused if the filter is applied without consideration of the rover's blind spot, which is the area behind the rover's camera. In order to account for the rover's blind spot the segmentation filter needs to be applied prior to the perspective transform. This allows the capture of a conical region where the rover cannot see. This additional information, used in conjunction with a binary inverse of the navigable terrain array, provides for higher levels of accuracy when segmenting obstacles - implementation of this can be seen in Listing 4. The listing shows the extraction of the `cone` array, and the `rock` array (which is covered in the following subsection) and finally subtracts them from the `obstacle_temp` array to obtain the `obstacle` array.

An example of the full process for extracting navigable terrain and obstacles can be seen in Figures 8, 9, 10, 11, 12 and 13. The first figure in the sequence shows the image from the camera on the front of the rover, and Figure 9 shows the perspective transform. Figure 10 shows the navigable terrain after the segmentation filer is applied using a threshold value of 165 for R, G, and B channels. Figure 11 and 12 show the navigable terrain inverse, and the cone, respectively. Finally, Figure 13 shows the obstacles. Figures 10 and 13 are combined to provide a full map which shows navigable terrain in blue and obstacles in red. It must be noted that applying segmentation filters prior to perspective transforms is desirable since the code implementations would be simplified, however, in practice this structure degrades the quality overall segmentation resulting in poorer performance.

Listing 4: test

```
# Apply perspective transform
warped = perspect_transform(img, source , destination)

# Apply color threshold to identify navigable terrain/obstacles/rock samples
# Threshold image for terrain
nav = color_thresh(warped , (165, 165, 165))

# Threshold image for gold rocks
hsv_warped = cv2.cvtColor(warped , cv2.COLOR_RGB2HSV)
lower_thres = np.array([0,100,110])
upper_thres = np.array([70,255,255])
rock = cv2.inRange(hsv_warped , lower_thres , upper_thres).astype(bool).astype(int)

# Threshold image for obstacles
obstacles_temp = color_thresh(warped , (135, 135, 135))
obstacles_temp ˆ= 1

# Threshold image for cone
# Cone derivation
nav_thresh = color_thresh(img, (165, 165, 165))
obstacles_thresh = nav_thresh.copy()
obstacles_thresh ˆ= 1
cone = perspect_transform(obstacles_thresh , source , destination)
cone ˆ= 1
cone = np.logical_and(obstacles_temp , cone)

# Thresholded image for obstacles (with cone and rocks removed)
obstacles = obstacles_temp - cone - rock
```

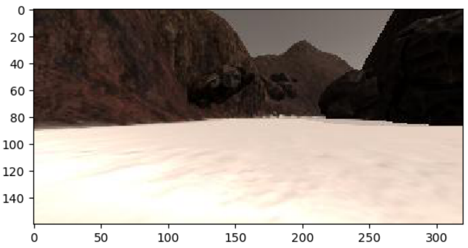


Figure 8: The original image taken from the camera mounted to the front of the rover.

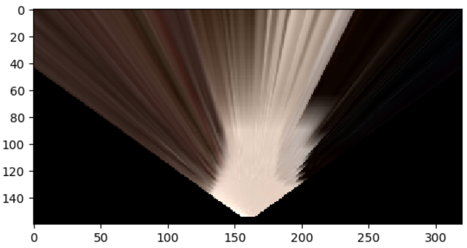


Figure 9: The image once it has undergone the perspective transform discussed in Section 2.2.1.

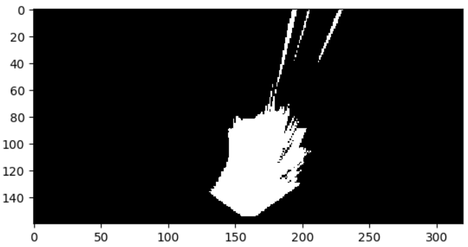


Figure 10: The resultant image of the segmentation for navigable terrain. Note that the white areas depict the navigable terrain areas.

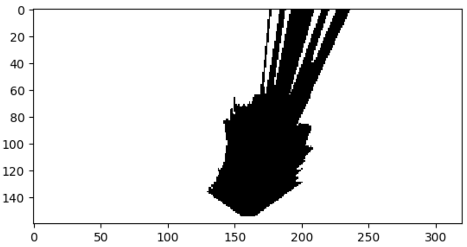


Figure 11: The resultant image for the inverse of the navigable terrain.

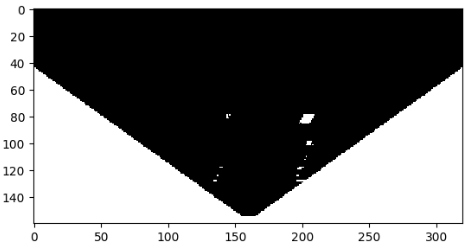


Figure 12: The resultant image for the derivation of rover blind spot.

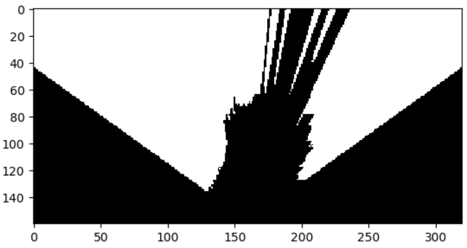


Figure 13: The final image which shows the rover obstacles. Note that the white areas depict the rover obstacles.

2.2.3 Segmentaiton: Rock Samples

Providing segmentation for rock samples saw unstable performance when filtering using RGB channels. This is due to the sample rocks holding different RGB values in darker regions, when compared to samples in lighter regions. Put simply, shadows affect the segmentation performance. An alternative colour representation known as Hue, Saturation, Value (HSV) is less susceptible to performance degradation from shadows and was employed in this instance. To determine the HSV values for the sample, a method was employed similar to that seen in Figure 5. Figures 14, and 15 show the process of segmenting the rock samples. Figure 16 shows the completed perspective transformed and segmented image of the navigable terrain, obstacles, and rocks.

Listing 5: text

```
# Apply perspective transform
warped = perspect_transform(img, source, destination)

# Threshold image for gold rocks
hsv_warped = cv2.cvtColor(warped, cv2.COLOR_RGB2HSV)
lower_thres = np.array([0,100,110])
upper_thres = np.array([70,255,255])
rock = cv2.inRange(hsv_warped, lower_thres, upper_thres).astype(bool).astype(int)
```

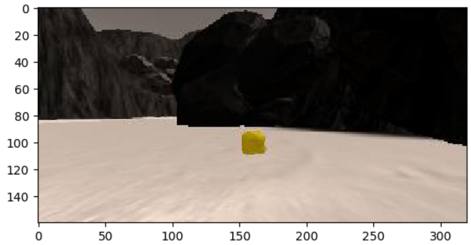


Figure 14: include a graphic of the original rock image

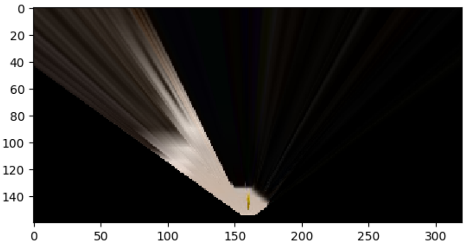


Figure 15: include a graphic of the filtered image - gray scale

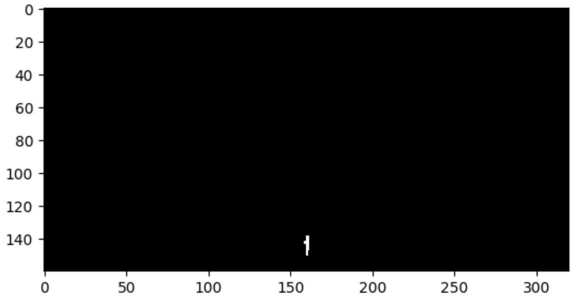


Figure 16: Include an image of the full image transformation with the rock and the blue and green

2.2.4 Rover Centric Coordinates

A coordinate frame is attached to the rover, in addition to establishing a fixed world coordinate frame. Assigning coordinate frames in this way allows the mathematical description of both position and orientation of the rover with respect to the world. A pictorial representation of the coordinate assignment can be seen in Figure 17.

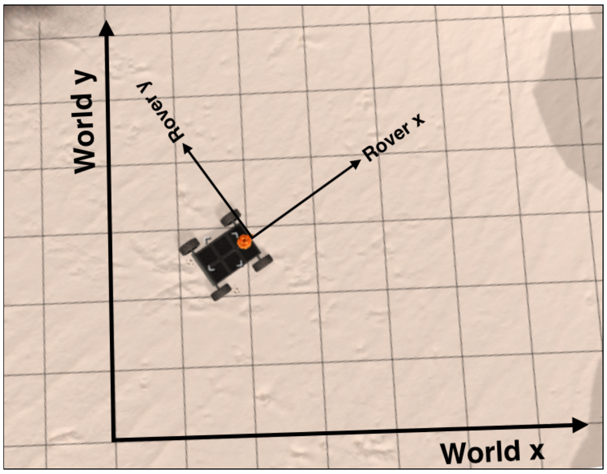


Figure 17: A coordinate frame is attached to, and moves with, the rover. This rover coordinate frame exists in the fixed world frame.

The image that is captured and processed by the XXXX function uses the source and destination points to map the recieved image into a 2D top down view point. Unfortunately, this function does not provide the correct orientation of the image with respect to the rover coordinate frame - this is shown in Figure 18. To ensure that the image is correctly positioned and oriented a coordinate frame transformation is performed, called XXXX, shown in Listing 6. This transformation translates the image so that it is positioned along the x-axis of the coordinate frame, and flips the x and y axes. The image which has been correctly positioned on the rover coordinate frame can be seen in Figure 19.

Listing 6: text

```
def rover_coords(binary_img):
    # Identify nonzero pixels
    ypos, xpos = binary_img.nonzero()
    # Calculate pixel positions with reference to the rover position being at the
    # center bottom of the image.
    x_pixel = -(ypos - binary_img.shape[0]).astype(np.float)
    y_pixel = -(xpos - binary_img.shape[1]/2 ).astype(np.float)
    return x_pixel, y_pixel
```

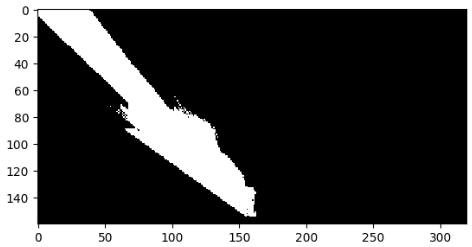


Figure 18: include a non-transformed image

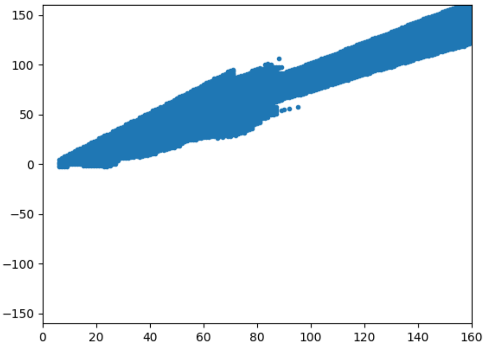


Figure 19: include an image of the transformed rover centric coord system

2.2.5 Mapping the the World Coordinate Frame

The rover navigates around the simulated environment generating information about the environment topology, using the segmentation techniques described in sections 2.2.2 and 2.2.3. Essentially, the rover captures information on navigable terrain and obstacles, and creates a map of the world. In order to do this, image information being processed needs to be transformed from the rover coordinate frame to the world coordinate frame. This requires further transformation. Listings 7, and 8 provide a function for 2D rotation, and 2D translation, respectively. Listing 9 uses both Listings 7, and 8 to perform the full transformation from rover to world coordinate frame. Figure 20 provides an example of the map that generated by the rover.

Listing 7: insert text

```
# A function which provides rotation in 2D
def rotate_pix(xpix, ypix, yaw):
    # Convert yaw to radians
    yaw_rad = yaw * np.pi / 180
    xpix_rotated = (xpix * np.cos(yaw_rad)) - (ypix * np.sin(yaw_rad))

    ypix_rotated = (xpix * np.sin(yaw_rad)) + (ypix * np.cos(yaw_rad))
    # Return the result
    return xpix_rotated, ypix_rotated
```

Listing 8: insert text

```
# A functions which provides a translation in 2D
def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # Apply a scaling and a translation
    xpix_translated = (xpix_rot / scale) + xpos
    ypix_translated = (ypix_rot / scale) + ypos
    # Return the result
    return xpix_translated, ypix_translated
```

Listing 9: insert text

```
# A function which performs the transformation from rover to world coordinates
def pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale):
    # Apply rotation
    xpix_rot, ypix_rot = rotate_pix(xpix, ypix, yaw)
    # Apply translation
    xpix_tran, ypix_tran = translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale)
    # Perform rotation, translation and clipping all at once
    x_pix_world = np.clip(np.int_(xpix_tran), 0, world_size - 1)
    y_pix_world = np.clip(np.int_(ypix_tran), 0, world_size - 1)
    # Return the result
    return x_pix_world, y_pix_world
```

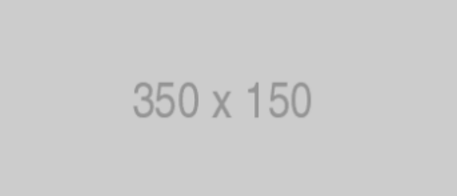


Figure 20: Show a picture of the mapped terrain, if possible

2.3 Autonomous Navigation

The rover is designed to move about the simulated environment autonomously. The main code executing the autonomous mode of navigation is encapsulated in `rover_drive.py`, shown in Appendix B. There are two key functions which are executed in this script: `perception.py` and `decision.py`. The `perception.py` function processes rover captured images, and provides an information set which is used by the `decision.py` function as input for autonomous navigation. Each of the functions are discussed in more detail below.

2.3.1 Perception Step

The perception step is applied to each image that is captured by the rover's front mounted camera. As previously mentioned, images are capture at approximately 36 Hz. The `perception_step` function can be seen, in full, in Appendix A. The function takes a single object oriented arguement, which details the current state of the rover, including the latest image captured by the camera. Using the image, navigable terrain, obstacles, and the prescence of rocks are determined and updated on the world map. Listing 10 shows the code which provides for world map updating - obstacles are on the red channel, navigable terrain is on the blue channel, and rocks are on the green channel.

Listing 10: insert text

```
# Add a small amount of colour to obstacles
    Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 2
    # Reset obstacle channel that rover currently sees as navigable terrain
    Rover.worldmap[nav_y_world, nav_x_world, 0] = 0

    # Update any rocks on the rock colour channel
    Rover.worldmap[rock_y_world, rock_x_world, 1] = 255

    # Reset anything on the navigable terrain channel that the rover currently sees as an obstacle
    Rover.worldmap[obstacle_y_world, obstacle_x_world, 2] = 0
    Rover.worldmap[nav_y_world, nav_x_world, 2] = 255
```

It is important to note that the segmentation of navigable terrain, and obstacles, in the current image can be in conflict with historically recorded areas on the world map. To put this simply, the segmentation of obstacles and navigable terrain is imperfect - this leads to differing results in images of the same area. To overcome this, when the red channel (obstacles) is being updated, all navigable terrain pixels in the current image are set to zero. Similarly, when the blue channel is being updated, all obstacles pixels in the current image are set to zero. This ensures that pixels in the world map are never simultaneously an obstacle and navigable terrain.

The main importance of the perception step is to take an image an to organise it into meaningful information so that decisions can be made using the decision step. Important parts are:

- determining the navigation angle based on the navagble terrain available
- determining if a rock can be seen, and switching the rover mode into one which readys the rover for rock collection

2.3.2 Decision Step

The `decision.py` function, shown in Appendix C, is applied after the current image has been processed, and the rover state updated. Principally, this function is concerned with rover navigation and focuses on obstacles avoidance, in addition to providing the rover with some capacity to negotiate dead ends in the environment. Of course, this is somewhat difficult to achieve reliably and consistently - often the rover becomes stuck on complex sections of the environment. Basic navigation decision making is undertaken by checking variables, such as velocity and navigable terrain, and changing the rover's state accordingly. The full logic flow diagram can be seen in Figure 21.

3 Results & Conclusion

4 Further Enhancements

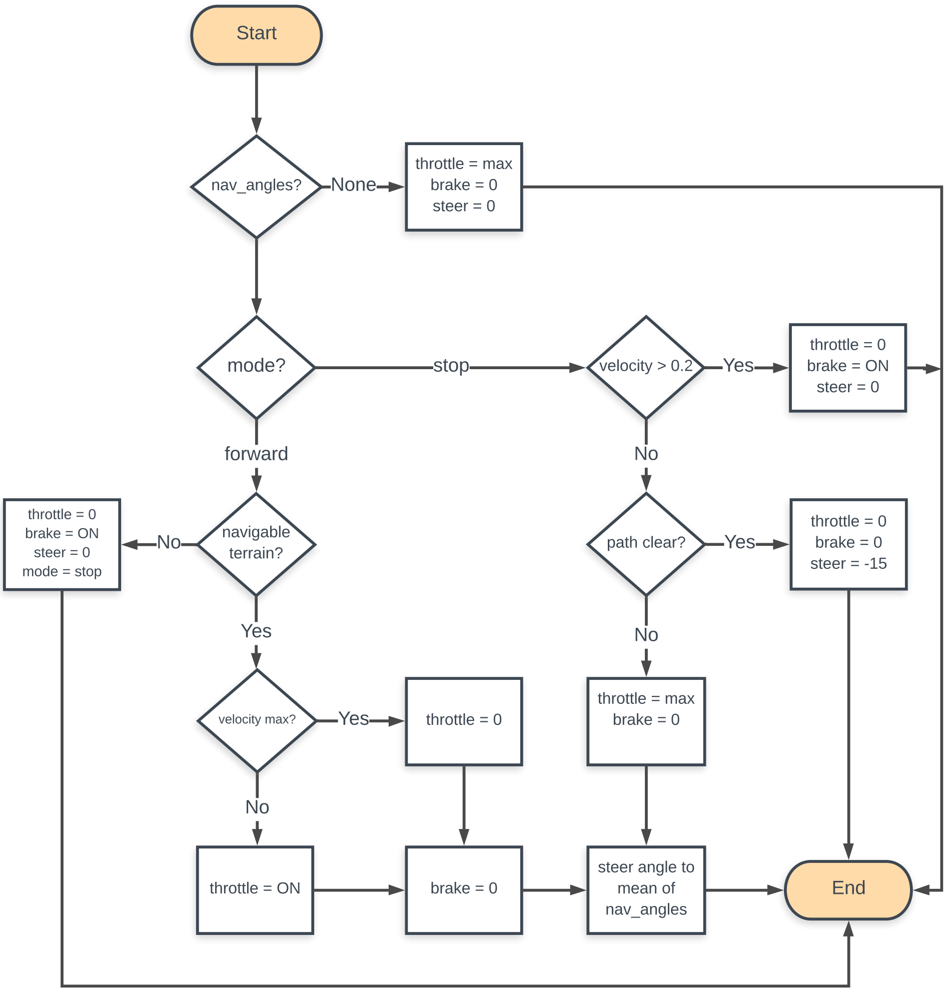


Figure 21: text