

Udacity: Search and Sample Return Report

Shane Reynolds

February 25, 2018

Contents

1	Introduction & Background	2
2	Methods and Implementation	3
2.1	Sensor Data	3
2.2	Image Processing	4
2.2.1	Perspective Transformation	4
2.2.2	Segmentation: Navigable Terrain & Obstacles	5
2.2.3	Segmentaiton: Rock Samples	9
2.2.4	Rover Centric Coordinates	10
2.2.5	Mapping the the World Coordinate Frame	11
2.3	Autonomous Navigation	12
2.3.1	Perception Step	12
2.3.2	Decision Step	13
3	Results & Conclusion	19
4	Further Enhancements	19
5	Appendix A	22
6	Appendix B	24
7	Appendix C	27

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 sub-systems 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.

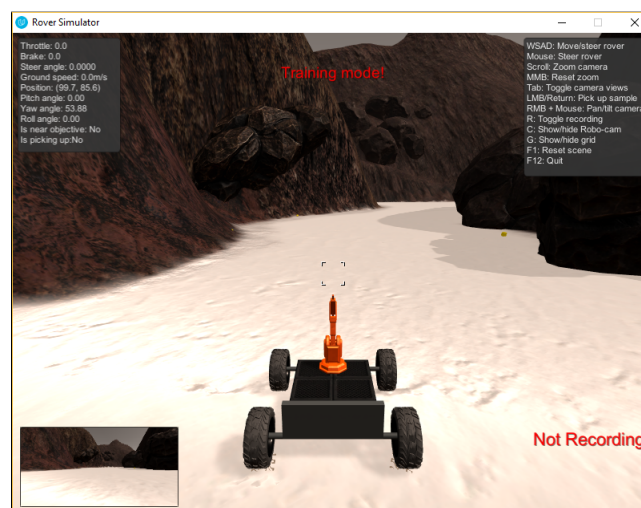


Figure 1: A screenshot of the mobile robotic rover at a standstill in the simulated environment.

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.

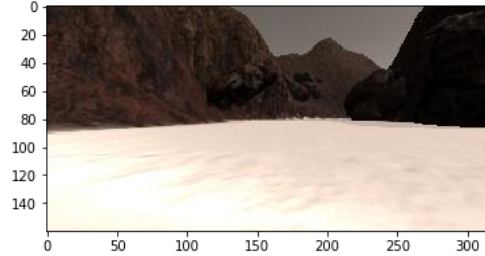


Figure 2: A single image of the simulated environment as shown from the robot's front mounted camera.

In addition to the camera data, 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 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	<code>img</code>	The image which is captured by the rover's front mounted camera - it is an <code>np.array</code> of dimension $(160, 320, 3)$ and type <code>uint8</code>
Position	<code>pos</code>	Position of the rover with respect to a fixed world coordinate frame - it is a tuple of type <code>np.float</code>
Yaw	<code>yaw</code>	The yaw of the rover with respect to the world coordinate frame - it is of type <code>np.float</code> and is one part of the rover orientation description
Pitch	<code>pitch</code>	The pitch of the rover with respect to the world coordinate frame - it is of type <code>np.float</code> and is one part of the rover orientation description
Roll	<code>roll</code>	The roll of the rover with respect to the world coordinate frame - it is of type <code>np.float</code> and is one part of the rover orientation description
Current Velocity	<code>velocity</code>	The velocity of the rover - is of type <code>np.float</code> and is capped at 2m s^{-1} in this project
Steering Angle	<code>steer</code>	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 <code>np.float</code> and is bound between $\pm 15\text{deg}$
Throttle Value	<code>throttle</code>	Represents the value of locomotive force applied by the rover motors - is of type <code>np.float</code> and is binary in operation in that throttle is either applied, or not
Brake Value	<code>brake</code>	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 describe 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

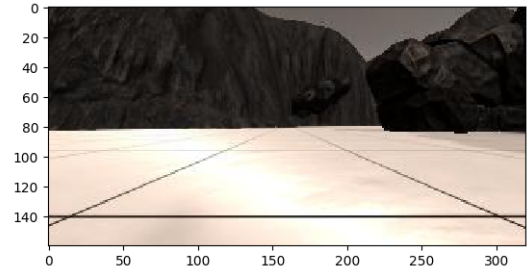


Figure 3: A 1x1 meter grid used to help establish points of reference.

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
```

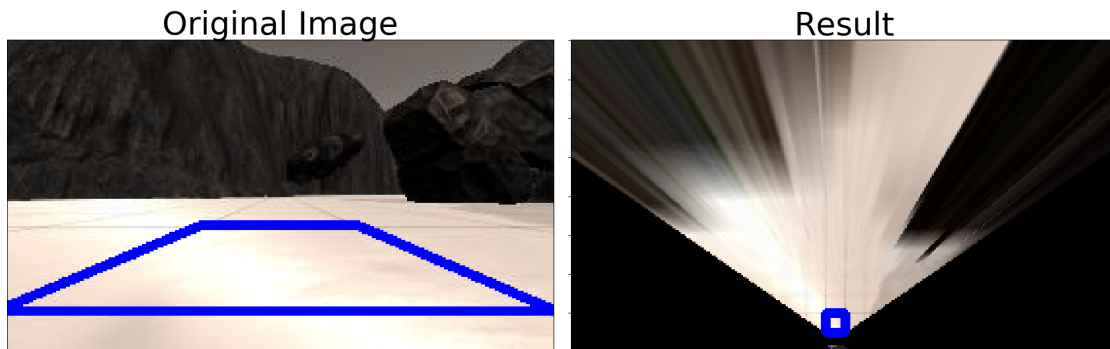


Figure 4: The original image from the rover's front mounted camera can be seen on the left, with the image after the perspective transform has been applied can be seen on the right.

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.

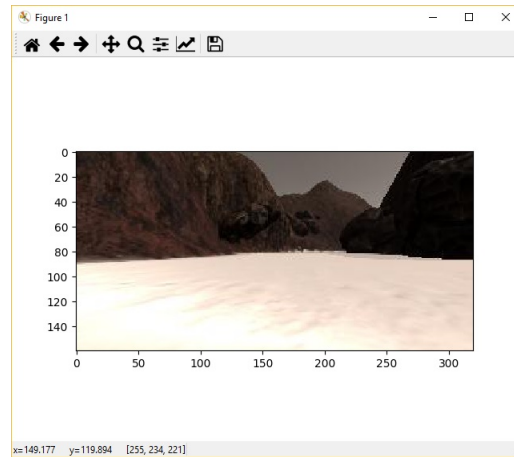


Figure 5: Using XXXX a crude analysis of the image was undertaken to determine the threshold values for R, G, and B values which represent navigable terrain.

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

```
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
```

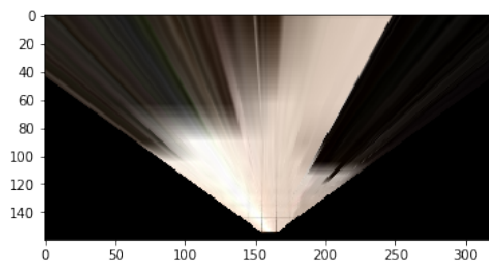


Figure 6: The perspective transformed image prior to the segmentation filter application.

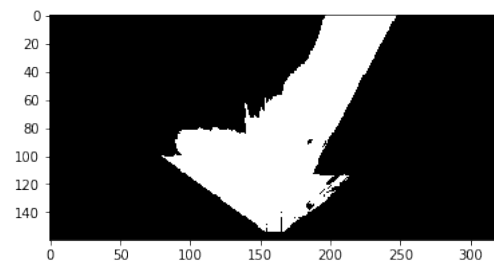


Figure 7: The perspective transformed image after the segmentation filter is applied - note that this is shown in grayscale.

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 filter 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 ^= 1

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

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

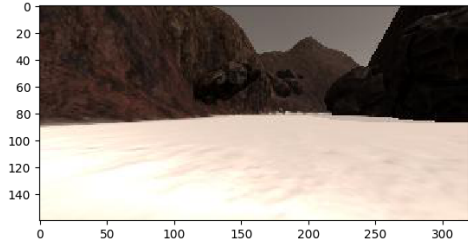


Figure 8: The original image taken from the camera mounted to the front of the rover.

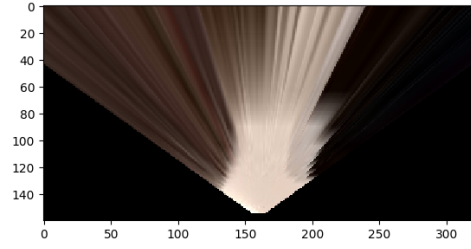


Figure 9: The image once it has undergone the perspective transform discussed in Section 2.2.1.

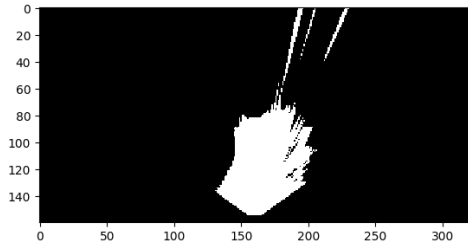


Figure 10: The resultant image of the segmentation for navigable terrain. Note that the white areas depict the navigable terrain areas.

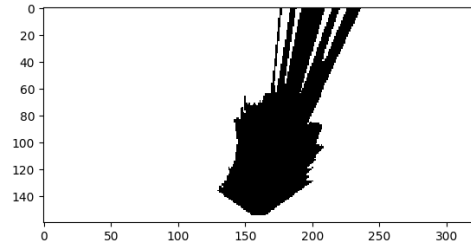


Figure 11: The resultant image for the inverse of the navigable terrain.

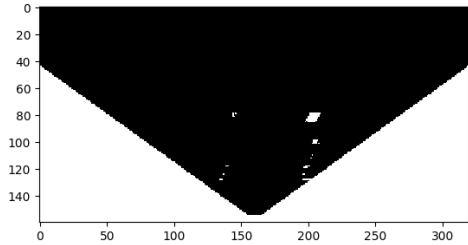


Figure 12: The resultant image for the derivation of rover blind spot.

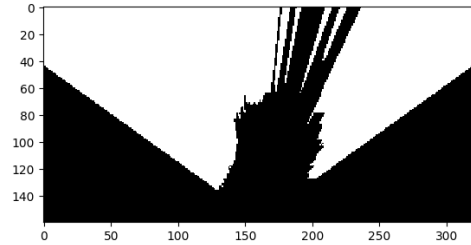


Figure 13: The final image which shows the rover obstacles. Note that the white areas depict the rover obstacles.

2.2.3 Segmentation: 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)
```

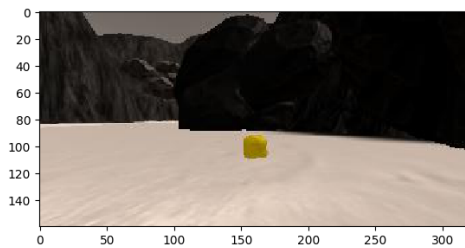


Figure 14: include a graphic of the original rock image

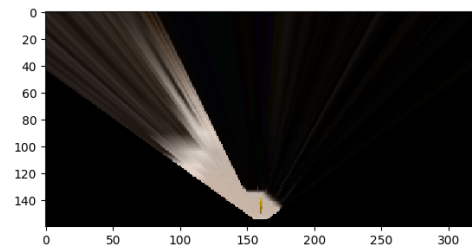


Figure 15: include a graphic of the filtered image - gray scale

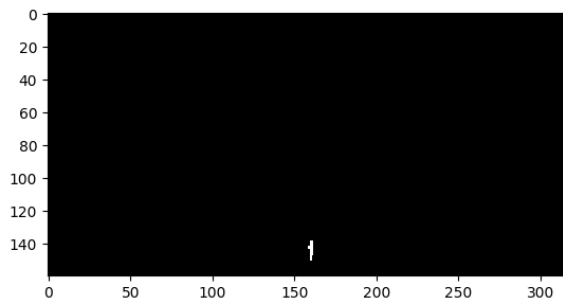


Figure 16: Include an image of the full image transformation with the rock and the blue and green

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.

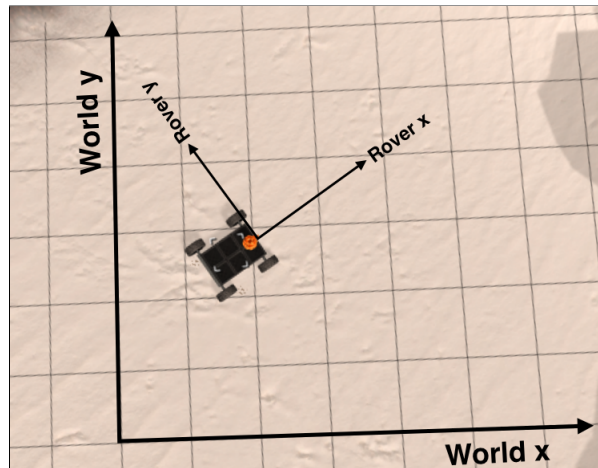


Figure 17: A coordinate frame is attached to, and moves with, the rover. This rover coordinate frame exists in the fixed world frame.

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
```

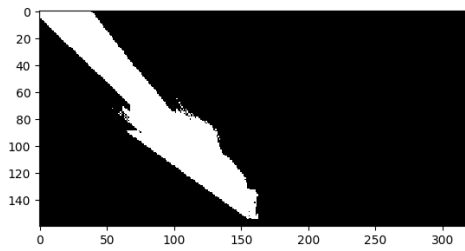


Figure 18: include a non-transformed image

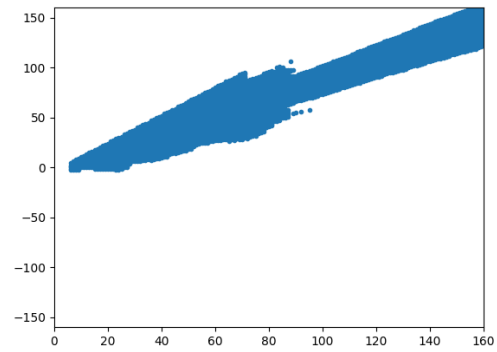


Figure 19: include an image of the transformed rover centric coord system

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
```

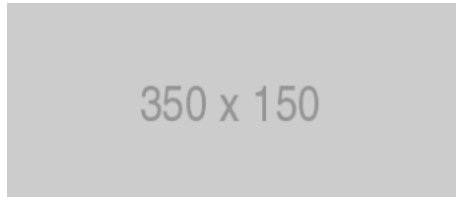


Figure 20: Show a picture of the mapped terrain, if possible

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 `drive_rover.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 captured at approximately 36 Hz. The `perception_step` function can be seen, in full, in Appendix A. The function takes a single object oriented argument, which details the current state of the rover, including the latest image captured by the camera. Using the image, navigable terrain, obstacles, and the presence 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 and 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 navigable terrain available
- determining if a rock can be seen, and switching the rover mode into one which readys the rover for rock collection

The perception step also helps to determine whether there is a rock present in the image, and if so will direct the rover into rocking mode which collects rocks.

Listing 11: i

```
f (sum(sum(rock)) > 20) and Rover.mode != "stuck":
    Rover.mode = "rocking"
    Rover.rock_spot_count = 0 # The rover will reset the rock spot counter
    distances, angles = to_polar_coords(xpix_rock, ypix_rock)
    Rover.rock_dists = distances
    Rover.rock_angles = angles
    # If the Rover loses track of the rock, the Rover still remains in the rocking
    # mode, however, will start to count down to switching back to forward mode
    # provided the Rover cannot relocate the rock
    elif Rover.mode == "rocking":
        Rover.rock_spot_count += 1
        Rover.rock_angles = 0
```

All navigation is done by taking navigation pixels determined by the segmentation function and converting them into angles using the polar conversion function

Listing 12: text

```
def to_polar_coords(x_pixel, y_pixel):
    # Convert (x_pixel, y_pixel) to (distance, angle)
    # in polar coordinates in rover space
    # Calculate distance to each pixel
    dist = np.sqrt(x_pixel**2 + y_pixel**2)
    # Calculate angle away from vertical for each pixel
    angles = np.arctan2(y_pixel, x_pixel)
    return dist, angles
```

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 and locate rocks. The decision step decides between 4 distinct Rover modes: forward, rocking, stop, and stuck.

The forward state code snippet is shown in Listing 11, and the flow diagram explaining the logic behind the operation is shown in Figure 21. When in the forward state the rover checks that there is navigable terrain in front of it. If there is sufficient navigable terrain, and the velocity is at the maximum velocity, then the rover will continue in this mode. If the rover is below maximum velocity,

then the throttle will be applied. If the rover has zero velocity, but has sufficient navigable terrain, then the clock for determining if the rover is stuck will start to count. If the stuck counter, which is stored in the object variable `Rover.vel`, reaches 400, then the rover will change state from `forward` to `stuck`. The timer for this state change was implemented to provide some hysteresis preventing the rover from rapid state switching. Note that if the Rover's velocity reaches above a threshold while the stuck counter is counting, then it is deemed that the rover has freed itself from the obstacle and the counter is reset.

The steering direction is determined by a weighted moving average of the present steering angle stored in `Rover.nav_angles`, with a weight of $1/3$, and the previous steering angle, with a weight of $2/3$. This provides a slight reduction in erratic corrections given extreme values in `Rover.nav_angles`. Erratic changes are further ameliorated by bounding the steering angle between -15° and 15° .

Listing 13: Code snippet for the operation of the forward state of the rover

```
# Check for Rover.mode status
if Rover.mode == 'forward':
    # If the Rover is no longer stuck, then
    if abs(Rover.vel) > 0.08:
        Rover.vel_count = 0 # Reset the stuck counter
    # Check the extent of navigable terrain
    if len(Rover.nav_angles) >= Rover.stop_forward:
        # If mode is forward, navigable terrain looks good
        # and velocity is below max, then throttle
        if Rover.vel < Rover.max_vel:
            # Set throttle value to throttle setting
            Rover.throttle = Rover.throttle_set
            # If the rover is stuck
            if abs(Rover.vel) < 0.05:
                # Some hysteresis so the robot doesn't switch modes rapidly
                Rover.vel_count += 1
                if Rover.vel_count > 400:
                    Rover.vel_count = 0 # Reset the zero velocity count
                    Rover.stuck_count = 0 # Start the stuck counter
                    Rover.throttle = 0
                    Rover.brake = 0
                    Rover.steer = 0
                    Rover.mode = 'stuck'
            else: # Else coast
                Rover.throttle = 0
                Rover.brake = 0

        # Set steering to average angle clipped to the range +/- 15
        # Note that the steering angle has been smoothed by using a
        # two time period weighted average
        Rover.steer =
            (2*Rover.steer +
             np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15))/3

# If there's a lack of navigable terrain pixels then go to 'stop' mode
elif len(Rover.nav_angles) < Rover.stop_forward:
    # Set mode to "stop" and hit the brakes!
    Rover.throttle = 0
    # Set brake to stored brake value
    Rover.brake = Rover.brake_set
    Rover.steer = 0
    Rover.mode = 'stop'
```

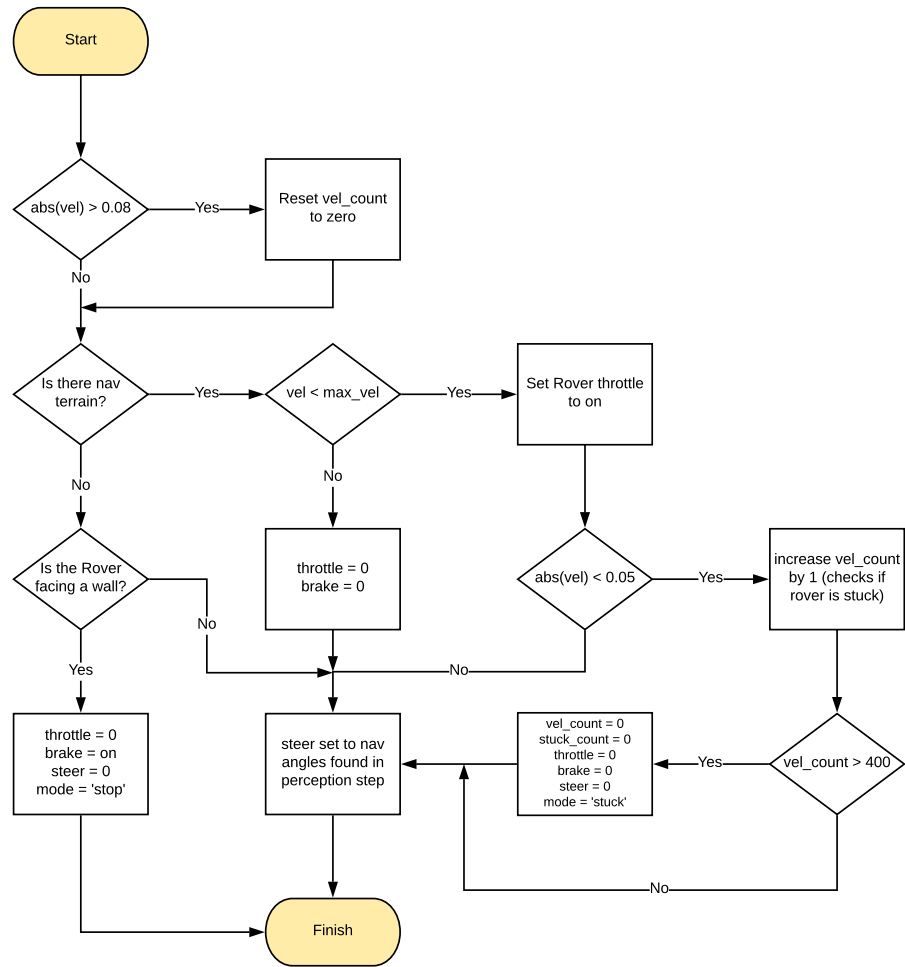


Figure 21: Flow diagram showing the operation of the rover when it is in the forward state of operation.

The rocking state is used when the rover detects a rock, or the current image has rock like characteristics. The change from the current state to the rocking state is determined by the code snippet shown in Listing 12. This state change will occur if the rover is not in the stuck state, and there are more than 20 pixels in the current image which have the characteristics of a rock. Note that if the rover is in rocking mode, but loses track of the rock (that is the rock is not persistent in the image) then a counter starts which will eventually switch the rover back to the forward state. The timer is employed to stop rapid state switching. The rocking state code snippet is shown in Listing 12, and the flow diagram explaining the logic behind the operation is shown in Figure 22.

Listing 14: Code snippet showing how the rover changes to the rocking state

```
# If the rover detects a rock, and it is not in stuck mode, the Rover will
# switch to the rocking mode (i.e. the mode used to search for rocks)
if (sum(sum(rock)) > 20) and Rover.mode != "stuck":
    Rover.mode = "rocking"
    Rover.rock_spot_count = 0 # The rover will reset the rock spot counter
    distances, angles = to_polar_coords(xpix_rock, ypix_rock)
    Rover.rock_dists = distances
    Rover.rock_angles = angles
# If the Rover loses track of the rock, the Rover still remains in the rocking
# mode, however, will start to count down to switching back to forward mode
# provided the Rover cannot relocate the rock
elif Rover.mode == "rocking":
    Rover.rock_spot_count += 1
    Rover.rock_angles = 0
```

Listing 15: Code snippet showing rover operation in the rocking state

```
# If the rover is in the vicinity of a rock - this is set in perception
elif Rover.mode == 'rocking':
    if abs(Rover.vel) > 0.08:
        Rover.vel_count = 0 # Reset the stuck counter
    if Rover.rock_spot_count > 200:
        Rover.mode = "forward"
    else:
        if Rover.vel < 0.5:
            Rover.throttle = 0.1
            Rover.brake = 0
            if abs(Rover.vel) < 0.05:
                # Some hysteresis so the robot doesn't switch modes rapidly
                Rover.vel_count += 1
                if Rover.vel_count > 400:
                    Rover.vel_count = 0 # Reset the zero velocity count
                    Rover.stuck_count = 0 # Start the stuck counter
                    Rover.throttle = 0
                    Rover.brake = 0
                    Rover.steer = 0
                    Rover.mode = 'stuck'
            elif Rover.vel >= 0.5:
                Rover.throttle = 0

        # This will direct the rover towards the rock
        Rover.steer = np.clip(np.mean(Rover.rock_angles * 180/np.pi), -15, 15)

        # This will stop the Rover if it is near a rock sample
        # Note this is a condition for picking up samples
        if Rover.near_sample:
            Rover.throttle = 0
            Rover.brake = Rover.brake_set
```

When in the rocking state, the rover will first check if timer if the rock image has been lost. If the timer is greater than 200, then the rover will switch back to the forward state. The rover will then check if it is moving, and start a stuck counter similar to that used in forward mode if

the velocity is below a certain threshold. If the counter `Rover.vel_count` goes above 400 then the rover will switch to the stuck state. Provided the rover remains in the rocking state, the rover will approach the rock using `Rover.rock_angles` to steer, at a maximum velocity of 0.5. Note that if the rover loses track of the rock image, then the steering angle is set to 0. Once the rover is near the sample, the `Rover.near_sample` flag is tripped, and the rover will stop to pick up the rock, after which the state will change to forward.

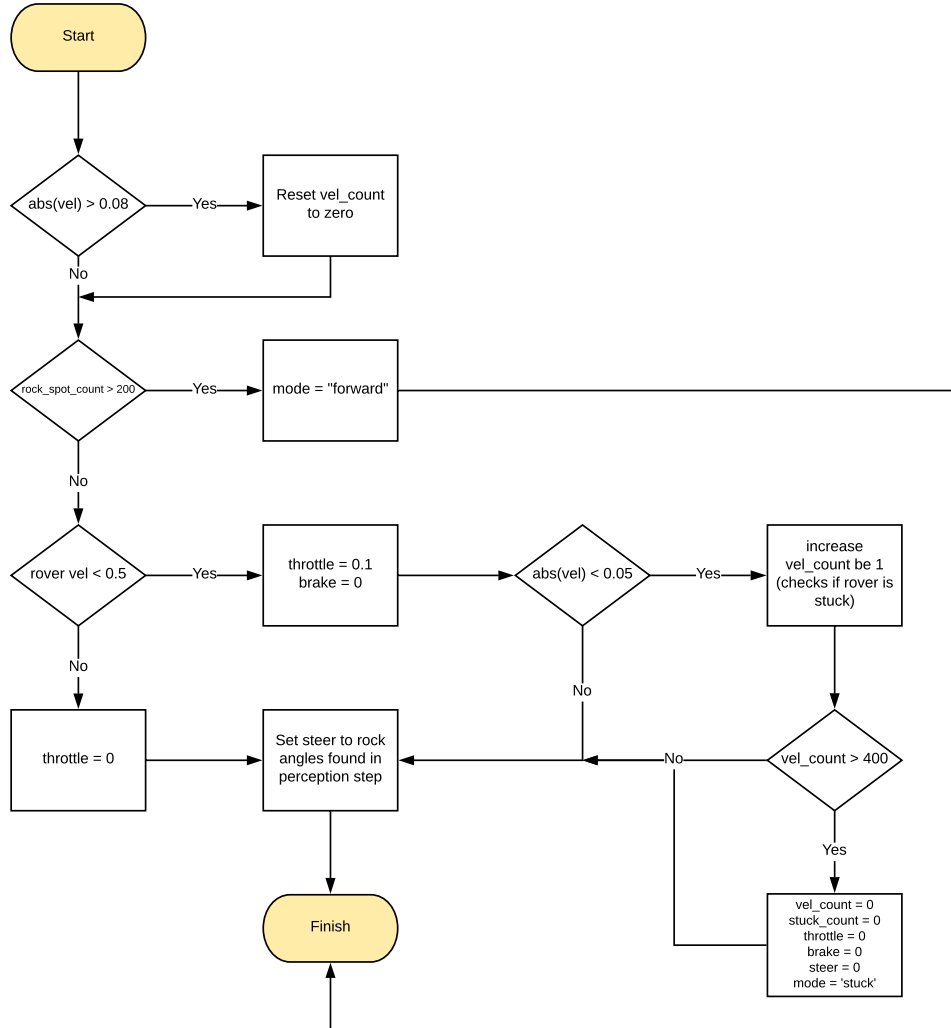


Figure 22: Flow diagram showing the operation of the rover when it is in the rocking state of operation.

The stuck state code snippet can be seen in Listing 16, and the flow diagram explaining the logic behind the operation is shown in Figure 23. When the rover is in the stuck state, a counter is increased by one, and the throttle is set to -0.1, with the steering angle set to 0 so the rover will reverse away from the obstacle. The rover will persist in this mode until the counter reaches 300, at which time the rover will change states to the forward state.

Listing 16: Code snippet showing the operation of the rover in the stuck state

```
# This is the Rover stuck mode - that is if the wheels are caught on the terrain
elif Rover.mode == 'stuck':
    Rover.stuck_count += 1 # Add a count to the hysteresis
    Rover.steer = 0
    Rover.throttle = -0.1 # This will reverse the rover with 0 steer angle
    Rover.brake = 0

# This behaviour, once triggered only needs to occur for 300 processed frames
# which is less than 10 seconds
if Rover.stuck_count > 300:
    Rover.stuck_count = 0 # Reset the stuck counter
    Rover.steer = 0 # Reset the Rover steering
    Rover.mode = 'forward' # Hopefully the rover is unstuck
```

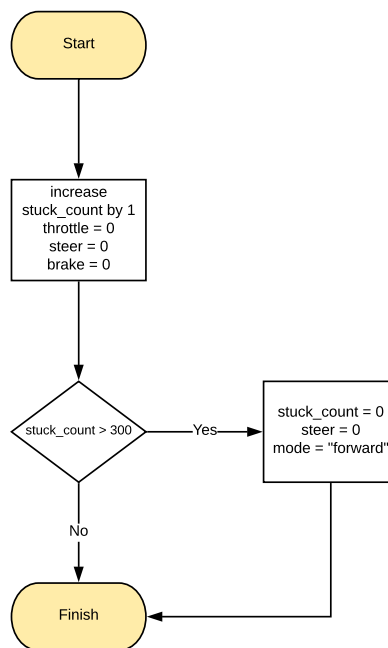


Figure 23: Flow diagram showing the operation of the rover when it is in the stuck state of operation.

Finally, the stop state code snippet can be seen in Listing 17, and the flow diagram explaining the logic behind the operation is shown in Figure 24. When in the stop state, the rover will first check if the velocity is greater than 0.2, in which case it will apply the brake to bring the rover to a rapid stop. The rover will then check the navigable terrain. If there is no navigable terrain, then the rover will turn counter clockwise on the spot, until sufficient navigable terrain is visible. At this point the rover will recommence in the forward state.

Listing 17: Code snippet showing the operation of the rover in the stop state

```
# If we're already in "stop" mode then make different decisions
elif Rover.mode == 'stop':
    # If we're in stop mode but still moving keep braking
    if Rover.vel > 0.2:
        Rover.throttle = 0
        Rover.brake = Rover.brake_set
        Rover.steer = 0
    # If we're not moving (vel < 0.2) then do something else
    elif Rover.vel <= 0.2:
        # Now we're stopped and we have vision data to see if there's a path forward
        if len(Rover.nav_angles) < Rover.go_forward:
            Rover.throttle = 0
            # Release the brake to allow turning
            Rover.brake = 0
            # Turn range is +/- 15 degrees, when stopped the next line
            # will induce 4-wheel turning
            Rover.steer = -15 # Could be more clever here about which way to turn
        # If we're stopped but see sufficient navigable terrain in front then go!
        if len(Rover.nav_angles) >= Rover.go_forward:
            # Set throttle back to stored value
            Rover.throttle = Rover.throttle_set
            # Release the brake
            Rover.brake = 0
            # Set steer to mean angle
            Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)
            Rover.mode = 'forward'
```

3 Results & Conclusion

The code implementations seen in the perception step and the decision step allowed the rover to navigate the map in almost all circumstances. The full set of code implementations can be found at the following Github repository:

<https://github.com/skreynolds/RoboND-Rover-Project.git>

There were only a few minor instances where the rover would become stuck with no chance to recover. These minor situations would occur when the rover would have the undercarriage balanced on a rock providing no contact between the rover wheels and the ground, or places that the rover itself would become embedded into the obstacle. A video of the rover navigating successfully through the terrain can be seen at the following YouTube link:

<https://youtu.be/-h0lGIr9gxk>

The implementation was able to map well over the required 40% of the map, and the fidelity of the mapping never fell below 80% - most of the time it was above 90%. Further, the rover was able to autonomously locate rock samples and pick them up in most instances.

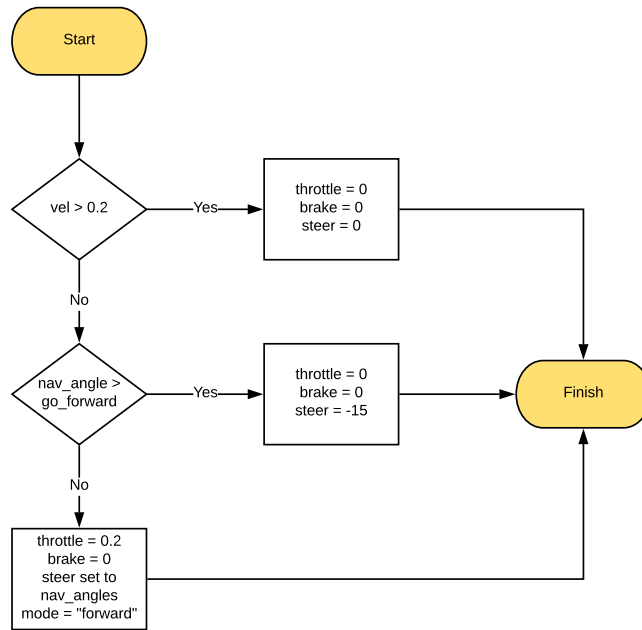


Figure 24: Flow diagram showing the operation of the rover when it is in the stuck state of operation.

4 Further Enhancements

Enhancements could be made by instructing the rover to return back to its original position when it has found all of the rock samples present in the map. This would require the rover to take note of the Cartesian coordinates at which it was positioned when the simulation started. Further, the number of samples collected would also need to be monitored. When the rover had collected 6 samples (this is the number of samples in each simulation), then the rover would switch to a newly created state which would see the rover return to its original position. Further, adjustments to the rover's forward state of navigation may help it to avoid areas which cause it to become irrecoverably stuck.

5 Appendix A

```
1 import numpy as np
2 import cv2
3
4 # Identify pixels above the threshold
5 # Threshold of RGB > 160 does a nice job of identifying ground pixels only
6 def color_thresh(img, rgb_thresh=(160, 160, 160)):
7     # Create an array of zeros same xy size as img, but single channel
8     color_select = np.zeros_like(img[:, :, 0])
9     # Require that each pixel be above all three threshold values in RGB
10    # above_thresh will now contain a boolean array with "True"
11    # where threshold was met
12    above_thresh = (img[:, :, 0] > rgb_thresh[0]) \
13        & (img[:, :, 1] > rgb_thresh[1]) \
14        & (img[:, :, 2] > rgb_thresh[2])
15    # Index the array of zeros with the boolean array and set to 1
16    color_select[above_thresh] = 1
17    # Return the binary image
18    return color_select
19
20 # Define a function to convert from image coords to rover coords
21 def rover_coords(binary_img):
22     # Identify nonzero pixels
23     ypos, xpos = binary_img.nonzero()
24     # Calculate pixel positions with reference to the rover position being at the
25     # center bottom of the image.
26     x_pixel = -(ypos - binary_img.shape[0]).astype(np. float)
27     y_pixel = -(xpos - binary_img.shape[1]/2 ).astype(np. float)
28     return x_pixel, y_pixel
29
30
31 # Define a function to convert to radial coords in rover space
32 def to_polar_coords(x_pixel, y_pixel):
33     # Convert (x_pixel, y_pixel) to (distance, angle)
34     # in polar coordinates in rover space
35     # Calculate distance to each pixel
36     dist = np.sqrt(x_pixel**2 + y_pixel**2)
37     # Calculate angle away from vertical for each pixel
38     angles = np.arctan2(y_pixel, x_pixel)
39     return dist, angles
40
41 # Define a function to map rover space pixels to world space
42 def rotate_pix(xpix, ypix, yaw):
43     # Convert yaw to radians
44     yaw_rad = yaw * np.pi / 180
45     xpix_rotated = (xpix * np.cos(yaw_rad)) - (ypix * np.sin(yaw_rad))
46
47     ypix_rotated = (xpix * np.sin(yaw_rad)) + (ypix * np.cos(yaw_rad))
48     # Return the result
49     return xpix_rotated, ypix_rotated
50
51 def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
52     # Apply a scaling and a translation
53     xpix_translated = (xpix_rot / scale) + xpos
54     ypix_translated = (ypix_rot / scale) + ypos
55     # Return the result
56     return xpix_translated, ypix_translated
57
58
59 # Define a function to apply rotation and translation (and clipping)
60 # Once you define the two functions above this function should work
61 def pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale):
62     # Apply rotation
63     xpix_rot, ypix_rot = rotate_pix(xpix, ypix, yaw)
64     # Apply translation
65     xpix_tran, ypix_tran = translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale)
66     # Perform rotation, translation and clipping all at once
67     x_pix_world = np.clip(np.int(xpix_tran), 0, world_size - 1)
68     y_pix_world = np.clip(np.int(ypix_tran), 0, world_size - 1)
69     # Return the result
70     return x_pix_world, y_pix_world
71
72 # Define a function to perform a perspective transform
73 def perspect_transform(img, src, dst):
74
75     M = cv2.getPerspectiveTransform(src, dst)
76     warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))# keep same size as input image
77
78     return warped
79
80
81 # Apply the above functions in succession and update the Rover state accordingly
82 def perception_step(Rover):
83     # Perform perception steps to update Rover()
84     # TODO:
85     # NOTE: camera image is coming to you in Rover.img
86
87     # 1) Define source and destination points for perspective transform
88     img = Rover.img
89     dst_size = 5
90     bottom_offset = 6
91     source = np.float32([[14, 140], [301, 140], [200, 96], [118, 96]])
92     destination = np.float32([[img.shape[1]/2 - dst_size, img.shape[0] - bottom_offset],
93         [img.shape[1]/2 + dst_size, img.shape[0] - bottom_offset],
94         [img.shape[1]/2 + dst_size, img.shape[0] - 2*dst_size - bottom_offset],
95         [img.shape[1]/2 - dst_size, img.shape[0] - 2*dst_size - bottom_offset],
96         ])
97
98     # 2) Apply perspective transform
99     warped = perspect_transform(img, source, destination)
100
101     # 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
102     # Threshold image for terrain
103     nav = color_thresh(warped, (165, 165, 165))
```

```

104
105 # Threshold image for gold rocks
106 hsv_warped = cv2.cvtColor(warped, cv2.COLOR_RGB2HSV)
107 lower_thres = np.array([0,120,120])
108 upper_thres = np.array([40,255,255])
109 rock = cv2.inRange(hsv_warped, lower_thres, upper_thres).astype( bool).astype( int)
110
111 # Threshold image for obstacles
112 obstacles_temp = color_thresh(warped, (135, 135, 135))
113 obstacles_temp ^= 1
114
115 # Threshold image for cone
116 # Cone derivation
117 nav_thresh = color_thresh(img, (165, 165, 165))
118 obstacles_thresh = nav_thresh.copy()
119 obstacles_thresh ^= 1
120 cone = perspect_transform(obstacles_thresh, source, destination)
121 cone ^= 1
122 cone = np.logical_and(obstacles_temp, cone)
123
124 # Thresholded image for obstacles (with cone and rocks removed)
125 obstacles = obstacles_temp - cone - rock
126
127 # 4) Update Rover.vision_image (this will be displayed on left side of screen)
128 # Example: Rover.vision_image[:,0] = obstacle color-thresholded binary image
129 #         Rover.vision_image[:,1] = rock_sample color-thresholded binary image
130 #         Rover.vision_image[:,2] = navigable terrain color-thresholded binary image
131 Rover.vision_image[:,0] = obstacles*255
132 Rover.vision_image[:,1] = rock*255
133 Rover.vision_image[:,2] = nav*255
134
135 # 5) Convert map image pixel values to rover-centric coords
136 # Terrain
137 xpix_nav, ypix_nav = rover_coords(nav)
138
139 # Obstacles
140 xpix_obstacles, ypix_obstacles = rover_coords(obstacles)
141
142 # Rocks
143 xpix_rock, ypix_rock = rover_coords(rock)
144
145 # 6) Convert rover-centric pixel values to world coordinates
146 # Define parameters which will transform image
147 xpos, ypos = Rover.pos
148 yaw = Rover.yaw
149 world_size = 200
150 scale = 10
151
152 # Terrain
153 nav_x_world, nav_y_world = pix_to_world(xpix_nav, ypix_nav, xpos, ypos, yaw, world_size, scale)
154
155 # Obstacles
156 obstacle_x_world, obstacle_y_world = pix_to_world(xpix_obstacles, ypix_obstacles, xpos, ypos, yaw, world_size, scale)
157
158 # Rocks
159 rock_x_world, rock_y_world = pix_to_world(xpix_rock, ypix_rock, xpos, ypos, yaw, world_size, scale)
160
161 # 7) Update Rover worldmap (to be displayed on right side of screen)
162 # Example: Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1
163 #         Rover.worldmap[rock_y_world, rock_x_world, 1] += 1
164 #         Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 1
165
166 # Add a small amount of colour to obstacles
167 Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 2
168 # Reset obstacle channel that rover currently sees as navigable terrain
169 Rover.worldmap[nav_y_world, nav_x_world, 0] = 0
170
171 # Update any rocks on the rock colour channel
172 Rover.worldmap[rock_y_world, rock_x_world, 1] = 255
173
174 # Reset anything on the navigable terrain channel that the rover currently sees as an obstacle
175 Rover.worldmap[obstacle_y_world, obstacle_x_world, 2] = 0
176 Rover.worldmap[nav_y_world, nav_x_world, 2] = 255
177
178 # 8) Convert rover-centric pixel positions to polar coordinates
179 # Update Rover pixel distances and angles
180 # Rover.nav_dists = rover_centric_pixel_distances
181 # Rover.nav_angles = rover_centric_angles
182
183 # If the rover detects a rock, and it is not in stuck mode, the Rover will
184 # switch to the rocking mode (i.e. the mode used to search for rocks)
185 if ( sum( sum(rock)) > 20) and Rover.mode != "stuck":
186     Rover.mode = "rocking"
187     Rover.rock_spot_count = 0 # The rover will reset the rock spot counter
188     distances, angles = to_polar_coords(xpix_rock, ypix_rock)
189     Rover.rock_dists = distances
190     Rover.rock_angles = angles
191 # If the Rover loses track of the rock, the Rover still remains in the rocking
192 # mode, however, will start to count down to switching back to forward mode
193 # provided the Rover cannot relocate the rock
194 elif Rover.mode == "rocking":
195     Rover.rock_spot_count += 1
196     Rover.rock_angles = 0
197
198 # The code will always take the updated navigation distances and angles
199 # irrespective of whether or not a rock is detected
200 distances, angles = to_polar_coords(xpix_nav, ypix_nav)
201 Rover.nav_dists = distances
202 Rover.nav_angles = angles
203
204 return Rover

```

6 Appendix B

```
1  # Do the necessary imports
2  import argparse
3  import shutil
4  import base64
5  from datetime import datetime
6  import os
7  import cv2
8  import numpy as np
9  import socketio
10 import eventlet
11 import eventlet.wsgi
12 from PIL import Image
13 from flask import Flask
14 from io import BytesIO, StringIO
15 import json
16 import pickle
17 import matplotlib.image as mpimg
18 import time
19
20 # Import functions for perception and decision making
21 from perception import perception_step
22 from decision import decision_step
23 from supporting_functions import update_rover, create_output_images
24 # Initialize socketio server and Flask application
25 # (learn more at: https://python-socketio.readthedocs.io/en/latest/)
26 sio = socketio.Server()
27 app = Flask(__name__)
28
29 # Read in ground truth map and create 3-channel green version for overplotting
30 # NOTE: images are read in by default with the origin (0, 0) in the upper left
31 # and y-axis increasing downward.
32 ground_truth = mpimg.imread('../calibration_images/map_bw.png')
33 # This next line creates arrays of zeros in the red and blue channels
34 # and puts the map into the green channel. This is why the underlying
35 # map output looks green in the display image
36 ground_truth_3d = np.dstack((ground_truth*0, ground_truth*255, ground_truth*0)).astype(np. float)
37
38 # Define RoverState() class to retain rover state parameters
39 class RoverState():
40     def __init__(self):
41         self.start_time = None # To record the start time of navigation
42         self.total_time = None # To record total duration of navigation
43         self.img = None # Current camera image
44         self.pos = None # Current position (x, y)
45         self.yaw = None # Current yaw angle
46         self.pitch = None # Current pitch angle
47         self.roll = None # Current roll angle
48         self.vel = None # Current velocity
49         self.vel_count = 0 # Start the velocity count at zero
50         self.stuck_count = None # Start the stuck count at zero
51         self.steer = 0 # Current steering angle
52         self.throttle = 0 # Current throttle value
53         self.brake = 0 # Current brake value
54         self.nav_angles = None # Angles of navigable terrain pixels
55         self.nav_dists = None # Distances of navigable terrain pixels
56         self.rock_angles = None # Angles of navigable rock pixels
57         self.rock_dists = None # Distances of navigable rock pixels
58         self.rock_spot_count = None
59         self.ground_truth = ground_truth_3d # Ground truth worldmap
60         self.mode = 'forward' # Current mode (can be forward or stop)
61         self.throttle_set = 0.2 # Throttle setting when accelerating
62         self.brake_set = 10 # Brake setting when braking
63         # The stop_forward and go_forward fields below represent total count
64         # of navigable terrain pixels. This is a very crude form of knowing
65         # when you can keep going and when you should stop. Feel free to
66         # get creative in adding new fields or modifying these!
67         self.stop_forward = 50 # Threshold to initiate stopping
68         self.go_forward = 500 # Threshold to go forward again
69         self.max_vel = 2 # Maximum velocity (meters/second)
70         # Image output from perception step
71         # Update this image to display your intermediate analysis steps
72         # on screen in autonomous mode
73         self.vision_image = np.zeros((160, 320, 3), dtype=np. float)
74         # Worldmap
75         # Update this image with the positions of navigable terrain
76         # obstacles and rock samples
77         self.worldmap = np.zeros((200, 200, 3), dtype=np. float)
78         self.samples_pos = None # To store the actual sample positions
79         self.samples_to_find = 0 # To store the initial count of samples
80         self.samples_located = 0 # To store number of samples located on map
81         self.samples_collected = 0 # To count the number of samples collected
82         self.near_sample = 0 # Will be set to telemetry value data["near_sample"]
83         self.picking_up = 0 # Will be set to telemetry value data["picking_up"]
84         self.send_pickup = False # Set to True to trigger rock pickup
85
86 # Initialize our rover
87 Rover = RoverState()
88
89 # Variables to track frames per second (FPS)
90 # Initialize frame counter
91 frame_counter = 0
92 # Initialize second counter
93 second_counter = time.time()
94 fps = None
95
96 # Define telemetry function for what to do with incoming data
97 @sio.on('telemetry')
98 def telemetry(sid, data):
99
100     global frame_counter, second_counter, fps
101     frame_counter+=1
102     # Do a rough calculation of frames per second (FPS)
103     if (time.time() - second_counter) > 1:
104         fps = frame_counter
```



```

105     frame_counter = 0
106     second_counter = time.time()
107     print("Current FPS: {}".format(fps))
108
109     if data:
110         global Rover
111         # Initialize / update Rover with current telemetry
112         Rover, image = update_rover(Rover, data)
113
114         if np.isfinite(Rover.vel):
115
116             # Execute the perception and decision steps to update the Rover's state
117             Rover = perception_step(Rover)
118             Rover = decision_step(Rover)
119
120             # Create output images to send to server
121             out_image_string1, out_image_string2 = create_output_images(Rover)
122
123             # The action step! Send commands to the rover!
124
125             # Don't send both of these, they both trigger the simulator
126             # to send back new telemetry so we must only send one
127             # back in response to the current telemetry data.
128
129             # If in a state where want to pickup a rock send pickup command
130             if Rover.send_pickup and not Rover.picking_up:
131                 send_pickup()
132                 # Reset Rover flags
133                 Rover.send_pickup = False
134             else:
135                 # Send commands to the rover!
136                 commands = (Rover.throttle, Rover.brake, Rover.steer)
137                 send_control(commands, out_image_string1, out_image_string2)
138
139             # In case of invalid telemetry, send null commands
140             else:
141
142                 # Send zeros for throttle, brake and steer and empty images
143                 send_control((0, 0, 0), '', '')
144
145             # If you want to save camera images from autonomous driving specify a path
146             # Example: $ python drive_rover.py image_folder_path
147             # Conditional to save image frame if folder was specified
148             if args.image_folder != '':
149                 timestamp = datetime.utcnow().strftime('%Y_%m_%d_%H_%M_%S_%f')[:-3]
150                 image_filename = os.path.join(args.image_folder, timestamp)
151                 image.save('{} .jpg'.format(image_filename))
152
153             else:
154                 sio.emit('manual', data={}, skip_sid=True)
155
156 @sio.on('connect')
157 def connect(sid, environ):
158     print("connect ", sid)
159     send_control((0, 0, 0), '', '')
160     sample_data = {}
161     sio.emit(
162         "get_samples",
163         sample_data,
164         skip_sid=True)
165
166 def send_control(commands, image_string1, image_string2):
167     # Define commands to be sent to the rover
168     data={
169         'throttle': commands[0].__str__(),
170         'brake': commands[1].__str__(),
171         'steering_angle': commands[2].__str__(),
172         'inset_image1': image_string1,
173         'inset_image2': image_string2,
174     }
175     # Send commands via socketIO server
176     sio.emit(
177         "data",
178         data,
179         skip_sid=True)
180     eventlet.sleep(0)
181
182 # Define a function to send the "pickup" command
183 def send_pickup():
184     print("Picking up")
185     pickup = {}
186     sio.emit(
187         "pickup",
188         pickup,
189         skip_sid=True)
190     eventlet.sleep(0)
191
192
193 if __name__ == '__main__':
194     parser = argparse.ArgumentParser(description='Remote Driving')
195     parser.add_argument(
196         'image_folder',
197         type= str,
198         nargs='?',
199         default='',
200         help='Path to image folder. This is where the images from the run will be saved.'
201     )
202     args = parser.parse_args()
203
204     #os.system('rm -rf IMG_stream/*')
205     if args.image_folder != '':
206         print("Creating image folder at {}".format(args.image_folder))
207         if not os.path.exists(args.image_folder):
208             os.makedirs(args.image_folder)
209         else:
210             shutil.rmtree(args.image_folder)
211             os.makedirs(args.image_folder)

```

```
212     print("Recording this run ...")
213 else:
214     print("NOT recording this run ...")
215
216 # wrap Flask application with socketio's middleware
217 app = socketio.Middleware(sio, app)
218
219 # deploy as an eventlet WSGI server
220 eventlet.wsgi.server(eventlet.listen(''), 4567), app)
```

7 Appendix C

```
1 import numpy as np
2
3
4 # This is where you can build a decision tree for determining throttle, brake and steer
5 # commands based on the output of the perception_step() function
6 def decision_step(Rover):
7     print(Rover.mode)
8     # Implement conditionals to decide what to do given perception data
9     # Here you're all set up with some basic functionality but you'll need to
10    # improve on this decision tree to do a good job of navigating autonomously!
11
12    # Example:
13    # Check if we have vision data to make decisions with
14    if Rover.nav_angles is not None:
15        # Check for Rover.mode status
16        if Rover.mode == 'forward':
17            # If the Rover is no longer stuck, then
18            if abs(Rover.vel) > 0.08:
19                Rover.vel_count = 0 # Reset the stuck counter
20            # Check the extent of navigable terrain
21            if len(Rover.nav_angles) >= Rover.stop_forward:
22                # If mode is forward, navigable terrain looks good
23                # and velocity is below max, then throttle
24                if Rover.vel < Rover.max_vel:
25                    # Set throttle value to throttle setting
26                    Rover.throttle = Rover.throttle_set
27                # If the rover is stuck
28                if abs(Rover.vel) < 0.05:
29                    # Some hysteresis so the robot doesn't switch modes rapidly
30                    Rover.vel_count += 1
31                    if Rover.vel_count > 400:
32                        Rover.vel_count = 0 # Reset the zero velocity count
33                        Rover.stuck_count = 0 # Start the stuck counter
34                        Rover.throttle = 0
35                        Rover.brake = 0
36                        Rover.steer = 0
37                        Rover.mode = 'stuck'
38            else: # Else coast
39                Rover.throttle = 0
40                Rover.brake = 0
41
42            # Set steering to average angle clipped to the range +/- 15
43            # Note that the steering angle has been smoothed by using a
44            # two time period weighted average
45            Rover.steer = (2*Rover.steer + np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15))/3
46
47        # If there's a lack of navigable terrain pixels then go to 'stop' mode
48        elif len(Rover.nav_angles) < Rover.stop_forward:
49            # Set mode to "stop" and hit the brakes!
50            Rover.throttle = 0
51            # Set brake to stored brake value
52            Rover.brake = Rover.brake_set
53            Rover.steer = 0
54            Rover.mode = 'stop'
55
56        # If the rover is in the vicinity of a rock - this is set in perception
57        elif Rover.mode == 'rocking':
58            if abs(Rover.vel) > 0.08:
59                Rover.vel_count = 0 # Reset the stuck counter
60            if Rover.rock_spot_count > 200:
61                Rover.mode = "forward"
62            else:
63                if Rover.vel < 0.5:
64                    Rover.throttle = 0.1
65                    Rover.brake = 0
66                    if abs(Rover.vel) < 0.05:
67                        # Some hysteresis so the robot doesn't switch modes rapidly
68                        Rover.vel_count += 1
69                        if Rover.vel_count > 400:
70                            Rover.vel_count = 0 # Reset the zero velocity count
71                            Rover.stuck_count = 0 # Start the stuck counter
72                            Rover.throttle = 0
73                            Rover.brake = 0
74                            Rover.steer = 0
75                            Rover.mode = 'stuck'
76                elif Rover.vel >= 0.5:
77                    Rover.throttle = 0
78
79            # This will direct the rover towards the rock
80            Rover.steer = np.clip(np.mean(Rover.rock_angles * 180/np.pi), -15, 15)
81
82            # This will stop the Rover if it is near a rock sample
83            # Note this is a condition for picking up samples
84            if Rover.near_sample:
85                Rover.throttle = 0
86                Rover.brake = Rover.brake_set
87
88        # If we're already in "stop" mode then make different decisions
89        elif Rover.mode == 'stop':
90            # If we're in stop mode but still moving keep braking
91            if Rover.vel > 0.2:
92                Rover.throttle = 0
93                Rover.brake = Rover.brake_set
94                Rover.steer = 0
95            # If we're not moving (vel < 0.2) then do something else
96            elif Rover.vel <= 0.2:
97                # Now we're stopped and we have vision data to see if there's a path forward
98                if len(Rover.nav_angles) < Rover.go_forward:
99                    Rover.throttle = 0
100                    # Release the brake to allow turning
101                    Rover.brake = 0
102                    # Turn range is +/- 15 degrees, when stopped the next line will induce 4-wheel turning
103                    Rover.steer = -15 # Could be more clever here about which way to turn
104                    # If we're stopped but see sufficient navigable terrain in front then go!
```

```

105         if len(Rover.nav_angles) >= Rover.go_forward:
106             # Set throttle back to stored value
107             Rover.throttle = Rover.throttle_set
108             # Release the brake
109             Rover.brake = 0
110             # Set steer to mean angle
111             Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)
112             Rover.mode = 'forward'
113
114         # This is the Rover stuck mode - that is if the wheels are caught on the terrain
115         elif Rover.mode == 'stuck':
116             Rover.stuck_count += 1 # Add a count to the hysteresis
117             Rover.steer = 0
118             Rover.throttle = -0.1 # This will reverse the rover with 0 steer angle
119             Rover.brake = 0
120
121         # This behaviour, once triggered only needs to occur for 300 processed frames
122         # which is less than 10 seconds
123         if Rover.stuck_count > 300:
124             Rover.stuck_count = 0 # Reset the stuck counter
125             Rover.steer = 0 # Reset the Rover steering
126             Rover.mode = 'forward' # Hopefully the rover is unstuck
127
128         # Just to make the rover do something
129         # even if no modifications have been made to the code
130         else:
131             Rover.throttle = Rover.throttle_set
132             Rover.steer = 0
133             Rover.brake = 0
134
135         # If in a state where want to pickup a rock send pickup command
136         if Rover.near_sample and Rover.vel == 0 and not Rover.picking_up:
137             Rover.send_pickup = True
138
139     return Rover
140

```