Udacity: Search and Sample Return Report

Shane Reynolds

February 25, 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.

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.

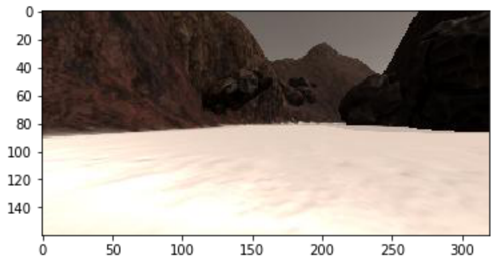

Figure 2: A single image of the simulated environment as shown from the robot's front mounted camera.

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 4. 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: Approximate points determined from the destination and source images use to create the perspective transformation matrix.

Source	**Destination**
(14.94,139.83)	(150,150)
(301.57,139.83)	(160,150)
(199.08,95.82)	(160,140)
(119.32,95.75)	(150,140)

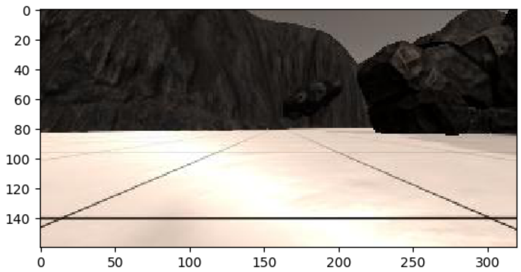

Figure 3: A 1x1 meter grid used to help establish points of reference.

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
```

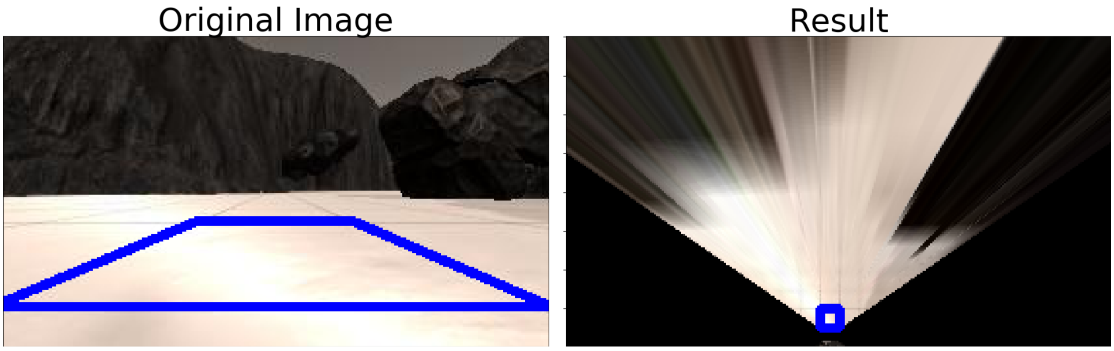

Figure 4: The original image from the rover's front mounted camera can be seen on the left, with the image after the perspective transform has been applied can be seen on the right.

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.

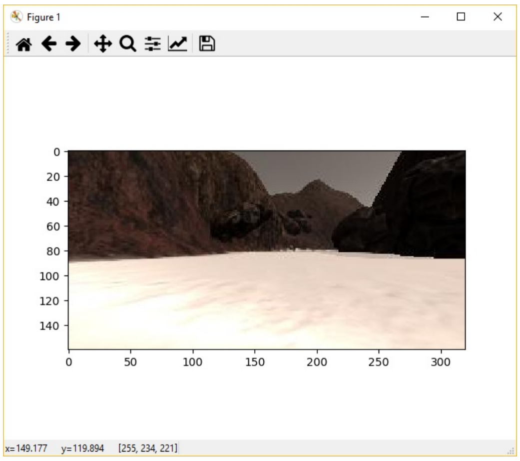

Figure 5: Using the figure display tool in Spyder, a Python IDE, crude analysis of the image was undertaken to determine the threshold values for R, G, and B values which represent navigable terrain.

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: Code snippet of the colour thresholding function

```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
```

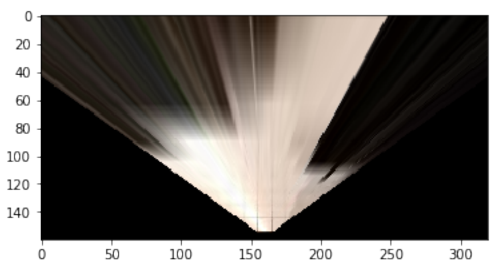

Figure 6: The perspective transformed image prior to the segmentation filter application.

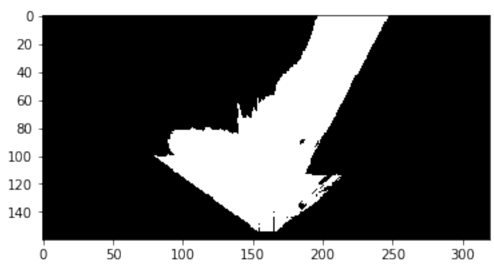

Figure 7: The perspective transformed image after the segmentation filter is applied - note that this is shown in grayscale.

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: Code snippet showing how the image is segmented

```
# 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
```

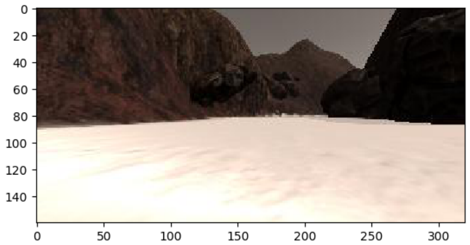

Figure 8: The original image taken from the camera mounted to the front of the rover.

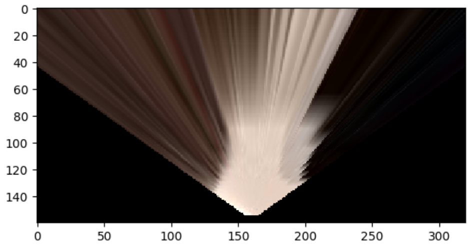

Figure 9: The image once it has undergone the perspective transform discussed in Section 2.2.1.

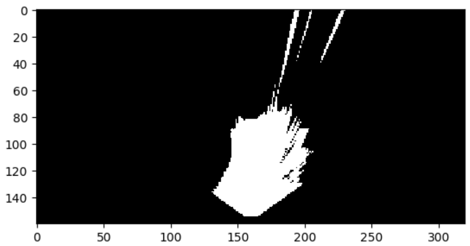

Figure 10: The resultant image of the segmentation for navigable terrain. Note that the white areas depict the navigable terrain areas.

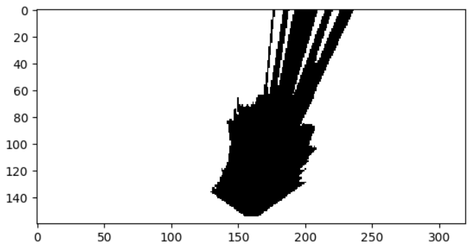

Figure 11: The resultant image for the inverse of the navigable terrain.

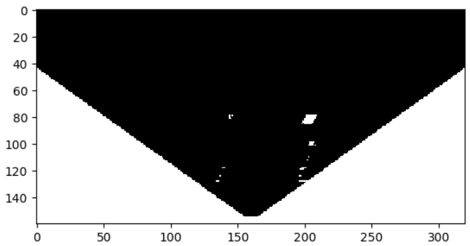

Figure 12: The resultant image for the derivation of rover blind spot.

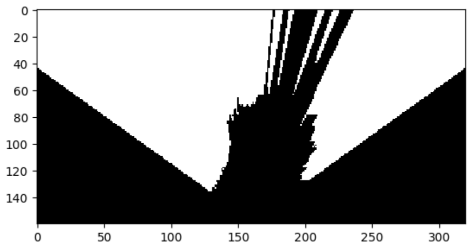

Figure 13: The final image which shows the rover obstacles. Note that the white areas depict the rover obstacles.

2.2.3 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 sample rocks.

Listing 5: Code snippet showing how CV2 HSV is used to detect rock samples

```
# 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)
```

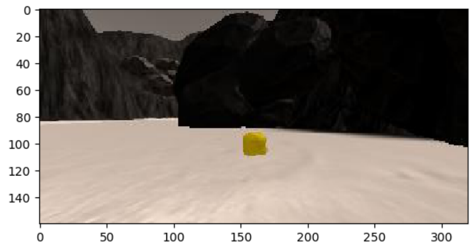

Figure 14: An image of a rock sample

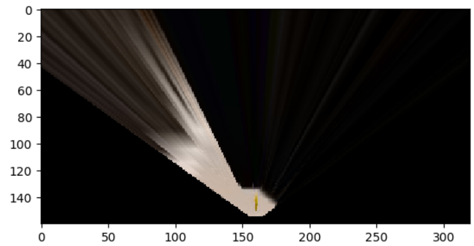

Figure 15: Perspective transform of original image

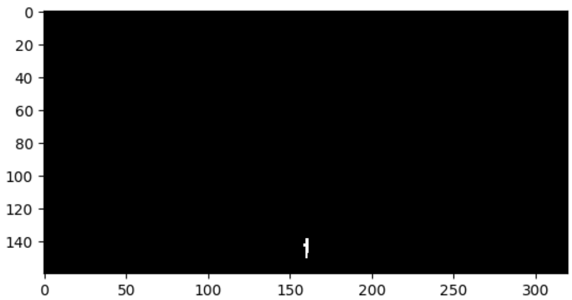

Figure 16: Segmentation of the perspective transformed image, showing the detected rock sample.

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.

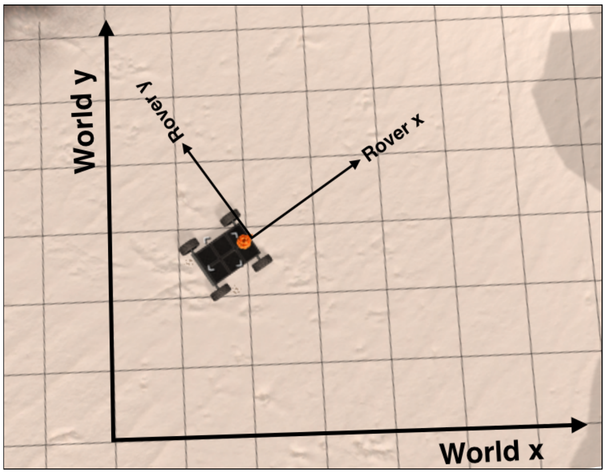

Figure 17: A coordinate frame is attached to, and moves with, the rover. This rover coordinate frame exists in the fixed world frame.

The image that is captured and processed by the `perception.py` function uses the source and destination points to map the received 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 is, 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: Code snippet showing function for transforming perspective transformed image to rover centric coordinates

```
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
```

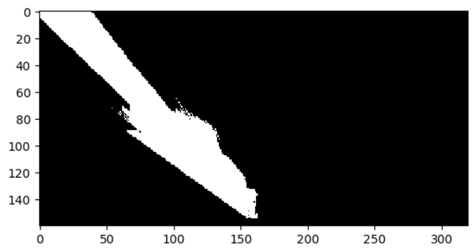

Figure 18: Perspective transformed, segmented image for navigable terrain prior to converting the image to rover centric coordinates

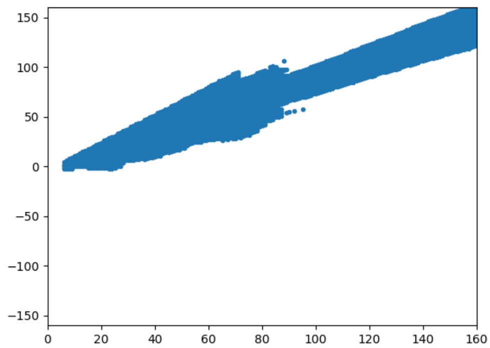

Figure 19: Perspective transformed, segmented image for navigable terrain after to converting the image to rover centric coordinates

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.

Listing 7: A code snippet showing a function which rotates pixels in an image by the angle yaw

```
# 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: A code snippet which shows a function used for translation of pixels in an image

```
# 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: A code snippet which shows the combined rotation and translation required for mapping the rover centric coordinates to the world frame

```
# 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
```

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 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: A code snippet showing how the world map is updated as the rover scouts an area

```
# 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

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. All navigation is done by taking navigation pixels determined by the segmentation function and converting them into angles using the polar conversion function, shown in Listing 11.

Listing 11: A code snippet showing the function for converting Cartesian coordinates to polar Coordinates for a given pixel

```
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 states: forward, rocking, stop, and stuck.

The forward state code snippet is shown in Listing 12, and the flow diagram explaining the logic behind the operation is shown in Figure 20. 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^o and 15^o.

Listing 12: 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'
```

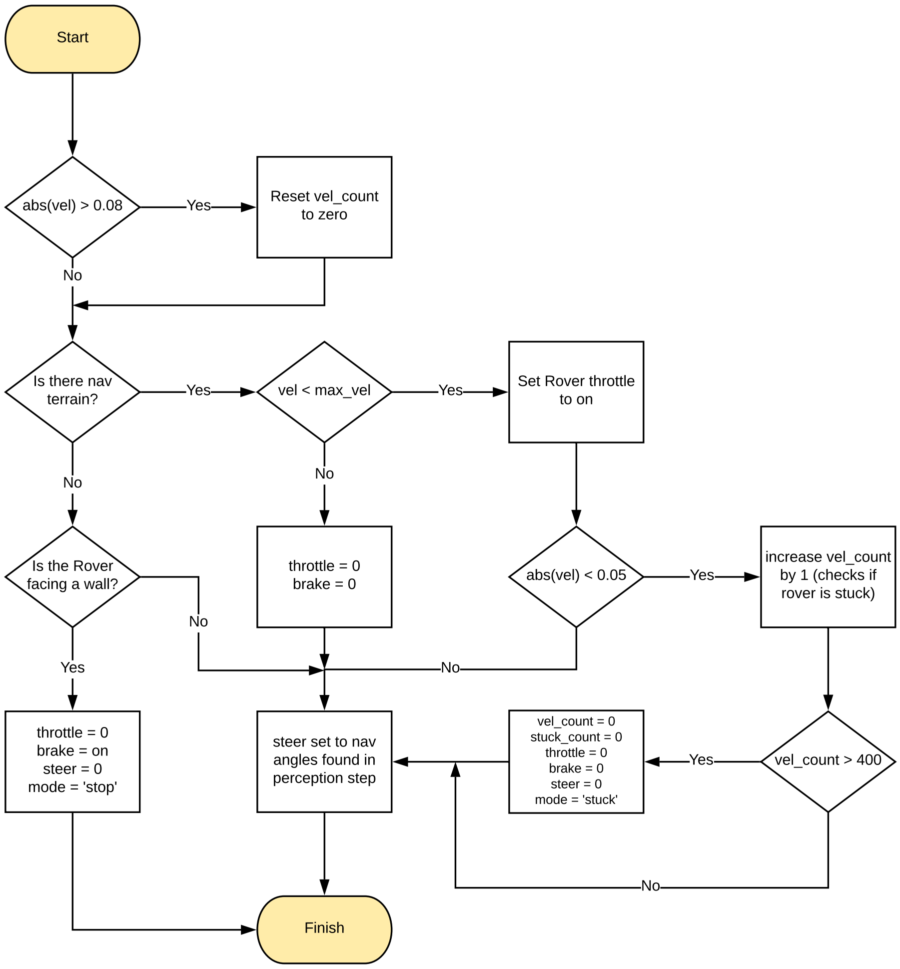

Figure 20: Flow diagram showing the operation of the rover when it is in the forward state of operation.

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 13. 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 14, and the flow diagram explaining the logic behind the operation is shown in Figure 21.

Listing 13: 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 14: 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.

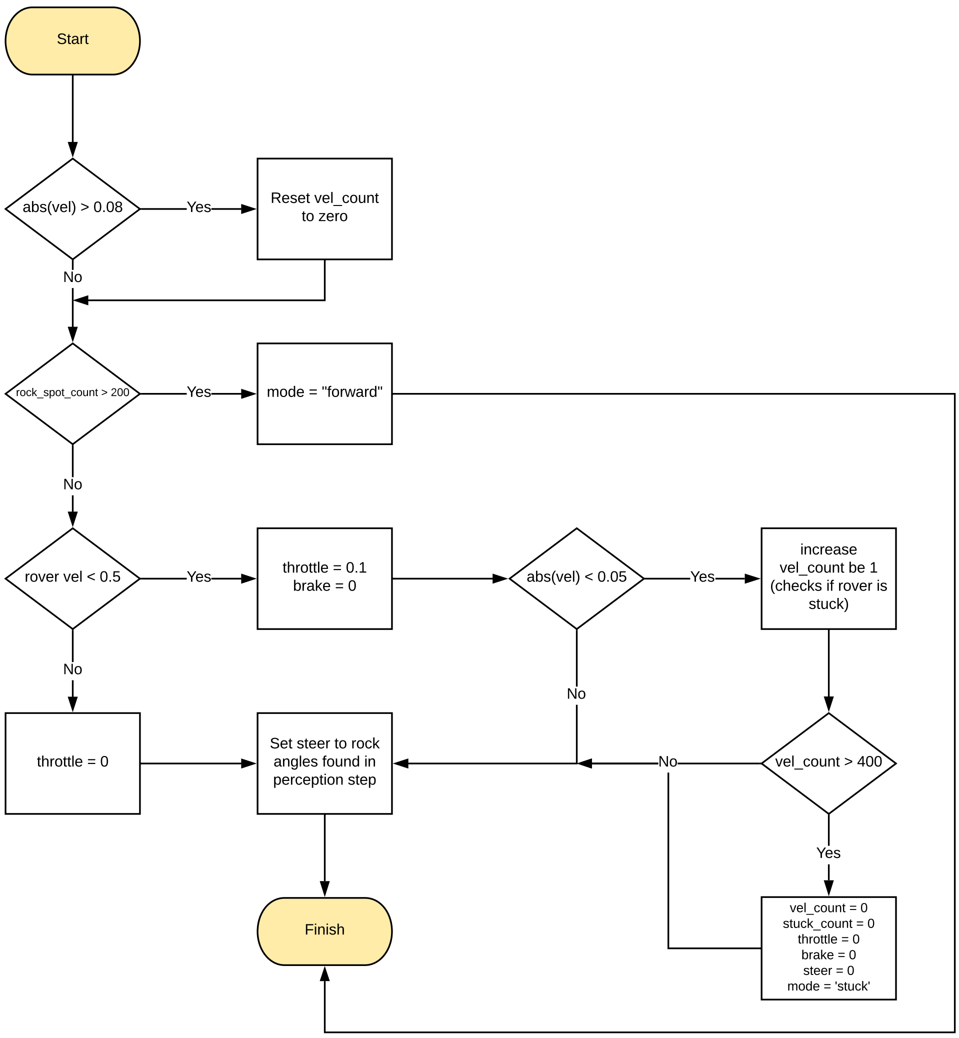

Figure 21: Flow diagram showing the operation of the rover when it is in the rocking state of operation.

The stuck state code snippet can be seen in Listing 15, and the flow diagram explaining the logic behind the operation is shown in Figure 22. 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 15: 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
```

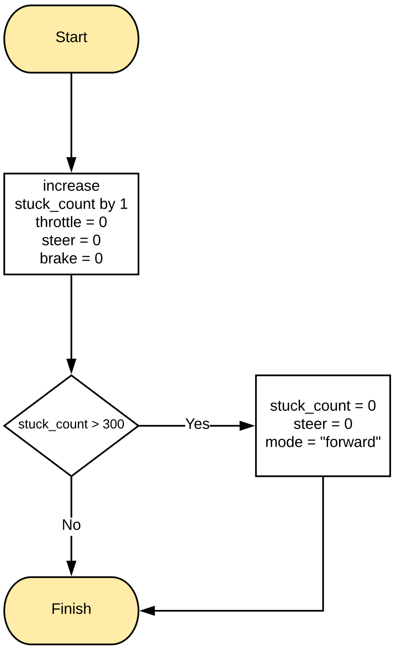

Figure 22: Flow diagram showing the operation of the rover when it is in the stuck state of operation.

Finally, the stop state code snippet can be seen in Listing 16, and the flow diagram explaining the logic behind the operation is shown in Figure 23. 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 16: 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/-hOlGIr9gxk`

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.

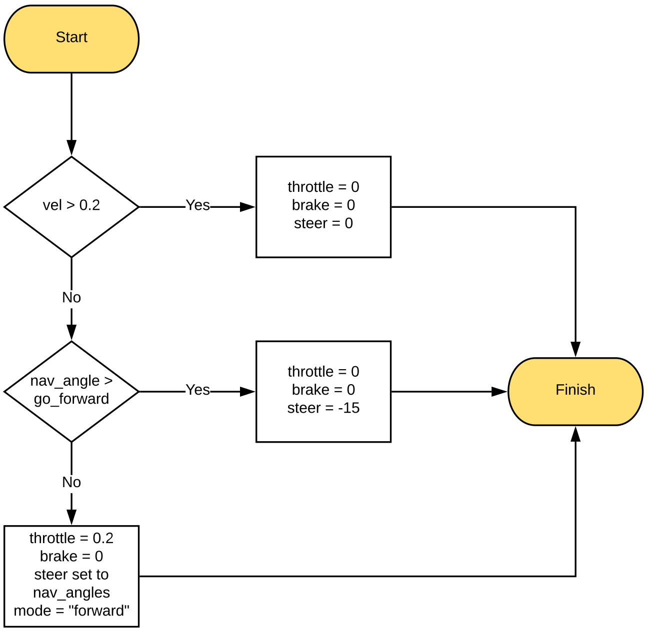

Figure 23: Flow diagram showing the operation of the rover when it is in the stuck state of operation.

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 becomes irrecoverably stuck.

5 Appendix A

The full implementation of the Python code for `perception.py` which provides the rover with the ability to process images.

```
import numpy as np
import cv2

# Identify pixels above the threshold
# Threshold of RGB > 160 does a nice job of identifying ground pixels only
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

# Define a function to convert from image coords to rover coords
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


# Define a function to convert to radial coords in rover space
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

# Define a function to map rover space pixels to world space
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

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


# Define a function to apply rotation and translation (and clipping)
# Once you define the two functions above this function should work
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

# Define a function to perform a perspective transform
def perspect_transform(img, src, dst):

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

    return warped


# Apply the above functions in succession and update the Rover state accordingly
def perception_step(Rover):
    # Perform perception steps to update Rover()
    # TODO:
    # NOTE: camera image is coming to you in Rover.img

    # 1) Define source and destination points for perspective transform
    img = Rover.img
    dst_size = 5
    bottom_offset = 6
    source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
    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],
                  ])

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

```python

    # 3) 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([0,120,120])
    upper_thres = np.array([40,255,255])
    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

    # 4) Update Rover.vision_image (this will be displayed on left side of screen)
        # Example: Rover.vision_image[:,:,0] = obstacle color-thresholded binary image
        #          Rover.vision_image[:,:,1] = rock_sample color-thresholded binary image
        #          Rover.vision_image[:,:,2] = navigable terrain color-thresholded binary image
    Rover.vision_image[:,:,0] = obstacles*255
    Rover.vision_image[:,:,1] = rock*255
    Rover.vision_image[:,:,2] = nav*255

    # 5) Convert map image pixel values to rover-centric coords
    # Terrain
    xpix_nav, ypix_nav = rover_coords(nav)

    # Obstacles
    xpix_obstacles, ypix_obstacles = rover_coords(obstacles)

    # Rocks
    xpix_rock, ypix_rock = rover_coords(rock)

    # 6) Convert rover-centric pixel values to world coordinates
    # Define parameters which will transform image
    xpos, ypos = Rover.pos
    yaw = Rover.yaw
    world_size = 200
    scale = 10

    # Terrain
    nav_x_world, nav_y_world = pix_to_world(xpix_nav, ypix_nav, xpos, ypos, yaw, world_size, scale)

    # Obstacles
    obstacle_x_world, obstacle_y_world = pix_to_world(xpix_obstacles, ypix_obstacles, xpos, ypos, yaw, world_size, scale)

    # Rocks
    rock_x_world, rock_y_world = pix_to_world(xpix_rock, ypix_rock, xpos, ypos, yaw, world_size, scale)

    # 7) Update Rover worldmap (to be displayed on right side of screen)
        # Example: Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1
        #          Rover.worldmap[rock_y_world, rock_x_world, 1] += 1
        #          Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 1

    # 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

    # 8) Convert rover-centric pixel positions to polar coordinates
    # Update Rover pixel distances and angles
        # Rover.nav_dists = rover_centric_pixel_distances
        # Rover.nav_angles = rover_centric_angles

    # 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

    # The code will always take the updated navigation distances and angles
    # irrespective of whether or not a rock is detected
    distances, angles = to_polar_coords(xpix_nav, ypix_nav)
    Rover.nav_dists = distances
    Rover.nav_angles = angles

    return Rover
```

6 Appendix B

The full implementation of the Python code for `drive_rover.py` which provides the rover with the ability to function autonomously.

```
# Do the necessary imports
import argparse
import shutil
import base64
from datetime import datetime
import os
import cv2
import numpy as np
import socketio
import eventlet
import eventlet.wsgi
from PIL import Image
from flask import Flask
from io import BytesIO, StringIO
import json
import pickle
import matplotlib.image as mpimg
import time

# Import functions for perception and decision making
from perception import perception_step
from decision import decision_step
from supporting_functions import update_rover, create_output_images
# Initialize socketio server and Flask application
# (learn more at: https://python-socketio.readthedocs.io/en/latest/)
sio = socketio.Server()
app = Flask(__name__)

# Read in ground truth map and create 3-channel green version for overplotting
# NOTE: images are read in by default with the origin (0, 0) in the upper left
# and y-axis increasing downward.
ground_truth = mpimg.imread('../calibration_images/map_bw.png')
# This next line creates arrays of zeros in the red and blue channels
# and puts the map into the green channel.  This is why the underlying
# map output looks green in the display image
ground_truth_3d = np.dstack((ground_truth*0, ground_truth*255, ground_truth*0)).astype(np. float)

# Define RoverState() class to retain rover state parameters
class RoverState():
    def __init__(self):
        self.start_time = None # To record the start time of navigation
        self.total_time = None # To record total duration of naviagation
        self.img = None # Current camera image
        self.pos = None # Current position (x, y)
        self.yaw = None # Current yaw angle
        self.pitch = None # Current pitch angle
        self.roll = None # Current roll angle
        self.vel = None # Current velocity
        self.vel_count = 0 # Start the velocity count at zero
        self.stuck_count = None # Start the stuck count at zero
        self.steer = 0 # Current steering angle
        self.throttle = 0 # Current throttle value
        self.brake = 0 # Current brake value
        self.nav_angles = None # Angles of navigable terrain pixels
        self.nav_dists = None # Distances of navigable terrain pixels
        self.rock_angles = None # Angles of navigable rock pixels
        self.rock_dists = None # Distances of navigable rock pixels
        self.rock_spot_count = None
        self.ground_truth = ground_truth_3d # Ground truth worldmap
        self.mode = 'forward' # Current mode (can be forward or stop)
        self.throttle_set = 0.2 # Throttle setting when accelerating
        self.brake_set = 10 # Brake setting when braking
        # The stop_forward and go_forward fields below represent total count
        # of navigable terrain pixels.  This is a very crude form of knowing
        # when you can keep going and when you should stop.  Feel free to
        # get creative in adding new fields or modifying these!
        self.stop_forward = 50 # Threshold to initiate stopping
        self.go_forward = 500 # Threshold to go forward again
        self.max_vel = 2 # Maximum velocity (meters/second)
        # Image output from perception step
        # Update this image to display your intermediate analysis steps
        # on screen in autonomous mode
        self.vision_image = np.zeros((160, 320, 3), dtype=np. float)
        # Worldmap
        # Update this image with the positions of navigable terrain
        # obstacles and rock samples
        self.worldmap = np.zeros((200, 200, 3), dtype=np. float)
        self.samples_pos = None # To store the actual sample positions
        self.samples_to_find = 0 # To store the initial count of samples
        self.samples_located = 0 # To store number of samples located on map
        self.samples_collected = 0 # To count the number of samples collected
        self.near_sample = 0 # Will be set to telemetry value data["near_sample"]
        self.picking_up = 0 # Will be set to telemetry value data["picking_up"]
        self.send_pickup = False # Set to True to trigger rock pickup
# Initialize our rover
Rover = RoverState()

# Variables to track frames per second (FPS)
# Intitialize frame counter
frame_counter = 0
# Initalize second counter
second_counter = time.time()
fps = None


# Define telemetry function for what to do with incoming data
@sio.on('telemetry')
def telemetry(sid, data):

    global frame_counter, second_counter, fps
    frame_counter+=1
    # Do a rough calculation of frames per second (FPS)
    if (time.time() - second_counter) > 1:
```

```python
        fps = frame_counter
        frame_counter = 0
        second_counter = time.time()
    print("Current FPS: {}".format(fps))

    if data:
        global Rover
        # Initialize / update Rover with current telemetry
        Rover, image = update_rover(Rover, data)

        if np.isfinite(Rover.vel):

            # Execute the perception and decision steps to update the Rover's state
            Rover = perception_step(Rover)
            Rover = decision_step(Rover)

            # Create output images to send to server
            out_image_string1, out_image_string2 = create_output_images(Rover)

            # The action step!  Send commands to the rover!

            # Don't send both of these, they both trigger the simulator
            # to send back new telemetry so we must only send one
            # back in respose to the current telemetry data.

            # If in a state where want to pickup a rock send pickup command
            if Rover.send_pickup and not Rover.picking_up:
                send_pickup()
                # Reset Rover flags
                Rover.send_pickup = False
            else:
                # Send commands to the rover!
                commands = (Rover.throttle, Rover.brake, Rover.steer)
                send_control(commands, out_image_string1, out_image_string2)

        # In case of invalid telemetry, send null commands
        else:

            # Send zeros for throttle, brake and steer and empty images
            send_control((0, 0, 0), '', '')

        # If you want to save camera images from autonomous driving specify a path
        # Example: $ python drive_rover.py image_folder_path
        # Conditional to save image frame if folder was specified
        if args.image_folder != '':
            timestamp = datetime.utcnow().strftime('%Y_%m_%d_%H_%M_%S_%f')[:-3]
            image_filename = os.path.join(args.image_folder, timestamp)
            image.save('{}.jpg'.format(image_filename))

    else:
        sio.emit('manual', data={}, skip_sid=True)

@sio.on('connect')
def connect(sid, environ):
    print("connect ", sid)
    send_control((0, 0, 0), '', '')
    sample_data = {}
    sio.emit(
        "get_samples",
        sample_data,
        skip_sid=True)

def send_control(commands, image_string1, image_string2):
    # Define commands to be sent to the rover
    data={
        'throttle': commands[0].__str__(),
        'brake': commands[1].__str__(),
        'steering_angle': commands[2].__str__(),
        'inset_image1': image_string1,
        'inset_image2': image_string2,
        }
    # Send commands via socketIO server
    sio.emit(
        "data",
        data,
        skip_sid=True)
    eventlet.sleep(0)

# Define a function to send the "pickup" command
def send_pickup():
    print("Picking up")
    pickup = {}
    sio.emit(
        "pickup",
        pickup,
        skip_sid=True)
    eventlet.sleep(0)


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Remote Driving')
    parser.add_argument(
        'image_folder',
         type=str,
        nargs='?',
        default='',
         help='Path to image folder. This is where the images from the run will be saved.'
    )
    args = parser.parse_args()

    #os.system('rm -rf IMG_stream/*')
    if args.image_folder != '':
        print("Creating image folder at {}".format(args.image_folder))
        if not os.path.exists(args.image_folder):
            os.makedirs(args.image_folder)
        else:
            shutil.rmtree(args.image_folder)
```

```
            os.makedirs(args.image_folder)
        print("Recording this run ...")
    else:
        print("NOT recording this run ...")

    # wrap Flask application with socketio's middleware
    app = socketio.Middleware(sio, app)

    # deploy as an eventlet WSGI server
    eventlet.wsgi.server(eventlet.listen(('', 4567)), app)
```

7 Appendix C

The full implementation of the Python code for decision.py which provides the rover with the ability to utilise information determined from the perception step. Principally this function is used to change the rover from state to state.

```python
import numpy as np


# This is where you can build a decision tree for determining throttle, brake and steer
# commands based on the output of the perception_step() function
def decision_step(Rover):
    print(Rover.mode)
    # Implement conditionals to decide what to do given perception data
    # Here you're all set up with some basic functionality but you'll need to
    # improve on this decision tree to do a good job of navigating autonomously!

    # Example:
    # Check if we have vision data to make decisions with
    if Rover.nav_angles is not None:
        # 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'

        # 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

        # 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'

        # 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


    # Just to make the rover do something
    # even if no modifications have been made to the code
    else:
        Rover.throttle = Rover.throttle_set
        Rover.steer = 0
        Rover.brake = 0

    # If in a state where want to pickup a rock send pickup command
    if Rover.near_sample and Rover.vel == 0 and not Rover.picking_up:
        Rover.send_pickup = True

    return Rover
```