Search and Sample Return Project Writeup

Author: Roberto Zegers Rusche
Date: July 31th 2018
GitHub repository: https://github.com/digitalgroove/RoboND_Project_1
Jupyter Notebook: Jupyter Notebook/Custom_functions.ipynb
Dependencies: To run the submitted code and Jupyter Notebook, set up and activate a virtual environment as described here:
https://github.com/udacity/RoboND-Python-StarterKit/blob/master/doc/configure_via_anaconda.md

Summary: Here I'll talk about the approach I took, how I worked through challenges that I encountered, what algorithms I wrote, provide an explanation of the code, and ultimately how I iterated throughout the development process.

Part 1: Training and Calibration

To complete this part, I downloaded the rover simulator and captured data in Training Mode. Next I used the Online Lab to test the code provided by the course and completed following tasks:

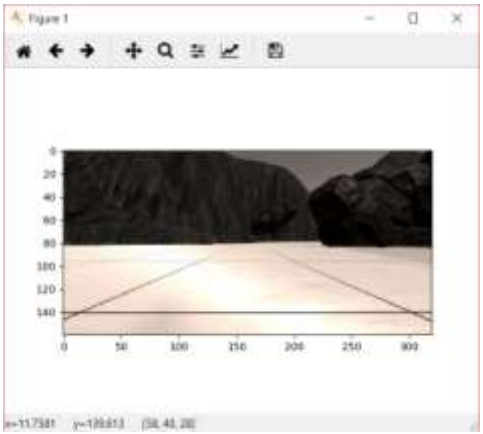
1.- Select coordinates from grid image to use as input to wrap a camera image

To wrap a camera image to a top-view perspective image I first ran this code on a local machine to obtain the input coordinates:

Python code:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
#%matplotlib notebook
image = mpimg.imread('calibration_images/example_grid1.jpg')
plt.imshow(image)
plt.show()
```

Output:



Using the mouse pointer and the position readout in the lower left side of the window above I took note of the coordinates of the corners of the square projected on the ground.
Coordinates [10, 140], [300, 140], [200, 95] and [118, 95] where defined as inputs for the Perspective Transform function:

```
source = np.float32([[10, 140], [300, 140],[200, 95],[118, 95]])
```

and then passed as input when calling perspect_transform():

```
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

dst_size = 5
bottom_offset = 6
source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])

destination = np.float32([[image.shape[1]/2 - dst_size, image.shape[0] - bottom_offset],
                  [image.shape[1]/2 + dst_size, image.shape[0] - bottom_offset],
                  [image.shape[1]/2 + dst_size, image.shape[0] - 2*dst_size - bottom_offset],
                  [image.shape[1]/2 - dst_size, image.shape[0] - 2*dst_size - bottom_offset],
                  ])
warped = perspect_transform(grid_img, source, destination)
plt.imshow(warped)
```

2.- Write a function to allow for navigable terrain identification.

I wrote a function called color_thresh() that reads the color value on every channel on each pixel of an image and compares its value with a threshold. It outputs a binary image that contains ones where the threshold was surpassed and zeros where not.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

# Read in the image
image_name = 'sample6.jpg'
image = mpimg.imread(image_name)

# Define a function to perform a color threshold
def color_thresh(img, rgb_thresh=(0, 0, 0)):
    # An empty array the same size in x and y as the image
    # but just a single channel
    color_select = np.zeros_like(img[:,:,0])
    # Apply the thresholds for RGB and assign 1's
    # where threshold was exceeded
    for row in range (img.shape[0]):
        for col in range(img.shape[1]):
            for channel in range(3):
                if img[row,col,channel] > rgb_thresh[channel]:
                    color_select[row,col] = 1
                else:
                    color_select[row,col] = 0
    # Return the single-channel binary image
    return color_select

# Define color selection criteria
red_threshold = 174
green_threshold = 174
blue_threshold = 174
rgb_threshold = (red_threshold, green_threshold, blue_threshold)

# pixels below the thresholds
colorsel = color_thresh(image, rgb_thresh=rgb_threshold)

# Display the original image and binary
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(21, 7), sharey=True)
f.tight_layout()
ax1.imshow(image)
ax2.imshow(colorsel, cmap='gray')
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
#plt.show() # Uncomment if running on your local machine
```

This algorithm is different to the solution proposed by the course, but gets identical results and passes the Perception Step Online Quiz.


Input: Original Image


Output: Binary thresholded image

To add the capability to identify obstacles and rock samples I took the solution provided by the course and rewrote it so that it to takes as input a min and a max threshold value on each channel.

Below you see the improved function that takes as input a min and a max threshold value:

```
# Identify pixels above the threshold
# Threshold of RGB > 160 does a nice job of identifying ground pixels only
def color_thresh(img, rgb_thresh_min=(160, 160, 160), rgb_thresh_max = (255,255,255)):
    # 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
    within_thresh = (img[:,:,0] > rgb_thresh_min[0]) \
            & (img[:,:,1] > rgb_thresh_min[1]) \
            & (img[:,:,2] > rgb_thresh_min[2]) \
            & (img[:,:,0] < rgb_thresh_max[0]) \
            & (img[:,:,1] < rgb_thresh_max[1]) \
            & (img[:,:,2] < rgb_thresh_max[2])
    # Index the array of zeros with the boolean array and set to 1
    color_select[within_thresh] = 1
    return color_select

threshed = color_thresh(warped)
plt.imshow(threshed, cmap='gray')
#scipy.misc.imsave('../output/warped_threshed.jpg', threshed*255)
```

Below you can find the color values used to identify navigable terrain, obstacles and rock samples:

Example call to color_thresh function to identify **navigable terrain**:

```
rgb_nav_min = (160,160,160)
rgb_nav_max = (255, 255, 255)
navigable_threshed = color_thresh(warped, rgb_nav_min, rgb_nav_max)
```

Example call to color_thresh function to identify **obstacles**:

```
rgb_nav_min = (0,0,0)
rgb_nav_max = (170, 170, 170)
threshed_obstacles = color_thresh(warped, rgb_nav_min, rgb_nav_max)
```

Example call to color_thresh function to identify **rock samples**:

```
rgb_nav_min = (110,110,0)
rgb_nav_max = (210, 210, 50)
threshed_rocks = color_thresh(warped, rgb_nav_min, rgb_nav_max)
```

You can run and test this function on the Jupyter Notebook which I included as part of my project submission under the folder “Jupyter Notebook” and filename "Custom_functions.ipynb".

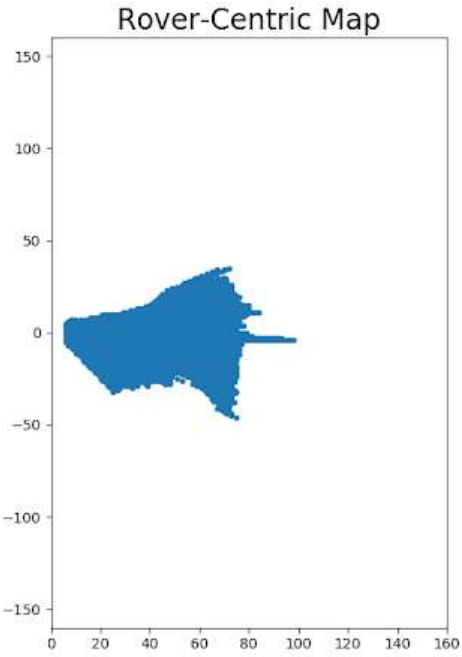
3.- Extract x and y from an wrapped image and return x and y in rover coordinates

To solve this task I wrote the algorithm shown below. It produces a correct output image and works well when running a quiz test. However when submitting it through the website to the automatic grading tool for programming assignments, the check fails and the following error message appears: Oops, looks like you got an error! 'list' object has no attribute 'sum'
I send the full code to my mentor seeking advice or guidance. He responded that the fail to pass could be due to an error of the online quiz checker tool.

Custom made rover_coords() function:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2
from extra_functions import *

# Read in the sample image
image = mpimg.imread('sample.jpg')

def rover_coords(binary_img):
# TODO: fill in this function to
# Calculate pixel positions with reference to the rover
# position being at the center bottom of the image.
    x_pixel = []
    y_pixel = []
    for col in range(binary_img.shape[1]):
        for row in range(binary_img.shape[0]):
            if binary_img[row,col] == 1:
                y_pixel.append(binary_img.shape[1]/2-col)
                x_pixel.append(binary_img.shape[0]-row)
    return x_pixel, y_pixel

# Perform warping and color thresholding
warped = perspect_transform(image, source, destination)
colorsel = color_thresh(warped, rgb_thresh=(160, 160, 160))
# Extract x and y positions of navigable terrain pixels
# and convert to rover coordinates
xpix, ypix = rover_coords(colorsel)

# Plot the map in rover-centric coords
fig = plt.figure(figsize=(5, 7.5))
plt.plot(xpix, ypix, '.')
plt.ylim(-160, 160)
plt.xlim(0, 160)
plt.title('Rover-Centric Map', fontsize=20)
#plt.show() # Uncomment if running on your local machine
```

Result using the online Test Run:



Unfortunately I also had difficulties to get this function to work on a local Jupyter Notebook.
In order to move on with the assignment, I will from here on use the rover_coords() function provided as solution to the quiz by the course:

```
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 = np.absolute(ypos - binary_img.shape[0]).astype(np.float)
    y_pixel = -(xpos - binary_img.shape[0]).astype(np.float)
    return x_pixel, y_pixel
```

4.- Write a function to rotate and function to translate the mapped pixels

I wrote two functions shown below and tested them using the Online Lab tool. Then I called rotate_pix() and translate_pix() in sequence from within the function pix_to_world() to perform the transform from rover-centric coordinates to world coordinates.

Function to apply a rotation to pixel positions:

```
def rotate_pix(xpix, ypix, yaw):
    # Convert yaw to radians, then apply a rotation
    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 xpix_rotated, ypix_rotated
```

Function to perform a translation:

```
def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # Apply a scaling and a translation
    xpix_translated = np.int_(xpos + (xpix_rot / scale))
    ypix_translated = np.int_(ypos + (ypix_rot / scale))
    return xpix_translated, ypix_translated
```

Rotation and translation are then called from a function already implemented called pix_to_world() which also adds clipping.

```
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 x_pix_world, y_pix_world
```

5.- Integrate all previously developed functions into one main processing function

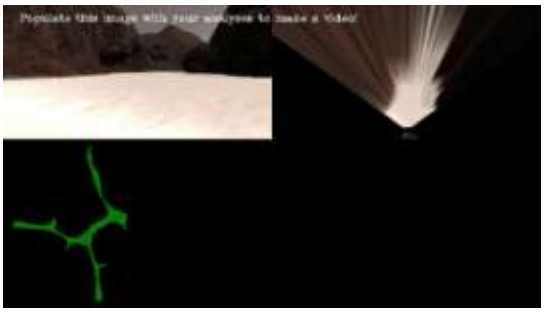
Final step for Training / Calibration was to populate the **process_image()** function with the appropriate image processing to get from raw images to a map. This function creates an output image demonstrating that the mapping pipeline works:

```
def process_image(img):
    # 1) Define source and destination points for perspective transform
    dst_size = 5
    source = np.float32([[200, 95],[300, 140],[10, 140],[118, 95]])
    destination = np.float32([[165, 135],[165, 145],[155, 145],[155, 135]])
    # 2) Apply perspective transform
    warped = perspect_transform(img, source, destination)
    # 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
    rgb_nav_min = (170,170,170)
    rgb_nav_max = (255, 255, 255)
    navigable_threshed = color_thresh(warped, rgb_nav_min, rgb_nav_max)
    rgb_obs_min = (0,0,0)
    rgb_obs_max = (170,170,170)
    threshed_obstacles = color_thresh(warped,rgb_obs_min, rgb_obs_max)
    rgb_rock_min = (110, 110, 5)
    rgb_rock_max = (210, 210, 145)
    threshed_rocks = color_thresh(warped, rgb_rock_min, rgb_rock_max)
    # 4) Convert thresholded image pixel values to rover-centric coords
    x_nav_px, y_nav_px = rover_coords(navigable_threshed)
    x_obs_px, y_obs_px = rover_coords(threshed_obstacles)
    # 5) Convert rover-centric pixel values to world coords
    xpos = data.xpos[data.count]
    ypos = data.ypos[data.count]
```

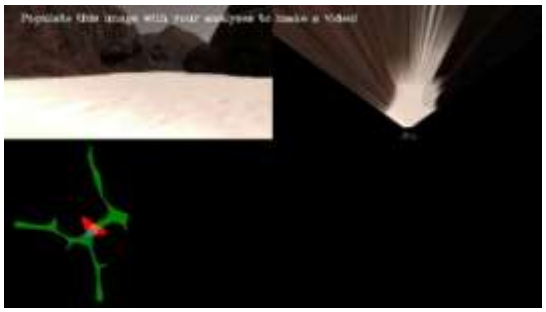
```
    yaw = data.yaw[data.count]
    w_size = data.worldmap.shape[0]
    scale = 2 * dst_size # defines how big squares are in the perspective transform
    nav_x_w, nav_y_w = pix_to_world(x_nav_px, y_nav_px, xpos, ypos, yaw, w_size, scale)
    obs_x_w, obs_y_w = pix_to_world(x_obs_px, y_obs_px, xpos, ypos, yaw, w_size, scale)
    # to account for overlap between the two
    data.worldmap[nav_y_w, nav_x_w, 2] += 1
    data.worldmap[obs_y_w, obs_x_w, 0] += 1
    nav_pix = data.worldmap[:,:, 2] > 2 # blue channel
    data.worldmap[nav_pix,0] = 0 # red channel
    data.worldmap[nav_pix,2] = 255 # blue channel
    obs_pix = data.worldmap[:,:, 0] > 6 # red channel
    data.worldmap[obs_pix,0] = 255 # red channel

    output_image = np.zeros((img.shape[0] + data.worldmap.shape[0], img.shape[1]*2, 3))
    # Populate regions of the image with various output
    output_image[0:img.shape[0], 0:img.shape[1]] = img
    warped = perspect_transform(img, source, destination)
    output_image[0:img.shape[0], img.shape[1]:] = warped # warped image in the upper right
    # Overlay worldmap with ground truth map
    map_add = cv2.addWeighted(data.worldmap, 1, data.ground_truth, 0.5, 0)
    # Flip map overlay so y-axis points upward and add to output_image
    output_image[img.shape[0]:, 0:data.worldmap.shape[1]] = np.flipud(map_add)
    # Put some text over the image
    cv2.putText(output_image,"Populate this image with your analyses to make a video!", (20, 20),
                cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
    data.count += 1 # Keep track of the index in the Databucket()
    return output_image
```

For the completeness of this write up, it is worth mentioning that inside the Online Lab tool the Python module **moviepy** was used to apply frame by frame to all images in the dataset the process_image() function. The short output video is included as part of this project submission within the folder “Writeup” under the filename **Roberto Zegers - test mapping.mp4**

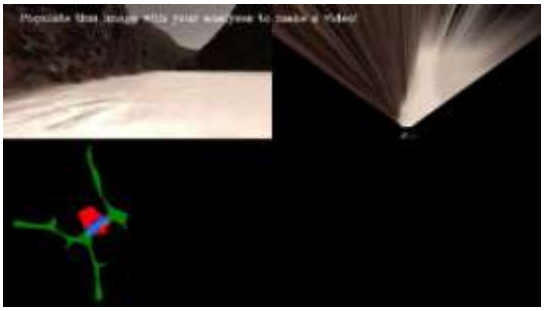
Below are four individual frames of that video:



Video Timestamp Position 0m 0s Frame 0



Video Timestamp Position 0m 0s Frame 25



Video Timestamp Position 0m 2s Frame 35

Video Timestamp Position 0m 4s Frame 42

Part 2: Autonomous Navigation and Mapping

Once mapping was working, next step was to modify **perception.py** and **decision.py** in the project folder to allow the rover to navigate and map in autonomous mode.

You can see my development process and the history of changes by clicking commits on my project Github repository: https://github.com/digitalgroove/RoboND_Project_1

I developed the solution roughly following the steps outlined below:

1.- Implemented a first working version for autonomous mode

As starting point I used the code provided on the GitHub project repository for the Unity rover search and sample return project. To get the initial code structure up and running quickly I implemented the instructions provided by the Project Walkthrough Video summarized here:

a) Modify the perspective transform helping function to output a mask of the wrapped image
I added as output to the perspective_transform a binary image that contains ones inside the camera field of view and zeros outside of it. This is later used when we use the inverted binary image of navigable terrain to identify obstacles.
For this I added next line to perspect_transform() and the mask to the return statement:

```
mask = cv2.warpPerspective(np.ones_like(img[:,:,0]), M, (img.shape[1], img.shape[0]))
return warped, mask
```

b) Add a function to identify rocks
I added a customized function that checks values greater than a certain threshold value in the Red and Green Channels and lower than a certain threshold value in the Blue channel. Later on I replaced this function by the improved version of the color_thresh() function that allows imposing both a lower and upper boundary to perform a color selection.

```
def find_rocks(img, levels=(110,110, 50)):
    rockpix = ((img[:,:,0] > levels[0]) \
                & (img[:,:,1] > levels[1]) \
                & (img[:,:,2] < levels[2]))

    color_select = np.zeros_like(img[:,:,0])
    color_select[rockpix] = 1

    return color_select
```

c) Populate the perception_step() function calling the processing functions in succession
I filled the perception_step() function within the perception.py script with the appropriate image processing functions to create a map and update the Rover() object with data.
This procedure is an almost identical as the last step in Part 1 (see above). In summary I had to:

- Define source and destination points for the perspective transform
- Apply a perspective transform
- Apply a color threshold to identify navigable terrain
- Identify obstacles (did this first as suggested: calculating the inverse of the navigable terrain)
  `obs_map = np.absolute(np.float32(threshed) - 1) * mask`
- Apply the customized function to identify rock samples
- Convert camera image pixel values to rover-centric coordinates
- Convert rover-centric pixel values to world coordinates

- Update Rover worldmap (to be displayed on right side of screen) and add the code that accounts for overlap between navigable and obstacle pixels
- Convert rover-centric pixel positions to polar coordinates to later implement steering

With these changes in place I got a first working version for autonomous mode.
To test the code I run drive_rover.py and launched the simulator in "Autonomous Mode".

2.- Replaced color_threshold() to allow for lower and upper boundary color selection

My first modification to the base solution provided by the walkthrough video was to modify the provided color_threshold function making it more general by allowing it to take in account a lower and an upper color value when selecting pixels. This allows being more specific about choosing colors and capable of identifying rock samples.
Replace:

```
obs_map = np.absolute(np.float32(threshed) - 1) * mask
```

by:

```
rgb_obs_min = (0,0,0)
rgb_obs_max = (170,170,170)
threshed_obstacles = color_thresh(warped,rgb_obs_min, rgb_obs_max)
obs_map = np.float32(threshed_obstacles) * mask
```

3.- Improved map fidelity by updating the code that accounts for overlapping

One of the main difficulties I faced was to get the portion of the code right that accounts for overlap between the blue (navigable terrain) and red (obstacles) color channels. I replaced the code provided by the Project Walkthrough Video with the code shown under Part1, Step 5.

Within the **decision_step()** function replace:

```
Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 10
Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1
```

by:

```
Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 1   # worldmap in the blue channel
Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1 # worldmap in the red channel

nav_pix = Rover.worldmap[:,:, 2] > 2 # if blue channel is > 0
Rover.worldmap[nav_pix,0] = 0 # set the red channel to zero (discard the obstacles)
Rover.worldmap[nav_pix,2] = 255 # set the blue channel to 255

obs_pix = Rover.worldmap[:,:, 0] > 6 # red channel is > 0 (navigable terrain)
Rover.worldmap[obs_pix,0] = 255 # set the red channel to 255
```

4.- Kept the robot left by adding a mask to crop navigable pixels on the right side

I wrote a mask that causes the robot to ignore a certain portion of the navigable terrain to its right when steering. As consequence the robot will tend towards turning left, keeping most of the time a wall on its left side. My first step here was to implement a helper function that remaps values in percent to pixels. This allows creating a mask using as input a percent instead an absolute value.

Helper function:

```
def remap_values(value, inMin, inMax, outMin, outMax):
    # Figure out how 'wide' each range is
    inSpan = inMax - inMin
    outSpan = outMax - outMin
    # Convert the left range into a 0-1 range (float)
    valueScaled = float(value - inMin) / float(inSpan)
    # Convert the 0-1 range into a value in the right range.
    return outMin + (valueScaled * outSpan)
```

With this helper function in place, the code that creates a mask based on horizontal and vertical starting and ending percent values is:

```
# Function that masks a range of pixels
def mask_selection(nav_binary):

    H_start_percent = 0 # percent value
    H_end_percent = 60 # percent value
    V_start_percent = 0 # percent value
    V_end_percent = 90 # percent value

    driving_mask = np.zeros((nav_binary.shape[0], nav_binary.shape[1])) # init matrix of zeros

    H_start_col = int(round(remap_values(H_start_percent, 0, 100, 0, nav_binary.shape[1])))
    H_end_col = int(round(remap_values(H_end_percent, 0, 100, 0, nav_binary.shape[1])))
    V_start_col = int(round(remap_values(V_start_percent, 0, 100, 0, nav_binary.shape[0])))
    V_end_col = int(round(remap_values(V_end_percent, 0, 100, 0, nav_binary.shape[0])))
    driving_mask[V_start_col:V_end_col,H_start_col:H_end_col] = 1 # select rows & cols

    mask_nav = nav_binary * driving_mask # apply mask

    return mask_nav
```

Note: One further improvement that can be made is that this function could accept as input the horizontal and vertical percent values. This would make it more general and reusable.

Example function output:

```
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Apply the mask to crop the navigable terrain detected:

```
navigable_masked = mask_selection(navigable_threshed)
```

To display the new cropped version of navigable terrain on the screen replace:

```
Rover.vision_image[:,:,2] = navigable_threshed * 255
```

By:

```
Rover.vision_image[:,:,2] = navigable_masked * 255
```

Output image, current image of cropped navigable terrain ahead shown in a green circle:

Finally to be able to steer the robot using the cropped navigable terrain we convert the masked image to rover centric coordinates, apply a transform to polar coordinates and store those into Rover.nav_angles:

```
x_navM_pix, y_navM_pix = rover_coords(navigable_masked)
```

```
dist_navM, angles_navM = to_polar_coords(x_navM_pix, y_navM_pix)
Rover.nav_angles = angles_navM
```

You can run and test this function on the Jupyter Notebook which I included as part of my project submission under the folder “Jupyter Notebook” and filename "Custom_functions.ipynb".

With these changes in place the robot will keep left most of the time and it can achieve high efficiency mapping terrain and looking for rock samples.

5.- Mapped terrain only when the rover is on even ground

Here I implemented the code necessary to avoid the robot mapping the terrain when roll and pitch angles are not close to zero (e.g. braking or turning hard). For this I used an “if” statement that checks that roll and pitch are close to zero and only then uses the images for mapping.

```
# check that the rover is on even ground
if ((Rover.pitch <= 1) or (Rover.pitch >= 359)) and ((Rover.roll <= 1) or (Rover.roll >= 359)):
    # account for overlap between the two
    Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 1   # worldmap in the blue channel
    Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1 # worldmap in the red channel
    nav_pix = Rover.worldmap[:,:, 2] > 2 # if the blue channel is > 0 (navigable terrain)
    Rover.worldmap[nav_pix,0] = 0 # set the red channel to zero (discard the obstacles)
    Rover.worldmap[nav_pix,2] = 255 # set the blue channel to 255
    obs_pix = Rover.worldmap[:,:, 0] > 6 # if the red channel is > 0
    Rover.worldmap[obs_pix,0] = 255 # ...set the red channel to 255
```

Other smaller modifications I made at this point:
- Experimented with modifying the portion of cropped navigable pixels. Expanded mask of navigable terrain from 55% to 60% horizontally and tested the robots behavior.

6.- Labeled each “action” executed inside every “mode” in the robots decision tree

Within the **decision.py** script I added an action label with a name to be able to identify performing actions inside each different robot mode (forward and stop), such as throttle, coast, brake, turn in place or go again.
Example code (just a code fragment, not the full implementation):

```
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:
        Rover.action = 'throttle'
```

Then I expanded the class defined in drive_rover.py accordingly.
Next I plotted the current mode and action on top of the map display. Inside supporting_functions.py I added the text to output:

```
    cv2.putText(map_add,"Mode: "+str(Rover.mode), (0, 100),
                cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
    cv2.putText(map_add,"Action: "+str(Rover.action), (0, 115),
                cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
```

Output image, current “mode” and “action” plotted over the map (shown in a red circle):

Additionally I made smaller changes such as refactor rock detection, I stored values inside the Rover object instead of using local variables. I also tried using self.rock_ang and self.rock_dist to steer the robot, but later dismissed this method.

7.- Added functionality to steer the robot toward rocks detected in its vision field

Within the **decision.py** script I added a condition to check for rocks before checking the extent of navigable terrain.

```
if np.count_nonzero(Rover.rock_ang) > 1 and not Rover.collected:
```

With this change in place, if rocks are detected the Rover changes to mode = 'Go to rock'. Inside this mode the steering angle becomes the average angle of rock pixels clipped to +/- 15 degrees. I also added three different possible actions: braking, throttle to rock and coast to rock:

```
# Check for rocks
if np.count_nonzero(Rover.rock_ang) > 1 and not Rover.collected:
    Rover.mode = 'Go to rock'
    # Set steering to average angle clipped to the range +/- 15
    Rover.steer = np.clip(np.mean(Rover.rock_ang * 180/np.pi), -15, 15)
    if Rover.vel > 0.6:
        Rover.action = 'breaking'
        Rover.throttle = 0
        Rover.brake = Rover.brake_set
        Rover.steer = 0
    if Rover.vel < 0.4:
        Rover.action = 'throttle to rock'
        # Set throttle value to throttle setting
        Rover.brake = 0
        Rover.throttle = 0.1
    else: # Else coast
        Rover.action = 'coast to rock'
        Rover.brake = 0
        Rover.throttle = 0
# If no rocks where found check the extent of navigable terrain
elif 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:
         Rover.mode = 'forward'
```

Then I added an else condition that executes when the robot is neither in forward nor in stop mode, and no rock samples are detected. This condition executes basically when the robot is driving towards a rock but then fails to detect the rock in one or more image frames. The logic added keeps the robot going to the same direction it was driving before for a certain time and prevents the robot from changing the mode and abort moving towards the rock sample.
A timeout counter called “elsecounter” was implemented for this functionality (Note: later it was renamed to “rock_timeout”). If a rock is detected again before this counter exceeds the timeout value, the robot continues moving towards the rock with an updated steering angle. If no rock is detected and the counter exceeds the set timeout value the robot mode changes to “forward”.

```
else:
    Rover.elsecounter += 1 # to keep going to rock
    Rover.action = 'else 1...'
    if Rover.mode == 'Go to rock':
        Rover.steer = Rover.steer_cache
        if Rover.elsecounter > 10:
            Rover.mode = 'forward'
            Rover.elsecounter = 0
```

Another change was made to perception.py: each time a rock pixel is detected, the average value is stored as a variable that acts as cache steering angle (for the situation explained above):

```
if Rover.rock_ang is not None:
     Rover.steer_cache = np.clip(np.mean(Rover.rock_ang * 180/np.pi), -15, 15)
```

The capability to pick up samples use the functionality that's already in place but complemented by a command to change to ‘forward’ mode, to clean all stored rock angles and to set flag rock collected to True once the rock was picked up:

```
# 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
     Rover.mode = 'forward'
     Rover.rock_ang = None
         Rover.collected = True
```

Also the Rover class (drive_rover.py) was updated to keep track of variables in accordance to the changes made to the state machine (decision.py):

```
self.collected = False # Flag rocks collected
self.elsecounter = 0
self.steer_cache = None # To store the average angle to a rock
```

Once these new features passed the testing, I fine tuned the velocity when the robot moves towards a rock. Also I added a new condition inside decision.py to stop the robot when it is near a rock using the Rover.near_sample flag:

```
if Rover.near_sample:
    Rover.action = 'braking'
    Rover.throttle = 0
    Rover.brake = Rover.brake_set
    Rover.steer = 0
```

Next, inside the timeout counter used to keep the steering angle towards a rock I added a condition to brake when the velocity is too high:

```
else:
    Rover.elsecounter += 1
    Rover.action = 'else 1...'
    if Rover.mode == 'Go to rock':
        Rover.steer = Rover.steer_cache
        if Rover.vel > 0.6:
            Rover.action = 'else 1 breaking'
            Rover.throttle = 0
            Rover.brake = Rover.brake_set
            Rover.steer = 0
        if Rover.elsecounter > 60:
            Rover.mode = 'forward'
            Rover.elsecounter = 0
```

To make sure that the robot always stops when it is near a rock sample I replaced:

```
if Rover.vel > 0.6:
```

By this:

```
if Rover.vel > 0.6 or Rover.near_sample:
```

As seen in this code snippet:

```
else:
    Rover.elsecounter += 1
    Rover.action = 'else 1...'
    if Rover.mode == 'Go to rock':
        Rover.steer = Rover.steer_cache
        if Rover.vel > 0.6 or Rover.near_sample:
            Rover.action = 'Brake near sample'
            Rover.throttle = 0
            Rover.brake = Rover.brake_set
            Rover.steer = 0
```

Other minor adjustments that I made to the code at this stage were:
- Fixed a bug in the mask_selection() function
- Increased throttle to 0.5 allowing for a higher maximum velocity to minimize total time.
- Increased mask of navigable pixels to 95 percent vertically.

8.- Masked rocks to consider only rock samples detected on the front and left side

First I made the changes necessary to effectively use the general threshold function color_thresh() instead of the custom function find_rocks() to detect rocks. For this, inside perception_step():

```
    rgb_rock_min = (110,110, 0)
    rgb_rock_max = (210, 210, 50)
    threshed_rocks = color_thresh(warped, rgb_rock_min, rgb_rock_max)
```

Then I made the change to use mask_selection() (see step nr. 4 above) to mask the rocks detected (crop the right-most side). Again inside perception_step() I added:

```
rocks_masked = mask_selection(threshed_rocks)
```

Then I made a change to calculate the robot centric coordinates:

```
rock_x, rock_y = rover_coords(rocks_masked)
```

And then we use the masked rocks to transform to polar coordinates to get the angles to navigate:

```
Rover.rock_dist, Rover.rock_ang = to_polar_coords(rock_x, rock_y)
```

Finally I also updated the code that checks for rocks as seen in this code snippet:

```
#See if we can find some rocks
rock_x, rock_y = rover_coords(rocks_masked)
Rover.rock_dist, Rover.rock_ang = to_polar_coords(rock_x, rock_y)
if np.count_nonzero(Rover.rock_ang) < 3:
    Rover.rock_ang = None
if rocks_masked.any(): # gives True if at least 1 element of rocks_masked is True
    rock_x_world, rock_y_world = pix_to_world(rock_x, rock_y, Rover.pos[0], Rover.pos[1], Rover.yaw,
world_size, scale)
    rock_idx = np.argmin(Rover.rock_dist) # minimum distance rock pixel
    rock_xcen = rock_x_world[rock_idx]
    rock_ycen = rock_y_world[rock_idx]
    if Rover.rock_ang is not None:
        Rover.steer_cache = np.clip(np.mean(Rover.rock_ang * 180/np.pi), -15, 15)
    Rover.worldmap[rock_ycen,rock_xcen, 1] = 255 # update the rover world map to be 255 at center
    Rover.vision_image[:,:,1] = rocks_masked * 255 # put those rock pixels onto the vision image
```

Other smaller changes I made at this point were:
- Fine tune braking/throtteling when near sample
- Rename elsecounter to rock_timeout
- Set back max speed of the robot to 2 mt/s a results were not good.

9. Added mode 'unstuck'.

Finally I added a “unstuck” mode to the decision tree. This mode will help in situation where the robot cannot move forward because it is stuck hitting a rock or driving on uneven terrain. To implement this functionality, I wrote a timer that increases when the rover is applying full throttle but its speed keeps below 0.1:

```
    if Rover.vel < Rover.max_vel:
                    Rover.action = 'throttle'
                    # Set throttle value to throttle setting
                    Rover.throttle = Rover.throttle_set
                    if Rover.vel < 0.1:
                        Rover.navthrottle_timeout += 1
                        if Rover.navthrottle_timeout > 100:
                            Rover.mode = 'unstuck'
                            Rover.navthrottle_timeout = 0
```

Also inside the **decision_step()** function within the decision.py script I added the conditional statements that includes the logic to be executed when the rover is in “unstuck” mode:

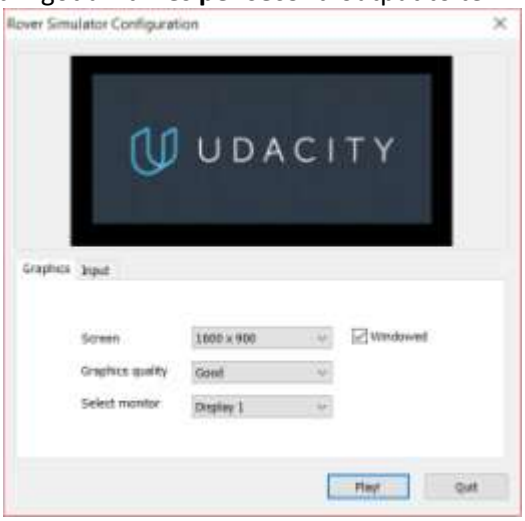
```
    elif Rover.mode == 'unstuck':
        # If we're in stop mode but still moving keep braking
        Rover.unstuck_timeout += 1 # for debugging
        if Rover.unstuck_timeout < 60:
            Rover.action = 'reversing'
            Rover.throttle = -1
            Rover.brake = 0
            Rover.steer = 15
        elif Rover.unstuck_timeout < 65:
```

```
            Rover.throttle = 0
            Rover.brake = Rover.brake_set
        elif Rover.unstuck_timeout < 100:
            Rover.action = 'pushing'
            Rover.brake = 0
            Rover.throttle = 2
            Rover.steer = 0
        else:
            Rover.mode = 'forward'
            Rover.unstuck_timeout = 0
```

With the code explained above, the rover does a reasonable job of navigating mapping, and collecting rock samples. It is **able to map at least 90% of the environment at 70% fidelity or more** and locate and pick up most of the time at least five of the rock samples.

Program execution: launching in autonomous mode

I executed the Rover Simulator with a screen resolution of **1600 x 900** and graphics quality set to **good**. I got a **frames per second** output to terminal from 11 to 25 fps.



Execute:

$ python drive_rover.py

To create a video output of the Rover in Project 1 the screen was directly captured using Snagit as screen recording program. The video is included as part of my project submission within the folder "Writeup" under the filename **Roberto Zegers - Sample Return Challenge.mp4**