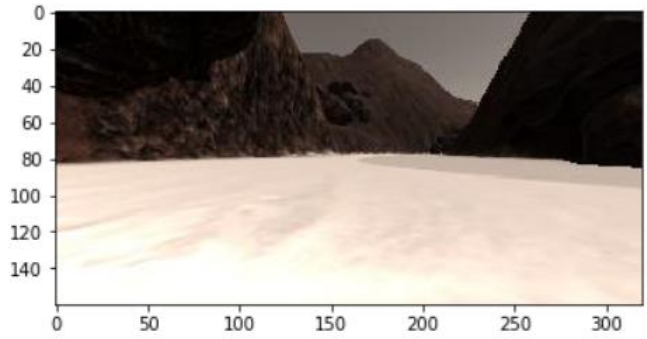
Notebook Analysis

The notebook has been very useful companion to the lessons. It was possible to familiarize with the tools and libraries of functions needed and the necessary steps involved in the perception phase. In particular processing of the images as a multi layered matrix. This enables getting information which helps the decision process. The function in the notebook have been completed and where possible improved. Examples of executions are the following:

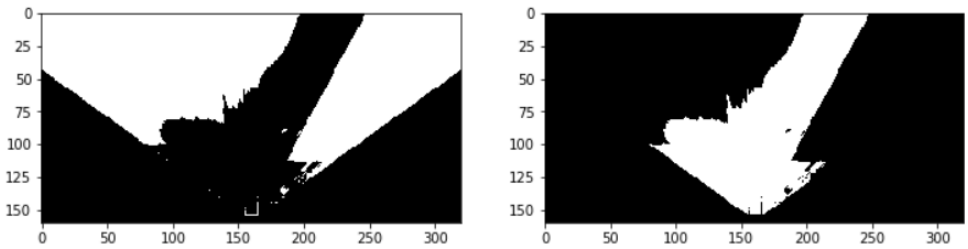
```
Total of 1042 images in the list.
channels = 3, rows = 160, columns = 320
min_val = 0, max_val = 255, mean_val = 130
```



Detection of obstacles, navigation parts and rocks has been improved by better threshold values.

```
def color_thresh(img, rgb_thresh=(160, 160, 160)):
    above_thresh = (img[:,:,0] > rgb_thresh[0]) \
                 & (img[:,:,1] > rgb_thresh[1]) \
                 & (img[:,:,2] > rgb_thresh[2])

    color_select = np.zeros_like(img[:,:,0])
    color_select[above_thresh] = 1
    return color_select
#================================================================

# Using threshold of RGB > 160 as it does a nice job of identifying ground pixels only
nav_map = color_thresh(warped)

# create the obstacles map (inverting the nav_map mainly)
obstacles_map = np.absolute(np.float32(nav_map) - 1) * mask

scipy.misc.imsave('./output/warped_threshed.jpg', nav_map*255)
fig = plt.figure(figsize=(12,3))
plt.subplot(121)
plt.imshow(obstacles_map, cmap='gray')
plt.subplot(122)
plt.imshow(nav_map, cmap='gray')
```

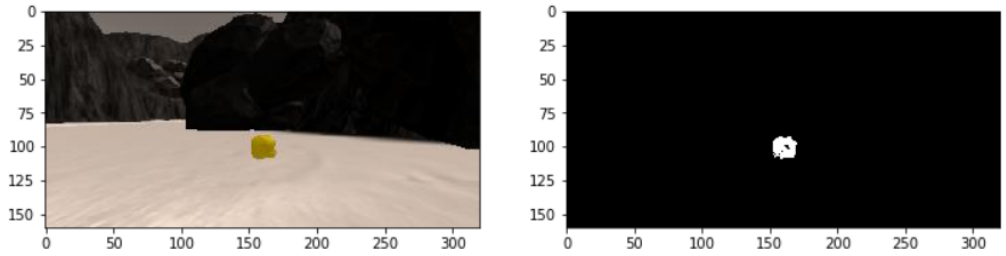
```
<matplotlib.image.AxesImage at 0xcf943c8>
```



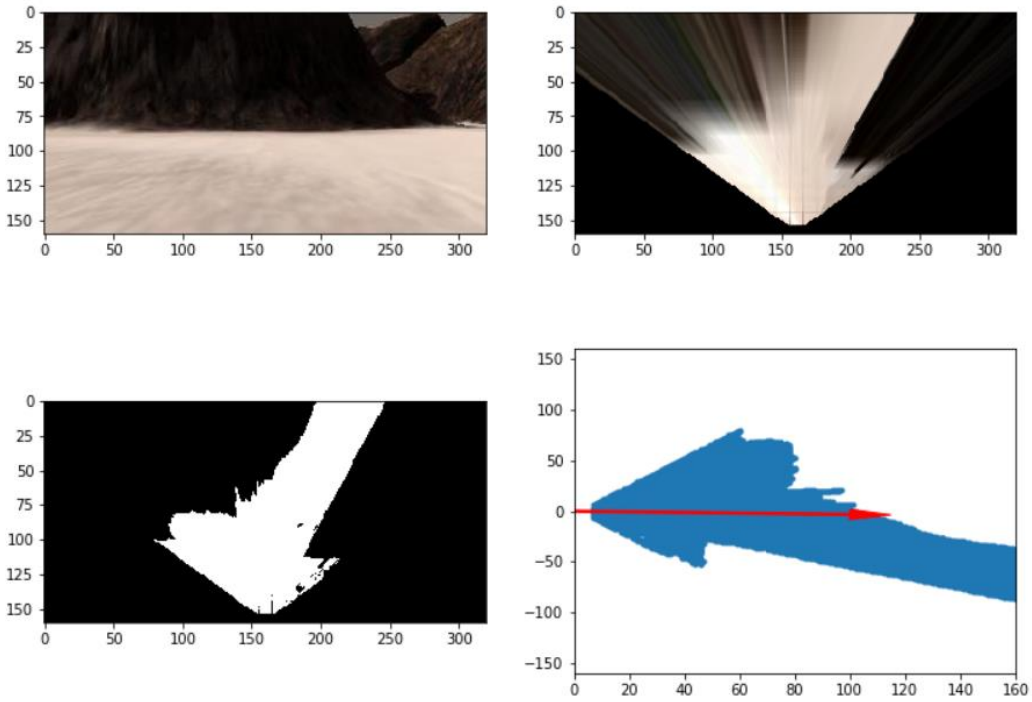
A dedicated function to detect the rocks has also been inserted in the notebook.

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

    # create a black image
    color_select = np.zeros_like(img[:,:,0])
    # color with white all the rock pixels
    color_select[rockpix] = 1
    return color_select
#================================================================

#select the rock from the environment
rock_map = find_rocks(rock_img)

fig = plt.figure(figsize=(12,3))
plt.subplot(121)
plt.imshow(rock_img)
plt.subplot(122)
plt.imshow(rock_map, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0xc25fc50>
```



The simulator has been used in training mode to record enough variety of camera images. An overall sample of 1042 images has been recorded and stored in the dataset directory of this submission. The images have been analyzed by adopting/improving the various basic functions of transformation. Direction of movement has been calculated from all the angles of each coordinate within the navigable terrain. The average angle is a first choice for direction of movement.



The final image has been customized via the image_process() function to reflect this work.

Autonomous Navigation and Mapping

The first step has been to run the already available scaffolding code by adding the different parts as from several hints and templates I could see in the Slack and chat with other students. This was useful to understand the mechanism of communication between the Simulator used in autonomous mode and my python code/process.

Code organization

The template code shows lot of repetitive tasks; somehow error prone, therefore the code has been organized in 3 main files:

- nd_functions.py: containing all of the functions collected together
- nd_classes.py: containing the RoverStatus() class and the DataBucket() class.
- drive_rover.py: which is implements the connection of the main functions and the simulator via socket.

As part of the code optimization, I have created several get/set methods inside the RoverStatus() class and use those in the various decision steps.

The simulator provides the data dictionary and functions like the **perception_setp()**, **rover_update()**, and **decision_step()** are been called in order to process the data, get insight and finally make decision to driven the next steps of the rover.

The perception step requires reuse of the previous work done in the notebook. It allows for the detection of navigable terrain, obstacles, rocks and their angles and distances which are need for the decision step. The update() function is then called to update the RoverStatus() class. The same can be told for the **output_image()**, this can actually be customized based on personal flavors.

The most interesting part is the decision step. Here the main strategy is to get all the pixels which make a navigable terrain. For each of them get the distance to the rover and the relative angle. The velocity vector is then oriented in line with the average angle of navigable terrain pixel which keeps the rover more or less in the middle of the navigable map.

Decision tree:

The decision tree is made of 5 parts:

1. Detection of samples: when a sample is detected, the rover would stop, then move towards the rock and pick it up. I have seen that this part performs reasonably on a big number of runs. However, this is not optimal as it still can have issues if there are 2 rocks which are detected.

2. Return Home: at the beginning the initial position it is stored. When the rover has picked up all the 6 samples, then it travels back to find the initial position. Home is considered if we enter a circle having as center the initial coordinates and a radius R. in the code this is 25.

3. Stuck: In this case the rover is in forward mode but is not moving. We have throttle but the speed is null. If this the case the rover waits for 4 seconds and try to rotate hopping to find a good angle so to move on.

4. Forward movement and stop: Here we would check if there is navigable terrain and then move on the average angle of the navigable terrain. There is not much added to the initial scaffolding code.

5. Stop: in case of stop condition the rover is rotated in the direction where there is more navigable parts. Not much here with respect to the scaffolding code.

How to improve

At the moment of this write-up, I am driven from the deadline and submit the project. In an ideal case where more time is available and also for having even more fun (I have a full time job and a family with 3 kids) I would like to optimize the following:

Speed of exploration of the map.

Here it is required a better routine on the actuation… maybe a set of functions which take care of acceleration/deceleration in alignment with the distance to the targets (obstacles/samples/etc…)

Improved exploration algorithm

What I want is to avoid revisiting areas which has already been visited. Following the same side of the wall like in a labyrinth might be a good starting point. Another option is keeping track of the previous decisions made, whenever the min/max navigable angles are in a wider range then store the decision which was taken. Eventually follow at first the opposite direction. The choices are actually many.

Configuration:

Simulator: 1920x1200 (Fantastic), FPS: 28

Results:

With the test I have done I can explore >99% of the map, collect all the 6 samples, returned home and have a fidelity figure of > 70%.