

# Project: Search and Sample Return Writeup

---

Created by: [Ananta Kumar Roy](#)

---

Link to the video - <https://www.youtube.com/watch?v=OmjGe6bz-YM>

The goals / steps of this project are the following:

- **Training / Calibration**

1. Download the simulator
  - 1.1. Mac([https://s3-us-west-1.amazonaws.com/udacity-robotics/Rover+Unity+Sims/Mac\\_Roversim.zip](https://s3-us-west-1.amazonaws.com/udacity-robotics/Rover+Unity+Sims/Mac_Roversim.zip))
  - 1.2. Linux([https://s3-us-west-1.amazonaws.com/udacity-robotics/Rover+Unity+Sims/Linux\\_Roversim.zip](https://s3-us-west-1.amazonaws.com/udacity-robotics/Rover+Unity+Sims/Linux_Roversim.zip))
  - 1.3. Windows([https://s3-us-west-1.amazonaws.com/udacity-robotics/Rover+Unity+Sims/Windows\\_Roversim.zip](https://s3-us-west-1.amazonaws.com/udacity-robotics/Rover+Unity+Sims/Windows_Roversim.zip))
2. Take data in "Training Mode"
3. The functions in the Jupyter test notebook were tested before using them in the different files in the code directory
4. Added functions to detect obstacles and samples of interest (golden rocks)
5. Filled in the ``process_image()`` function with the appropriate image processing steps (perspective transform, color threshold etc.) to get from raw images to a map. The ``output_image`` that was created in this step demonstrates the proper working of the mapping pipeline.
6. Used ``moviepy`` to process the images in the saved dataset in `test_dataset` folder with the ``process_image()`` function.
7. The submission includes a movie of the final working code.

- **Autonomous Navigation / Mapping**

- \* Filled in the ``perception_step()`` function within the ``perception.py`` script with the appropriate image processing functions to create a map and updated ``Rover()`` data
- \* Filled in the ``decision_step()`` function within the ``decision.py`` script with conditional statements that take into consideration the outputs of the ``perception_step()`` in deciding how to issue throttle, brake and steering commands.
- \* The perception and decision functions were iterated until the rover did a reasonable (atleast 40% of terrain mapped with atleast 60% fidelity) job of navigating and mapping.

**Here** I will consider the rubric points individually and describe how I addressed each point in my implementation.

## **Writeup / README**

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

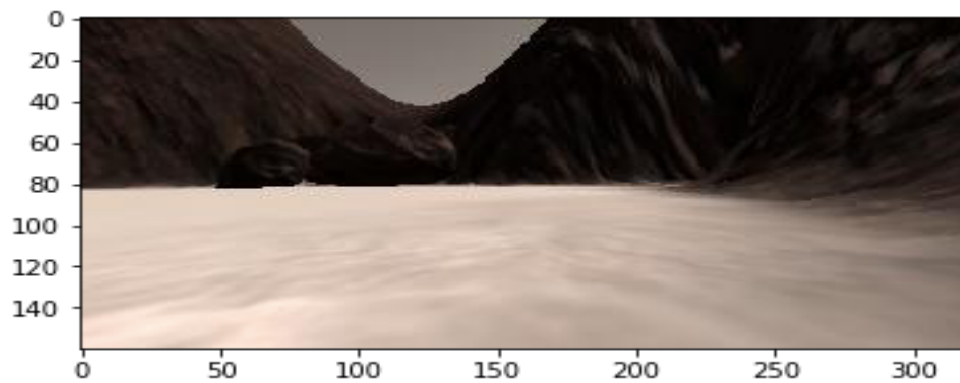
This is the README in pdf format which is based on the template provided.

## **Notebook Analysis**

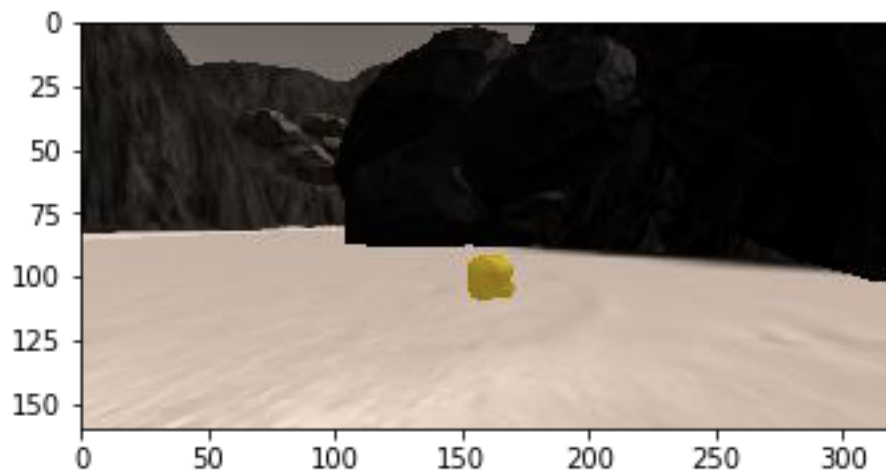
1. Run the functions provided in the notebook on test images (first with the test data provided, next on data you have recorded). Add/modify functions to allow for color selection of obstacles and rock samples.

\* The **first** step involved is the choice of a random image from the test images dataset

```
<matplotlib.image.AxesImage at 0x121919fd0>
```



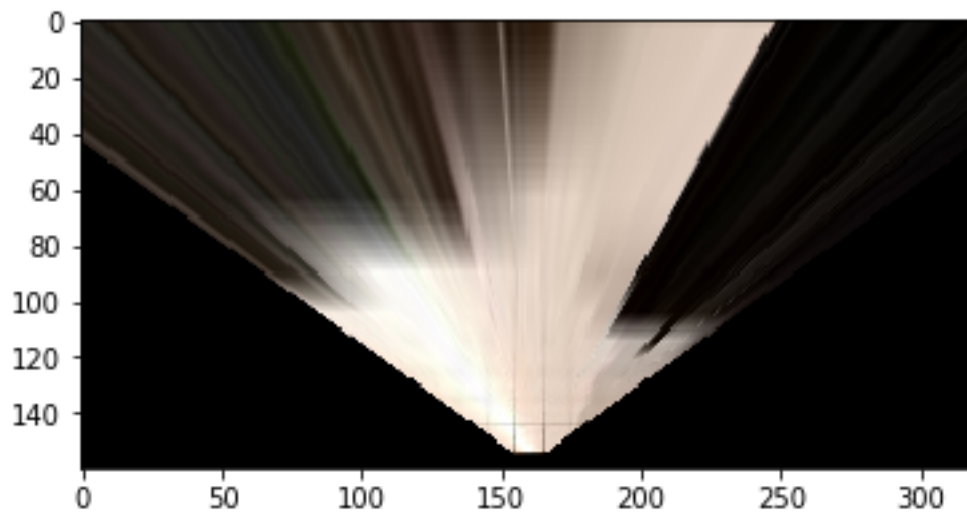
*Example image of terrain*



*Example image of terrain with rock in front*

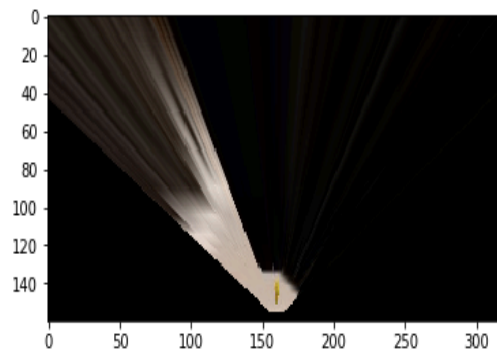
\* **Second**, we perform perspective transform on the image to change the viewing position to a top down

```
<matplotlib.image.AxesImage at 0x121c6f438>
```



*Perspective transform on terrain image*

```
Out[28]: <matplotlib.image.AxesImage at 0x11da442eb8>
```

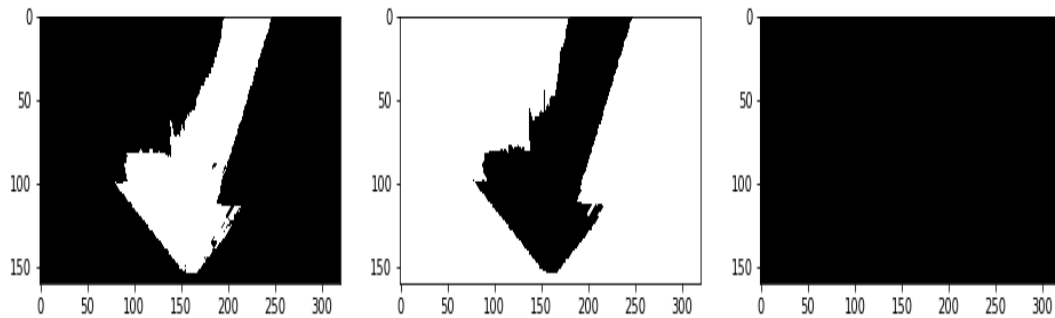


**Color Thresholding**

*Perspective transform applied on image of terrain with rock*

- **Third**, we apply the thresholding logic to detect navigable terrain, rocks and obstacles. The screenshot of the rock and obstacles detection code is also attached as required by the rubric.

<matplotlib.image.AxesImage at 0x121da96d8>

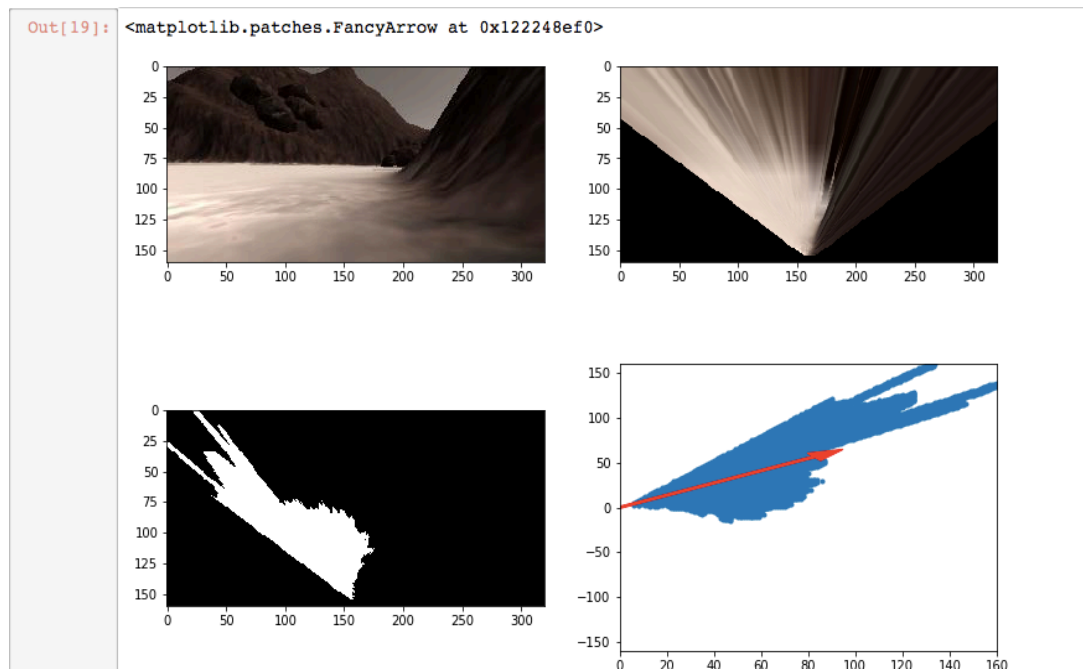


*Color thresholded images-Navigable ,obstacle and rock(here no rocks are present)*

```
17 #The compliment of the navigable terrain are the obstacles
18 def obstacles(img,rgb_thresh=(160,160,160)):
19     color_select = np.zeros_like(img[:, :,0])
20     below_thresh = (img[:, :,0] < rgb_thresh[0]) & (img[:, :,1] < rgb_thresh[1]) & (img[:, :,2] < rgb_thresh[2])
21     color_select[below_thresh] = 1
22     return color_select
23
24 #Detecting the rocks
25 def rockFind(img,rgb_thresh=(120, 120, 50)):
26     color_select = np.zeros_like(img[:, :,0])
27     above_thresh = (img[:, :,0] > rgb_thresh[0]) & (img[:, :,1] > rgb_thresh[1]) & (img[:, :,2] < rgb_thresh[2])
28     color_select[above_thresh] = 1
29     return color_select
30
```

*Obstacle and Rock detection functions*

\* **Fourth**, the thresholded transformed image is then viewed in rover coordinate system. Also, we need the polar coordinate system to get the distance of navigable terrain and the mean angle at which the rover will steer itself. Another random image was used for this purpose in the test notebook.



*Perspective transform, color thresholding and rover coordinate frame image*

2. Populated the `process_image()` function with the appropriate analysis steps to map pixels identifying navigable terrain, obstacles and rock samples into a worldmap. Ran `process_image()` on the test data using the `moviepy` functions provided to create video output of the result.

## **Autonomous Navigation and Mapping**

1. Filled in the `perception_step()` (at the bottom of the `perception.py` script) and `decision_step()` (in `decision.py`) functions in the autonomous mapping scripts. A brief description of the different functions used is provide below :

- The first function that we use in the perception step is the perspective transform. It is accomplished by first defining the source and destination points. The source points are calculated roughly by the four corners of the grid that is in front of the rover camera. The destination points are

calculated by using the source points .The perspective transform is then calculated by the `getPerspectiveTransform` function. The `warpPerspective` function then helps to get the warped image.

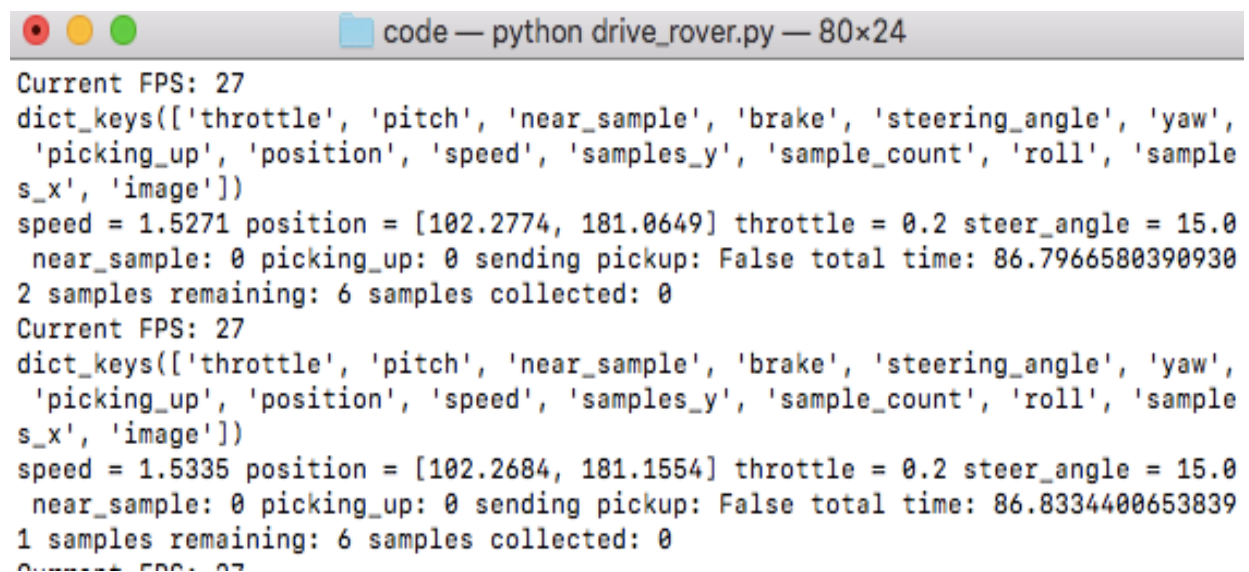
- Then we apply color thresholding to the warped images to find the navigable terrain, obstacles and the golden rocks
- The next step involves the display of the images obtained from the `perception_step` on screen in the autonomous mode. The `perception_step` returns the navigable terrain and the obstacles as an arrays with values between 0 and 1.Hence, we need to multiply this with 255 to obtain an image for the specific channel- red for the obstacles and blue for the navigable terrain
- Step #5 of the `perception_step` involves conversion to the rover coordinate frame of reference.
- Step #6 is the conversion to the world coordinate system from the rover coordinate frame using the `pix_to_world` function
- The next step involves updating the world map
- We also need to make use of the polar coordinate system to calculate the pixel distance and angle for the navigable terrain
- The `decision_step()` function in the `decision.py` file handles all the decision making of the rover so that it can traverse the terrain more efficiently. For a non zero value of the navigable angle that we calculated in the last step of `perception_step()`, there are two major conditions- whether the rover is in a forward mode already or whether the rover is in the stop mode.
  - The approach that I employed for rover motion is to try as much as possible for the rover to hug the walls. This was done by adding in a bias value to the mean value of the angles for the navigable terrain. The

choice of bias value is a random value at present in my code(the value 8 in my case).

- The rover steering value must be clipped between 15 and -15 and hence we make use of np.clip function.

2. The rover is now ready to be launched in the autonomous mode . The simulator is run in “Good ” graphics quality with a screen resolution setting of 640 X 480. Also, the frame rate for the simulator in my case is 27 fps which can be observed from the attached image.

The results are as per the rubrics for a mapped area of > 40% , the fidelity of 66% is achieved. Also, the rock detection works perfectly and a total of 4 rocks were detected in the 46% mapped area.



```
code — python drive_rover.py — 80x24
Current FPS: 27
dict_keys(['throttle', 'pitch', 'near_sample', 'brake', 'steering_angle', 'yaw',
'picking_up', 'position', 'speed', 'samples_y', 'sample_count', 'roll', 'sample
s_x', 'image'])
speed = 1.5271 position = [102.2774, 181.0649] throttle = 0.2 steer_angle = 15.0
near_sample: 0 picking_up: 0 sending pickup: False total time: 86.7966580390930
2 samples remaining: 6 samples collected: 0
Current FPS: 27
dict_keys(['throttle', 'pitch', 'near_sample', 'brake', 'steering_angle', 'yaw',
'picking_up', 'position', 'speed', 'samples_y', 'sample_count', 'roll', 'sample
s_x', 'image'])
speed = 1.5335 position = [102.2684, 181.1554] throttle = 0.2 steer_angle = 15.0
near_sample: 0 picking_up: 0 sending pickup: False total time: 86.8334400653839
1 samples remaining: 6 samples collected: 0
Current FPS: 27
```

Though the project meets the minimum project rubric requirements, there are a few points which can be improved .Two of them which I would like to keep working on this project include-

- The wall hugging technique can be improved. Instead of choosing a random value by hit and trial, the information about the navigable pixels ahead can be used more efficiently by ignoring the far away pixels and concentrating on the pixels nearer to the rover camera.
- The path already traversed by the rover should not be repeated. So, the next area which I would like to work on is to



create a logic where the rover does not traverse the path already taken. This would help to better the fidelity percentage.