Search and Sample Return

The Search and Sample return project consists of two phases: a learning exercise and experimentation environment in a Jupyter notebook (Rover\_Project\_Test\_Notebook.ipynb) and a set of scripts which allow the RoverSim to operate in autonomous mode.

The project is available in github at https://github.com/carlosrodriguez6/RoboND-Rover-Project.git

# Notebook Analysis

The project notebook builds on the earlier exercises to build a ‘perception’ pipeline that allows the rover to create a map of real world coordinates as it travels the simulator’s 3D environment.

The notebook was initially run without modifications using sample images and data included with the original git project. Then it was modified to allow for color selection of obstacles and rock samples. Lastly a “training” run was made in the simulator and images from this run were used as inputs for predefined and modified functions.

## Imports and setup

The first two or three cells of the notebook simply shows how to load an image into matplotlib.image and render it to screen using plt.imshow(), and thus I won’t go into detail here, except to mention that in order to make the calibration cell useful I had to add the %matplotlib notebook directive right after the matplotlib import statements.

## Calibration images

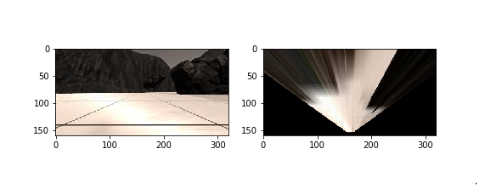
The next code cell is provided to show calibration images. These allow to:

* Define the source and destination frames of a perspective transform, and
* Determine the color thresholds that define pixels belonging to a rock.

## Perspective Transform

For the initial iteration the values obtained by evaluating the x-y positions of the corners of the sample grid square were consistent with the prepopulated values in the perspective transform cell, so those were left unmodified.

The following image shows the pre-packed sample grid image against the warped version:



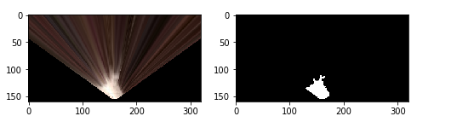
## Color Thresholds

The Color Thresholding cell provides a function with lower RGB bound to find terrain and output the results of the operation to a binary image. For the initial iteration, the default values for the RGB threshold appeared adequate and were left unmodified.

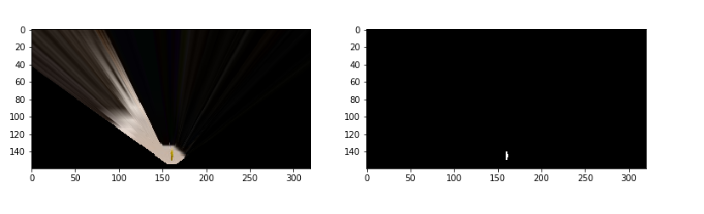
Example data:



From training run:



For rocks, I found that the more obvious approach was to define a function that logically combined upper and lower thresholds for all channels. This led to the definition of the color\_thresh\_upper\_lower() function, which appears to work well. The following is the output of the function compared to the already warped example\_rock1 image:



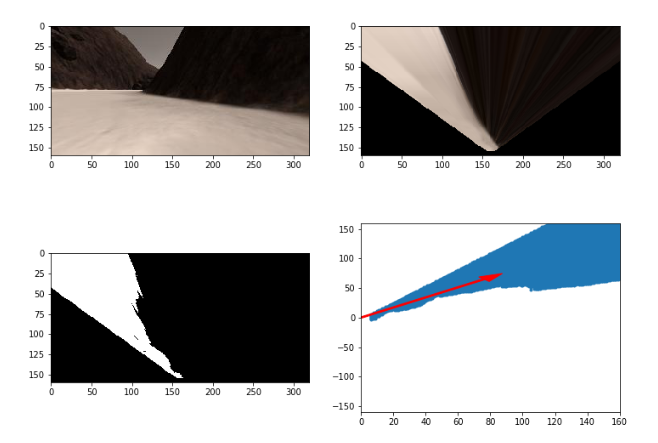
For obstacle color selection I followed the recommendation in the notebook and simply selected the opposite of the ‘ground’ pixels:

threshold\_obstacles = np.absolute(np.float32(threshed) - 1)

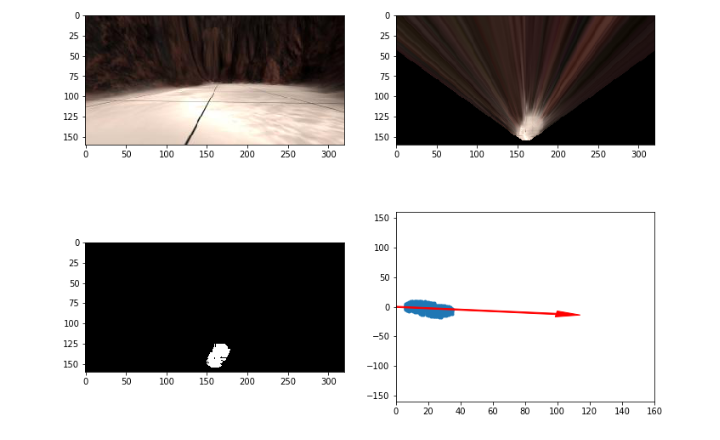
## Coordinate Transforms

The cell for coordinate transformations did not require any changes. This cell defines a conversion to a rover centric coordinate system for the pixel positions of the thresholded image, as well as a function to convert these positions to polar coordinates centered on the rover. The following figure shows the warp and threshold operations for terrain, as well as a final conversion to coordinates centered on the rover, overlaid with the average of the polar angles from the rover to all the pixels in that terrain sample:

Example Data:



Training run:



## The process\_image loop

The next cell sets up the data structures needed to complete the second part of the notebook assignment, which uses the .csv log and the images captured during the sample and training runs to repeatedly call the process\_image() function. At the end of these iterations a reasonable map of the terrain traversed during the runs should be achieved, including any rocks found along the way, as well as a rough map of the obstacle features. Since the only change made to this cell was changing the file paths for the log files and the IMG directory I won’t go into detail here.

The process\_image() function is meant to leverage the previously defined functions in the notebook to perform a perception or mapping step for each image captured by the rover’s camera. In addition to filling in the necessary code to fulfill the requirements of the assignment, I added some functionality to make it easier to visualize what was going on in each step. In the following summary all line numbers refer to the process\_image notebook cell.

Firstly, I increased the size of the mosaic image for each frame (line 25):

output\_image = np.zeros((img.shape[0] + data.worldmap.shape[0], img.shape[1]\*3, 3))

I wanted to show the original, warped and thresholded terrain image on the first row so I wrote a helper function to convert the binary image to RGB (binary\_to\_RGB in the notebook) so I could add it to the mosaic, then I modified the output image statements:

# Put the original image in the upper left hand corner

output\_image[0:img.shape[0], 0:img.shape[1]] = img

# Add the warped image next to it

output\_image[0:img.shape[0], img.shape[1]:2\*img.shape[1]] = warped

# convert the thresholded image for the terrain to RGB so we can show it in the mosaic

# and put it in the first row, third column

output\_image[0:img.shape[0], 2\*img.shape[1]:] = binary\_to\_RGB(threshed)

For both the included rock image samples and from the rocks found during my training run, I arrived at upper and lower RGB thresholds of (255, 255, 50) and (110, 110, 0) for the rock’s yellow. After applying the color\_thresh\_upper\_lower() function to the warped image, I stitched the threshold image for the rocks in the second row, next to the ground truth map. Since the rock pixels rendered as a tiny blip, I made the background of that image a little lighter so it would stand out (lines 52-56).

gray = (yellow\_threshed == 0)

# paint non-rock pixels gray so we can tell the difference

ret\_rock[gray] = [20,20,20]

# and output it to the mosaic image

output\_image[img.shape[0] + 40:,data.worldmap.shape[1]:(data.worldmap.shape[1] + 320)] = ret\_rock

I obtained the obstacle thresholded image by simply negating the terrain results, but I decided to not display them separately and instead concentrated on the changes needed to populate the map. Firstly I converted the thresholded images to rover coordinates (lines 62 – 64).

xpix, ypix = rover\_coords(threshed)

xpix\_rock,ypix\_rock = rover\_coords(yellow\_threshed)

xpix\_obs,ypix\_obs = rover\_coords(threshold\_obstacles)

The rover coordinate arrays were converted to world coordinates (lines 68 – 73).

navigable\_x\_world, navigable\_y\_world = pix\_to\_world(xpix, ypix, data.xpos[data.count],

data.ypos[data.count], data.yaw[data.count], 200, 10)

rock\_x\_world, rock\_y\_world = pix\_to\_world(xpix\_rock, ypix\_rock, data.xpos[data.count], data.ypos[data.count], data.yaw[data.count], 200, 10)

obs\_x\_world, obs\_y\_world = pix\_to\_world(xpix\_obs, ypix\_obs,

data.xpos[data.count],

data.ypos[data.count], data.yaw[data.count], 200, 10)

The worldmap array was set up so that the red channel would be obstacles, the blue channel terrain and the green channel the rocks. However I found it made more sense to just make the rocks white wherever they were found (lines 76 – 78)

#update the map, make the rocks white on the map.

data.worldmap[navigable\_y\_world, navigable\_x\_world, 2] = 255

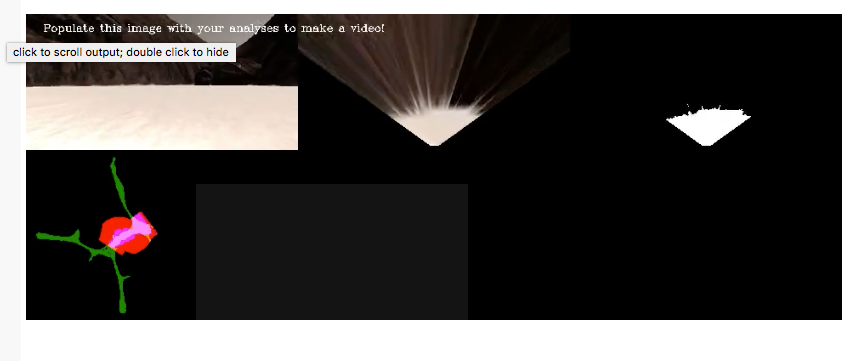
data.worldmap[rock\_y\_world, rock\_x\_world, :] = 255

data.worldmap[obs\_y\_world, obs\_x\_world, 0] = 255

After running the “driver” cell and the video rendering cell, the results for the run using the canned data can be found in output/test\_mapping.mp. The results for the run using the data from the training run can be found in output/training\_run\_mapping.mp4.

The following figures show screenshots of the final frame for both runs.

Example data:



Training run:

