Robotic Nanodegree Writeup.

Notebook Analysis

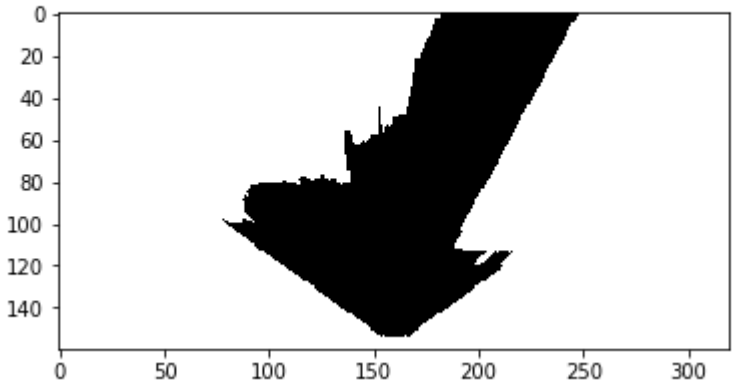
1. Run the functions provided in the notebook on test images (first with the test data provided, next on data you have recorded). Add/modify functions to allow for color selection of obstacles and rock samples.

   Finding the obstacles (i.e. the walls, large stones) is the same process as finding the navigable region. These two function take the same perspective-transformed image. Once the program is successful in one of them, take the inverse would result in the correct output for another task. Since the Rover_Project_Test_Notebook.ipynb has already provided a function color_thresh(), which identifies the navigable region quite successfully. I copied the function an named it inverse(), modify it to produce all the pixels that has value less than 160, where the provided color_thresh() function it produces the pixels that with a value larger than 160.

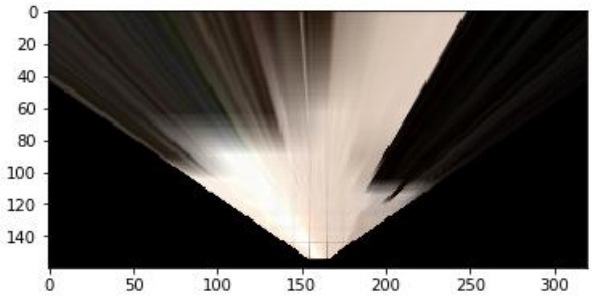
```
def inverse(img, rgb_thresh=(160,160,160)):
    color_select = np.zeros_like(img[:,:,0])
    below_thresh = (img[:,:,0] <= rgb_thresh[0]) \
                & (img[:,:,1] <= rgb_thresh[1]) \
                & (img[:,:,2] <= rgb_thresh[2])
    color_select[below_thresh] = 1
    return color_select
```

   And in the jupyter notebook I test this function.

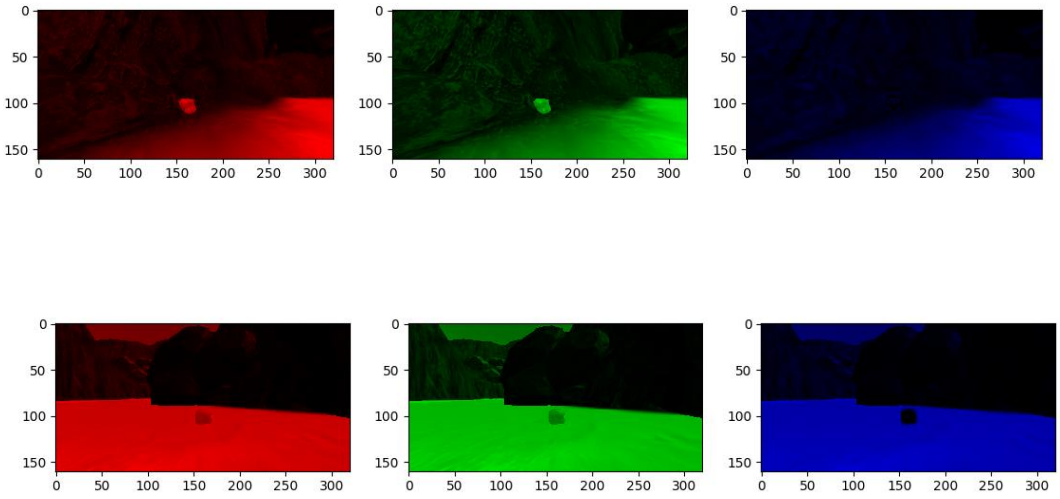
```
Out[9]: <matplotlib.image.AxesImage at 0x206e7a7d128>
```



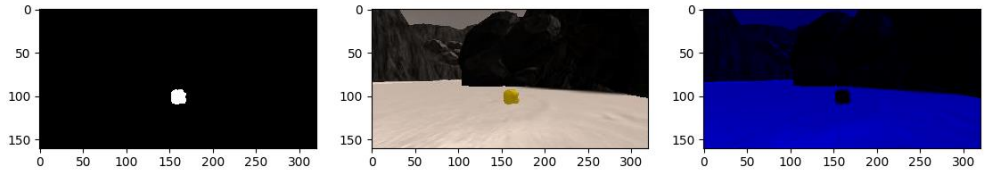
   In the above picture, the white region marks the obstacles. The input image is the following one.

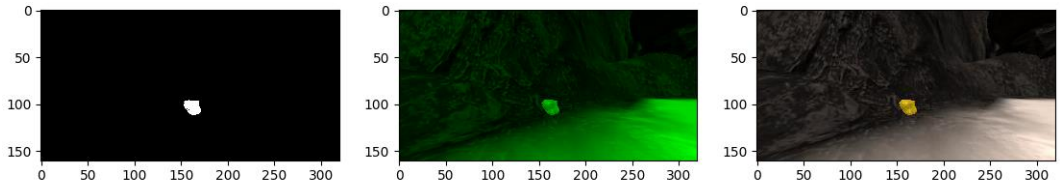


Since the rocks are yellow in color, a high value in both green channel and red channel with a low value in blue channel would result in a yellow like color. To test my hypothesis, I wrote a separate file called test.py, I'll submit this file as well. I used the example image from /calibration_images as input.

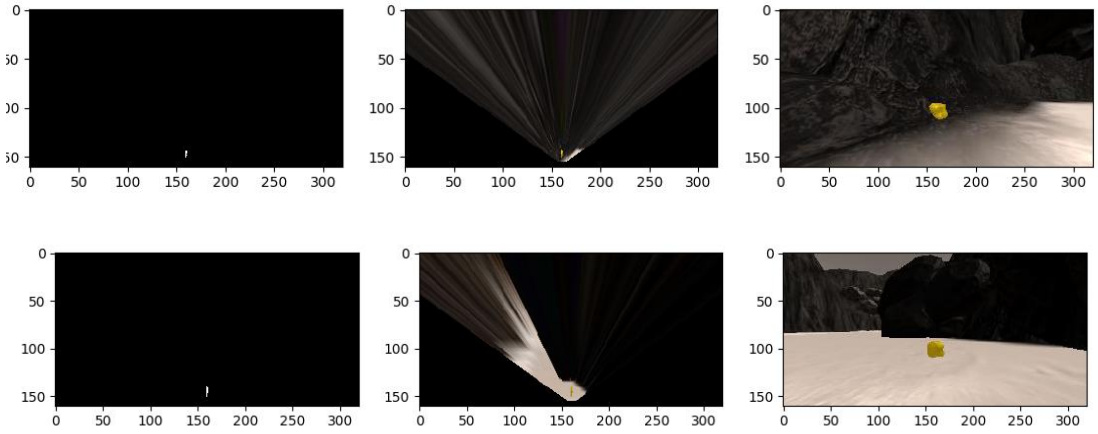


The above is the result of separating each channel of example_rock2.jpg and example_rock1.jpg. It perfectly matches my expectation. I modified the inverse() function to make it produce the pixels has a yellow color. Clearly the parameter value (160,160,160) cannot hold, after a few attempts, I found that selecting the pixels with red and green values higher than 100 and blue value lower than 76 would produce an ideal result.





The above is the result of identifying rocks in example_rock1.jpg and example_rock2.jpg. To test it in the perspective transformed view, I copied the perspective transform function from jupyter notebook. The result is as following.



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

   Since the jupyter notebook has provided a lot of functional parts for process image(), I used the result directly from the previous part such as dst_size, bottom_offset, source and destination remain unchanged. Using the color_thresh function, inverse of color_thresh function and find_rock, the program is able to determine navigable regions, obstacles and rock samples. With the help of provided functions, I convert each of them to rover centric coordinate, then use the rover centric coordinates to get the world coordinates of each category. The world size and scale remain unchanged as 200 and 10. I marked the obstacles as red, navigable region as blue and rocks as green. The video output obtained from the test data is together submitted.

Autonomous Navigation and Mapping

1. Fill in the `perception_step()` (at the bottom of the `perception.py` script) and `decision_step()` (in `decision.py`) functions in the autonomous mapping scripts and an explanation is provided in the writeup of how and why these functions were modified as they were.

1) The perception_step()

I used the functions and parameters from Jupyter Notebooks (i.e. dst_size, bottom_offset, source, destination and so on), the functions also include the two that was used for identifying rocks and obstacles. First I applied a perspective transform to the Rover.img. From this I can get the binary image that will be mapped onto the ground truth map. The navigable region is marked in blue and obstacles are marked in red.

Next, the original Rover.img is fed to find_rock() function to identify any rocks in the camera view. This result is used to determine whether or not there is any rock within the rover's "sight". The reason of identifying rocks with the original image rather than the perspective transformed image is that the original image will have more pixels available especially when the rock is still quite far from the Rover. This would provide proficient time for the rover to adjust its attitude straight towards the rock. If inside this camera view there are more than 2 pixels is considered as a rock, the rover will keep a "True" value in Rover.rock_exist which is a new field. In practice, 2 pixels rarely but still would cause a fake "True" value, but higher value would result in insufficient reaction time gap.

The perspective transformed binary image of obstacles, navigable regions and rocks is timed by 255 (in order to make them distinguishable for human eyes) and sent to different color channels of Rover.vision_image. It is displayed on the top part at the bottom left coner. Then these three images are converted to rover-centric coordinates and from those we obtain the world coordinates. Before the Rover.worldmap gets updated, we check the roll and pitch of the rover to make sure that these results are accurate enough. Here I used the absolute value of the roll and pitch. If all of them are less than 0.9 then we record it. This would result in an overall fidelity above 60%.

Lastly before the Rover object is returned, I calculate the polar coordinate parameters of both navigable regions and rocks. The angle of the rock indicates the direction relative to current position of rover, this would make picking up rocks easy. I also added Rover.rock_go, Rover.rock_stop, Rover.rock_dist, Rover.rock_angles to support picking up rocks.

2) The decision_step:

At the beginning, I declared 3 new variables, valid_angle(the correct direction to go to), valid_go(the correct limit for initiate 'forward' status) and valid_stop(the correct limit for initiate 'stop' status). The actual value of them would be either Rover.nav_angles, Rover.go_forward, Rover.stop_forward or Rover.rock_angles, Rover.rock_go, Rover.rock_stop. The actual value is determined by Rover.rock_exist which is obtained from perception step. If rocks exist in sight, these are set to the rock parameters. I also let the Rover to check if it should pick up any rock prior than any other decision.

If not rock need to be picked up, check if rover is stuck. Stuck happens when the status is 'forward' and there are navigable region or rocks in sight but rover has a ground speed equals to 0. To solve this, the concept is quite simple, as long as the rover is in the 'forward' mode and throttle is larger than 2, when speed is 0, set the mode to 'stuck', set throttle to be negative (to make it move backward), and update the Rover object. In stuck mode, it turns the rover left or right at random and go back for a while by adding the throttle a little bit until throttle equals to 0. Although this part of code does not work as I expected, it does free the rover from a stuck.

The other parts of decision_steop() remain unchanged, rover goes to the direction where there is rock samples or there is a clear and broad path

2. Launching in autonomous mode your rover can navigate and map autonomously. Explain your results and how you might improve them in your writeup.

**Note: running the simulator with different choices of resolution and graphics quality may produce different results, particularly on different machines! Make a note of your simulator settings (resolution and graphics quality set on launch).

Here I run the simulator with screen resolution as 1280*720 and graphic quality set to fantastic, it runs in a window. The rover produces a map with fidelity higher than 60% and often maps more than 40% of the map within 300 seconds.

There is a few problems with the current rover, which can be improved in the future.

1) It is easy for the rover to be trapped in the top right part of the map, where there is a lot of space with two narrow paths. From the camera view we can see that it is difficult to distinguish the entrance of those two paths even for human eyes. Always drive itself to the broadest direction might trap the rover easily when the rover is near the edge of a large "playground". To solve this problem, I think a "wall crawler" strategy might be effective. As long as the map has some sort of closed shape, the wall crawler can traverse the edge of the map without going back or being trapped.
2) The "retreat" mechanic for solving stuck problem sometimes does not work appropriately. There are situations where the rover cannot move forward or backward but it can turn around. Perhaps making the rover turn prior than move backward would be a better solution.
3) The rover sometimes goes back to the place where it came, regardless whether the region in front of it is visited after picking up a rock sample. This is the result from the Rover.mode filed after picking up a rock sample. It is a "stop" mode, and the rock samples are near the walls. When the rover is in "stop" mode and it indeed is at full stop where there is no navigable region in front, it turns right by default. To improve this, we could make the rover remember discovered region. When a full stop occurs with no navigable region in front, the rover turns and check if the region in the camera view is visited, if not, move forward. If it is visited, turn to another direction and move on.