Arnold Kompaniyets

Robo ND (Term 1)
March 23, 2018

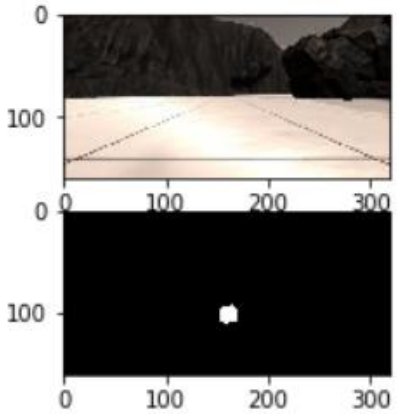
**Search and Sample Return**
**Write-Up**

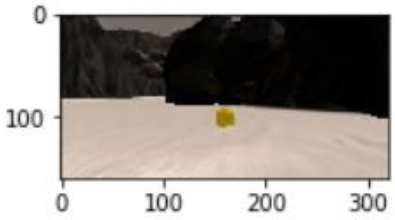
**Jupyter Test Notebook Analysis:**

This Jupyter notebook is organized in a way that allows for analysis by simply moving down each dedicated block of code, so it will be taken as such:

1) The first few blocks of code, highlighting specific text, as well as importing important functions, were left unmodified.
2) Calibration data: this block was modified slightly, using the 'gold_thresh()' (described below) in order to see which RGB values would work best to identify the gold rocks. The function was used on the test image, and a third, binary image was output, with ideally only the location of the gold rock producing the white space.

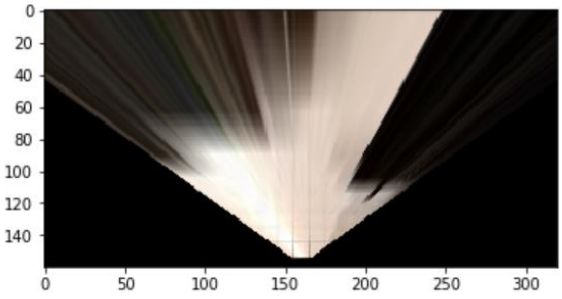
```
<matplotlib.image.AxesImage at 0xbc3d588>
```



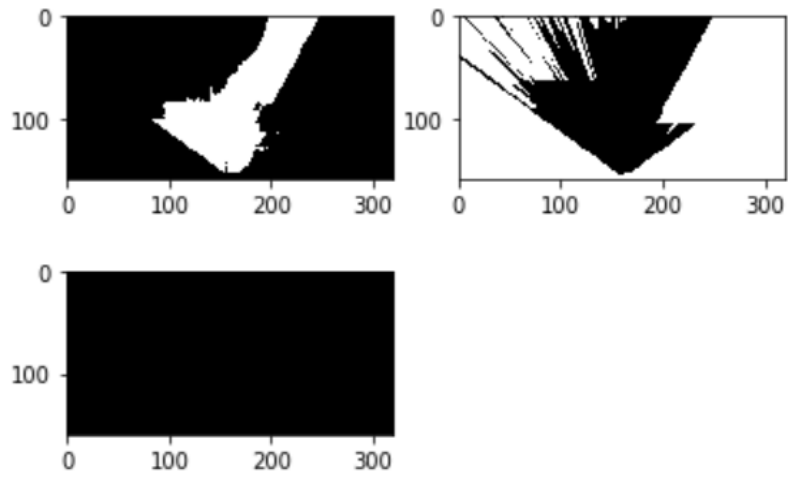

3) Perspective transform: this block was left unmodified.

```
<matplotlib.image.AxesImage at 0xa091978>
```



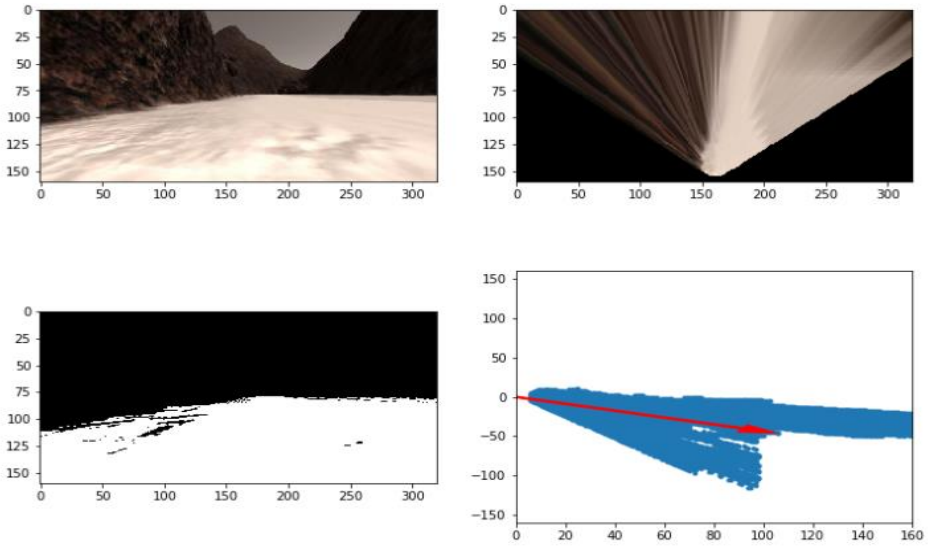
4) Color thresholding: the 'color_thresh()' function parameters were adjusted to 195, 175, 165 to better discriminate the navigable terrain. Likewise, a 'below_thresh()' function was created (nearly identical to 'color_thresh', but with the image values qualifying if below the threshold values), with said threshold values set to 50, 40, 40. The separation in RGB thresholds served to not include the sky in either function. A third function, 'gold_thresh()' was also created, taking only the image as its argument. Relatively identical to the two functions above, this one set specific RGB boundaries for the gold rocks (identified using 'imagecolorpicker.com').

```
<matplotlib.image.AxesImage at 0xbf5a518>
```

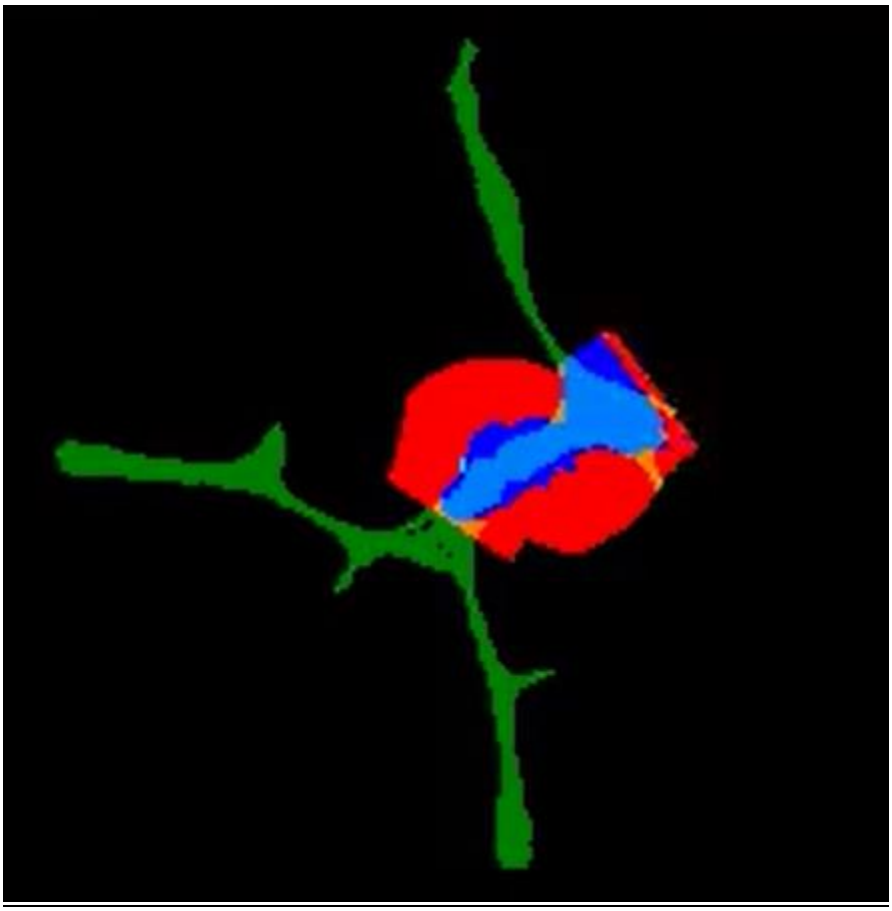



5) Coordinate transformations: these functions were left as is, with only a few minor additions made to the 'to_polar_coords' functions, as a way to identify distances only at specific angles, the functionality of which will be described later.

```
<matplotlib.patches.FancyArrow at 0x9e3cbe0>
```



6) Databucket class: this block of code was also left unchanged.
7) Process_image() function: the source and destination points were copied directly from the code above, as was the 'dst_size' and 'bottom_offset'. The forward-facing camera image from the rover was first transformed using the 'perspect_transform()' function. Then, this image was entered into all three color threshold functions to produce binary images containing the navigable terrain, obstacles, and gold rocks. Following, each of the binary images was transformed into rover coordinates and then to world coordinates. Lastly, the third channel of the worldmap was updated with the navigable terrain world coordinates, the first channel with the obstacles, and all channels with the gold rock world coordinates. This step was modified by also setting the first channel values to zero if those same coordinates are also represented in the navigable terrain. The rest of the code was left unchanged.
8) Output image movie: lastly, the 'process_image()' function was used on all the images in the test folder, with an output movie being produced with moviepy. This same function was also used on data recorded in training mode of the rover simulator.



Blue – navigable terrain, Red – obstacles, Small white rectangle – gold rock

**Autonomous Navigation and Mapping:**

1) Perception.py – First, the 'color_thresh()', 'below_thresh()', and 'gold_thresh()' functions were imported from the jupyter notebook; however, the RGB threshold values were modified to suit the autonomous simulation best (170, 170, 165 for the navigable terrain; 65, 50, 50 for the obstacles; 110-255, 110-255, 0-60 for the gold rocks).

The 'rover_coords()', 'to_polar_coords()', 'rotate_pix()', 'translate_pix()', 'pix_to_world()', and 'perspect_transform()' functions were left unchanged. In the 'perception_step()', first the Rover.img was changed to simply 'img'. As in the Jupyter notebook, the 'dst_size', 'bottom_offset', 'source' and 'destination' were directly imported. Then, the front-facing camera image was transformed using 'perspect_transform()'. Next, this image was placed into each color threshold function to produce a binary image of navigable terrain, obstacles, and gold rocks.

Additionally, the color threshold functions were used on the front-facing camera image to produce binary versions of the untransformed images. These three images were then used to update each channel of the Rover.vision_image variable, each value multiplied by 255 to produce the most contrast of color. This vision image copies the front-facing camera, but allows the user to visualize the navigable terrain from this perspective.

The transformed binary images were then placed into rover coordinates, followed by world coordinates. In updating the Rover.worldmap, a condition was set up, in that the worldmap would be updated only when Rover.roll and Rover.pitch were very close to zero. Otherwise, this update step remained the same as in the Jupyter notebook. An attempt was made to have the obstacle map first replace the previously identified navigable terrain with each image update (in order to perhaps eliminate terrain identified as navigable outside the actual truth map), but this proved to not be any more effective than the method before. Test code was also made to not include any terrain identified as navigable outside of the ground truth map, but of course, this artificially maintains fidelity at 100% in any and all conditions.

Continuing, if any values were entered for the map of gold rocks, 'to_polar_coords()' was used to attain the lists of distances and angles. After, an index of the minimum distance value was assigned to a variable, with the x and y positions of said distance value assigned directly after. At this point, the worldmap was updated with the x and y positions of the gold rock minimum distance values. Lastly, the distances and angles were updated to Rover.gld_dists and Rover.gld_angles (if none were present, the variables were set to None).

In addition, a variable 'x_angles' was created, being a list of Boolean values

(identical in length to the 'nav_angles' list), only containing True values where the angles are equal to zero. Another list, 'zero_lists' was then created, attaining the distances only where the angles are equal to zero, i.e. the navigable distances directly ahead of the rover. This same methodology was done for a list called 'far_dists', composed of only the distances where the angles are greater than -15 (i.e. far right of the rover).

To finish the file, if any values existed for 'nav_terrain', 'zero_dists', or 'far_dists', they were set to the newly defined Rover class variables in 'drive_rover.py'. If any of them were without values, the variables were set to 1. Since the length of some variables is referenced in 'decision.py', these variables were set to a list containing one value.

2) Decision.py – In this folder, I started out by importing the 'time' and 'random' functions, in case I required them (I ended up not using either in my current code, but did experiment with them and could perhaps use these functions in future updates to my code).

I first attempted to use the pre-populated code and modify it accordingly, but I ended up choosing to simply start fresh and type up the code myself. My initial condition took the scenario that the rover mode is 'forward', and there is some data present for gold rocks. In this condition, if the average distance to the gold rock is less than 13, the robot is stopped and, once the velocity is zero, Rover.send_pickup is set to True. Otherwise, the rover is directed towards the gold rock by using the average of the gold rock angle list, up to a maximum velocity of 0.6.

* One source of trouble that I did identify in this code block is the scenario in which the rover catches on a rock or wall, potentially making the average of 'gld_dists' less than 13 but not close enough to the sample to switch Rover.near_sample to True. This code would need to be modified in the future to allow for greater obstacle collision while moving toward the gold rocks.

The rest of the code focuses on Rover.nav_angles being populated with something. The first code block addresses the rover being in forward mode. In the code submitted, I only set up two scenarios. If the average of navigable distances is less than 20, if the total number of navigable angle points is less than 200, the maximum of the navigable distances in front of the rover is less than 20 (should a rock appear in from of the rover), or the maximum of navigable distances beyond -15 degrees is less than 10 (should the rover get too close to a wall on the right), the throttle is turned to zero and the rover mode is set to 'stop'. Otherwise, the steering angle of the rover is set to the average of the navigable angles, up to the

set maximum velocity (1 in this case). After various testing, it became clear that the rover could traverse the environment more thoroughly by hugging one of the walls. As such, the steering values of left turns (angle averages above zero) were weighted to only 0.5 of its value, and 6.5 was subtracted from the resulting number. 5 was subtracted from the right turns as well. This was done to generally steer the robot towards the right.

The next code block addresses the rover being in 'stop' mode. The code first makes the rover come to a stop. After this, the code states that until specific conditions are met (identical to the four presented above that place the rover into 'stop' mode, although this time with larger minimum values to exit the conditional loop), the rover will set its steering angle to 15 (meaning it turns left, which is ideal for a rover that intentionally trends towards the right). Once the conditions are satisfied, meaning the rover has a clear path identified, its mode is switched back to 'forward'.

Another attempt was made to create another rover mode called 'obstacle', which took the case of the rover being unknowingly stuck on a rock (meaning its throttle is maxed out but velocity is unchanging). It doesn't appear that this code worked as intended, so further work would need to be done on it (perhaps implementing a predetermined time delay before determining that a rover is indeed "stuck").

The last block of code addresses the possible scenario that none of the conditions above are met, in which case the rover is programmed to stop and switch its mode to 'stop'.

3) Autonomous mode – successfully having the rover navigate its environment took a few stages of development.

- Technical point: Screen resolution – 1024 x 768
  Graphics quality – Good
  Average FPS – 30

First, my goal was to simply have the rover be able to traverse the navigable terrain without stopping at dead ends. With this, I turned off the gold rock searching parameters and only focused on the angles for navigable terrain. Afterwards, I set the first series of conditions to enter 'stop' mode (the limit on 'nav_dists' and 'nav_angles'). The steering angles were yet unweighted, so the rover explored its terrain purely based on the average of navigable angles.

Once I felt that the rover was capable enough to traverse over 60% of its terrain, I replaced the gold rock parameters. After a few test runs, I determined that 13 was an appropriate average length minimum to set the 'Rover.send_pickup' parameter to True.

At this point, I began to notice more and more difficulties with the rover running into large rocks in the terrain. This was when I decided to add the 'zero_dists' parameter, specifically using the maximum value of this list. My logic behind this was that if a large rock (beginning just past the maximum navigable distance) was directly in front of a rover, the mode would be switched to stop, even if there were plenty of navigable angles to the left and right of the big rock.

With the aforementioned perimeters in place, the rover had a much easier time navigating its terrain. After lowering the maximum velocity from 2 to 1, this task was made even easier. Unfortunately, I began to notice at this point in time that the rover's final mapping percentage was heavily tied into its starting position. More specifically, the rover tended to get stuck in specific pathways, because the average of navigable angles would always lead it down the same path. At this point, I knew I needed to attain a better method of traversing the map. Since the environment is completely enclosed, it made great sense to have the rover stick close to one of the walls and map the environment that way.

After quite a few failed attempts, I decided to just decrease the weight of any left-turning values to 0.5 and sway all directional values to the right by a specified amount. Since the rover was already programmed to turn left, this allowed the rover to sway to the right and turn left slightly if it got too close to a wall. This naturally led to the issue of the rover becoming stuck on the rock wall , which led me to placing 'far_dists' in the 'stop' mode parameter as well. This allowed the rover to sense if it was too close to an obstacle on the right side and turn slightly left accordingly.

In regards to mapping, I found out readily that fidelity decreased quickly with only minimal mapping. After reading further, I saw that offset pitch and roll can quickly deteriorate mapping functionality, so I adjusted the 'perception.py' file to only map when pitch and roll were very close to zero. This allowed the rover to maintain its mapping fidelity above 80% even after exploring most of the total map.

**Overall thoughts for improvement:**

Certainly, there is much room for improvement with my code, and I may definitely come back to make various changes and additions. Firstly, it appears that the map may be better suited to exploration when sticking to the left wall, so making the necessary adjustments for that could improve the rover's functionality with that alone. Second, the rover currently heads straight towards a gold rock when it is spotted, and while this is normally fine, there are instances where the rover can collide with a wall or large rock on the way; therefore, it will be a good idea to add the functions for obstacle avoidance when heading towards the gold rocks as well. Lastly, another condition can be added to the rover,

stating that once a certain percentage of the map has been covered, and all of the rocks have been collected, the rover can set its general trajectory to the very center of the map (where it started). To make mapping faster, it would be beneficial to have the rover place greater weight on heading towards unexplored areas of the map, although I haven't quite figured out how I could add that weight.