# Project: Search and Sample Return

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.*

You're reading it :)

## 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.*

First I tested the color_thresh() function, as provided during the lesson. Its job is to find navigable pixels and its output can be seen in Figure 1. The white pixels forward and slightly to the right suggest a direction the rover could potentially head towards next.

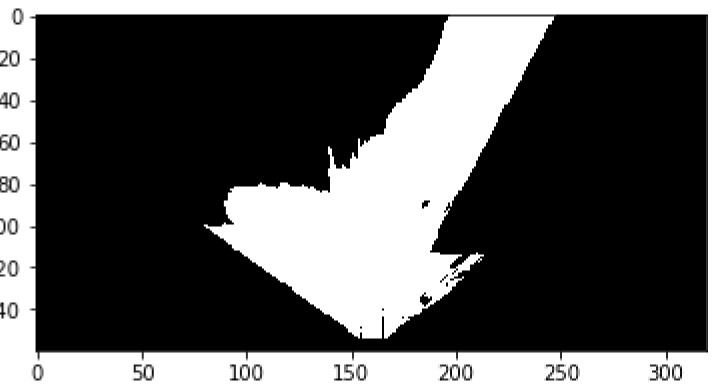

Figure 1 - The output of the color_thresh() function that shows navigable pixels in white

Next I added the obstacle_threshold() function to colour select places where the rover cannot move. In this world I believe all pixels could be classified as either navigable or obstacles, and so initially I set the obstacles function to return the inverse of the output of the color_thresh()

function (that finds the navigable pixels) using the np.invert() function. This worked well on the test images, an example of which can be seen in Figure 2. However I ran into some problems with this approach later in the project when trying to display the obstacle image on the left-hand side window of the autonomous driving rover application - it just turned that window all red. I think the issue is due to different colour mappings, but I couldn't find a way to resolve it. So instead I changed the obstacle_thresh() function to not use np.invert(), but instead function in exactly the same way as color_thresh() but returning places below the RGB colour threshold, not above. This worked successfully, so is the version of the function that can be found in *perception.py* (the Jupyter notebook still has the original version).

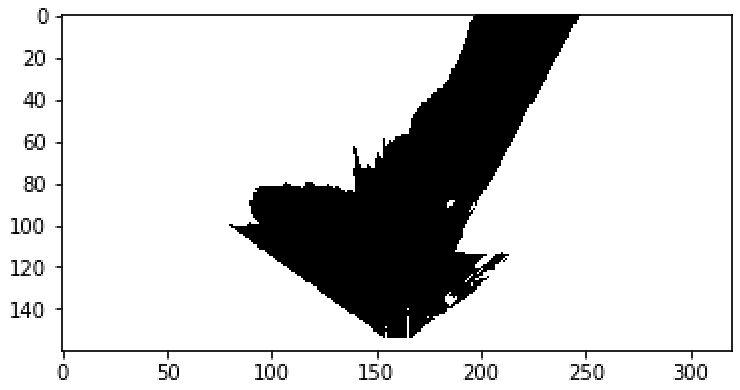

Figure 2 - The output of the obstacle_thresh() function that shows obstacles in white

Finally I made the rock_thresh() function to identify the rocks that the rover needs to collect. It detects the yellow colour of the rocks with the OpenCV cv2.inRange() function that selects only colours between a low and hi RGB threshold. I started of setting the thresholds based on the khaki and darkkhaki colours from this colour chart [(http://www.rapidtables.com/web/color/Yellow_Color.htm).](http://www.rapidtables.com/web/color/Yellow_Color.htm) I then adjusted the colours until I found the best values (in particular I found the blue component needed to be much lower). An example input and output of the function can be seen in Figure 3 - you can clearly see it does a good job identifying the yellow rock where it shows the white pixels in the output.

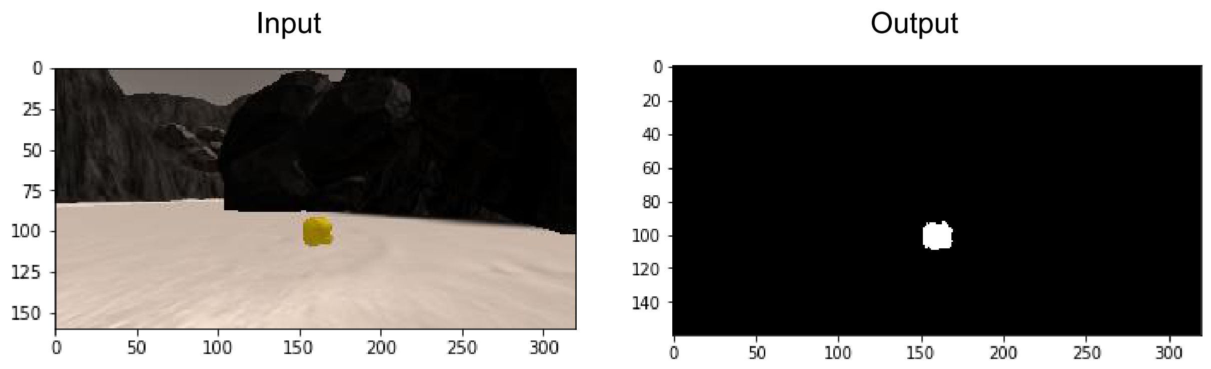

Figure 3 - The left hand side shows an image from the rover containing a yellow rock. The right hand side shows the output of the rock_thresh() function that has correctly identified the rock location with the white pixels.

*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.*

Firstly the original image was added to the top-left corner of the output mosaic as per the original notebook. Similarly the warped image was added to the top-right corner and the world map to the bottom-left corner. My main addition was to call the color_thresh() function to identify the navigable pixels and add this to the bottom-right corner of the output mosaic. This is the final 4-picture frame that I used to create the video with moviepy. One important learning step for me here was that I had to convert the black and white, single-channel output from color_thresh() into a full 3-channel 'colour' image for it to be understood properly by moviepy. I achieved this with the following 2 lines of code. It first converts the 0 or 1 output from color_thresh into 0 or 255, then stacks 3 copies on top of each other so it fills all RGB channels:

```
navigable = color_thresh(warped) * np.array(255)
navigable_color = np.dstack((navigable,navigable,navigable))
```

I also experimented with adding the output of rock_threshold() into the mosaic, but decided to use the color_thresh() for the final video.

## 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.*

## perception_step()

There is not too much to comment on regards to the perception_step() function as it is largely a replication of the functions I developed in the Jupyter Notebook which I described in the first section of the report. There are a few additions, however.

Firstly, in the section that maps the navigable terrain, obstacles and rocks (lines 162-169) I added an if statement to prevent updating if either the rover's roll or pitch is above a certain threshold. The camera projection is only technically valid if raw and pitch are exactly zero, so if these values are too high it will cause errors in our mapping and reduce the fidelity. In an ideal world we could only map if roll = pitch = 0, however in reality this would result in us hardly ever updating the map at all as the roll and pitch do increase as the robot moves around, especially when doing larger accelerations and turns. To start off with a set a limit of 1 degree for both, but I found the map created had too many errors and the fidelity was below the threshold of 60%. So I decreased the threshold to 0.1 for each. This went the other way and led to the map not being updated enough (although the fidelity was great!). In the end I settled on a threshold of 0.4, which seemed to give a good balance between efficiently mapping the environment whilst not introducing too many errors.

Secondly, in addition to converting the rover-centric navigable pixels to polar coordinates, I also converted the pixels containing rocks to polar coordinates to (lines 176-182). This will allow the rover to navigate towards the rocks when it finds them as part of the decision step, which I will describe in the next section.

## decision_step()

The main 'brain' of the Rover lives inside the decision_step() function.

First, the Rover checks if it is the first time the function has been called, using the Rover.count variable that I added. If it is it saves the current location to the Rover.start_pos variable that I also added. This will be used later at the very end of the robot's mission so it knows where it has to return home.

Second, the Rover checks if all the samples have been found. If so it knows that it is time for it to return home. It then checks if it is now at the home location - if so then the mission is complete, congratulations! If not then the rover should keep on moving, using the code sections that follow until it does reach the goal. Note that in the current implementation of the code the robot does not intelligently make its way towards the goal, it just keeps driving around and hopefully (by chance) will make it back at some time, at which point the mission would complete. If I can devise a good way to direct the robot specifically to the home location it could be added here, but this is a work in progress. You can see some commented out code which is my best attempt to get it work, but it’s not reliable so I leave it out for now.

Thirdly, the Rover checks if it is currently near a rock. If it is then it issues a command to pick it up. It sounds simple but I had to add a little hack to get it working. I found that the Rover.near_sample flag seems to stay asserted for a very short time even after the sample has been picked up and the Rover.pick_up flag has been deasserted. Due to this (bug?) if I am not careful the robot can get stuck in an infinite loop at this point since having the Rover.near_sample flag asserted then causes the Rover.pick_up flag to become asserted again. To solve this problem I added a timeout feature in line 56 which prevents the sample pickup code executing until at least 200 frames (~8 seconds) have passed since the pick_up flag was deasserted. It’s not ideal, but it works well for this scenario, but probably it would be better to fix the issue for future classes.

Next, the Rover checks if it can see a rock (based on the polar coordinates as calculated in the perception_step() function). If it can then it drives slowly, directly towards it. I set the throttle value lower than in normal driving mode (0.1) to help prevent the robot overshooting the goal. If the robot couldn’t stop in time when it gets near the rock and it loses sight of it in the camera it would not be trivial to get it back into frame and the robot may end up doing something else instead, so I would prefer to avoid this situation. An enhancement to this would be to use a proper PID controller to guide the robot more efficiently and gracefully to the target, but this implementation is simple and works well for the purposes of this project.

Finally, if the robot is not in any of the special states previously checked for, it runs the generic exploration code originally provided whereby the robot steers towards the areas with greatest space and freedom to explore. This is almost unchanged for original, but in addition to the ‘forward’ and ‘stop’ states, I added a third, ‘random’ state. One issue I found with the original code was that sometimes the rover gets stuck driving head first into a wall, can’t go anywhere, and can’t get away. Also sometimes the robot get stuck driving in big loops around familiar territory without exploring new areas of the map. The random state can help with both these situations. Whenever the robot is in the ‘forward’ mode, there is a certain chance (currently 1%) that it will get transitioned to the ‘random’ state. In the ‘random’ state there is a certain chance (currently 5%) that it will return to the ‘forward’ state, else it will stay in the ‘random’ state, whereby the vehicle will turn around via 4-wheel turning. This successfully helps it get unstuck when it is repeatedly bashing its head into the wall, and also works well to allow it to explore new places. Of course it can sometimes cause unwanted actions, like suddenly turning the rover around when it’s in the middle of exploring a new, unmapped area, but it’s a small price to pay for the benefits of never getting completely stuck or never visiting new areas.

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

*Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.*

I am running the RoverSim on MacOS X. I used 800x600 resolution with graphics quality set to ‘good’. I took a screen capture video of my rover in action, you can check it out here https://youtu.be/UBKTHy4cSkU (please note the video runs in 4x real time). This particular run ran for about 13 minutes, in which time it mapped 46.5% of the environment at 61.1% fidelity. It also managed to pick up 3 rocks. (NB: despite this the lower-right hand panel just says that 2 rocks were found, which is strange. I posted this issue onto Slack and the Udacity team are looking into it - possibly a bug, or too tight a constraint on the detection distance on line 90 of the supporting_functions.py sheet). I aborted the run here since the requirements to pass the project were already met (40% map, 60% fidelity, 1 rock) and I didn’t want the video file to become unmanageably large. However there was nothing wrong with the robot’s performance and it wasn’t stuck, etc, so I am confident I the current program could eventually map the whole environment, collect all the rocks and return home successfully, given enough (probably large) amount of time.

The main weakness of the current program is it doesn’t take into account where the rover has already been to prevent revisiting completed areas nor actively look for new regions of the mpa to explore. Thus whilst given enough time it *should* eventually find all areas, there is no guarantee of this, and it may take a while. Similarly, if it does find all rocks then the robot does not explicitly direct itself towards the goal in the current implementation. It continues to drive around as before and, if by chance it reaches the start point it will brake and complete the mission. This is definitely something I would like to improve further and I intend to continue working on it!

Other things that could be improved is the balance between the amount of mapping data that is logged and the fidelity. As discussed in the perception_step() section I added limits on roll and pitch values that are acceptable if the pixels captured by the camera can count as mapping values. I would prefer to set the limit lower than the final value (0.3) to allow me to increase the fidelity further, but it caused the amount of area that I was mapping to fall too slowly. If I can improve the amount of map area mapped using the more intelligence exploration algorithms, as hypothesised in the previous paragraph, it should also be possible to reduce the roll and pitch limit values further to further improve the fidelity.

In summary the rover I built passed the requirements of the project and given enough time could realistically complete the whole sample and return challenge too. I am proud of the work I did as I had to learn a lot of new skills along the way to achieve this. I also presented some potential areas of improvement and I plan to start executing on these before further challenges are released. Thanks for a fun first project!