ROVER PROJECT

This document describes the sensing and actuation modules that enable the rover to map the simulated world and identify the location of rocks. The sensing module describes how navigable terrain, rocks, and obstacles are identified in an image. The actuation module describes how the rover decides to navigate the environment using its internal map of the world as well as the its image sensors.

SIMULATOR SETTINGS

The following results were obtained with the following simulator settings:

- The FPS on the terminal ranged from 23-30 FPS and was mostly at 25 FPS and has dipped to as low as 10.
- The screen resolution was 1024 X 768 and the graphics quality set to "Good".

PERCEPTION: SENSING THE WORLD

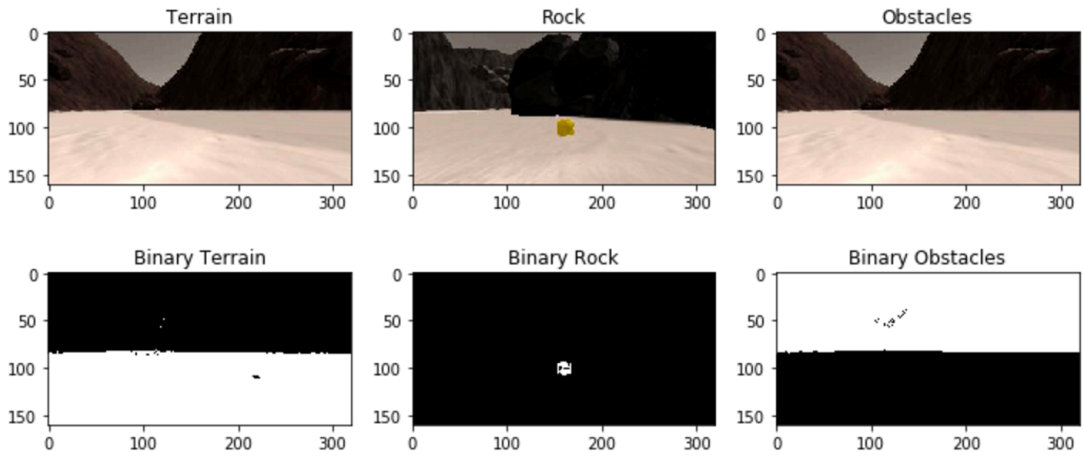
The rover uses color to determine whether pixels in the input image correspond to navigable terrain, golden rocks, or obstacles. This is possible since these objects occupy different RGB color ranges in the simulated environment. The following inequalities were used to determine whether a pixel corresponds to terrain, rock, or obstacle.

$$\textbf{Terrain:}\quad 160 < R < 255 \quad 160 < G < 255 \quad 160 < B < 255$$

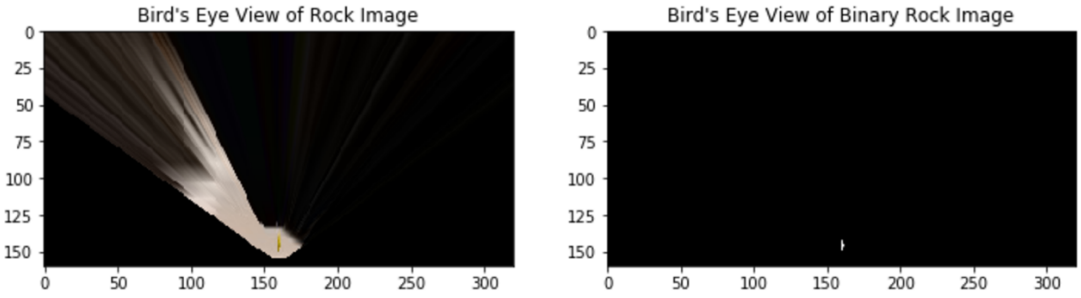
$$\textbf{Rocks:}\quad 50 < R < 255 \quad 50 < G < 255 \quad 0 < B < 10$$

$$\textbf{Obstacles:}\quad 0 < R < 160 \quad 0 < G < 160 \quad 0 < B < 160$$

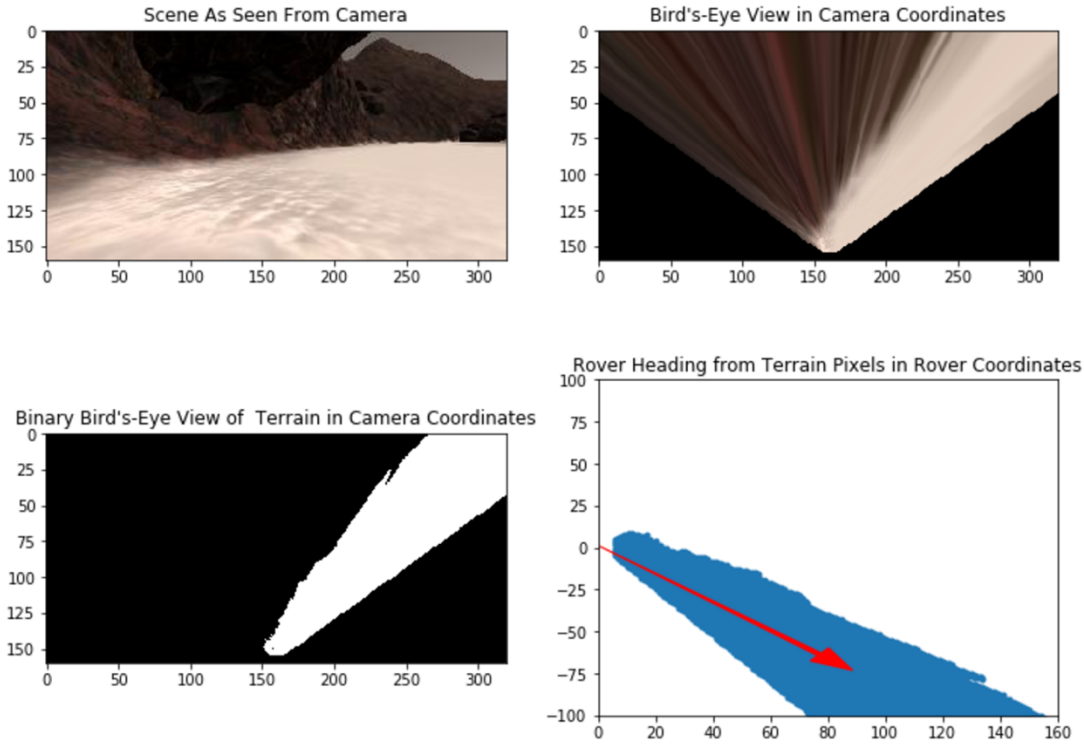
The images below illustrate how the inequalities above enable us to extract terrain, rocks, and obstacles. For example, when the pixels in the image titled "Rock" are tested using the Rock inequality above we get back the binary image titled "Binary Rock" where only pixels belonging to the rock are in white. This is accomplished by the method *color_thresh* in *perception.py*.



The code performs the image thresholding operations above in the bird's-eye view of the scene. This is valid since obtaining the bird's eye-view (using the *perspect_transform* method in *perception.py*) is a one-to-one. As an example, the image in the left panel below shows a bird's-eye view of a scene with a golden rock (the scene corresponds to the image title "Rock" above). After applying the color threshold corresponding to rocks to this bird's-eye view we are left with the image in the right panel below where only pixels corresponding to the rock are in white and are seen form a bird's eye view.



The bird's eye-view becomes even more useful once mapped from camera coordinates to rover coordinates using the method *rover_coords* in *perception.py*. In rover coordinates the heading required to reach a target can be determined as shown below. The image in the top left panel corresponds to the scene as seen by the camera while the one in the top right corresponds to the bird's eye-view. The image in the bottom left panel is the binary image corresponding to applying the *Terrain* RGB thresholds mentioned above. Lastly, the image in the bottom right corresponds to terrain pixels in the rover's coordinate frame. Transforming the terrain pixels (in rover coordinates) to polar form (using the method *to_polar_coords* in *perception.py*) finally allows to determine the mean angle of the pixels with respect to the rover and thus heading/steering angle needed to reach them.



The last step in the perception process is to annotate the world map with findings. This is accomplished by transforming pixels of interest from the rover's coordinate frame to the world's coordinate frame using the method *pix_to_world* in *perception.py*. As an example, the image below shows the rover observing a golden rock. Using the Rock RGB thresholds the rover extracts rock pixels in its coordinate frame. Next the location of these pixels in the world coordinate frame is determined using *pix_to_world* and the rover's position and yaw angle in the world frame. Finally, the rock pixels in the world coordinate frame are colored as shown in the map in the bottom right inset of the image below (small white square corresponds to the rock).

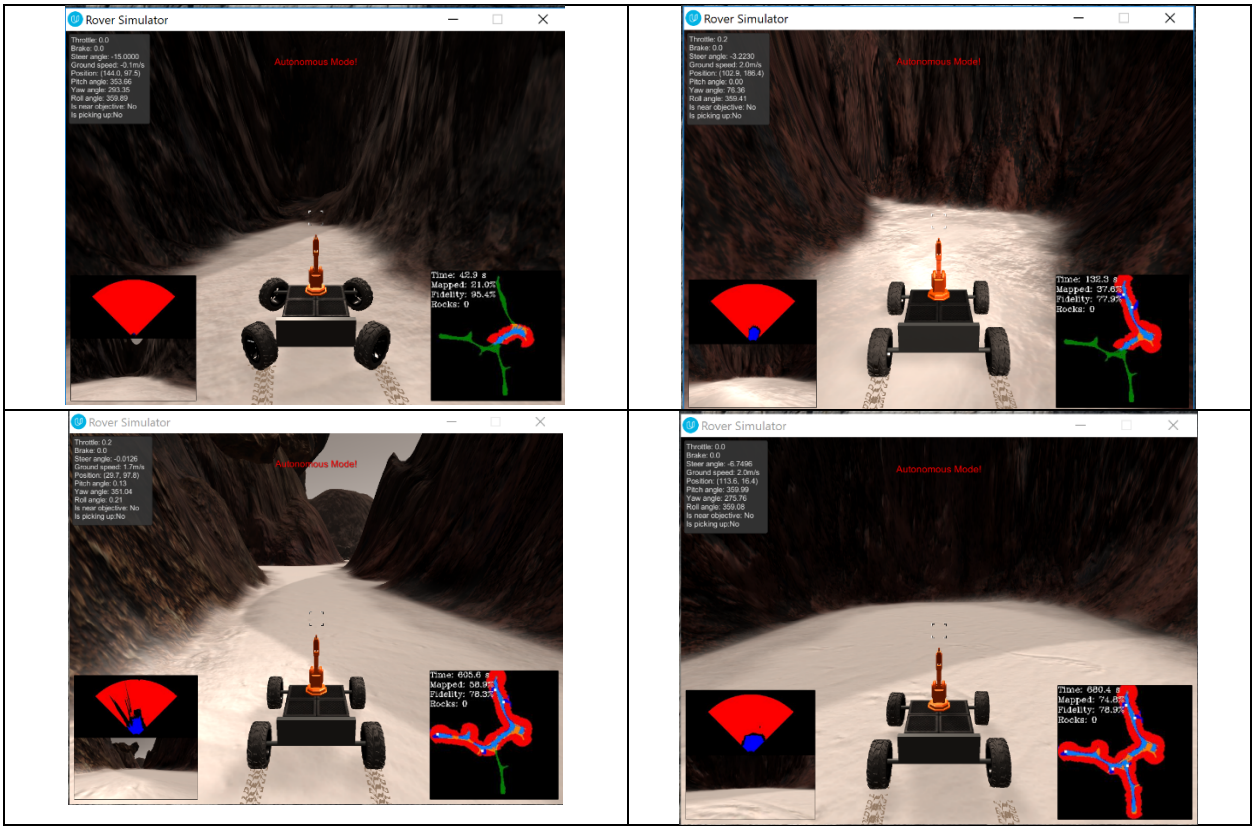
ACTUATION: DETERMINING WHERE TO GO

To map the environment the rover needs to have a policy that prescribes where it should go next. In this project, I steered the robot using a policy that favors steering angles that lead the rover to areas that have never been visited (or have been visited the least number of times).

To implement this policy, the rover maintains a duplicate map of the world (called *self.worldmap_visited* in *drive_rover.py*). This duplicate map is the same size as the world map (200 x 200) and maintains a count of the number of times the rover has visited each location of the world. When deciding where to go, the rover examines all steering angles within 45 degrees of the direction of maximum navigable terrain. The rover then selects the steering angle resulting in the rover traveling towards a part of the world that has never been visited (or has been visited the least number of times) according to the current duplicate world map. This logic is implemented in *lines 26-53* of *decision.py*.

This steering logic prevents the rover from "going in circles indefinitely" and allows the rover to map most (if not all) of the simulated world*. Note, the rover may visit the same location repeatedly but eventually turns away from that location because the visit counts increase relative to other areas on the map*.

The images below show this policy in action on a *1024x768 screen resolution with graphics quality "Good"*. Each image shows the rover having explored a larger fraction of the simulated world with identified rocks appearing as small white squares. Going from left to right and top to bottom one sees the rover having mapped approximately 21%, 37%, 58%, and 78% of the world map and have identified the location of 5 rocks.


Rover Simulator
Autonomous Mode!
Rover Simulator
Autonomous Mode!
Rover Simulator
Autonomous Mode!
Rover Simulator
Autonomous Mode!