Liam Page
Udacity Robotics
11 October 2017

Project 1 - Rover

Notebook Analysis

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.

Much of the work performed on the *process_image* function is based on the instructions and previous exercises. Defining the source and destination points for the perspective transform are based on the values and function calls from the project exercises. These points are then passed to the *perspect_transform* function, which maps the robot camera image to a top-down, 2D plane.

The warped image is then passed to the color thresholding functions. For the color thresholding, I modified *color_thresh* to filter on values greater than *or equal to* the provided tuple. In addition, I added an optional argument, *min_thresh*, which searched for values below the threshold instead of above the threshold. I also created a wrapper function, *color_range*, which could filter on values between two provided tuples, inclusive. The *color_range* function was instrumental in the identification of the yellow rock formations. For both the navigable terrain and rock formations, I loaded still images in the GIMP image editor, and manually recorded what appeared to be the darkest and lightest colors so provide more precise ranges of color thresholding. Threshold values used are as follows:

Target	Min RGB	Max RGB
Obstacles	(1, 1, 1)	(189, 169, 159)
Rocks	(140, 115, 0)	(255, 255, 90)
Navigable	(190, 170, 160)	None

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.

For creating the world map, again much of the work was performed in previous course exercises. Each world target point (i.e. obstacles, rocks, and navigable terrain), are converted to rover coordinates, where the rover is at the left-middle point of the world map. These points are then translated to world map coordinates using the robots known x and y position in the world. The world map is then updated with the translated points applied to the separate RGB color channels, but only if the rover's pitch and roll are within +/- 1 degree of normal. Since the perspective warp and translation functions assume we are a 0 degrees pitch and roll, this check prevents the application of inaccurate data to the world map if the robot hits an obstacle or strong incline.

For a video of the test output, see RoboND-Rover-Project\output\test_mapping.mp4.

As an aside, I found it more helpful to edit the actual perspective.py file directly and test the results in the simulation's autonomous mode, though I see the benefit of being able to apply new configuration options to a constant data set. That I could take control of the rover in autonomous mode and focus on problem areas was a big help.

Autonomous Navigation and Mapping

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.*

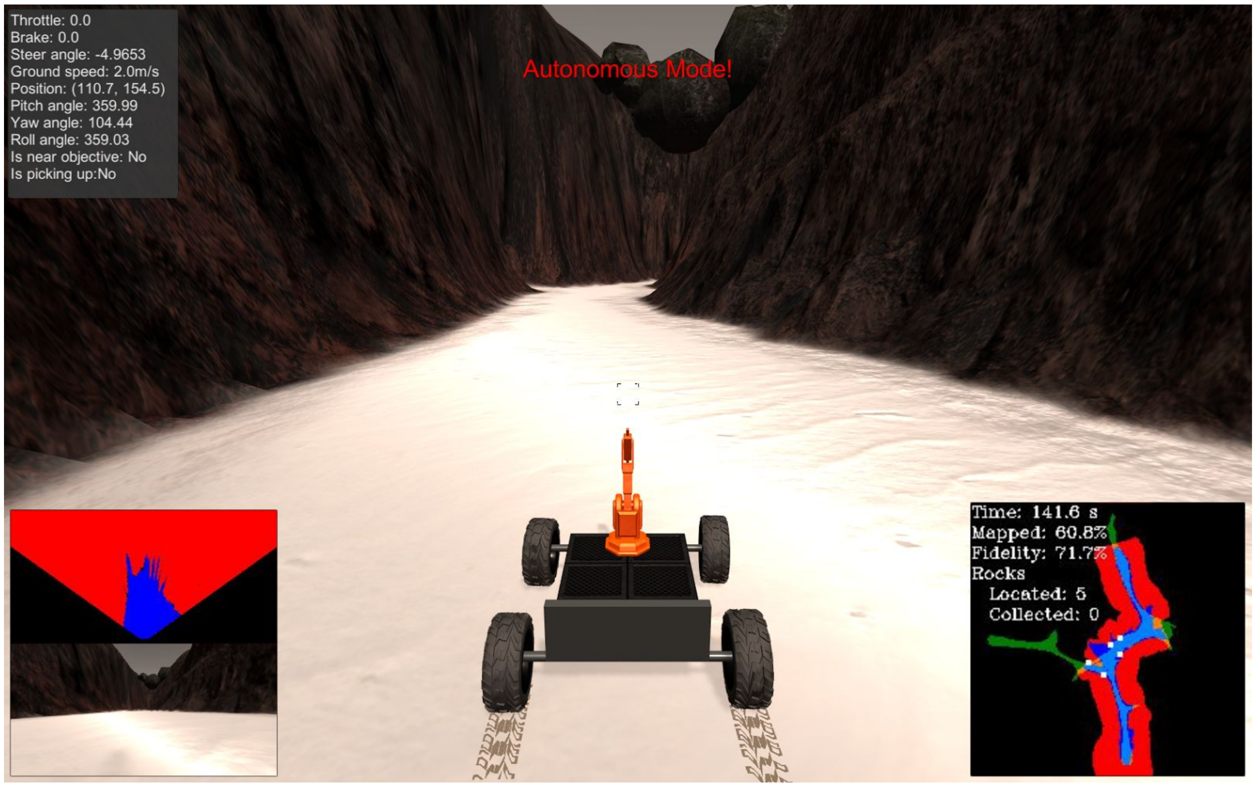
For the *perception_step* function, much of what was added or modified is described in the previous section. Every update, the *rover.vision_image* image is updated with the color-thresholded, warped image using the thresholding values covered earlier. Likewise, the *rover.worldmap* image is updated with the rock, navigable, and obstacle data every update, as long as the pitch and roll are within +/- 1 degree of normal. The *rover.nav_dists* and *rover.nav_angles* fields are updated with rover-centric polar coordinates, which are used by the *decion_step* function to determine the steering angle.

The *decision_step* function itself is not modified from its original form in the project repo. In testing the *perception_step* changes, I found that the rover was able to satisfy the conditions for success without any additional modifications. Essentially, the current decision logic is a state machine with two possible states, "forward" and "stop". In state "forward", the rover will continue accelerating up to it's maximum velocity if there are at least 50 navigable points in the rover's vision. It will also steer towards the average angle of these points. However, if there are less than 50 navigable points, the state changes to the "stop" state. In "stop", the rover throttles back until its velocity is near zero, at which point it rotates in-place until there are greater than 500 navigable points in its vision. Once that condition is met, it will return to the "forward" state and steer towards the average angle to the points.

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

Game Settings:
Resolution: 1680x1050
Quality: Fantastic

Overall, the robot performs well despite the lack of modification to the *decision_step* function. On my last arbitrary run-through, the rover was able to achieve a mapping of 60.8% of the map, with 71.7% fidelity, and five rocks found at 141.6 seconds. Eventually, though the robot will inevitably run into a mid-path obstacle and become stuck.



```
Throttle: 0.0
Brake: 0.0
Steer angle: -4.9653
Ground speed: 2.0m/s
Position: (110.7, 154.5)
Pitch angle: 359.99
Yaw angle: 104.44
Roll angle: 359.03
Is near objective: No
Is picking up:No
```

```
Time: 141.6 s
Mapped: 60.8%
Fidelity: 71.7%
Rocks
  Located: 5
  Collected: 0
```

By setting the color thresholds appropriately, the rover is able to accurately identify the navigable terrain, and avoid driving into walls. Rock locations are identified on the map, since any intersection of the world map projected points with a true rock point is marked as a found sample. Running into obstacles in a wide open space is still very much an issue, as the average angles of the navigable terrain on either side of the obstacle can result in a steering angle of near zero, directly into the obstacle.

To improve future performance and add functional enhancements, I would propose the following for future work:

- If a rock sample is detected via color thresholding, correct the rover's steering angle to the average of the rock points. When the closest point is within range, issue a pickup order to the rover.
- To assist with exploration and rock sample pickup, we could also take a weighted mean of navigable and rock sample angles to determine steering direction, instead of a simple mean. Navigable points which have not been detected or explored before could receive a higher weight than previously explored points. Rock samples would receive a much higher weight, which would force the rover in the direction of the sample.

- To avoid mid-path obstacle collisions, we could check if the chosen steering angle would put us on a collision path with an obstacle within a certain range. If so, we should temporarily pick a hard left or right steering angle to avoid the collision, and return to the normal steering angle selection once the obstacle no longer collides with our steering path.
- A greater fidelity could be achieved by only coloring points on the map which are close to the rover. In other words, we have less confidence in the position of objects farther away from the rover, so we should only map points which are close to the rover. If the map model allows, we could also include a confidence measure when marking map positions, instead of a using a binary representation of our perception of the world. For example, each time we observe an obstacle, navigable terrain, or rock on the map, we increment a counter by one. Points we've mapped many times would have a higher count, which indicates that we have a higher confidence that the mapping is accurate.