Autonomous Rover

Search and Sample Return

Project Submission Writeup Report

for the

Udacity Robotics Software Engineer Nanodegree Program – Term 1

Submitted by:

Stewart Teaze

March 17, 2018

Project Goals

The goals of this project are:

Training / Calibration

- Download the Rover Simulator, and capture/record data in Simulator **Training Mode**.
- Test out the functions in the provided Jupyter Notebook.
- Add functions to detect obstacles and samples of interest (golden rocks).
- The `process_image()` function is filled in with the appropriate image processing steps (perspective transform, color threshold etc.) to get from raw images to a map. The `output_image` created in this step demonstrates a working mapping pipeline.
- The `moviepy` utility is used for processing the images in the saved dataset, by invoking the `process_image()` function, once for each video frame. The resulting video is included in the submitted files.

Autonomous Navigation / Mapping

- Fill in the `perception_step()` function within the `perception.py` script with the appropriate image processing functions to create a map and update `Rover()` data (similar to what was done with `process_image()` in the notebook).
- Fill in the `decision_step()` function within the `decision.py` script with conditional statements that take into consideration the outputs of the `perception_step()` in deciding how to issue throttle, brake and steering commands.
- Command the Rover, operating in Simulator **Autonomous Mode**. Iteratively update and improve the perception and decision functions, until these rover-controlling algorithms do a reasonable (according to a defined metric) job of navigating and mapping.

Submitted Files

1. Rover_Project_Test_Notebook.ipynb – Contains updated Jupyter Notebook.
2. robot_log.csv – Contains record of images, rover position, & steering commands recorded during training session, while running the Simulator in Training Mode.
3. IMG directory – Contains all the image files referenced in robot_log.csv
4. test_mapping.mp4 – moviepy video of autonomous mapping process, created from images output by the process_image() function.
5. drive_rover.py - Contains script for driving the rover in autonomous mode.
6. perception.py - Contains updated perception_step(), and supporting utility functions.
7. decision.py – Contains decision_step(), with significant modifications.
8. RoboND_Term1_Project1_Rover_Search_and_Sample_Return_720p_031618a.mp4 – Video recording of rover driving autonomously thru the test area, mapping 96% of terrain, with > 80% accuracy; with the rover locating and picking up 5 “golden sample rocks”.
9. writeup.pdf - This file is the writeup itself.

These project code and data files can be found at:

https://github.com/blunderbuss9/RoboND-T1P1.html

Rover Simulator and Autonomous Driver Server

The Udacity-provided vehicle-driving Simulator (based on the Unity3D visualization game engine API), can be operated in “Training Mode” or “Autonomous Mode”.

TRAINING MODE: When operated in Training Mode, the Simulator can be manually operated, to generate imagery and steering command data, which can be recorded and saved to disk. The recorded imagery and steering command data can then be used to develop and test a Rover state perception implementation.

AUTONOMOUS MODE: When the Simulator is operated in Autonomous Mode, and used simultaneously and in conjunction with the drive_rover.py autonomous driver server (which utilizes the algorithms in perception.py and decision.py, to perceive the environment, and output steering as well as other commands, that are sent back to the simulator program, corresponding to each frame of video received from the simulator), the simulated Rover will drive autonomously around the Rover’s “world”; mapping the environment, while searching for and recovering samples of interest (golden rocks). To start the drive_rover.py autonomous driver server, execute the following command, after the Simulator is started and placed in Autonomous Mode:

```
python drive_rover.py
```

Rubric Points

---

1. Writeup / README

This Writeup addresses each of the project rubric points (project requirements), and discusses how each dealt with.

The following sections of this document provide details of how each individual requirement was addressed in the project implementation.

2. Notebook Analysis (Training / Calibration)

a. Download the Rover Simulator

I used the Windows10 environment for running the Udacity-provided Rover Simulator, and obtained the simulator from: https://s3-us-west-1.amazonaws.com/udacity-robotics/Rover+Unity+Sims/Windows_Roversim.zip

After unzipping the downloaded simulator .zip file, and installing the package, the Simulator can be run by double-clicking on the installed desktop icon.

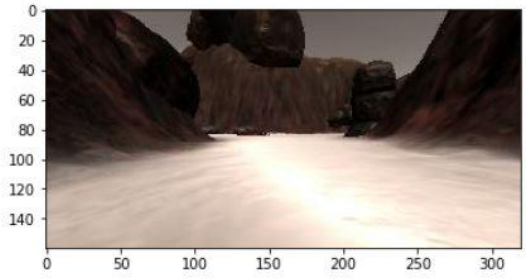
b. Capture/Record Data in Simulator Training Mode

To capture/record simulator training data, run the simulator and select Training Mode, then press the "r" key on the keyboard to begin recording rover camera image frames, as well as rover position and orientation data. Maneuver the rover around the entire environment, to obtain enough data to utilize as input in the Jupyter Notebook implementation, for developing and testing the process_image() and associated utility functions. The file containing the record of image frames and associated rover position/steering commands recorded during training session, while running the Simulator in Training Mode, was saved as robot_log.csv; which is one of the files provided with the project submission.

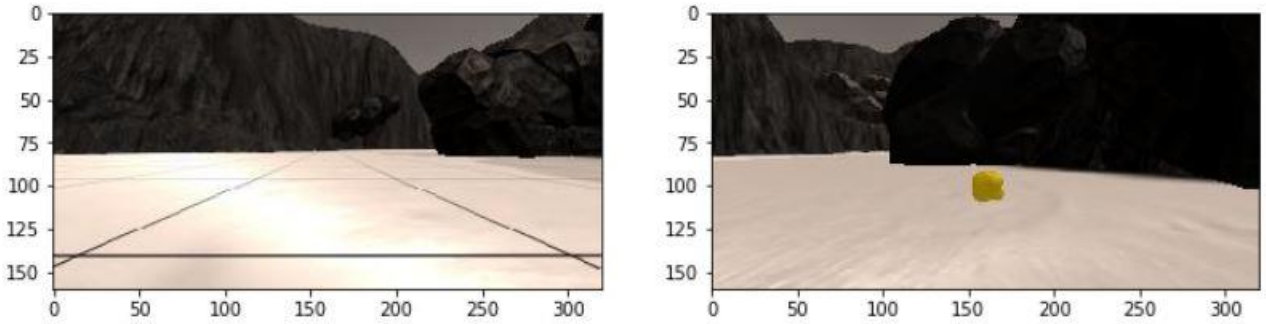
c. Test Out the Functions in the Jupyter Notebook

Before modifying any of the functions in the provided Jupyter Notebook project file, all of the cells were run, in order, to exercise, test and understand the provided capabilities.

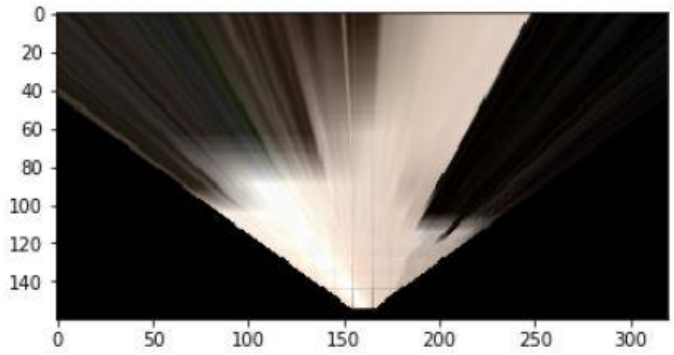
The 3rd code cell was modified to read in a random image from my recorded simulator data; here is an example output image, produced by running this code cell:



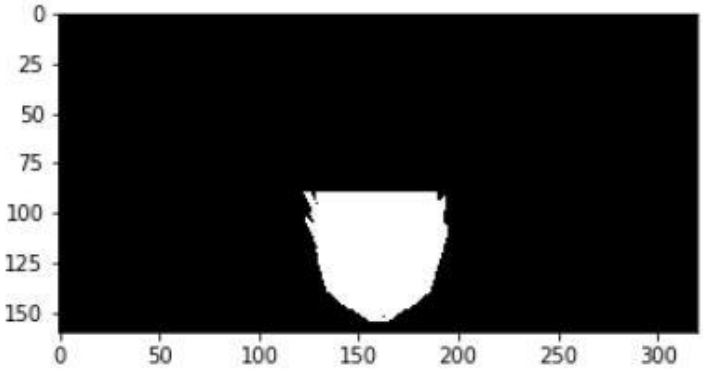
The 4th code cell reads in two provided example grid and rock sample calibration images; here are the output images, produced by running this code cell:



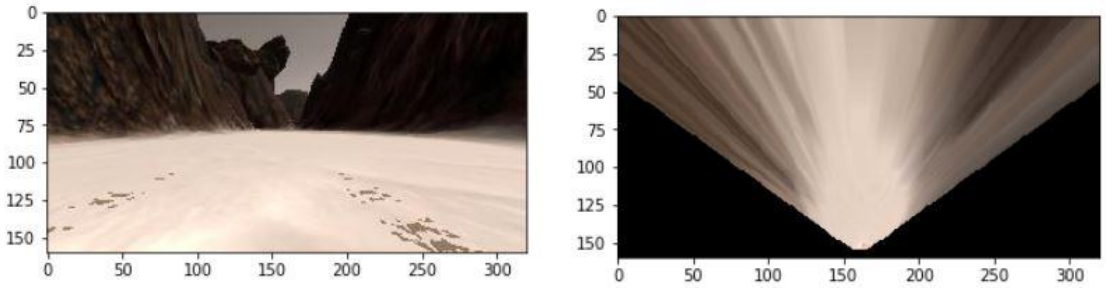
The 5th code cell uses the example grid image from above, for choosing source points for the grid cell in front of the rover (each grid cell is 1 square meter in the sim). The perspect_transform() function is then called, using the defined source and destination points, to "warp" the rover front camera image into an overhead image, which can be used for mapping purposes. Here is the image output produced by running this code cell:



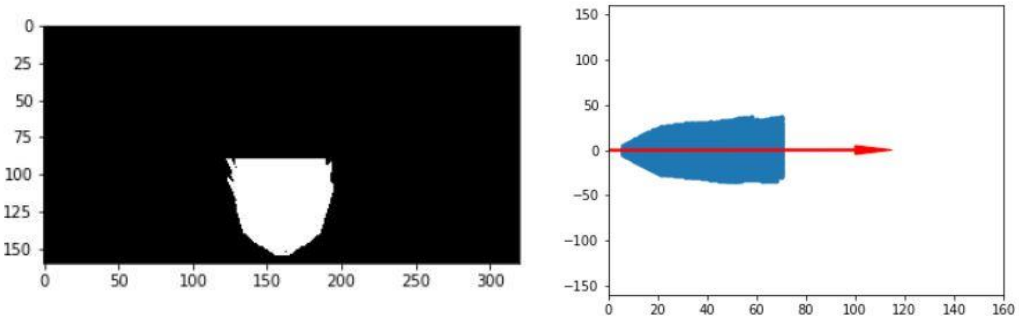
The 6th code cell contains the color thresholding function from the lesson. It is applied it to the warped grid sample image, using source and destination points, calibrated using the grid, to create a top-down map image, useful for identifying navigable terrain. Here is the output of this code cell:



The 7th code cell contains coordinate transformations and the masking utility function that I added. The first two images below show, on the left, a random image taken from the input data set; and on the right, the warped birds-eye view image:

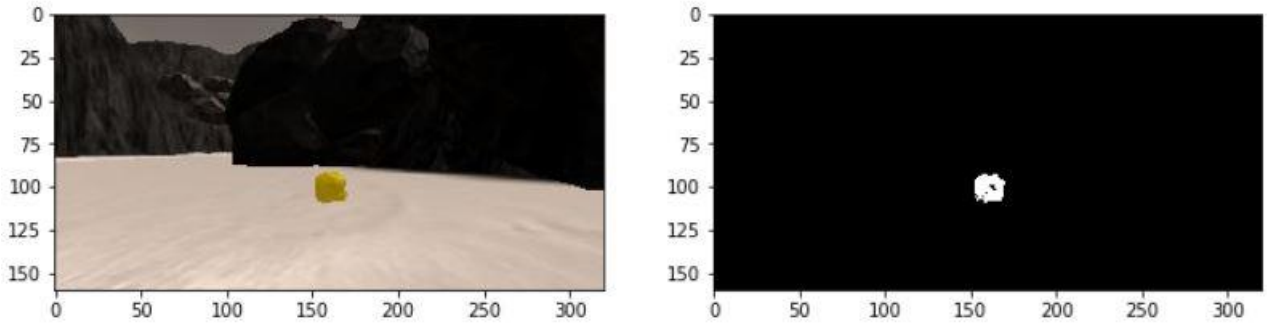


The next set of two images below show, on the left, a color thresholded & masked image; and on the right, this same image converted to rover coordinates, with the rover driving steering angle plotted as the red arrow overlay.:



d. Detecting Obstacles and Samples of Interest

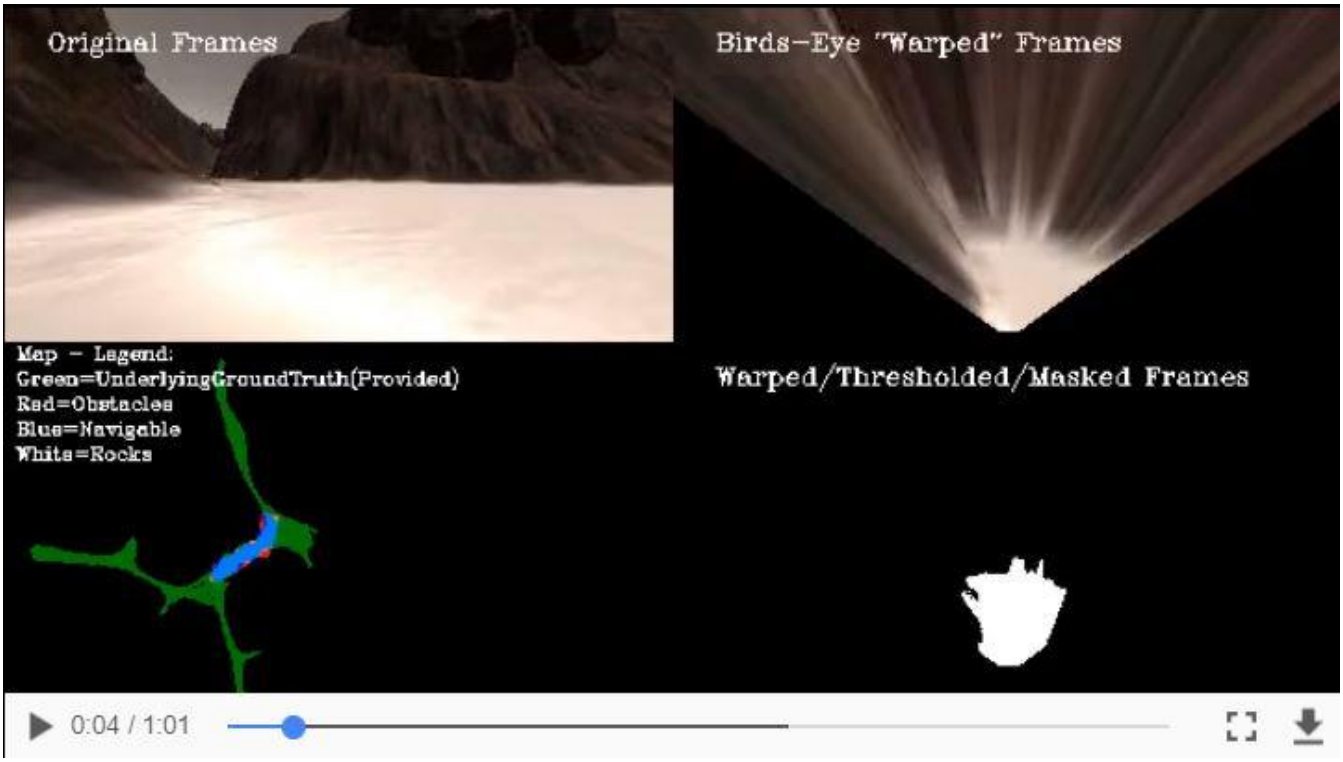
The 8th code cell contains the find_rocks() utility function. The two images below show, on the left, the provided rock test image, and on the right, the output from the color-thresholded (to detect gold pixels) image:



The method used for detecting obstacles, is implemented in, and discussed below, in the process_image() section.

e. Updated process_image() Function

The `process_image()` function was filled in with the appropriate image processing steps (perspective transform, color threshold etc.) to get from raw images to a map showing: 1) the original provided map pixels in green, 2) the detected navigable pixels shown overlayed in blue pixels, 3) the obstacle pixels overlayed in red pixels, and 4) the detected rocks shown overlayed in white pixels. The output images produced by process_image() contain four quadrants of mosaic images: 1) Original Frames are shown in the upper-left quadrant, 2) corresponding Birds-Eye View "Warped" image frames are shown in the upper-right quadrant, 3) Warped/Thresholded/Masked image frames are shown in the lower-right quadrant, and 4) The map, with underlying ground truth(provided), navigable, obstacle, and detected rocks overlays. A sample of one of these images is shown below:



Below is the PSUEDO CODE and NOTES comment section from the process_image() function, which describes the details of the pipeline process sequence implementation used in the code:

```
PSUEDO CODE for process_image():
  1) Apply perspective transform to generate "warped birds-eye view" from original input image
  2) Apply color threshold to identify navigable terrain/obstacles/rock samples
  3) Apply a mask to thresholded image, to eliminate sky, and other extreme extraneous detections
  4) Convert thresholded image pixel values to rover-centric coords
  5) Convert rover-centric pixel values to world coords
  6) Update worldmap (to be displayed in lower-left quadrant)
  7) Construct a mosaic image consisting of original, warped, thresholded, & map images
  8) Place text labels overlayed on top of each of the Quadrants in the image
  9) Return the mosaic/labeled image
NOTES:
  - The "source" and "destination" point variables, used for perspective transform,
    are defined in Code Cell 5 above
  - The "data" object of Class Databucket(), containing general global processing data,
    is defined in Code Cell 8 above
```

f. Utilize moviepy Utility to Create Video from process_image() Output Images

The `moviepy` utility is used for processing the images in the saved dataset, by invoking the `process_image()` function, once for each video frame. The resulting video is included in the submitted files.

3. Autonomous Navigation and Mapping

- Updated the `perception_step()` function within the `perception.py` script with the appropriate image processing functions to create a map and updated `Rover()` data (similar to what was done with `process_image()` in the notebook).
- Updated the `decision_step()` function within the `decision.py` script with conditional statements that take into consideration the outputs of `perception_step()` in deciding how to issue throttle, brake and steering commands.
- Operated the rover in Simulator **Autonomous Mode,** interfaced to and commanded by the drive_rover.py autonomous rover perception/mapping/command server.
- Iteratively updated and improved the perception and decision functions, until those rover-controlling algorithms did a reasonable job of navigating, mapping, as well as identifying rocks, and commanding the rover to approach and pickup these rocks (see Development Log section below, for details). The final recorded session, in which the commanded Rover achieved perception and gathering of 5 of the 6 possible golden rocks, is saved in the following file, which is included in the project submission package: RoboND_Term1_Project1_Rover_Search_and_Sample_Return_720p_031618a.

a. Development Log

MAR 12 9:37am: OK, my "Rover Driver Program" is commanding the Rover to move around and map the "Gaming Area" semi-decently in Sim Autonomous mode.

MAR 12 8:09pm: I've got the Rover tooling around and mapping the entire gaming area autonomously... I had to enter a significant "left turning bias", to get it to stick to the left hand walls, and make it down every channel and crevice.

MAR 12 5:48pm: The rover tended to get stuck along the sides of canyon walls, so I had to modify the code to detect these “stuck” situations, and apply enough throttle to get out of the situation.

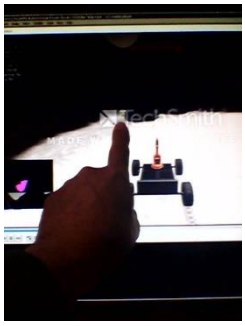
MAR 14 8:09am: I've got things so that I can identify/map both terrain and rocks in Jupyter Notebook... now have to move new rock id/map code into Rover Perception code... then need to control the Rover so it goes over to the rocks (and stop when the simulator sets the Rover.rock_nearby RoverState Class variable to True, at which point I can command the rover to perform a pickup, which will cause the rover to automatically pick up the nearby rock and put them in its bed).

MAR 14 1:03pm: I made a couple more "fine control adjustments", the Rover can now tool around the environment forever, without ever getting stuck, and identifies/locates (on average) 4 of 6 of the target rocks... It maps about 97% of the terrain (requirement is 40% or better), at >81%

accuracy (>60% is required). There were about 4 specific tricks I came up with to have it do a good job traversing the environment, mapping navigable terrain & target rocks, and not getting stuck. First was to steer along the left side of the pathways, another thing was to go slower than the default speed (helped in a number of ways – in detecting rocks, or in maneuvering out of difficult situations), another was to "power out" of situations where the Rover would get stuck, and I also found a good way to mask (not consider) terrain above the horizon (greatly reduced false reporting of terrain and rocks). Another important trick was to use the numpy standard deviation function, to determine if a large bolder was straight ahead (such as appear in the middle, open area, at the center of the terrain), and then take corrective action so that the rover wouldn't drive straight at the boulder (because the detected terrain was "balanced" on both sides of the boulder) and get stuck.

MAR 14 6:09pm: Installing Camtasia to record an autonomous-driven Simulator session. 1080p files are too big – going to 720p moving forward. Can use Handbrake application to encode and compress the video files when necessary… but found it was unnecessary if I used 720p setting in Camtasia to begin with.

MAR 14 9:10pm: When a rock gets randomly hidden in the crevice behind the big rock in that corner, it is hard to detect... I'd have to do a whole lot of special processing to maneuver the Rover into a position that it could detect it.



MAR 16 11:29am: Hooray – got a rock! Got another! I had needed to sleep on it, and attack the problem with a "fresh brain".

MAR 16 11:40am: It was tough, but I figured out what had made it so difficult to implement steering towards the rocks and slowing down, when I had attempted it before (There was default decision code that made the Rover do something squirrelly when the speed dropped too low, that had to be "worked around", as a "special case", when I was steering towards a rock).

MAR 16 1:16pm: In final recording session, achieved perception and gathering of 5 of the 6 possible golden rocks.

MAR 16 2:13pm: OBSERVATION: The Rover Simulator camera display is basically the same as the Predator UAV Nose Camera HUD (Heads-Up Display) in the Predator GCS (Ground Control Station) Pilot rack, and the map display in the simulator lower right-hand corner is the same as the Tracker Map display in the center rack of the Predator GCS.