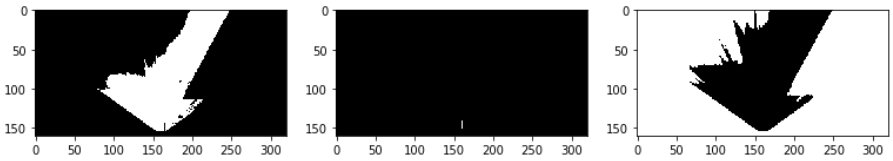
Robotics Nanodegree writeup for Project 1
Submission located at https://github.com/Chambana/RoboND-Rover-Project

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.	**Describe in your writeup (and identify where in your code) how you modified or added functions to add obstacle and rock sample identification.** The process was very similar for color selection of obstacles, rock samples, and the road. My first step was to manually created RGB threshold ranges for the obstacles, the road, and the gold nuggets. My initial RGB values for these threshold ranges were obtained using the provided matplotlib plot as a starting point and refining the ranges with trial and error. Of note, I believe that these threshold could possibly be refined further for even greater fidelity, but they currently meet specification. These 3 threshold ranges (e.g. for the road, obstacles, and rock samples) were all applied independently using functions color_thresh(), obstacle_thresh(), and gold_nugget_thresh()) to an image taken from the rover's perspective. Per previous exercises, the rover-provided image was first warped (in the function perspect_transform()) using a 1 meter grid to 10 pixel mapping. This process yielded three binary images representing the presence (and non-presence) of the road, obstacles, and samples. These binary images representing the road pixels, rock sample pixels, and obstacle

pixels are shown below, respectively:



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.

Describe in your writeup how you modified the `process_image()` to demonstrate your analysis and how you created a worldmap. Include your video output with your submission.

Writing process_image() was a 7 step process. First, I added code to define the parameters of the perspective transform ("warping"), to specify how the perspective transform will map pixels in the source image (rover perspective) to the destination image (overhead, flattened perspective). I then invoked the perspect_transform() function using these parameters, yielding a warped image from an overhead perspective. Next, I independently applied the 3 thresholding operations specified above to this perspective transformed image -- providing 3 overhead perspective binary imagse showing the location of road pixels, obstacles pixels, and rock sample pixels. However, these images were still represented with respect to a camera frame and needed to be converted to rover centric coordinates, effectively performing a translation

	and rotation operation. More simply, this moves the rover's perspective from the bottom middle of the frame to, instead, looking down the positive X axis from the origin (0,0). This was accomplished by calling the rover_coords() function on all three thresholded images (road, obstacles, nuggets). Finally, I had to represent these coordinates in World frame. Again, this called for a rotation and translation using a typical homogeneous transform matrix and the matrix format appropriate for a CCW rotation. These transformed points were then applied to the appropriate dimension of the 3 dimensional worldmap array to mark the areas that have either road, obstacles, or nuggets. A mosaic display of the perspective view, the overhead view, and the worldmap was created by copying the relevant numpy arrays to arrays of the rectangular display rectangle. Of note, a weighted image combine operation was applied to the worldmap and the provided ground truth data to create an overlay effect. Video of my analysis on the provided sample data is attached in the output directory of my github repository..

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()` and `decision_step()`functions have been filled in and their functionality explained in the writeup.** The perception_step() function was lifted almost verbatim from the process_image() function detailed above. Notably, Step 4 was changed to include the creation of a “Rover Vision” image. This image was displayed in the lower left of the real-time display in the simulator and displays, from an overhead perspective, the location of the road pixels in blue, the location of the obstacle pixels in red, and the location of the nugget pixels in green. Another notable change was in Step 7, where the code updates the world map. I added a conditional statement to restrict updates to the world map to specific moments when the rover’s roll and pitch are both within a +/-0.2 range of 0 (note that pitch and roll range from 0-360). This limited noisy measurements from entering the world map and provided better fidelity scores. The range 0.2 was arbitrarily chosen through limited trial and error and stricter ranges could yield even higher fidelity. The provided code for decision_step() enables 2 modes, “forward” and “stop”. In forward mode, the Rover will steer towards the mean of the angles present in the navigable road (the limits on steering are +/-15 degrees, so you can’t steer directly to a given yaw angle). I made four primary functionality changes to the code provided in the decision_step function: added rock sample detection and pickup logic, added detection and handling of a “rover is stuck” situation, added detection and handling of a “rover is

	driving in a circle” situation, and added functionality to quit and change the mode to “mission complete” if all rock samples have been collected and the rover is within a user-selectable range from the start location. For detecting and picking up rock samples, I used a 2 step procedure. First, if a rock sample was detected in the image, I would steer to the mean navigation angle associated with the rock sample pixels. Once I detected I was near the objective (e.g. the rock sample), I initially just stopped the vehicle prior to initiating a pickup command. This was to overcome edge cases where I would overshoot the rock sample after initiating a pickup command. Subsequently, once stopped, I would send the pickup command. These rock sample pickup sequences trump all other control logic. Second, I found that the rover would get “stuck” on obstacles in the roadway. To detect this situation, I instituted logic to detect if the vehicle was not responding to throttle commands (e.g. velocity==0 and throttle!=0). I wrote a brief “breakout” function to take evasive action in this situation -- which simply commanded slight, but repeated steering commands intended to “unstick” the rover. Further, I also detected an edge case where the rover could get stuck in a loop and drive donuts for a very long, but not infinite, amount of time. So, I crafted logic to detect a long, consistent turn and, similar to above, integrated slight evasive action steering controls to break out of the loop. All of the code corresponding to detecting and handling this condition are contained in state variables containing the word “donut”. Lastly, I added code to log the start point X,Y coords at the onset of the simulation and, subsequently, calculate my distance “Home” using a simple pythagorean theorem calculation. This calculation is used to decide “mission accomplished” once the Rover has collected all 6 rock

	samples and has returned to within, for instance, 5 meters of the start location.

Launching in autonomous mode your rover can navigate and map autonomously. Explain your results and how you might improve them in your writeup.	**By running `drive_rover.py` and launching the simulator in autonomous mode, your rover does a reasonably good job at mapping the environment.** **The rover must map at least 40% of the environment with 60% fidelity (accuracy) against the ground truth. You must also find (map) the location of at least one rock sample** **Note: running the simulator with different choices of resolution and graphics quality may produce different results, particularly on different machines! Make a note of your simulator settings (resolution and graphics quality set on launch) and frames per second (FPS output to terminal by `drive_rover.py`) in your writeup when you submit the project so your reviewer can reproduce your results.** I primarily ran my simulation in the 1920x1080/beautiful mode, however, I found that I have similar results in the 640x480/fastest mode. I consistently stay above 60% fidelity when mapping >96% of the map and collecting (and eventually returning) all 6 rock samples.

	My Rover has, thus far, been able to free itself from any obstacle collision or donut loop driving condition. There are situations where my Rover takes a very long time to find the 6th and final rock sample if it's particularly well hidden and my Rover could be improved to incorporate smarter logic with respect to not revisiting areas already searched. Further, if I slowed down the Rover speed or increased the strictness of my pitch/roll ranges (used for filtering updates to the world map), I could likely achieve greater fidelity. To further improve fidelity, I could develop more stringent color threshold ranges, to ensure only accurate pixels are mapped to the world map -- at the expense of the speed of mapping. The code did not crash during any of my final test trials and, while many critical math operations are in try/except blocks to handle exceptions, I do periodically receive a warning when I do a mean operation on certain matrices. This warning has no effect on the functionality of the code.

To have a standout submission on this project you should not only successfully navigate and map the environment in "Autonomous Mode", but also figure out how to collect the samples and return them to the starting point (middle of the map) when you have found them all!	My code can map nearly the entire environment with above specification fidelity, collect all 6 rocks, and will quit when the rover returns to within a (user-selectable) distance from starting location -- assuming all rocks have already been collected. Further, my code will detect if the vehicle has entered an “endless donut” and will provide logic to break out of this loop. My code provides similar logic to get out of situations where the vehicle is stuck (e.g. the velocity is 0 and the throttle is non-zero). An image showing a successful run is attached in my github directory under the output directory.