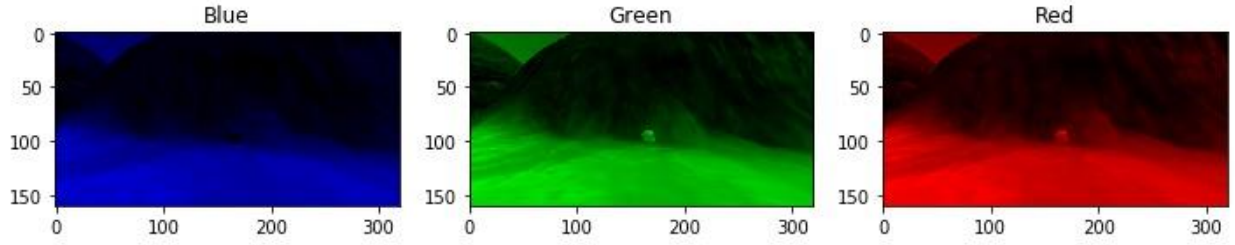
Search and Sample Return

A brief explanation for items in the project rubric is provided below:

Notebook Analysis

1. *color_thresh_cv2(rgb_img, rgb_thresh_low=[0,0,0], rgb_thresh_high=[160,160,160])*: function performs thresholding on R,G and B channels
    a. Reads in a RGB image, converts to BGR image and applies min and max thresholds to individual color channels using inbuilt OpenCv method *cv2.inRange*
2. *Rock Sample and Obstacle identification:*
    a. Rock samples, navigable terrain and obstacles were identified by applying min and max threshold values for blue, green, red channels. Rocks had low max thresholds in blue and higher in green and red. Terrain had high min and max threshold in all three channels. The remaining pixels were identified as obstacles (Figure 1).

A



B

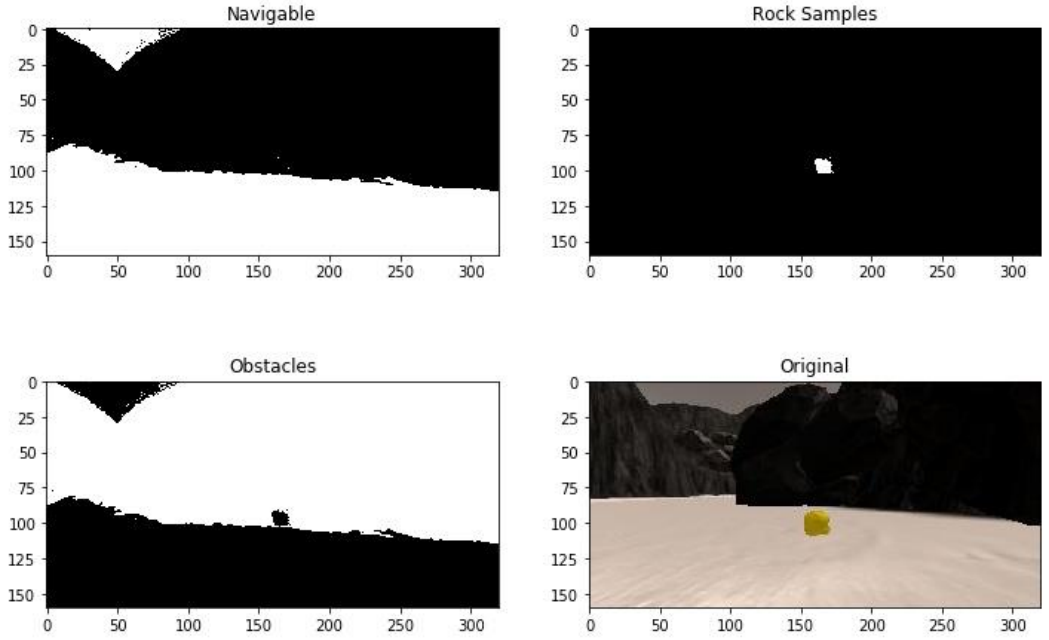


Figure 1. A. Blue, Green and Red channels from a sample image. Color thresholds: rock (min=[0 80 80], max=[60 255 255]) and terrain (min=[100 100 100], max=[255 255 255]). Color threshold for terrain were increased in **perception.py** to force rover onto center of the navigable terrain in the map. B. Images showing results of applying thresholds for rocks, terrain and obstacles

3. *process_image (img):* This method takes an image generated by rover camera as input and outputs a mosaic image - original image + color coded image representing rocks, terrain and obstacles + ground truth map overlaid with navigation map. The following steps were carried out by this method
    a. Performed perspective transform on the camera image using source and destination points. Source points were identified from camera view image. Destination points spanned the center point on the x-axis and offset from the bottom.
    b. Color threshold was applied to identify terrain, rock and obstacle pixels using the method *color_thresh_cv2()*. Appropriate threshold values were used after trial and error
    c. Converted rock, terrain and obstacle pixels to rover-centric coordinates
    d. Converted rover centric coordinates to world coordinates
    e. World map was created by creating a 3D array. Each channel of 3D array corresponded to rock, terrain and obstacle pixels in world coordinates.
    f. Mosaic image generation: a large blank image was generated and following images were added to different positions to generate the mosaic:
        i. Image from camera view
        ii. Warped image
        iii. World map was blended with ground truth

Autonomous Navigation and Mapping

1. perception.py: Two changes were made. *Color_thresh_cv2* method was added to identify obstacles, rocks and terrain through thresholding. perception_step(Rover) was modified.
    a. *perception_step(Rover)*:
        i. Performed perspective transform to generate a warped image.
        ii. Binary images of terrain, rocks and obstacles was generated from warped image using color thresholding
        iii. Rover.vision state variable was updated with binary images of rock, terrain and obstacles
        iv. Rocks, terrain and obstacles pixel location were converted to rover-centric coordinates
        v. Rover-centric coordinates were converted to coordinates in world frame
        vi. World map of rover was updated with newly identified rock, terrain and obstacle pixels in world frame. This was done by setting rock, terrain and obstacle pixels to 1 in the green, blue and red channels.
        vii. Rover-centric coordinates were converted to polar form and Rover state was updated with navigable angles and distances information

2. decision.py: The basic intuition is that to make a decision to move in direction of motion, the rover only needs to know whether there are obstacles nearby. Therefore, the rover only considers pixels in the close range and uses them to determine navigable angular regions. Angular regions are identified as contiguous regions of close range navigable pixels in polar coordinates. If the width in unobstructed view is above a certain threshold (say >= 10 degrees), that direction is considered navigable. If no such angular region is found, an obstacle is reported.

The methods *decision_step(), get_steer_angle(), no_obstacle_ahead()* were added/modified in decision.py

a. *decision_step(Rover)*: This method reads in Rover state information and updates its mode of action. The decision step cycles the Rover between 3 modes of action namely 'forward', 'stop', 'blocked' (Figure 2). The state transition diagram for these three modes are provided below. Briefly,
    1. Forward mode: if there are no obstacles, the rover continues in forward mode in the direction determined by *get_steer_angle()*. If there are obstacles, the rover throttle is set to zero and rover mode is set to 'stop'.
    2. Stop mode: if there are no obstacles, rover throttle is set and rover mode is set to 'forward'. If there are obstacles, then rover steers to right by 10 degrees to find navigable terrain. Rover continues in the stop mode
    3. Blocked: Rover is steered to the right by 20 degree and rover mode is set to 'forward'. The newly added 'blocked' mode refers to a situation where the rover is in 'forward' mode with no obstacles but is somehow stuck and is unable to move. This mode is entered if rover is stuck (vel < 0.05) for > 10sec continuously.

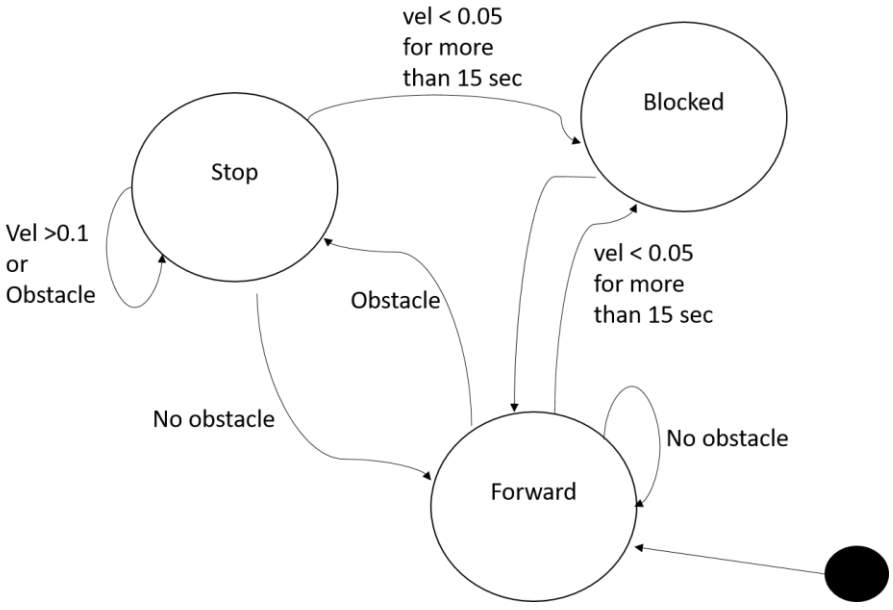


Figure2. State transition diagram

b. *no_obstacle_ahead(dist, angles, thresh, angle_thresh)*: Determines if field in front of rover is navigable. Also returns list of navigable angular regions (2-tuple of bounding angles) that are available to choose from. Only beams of angular width > 10 degrees are acceptable (selected by trial and error).

c. *get_steer_angle():* determines the steer angles based provided list of direction beams. The valid range of angles is from -90 to 90. The steering angle takes the mid value of most positive direction beam (leftmost). This ensures rover's behavior to be a wall crawler (with wall to its left). Note that wall crawling is **akin to depth-first search on a graph** and I think that is the best search strategy for optimizing time.

3. drive_rover.py: New state variables were added to keep track of stopping time Added new fields to rover state, max_allowed_left, blocked_start_time

4. supporting_function.py:

    a. *update_rover(Rover, data)*: updates the mode to 'blocked' if the vel < 0.05 continuously for > 10sec

Results

1. Map coverage: Covers > 80% of the map
2. Sample detection: Detects 4-5 samples. With further work (undergoing) it should be able to pick samples
3. Fidelity: A major weakness. Fidelity ranges between 30-70%. Because of its wall crawling, most of its time is spend on slanted slopes with a few degrees variation in both pitch and roll. To improve fidelity a stricter threshold has been tried for terrain so that it does not hug the wall. This has resulted in consistent performance of ~60% range. Using low max velocity and throttle values and braking also seems to improve performance
4. Mapping Time: Takes about 600-800 seconds before stopping (> 80% coverage). Higher velocities cannot be used near the wall because it leads to crashing which may adversely affect fidelity. The only solution is to be able to stay away from the wall and move at higher velocities. Another strategy is to use long range information such that rover moves faster on long navigable stretches.
5. Anomalous behaviors: Rover sometimes reverses direction and revisits previously mapped areas (this reduces time performance). It also sometimes gets stuck between 'blocked' and 'stop' modes indefinitely (if it takes > 10 seconds to come out of stop and velocity is zero). The incidence depends on whether the rover comes across a specific region of the map. It happens 1/5 times.