Project: Search and Sample Return

The goals / steps of this project are the following:

1. **Training / Calibration**
   - To download the simulator and take data in "Training Mode".
   - To test out the functions in the Jupyter Notebook provided.
   - To add functions to detect obstacles and samples of interest (golden rocks).
   - To fill in the `process_image()` function 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 should demonstrate that the mapping pipeline works.
   - To use `moviepy` to process the images in the saved dataset with the `process_image()` function. Then, include the video produced as part of the project submission.

2. **Autonomous Navigation / Mapping**
   - To 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 is done with `process_image()` in the notebook).
   - To 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.
   - To iterate on the perception and decision function until the rover does a reasonable job of navigating and mapping (it is requirement to map at least 40% of the environment at 60% fidelity and locate at least one of the rock samples).

Notebook Analysis

Color thresholding

The color thresholding function has been modified so that not only the produced map includes navigable terrain but also obstacles and the positions of the rock samples that are being searched for.

For obstacle detection, the color selection that was used to detect ground pixels was inverted, where the pixels above the threshold represent navigable terrain and the pixels below it represent obstacles. The following code is added to achieve this functionality:

```
obstacle_thresed= (navigable_threshed * -1) +1
```

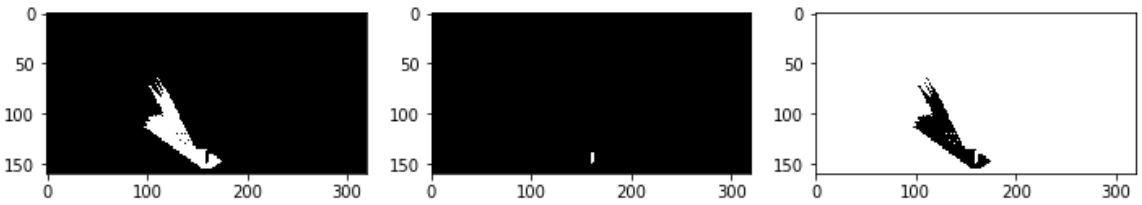
For rocks detection, first the image is converted from BGR to HSV using openCV's. Then a range of the yellow color (rocks' color) is defined in HSV by knowing that the color yellow is a mix of the colors green and red. Later, the HSV image is threshed to only obtain the yellow colors. This is the used threshold function to detect rocks:

```
def rock_thresh(img):

    # Convert BGR to HSV
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    # define range of yellow color in HSV
    lower_yellow = np.array([50,100,100])
    upper_yellow = np.array([255,255,255])

    # Threshold the HSV image to get only yellow colors
    yellowMask = cv2.inRange(hsv, lower_yellow, upper_yellow)

    return yellowMask
```

The following figures represent the navigable terrain threshed, rocks threshed and obstacles threshed respectively.

```
fig = plt.figure(figsize=(12,3))
plt.subplot(131)
plt.imshow(navigable_threshed,cmap='gray')
plt.subplot(132)
plt.imshow(rock_threshed, cmap='gray')
plt.subplot(133)
plt.imshow(obstacle_thresed, cmap='gray')


#scipy.misc.imsave('../output/warped_threshed.jpg', threshed*255)
```

```
Out[7]: <matplotlib.image.AxesImage at 0x1a1588527b8>
```



function to process stored images

The process_image() function is modified by adding in the perception step processes to perform image analysis and mapping in the following steps:

1. Define source and destination points for perspective transform.
2. Define calibration box in source (actual) and destination (desired) coordinates, these source and destination points are defined to warp the image to a grid where each 10x10 pixel square represents 1 square meter. The destination box will be 2*dst_size on each side.
3. Set a bottom offset to account for the fact that the bottom of the image is not the position of the rover but a bit in front of it

```
# 1) Define source and destination points for perspective transform
# Define calibration box in source (actual) and destination (desired) coordinates
# These source and destination points are defined to warp the image
# to a grid where each 10x10 pixel square represents 1 square meter
# The destination box will be 2*dst_size on each side
dst_size = 5
# Set a bottom offset to account for the fact that the bottom of the image
# is not the position of the rover but a bit in front of it
# this is just a rough guess, feel free to change it!
bottom_offset = 6
source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
destination = np.float32([[img.shape[1]/2 - dst_size, img.shape[0] - bottom_offset],
                  [img.shape[1]/2 + dst_size, img.shape[0] - bottom_offset],
                  [img.shape[1]/2 + dst_size, img.shape[0] - 2*dst_size - bottom_offset],
                  [img.shape[1]/2 - dst_size, img.shape[0] - 2*dst_size - bottom_offset],
                  ])
```

4. Apply perspective transform.

```
# 2) Apply perspective transform
warped = perspect_transform(img, source, destination)
```

5. Apply color threshold to identify navigable terrain.
6. Apply color threshold to identify obstacles.
7. Apply color threshold to identify rocks.

```
# 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
# color threshold to detect navigable terrain
navigable_threshed= navigable_thresh(warped, rgb_thresh=(160, 160, 160))
# color threshold to detect rock samples
rock_threshed = rock_thresh(warped)
# color threshold to detect obstacles
obstacle_thresed= (navigable_threshed *-1) +1
```

8. Calculate pixel values in rover-centric coords and distance/angle to all pixels for navigable terrain, rock samples and obstacle.

```
# 4) Convert thresholded image pixel values to rover-centric coords
# Calculate pixel values in rover-centric coords and distance/angle to all pixels for navigable terrain
xpix_navigable, ypix_navigable = rover_coords(navigable_threshed)
dist_navigable, angles_navigable = to_polar_coords(xpix_navigable, ypix_navigable)
mean_dir_navigable = np.mean(angles_navigable)

# Calculate pixel values in rover-centric coords and distance/angle to all pixels for rock samples
xpix_rock, ypix_rock = rover_coords(rock_threshed)
dist_rock, angles_rock = to_polar_coords(xpix_rock, ypix_rock )
mean_dir_rock = np.mean(angles_rock)

# Calculate pixel values in rover-centric coords and distance/angle to all pixels for obstacles
xpix_obstacle, ypix_obstacle = rover_coords(obstacle_thresed)
dist_obstacle, angles_obstacle = to_polar_coords(xpix_obstacle, ypix_obstacle)
mean_dir_obstacle = np.mean(angles_obstacle)
```

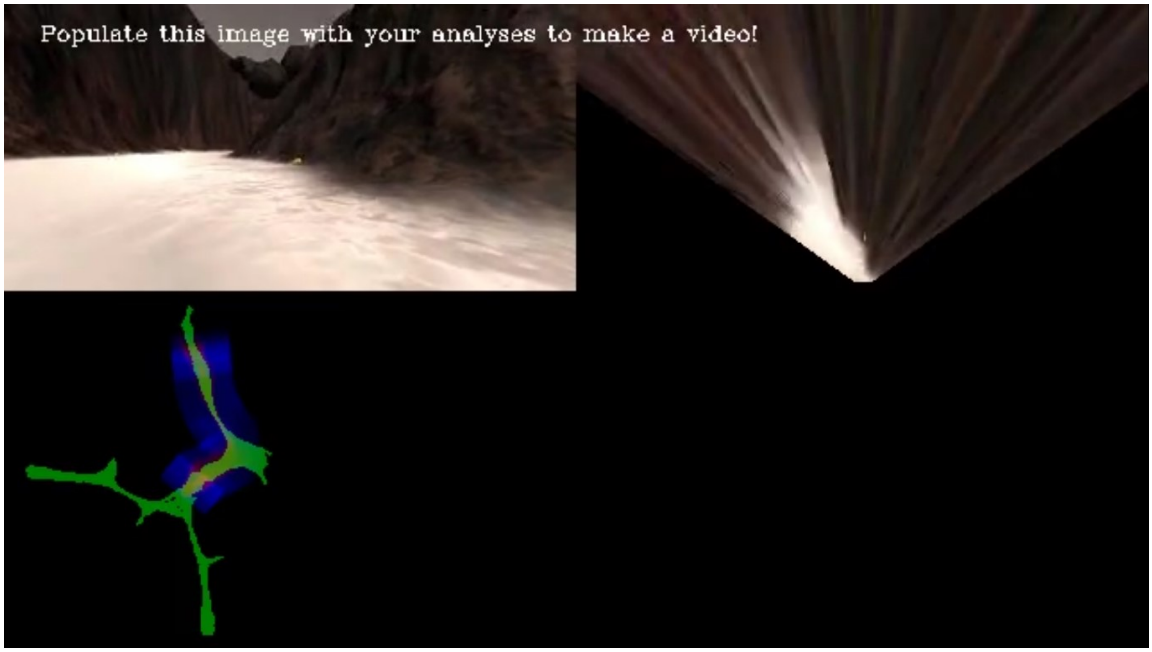
9. Get navigable terrain, rock samples and obstacles pixel positions in world cords.

```
# 5) Convert rover-centric pixel values to world coords
scale = 10
# Get navigable pixel positions in world coords
x_world, y_world = pix_to_world(xpix_navigable, ypix_navigable, data.xpos[data.count],
                                data.ypos[data.count], data.yaw[data.count],
                                data.worldmap.shape[0], scale)


# Get rock sample pixel positions in world coords
rock_x_world, rock_y_world = pix_to_world(xpix_rock, ypix_rock, data.xpos[data.count],
                                data.ypos[data.count], data.yaw[data.count],
                                data.worldmap.shape[0], scale)

# Get obstacle pixel positions in world coords
obstacle_x_world, obstacle_y_world = pix_to_world(xpix_obstacle, ypix_obstacle, data.xpos[data.count],
                                data.ypos[data.count], data.yaw[data.count],
                                data.worldmap.shape[0], scale)
```

10. Update worldmap (to be displayed on right side of screen) where the navigable terrain will be represented in red, rock samples in green and obstacle in blue.

```
# 6) Update worldmap (to be displayed on right side of screen)
data.worldmap[obstacle_y_world, obstacle_x_world, 2] += 1
data.worldmap[rock_y_world, rock_x_world, 1] += 1
data.worldmap[y_world, x_world, 0] += 1
```

This is a screenshot from the created video using moviepy functions after running process_image() function on the recorded data.



Autonomous Navigation and Mapping

Perception

In the perception_step() function the same algorithm as in process_image() function is used to perform image analysis and mapping. In this function all the Realtime captured data is coming from the RoverState() class from drive_rover.py such as the camera image, current position, yaw , etc.

The modifications done to perception function include:

Updating the Rover.vision_image in a way that will increase the fidelity of the map using the following code:

```
# 4) Update Rover.vision_image (this will be displayed on left side of screen)
white_image = np.ones_like(Rover.img[:,:,0])
white_warped = perspect_transform(white_image, source, destination)

Rover.vision_image[:,:,0] = obstacle_thresed *255*white_warped
Rover.vision_image[:,:,1] = rock_threshed *255*white_warped
Rover.vision_image[:,:,2] = navigable_threshed *255*white_warped
```

This code generates a white image sized as the captured image by the rover. This image is then transformed using the perpective_transform function resulting in a warped image that has black pixels. The black pixels represent unknown areas and therefore they must be neglected in the color threshed rock, navigable terrain, and obstacle images. To neglect these unknow pixels, the color threshed images are multiplied by the RGB white_warped image as shown in the above code.

drive_rover.py

In this code, the maximum velocity value for the rover was reduced from 2.0 to 1.6 (meters/second) as that as resulted in a higher fidelity rates, also the threshold to initiate stopping is set at 100 to avoid collisions.

```
# The stop_forward and go_forward fields below represent total count
# of navigable terrain pixels.  This is a very crude form of knowing
# when you can keep going and when you should stop.  Feel free to
# get creative in adding new fields or modifying these!
self.stop_forward = 100 # Threshold to initiate stopping
self.go_forward = 500 # Threshold to go forward again
self.max_vel = 1.6 # Maximum velocity (meters/second)
```

Launching in autonomous mode

The autonomous mode results show that the rover does map more than 40% of the environment at more than 60% fidelity and can locate the rock samples.

The Simulator settings were set at 1280 X 600 resolution with Good graphics quality at 17 frames per second.

By reducing the speed and neglecting the unwanted parts from the warped image, the rover's fidelity rates got increased. To better improve the results the rocks should have been picked and not only located. This should have been done by making the rover get closer to the rock samples and then stopping when it is closer to them. This was not done due to time constrains by the author.