

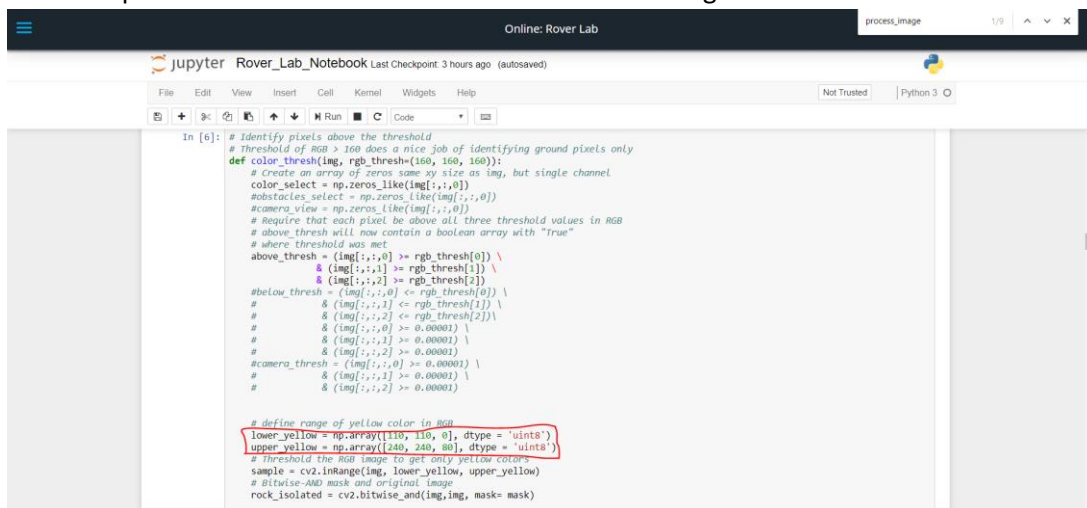
Project: Search and Sample Return

Writeup / README

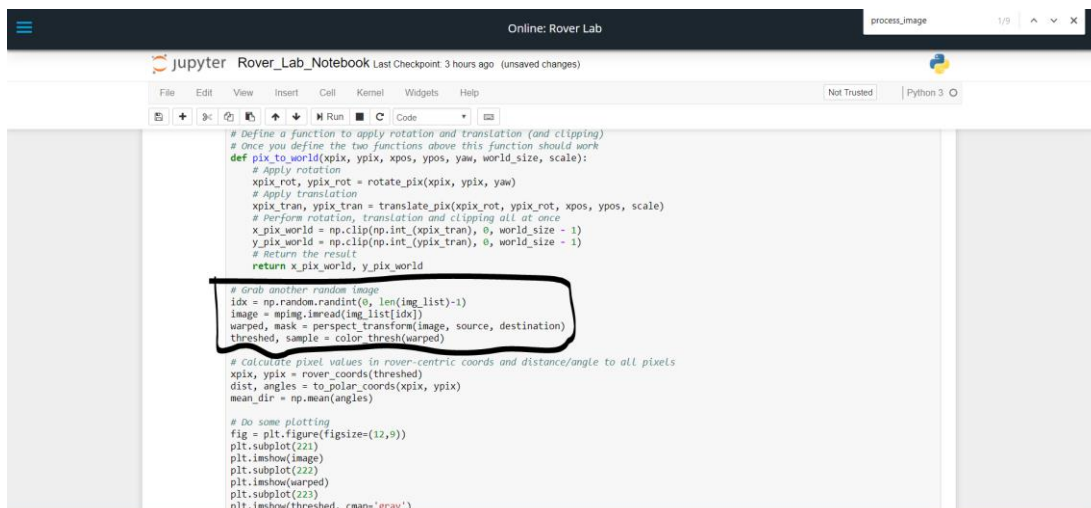
Notebook Analysis

1. 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.

I first ran the functions in the notebook to see what they did. After going through and understanding how, I modified them to test changes in thresholding to produce the best filter for the navigable terrain as well as the gold rocks. I also tried several different ways to produce a nice warped navigation image. I only used the test images provided because it was just easier and could randomly generate another to test multiple cases with the default code that called in images.



```
In [6]: # Identify pixels above the threshold
# Threshold of RGB > 160 does a nice job of identifying ground pixels only
def color_thresh(img, rgb_thresh=(160, 160, 160)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:, :, 0])
    # obstacles select = np.zeros_like(img[:, :, 0])
    # camera view = np.zeros_like(img[:, :, 0])
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "true"
    # where threshold was met
    above_thresh = (img[:, :, 0] >= rgb_thresh[0]) \
        & (img[:, :, 1] >= rgb_thresh[1]) \
        & (img[:, :, 2] >= rgb_thresh[2])
    # below_thresh = (img[:, :, 0] <= rgb_thresh[0]) \
    # & (img[:, :, 1] <= rgb_thresh[1]) \
    # & (img[:, :, 2] <= rgb_thresh[2])
    #
    # & (img[:, :, 0] >= 0.00001) \
    # & (img[:, :, 1] >= 0.00001) \
    # & (img[:, :, 2] >= 0.00001)
    # camera_thresh = (img[:, :, 0] >= 0.00001) \
    # & (img[:, :, 1] >= 0.00001) \
    # & (img[:, :, 2] >= 0.00001)
    #
    # Define range of yellow color in RGB
    lower_yellow = np.array([110, 110, 0], dtype='uint8')
    upper_yellow = np.array([240, 240, 80], dtype='uint8')
    # Threshold the RGB image to get only yellow colors
    sample = cv2.inRange(img, lower_yellow, upper_yellow)
    # Bitwise-AND mask and original image
    rock_isolated = cv2.bitwise_and(img, img, mask=sample)
```



```
# Define a function to apply rotation and translation (and clipping)
# Once you define the two functions above this function should work
def pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale):
    # Apply rotation
    xpix_rot, ypix_rot = rotate_pix(xpix, ypix, yaw)
    # Apply translation
    xpix_tran, ypix_tran = translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale)
    # Perform rotation, translation and clipping all at once
    x_pix_world = np.clip(np.int(xpix_tran), 0, world_size - 1)
    y_pix_world = np.clip(np.int(ypix_tran), 0, world_size - 1)
    # Return the result
    return x_pix_world, y_pix_world

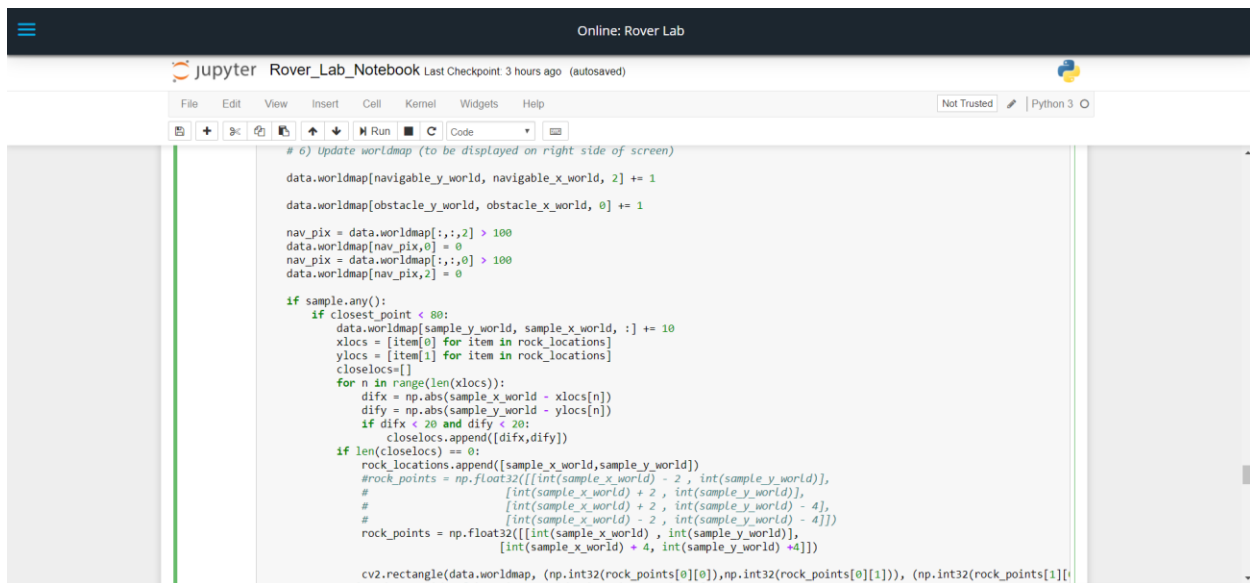
# Grab another random image
idx = np.random.randint(0, len(img_list)-1)
image = mpimg.imread(img_list[idx])
warped, mask = perspective_transform(image, source, destination)
threshed, sample = color_thresh(warped)

# Calculate pixel values in rover-centric coords and distance/angle to all pixels
xpix, ypix = rover_coords(threshed)
dist, angles = to_polar_coords(xpix, ypix)
mean_dir = np.mean(angles)

# Do some plotting
fig = plt.figure(figsize=(12,9))
plt.subplot(221)
plt.imshow(image)
plt.subplot(222)
plt.imshow(warped)
plt.subplot(223)
plt.imshow(threshed, cmap='gray')
```

2. 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.

It did not take long to finish the functions for identifying navigable terrain, obstacles and rock samples. It was a simple mask and apply to different channels, and taking each respective set of pixels and transform them to the rover coordinate system for recognition of its path, and then further coordinate transformation to world pixels was done through the transform functions. I was also able to Locate rock samples with a square indicator using the closest pixel of the sample to the rover in the provided test data but later found out it was unnecessary and was already provided.



```
# 6) Update worldmap (to be displayed on right side of screen)

data.worldmap[navigable_y_world, navigable_x_world, 2] += 1

data.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1

nav_pix = data.worldmap[:, :, 2] > 100
data.worldmap[nav_pix, 0] = 0
nav_pix = data.worldmap[:, :, 0] > 100
data.worldmap[nav_pix, 2] = 0

if sample.any():
    if closest_point < 80:
        data.worldmap[sample_y_world, sample_x_world, :] += 10
        xlocs = [item[0] for item in rock_locations]
        ylocs = [item[1] for item in rock_locations]
        closelocs=[]
        for n in range(len(xlocs)):
            difx = np.abs(sample_x_world - xlocs[n])
            dify = np.abs(sample_y_world - ylocs[n])
            if difx < 20 and dify < 20:
                closelocs.append([difx,dify])
        if len(closelocs) == 0:
            rock_locations.append([sample_x_world,sample_y_world])
            #rock_points = np.float32([[int(sample_x_world) - 2 , int(sample_y_world)],
            #                        [int(sample_x_world) + 2 , int(sample_y_world)],
            #                        [int(sample_x_world) - 2 , int(sample_y_world) - 4],
            #                        [int(sample_x_world) + 2 , int(sample_y_world) - 4]])
            rock_points = np.float32([[int(sample_x_world) , int(sample_y_world)],
                                     [int(sample_x_world) + 4, int(sample_y_world) + 4]])
            cv2.rectangle(data.worldmap, (np.int32(rock_points[0][0]),np.int32(rock_points[0][1])), (np.int32(rock_points[1][0],np.int32(rock_points[1][1])), (np.int32(rock_points[1][1]),np.int32(rock_points[1][0])), (255,0,0))
```

Autonomous Navigation and Mapping

1. 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.

I modified the perception.py script to mirror the same functions that were created earlier and modified in the online notebook. I modified the decision.py script to include a conditional statement if it got stuck using the telemetry, so if the throttle was active but the velocity was zero, it would back up and turn as if to perform a three point turn. I tried modifying the steering command to hug the wall with an offset to the default output but it either hugged the wall too much or not enough so I wasn't able to find the magic number.

```
10
11 # Example:
12 # Check if we have vision data to make decisions with
13 ▼ if Rover.nav_angles is not None:
14     # Check for Rover.mode status
15     if Rover.mode == 'forward':
16         # Check the extent of navigable terrain
17     ▼ if len(Rover.nav_angles) >= Rover.stop_forward:
18         # If mode is forward, navigable terrain looks good
19         # and velocity is below max, then throttle
20         Rover.throttle = Rover.throttle_set
21     ▼ if Rover.vel > Rover.max_vel:
22         # Set throttle value to throttle setting
23         Rover.throttle = 0
24     ▼ elif Rover.vel == 0 and Rover.throttle == Rover.throttle_set and Rover.total_time > 1:
25         Rover.mode = 'stuck'
26         if Rover.total_time - Rover.timer < 0.5:
27             Rover.mode = 'forward'
28         else:
29             Rover.mode = 'stuck'
30     Rover.brake = 0
31     # Set steering to average angle clipped to the range +/- 15
32     Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15) # + 8, -15, 15)
33
```

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

After trying different techniques, I found that with the default steering decision, the robot was able to traverse around obstacles and to all branches of the map well and was able to map on average 75% and with a fidelity of around 60%. Identifying the rocks was easy but I did not pick them up. So to improve, I would add a conditional statement that if a rock was recognized, it would use the location of the pixels to guide it toward the rock, and when close enough to stop and pick it up. Also, I would improve by finding the right steering condition for it to hug the wall. I used the fantastic graphics with a resolution of 1280 x 1024 with the default FPS setting in drive_rover.py.