README

Project 1

Summary

This project over all needs to create an autonomous system for navigation, identifying samples, and mapping the area.

To execute the above, following are the steps required.

1. Setting up Jupyter Notebook.
2. Writing Process_image function
3. Making suitable changes in perception.py, Decision.py and drive_rover.py.
4. Running in autonomous mode to check for efficiency.

This documentation includes the explanation to the codes added or modified in the entire project, along with the codes and images. The document is divided into the following three segments

1. Jupyter Notebook Explanation- Process Image and perception step
2. Decision.py
3. Challenges, obstacles and other ideas for the project.

1. Notebook Explanation

Step 1 - Importing Libraries

This Notebook start with importing all the important libraries such as OpenCV, Numpy, Matplotlib, Scipy, Glob etc. These libraries and packages help us modify, alter or identify the necessary parameters needed for various operations such as avoiding obstacles, identifying navigable terrain and recognizing sample rocks later.

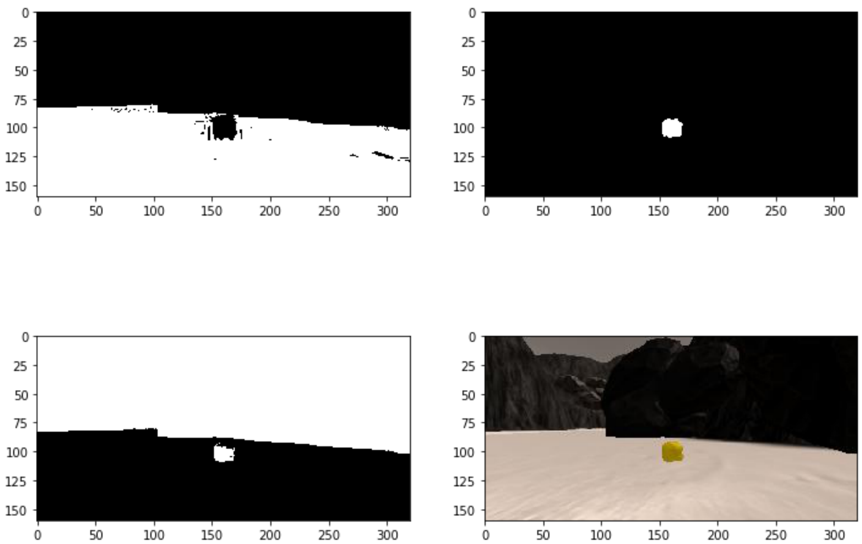
```python
%matplotlib inline
#%matplotlib qt # Choose %matplotlib qt to plot to an interactive window (note it may show up behind your browse
# Make some of the relevant imports
import cv2 # OpenCV for perspective transform
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import scipy.misc # For saving images as needed
import glob  # For reading in a list of images from a folder
```

Step 2 - Color Thresholding

The next part is important where we specify the color threshold for path, obstacle and rock separately in the following codes.

```python
# In the simulator you can toggle on a grid on the ground for calibration
# You can also toggle on the rock samples with the 0 (zero) key.
# Here's an example of the grid and one of the rocks
example_grid = './calibration_images/example_grid1.jpg'
example_rock = './calibration_images/example_rock1.jpg'
grid_img = mpimg.imread(example_grid)
rock_img = mpimg.imread(example_rock)

def color_thresh(img, rgb_thresh=(160, 160, 160, 100 , 100,50)):
    # Create an array of zeros same xy size as img, but single channel
    color_select_path = np.zeros_like(img[:,:,0])
    color_select_rock = np.zeros_like(img[:,:,0])
    color_select_obstacle = 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])
    yellow_thresh = (img[:,:,0] > rgb_thresh[3] ) \
                & (img[:,:,1] > rgb_thresh[4] ) \
                & (img[:,:,2] < rgb_thresh[5] )
    below_thresh = (img[:,:,0] < rgb_thresh[0]) \
                & (img[:,:,1] < rgb_thresh[1]) \
                & (img[:,:,2] < rgb_thresh[2])
    # Index the array of zeros
    color_select_path[above_thresh] = 1
    color_select_rock[yellow_thresh] = 1
    color_select_obstacle[below_thresh] = 1
    # Return the binary image
    return color_select_path, color_select_rock, color_select_obstacle
```

Here we use numpy to apply the threshold for the three different things. Using these thresholds, we then plot the images to see the outcome using matplotlib.

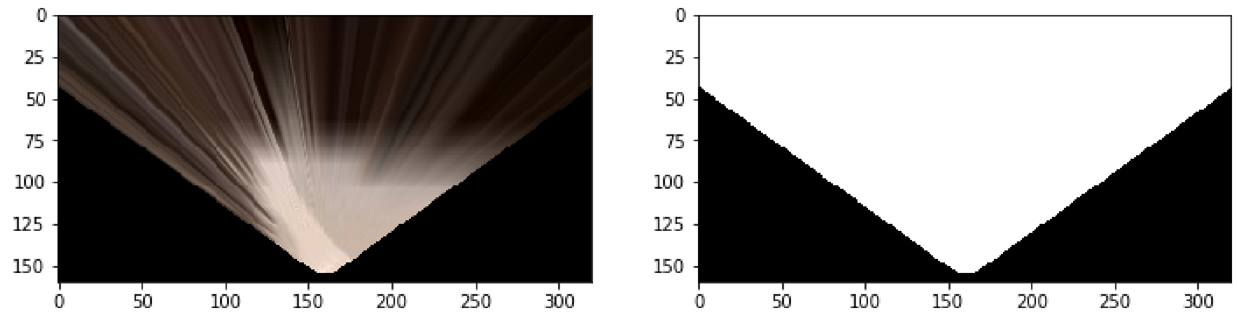


Step 3- Perception Transform

Next we perform Perception Transform. The idea is to convert the image available to the rover from a different point of view as done in the images given below. The view from camera of rover is used to convert into a map.

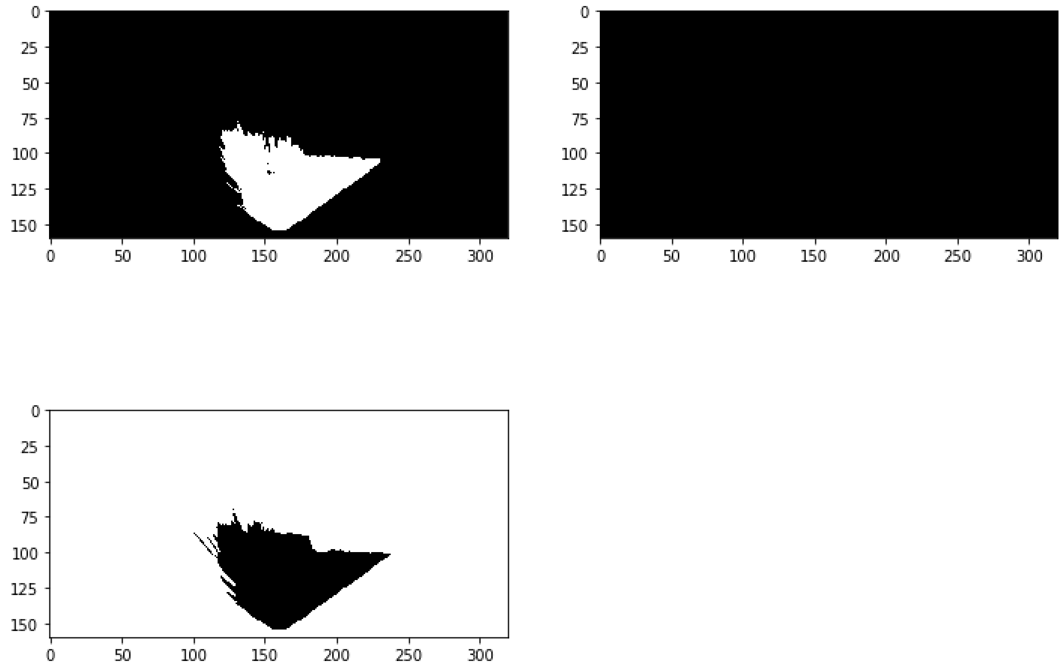
```
def perspect_transform(img, src, dst):
        #We get the transformation matrix using cv2.getPerspectivTransform()
        M = cv2.getPerspectiveTransform(src, dst)
        warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))# keep same size as input image
        outView = cv2.warpPerspective(np.ones_like(img[:,:,0]), M, (img.shape[1],img.shape[0]))
        return warped, outView

dst_size = 5
bottom_offset = 6
#we carefully choose the source and destination positions
source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
destination = np.float32([[image.shape[1]/2 - dst_size, image.shape[0] - bottom_offset],
            [image.shape[1]/2 + dst_size, image.shape[0] - bottom_offset],
            [image.shape[1]/2 + dst_size, image.shape[0] - 2*dst_size - bottom_offset],
            [image.shape[1]/2 - dst_size, image.shape[0] - 2*dst_size - bottom_offset],
            ])
warped, outView = perspect_transform(image, source, destination)
fig = plt.figure(figsize=(12,3))
plt.subplot(121)
```

```
plt.imshow(warped)
plt.subplot(122)
plt.imshow(outView, cmap='gray')
#scipy.misc.imsave('../output/warped_example.jpg', warped)
```



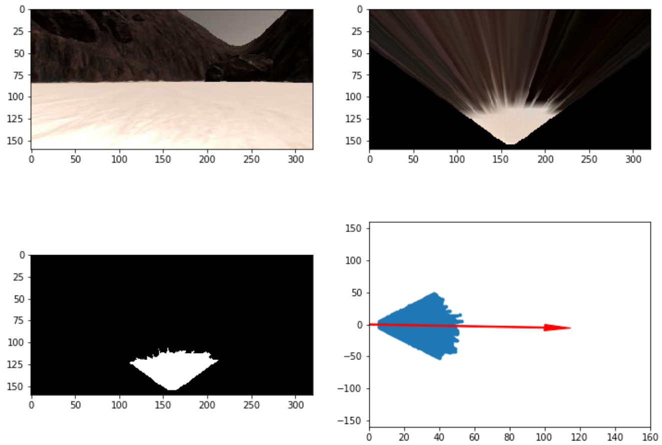
Similarly, using the thresholding methods we used before, we plot maps with thresholding as follows.

```
fig = plt.figure(figsize=(12,9))
plt.subplot(221)
plt.imshow(threshed_path,cmap='gray')
plt.subplot(222)
plt.imshow(threshed_rock,cmap='gray')
plt.subplot(223)
plt.imshow(threshed_obs, cmap='gray')
```



Step 4- Coordinate Transform

This is an important step in mapping the area. That is done by defining various functions to perform transformations, such as rotation, translation, converting to polar coordinates, localizing the location of rover etc. Th functions used here are **rover_coords, to_polar_coords, rotate_pix, translate_pix** and **pix_to_world**. The results of those are as follows.



Pandas is then imported to read the CSV file saved before and to read in ground truth map and create a 3-channel image with it. We then create a class called Databucket to be the data container.

Step 5- process_image

In the next step, we create a function named process_image to create a video from each single frame that we have saved before, by adding the perception transformations performed earlier. All the steps are joined together to create one video.

```
def process_image(img):
    # Example of how to use the Databucket() object defined above
    # to print the current x, y and yaw values
    # print(data.xpos[data.count], data.ypos[data.count], data.yaw[data.count])
    dst_size = 8

    # TODO:
    # 1) Define source and destination points for perspective transform
    # 2) Apply perspective transform
    warped, mask = perspect_transform(img, source, destination)
    # 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
    threshed_path, threshed_rock, threshed_obs = color_thresh(warped)
    obstacles_world = np.absolute(np.float32(threshed_obs))*mask
```

```
# 4) Convert thresholded image pixel values to rover-centric coords
 xpix, ypix = rover_coords(threshed_path)
# 5) Convert rover-centric pixel values to world coords
world_size = data.worldmap.shape[0]
scale = 2*dst_size
xpos = data.xpos[data.count]
ypos = data.ypos[data.count]
yaw = data.yaw[data.count]
x_world, y_world = pix_to_world(xpix,ypix,xpos,ypos,yaw,world_size,scale)

obs_xpix, obs_ypix = rover_coords(obstacles_world)
obs_xworld, obs_yworld = pix_to_world( obs_xpix,obs_ypix,xpos,ypos,yaw,
world_size,scale)

rock_xpix, rock_ypix = rover_coords(threshed_rock)
rock_xworld, rock_yworld =
pix_to_world(rock_xpix,rock_ypix,xpos,ypos,yaw,world_size,scale)
# 6) Update worldmap (to be displayed on right side of screen)

data.worldmap[y_world, x_world,2] = 255
data.worldmap[obs_yworld,obs_xworld,0] = 255
data.worldmap[rock_yworld,rock_xworld,1] = 255

#below, these two methods prevent the obstacle areas to overwrite the path area.
nav_pix = data.worldmap[:,:,2] > 0
data.worldmap[nav_pix, 0] = 0

# 7) Make a mosaic image, below is some example code
# First create a blank image (can be whatever shape you like)
output_image = np.zeros((img.shape[0] + data.worldmap.shape[0], img.shape[1]*2, 3))
# Next you can populate regions of the image with various output
output_image[0:img.shape[0], 0:img.shape[1]] = img

# Let's create more images to add to the mosaic, first a warped image
warped, mask = perspect_transform(img, source, destination)
# Add the warped image in the upper right hand corner
output_image[0:img.shape[0], img.shape[1]:] = warped

# Overlay worldmap with ground truth map
map_add = cv2.addWeighted(data.worldmap, 1, data.ground_truth, 0.5, 0)
# Flip map overlay so y-axis points upward and add to output_image
output_image[img.shape[0]:, 0:data.worldmap.shape[1]] = np.flipud(map_add)


# Then putting some text over the image
cv2.putText(output_image,"Populate this image with your analyses to make a video!",
(20, 20),
```

```
    cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
    if data.count < len(data.images) - 1:
        data.count += 1 # Keep track of the index in the Databucket()

    return output_image
```

Using the codes mentioned in the notebook, we create a video that shows how the map is being creating using the pipeline we just developed. The video is attached as test_mapping.mp4.

```
In [31]:
# Import everything needed to edit/save/watch video clips
from moviepy.editor import VideoFileClip
from moviepy.editor import ImageSequenceClip


# Define pathname to save the output video
output = './output/test_mapping.mp4'
data = Databucket() # Re-initialize data in case you're running this cell multiple times
clip = ImageSequenceClip(data.images, fps=60) # Note: output video will be sped up because
                                          # recording rate in simulator is fps=25
new_clip = clip.fl_image(process_image) #NOTE: this function expects color images!!
%time new_clip.write_videofile(output, audio=False)
```

```
[MoviePy] >>>> Building video ./output/test_mapping.mp4
[MoviePy] Writing video ./output/test_mapping.mp4

100%|██████████| 283/283 [00:06<00:00, 40.45it/s]

[MoviePy] Done.
[MoviePy] >>>> Video ready: ./output/test_mapping.mp4

CPU times: user 3.87 s, sys: 101 ms, total: 3.97 s
Wall time: 7.42 s
```

2. Explanation of decision.py

The decision.py script is to control the movement of Rover through throttle, brakes and steering. The codes below present various settings for throttle, brakes and steering in various situations like, being around a rock sample, getting stuck at an obstacle or while simply moving within the navigable terrain. The codes are mentioned in the decision.py file.

3. Challenges and ideas

I got to learn a lot about how complicated actions can be taken by simplifying the problem statement. The perception transformation and decision.py were examples of that. The challenging part for me was to get comfortable with coding and testing. If i did have a little more time and scope, I would have liked to introduce other elements through the Unity Simulator, in order to present the rover with more options and scope for taking decisions.

Also I would like to make the codes more robust, with further adding more functions in decision.py where the rover is not just responding to its immediate environment but also intelligently mapping the area.