Laura Curioso
4 Sep 2017

Project 1: Search and Sample Return Writeup

1. Notebook Analysis:

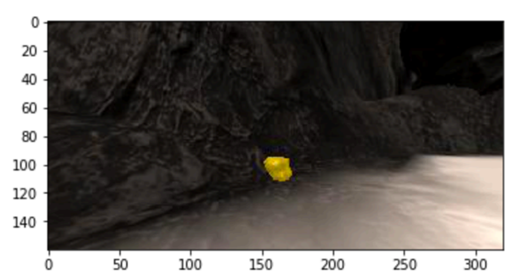
   a. Simulator and Environment Setup

      i. After the simulator was downloaded I setup the environment on a mac using Anoconda and downloaded the RoverND starter kit. Code blocks were run on Jupyter Notebook where I was able to test out code from each module. After recording some test data I was able to see the output images in a .csv file which I could use for calibration and test cases for the modules.

   b. Perception Step

      i. Using the notebook, I was able to import the matplotlib library which allowed an image to be plotted and show the pixel locations on the 2D image. From here the numpy library printing functions: data type, image shape, pixel min value, and pixel max value prints out their respective data concerning the selected image. These images are 8 bit so the output reads "unit8". Image shape outputs the array size values for the channels. The pixel min and max are 0 and 255. 0 counts as an integer which is why the maximum value is 255 not 256 for pixel colors. Image data recorded throughout this project is kept with these array and pixel properties. Figure 1 Below shows the test code run based on the sample image that provides the image data:

```
In [46]: # Define the filename, read and plot the image
         filename = 'RoboND-Rover-Project/calibration_images/example_rock2.jpg'#'images/robocam_2017_05_2
         image = mpimg.imread(filename)
         plt.imshow(image)
         plt.show()
```




```
In [47]: import numpy as np
         print(image.dtype, image.shape, np.min(image), np.max(image))

uint8 (160, 320, 3) 0 255
```

Figure 1: Numpy Print Image Functions

      ii. Next, the image was separated out into the red, green and blue channels by running the code below shown in Figure 2:

```
In [5]: red_channel = np.copy(image)
red_channel[:,:,[1, 2]] = 0 # Zero out the green and blue channels
green_channel = np.copy(image)
green_channel[:,:,[0, 2]] = 0 # Zero out the red and blue channels
blue_channel = np.copy(image)
blue_channel[:,:,[0, 1]] = 0 # Zero out the red and green channels
fig = plt.figure(figsize=(12,3)) # Create a figure for plotting
plt.subplot(131) # Initialize subplot number 1 in a figure that is 3 columns 1 row
plt.imshow(red_channel) # Plot the red channel
plt.subplot(132) # Initialize subplot number 2 in a figure that is 3 columns 1 row
plt.imshow(green_channel)  # Plot the green channel
plt.subplot(133) # Initialize subplot number 3 in a figure that is 3 columns 1 row
plt.imshow(blue_channel)  # Plot the blue channel
plt.show()
```

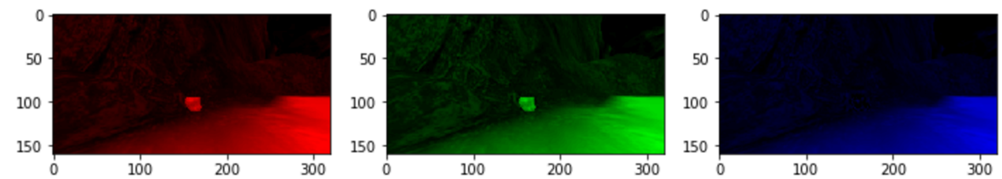


Figure 2: Red, Green, Blue Channel Plots of the Image

iii. Next the image was color thresholded by running the following code using threshold values of (160, 165, and 160):

```
In [231]: # Identify pixels above the threshold
# Threshold of RGB > 160 does a nice job of identifying ground pixels only - changed to 165

def color_thresh(img, rgb_thresh=(165, 160, 160)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = 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])
    # Index the array of zeros with the boolean array and set to 1
    color_select[above_thresh] = 1
    # Return the binary image
    return color_select

threshed = color_thresh(warped)
plt.imshow(threshed, cmap='gray')
#scipy.misc.imsave('../output/warped_threshed.jpg', threshed*255)

Out[231]: <matplotlib.image.AxesImage at 0x125faa7f0>
```

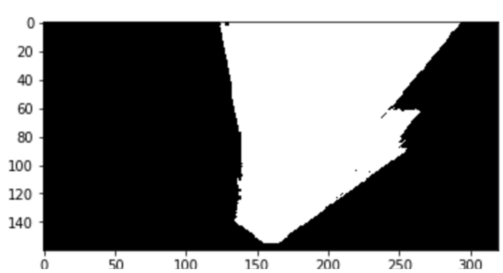


Figure 3:Thresholded Image Plot

iv. This allowed enough of the path to be “shown” by turning all pixels above 160 to black while the others below that pixel color range were turned to white by making those turn to an array of 1s' and 0's respectively. Typically True =1 and False = 0.

c. Perspective Transform

i. Purpose of the perspective transform is to turn the 3D map into a 2D map where we can tell the rover where to go by defining source and destination points. We have a nice grid layout and based on the pixel positions we can tell the robot move to this "location" based on this calibrated "scale". Take for instance a fisheye camera's curved image. This convolution would change it into a "flat" image.

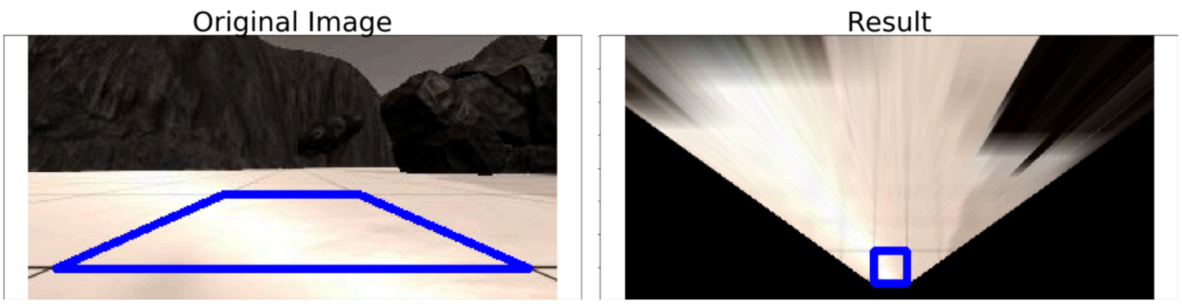
ii. We first start with a calibration image that has the "grid square" nicely centered in the image. After manually finding the points using the mouse cursor, you set those manual points as the source points. For setting the destination points, you want to consistently have the robot go to an offset that is ideally the "next square". The source points are identified in the "source" array and the destination points are offset by ½ the distance size which can be seen in Figure 4 below.

```
In [39]: #to warp the image:
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2
filename = 'images/image2.jpg'
image = mpimg.imread(filename)

def perspect_transform(img, src, dst):

    # Get transform matrix using cv2.getPerspectivTransform()
    M = cv2.getPerspectiveTransform(src, dst)
    # Warp image using cv2.warpPerspective()
    # keep same size as input image
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))
    # Return the result
    return warped

# Define source and destination points - i will round to make the intersections the same:
dst_size = 10#5
bottom_offset= 10#6
source = np.float32([[16, 141], [303 ,141],[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],
                  ])
```

Figure 4: Source and Desitnation Points Size and selection Math/Logic

iii. The "warped" function performs the perspective transform on the source and destination array yielding a "top-down" view of the image shown in Figure 5 below.

```
warped = perspect_transform(image, source, destination)
# Draw Source and destination points on images (in blue) before plotting
cv2.polylines(image, np.int32([source]), True, (0, 0, 255), 3)
cv2.polylines(warped, np.int32([destination]), True, (0, 0, 255), 3)
# Display the original image and binary
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 6), sharey=True)
f.tight_layout()
ax1.imshow(image)
ax1.set_title('Original Image', fontsize=40)

ax2.imshow(warped, cmap='gray')
ax2.set_title('Result', fontsize=40)
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
plt.show()
```



iv. In this top-down view, the floor grid is much more visible now and the color thresholding shows the "walls" which we will assign as "obstacles". This map has a high contrast between the "navigable terrain" and the "obstacle walls".

d. Warp, Threshold, and Map

i. Warped function returns the x y positions as an upside down based on the origin points (0, 0). To reverse the direction to give the rover meaningful polar coordinates to travel to, you define the x postiona nd y position as a binary image and place a "-" in front of the binary image shape using the code below:

x_pixel = -(ypos - binary_img.shape[0]).astype(np.float)

y_Pixel = -(xpos - binary_img.shape[1]/2).astype(np.float)

The added code in the notebook then performs a perspective transform and thresholds the image to give the "correct" mapped terrain relative to the rover position shown in

```python
In [ ]: import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        import numpy as np
        import cv2
        from extra_functions import perspect_transform, color_thresh, source, destination

        # Read in the sample image
        image = mpimg.imread('sample.jpg')


        def rover_coords(binary_img):
            # TODO: fill in this function to
            # Calculate pixel positions with reference to the rover
            ypos, xpos = binary_img.nonzero()
            # position being at the center bottom of the image.
            x_pixel = -(ypos - binary_img.shape[0]).astype(np.float)
            y_pixel = -(xpos - binary_img.shape[1]/2 ).astype(np.float)
            return x_pixel, y_pixel

        # Perform warping and color thresholding
        warped = perspect_transform(image, source, destination)
        colorsel = color_thresh(warped, rgb_thresh=(160, 160, 160))
        # Extract x and y positions of navigable terrain pixels
        # and convert to rover coordinates
        xpix, ypix = rover_coords(colorsel)

        # Plot the map in rover-centric coords
        fig = plt.figure(figsize=(5, 7.5))
        plt.plot(xpix, ypix, '.')
        plt.ylim(-160, 160)
        plt.xlim(0, 160)
        plt.title('Rover-Centric Map', fontsize=20)
        plt.show() # Uncomment if running on your local machine
```

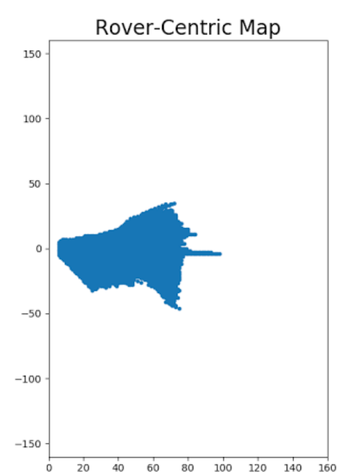


Figure 5: Flipped Image for Rover Polar Coordinates

e. Map to World Coordinates
   i. Now that we have polar rover coordinates we need to rotate them so that the rover x and y axes are parallel to the axes in the real world. (polar to rectangular) After this rotation is performed, the points area then translated

with a scale factor of a pixel (1x1m) so you need to multiply it by 10 (selected scale in project grid). This outputs integer values since pixels need this and can't be floats. Code below shows how this is done.

```
# Assume a scale factor of 10 between world space pixels and rover space pixels
scale = 10
# Perform translation and convert to integer since pixel values can't be float
x_world = np.int_(xpos + (x_rotated / scale))
y_world = np.int_(ypos + (y_rotated / scale))
```

ii. Since we have a map size of 200m x 200m, we want to clip the calculated extra pixel values that fall outside the map by implementing this code:

```
world_size = 200
x_pix_world = np.clip(x_world, 0, world_size - 1)
y_pix_world = np.clip(y_world, 0, world_size - 1)
```

iii. Implementing this with the code modules listed above, the code used in the Jupyter Notebook is what generated the rover space vs. world space maps. The output image is really zoomed in but it does show all 4 images. This can be seen in the Jupyter Notebook itself.

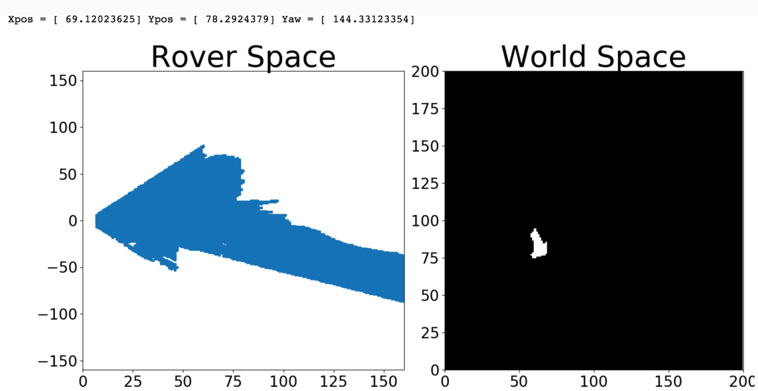
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2

# Read in the sample image
filename = 'images/image2.jpg'
image = mpimg.imread(filename)
# Rover yaw values will come as floats from 0 to 360
# Generate a random value in this range
# Note: you need to convert this to radians
    # before adding to pixel_angles
def perspect_transform(img):

    img_size = (img.shape[1], img.shape[0])
    # 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
    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
    bottom_offset = 6
    src = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
    dst = np.float32([[img_size[0]/2 - dst_size, img_size[1] - bottom_offset],
                      [img_size[0]/2 + dst_size, img_size[1] - bottom_offset],
```

```
                  [img_size[0]/2 + dst_size, img_size[1] - 2*dst_size - bottom_offset],
                  [img_size[0]/2 - dst_size, img_size[1] - 2*dst_size - bottom_offset],
                  ])

    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, img_size)# keep same size as input image
    return warped

def color_thresh(img, rgb_thresh=(170, 170, 170)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all thre 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])
    # Index the array of zeros with the boolean array and set to 1
    color_select[above_thresh] = 1
    # Return the binary image
    return color_select

def rover_coords(binary_img):
    # Identify nonzero pixels
    ypos, xpos = binary_img.nonzero()
    # Calculate pixel positions with reference to the rover position being at the
    # center bottom of the image.
    x_pixel = np.absolute(ypos - binary_img.shape[0]).astype(np.float)
    y_pixel = -(xpos - binary_img.shape[0]).astype(np.float)
    return x_pixel, y_pixel


# Rover yaw values will come as floats from 0 to 360
# Generate a random value in this range
# Note: you need to convert this to radians
    # before adding to pixel_angles
rover_yaw = np.random.random(1)*360

# Generate a random rover position in world coords
# Position values will range from 20 to 180 to
# avoid the edges in a 200 x 200 pixel world
rover_xpos = np.random.random(1)*160 + 20
rover_ypos = np.random.random(1)*160 + 20

```

```
# Note: Since we've chosen random numbers for yaw and position,
# multiple run of the code will result in different outputs each time.

# Define a function to apply a rotation to pixel positions
def rotate_pix(xpix, ypix, yaw):
    # TODO:
    # Convert yaw to radians
    # Apply a rotation
    yaw_rad = yaw * np.pi / 180
    xpix_rotated = (xpix * np.cos(yaw_rad)) - (ypix * np.sin(yaw_rad))
    ypix_rotated = (xpix * np.sin(yaw_rad)) + (ypix * np.cos(yaw_rad))

    # Return the result
    return xpix_rotated, ypix_rotated

# Define a function to perform a translation
def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # TODO:
    # Apply a scaling and a translation
    xpix_translated = (xpix_rot / scale) + xpos
    ypix_translated = (ypix_rot / scale) + ypos

    # Return the result
    return xpix_translated, ypix_translated

# 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)
    # Clip to world_size
    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

# No need to modify code below here
# Perform warping and color thresholding
warped = perspect_transform(image)
colorsel = color_thresh(warped, rgb_thresh=(160, 160, 160))
# Extract navigable terrain pixels
xpix, ypix = rover_coords(colorsel)
# Generate 200 x 200 pixel worldmap
worldmap = np.zeros((200, 200))
scale = 10
```

```
# Get navigable pixel positions in world coords
x_world, y_world = pix_to_world(xpix, ypix, rover_xpos,
                                rover_ypos, rover_yaw,
                                worldmap.shape[0], scale)
# Add pixel positions to worldmap
worldmap[y_world, x_world] += 1
print('Xpos =', rover_xpos, 'Ypos =', rover_ypos, 'Yaw =', rover_yaw)
# Plot the map in rover-centric coords

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))
f.tight_layout()
ax1.plot(xpix, ypix, '.')
ax1.set_title('Rover Space', fontsize=40)
ax1.set_ylim(-160, 160)
ax1.set_xlim(0, 160)
ax1.tick_params(labelsize=20)

ax2.imshow(worldmap, cmap='gray')
ax2.set_title('World Space', fontsize=40)
ax2.set_ylim(0, 200)
ax2.tick_params(labelsize=20)
ax2.set_xlim(0, 200)


plt.subplots_adjust(left=0.1, right=1, top=0.9, bottom=0.1)
```

f.



2. Rock Sample identification
    a. I used the find rocks function used in the walkthrough video as I wanted to get a head start to this project. Simar to the mask thresholding done above we assign a threshold logic for pixels in the yellow range of 110, 110, and 50. Another way this could have been done is to create a notch filter to filter out pixels between certain color values so we only look at yellow values. Looking up the color of the rock online from a RGB value site the color is around RGB(255, 223, 0). What I did later on is add logic code after finding these rocks to command the rover to go towards them.

```
#rock ROI detection:

    def find_rocks(img, levels=(110, 110, 50)):
        rockpix=((img[:,:,0] > levels[0]) \
            & (img[:,:,1] > levels[1]) \
            & (img[:,:,2] < levels[2]))

        color_select = np.zeros_like(img[:,:,0])
        color_select[rockpix] = 1

        return color_select

    rock_map = detect_rocks(rock_img)
    fig = plt.figure(figsize=(12,3))
    plt.subplot(121)
    plt.imshow(rock_img)
    plt.subplot(122)
    plt.imshow(rock_map, cmap = 'gray')
```

3. Process Image Modification
    a. Modified code below is from Jupyter Notebook:

```
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])

  # TODO:
  # 1) Define source and destination points for perspective transform


  # 2) Apply perspective transform
     #Applied perspective transform, used mask method found in opencv:
```

The mask technique was used to perform the perspective transform which assigned the source and destination points.

```
    # used the thresholded mask technique:
```

```
    warped,mask = perspect_transform(img, source, destination)

    # 3) Apply color threshold to identify navigable terrain/obstacles/rock samples

    #after the thresholding, assign the ignored area to the mask. Rover now has a binary map
- white area is navigable terrain
    # while the black area is obstacle.

    threshed = color_thresh(warped)
```

Obstacle map was assigned “black” by multiplying it by the mask to have the rover “ignore” that channel. Values were set to binary.

```
    obs_map = np.absolute(np.float32(threshed) -1) * mask

    # 4) Convert thresholded image pixel values to rover-centric coords
```

Made sure to set the rover coordinates to the thresholded image not the original image:

```
    xpix, ypix = rover_coords(threshed)

    # 5) Converted the  rover-centric pixel values to world coordinates:
```

Since the dst size = 5, multiplying it by 2 gives us the 10x10 m square grid

```
    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)
    obsxpix, obsypix = rover_coords(obs_map)
    obs_x_world, obs_y_world = pix_to_world(obsxpix, obsypix, xpos, ypos, yaw, world_size,
scale)

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

Maps each to a separate channel and gives it logic that it is navigable terrain base on >0 logic from defined obstacle matrix channel.

```
    data.worldmap[y_world, x_world, 2] =255
    data.worldmap[obs_y_world, obs_x_world, 0] = 255
    nav_pix = data.worldmap[:,:,2] > 0

    data.worldmap[nav_pix, 0] = 0
```

This is from the detect rocks function from the walkthrough video:

```
    rock_map = detect_rocks(warped, levels =(110,110,50))
    if rock_map.any():
        rock_x, rock_y = rover_coords(rock_map)
```

rock_x_world, rock_y_world = pix_to_world(rock_x, rock_y, xpos, ypos, yaw, world_size, scale)

data.worldmap[rock_y_world, rock_x_world, :] = 255

```
    # 7) Make a mosaic image, below is some example code
        # First create a blank image (can be whatever shape you like)
```

Define blank image to fill/populate the world map data to when the code is running so we can see where the rover goes and if it detects rocks along its path or not. Code below assigns each layer (warped image, worldmap, ground truth, flips the worldmap overlay,

```
    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
        # Here I'm putting the original image in the upper left hand corner
    output_image[0:img.shape[0], 0:img.shape[1]] = img

        # 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)
```

Adding text to image:

```
        # 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)
    data.count += 1 # Keep track of the index in the Databucket()

    return output_image
```

---

4. Perception _step() modification:
    a. Since the find rocks function was used from the walkthrough video this code was added to the perception step as well. Full code is below. Much of the code is the same here taken from the process image step in the notebook. Updated code to reference rover_image, rover_vision_image instead of “img” as it is coming from the rover itself and no longer our test image dataset.

```
# Apply the above functions in succession and update the Rover state accordingly
def perception_step(Rover):
    # Perform perception steps to update Rover()
    # TODO:
    # NOTE: camera image is coming to you in Rover.img
```

```
    # 1) Define source and destination points for perspective transform
    # 2) Apply 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
    image = Rover.img
    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, mask = perspect_transform(Rover.img, source, destination)

    # 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
    threshed = color_thresh(warped)
    obs_map = np.absolute(np.float32(threshed) - 1) * mask


    # 4) Update Rover.vision_image (this will be displayed on left side of screen)
        # Example: Rover.vision_image[:,:,0] = obstacle color-thresholded binary image
        #          Rover.vision_image[:,:,1] = rock_sample color-thresholded binary image
        #          Rover.vision_image[:,:,2] = navigable terrain color-thresholded binary image
# set threshed image to 255* in threshed channel, obstacles map *255 as well
    Rover.vision_image[:,:,2] = threshed * 255
    Rover.vision_image[:,:,0] = obs_map * 255
    # 5) Convert map image pixel values to rover-centric coords
    xpix, ypix = rover_coords(threshed)

    # 6) Convert rover-centric pixel values to world coordinates

    world_size = Rover.worldmap.shape[0]
    scale = 2 * dst_size
    x_world, y_world = pix_to_world(xpix, ypix, Rover.pos[0], Rover.pos[1],
               Rover.yaw, world_size, scale)

    obspix, obsypix = rover_coords(obs_map)
    obs_x_world, obs_y_world = pix_to_world(obspix, obsypix, Rover.pos[0], Rover.pos[1],
                     Rover.yaw, world_size, scale)

    # 7) Update Rover worldmap (to be displayed on right side of screen)
        # Example: Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1
```

```
    #        Rover.worldmap[rock_y_world, rock_x_world, 1] += 1
    #        Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 1
# favor navigable terrain to the obstacles so the rover steers away from them
# changed this to 18 as 20 was too sensitive - has the nice side effect of magnifying the rock detection as well
  Rover.worldmap[y_world, x_world, 2] += 18#20 #10
  Rover.worldmap[obs_y_world, obs_x_world, 0] +=1
  # 8) Convert rover-centric pixel positions to polar coordinates, convert from rectangular values
  dist, angles = to_polar_coords(xpix, ypix)

  # Update Rover pixel distances and angles
    # Rover.nav_dists = rover_centric_pixel_distances
    # Rover.nav_angles = rover_centric_angles
  Rover.nav_angles = angles
# used walkthrough video to implement rock mapping logic
  #assign rock map
  rock_map = find_rocks(warped, levels = (110, 110, 50))
  if rock_map.any():
    rock_x, rock_y = rover_coords(rock_map)
    rock_x_world, rock_y_world, = pix_to_world( rock_x, rock_y, Rover.pos[0], Rover.pos[1], Rover.yaw, world_size, scale)

    rock_dist, rock_ang = to_polar_coords(rock_x, rock_y)
    rock_idx = np.argmin(rock_dist)
    rock_xcen = rock_x_world[rock_idx]
    rock_ycen = rock_y_world[rock_idx]

    Rover.rock_dist, Rover.rock_angle = rock_dist, rock_ang

    Rover.worldmap[rock_ycen, rock_xcen, 1] = 255
    Rover.vision_image[:, :, 1] = rock_map * 255
  else:
    Rover.rock_dist = Rover.rock_angle = None
    Rover.vision_image[:,:,1] = 0

  return Rover
```

---

5. Decision.py

```
import numpy as np
# Ran Roversim at 1024 x 768 at the "good" graphics setting

# This is where you can build a decision tree for determining throttle, brake and steer
# commands based on the output of the perception_step() function
def decision_step(Rover):

  # Implement conditionals to decide what to do given perception data
```

```
    # Here you're all set up with some basic functionality but you'll need to
    # improve on this decision tree to do a good job of navigating autonomously!

    # Example:
    # Check if we have vision data to make decisions with
```

Added code here to have a "stop mode" when the rover detected rocks so it can slow down and pick up the rocks. Current code had rover driving by the rocks.

```
    if Rover.nav_angles is not None:
        # Check for Rover.mode status
        if Rover.mode == 'forward':
            # Check the extent of navigable terrain
            if Rover.rock_angle is not None and len(Rover.rock_angle) > 0:
```

have the rover steer to the rocks:

```
                Rover.steer = np.clip(np.mean(Rover.rock_angle * 180/np.pi), -15, 15) #-15, 15
```

Logic to lay off the throttle:

```
                if Rover.vel > Rover.max_vel/2:
                    Rover.brake = Rover.brake_set
                    Rover.throttle = 0
                elif Rover.vel > Rover.max_vel/4:
                    Rover.throttle = 0
                else:
                    Rover.throttle = Rover.throttle_set
                    Rover.brake = 0
```

Added rover near sample logic:

```
                if Rover.near_sample:
                    Rover.mode = 'pickup'
```

Logic to make the rover go again :

```
            elif len(Rover.nav_angles) >= Rover.stop_forward:
                # If mode is forward, navigable terrain looks good
                # and velocity is below max, then throttle
                if Rover.vel < Rover.max_vel:
                    # Set throttle value to throttle setting
                    Rover.throttle = Rover.throttle_set
                else: # Else coast
                    Rover.throttle = 0
                Rover.brake = 0
```

Want to see the rock angle, distance

```
            print('rock:', Rover.rock_angle, 'dist:', Rover.rock_dist)
            # make the rover steer towards rock after stopping:
            # Set steering to average angle clipped to the range +/- 15

            # Set steering to average angle clipped to the range +/- 15
            Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)#-15,15

            # Coast while turning
            if np.abs(Rover.steer) > 5 and Rover.vel > Rover.max_vel/2:
                Rover.throttle = 0
        # If there's a lack of navigable terrain pixels then go to 'stop' mode
        elif len(Rover.nav_angles) < Rover.stop_forward:
                # Set mode to "stop" and hit the brakes!

Lay off the throttle:
                Rover.throttle = 0
                # Set brake to stored brake value
                Rover.brake = Rover.brake_set
                Rover.steer = 0
                Rover.mode = 'stop'

    # If we're already in "stop" mode then make different decisions
    # added logic to pickup rocks
    elif Rover.mode == 'stop':
        # If we're in stop mode but still moving keep braking
        if Rover.vel > 0.2:
            Rover.throttle = 0
            Rover.brake = Rover.brake_set
            Rover.steer = 0
        # If we're not moving (vel < 0.2) then do something else
        elif Rover.vel <= 0.2:
            # Now we're stopped and we have vision data to see if there's a path forward.


            if len(Rover.nav_angles) < Rover.go_forward:
                Rover.throttle = 0
                # Release the brake to allow turning
                Rover.brake = 0
                # Turn range is +/- 15 degrees, when stopped the next line will induce 4-wheel
turning

turn left until it sees navigable terrain again:
                Rover.steer = -15 # -15Could be more clever here about which way to turn
            # If we're stopped but see sufficient navigable terrain in front then go!
            if len(Rover.nav_angles) >= Rover.go_forward:
                # Set throttle back to stored value
                Rover.throttle = Rover.throttle_set
                # Release the brake
```

```
            Rover.brake = 0
```

I didn't modify the code here to make the rover a "crawler" as I got good results after making the rover turn towards and pickup the rock. I added code so the rock would get out of its stuck position but it is rudimentary.

```
# Set steer to mean angle
            Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)#-15,15
            Rover.mode = 'forward'
    # sees rock, hits brake to slow rover down and stop to pick up rocks
    elif Rover.mode == 'pickup':
      Rover.brake = Rover.brake_set

      if Rover.vel < 0.2 and not Rover.picking_up:
        if Rover.near_sample:
          Rover.send_pickup = True
        else:
Set the mode to "stop"
            Rover.steer = 0
            Rover.mode = 'stop'

  # Just to make the rover do something
  # even if no modifications have been made to the code
  else:
    Rover.throttle = Rover.throttle_set
    Rover.steer = 0
    Rover.brake = 0

  return Rover
```