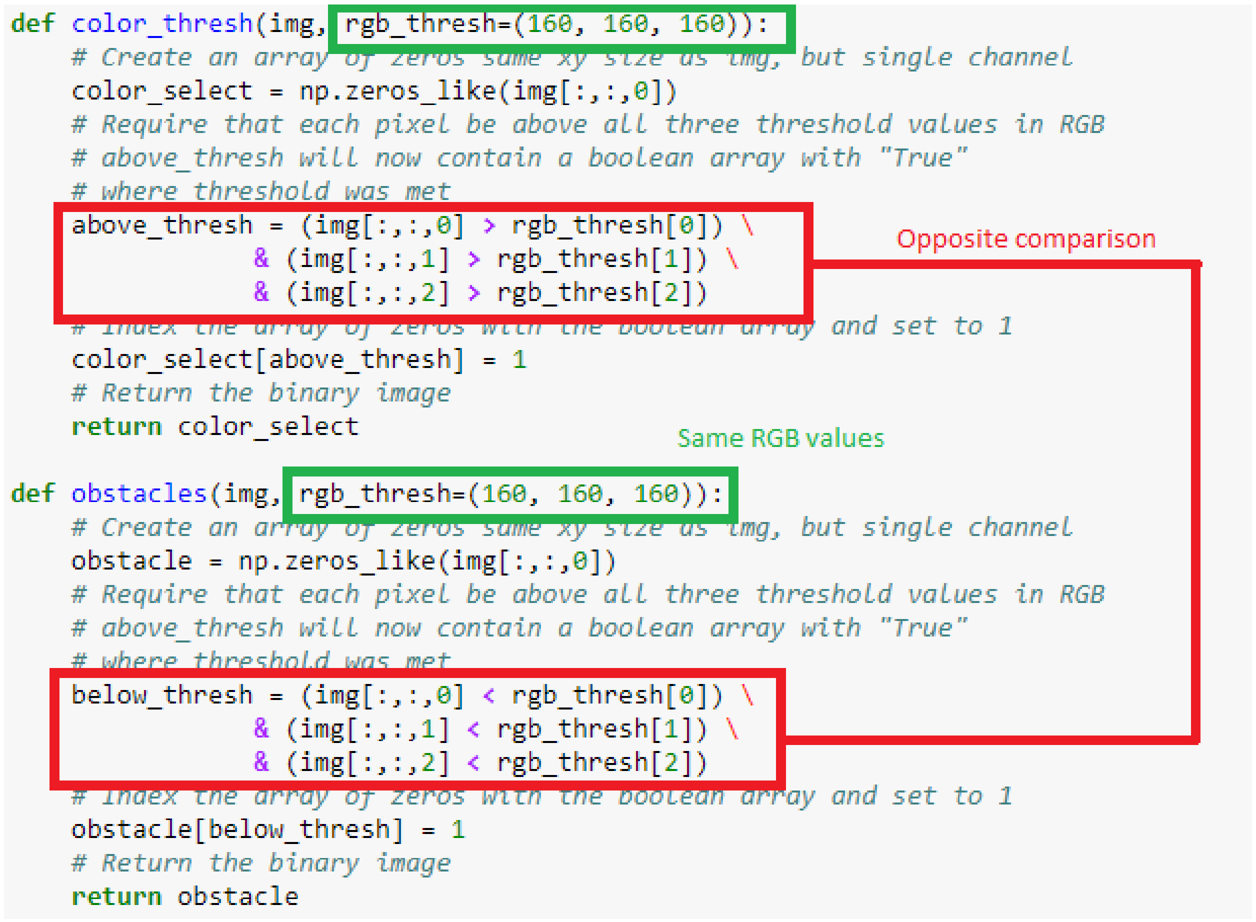
Training/Calibration

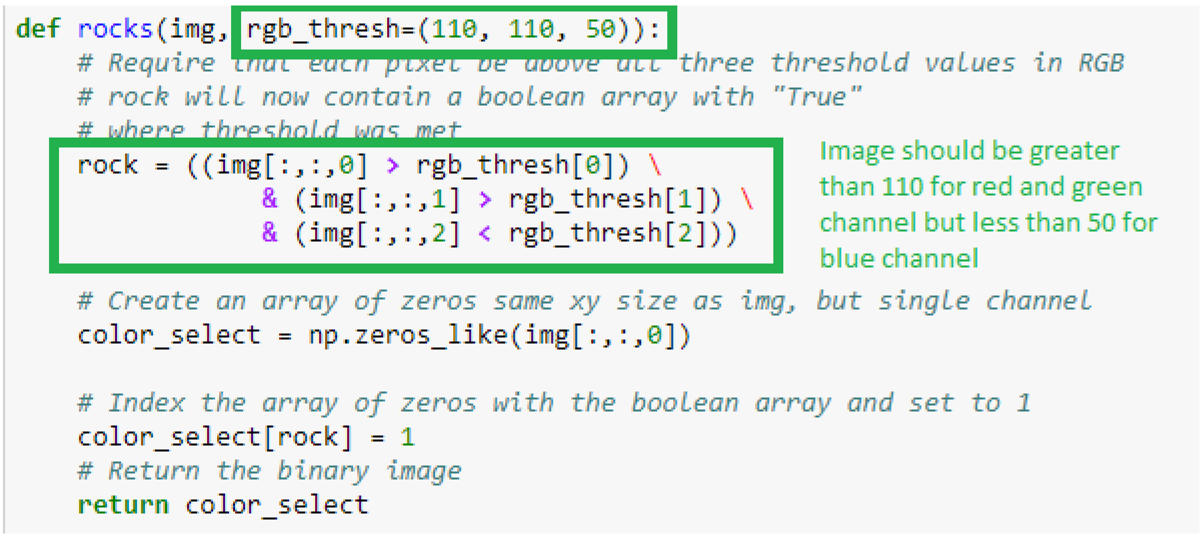
Jupyter Notebook:

My Jupyter notebook is contained within the zip file. The first change I made was to add a function to find rocks and another function to find obstacles. The obstacles() function was simply implementing the inverse of the color_thresh function. By sending in the same RGB values of 160, 160, 160 to the obstacles but then reversing the threshold check, I could identify obstacles. This is assuming that anything that is not navigable terrain is an obstacle.



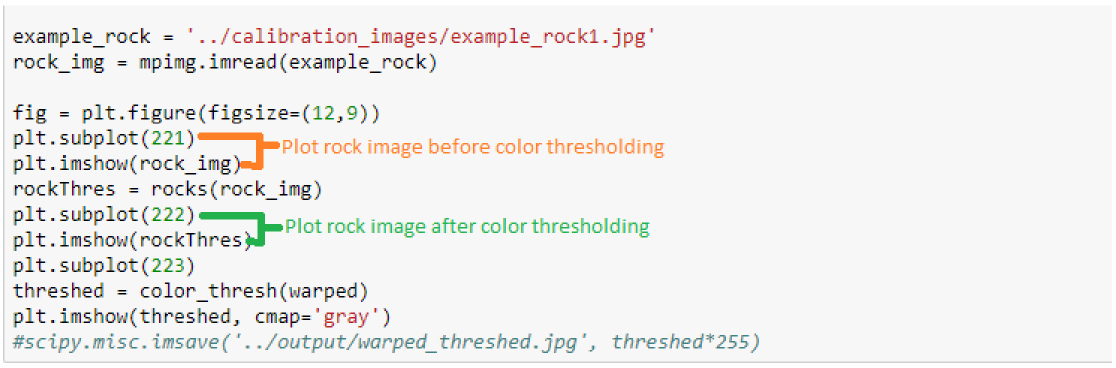
```
def color_thresh(img, rgb_thresh=(160, 160, 160)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "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

def obstacles(img, rgb_thresh=(160, 160, 160)):
    # Create an array of zeros same xy size as img, but single channel
    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
    below_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
    obstacle[below_thresh] = 1
    # Return the binary image
    return obstacle
```

To identify rocks, I created a rocks() function. I selected an image from the test data that contained a clear view of a yellow rock. I then used an online tool to determine an appropriate RGB value to detect the rock. In the notebook, I confirmed that the values I selected worked by plotting the output returned from the rocks() function.



```python
def rocks(img, rgb_thresh=(110, 110, 50)):
    # Require that each pixel be above all three threshold values in RGB
    # rock will now contain a boolean array with "True"
    # where threshold was met
    rock = ((img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,1] > rgb_thresh[1]) \
                & (img[:,:,2] < rgb_thresh[2]))
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])

    # Index the array of zeros with the boolean array and set to 1
    color_select[rock] = 1
    # Return the binary image
    return color_select
```

Image should be greater than 110 for red and green channel but less than 50 for blue channel

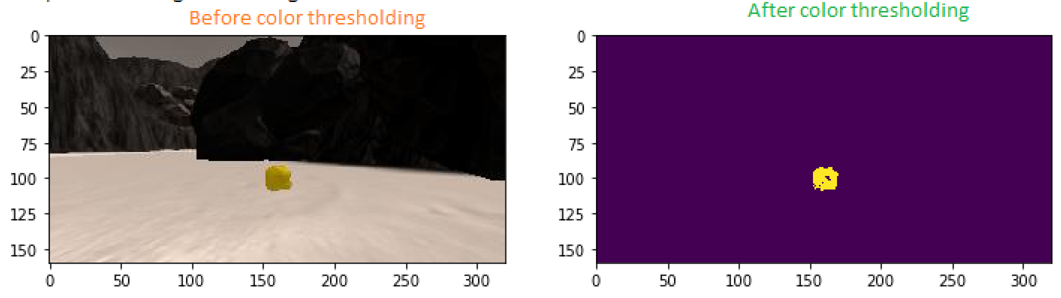


```python
example_rock = '../calibration_images/example_rock1.jpg'
rock_img = mpimg.imread(example_rock)

fig = plt.figure(figsize=(12,9))
plt.subplot(221)
plt.imshow(rock_img)
rockThres = rocks(rock_img)
plt.subplot(222)
plt.imshow(rockThres)
plt.subplot(223)
threshed = color_thresh(warped)
plt.imshow(threshed, cmap='gray')
#scipy.misc.imsave('../output/warped_threshed.jpg', threshed*255)
```

Plot rock image before color thresholding

Plot rock image after color thresholding

```
Out[6]: <matplotlib.image.AxesImage at 0x171566b5a90>
```



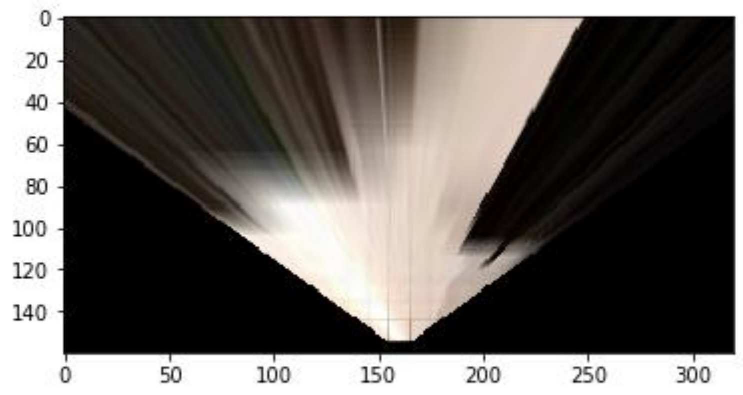
The next piece of code that was modified was the process_image() function. First, I added in code to determine the source and destination points to run the perspective transform function. The goal is to create a top down view (birds eye view) of the landscape rather than using the view from the rover's perspective. We find 4 points that define a square from the rover's perspective. These points are [[14, 140], [301 ,140], [200, 96], [118, 96]], and can be found by using the sample data. Next, we determine the desired coordinates for the top down view.

```
    # 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
    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
    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],
                  ])
```

After defining the source and destination, I send these values into the perspect_transform() function along with the image seen by the Rover. The function perspect_transform() will perform the perspective transformation, which produces the top down view of the landscape. This is known as the warped view. The warped view shows us the navigable terrain ahead of the Rover (with help of color thresholding functions), and it will allow us to take an average value of this terrain and move along that average path.

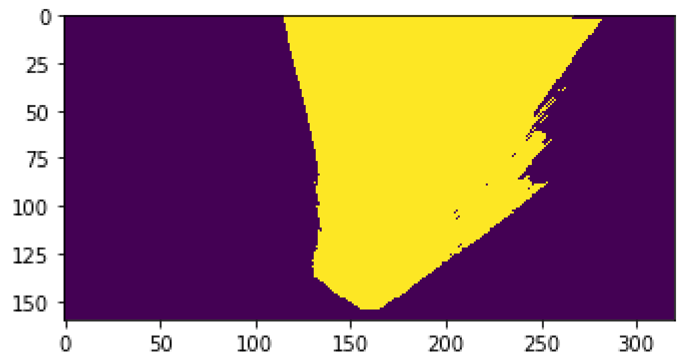
```
# 2) Apply perspective transform
warped = perspect_transform(img, source, destination)
```

```
# Define a function to perform a perspective transform
# I've used the example grid image above to choose source points for the
# grid cell in front of the rover (each grid cell is 1 square meter in the sim)
# Define a function to perform a perspective transform
def perspect_transform(img, src, dst):

    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))# keep same size as input image

    return warped
```



After returning a warped image, I call the color thresholding functions for identify navigable terrain, obstacles, and yellow rocks.

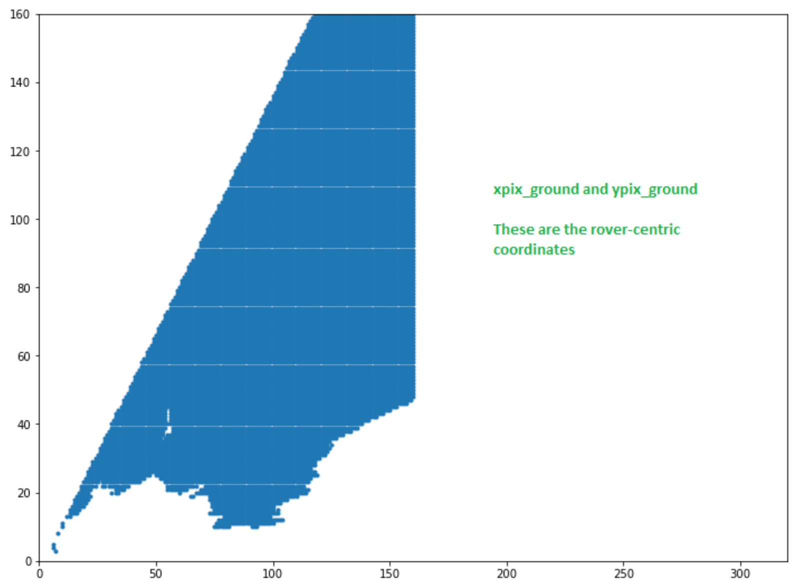
```
# 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
groundSel = color_thresh(warped)
obstacleSel = obstacles(warped)
rocksSel = rocks(warped)
```



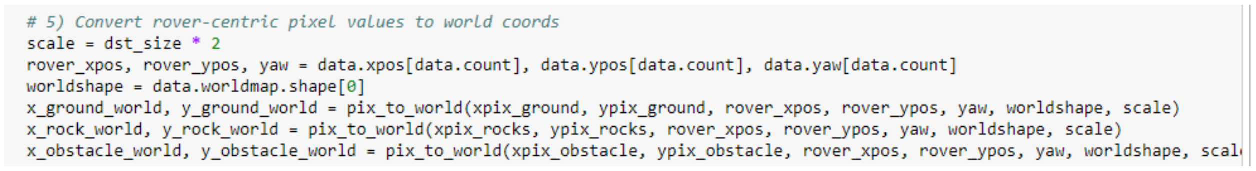
→ Output from groundSel

Next, I convert the terrain, obstacles, and rocks image pixel values to rover-centric coordinates. This is required because we need to have a fixed coordinate system with respect to the Rover. We need to be able to describe the environment (like obstacles, rocks, etc) with respect to the Rover. So, with this step, the Rover (rather its camera) will be at position (x, y) = (0, 0).

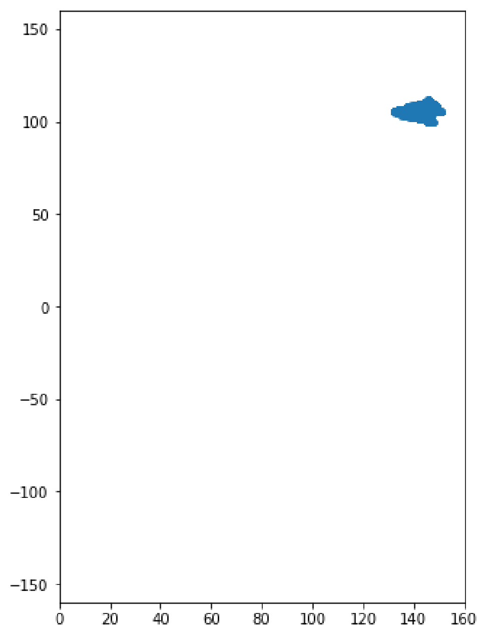
```
# 4) Convert thresholded image pixel values to rover-centric coords
xpix_ground, ypix_ground = rover_coords(groundSel)
xpix_obstacle, ypix_obstacle = rover_coords(obstacleSel)
xpix_rocks, ypix_rocks = rover_coords(rocksSel)
```



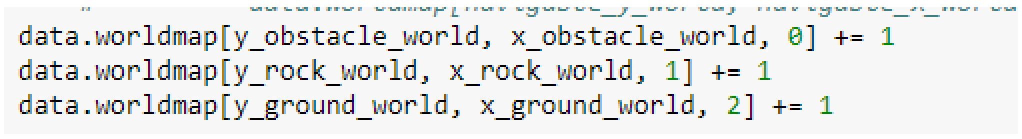
Once I have the rover coordinates, I need to position the Rover correctly in the world. This is done by calling pix_to_world(). The Rover object (in the notebook, it's the data object from the DataBucket class) provides me with the rovers x and y position, I have the rover-centric coordinates, I have the yaw value from the Rover object, and I have the shape of the world from the Rover object. With this, I'm able to determine the world coordinates of navigable terrain, rocks, and obstacles



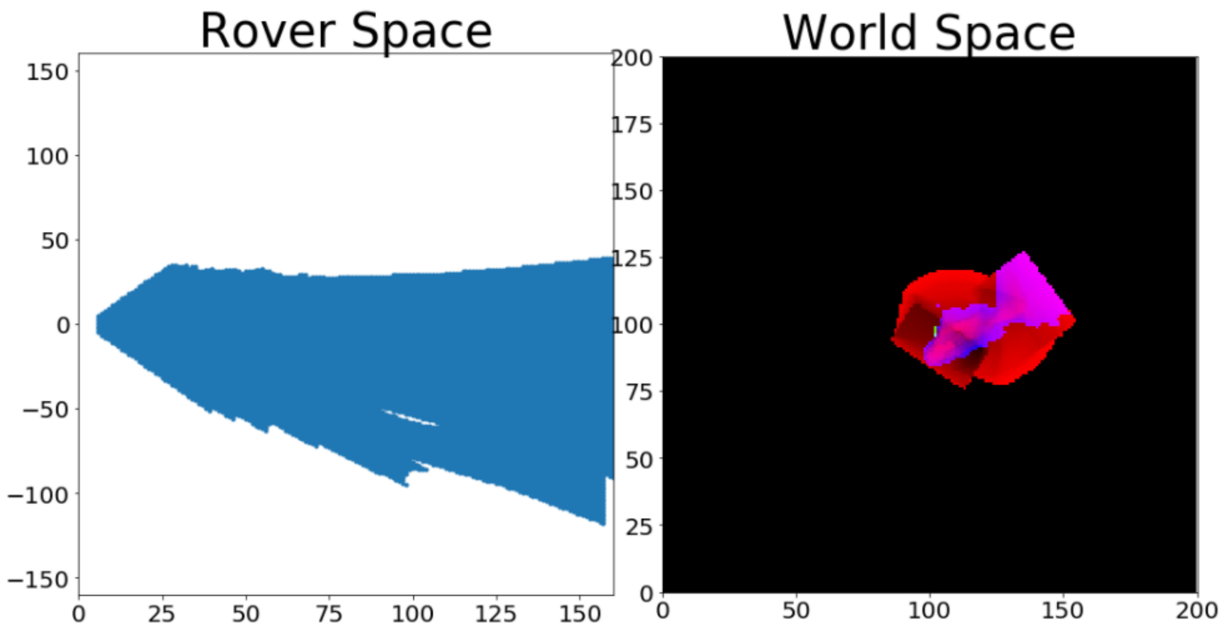
```
# 5) Convert rover-centric pixel values to world coords
scale = dst_size * 2
rover_xpos, rover_ypos, yaw = data.xpos[data.count], data.ypos[data.count], data.yaw[data.count]
worldshape = data.worldmap.shape[0]
x_ground_world, y_ground_world = pix_to_world(xpix_ground, ypix_ground, rover_xpos, rover_ypos, yaw, worldshape, scale)
x_rock_world, y_rock_world = pix_to_world(xpix_rocks, ypix_rocks, rover_xpos, rover_ypos, yaw, worldshape, scale)
x_obstacle_world, y_obstacle_world = pix_to_world(xpix_obstacle, ypix_obstacle, rover_xpos, rover_ypos, yaw, worldshape, scal
```



Lastly, I added code to the process_image() function which updates the world map with the world coordinates of the rocks, navigable terrain, and obstacles. This allows me to see the rocks, terrain, and obstacles overlaid on the world map. With this, when running the Rover, I can mark where the terrain is, where the rocks are, and where the obstacles are.



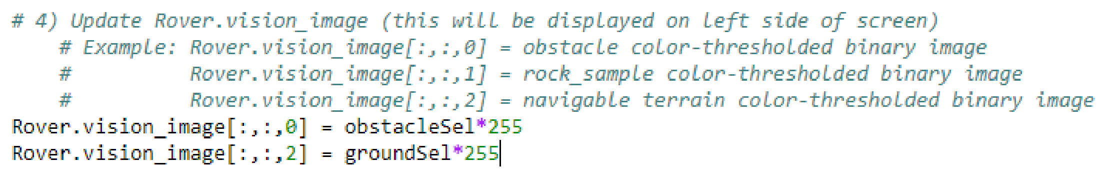
```
#           data.worldmap[navigable_y_world, navigable_x_world
data.worldmap[y_obstacle_world, x_obstacle_world, 0] += 1
data.worldmap[y_rock_world, x_rock_world, 1] += 1
data.worldmap[y_ground_world, x_ground_world, 2] += 1
```



Autonomous Navigation and Mapping

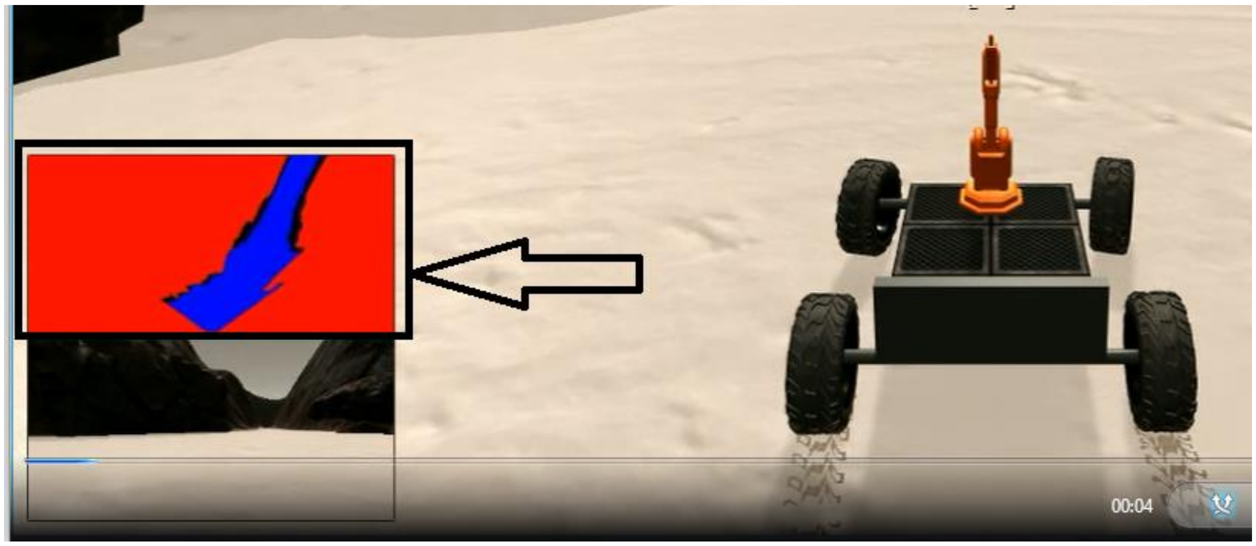
Code From perception.py:

The functions color_thresh(), obstacles(), and rocks() are copied as is from the Jupyter notebook. All code from process_image() was copied to the perception_step(), however, there were a few modification made. First, since the code is now taking in the Rover object instead of an image, I updated all references from "img" to "Rover.img". All information regarding the Rover is now obtained from the Rover object. Next, I added in the following lines:



```
# 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
Rover.vision_image[:,:,0] = obstacleSel*255
Rover.vision_image[:,:,2] = groundSel*255
```

Here, we need to update the image that will be displayed on the left side of the screen when running the rover. It displays the warped and color threshold version of the image. The same is also done further down in the code for the rocks image. The images are multiplied by 255 so the colors are fully visible. The obstacle image is shown in the red channel, the navigable terrain is shown in the blue channel, and the rocks are shown in the green channel.



The next difference between the notebook and the perception.py code is the following two lines

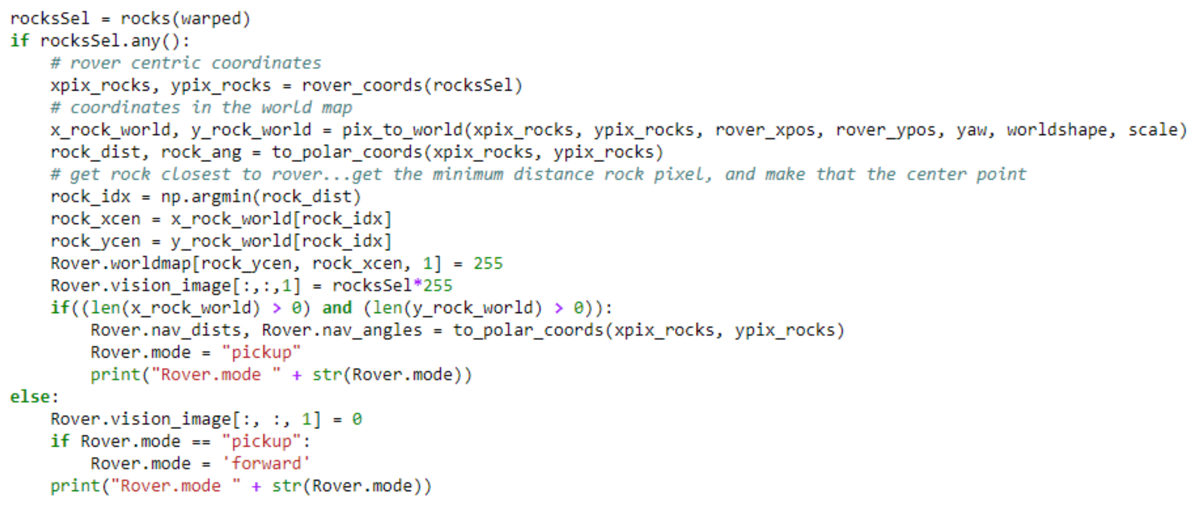
```
# each time we find an obstacle we add 1 to the red channel, if we find a lot of obstacles in one pixel, you deem it an obstacle
Rover.worldmap[y_obstacle_world, x_obstacle_world, 0] += 1
# each time we find a pixel of navigable terrain we add 10 to the blue channel, so we favor anywhere we find navigable terrain
Rover.worldmap[y_ground_world, x_ground_world, 2] += 10
```

Here, we first set the worldmap within the Rover object, not within the data object as was done in the notebook. Next, for the navigable terrain, instead of incrementing by only 1, we increment by 10. This is so that we favor anywhere we find navigable terrain over obstacles. Again, we add the obstacles to the red channel and the navigable terrain to the blue channel.

Next, we add code to convert the rover-centric pixel positions to polar coordinates, and assign each to the Rover objects nav_angles and nav_dists variables. These are required to tell the rover which angle to drive at and the distances of navigable terrain pixels.

```
dist, angles = to_polar_coords(xpix_ground, ypix_ground)
Rover.nav_angles = angles
Rover.nav_dists = dist
```

Lastly, we repeat the steps done for navigable terrain and obstacles for the rocks. First, we check that we see any rocks in the rover's view, if so, we get the rover-centric coordinates, we map it to the world coordinates, and then we get the closest rock and add that to the world map and the vision image. The other piece of code I added was to change the Rover.mode to a "pickup" mode if we see a rock. If we see a rock, I also update the Rover.nav_dists and Rov.nav_angles to be set to the coordinates of the rocks location. This way, the rover will drive towards a rock when it sees one.



```
rocksSel = rocks(warped)
if rocksSel.any():
    # rover centric coordinates
    xpix_rocks, ypix_rocks = rover_coords(rocksSel)
    # coordinates in the world map
    x_rock_world, y_rock_world = pix_to_world(xpix_rocks, ypix_rocks, rover_xpos, rover_ypos, yaw, worldshape, scale)
    rock_dist, rock_ang = to_polar_coords(xpix_rocks, ypix_rocks)
    # get rock closest to rover...get the minimum distance rock pixel, and make that the center point
    rock_idx = np.argmin(rock_dist)
    rock_xcen = x_rock_world[rock_idx]
    rock_ycen = y_rock_world[rock_idx]
    Rover.worldmap[rock_ycen, rock_xcen, 1] = 255
    Rover.vision_image[:,:,1] = rocksSel*255
    if((len(x_rock_world) > 0) and (len(y_rock_world) > 0)):
        Rover.nav_dists, Rover.nav_angles = to_polar_coords(xpix_rocks, ypix_rocks)
        Rover.mode = "pickup"
        print("Rover.mode " + str(Rover.mode))
else:
    Rover.vision_image[:, :, 1] = 0
    if Rover.mode == "pickup":
        Rover.mode = 'forward'
    print("Rover.mode " + str(Rover.mode))
```

Code From decision.py:

The only code I added to the decision.py file was an extra conditional checking for a “pickup” mode. If the rover has seen a rock, it enters the pickup mode. Once in this mode, the code will enter the

“elif Rover.mode == ‘pickup’”. The first thing I do is update the steering. Then I have 3 checks:

1. If the Rover is near the sample and the Rover is stopped and it’s not already picking up a sample, then let’s pickup the sample
2. If the Rover is near the sample but the Rover has not stopped yet (vel != 0) then we need to first stop the Rover before picking up the sample
3. If the Rover is near the sample and we are finished picking up the sample, the enter “stop” mode so that the code will exit the “pickup” condition.

I use a “flag” variable to indicate when the Rover is picking or not picking up a sample. This code has allowed the Rover to successfully pick up rock samples when it sees them.

Autonomous Run:

My results were good, the fidelity stayed relatively high throughout the run. My rover could pick up samples as well. I had to use manual mode two or three times quickly to adjust or get unstuck. The results are explained further below in the improvements.

Improvements:

1. Slow down when approaching a rock sample. The first pick up in the video was successful, however, the next pick up required manual mode because the rover overshot the sample. The rover needs to slowly approach the sample rather than stopping abruptly.
2. Rover would sometimes get stuck along a wall (see minute 3:23). The rover would have a bit of navigable terrain in its vision and would try to drive. However, it would be stuck along the wall. Since it saw navigable terrain, it continued to drive. A potential simple solution would be to

check after 10 seconds if the rover was still in the same spot. If so, turn the wheels slightly, and try again. Keep turning wheels until no longer stuck.

3. Another improvement would be to better search the map. There are some spots in the video that the rover misses because there was always more navigable terrain in another direction. This would require a better algorithm in choosing which direction to navigate.
4. Remember the locations traveled so the rover doesn't travel along the same path twice. If you resolve improvement #3 above and add this feature of remembering where the rover has traveled, then you should have the terrain close to 100% mapped.
5. The open space terrain seen at minute 2:30 would sometimes cause issues for the rover. The rover would start to go left but would then see sufficient terrain to the right, and would turn right. It would complete a circle and try to go left, but would then go to the right again. The rover would end up moving in a large circle because it was always pulled to the right. Again, if you improve #3 and #4, this would not happen. I had to use manual mode for a second to stop this from happening.
6. Of course, improve fidelity 😊