# Search and Sample Return

## Notebook Analysis

The notebook provided a quick way to learn the functions or ideas of the rover. Along with initializations and sample images, it showed how to change the perspective from front view to overhead view.

This was done by calling the opencv's function getPerspectiveTransform to obtain the 3x3 matrix which defines the transform, and then calling opencv's function warpPerspective which does the actual transform.

## color_thresh (Object Classification)

The next section was for classifying the type of objects in the view. This was simplified in that the traverseable ground was  more or less one colour the non-traverseable was another colour and rocks to be collected where a third colour.

I implemented a rock_thresh function which returned a 2 dimensional array in a similar way to the example for color_thresh. It was suggested that a single thresh function be implemented  with minimum and maximum colour detection. I did this in the next section but for this section I kept it easy and just modified the values in the rock_thresh function to detecte the appropriate ranges for rocks.

I also modified the display code in order to do testing and see that rocks were actually being detected.

## process_image

The process image source and destination perspective values used where the same as in the code above which tested out the perspective changes. As these were variables were already available I just reused them.  I saw in example code that a mask was created to mask out areas that aren't seen, I didn't use this method as it seemed superfluous. Instead I just used the color_thresh and rock_thresh functions to select objects that were appropriate and these automatically have unwanted areas masked out. This also allowed me to condense the code to calculate and colour the worldmap for traverseable areas.

```
xpix, ypix = rover_coords(perspect_transform(color_thresh(img),source,destination))

worldx,worldy = pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale)

data.worldmap[worldx,worldy,2] = 255 #make blue
```

Similar idea was used for the rocks colouring.

```
#find rocks
 rockX,rockY = rover_coords(perspect_transform(rock_thresh(img),source,destination))
rockWorldX,rockWorldY = pix_to_world(rockX,rockY,xpos,ypos,yaw,world_size,scale)
data.worldmap[rockWorldX,rockWorldY,:] = 255  #make white
```

# Problems and Solutions

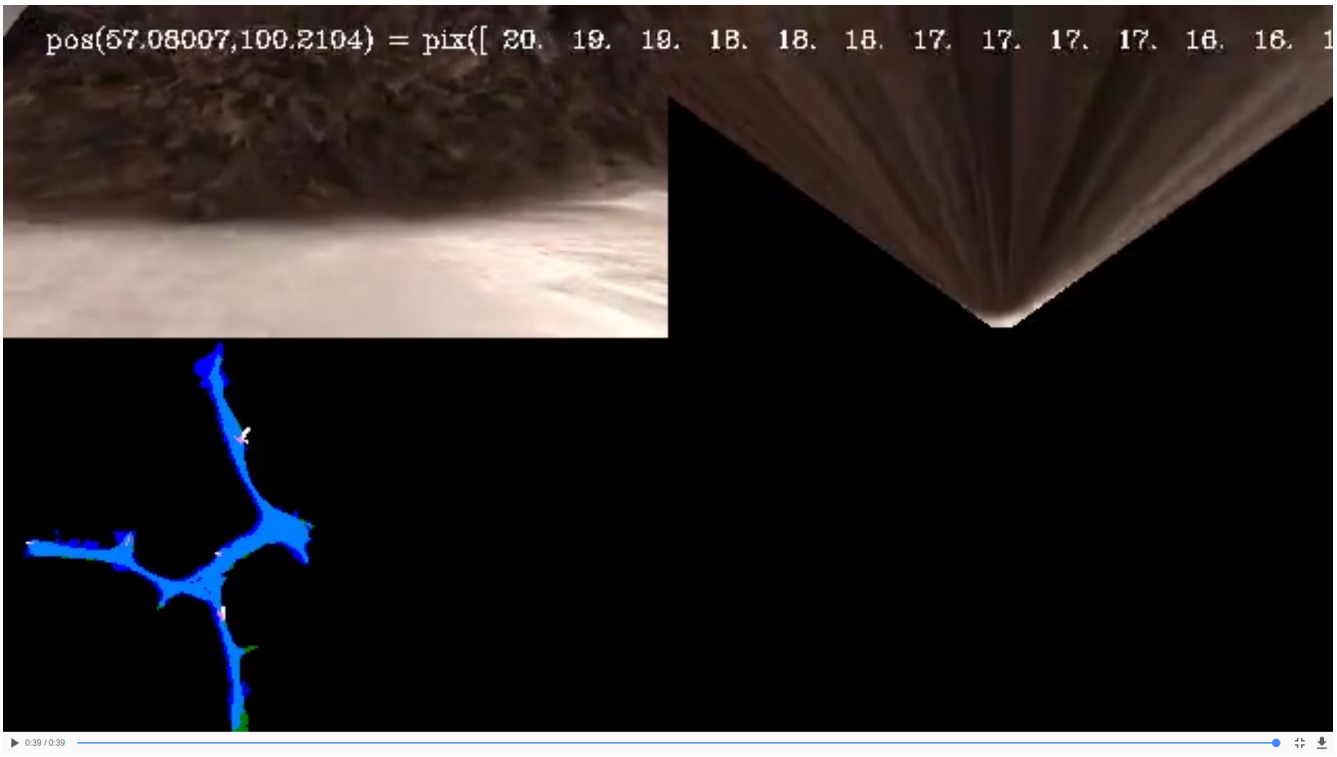
I found that the worldmap did not map onto the data.ground_truth, and after much experimenting realized that it was rotated 90 deg, and flipped vertically, this was corrected with the following code.

```
angle = -90
mapCenter = tuple(np.array(data.worldmap.shape[1::-1]) / 2)
rot_mat = cv2.getRotationMatrix2D(mapCenter, angle, 1.0)
rotatedWorldMap = cv2.warpAffine(data.worldmap, rot_mat, data.worldmap.shape[1::-1],
    flags=cv2.INTER_LINEAR)
 map_add = cv2.addWeighted(np.fliplr(rotatedWorldMap), 1, data.ground_truth, 0.5, 0)
```

The text output is just some positional locations and xpix values.

# Final Result

Using test data that traversed as much as the map as possible, the final frame of the output video is below.



# Autonomous Navigation and Mapping

For this module, I was to program a Mars rover to explore terrain and pick up any samples it found. I did not implement returning to the starting position. I'm happy to say that around 2000 seconds, 84% of the of the terrain was mapped 5 sample rocks located and 5 sample rocks picked up. The last rock must have been located inside a boulder.

Screen resolution 1440x900 fantastic graphics quality.

## perception_step()

### New functions

For this module I created a few new functions. The first is a color_thresh that accepted a min and max defined as follows:

def color_thresh(img, min=(160, 160, 160),max=(256,256,256)):

With this function I could specify a colour range for all three channels.

I then created functions for navThresh, rockThresh and unNavThresh which would supply the min and max values for the thresholds. This cleaned up the code a bit as they only needed to be passed the image instead of the threshold values as well.

### perception_step()

The source and destination perspectives are calculated to be used when the perspective transform is done.

```
    dst_size = 5
    bottom_offset = 6
    xpos, ypos = Rover.pos
    world_size = Rover.vision_image.shape[0]
    scale = dst_size * 4
    source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
    destination = np.float32([[Rover.img.shape[1]/2 - dst_size, Rover.img.shape[0] - 
bottom_offset],
                  [Rover.img.shape[1]/2 + dst_size, Rover.img.shape[0] - bottom_offset],
```

```
                  [Rover.img.shape[1]/2 + dst_size, Rover.img.shape[0] - 2*dst_size -
bottom_offset],
                  [Rover.img.shape[1]/2 - dst_size, Rover.img.shape[0] - 2*dst_size -
bottom_offset],
                  ])
```

The navigable selection is selected with the navThresh function (see above) and that is then transformed with the perspect_tranform. This way no mask is needed as the threshold already provides this.  The same is done for the nonNavigable and rocks views.

```
navigable = navThresh(Rover.img)
navigable = perspect_transform(navigable,source,destination);


nonNavigable = unNavThresh(Rover.img)
nonNavigable = perspect_transform(nonNavigable,source,destination);


rocks = rockThresh(Rover.img)
rocks = perspect_transform(rocks,source,destination)
```

The Rover.vision_image is updated with the previous output. I found it had to be multiplied by 255 so that the floating point values would show up in the bytes.

```
Rover.vision_image[:,:,0] = nonNavigable * 255;   # red
Rover.vision_image[:,:,1] = rocks * 255;          # green
Rover.vision_image[:,:,2] = navigable* 255;      # blue
```

New variables created to hold the views from the rover's perpective

```
xpix,ypix = Rover_coords(navigable)
xpixBad,ypixBad = Rover_coords(nonNavigable)
rockX,rockY = Rover_coords(rocks)
```

And these were converted to world coordinates

```
worldx,worldy = pix_to_world(xpix, ypix, xpos,ypos, Rover.yaw, world_size, scale)
worldxBad,worldyBad = pix_to_world(xpixBad, ypixBad, xpos,ypos, Rover.yaw, world_size,
scale)
rockWorldX,rockWorldY = pix_to_world(rockX,rockY,xpos,ypos,Rover.yaw,world_size,scale)
```

I only updated the worldmap if the Rover was within 1/2 a degree of level

```
if (Rover.roll < 0.5 or Rover.roll > 359.5) or (Rover.pitch < 0.5 or
        Rover.pitch > 359.5):
    Rover.worldmap[worldyBad.astype('int64'),worldxBad.astype('int64'),0] += 1 # not
navigable
    Rover.worldmap[rockWorldY.astype('int64'),rockWorldX.astype('int64'),1] += 1 # rocks
be here
```

```
        Rover.worldmap[worldy.astype('int64'),worldx.astype('int64'),2] += 1 #navigable
```

I then saved the polar equivalence for the rover's perspective for the naviable area and any rocks to be picked up.

```
    Rover.nav_dists,Rover.nav_angles = to_polar_coords(xpix,ypix)
    Rover.rock_dists,Rover.rock_angles = to_polar_coords(rockX,rockY)


    return Rover
```

## decision_step()

I ended up splitting the decision_step into a number of smaller routines to handle each of the rover functions. This was useful because there were duplicates between states, for example moving towards a rock or moving along a wall both used the move_toward routine.

### move_toward

The move_toward routine moves the rover towards an a point defined by the distance and angle arrays passed to it. Also passed in are maxspeed and followWall.  Wall following is wanted when just mapping out the scene. By following one wall you will eventually cover the whole maze. Wall following is not wanted when going to get a stone. Wall following was accomplished by adding an offset of to the angle it was moving.  The offset was determined in one of 4 ways.

1. 50% chance of 8 degrees

2. 25% chance of -8 degrees

3. 12.5% chance of 0.8 * np.std(Rover.nav_angles)

4. 12.5% chance of -0.8 * np.std(Rover.nav_angles)

So it was most likely to stay to the left, but would occasionally go to the right. The .8 np.std function was seen on another site, so I decided to try it, I couldn't say if it made a difference.

Steering was also modified to not be as jerky by only steering hard (at most +-15 degrees) if the distance is less then 30. If the distance was more then 30, then the steering was only incremented or decremented. If the distance is less then the current velocity then the mode is set to **stop**

### Stop

The stop routine sets the break or sets mode to **stopped**

### Turn

The turn routine will do hard terns (at most +-15 degrees) or if the Rover is current at the angle will set the mode to **forward**

isStuck

This routine attempts to determine if the rover is stuck and if so, how to get unstuck. If the rover is getting a rock the routine is ignored, unless it has been called 20 times. If the rover is turning the routine is ignored, unless it has been called 20 times. Otherwise the average speed is calculated to determine the rounded speed is 0 (< 0.5). If it is, then we are stopped and assumed to be stuck. The mode will be set to turning to the current direction + the unstuck angle, which is normally 180 degrees. The unstuck angle changes if it is stuck multiple times. I also decide which way to turn by using a random number.

### decision_step (after the modes were broken out)

The decision_step is broken into two parts, the rock gathering part and the navigating part. I was able to succeed in both of these, with the exception of rocks hidden behind boulders.

When a rock sample is noticed, the rover moves towards the rock until is is less then 10 away, otherwise the rover follows the most open area it can find, following along the wall.

## Final Thoughts

This has been a fun project that I have spent way too much time on :) The rover had a hard time distinguishing between when it was stuck and when it had just finished picking up a rock sample. Picking up a rock sample seemed similar to a child getting distracted by a pretty stone, and then forgetting what he was supposed to do. The rover would frequently be turned around after picking up a rock sample. Getting unstuck was the hardest thing to implement, because there are so many possible ways to get stuck. I'm sure this could be improved on. Another possible idea to get a higher mapping score is to watch for wide open spaces and when the rover is in an area with a lot of space on it, do a pirouette to scan the area.