Project: Search and Sample Return

The goals / steps of this project are the following:

Training / Calibration:

a. Download the simulator and take data in "Training Mode"
b. Test out the functions in the Jupyter Notebook provided
c. Add functions to detect obstacles and samples of interest (golden rocks)
d. Fill in the `process_image()` function with the appropriate image processing steps (perspective transform, color threshold etc.) to get from raw images to a map. The `output_image` you create in this step should demonstrate that your mapping pipeline works.
e. Use `moviepy` to process the images in your saved dataset with the `process_image()` function. Include the video you produce as part of your submission.

Autonomous Navigation / Mapping

a. Fill in the `perception_step()` function within the `perception.py` script with the appropriate image processing functions to create a map and update `Rover()` data (similar to what you did with `process_image()` in the notebook).
b. Fill in the `decision_step()` function within the `decision.py` script with conditional statements that take into consideration the outputs of the `perception_step()` in deciding how to issue throttle, brake and steering commands.
c. Iterate on your perception and decision function until your rover does a reasonable (need to define metric) job of navigating and mapping.

Image References

[image1]: ./misc/rover_image.jpg
[image2]: ./calibration_images/example_grid1.jpg
[image3]: ./calibration_images/example_rock1.jpg

[Rubric](https://review.udacity.com/#!/rubrics/916/view) Points
Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

You're reading it!

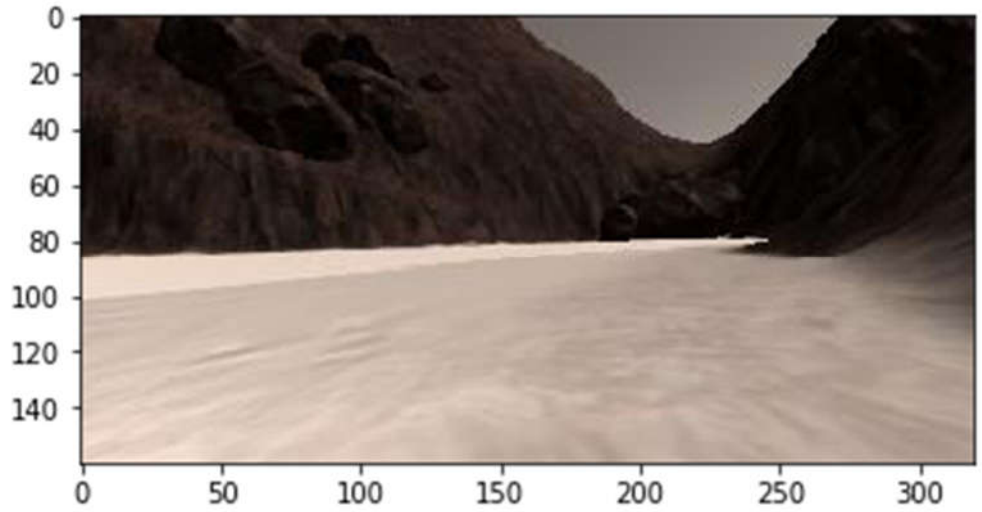
Notebook Analysis

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.

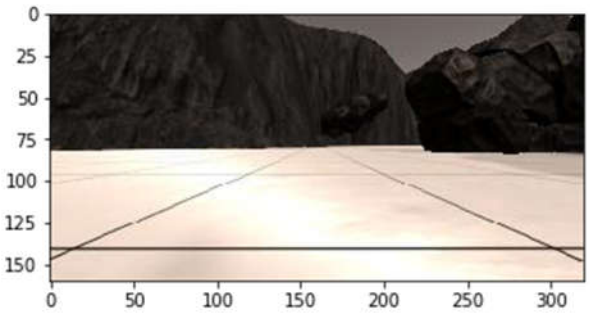
Here is an example of how to include an image in your writeup.

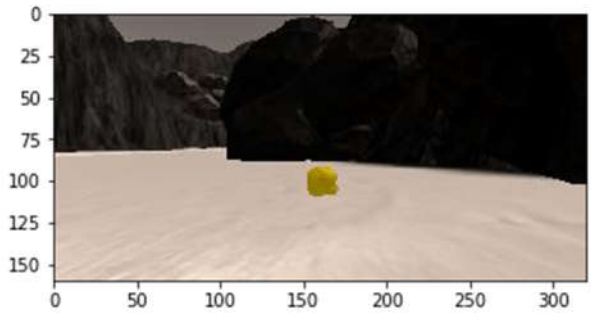
With Test Images:

Quick look at the data

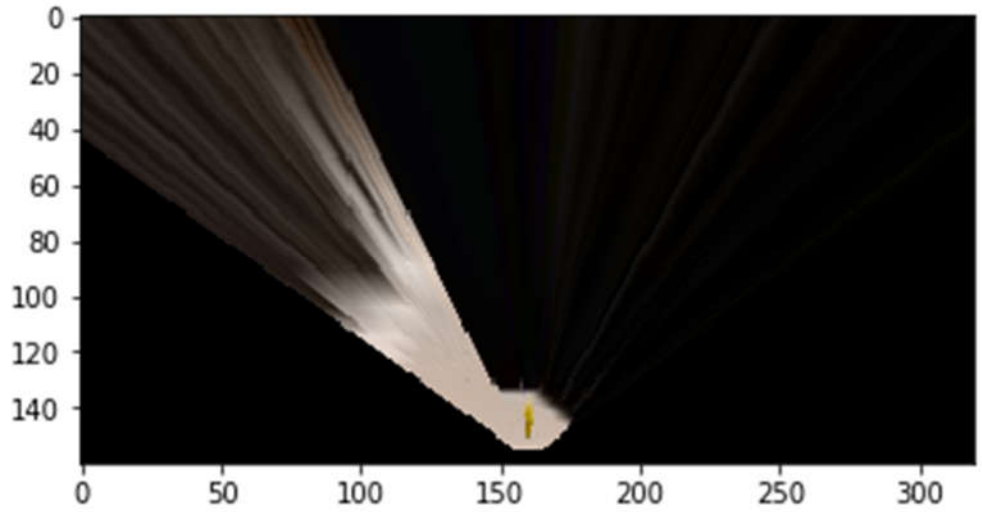


Calibration Data:

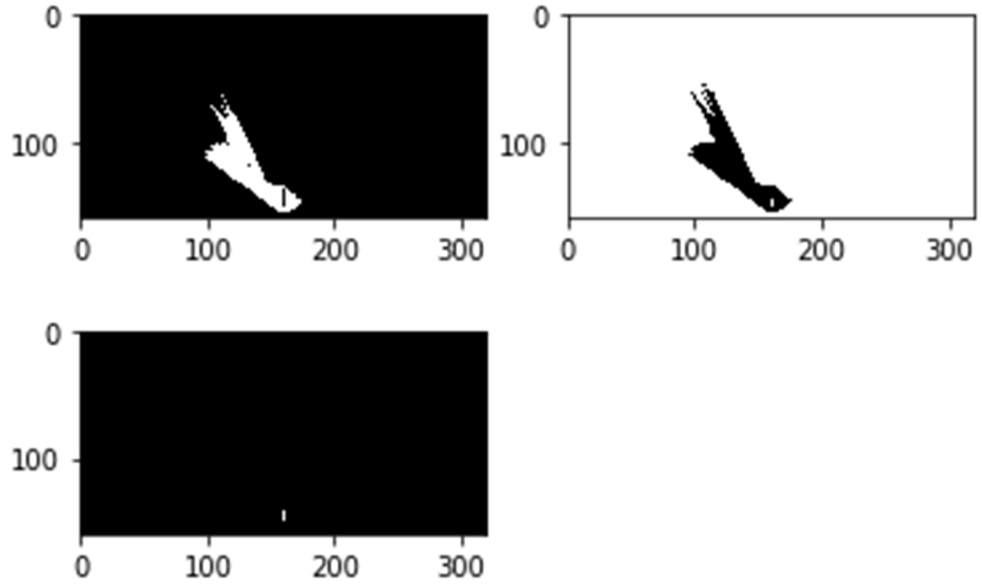




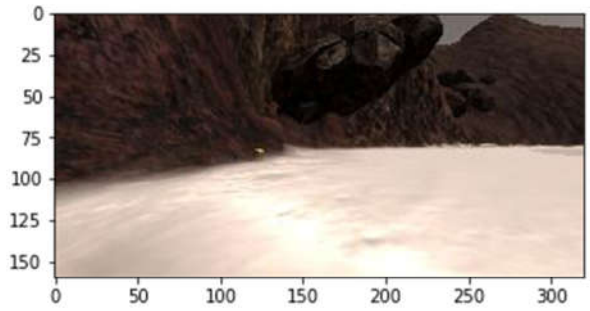
Perspective transform:

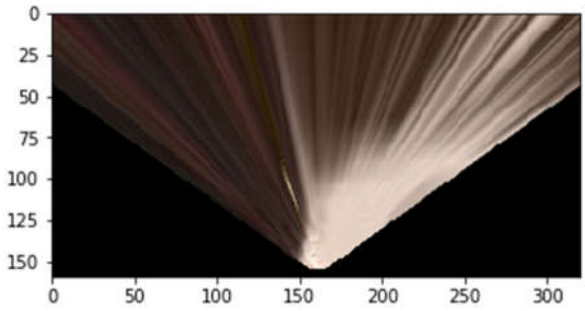


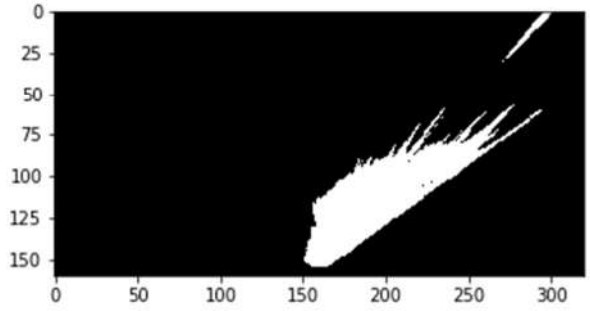
Color Thresholding:

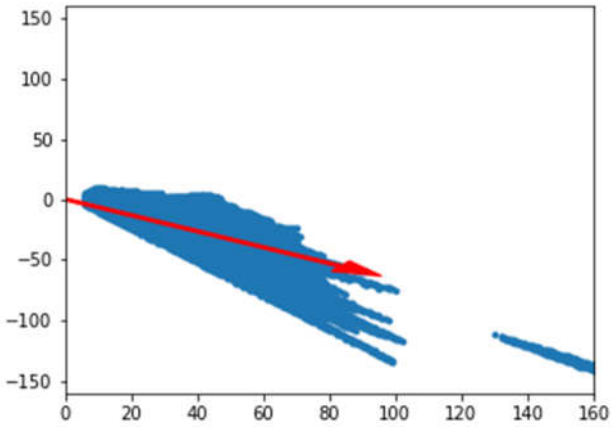


Co-ordinate Transformations:





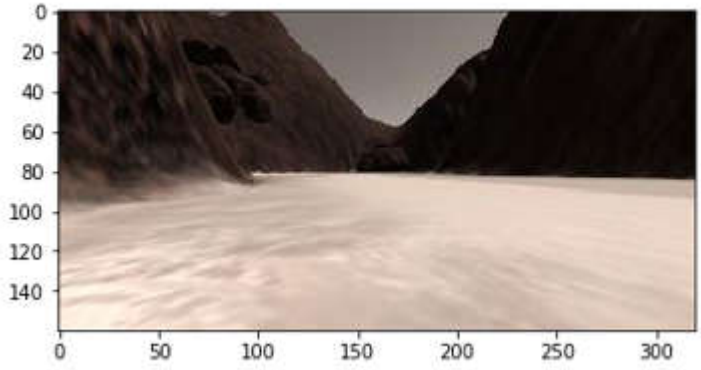




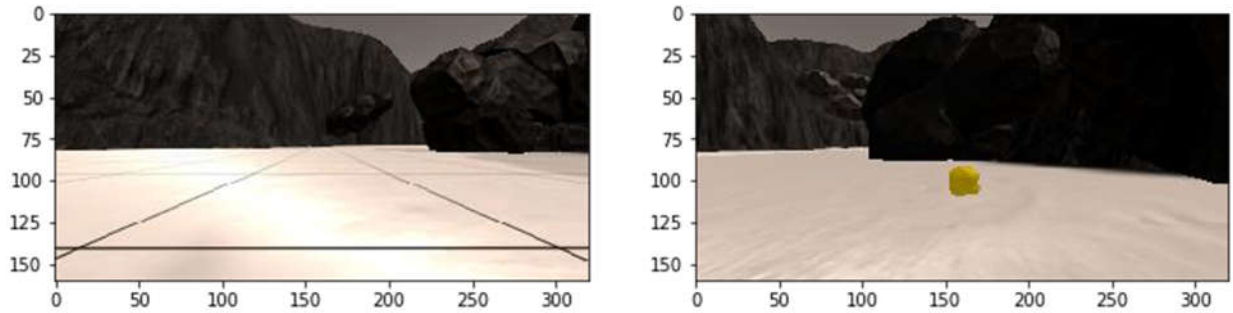
With recorded Images:

Quick look at the data

```
Out[3]: <matplotlib.image.AxesImage at 0x1e2ef26dba8>
```



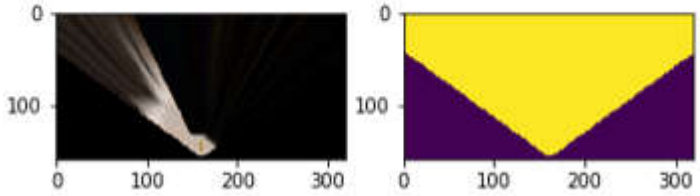
Calibration Data:



Perspective transform:

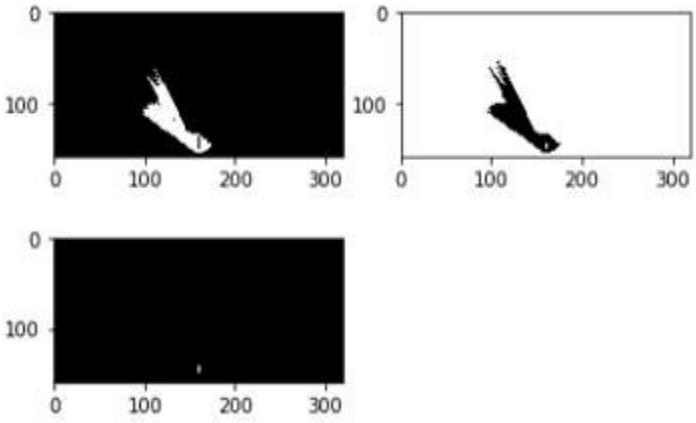
Perspective transform returns the warped image as well as the out of F.O.V region:

```
Out[5]: <matplotlib.image.AxesImage at 0x1e2f034fac8>
```



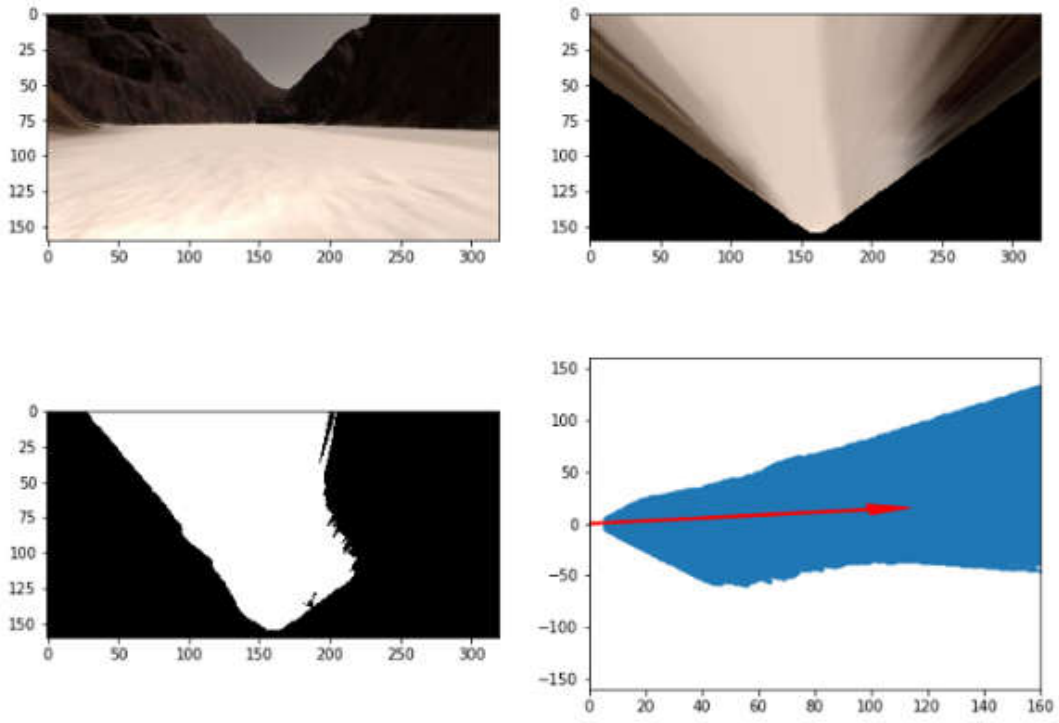
Color Thresholding:

```
Out[6]: <matplotlib.image.AxesImage at 0x1e2f03cb438>
```



Co-ordinate Transformations:

```
Out[7]: <matplotlib.patches.FancyArrow at 0x1e2f0494e10>
```



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.

Video with Recorded Dataset is shared in Output folder.

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.

Changes made to Perception.py script:

A. Changes are made to color_thresh() function to identify rock samples, navigable path and obstructions. The rgb threshold chosen for rock samples is (100,100,50) as the red and green pixels are brighter but the blue pixels are darker.

```
def color_thresh(img, rgb_thresh=(160, 160, 160)):
    color_select = np.zeros_like(img[:,:,0])
    obs_select = np.zeros_like(img[:,:,0])
    rock_select = np.zeros_like(img[:,:,0])
    rock_thresh=(100,100,50)
    navig_thresh = (img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,1] > rgb_thresh[1]) \
                & (img[:,:,2] > rgb_thresh[2])
    obs_thresh = (img[:,:,0] < rgb_thresh[0]) \
                & (img[:,:,1] < rgb_thresh[1]) \
                & (img[:,:,2] < rgb_thresh[2])
    rocksample_thresh = (img[:,:,0] > rock_thresh[0]) \
                & (img[:,:,1] > rock_thresh[1]) \
                & (img[:,:,2] < rock_thresh[2])
    color_select[navig_thresh] = 1
    obs_select[obs_thresh] = 1
    rock_select[rocksample_thresh] = 1
    return color_select,obs_select,rock_select
```

B. Changes are made to the perspect_transform() function to find the region which is out of field of view of the camera as shown here :

```
def perspect_transform(img, src, dst):

    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))
    out_of_view =  cv2.warpPerspective(np.ones_like(img[:,:,0]), M, (img.shape[1], img.shape[0]))
    return warped, out_of_view
```

C. Changes to the perception_step() is made to incorporate all the above updated method. Each of the color threshold image of navigable path, obstacle and the rock are multiplied by 255 (binary 1 is equivalent 255-pixel value) and the Rover. World map is updated. Average rock sample distance and angles are also added to the Rover state. Respective variable is added in Rover state in driver.py file.

```
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

    dst_size = 5
    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],
                  ])
    # 2) Apply perspective transform
    warped,out_of_view = perspect_transform(image, source, destination)
    # 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
    threshed,obs,rock = color_thresh(warped)
    obs_map=np.absolute(np.float32(obs)-1)*out_of_view
```

```
    # 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] = obs_map*255
    Rover.vision_image[:,:,1] = rock*255
    Rover.vision_image[:,:,2] = threshed*255
    # 5) Convert map image pixel values to rover-centric coords
    xpix_t, ypix_t = rover_coords(threshed)
    xpix_obs, ypix_obs = rover_coords(obs)
    xpix_rock, ypix_rock = rover_coords(rock)
    # 6) Convert rover-centric pixel values to world coordinates
    world_size = Rover.worldmap.shape[0]
    scale = 2 * dst_size
    xpix_world,ypix_world = pix_to_world(xpix_t, ypix_t, Rover.xpos[Rover.count],
Rover.ypos[Rover.count], Rover.yaw[Rover.count], world_size, scale)
    obs_xpix_world,obs_ypix_world = pix_to_world(xpix_obs, ypix_obs, Rover.xpos[Rover.count],
Rover.ypos[Rover.count], Rover.yaw[Rover.count], world_size, scale)
    rock_xpix_world,rock_ypix_world = pix_to_world(xpix_rock, ypix_rock, Rover.xpos[Rover.count],
Rover.ypos[Rover.count], Rover.yaw[Rover.count], 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
    Rover.worldmap[obs_ypix_world, obs_xpix_world, 0] += 50
    Rover.worldmap[rock_ypix_world, rock_xpix_world, 1] +=50
    Rover.worldmap[ypix_world,xpix_world, 2] += 50

    # 8) Convert rover-centric pixel positions to polar coordinates
```

```
rover_centric_distances,rover_centric_angles = to_polar_coords(xpix_t, ypix_t)
rock_distances,rock_angles = to_polar_coords(xpix_t, ypix_t)
# Update Rover and rock  pixel distances and angles
    Rover.nav_dists = rover_centric_distances
    Rover.nav_angles = rover_centric_angles
    Rover.rock_dist=rock_dist
    Rover.rock_angles=rock_angles

return Rover
```

Changes made to decision.py script:

Following changes are done to the decision.py script.

a. Forward mode detection is combined with detection of rock samples. If there is no rock samples detected and it is in forward mode then the throttle, steer and brake will be adjusted based on Rover.stop_forward value. Two additional logics are added for the following:
    a. Is_clear(Rover) function is added to find if the majority of the terrain in front of the rover is navigable terrain. This flag is used to change the mode of the Rover mode to 'stop' mode as evident in the code:

```
def is_clear(Rover):
  #Verifying if the pixels in front of the over are mainly ground
  clear = (np.sum(Rover.vision_image[140:150,150:170,2]) > 130)\
   & (np.sum(Rover.vision_image[110:120,150:170,2]) > 100)\
   & (np.sum(Rover.vision_image[150:153,155:165,2]) > 20)
  return clear
```

    b. Added a new variable Rover.random_steer to add a random number between (-5,5) to the steering angle when in forward mode. This is done to avoid repeatability in the path. This helps the rover to discover new path rather than taking the same path again and helps to avoid going in loops.

```
  Rover.random_steer = random.randint(-5,5)
```

Decision Tree code for forward mode:

```
if Rover.mode == 'forward'and len(Rover.rock_angles)==0 and Rover.near_sample ==0 :
    #Add a random steer angle to avoid loops
    Rover.random_steer = random.randint(-5,5)
    # Check the extent of navigable terrain
```

```
        if 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
            # Set steering to average angle clipped to the range +/- 15
            Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -20, 20) + Rover.random_steer
        # If there's a lack of navigable terrain pixels then go to 'stop' mode
        elif len(Rover.nav_angles) < Rover.stop_forward or not(is_clear(Rover)):
                # Set mode to "stop" and hit the brakes!
                Rover.throttle = 0
                # Set brake to stored brake value
                Rover.brake = Rover.brake_set
                Rover.steer = 0
                Rover.mode = 'stop'
```

b. Similarly, stop mode is also combined with rock sample detection. When the rover is in stop mode and there is no rock sample detected then steer, throttle and brake values are adjusted accordingly.

Decision Tree code for stop mode

```
elif Rover.mode == 'stop' and len(Rover.rock_angles)==0 and Rover.near_sample == 0 :
        # 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
                Rover.steer = -25 # Could 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
                # Set steer to mean angle
                Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -20, 20)
                Rover.mode = 'forward'
```

c. Rock sample is detected by the presence of non-zero element in Rover.rock_angles. Steering angle is calculated by the mean value of Rover.rock_angle values, so that Rover can pick up the rock samples. Steering, throttle and brake values are adjusted based on the number of non-zero elements in Rover.rock_angles

Decision Tree code for Rock sample detection:

```
elif len(Rover.rock_angles)!=0:
        #steer towards the rock sample
        Rover.steer = np.clip(np.mean(Rover.rock_angles * 180/np.pi), -15, 15)
        # if there is enough vision data for rock sample
        if len(Rover.rock_angles) >= Rover.rock_threshold :
```

```
        #check if the velocity is less than 1, throttle at slower rate to reach the rock
        if Rover.vel < 1:
            Rover.throttle = 0.1
            Rover.brake = 0
        # if the velocity is more than 1, stop throttle and apply brake at a steady rate
        elif Rover.vel>=1 :
            Rover.throttle =0
            Rover.brake = 5
    elif len(Rover.rock_angles) < Rover.rock_threshold:
        # set appropiate velocity and apply brake to reach towards the rock sample
        if Rover.vel < 0.7:
            Rover.throttle= 0.1
            Rover.brake =0
        elif Rover.vel >= 0.7:
            Rover.throttle =0
            Rover.brake = 5
        # check if Rock sample is nearby. If yes adjust setting to reach to the rock sample
        if Rover.near_sample :
            Rover.throttle=0
            Rover.brake=Rover.brake_set
            Rover.steer=0
            #check if the Rover has stopped and Rover is not already busy picking up samples
            if Rover.vel==0 and not (Rover.picking_up):
                # set the rock pick up flag to send the command
                Rover.send_pickup = True
```

d. Variables are added in driver.py for the Rover.rock_dist and Rover.Rock_angles, Rover.navcount,Rover.isclearflag, Rover.random_steer.

e. Logic is added to avoid obstacles by using is_clear (Rover) function:

```
elif not is_clear(Rover):
        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:
            Rover.throttle = 0
            Rover.brake = 0
            Rover.steer = -25
            Rover.mode = 'stop'
```

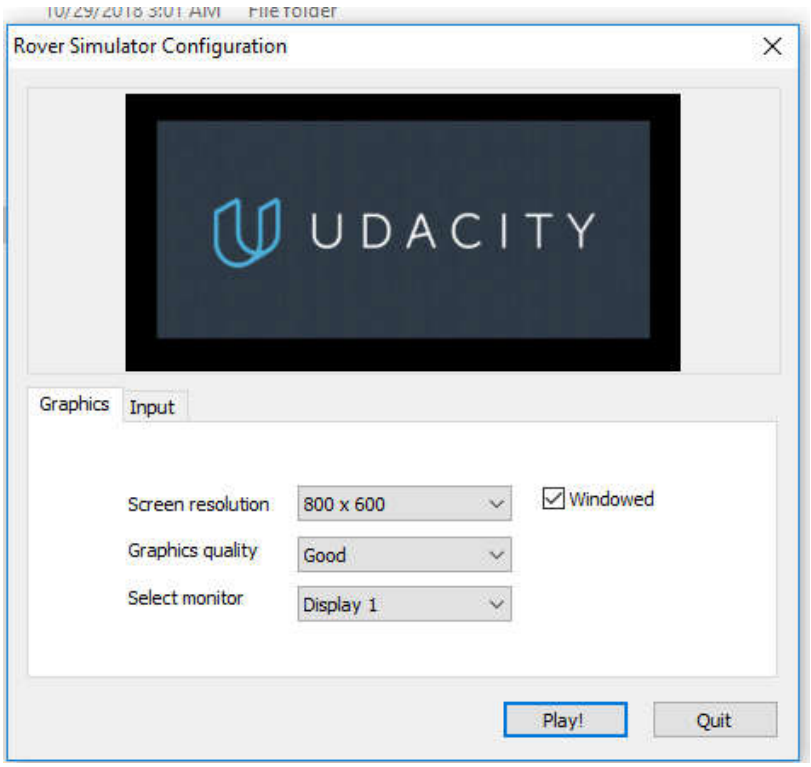
Autonomous Mode

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

Note: running the simulator with different choices of resolution and graphics quality may produce different results, particularly on different machines! Make a note of your simulator settings (resolution and graphics quality set on launch) and frames per second (FPS output to terminal by `drive_rover.py`) in your writeup when you submit the project so your reviewer can reproduce your results.

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

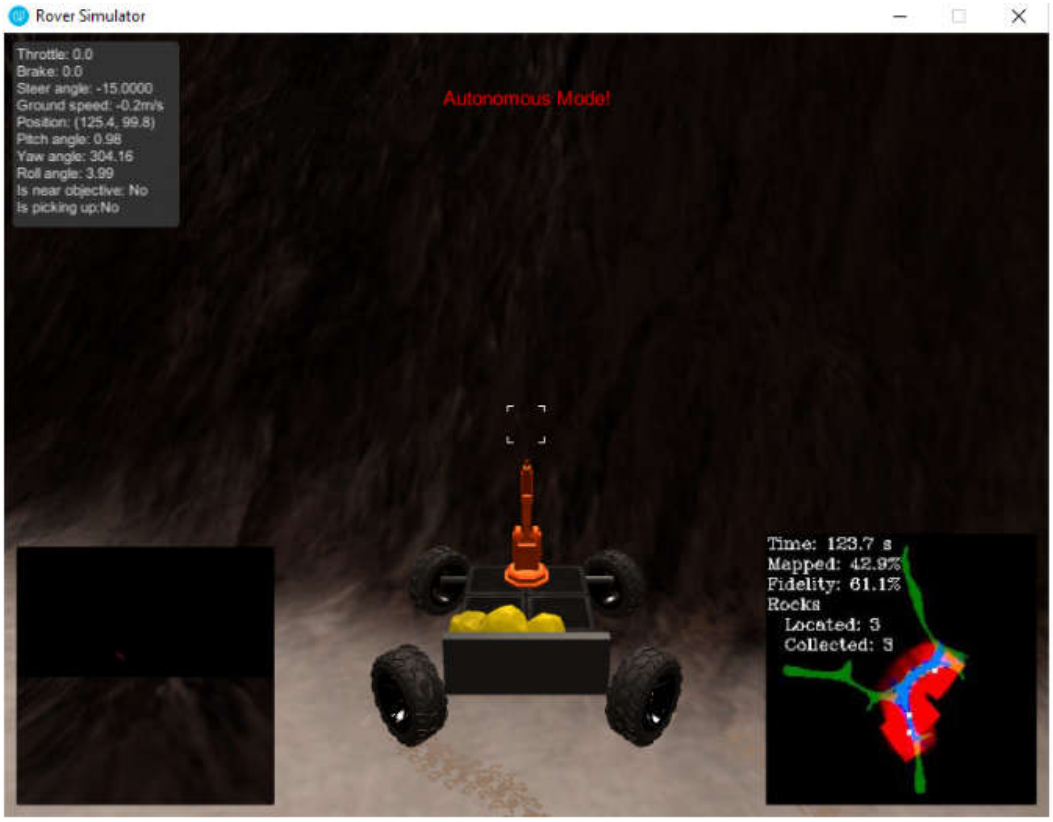
Simulator setting:



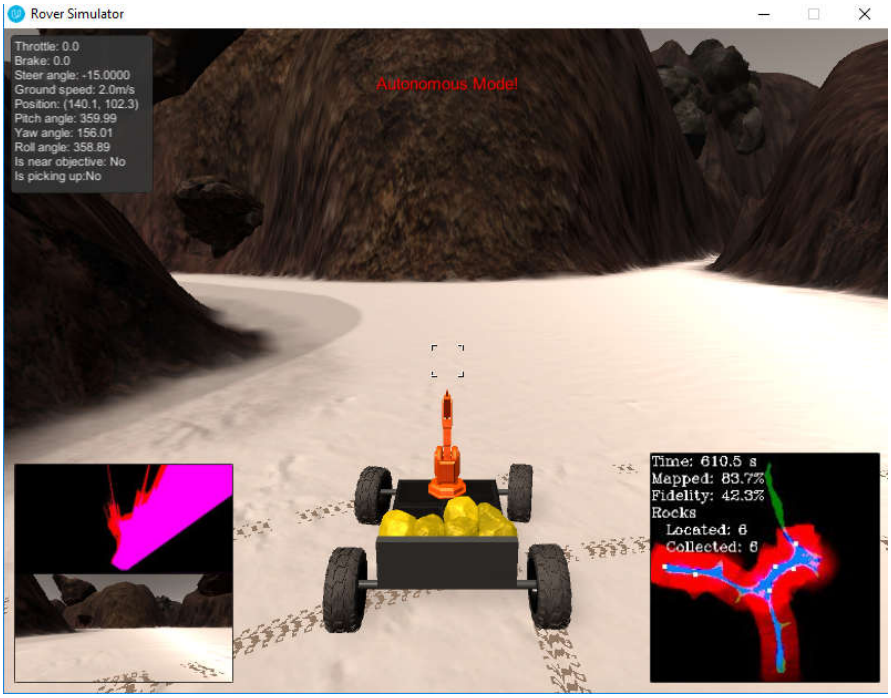
Rover is run in autonomous mode and the rover is able to pick all six rock samples. The fidelity and map covered is 61.9% and 42.9% respectively. Addition of random steering angle avoids the Rover to go in a loop. Moreover, whenever there are two rock samples located opposite to each other along the wall sides, Rover is not able to pick both the samples. These can be improved by improving the decision tree algorithm in future. The fidelity sometimes comes below 60%, which can be also improved by improving the decision tree. Below is a snapshot of rover screen in autonomous mode.

Autonomous Navigation Results:

Fidelity >60%, Map >40%, Rock detected and collected =3



Fidelity =42%, Map = 83.7% %, Rock detected & collected =6



Sometimes the Rover is getting stuck in below scenario. Need to improve decision tree to avoid this.