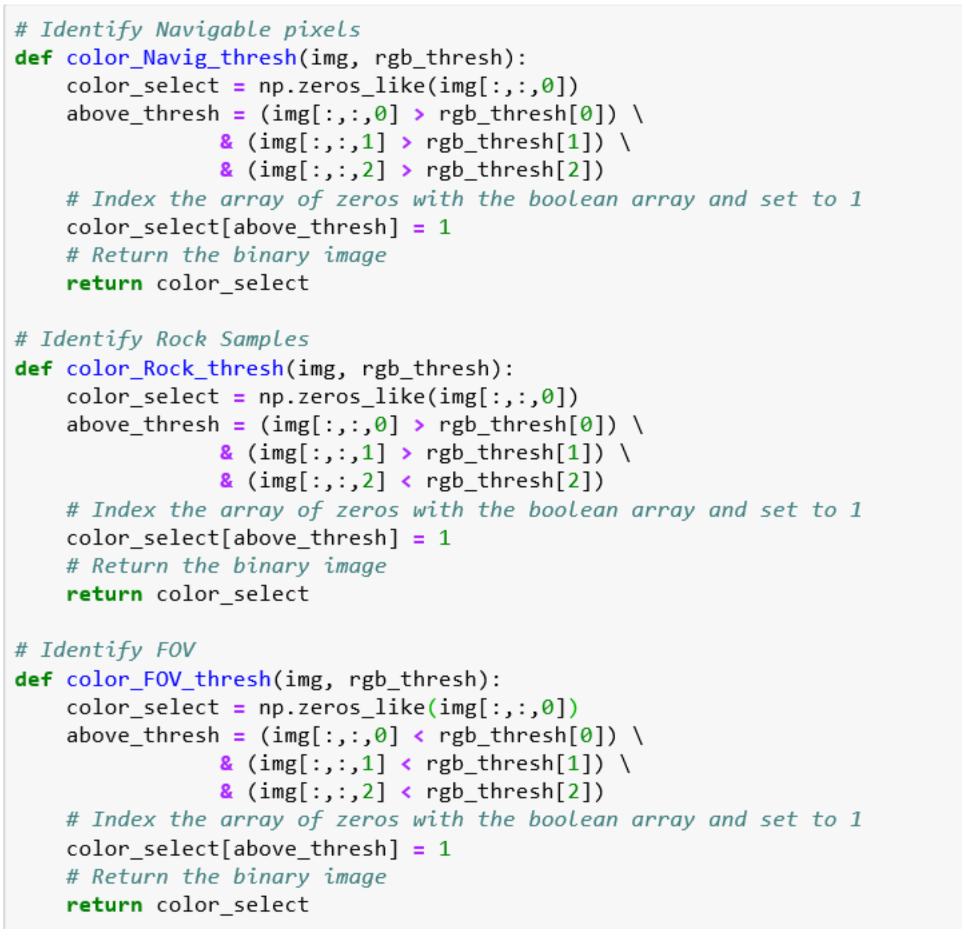
# Project: Search and Sample Return

## Notebook Analysis

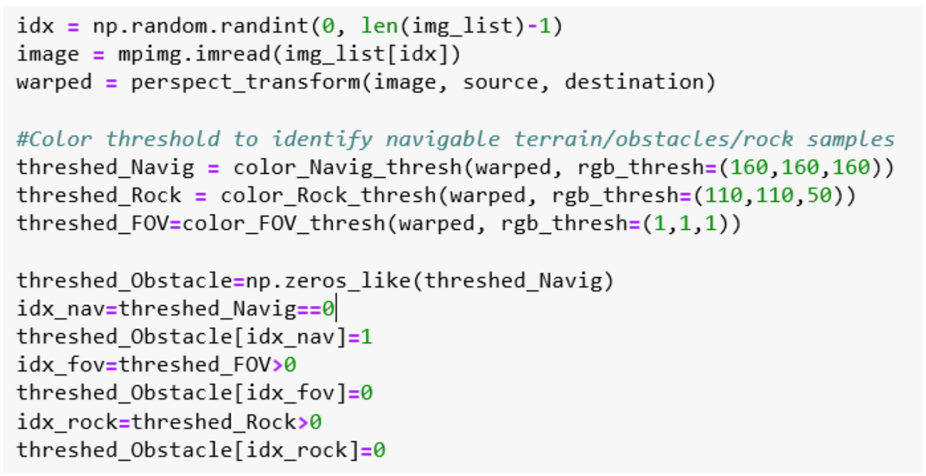
- *Colour Thresholding :* In the Jupyter Notebook, the colour thresholding function works for navigable terrains. I've modified this for Navigable pixels, Rock Sample Pixels and FOV(Field of view of camera mounted).The individual RGB thresholds are accordingly defined based on their colour for rocks, navigable area and FOV. After finding out the individual pixel positions for these, the obstacle pixels are identified. The code and output plots for these are described below.



```
# Identify Navigable pixels
def color_Navig_thresh(img, rgb_thresh):
    color_select = np.zeros_like(img[:,:,0])
    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

# Identify Rock Samples
def color_Rock_thresh(img, rgb_thresh):
    color_select = np.zeros_like(img[:,:,0])
    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

# Identify FOV
def color_FOV_thresh(img, rgb_thresh):
    color_select = np.zeros_like(img[:,:,0])
    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
```

**Script 1: Colour Thresholding**

The above functions are then called in for further actions based on the following code, which determines the FOV and obstacle pixels on the basis of indexes available for Navigable pixels and colour thresholding.



```
idx = np.random.randint(0, len(img_list)-1)
image = mpimg.imread(img_list[idx])
warped = perspect_transform(image, source, destination)

#Color threshold to identify navigable terrain/obstacles/rock samples
threshed_Navig = color_Navig_thresh(warped, rgb_thresh=(160,160,160))
threshed_Rock = color_Rock_thresh(warped, rgb_thresh=(110,110,50))
threshed_FOV=color_FOV_thresh(warped, rgb_thresh=(1,1,1))

threshed_Obstacle=np.zeros_like(threshed_Navig)
idx_nav=threshed_Navig==0
threshed_Obstacle[idx_nav]=1
idx_fov=threshed_FOV>0
threshed_Obstacle[idx_fov]=0
idx_rock=threshed_Rock>0
threshed_Obstacle[idx_rock]=0
```

**Script 2: Decisions to find out Navigable pixels, rocks, FOV and obstacle**

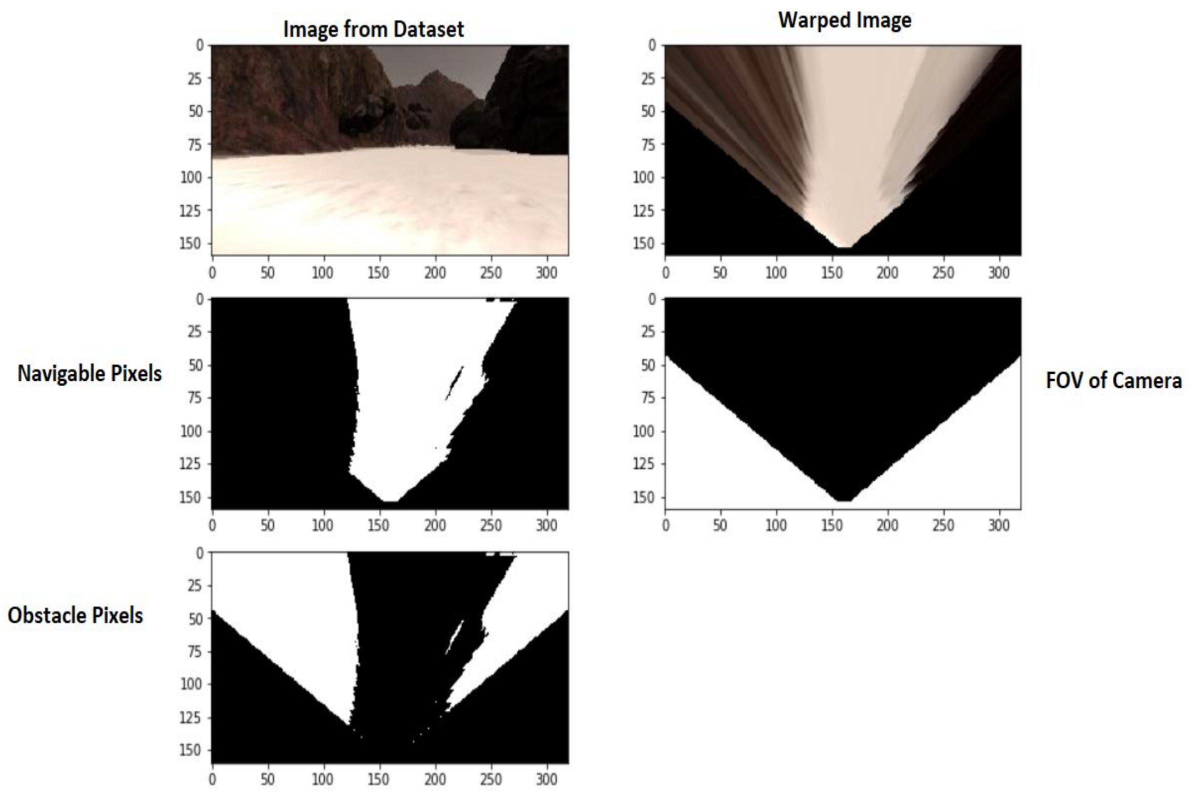


**Fig 1: Output Navigable pixels, rocks, FOV and obstacle (White Pixels in binary images)**

After implementation of script 1 and script 2, the output plots for navigable terrain, obstacles, rock samples and FOV looks like Fig 1. Here the white pixels are the respective outputs of warped image for each of the navigable terrain, FOV and obstacles.

- *Populate* process_image() function: In the process_image() function, the image clips are made to follow the following actions:
    1. Perspective Transform on Image : Based on source and destination points, the image is warped in a top down view.
    2. Colour Thresholding: On the warped image colour thresholding is done for navigable, rock sample, FOV and obstacle pixels as per the scripts shown in Script 1 and Script 2.

3. Rover Frame Transformation: The identified pixel positions for navigable area, rocks and obstacles are then converted in world frame for the rover to navigate.
4. World Frame Transformation: The rover frame pixels are further transformed in world frame using Translational and Rotational matrices with the Rover X, Rover Y and Rover Yaw angles at any particular instant with the image pixels in front.
5. Data Worldmap: The obtained pixels are then segregated in Red, Blue and Green pixels based on them being Obstacles, Navigable area or Rock samples(if available).

Analysis and Observation: Due to the improved Colour thresholding logic applied, along with correct rover and world frame transformations, the perception step gives a good fidelity by covering almost all the navigable area, obstacle and rocks with high precision.

The output looks like this:

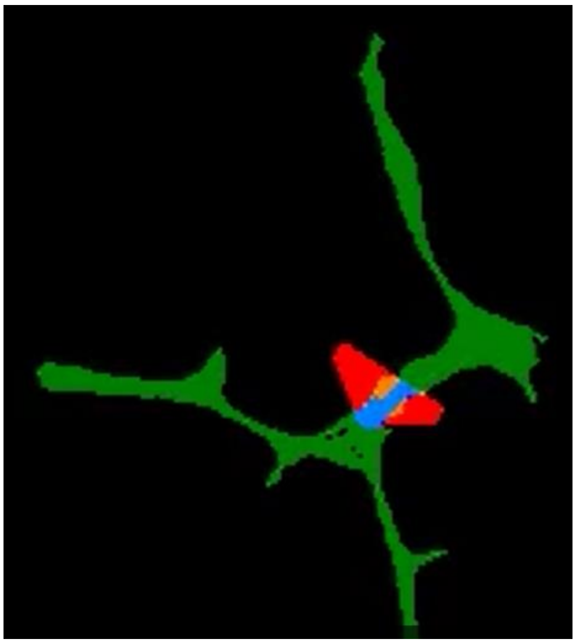
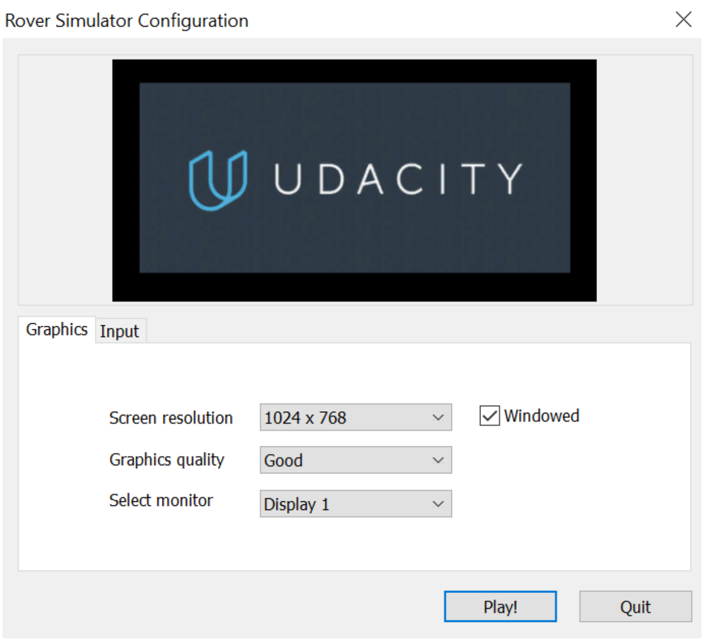


**Fig 2: Overlay of output for process_image on ground truth**

## Autonomous Navigation and Mapping

For mapping in autonomous mode, the following Rover simulator settings is maintained.

Screen Resolution - 1024x768
Graphics Quality - Good


Rover Simulator Configuration

Graphics | Input

Screen resolution: 1024 x 768 — [✓ Windowed]
Graphics quality: Good
Select monitor: Display 1

- *perception.py :* The perception_step() in perception.py script is modified based on Warping, Color Thresholding, transformation in rover and world frames. The main function for this script is to give accurate data classification based on image pixels for navigable terrain of Rover.

  First, the image is warped based on source and destination points for top down view. This is then allowed for Navigable pixel classification based on a certain RGB threshold. Similarly the FOV of camera and Rock samples are obtained based on the RGB threshold. After obtaining Navigable pixel, FOV of camera and Rock Samples (if any) the obstacles are identified based on the indexes of pixels remains uncovered.

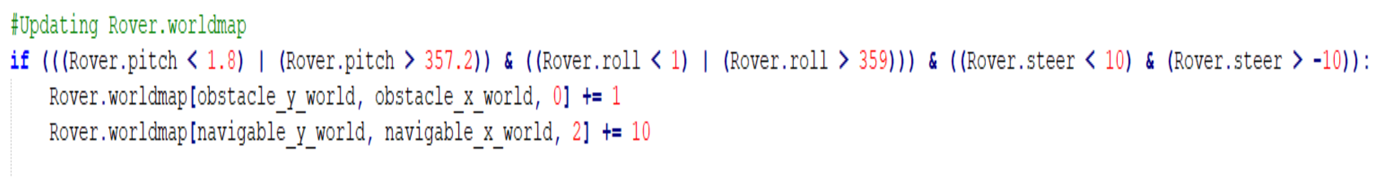
  *These pixels are named as:*

- "threshed_Navig": For navigable pixels
- "threshed_Rock": For Rock samples
- "threshed_FOV": For Field of view of Camera
- "threshed_Obstacle": For obstacle pixels

After that, the pixel positions (X,Y) for "threshed_Navig" and "threshed_Obstacle" are obtained in Rover frame which are further transformed in world frame with the help of translational and rotational matrix, given the Rover positions(X,Y) and Rover Yaw.
Similarly Rock pixels are updated in case any rock is found.

With this, Rover.vision_image and Rover.worldmap is updated as follows:

```
#Updating Rover.vision_image
Rover.vision_image[:,:,0] = threshed_Obstacle*255
Rover.vision_image[:,:,2] = threshed_Navig*255
```

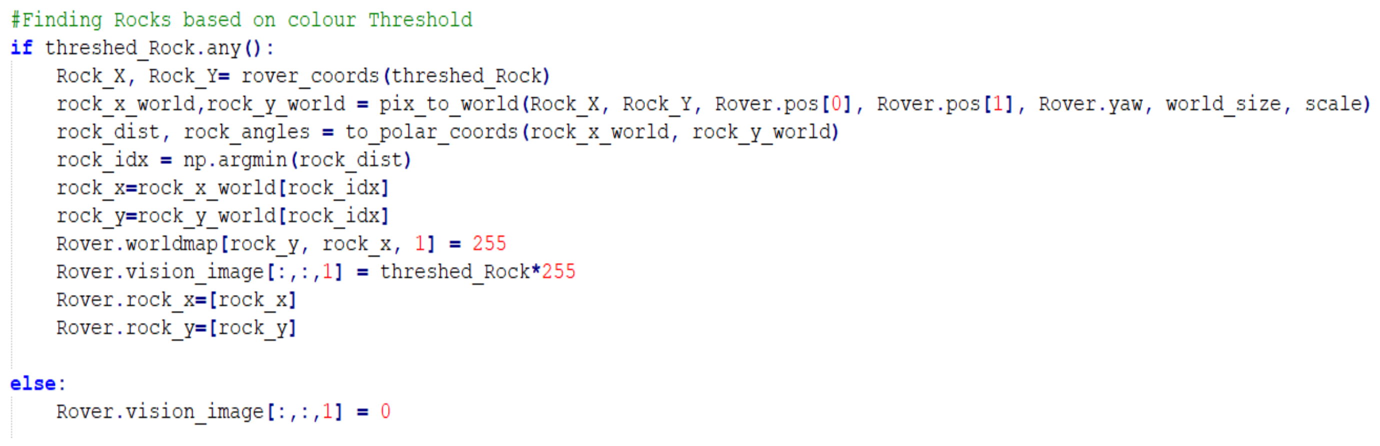


```
#Updating Rover.worldmap
if (((Rover.pitch < 1.8) | (Rover.pitch > 357.2)) & ((Rover.roll < 1) | (Rover.roll > 359))) & ((Rover.steer < 10) & (Rover.steer > -10)):
    Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1
    Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 10
```

**Script 3: Rover Display in Simulator**

Rover.worldmap is updated considering the Rover Roll and Pitch. It allows the world map to update only when the constraints of Roll, Pitch and Steer are satisfied. This prevents percepetion.py to perform on erroneous input data at times of high Roll or pitch where Rover might be facing sky at different camera angles.

The rock samples are updated like this:

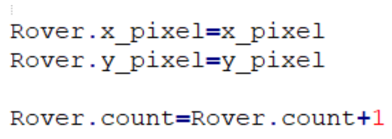


```
#Finding Rocks based on colour Threshold
if threshed_Rock.any():
    Rock_X, Rock_Y= rover_coords(threshed_Rock)
    rock_x_world,rock_y_world = pix_to_world(Rock_X, Rock_Y, Rover.pos[0], Rover.pos[1], Rover.yaw, world_size, scale)
    rock_dist, rock_angles = to_polar_coords(rock_x_world, rock_y_world)
    rock_idx = np.argmin(rock_dist)
    rock_x=rock_x_world[rock_idx]
    rock_y=rock_y_world[rock_idx]
    Rover.worldmap[rock_y, rock_x, 1] = 255
    Rover.vision_image[:,:,1] = threshed_Rock*255
    Rover.rock_x=[rock_x]
    Rover.rock_y=[rock_y]

else:
    Rover.vision_image[:,:,1] = 0
```

**Script 4: Rock Identification based on Color Thresholding**

Here the green channel is assigned a value of 255 in case Rock is available in camera image else the green channel is set to 0.

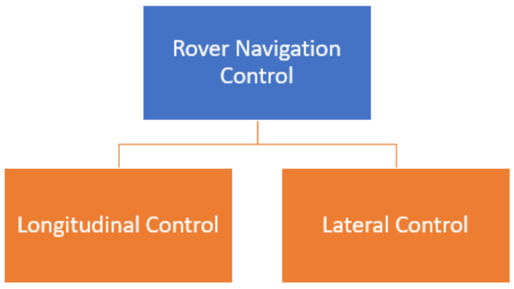
I've assigned the current (x,y) navigable pixels of Rover in Rover frame and the iteration count of drive_rover.py as Rover.count also in perception.py, these are later used in decision_step of decision.py:



```
Rover.x_pixel=x_pixel
Rover.y_pixel=y_pixel

Rover.count=Rover.count+1
```

- *decision.py():* In this script, the entire decision_step is written from scratch and nothing is used from the existing code.

  The rover requires to navigate in the given map area, with the help of Throttle, Brake and Steer Control. The given map here is a 2D map, which signifies that the control required will be for Lateral Control and Longitudinal Control.

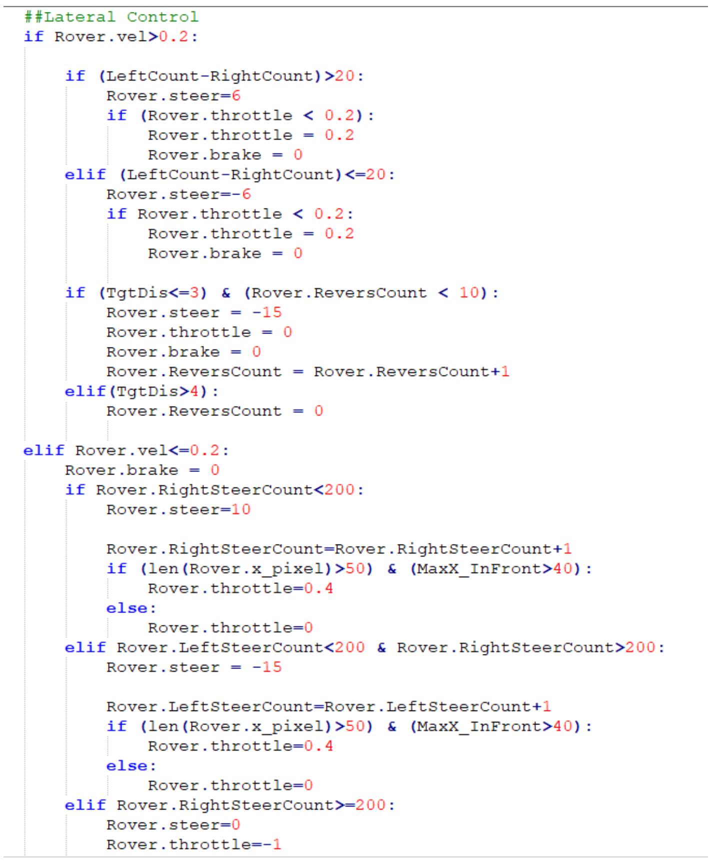


**Lateral Control :** The lateral control of Rover handles the Rover steer based on navigable pixels in Left and Right hand side of Rover and in case of no pixel being available in front. This is divided into 2 parts for Rover.vel (i.e., Rover Velocity) being more than 0.2 and Rover.vel less than 0.2. In case, when Rover Velocity is less than 0.2, and there is no pixel available in front signifying Rover has reached one of the 8 corners of map, the Lateral control allows the rover to steer in either of Left or Right hand side area based on the availability of navigable pixels.

The navigable area for Rover is not calculated by conventional means of total number of pixels available, rather the decision_step calculates the number of pixels exactly in front of Rover at the lateral positions of 0 (i.e., Rover.y_pixel = 0). This gives the number of pixels available for Rover to navigate in front. Hence with the help of Left pixels, Right Pixels and Front pixels, the Rover is allowed to navigate.

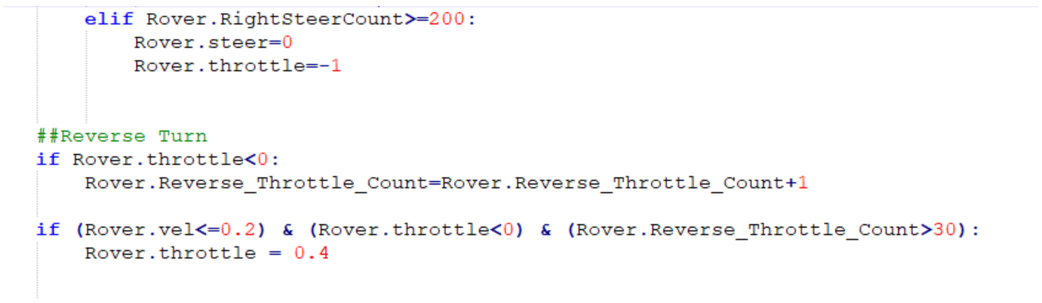
**Reverse Steer:** There are a few obstacles in the middle of the map, which restricts the Rover navigation when accidentally the Rover gets stuck into some of these obstacle. This issue is resolved by adding reverse control. In case Rover velocity is less than 0.2 although Rover Throttle being 0.4 due to misalignment of Rover camera because of getting stuck and increasing Rover pitch and Roll angle, the Rover is given a reverse throttle for a few iterations, if the rover increases the velocity by then, it comes out of the loop otherwise, it proves there are obstacles behind Rover as well because of which its unable to steer Reverse. At this point, the Rover is given a Left and Right Steer and a Front Throttle which forcefully brings the Rover out of the stuck obstacle.

This although happens rarely, as the decision_step doesn't allow the Rover to navigate through the obstacles, yet at some cases due to some fault in image input, the Rover may get stuck and in those instances the above logic will prove helpful.



```
##Lateral Control
if Rover.vel>0.2:

    if (LeftCount-RightCount)>20:
        Rover.steer=6
        if (Rover.throttle < 0.2):
            Rover.throttle = 0.2
            Rover.brake = 0
    elif (LeftCount-RightCount)<=20:
        Rover.steer=-6
        if Rover.throttle < 0.2:
            Rover.throttle = 0.2
            Rover.brake = 0

    if (TgtDis<=3) & (Rover.ReversCount < 10):
        Rover.steer = -15
        Rover.throttle = 0
        Rover.brake = 0
        Rover.ReversCount = Rover.ReversCount+1
    elif(TgtDis>4):
        Rover.ReversCount = 0

elif Rover.vel<=0.2:
    Rover.brake = 0
    if Rover.RightSteerCount<200:
        Rover.steer=10

        Rover.RightSteerCount=Rover.RightSteerCount+1
        if (len(Rover.x_pixel)>50) & (MaxX_InFront>40):
            Rover.throttle=0.4
        else:
            Rover.throttle=0
    elif Rover.LeftSteerCount<200 & Rover.RightSteerCount>200:
        Rover.steer = -15

        Rover.LeftSteerCount=Rover.LeftSteerCount+1
        if (len(Rover.x_pixel)>50) & (MaxX_InFront>40):
            Rover.throttle=0.4
        else:
            Rover.throttle=0
    elif Rover.RightSteerCount>=200:
        Rover.steer=0
        Rover.throttle=-1
```

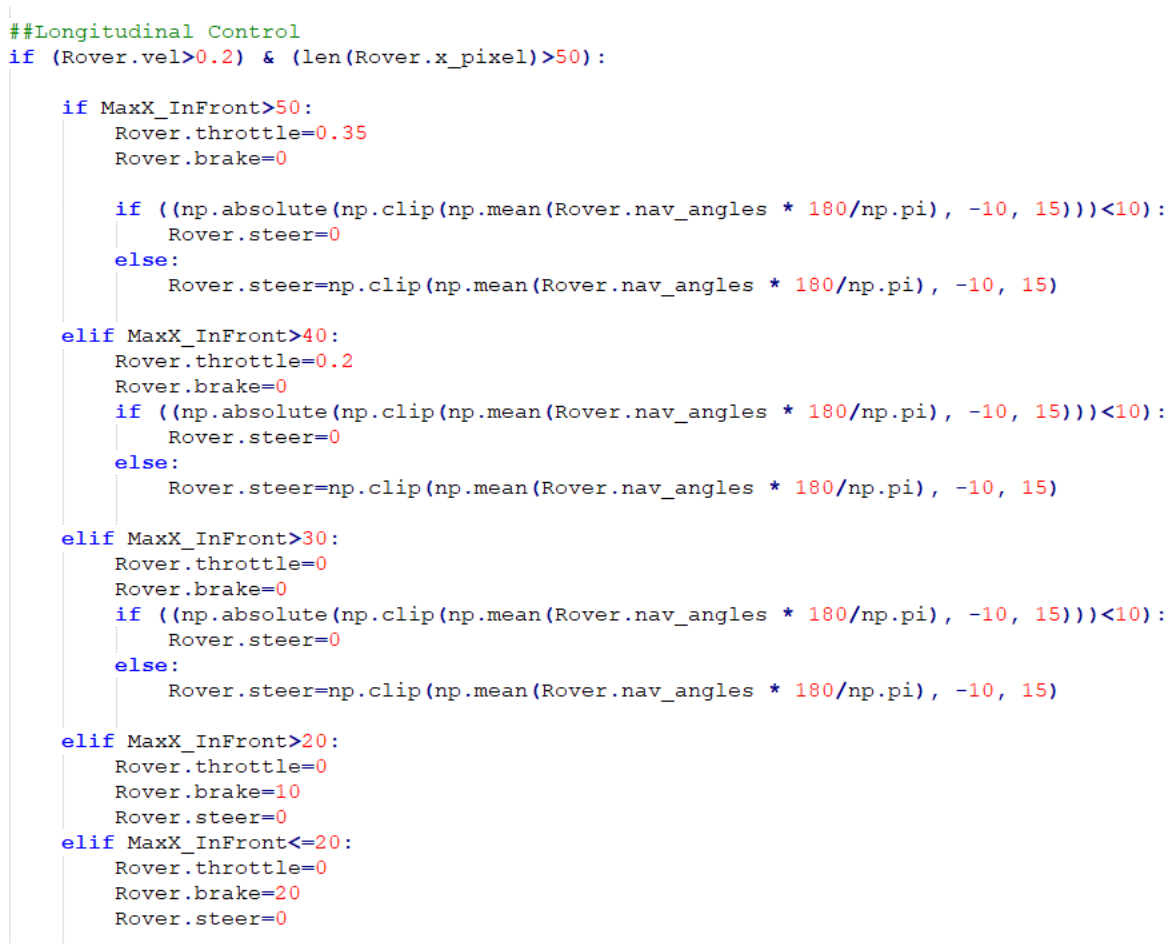
**Script 5: Lateral Control in decision_step**



```
    elif Rover.RightSteerCount>=200:
        Rover.steer=0
        Rover.throttle=-1


##Reverse Turn
if Rover.throttle<0:
    Rover.Reverse_Throttle_Count=Rover.Reverse_Throttle_Count+1

if (Rover.vel<=0.2) & (Rover.throttle<0) & (Rover.Reverse_Throttle_Count>30):
    Rover.throttle = 0.4
```

**Script 6: Reverse Turn in decision_step**

**Longitudinal Control:** The longitudinal control of Rover takes care of the throttle and brake of Rover. This is dependent on number of pixels available in front and maximum longitudinal pixel in Rover frame. Depending on the maximum length of pixel available in front, the Rover is set to varying ranges of Throttle and Brake input. Here at some cases Steer is also involved, for smooth navigation.



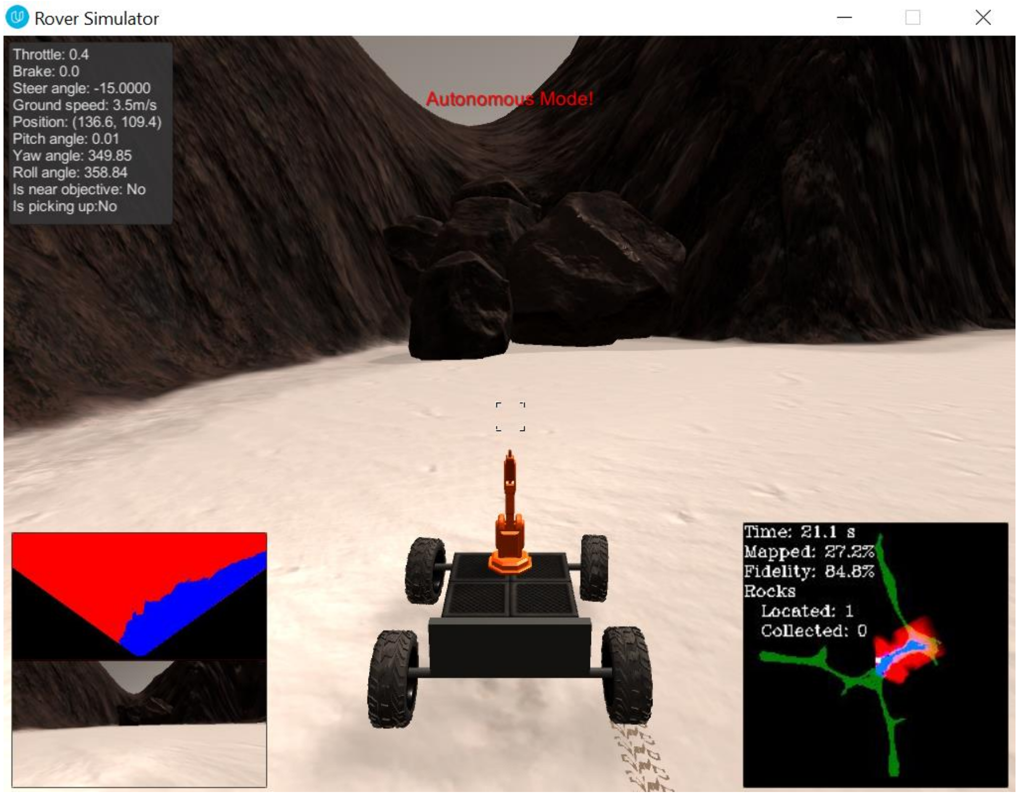
```
##Longitudinal Control
if (Rover.vel>0.2) & (len(Rover.x_pixel)>50):

    if MaxX_InFront>50:
        Rover.throttle=0.35
        Rover.brake=0

        if ((np.absolute(np.clip(np.mean(Rover.nav_angles * 180/np.pi), -10, 15)))<10):
            Rover.steer=0
        else:
            Rover.steer=np.clip(np.mean(Rover.nav_angles * 180/np.pi), -10, 15)

    elif MaxX_InFront>40:
        Rover.throttle=0.2
        Rover.brake=0
        if ((np.absolute(np.clip(np.mean(Rover.nav_angles * 180/np.pi), -10, 15)))<10):
            Rover.steer=0
        else:
            Rover.steer=np.clip(np.mean(Rover.nav_angles * 180/np.pi), -10, 15)

    elif MaxX_InFront>30:
        Rover.throttle=0
        Rover.brake=0
        if ((np.absolute(np.clip(np.mean(Rover.nav_angles * 180/np.pi), -10, 15)))<10):
            Rover.steer=0
        else:
            Rover.steer=np.clip(np.mean(Rover.nav_angles * 180/np.pi), -10, 15)

    elif MaxX_InFront>20:
        Rover.throttle=0
        Rover.brake=10
        Rover.steer=0
    elif MaxX_InFront<=20:
        Rover.throttle=0
        Rover.brake=20
        Rover.steer=0
```

**Script 7: Longitudinal Control in decision_step**

**Mapping of Unmapped Area:** The ground truth world map, is having around 8 corners of which 5 corner points are chosen. In case Rover has not mapped a particular area, the rover slope is calculated based on Rover world Position and the Corner Point of world Map, the Rover is then slowly steered towards the corner point based on the Rover yaw, calculated slope and a proportional gain.

Based on the applied algorithms in Lateral, Longitudinal and Reverse control for Rover, the fidelity of Rover is maintained and the Rover is able to navigate 100% of the World Map.

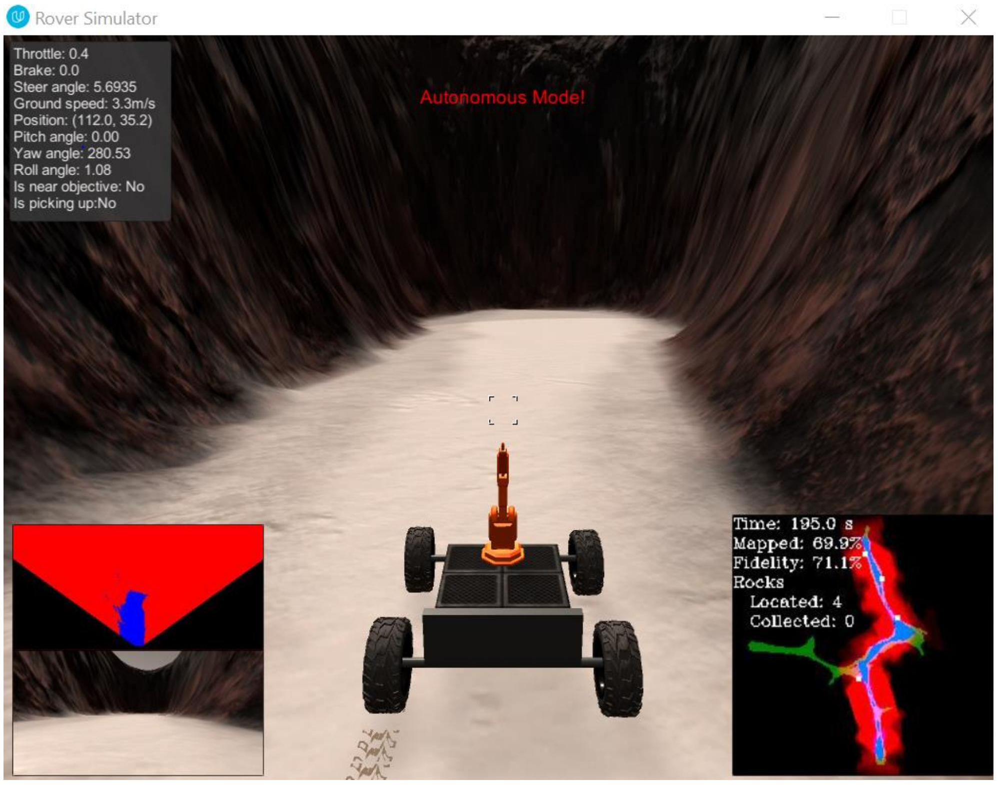
Analysis and Observation: As the Rover starts navigating, the fidelity of Rover remains at 84.8% with 27.2% of mapped area as shown in Fig 3. As the Rover further navigates, the fidelity comes down a bit, but overall remains above 60%. Fig 4 shows the fidelity as 71.1% at mapped area of 69.9%. In Fig 5, the fidelity of Rover is at 63.5% at mapped area of 79.8%.



```
Throttle: 0.4
Brake: 0.0
Steer angle: -15.0000
Ground speed: 3.5m/s
Position: (136.6, 109.4)
Pitch angle: 0.01
Yaw angle: 349.85
Roll angle: 358.84
Is near objective: No
Is picking up:No
```

```
Time: 21.1 s
Mapped: 27.2%
Fidelity: 84.8%
Rocks
  Located: 1
  Collected: 0
```

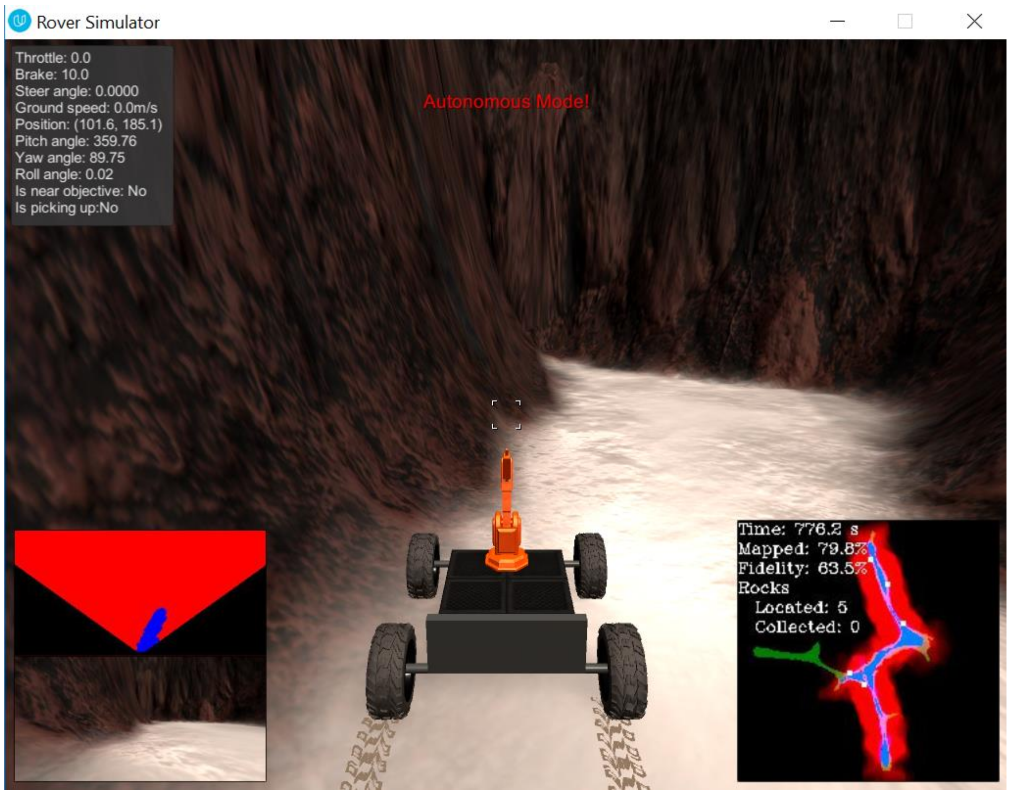
Fig 3: Fidelity – 84.8% at mapped area of 27.2%



```
Throttle: 0.4
Brake: 0.0
Steer angle: 5.6935
Ground speed: 3.3m/s
Position: (112.0, 35.2)
Pitch angle: 0.00
Yaw angle: 280.53
Roll angle: 1.08
Is near objective: No
Is picking up:No
```

```
Time: 195.0 s
Mapped: 69.9%
Fidelity: 71.1%
Rocks
  Located: 4
  Collected: 0
```

Fig 4: Fidelity – 71.1% at mapped area of 69.9%



```
Throttle: 0.0
Brake: 10.0
Steer angle: 0.0000
Ground speed: 0.0m/s
Position: (101.6, 185.1)
Pitch angle: 359.76
Yaw angle: 89.75
Roll angle: 0.02
Is near objective: No
Is picking up:No
```

```
Time: 776.2 s
Mapped: 79.8%
Fidelity: 63.5%
Rocks
  Located: 5
  Collected: 0
```

**Fig 5: Fidelity – 63.5% at mapped area of 79.8%**

## Conclusion:

- The fidelity is maintained above 60% for more than 40% of mapped area, sometimes at even better results.
- The Rover is able to map more than 4 Rock samples in every instance.
- At times, though fidelity may come down, but it increases again at around 50% of mapped area. With increase in percentage of Map area being covered, the fidelity degrades at cost of more percentage of Map being covered.

## Submission Criteria:

- **Fig 3-5, above meets the minimum submission criteria of this project where**:
  - 41.2% area is mapped
  - Fidelity is 70%
  - 1 Rock is located