Project Report

Phase 1

CSE483 Computer Vision

Submitted to:

Dr. Mahmoud Khalil

Eng. Mahmoud Selim

Submitted by:

Team 17

Anthony Amgad Fayek Selim 19P9880

Morcous Wael William Abrahim 19P8211

Marc Nagy Nasry Sorial 19P3041

Youssef Ashraf Mounir Showeter 19P4179

GitHub Repo:

https://github.com/Anthony-Amgad/Vision_Project_Team17

Code explained and methods used in each pipeline:

Packages:

First, those were the Packages needed in the project.

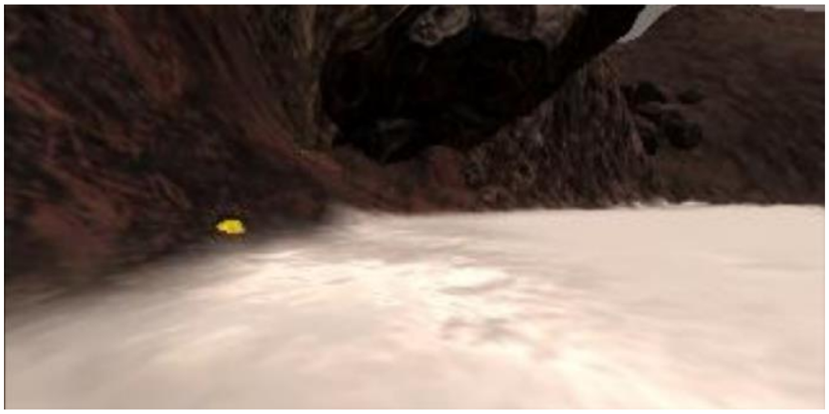
1) `numpy`
2) `opencv-python`
3) `python-socketio==4.6.1`
4) `eventlet`
5) `Pillow`
6) `Flask`
7) `matplotlib`
8) `scipy`
9) `imageio`
10) `python-engineio==3.13.2`

Introduction:

The following topics can be found in perception.py in the **perception_step()** function, which gets a parameter that contains the rover state.

Input image:

The first thing we have is our input images which we will get from the rover camera (Rover.img), which we assign to the variable (image). This Image looks like the following:

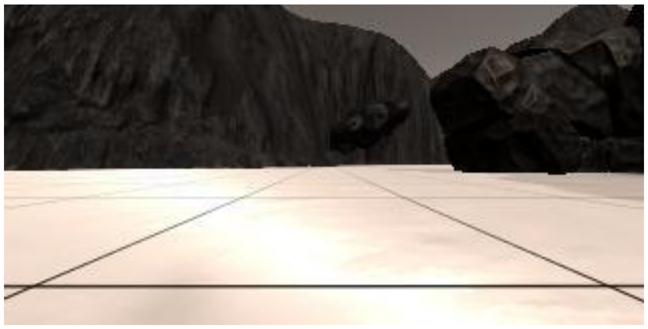


Warping the image:

In the following function we take the input image and warp it so we can view the map from top/bird-eye view.

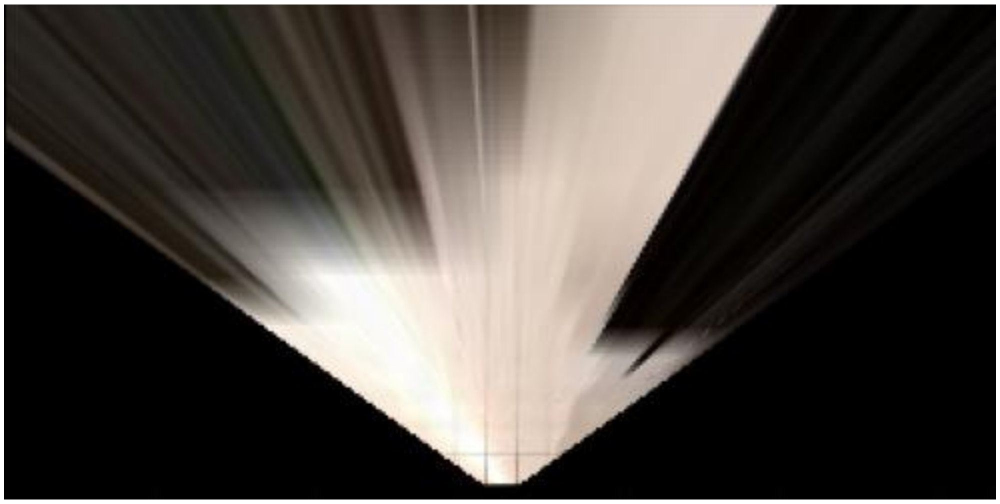
```
def perspect_transform(img, src, dst):

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

    return warped

dst_s = 5
bottom_offset = 5

source = np.float32([[14, 140],
                     [303, 140],
                     [200, 96],
                     [118, 96]])

destination = np.float32([[image.shape[1] / 2 - dst_s, image.shape[0] - bottom_offset],
                         [image.shape[1] / 2 + dst_s, image.shape[0] - bottom_offset],
                         [image.shape[1] / 2 + dst_s, image.shape[0] - 2*dst_s - bottom_offset],
                         [image.shape[1] / 2 - dst_s, image.shape[0] - 2*dst_s - bottom_offset]])

warped = perspect_transform(image, source, destination)
```

the following image has a grid where each square is one meter by one meter.



The warping function allows us to view the 1x1 meter in 10x10 pixels(2*dst_s) from a bird’s eye perspective.

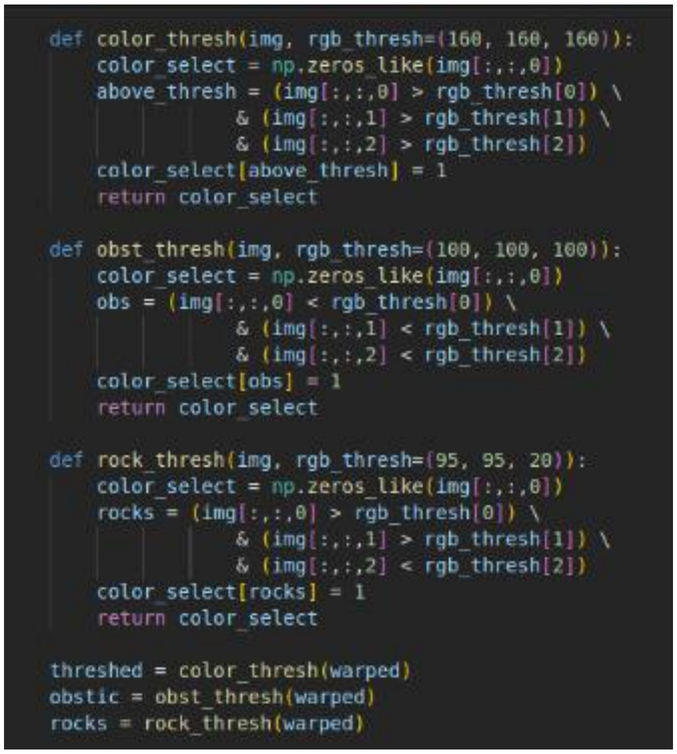
The source array shown above was the result from locating the coordinates of the corner pixels of the 1x1 meter grid.



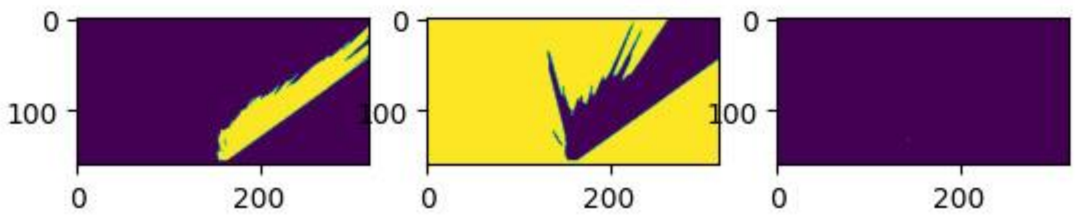
Applying Threshold:

it is time to apply it so that we can find navigable path by differentiating between terrain, rocks, and obstacles. How?

Each image is made up of 3 channels each assigned to a different color (RGB).So, when different samples were placed in color picker it was found that the following thresholds were the most appropriate options for each of their cases.



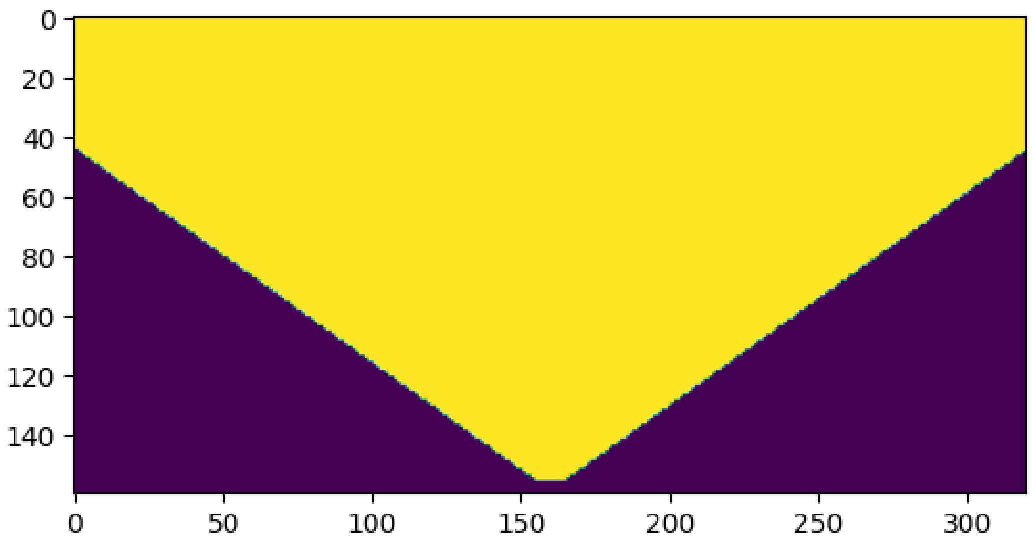
```
def color_thresh(img, rgb_thresh=(160, 160, 160)):
    color_select = np.zeros_like(img[:,:,0])
    above_thresh = (img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,1] > rgb_thresh[1]) \
                & (img[:,:,2] > rgb_thresh[2])
    color_select[above_thresh] = 1
    return color_select

def obst_thresh(img, rgb_thresh=(100, 100, 100)):
    color_select = np.zeros_like(img[:,:,0])
    obs = (img[:,:,0] < rgb_thresh[0]) \
                & (img[:,:,1] < rgb_thresh[1]) \
                & (img[:,:,2] < rgb_thresh[2])
    color_select[obs] = 1
    return color_select

def rock_thresh(img, rgb_thresh=(95, 95, 20)):
    color_select = np.zeros_like(img[:,:,0])
    rocks = (img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,1] > rgb_thresh[1]) \
                & (img[:,:,2] < rgb_thresh[2])
    color_select[rocks] = 1
    return color_select

threshed = color_thresh(warped)
obstic = obst_thresh(warped)
rocks = rock_thresh(warped)
```



Creating a mask:

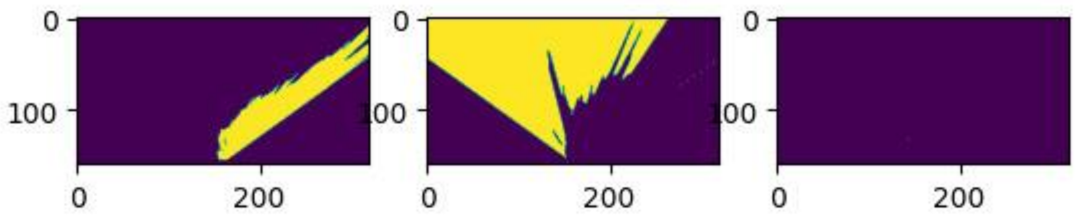
After thresholding the image, we saw that we need to increase the accuracy of thresholding for obstacles, so the following function implements a mask that does exactly that.

```
mask = np.ones_like(image[:,:,0], np.uint8)
mask = perspect_transform(mask, source, destination)
```



After creating the mask, we applied the following function to achieve a more realistic look for the warped thresholder obstacle image.

obstic = cv2.bitwise_and(obst_thresh(warped), mask)

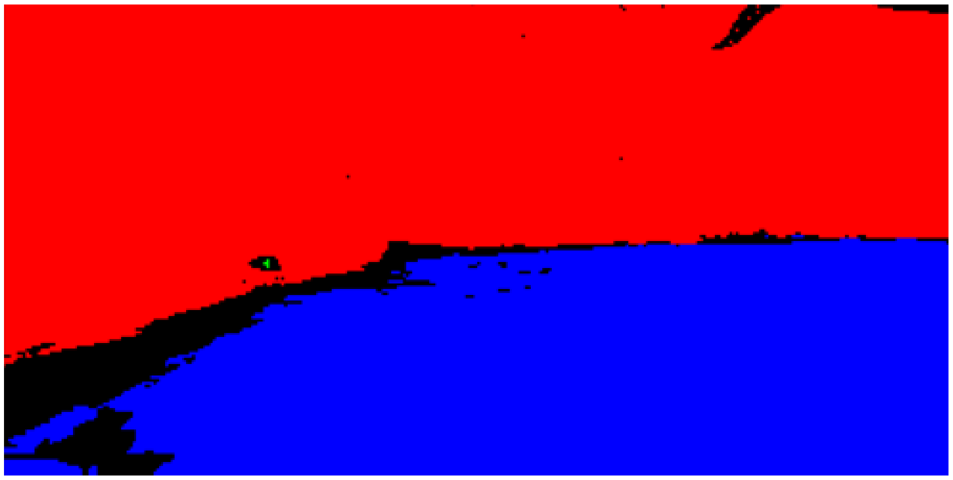


Applying threshold to visual image:

To improve visualization of the visual image we need to apply the threshold on it.

```
vision_image[:,:,2] = color_thresh(image)*255
vision_image[:,:,0] = obst_thresh(image)*255
vision_image[:,:,1] = rock_thresh(image)*255
```

The following image is the result after improving visualization:

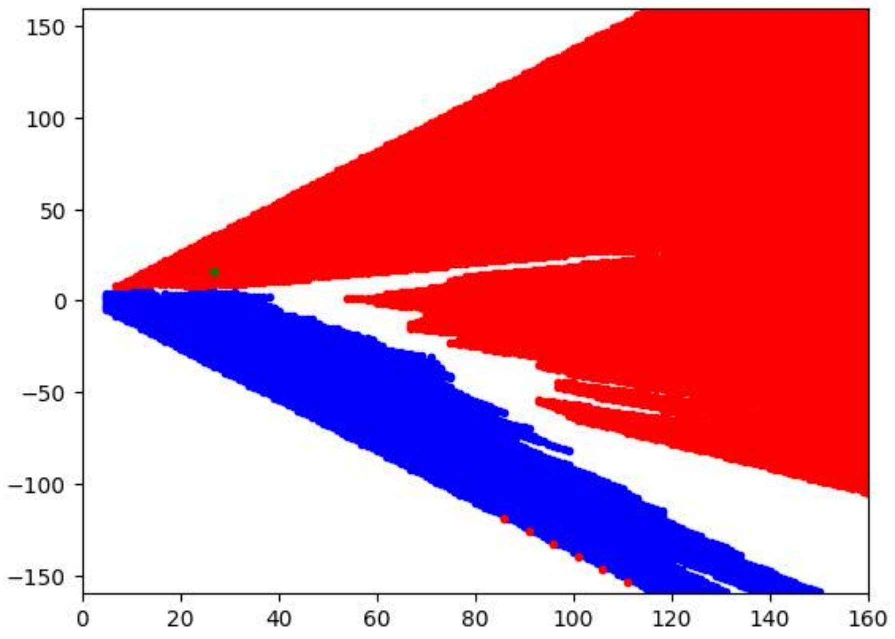


Converting map image pixel values to rover-centric coordinates:

The following function allows us to get the coordinates of each pixel relative to the rover's position (rover is at (0,0)).

```
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 = -(ypos - binary_img.shape[0]).astype(np.float)
    y_pixel = -(xpos - binary_img.shape[1]/2 ).astype(np.float)
    return x_pixel, y_pixel

xp, yp = rover_coords(threshed)
oxp, oyp = rover_coords(obstic)
rxp, ryp = rover_coords(rocks)
```

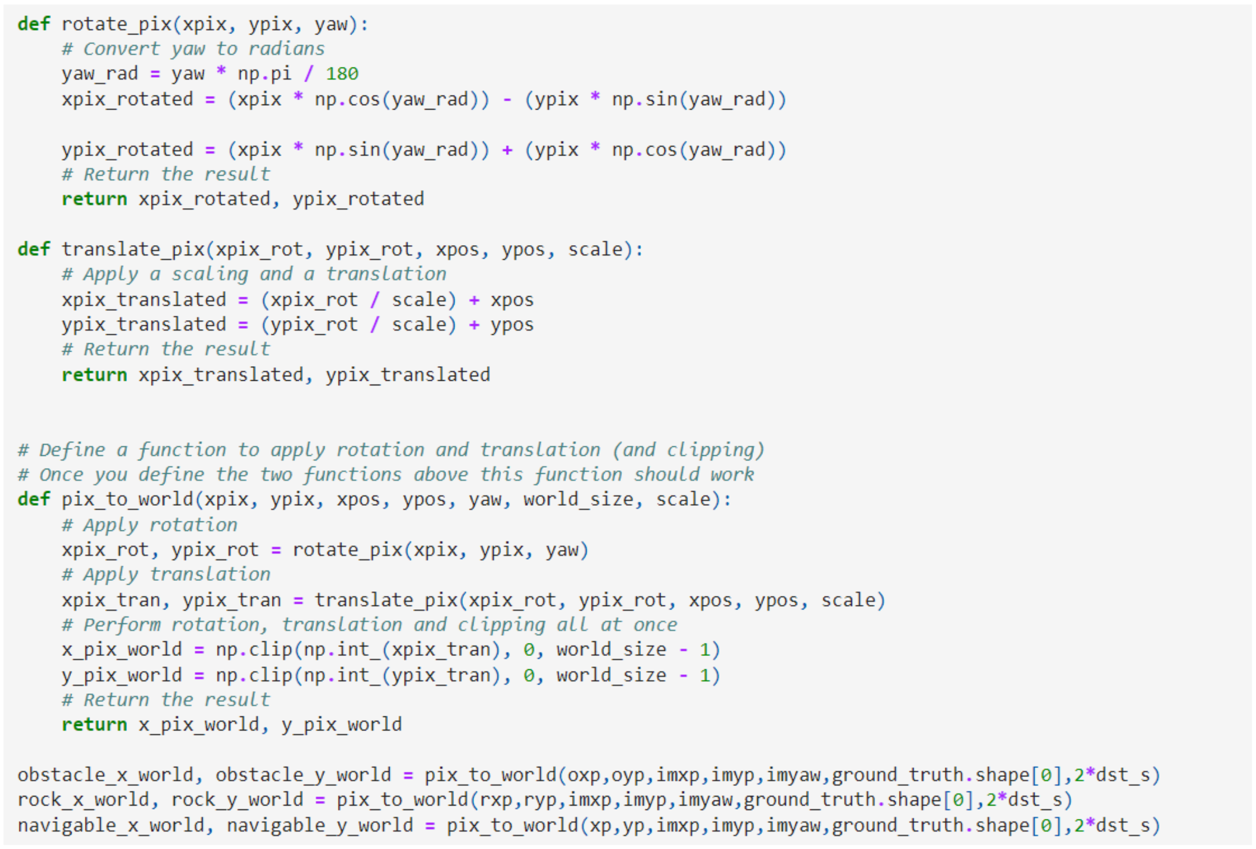
The following image shows the pixels when plotted on a graph.



Converting rover-centric pixel values to world coordinates:

the following functions allow us to rotate the rover-centric coordinates based on the rover's yaw and scale down the coordinates so that each 1x1 meter is represented in one pixel.

This is to show the gained information from the thresholds into the world map,

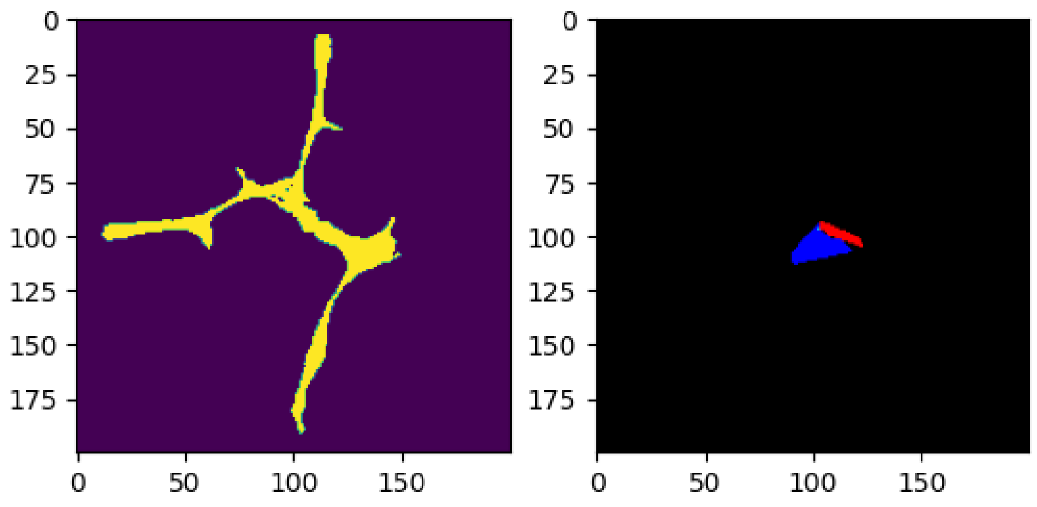


```
def rotate_pix(xpix, ypix, yaw):
    # Convert yaw to radians
    yaw_rad = yaw * np.pi / 180
    xpix_rotated = (xpix * np.cos(yaw_rad)) - (ypix * np.sin(yaw_rad))

    ypix_rotated = (xpix * np.sin(yaw_rad)) + (ypix * np.cos(yaw_rad))
    # Return the result
    return xpix_rotated, ypix_rotated

def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # 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)
    # Perform rotation, translation and clipping all at once
    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

obstacle_x_world, obstacle_y_world = pix_to_world(oxp,oyp,imxp,imyp,imyaw,ground_truth.shape[0],2*dst_s)
rock_x_world, rock_y_world = pix_to_world(rxp,ryp,imxp,imyp,imyaw,ground_truth.shape[0],2*dst_s)
navigable_x_world, navigable_y_world = pix_to_world(xp,yp,imxp,imyp,imyaw,ground_truth.shape[0],2*dst_s)
```

Updating Rover World Map:

We used the previous coordinate from the previous step to plot the gained information onto the rover's world map.

```
ground_truth[obstacle_y_world, obstacle_x_world, 2] = 255

ground_truth[rock_y_world, rock_x_world, 1] = 255

ground_truth[navigable_y_world, navigable_x_world, 0] = 255
ground_truth[navigable_y_world, navigable_x_world, 2] = 0

orgmap = mpimg.imread('../calibration_images/map_bw.png')
```

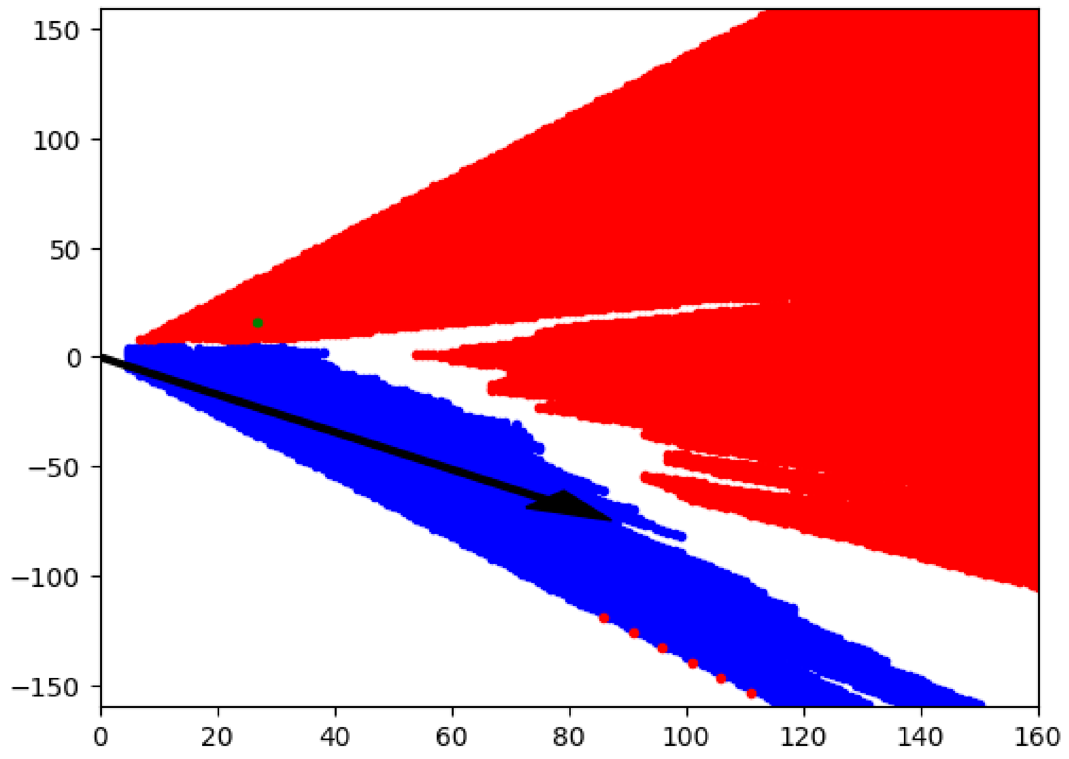


Moving the rover:

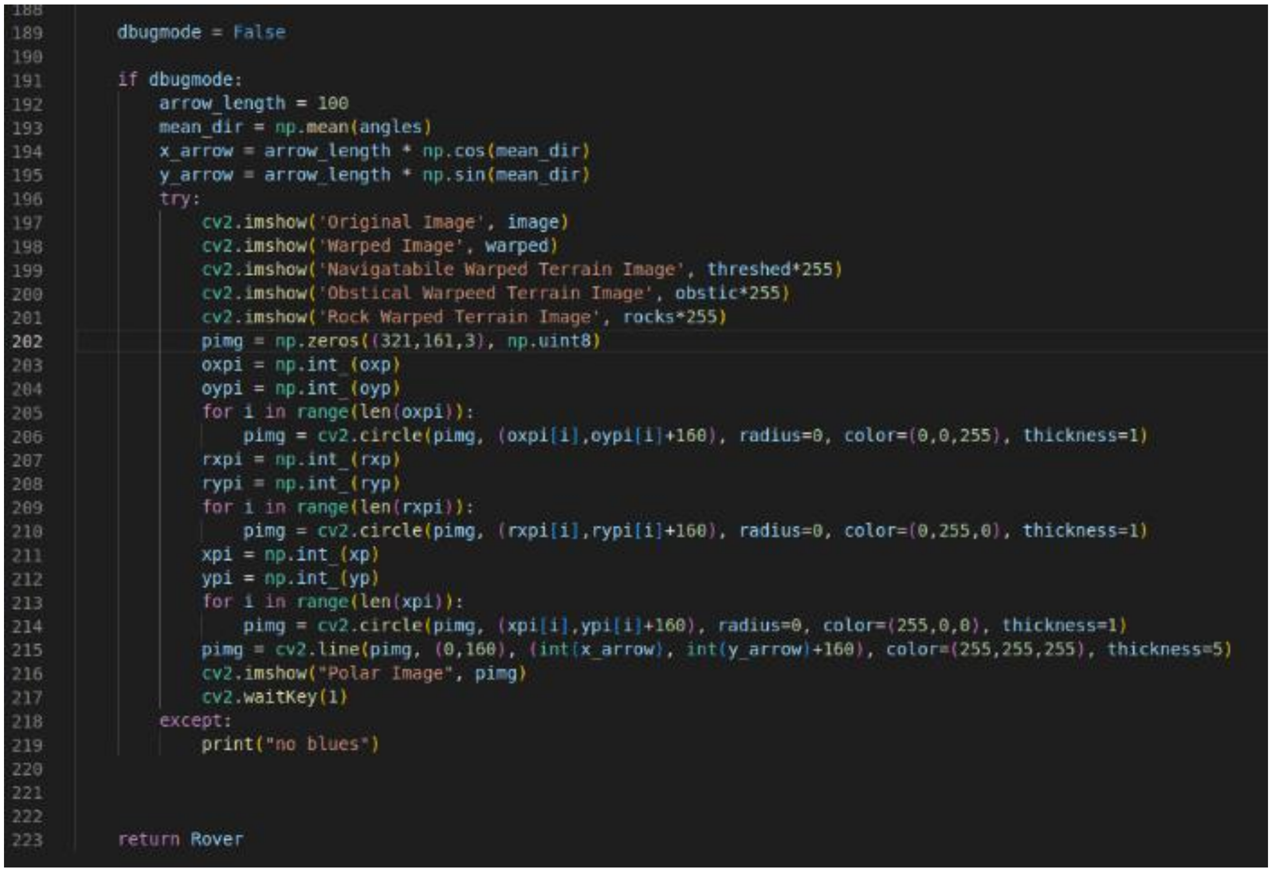
The navigable pixels are all converted from x, y to polar coordinates so that we have an angle for each pixel. Those angles are then passed to the rover state and returned from the function.

What happens is that the rover moves in the mean direction of those angles, as shown below:

```
def to_polar_coords(x_pixel, y_pixel):
    # Convert (x_pixel, y_pixel) to (distance, angle)
    # in polar coordinates in rover space
    # Calculate distance to each pixel
    dist = np.sqrt(x_pixel**2 + y_pixel**2)
    # Calculate angle away from vertical for each pixel
    angles = np.arctan2(y_pixel, x_pixel)
    return dist, angles

dist, angles = to_polar_coords(xp, yp)
```



Debugging mode:



```python
    dbugmode = False

    if dbugmode:
        arrow_length = 100
        mean_dir = np.mean(angles)
        x_arrow = arrow_length * np.cos(mean_dir)
        y_arrow = arrow_length * np.sin(mean_dir)
        try:
            cv2.imshow('Original Image', image)
            cv2.imshow('Warped Image', warped)
            cv2.imshow('Navigatabile Warped Terrain Image', threshed*255)
            cv2.imshow('Obstical Warpeed Terrain Image', obstic*255)
            cv2.imshow('Rock Warped Terrain Image', rocks*255)
            pimg = np.zeros((321,161,3), np.uint8)
            oxpi = np.int_(oxp)
            oypi = np.int_(oyp)
            for i in range(len(oxpi)):
                pimg = cv2.circle(pimg, (oxpi[i],oypi[i]+160), radius=0, color=(0,0,255), thickness=1)
            rxpi = np.int_(rxp)
            rypi = np.int_(ryp)
            for i in range(len(rxpi)):
                pimg = cv2.circle(pimg, (rxpi[i],rypi[i]+160), radius=0, color=(0,255,0), thickness=1)
            xpi = np.int_(xp)
            ypi = np.int_(yp)
            for i in range(len(xpi)):
                pimg = cv2.circle(pimg, (xpi[i],ypi[i]+160), radius=0, color=(255,0,0), thickness=1)
            pimg = cv2.line(pimg, (0,160), (int(x_arrow), int(y_arrow)+160), color=(255,255,255), thickness=5)
            cv2.imshow("Polar Image", pimg)
            cv2.waitKey(1)
        except:
            print("no blues")



    return Rover
```

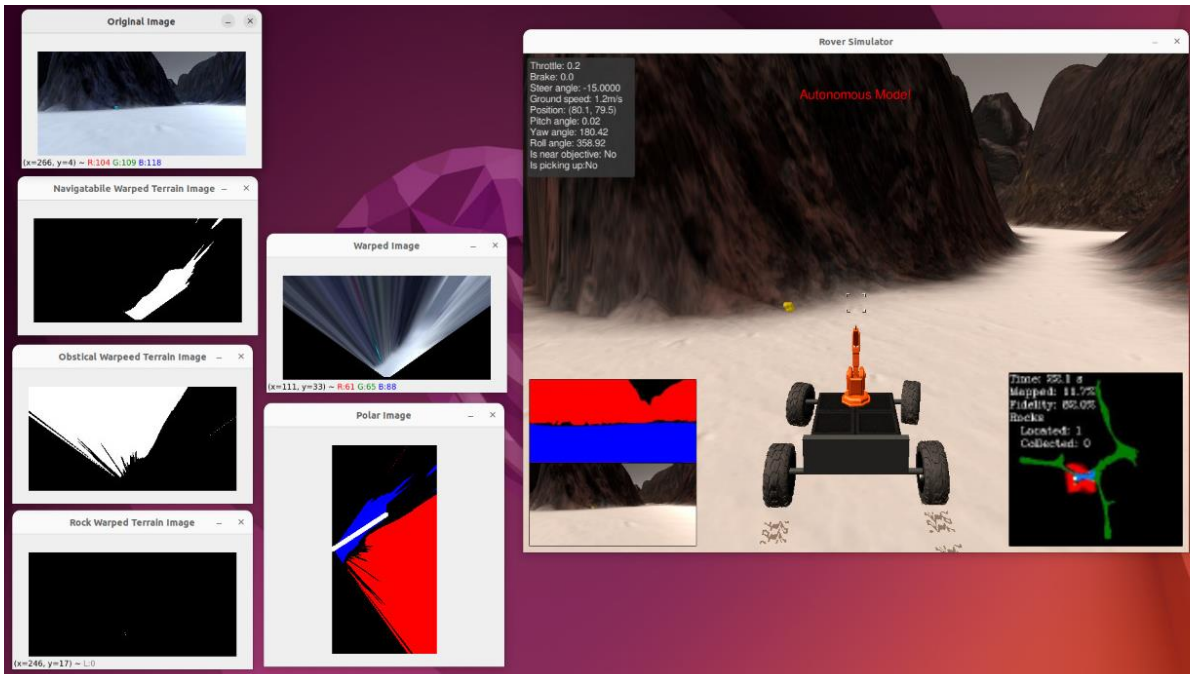
The debugging mode is toggled using the variable dbugmode which is found in the picture above (perception.py).

Debugging mode uses opencv to show windows containing :

- The original image received
- The warped image
- Navigatable terrain thresholder image
- Obstacle thresholder image
- Rock threholder image
- Polar image

The polar image is drawn on a discrete 2-D plane in opencv this is made by drawing circles with zero radius and one pixel thickness to show the rover centric coordinates of each of the discovered areas /information. An arrow is then drawn in the mean direction of the angles of the navigatable terrain pixels.This is by converting this mean angle back to x,y coordinates.

Shown below is the view of the simulator working while debugging mode is on.

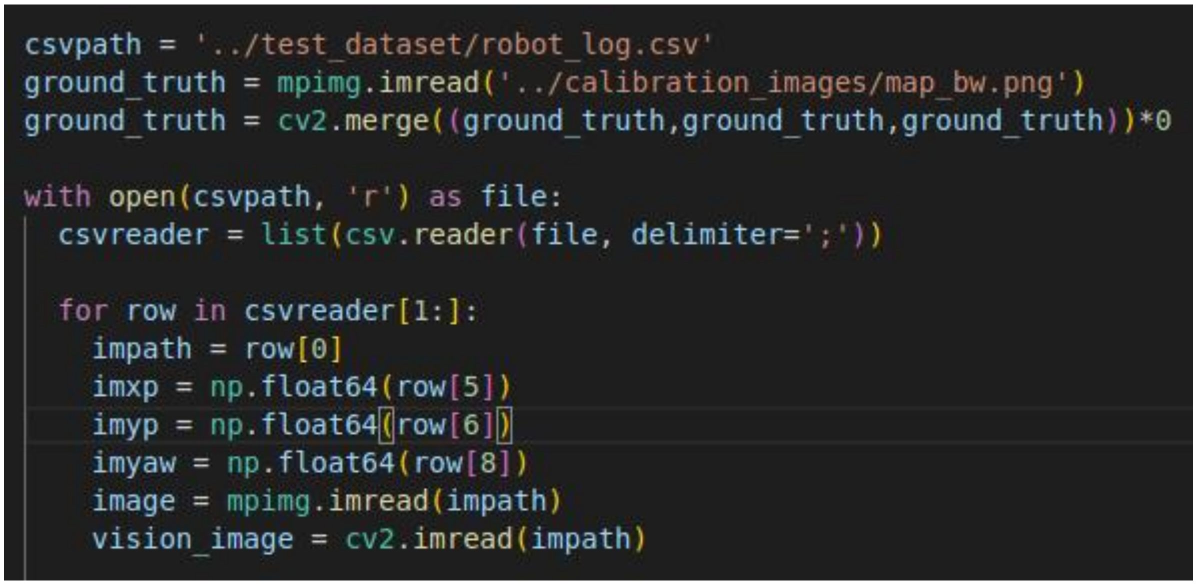

Original Image
(x=266, y=4) ~ R:104 G:109 B:118
Navigatabile Warped Terrain Image
Warped Image
(x=111, y=33) ~ R:61 G:65 B:88
Obstical Warpeed Terrain Image
Polar Image
Rock Warped Terrain Image
(x=246, y=17) ~ L:0
Rover Simulator
Autonomous Model

```
Throttle: 0.2
Brake: 0.0
Steer angle: -15.0000
Ground speed: 1.2m/s
Position: (80.1, 79.5)
Pitch angle: 0.02
Yaw angle: 180.42
Roll angle: 358.92
Is near objective: No
Is picking up:No
```

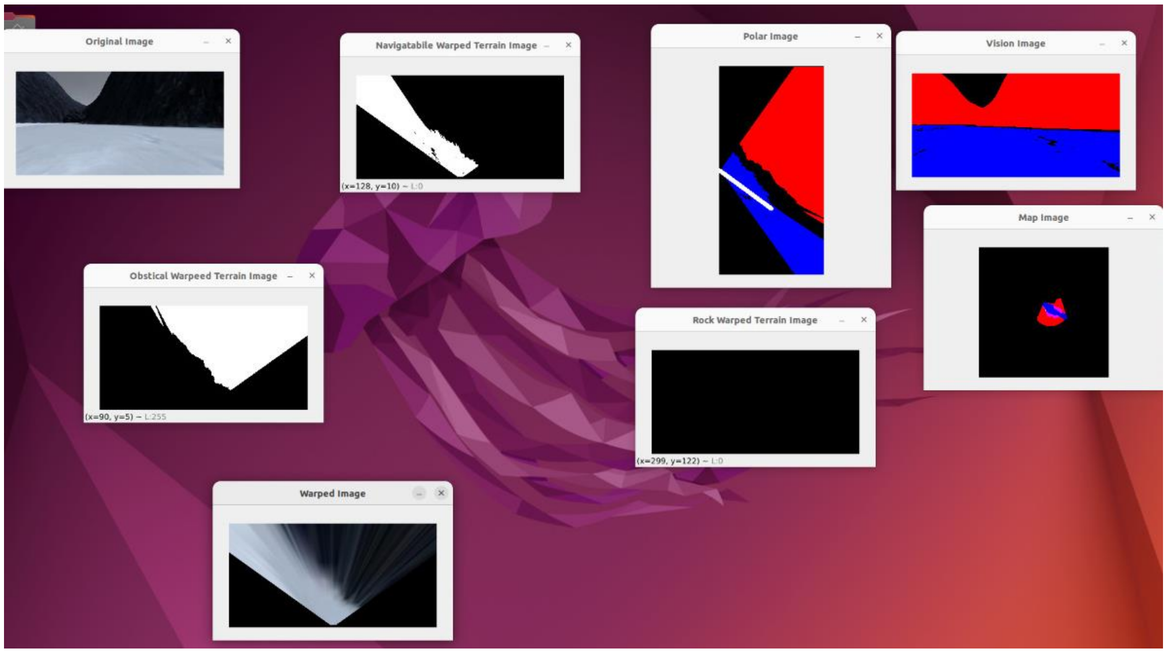
```
Time: 22.1 s
Mapped: 11.7%
Fidelity: 82.0%
Rocks
  Located: 1
  Collected: 0
```

Recorded Runs mode:

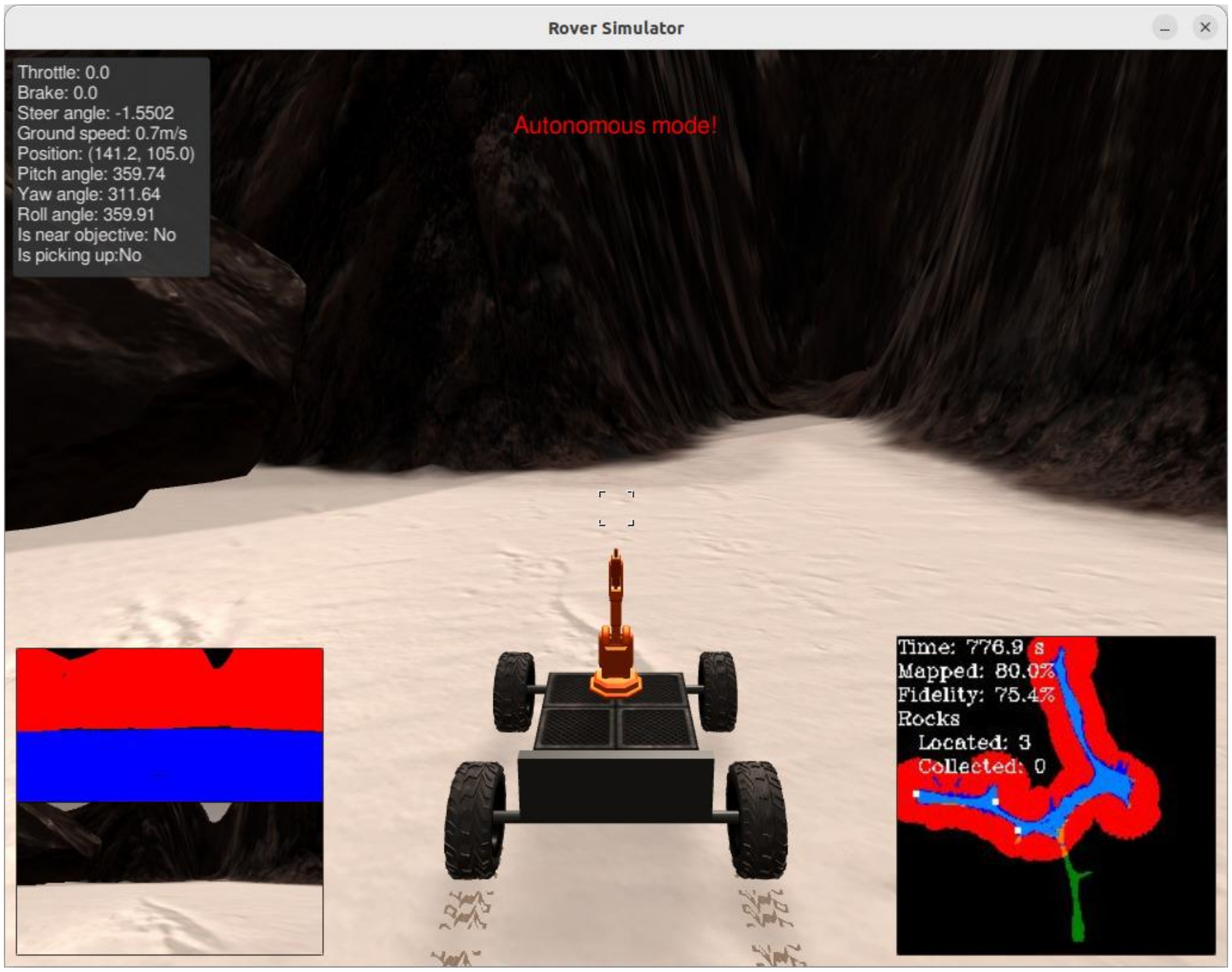
This mode is executed through the recordedDA.py file where the feed is collected via the csv file outputted from the recording. All other functionality is the same as above.



```python
csvpath = '../test_dataset/robot_log.csv'
ground_truth = mpimg.imread('../calibration_images/map_bw.png')
ground_truth = cv2.merge((ground_truth,ground_truth,ground_truth))*0

with open(csvpath, 'r') as file:
  csvreader = list(csv.reader(file, delimiter=';'))

  for row in csvreader[1:]:
    impath = row[0]
    imxp = np.float64(row[5])
    imyp = np.float64(row[6])
    imyaw = np.float64(row[8])
    image = mpimg.imread(impath)
    vision_image = cv2.imread(impath)
```

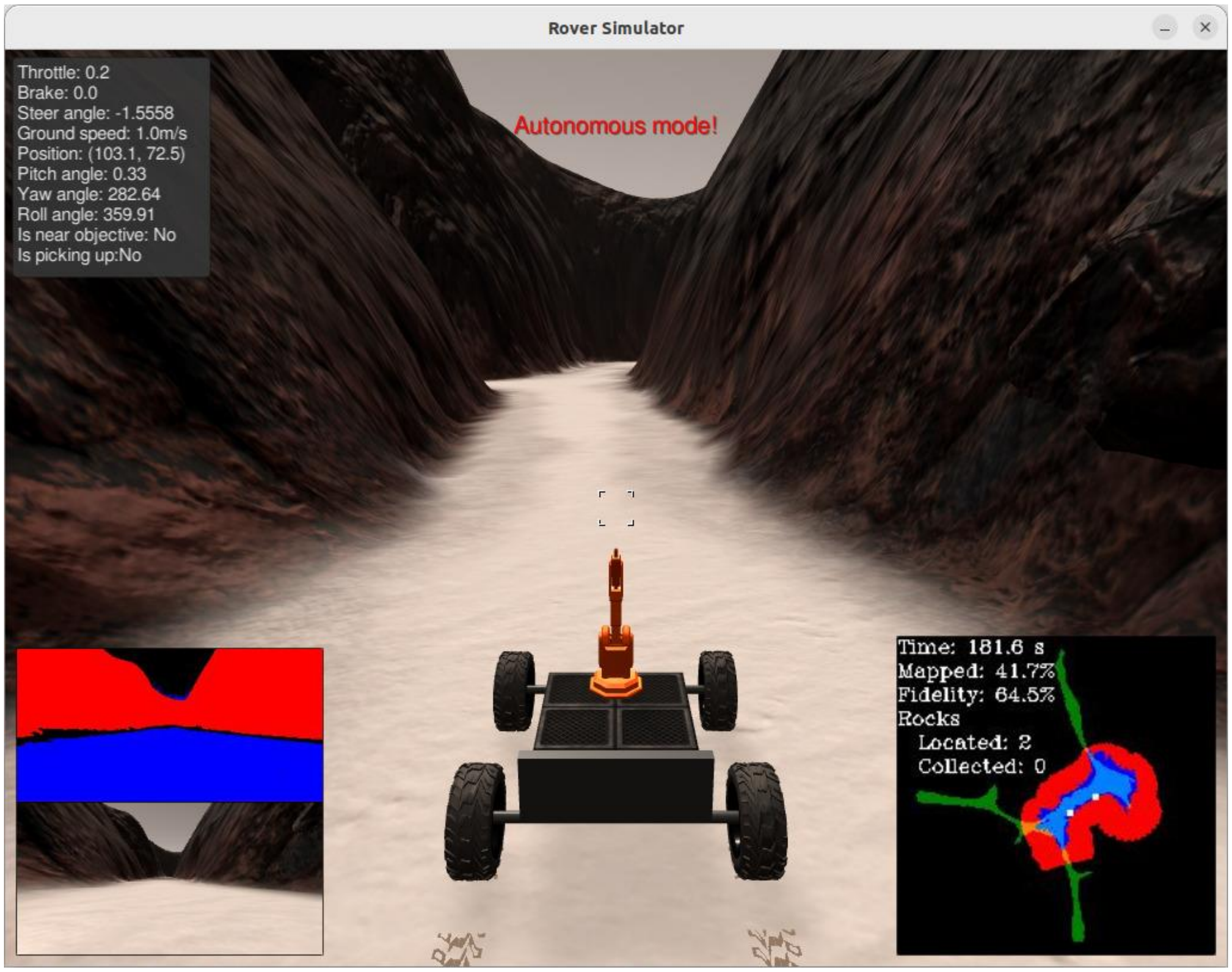


Screenshot from simulation:



```
Throttle: 0.0
Brake: 0.0
Steer angle: -1.5502
Ground speed: 0.7m/s
Position: (141.2, 105.0)
Pitch angle: 359.74
Yaw angle: 311.64
Roll angle: 359.91
Is near objective: No
Is picking up:No
```

```
Time: 776.9 s
Mapped: 80.0%
Fidelity: 75.4%
Rocks
  Located: 3
  Collected: 0
```



```
Throttle: 0.2
Brake: 0.0
Steer angle: -1.5558
Ground speed: 1.0m/s
Position: (103.1, 72.5)
Pitch angle: 0.33
Yaw angle: 282.64
Roll angle: 359.91
Is near objective: No
Is picking up:No
```

```
Time: 181.6 s
Mapped: 41.7%
Fidelity: 64.5%
Rocks
  Located: 2
  Collected: 0
```