Project Report

Phase 2

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

Phase 1 Recollection:

Packages:

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

Input Image:

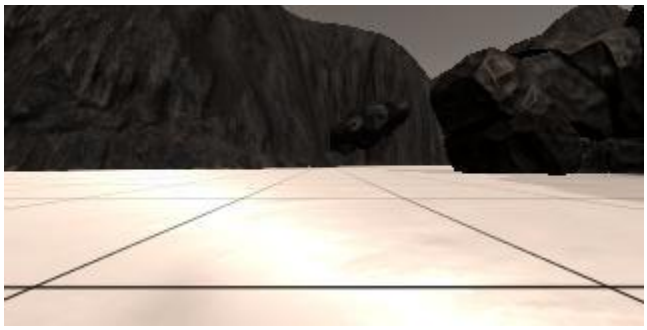
The first thing we have is our input images which we will get from the rover camera (Rover.img), which we assign to the variable (image). This Image looks like the following:

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.

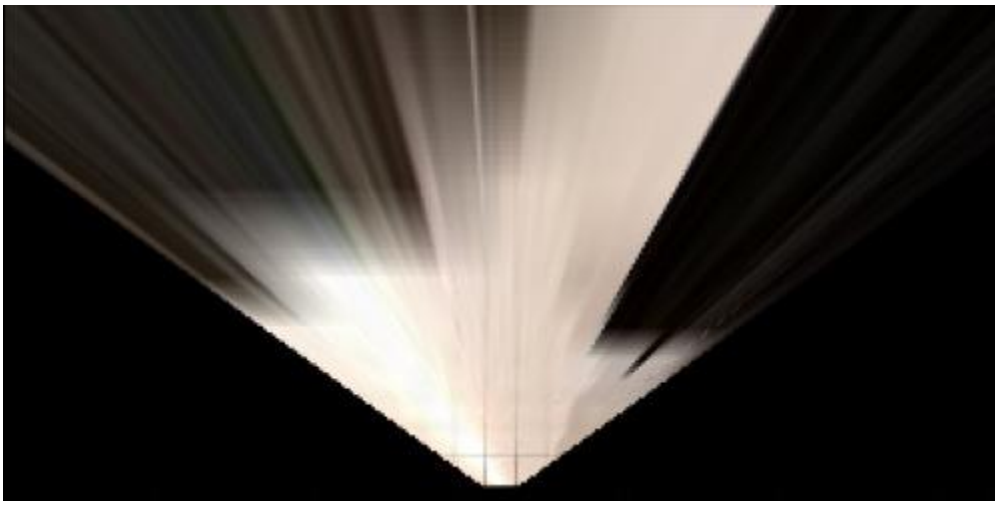
```
def perspect_transform(img, src, dst):

    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.

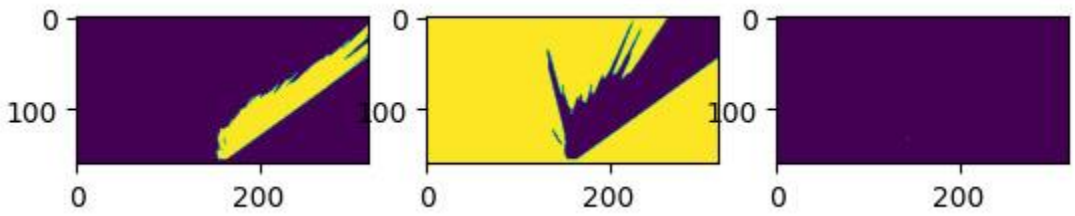


Applying Threshold:

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)
```

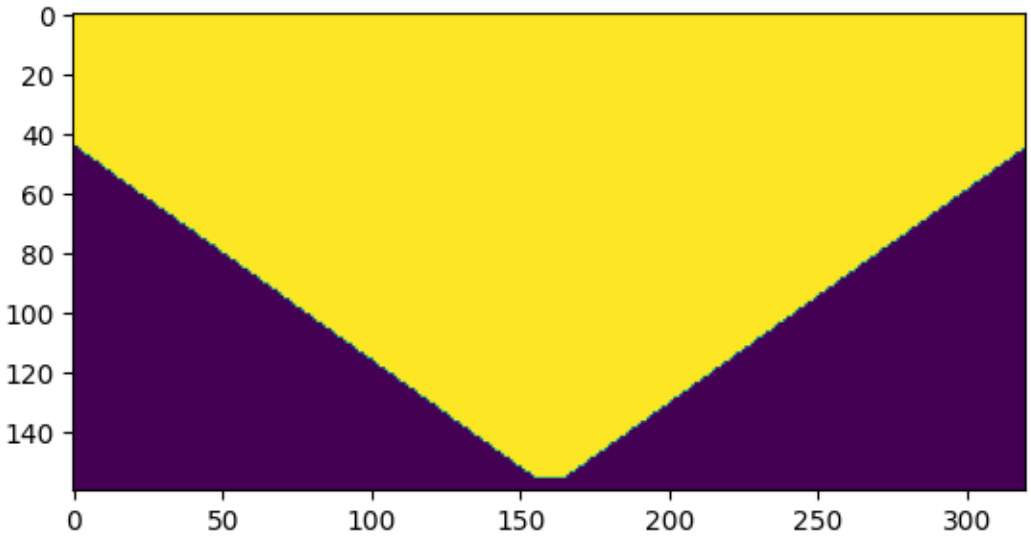


The rock_thresh function was updated for Phase 2 so that the values are changed to 85,85,30

Creating a mask:

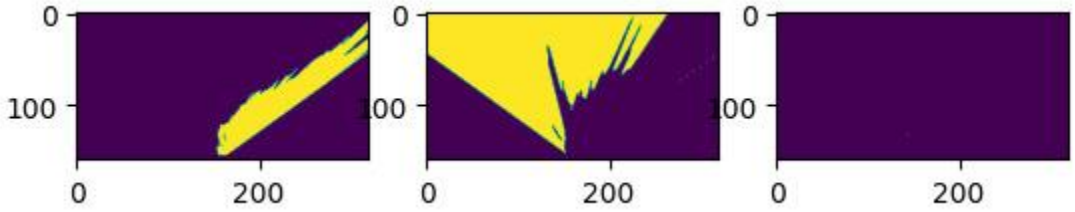
After thresholding the image, we saw that we need to increase the accuracy of thresholding for obstacles, so the following function implements a mask that does exactly that.

```
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)

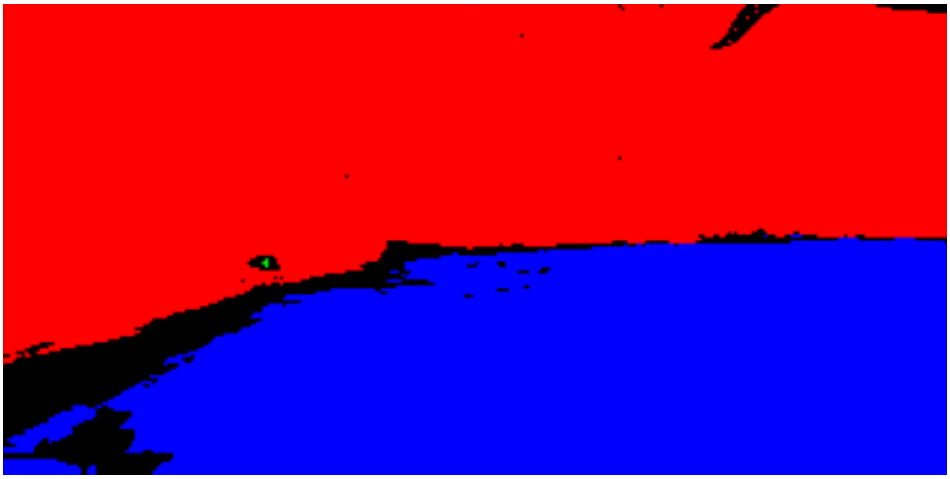


Applying threshold to visual image:

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:

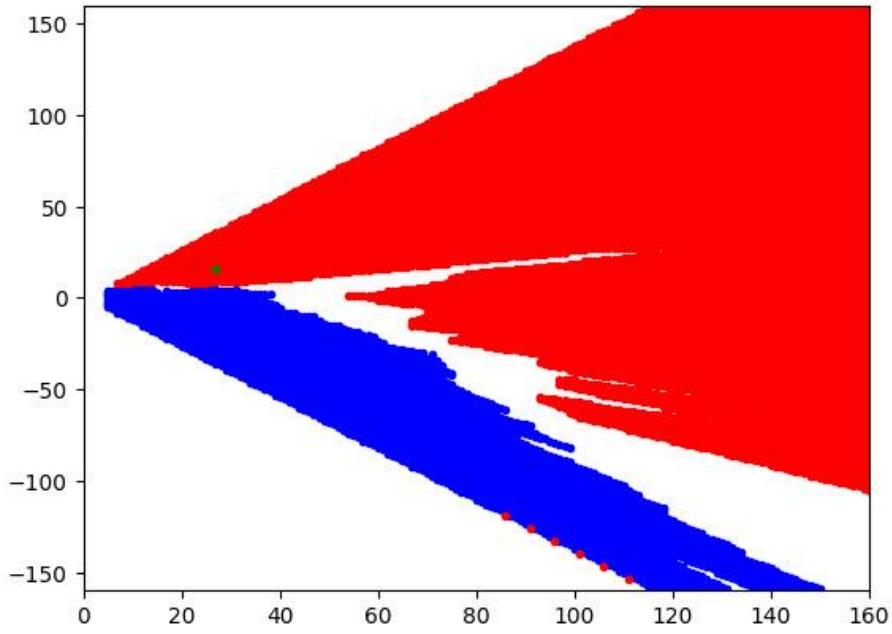


Converting map image pixel values to rover-centric coordinates:

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.

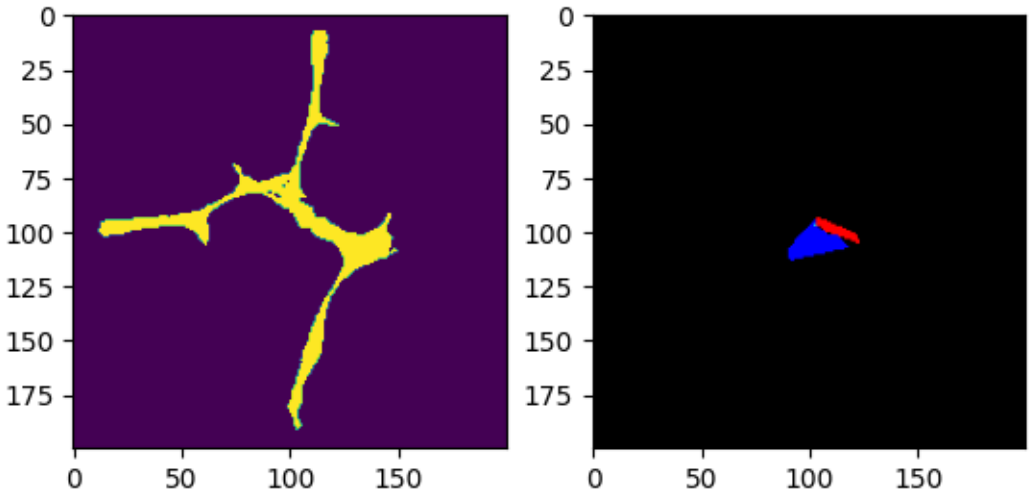
This is to show the gained information from the thresholds into the world map,

```
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:

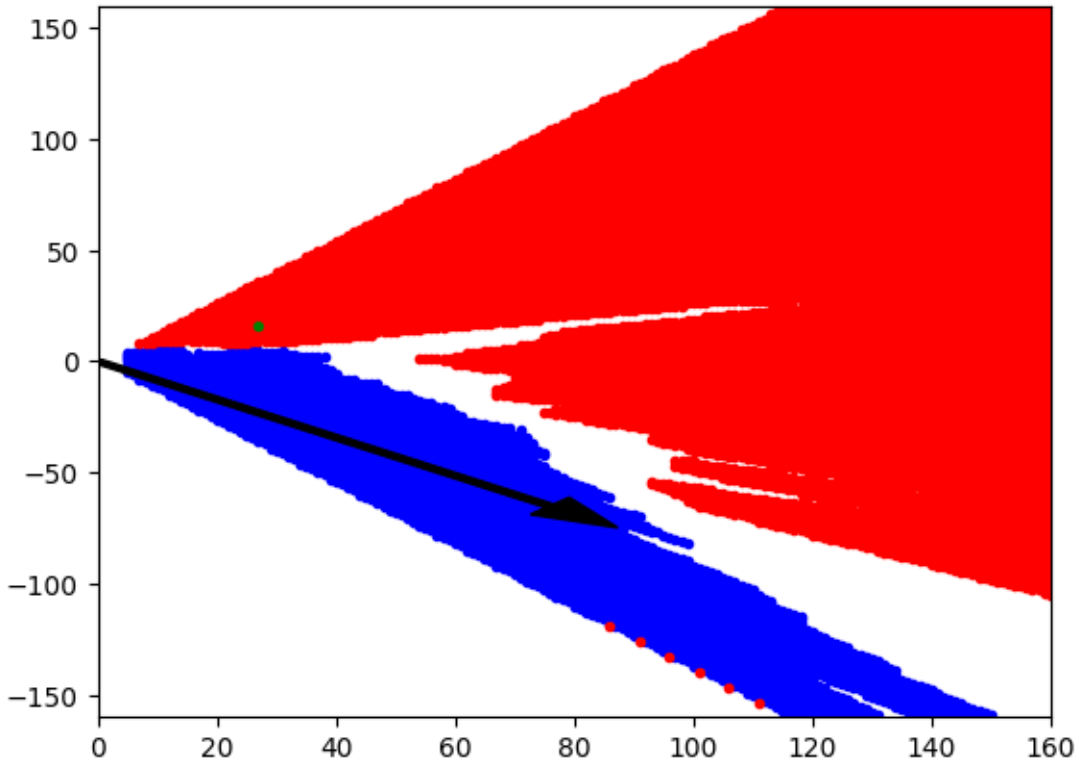
We used the previous coordinate from the previous step to plot the gained information onto the rover's world map.

```
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')
```



Rotating Image and drawing arrow of direction of movement for Debugging:

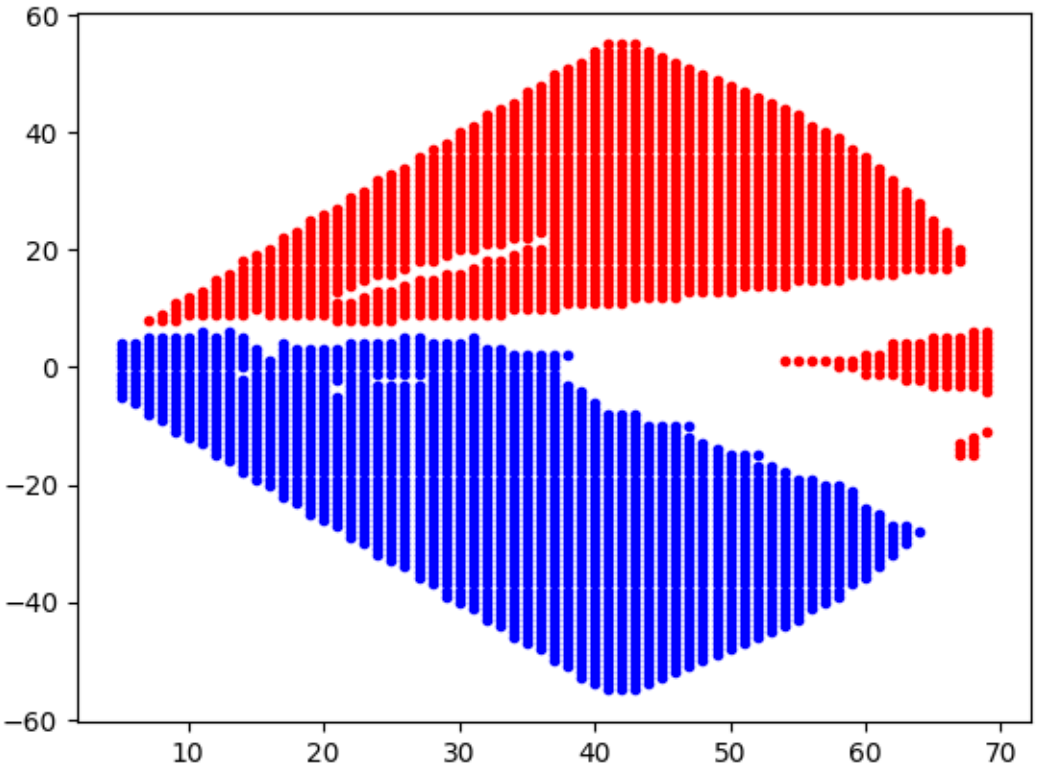
```
def to_polar_coords(x_pixel, y_pixel):
    # Convert (x_pixel, y_pixel) to (distance, angle)
    # in polar coordinates in rover space
    # Calculate distance to each pixel
    dist = np.sqrt(x_pixel**2 + y_pixel**2)
    # Calculate angle away from vertical for each pixel
    angles = np.arctan2(y_pixel,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

dist, angles = to_polar_coords(xp, yp)
arrow_length = 100
mean_dir = np.mean(angles)
x_arrow = arrow_length * np.cos(mean_dir)
y_arrow = arrow_length * np.sin(mean_dir)
plt.plot(xp, yp, '.', color='blue')
plt.plot(oxp, oyp, '.', color='red')
plt.plot(rxp, ryp, '.', color='green')
plt.ylim(-160, 160)
plt.xlim(0, 160)
arrow_length = 100
x_arrow = arrow_length * np.cos(mean_dir)
y_arrow = arrow_length * np.sin(mean_dir)
plt.arrow(0, 0, x_arrow, y_arrow, color='black', zorder=2, head_width=10, width=2)
plt.show()
```



Phase 2:

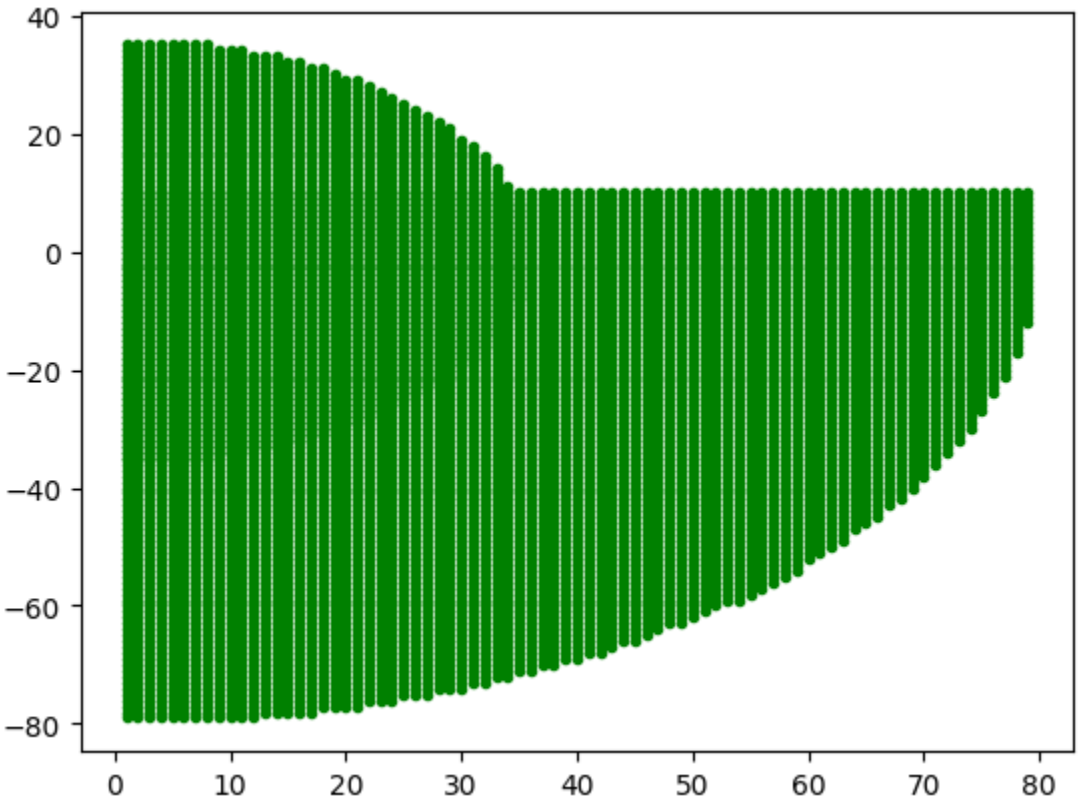
A big change that happened is that we don't take the entire warped image as granted so we set a perimeter of 70 pixels around the rover that we use only.

```
visdistance = np.sqrt(xp ** 2 + yp ** 2)
xnp = xp[visdistance<70]
ynp = yp[visdistance<70]

visdistance = np.sqrt(oxp ** 2 + oyp ** 2)
oxnp = oxp[visdistance<70]
oynp = oyp[visdistance<70]
```



We also use only a masked version of the rock threshold for detecting rocks at first This detects rocks on the right side of the rover which are 80 pixels away as well as any rock that are 36 pixels away. However, after detection and being locked in the rover uses the entire threshold of the rocks as in phase 1. This step is to ensure that the rover stays hugging the right wall as much as possible.

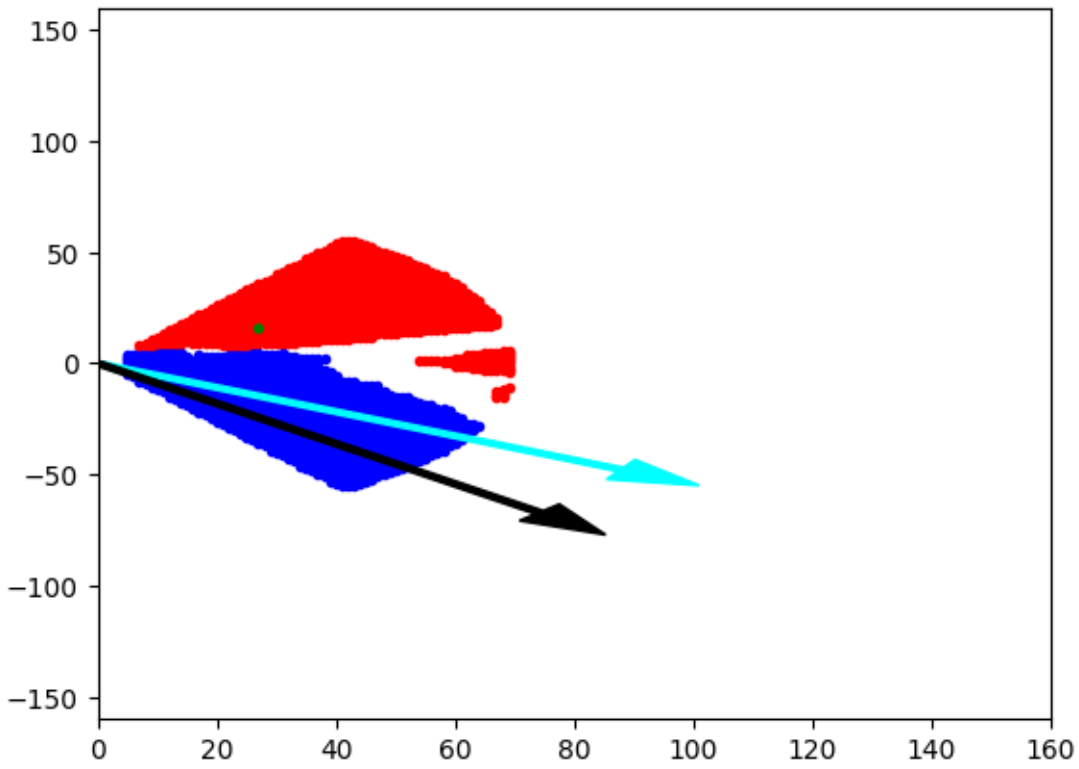
```
rvisdistance = np.sqrt(rxp ** 2 + ryp ** 2)
rxdp1 = rxp[rvisdistance<36]
rydp1 = ryp[rvisdistance<36]

samplepic[:,:150] = 0
rxp, ryp = rover_coords(samplepic)
rvisdistance = np.sqrt(rxp ** 2 + ryp ** 2)
rxdp2 = rxp[rvisdistance<80]
rydp2 = ryp[rvisdistance<80]
```



Hugging the Wall:

For the rover to actually hug the wall we used the standard deviation of the angles of navigable area. This is because whenever the std is higher that means the road that navigable has become wider allowing us for a sharper turn until we reach a wall to hug.

```
rxp, ryp = rover_coords(rocks)

dist, angles = to_polar_coords(xnp, ynp)
arrow_length = 100
mean_dir = np.mean(angles)
stdev = 0.8 * np.std(angles * 180/np.pi) * (np.pi/180)
x_arrow = arrow_length * np.cos(mean_dir)
y_arrow = arrow_length * np.sin(mean_dir)
x_new_arrow = arrow_length * np.cos(mean_dir - stdev)
y_new_arrow = arrow_length * np.sin(mean_dir - stdev)
plt.plot(xnp, ynp, '.', color='blue')
plt.plot(oxnp, oynp, '.', color='red')
plt.plot(rxp, ryp, '.', color='green')
plt.ylim(-160, 160)
plt.xlim(0, 160)
arrow_length = 100
x_arrow = arrow_length * np.cos(mean_dir)
y_arrow = arrow_length * np.sin(mean_dir)
plt.arrow(0, 0, x_arrow, y_arrow, color='cyan', zorder=2, head_width=10, width=2)
plt.arrow(0, 0, x_new_arrow, y_new_arrow, color='black', zorder=2, head_width=10, width=2)
plt.show()
```



This is however overwritten in 3 situations:

1) Whenever there is an obstacle which is 37 pixels away with an angle between 16 and -16 degrees, in which case we move with the original mean angle (the cyan arrow).
2) Whenever a rock is located using the mask explained above, in which case the rover enters a 'found' mode and acts accordingly
3) Whenever there's an obstacle within a perimeter of 13 pixels near the rover as it comes to a complete stop

When a rock enters the mask perimeter:

The Rover first enters a 'found' state in which it brakes completely. Then it moves into a 'locked in' state in which it keeps rotating until the rock becomes within 20 angles of steering of the rover. The next step is a 'pick' state in which the rover moves towards the mean angles of the rock pixels. The when the rocks enter a mean distance of 9 pixels the rover starts 'pickup' mode. In addition to that, if the rover loses sight of the rock while in 'pick' mode it returns to found mode to search for it again. This can be seen clearly in the decision.py file:

```
# Checking if any samples entered the vicinity masks for found
    if ((len(Rover.samples_angles) >= 1) or (len(Rover.samples_angles2) >= 13)) and (not
Rover.stuck) and Rover.mode != 'found' and Rover.mode != 'lockedin' and Rover.mode != 'pick'
and Rover.mode != 'pickup' and Rover.mode != 'stop':
        Rover.mode = 'found'
        print('FOUND')
        Rover.stuck_time = Rover.total_time
    # Braking entirely in this mode
    elif Rover.mode == 'found':
        Rover.send_pickup = False
        Rover.brake = 5
        Rover.throttle = 0
        if Rover.vel == 0 and (Rover.pitch < 0.5 or Rover.pitch > 359.5) and (Rover.roll < 0.5
or Rover.roll > 359.5):
            Rover.found_time = Rover.total_time
            Rover.mode = 'lockedin'
    # Searching for the sample and keeping it within 20 degrees
    elif Rover.mode == 'lockedin':
        if Rover.total_time - Rover.found_time <= 30:
            if  (len(Rover.samples_angles3) > 0):
                mangle = np.mean(Rover.samples_angles3 * 180/np.pi)
                if (mangle < 20) and (mangle > -20):
                    Rover.mode = 'pick'
                else:
                    Rover.brake = 0
                    Rover.steer = np.clip(mangle, -15, 15)
                    Rover.stuck_time = Rover.total_time
```

```
            else: # Sample is not found so keep turning left till it's found again
                Rover.brake = 0
                Rover.steer = 15
                Rover.stuck_time = Rover.total_time
        else: # If searching has been going on for 30 seconds or more with no results give up
            Rover.mode = 'stop'
        if np.mean(Rover.samples_dists3) < 9: # If samples are within 9 pixels go to pickup
            Rover.throttle = 0
            Rover.brake = Rover.brake_set
            Rover.steer = 0
            Rover.found_time = Rover.total_time
            Rover.mode = 'pickup'
    elif Rover.mode == 'pick' and (len(Rover.samples_angles3) > 0) and (not Rover.stuck):
        mangle = np.mean(Rover.samples_angles3 * 180/np.pi)
        if Rover.vel < Rover.max_vel: # Go towards rock
            # Set throttle value to throttle setting
            Rover.throttle = Rover.throttle_set
            Rover.brake = 0
            # Set steering to average angle clipped to the range +/- 15
            Rover.steer = np.clip(mangle, -15, 15)
        else: # Else coast
            Rover.throttle = 0
        if np.mean(Rover.samples_dists3) < 9: # If samples are within 9 pixels go to pickup
            Rover.throttle = 0
            Rover.brake = Rover.brake_set
            Rover.steer = 0
            Rover.found_time = Rover.total_time
            Rover.mode = 'pickup'
    elif Rover.mode == 'pick' and (len(Rover.samples_angles3) == 0) and (not Rover.stuck): #If
sample was lost search again
        Rover.mode = 'found'
    elif Rover.mode == 'pickup': # Break and start picking up
        if Rover.vel != 0:
            Rover.brake = Rover.brake_set
        elif (not Rover.picking_up) and (Rover.vel == 0):
            Rover.brake = Rover.brake_set
            Rover.stuck_time = Rover.total_time
            Rover.send_pickup = True
            Rover.mode = 'stop'
```

Stuck Handling:

The rover might get stuck in different terrain making it not able to move. This is handled using a stuck mode where the rover detects that it's velocity was less than 0.1 for four seconds it starts turning. This turning is always towards the left as we are hugging the right wall. The amount of turning also varies depending on the situation that the rover is in. Whether it's in the 'forward' mode or the 'pick' mode.

```
# Stuck condition if rover has almost 0 velocity for more than 4 seconds
    if Rover.vel <= 0.1 and (Rover.total_time - Rover.stuck_time > 4) and Rover.mode != 'lockedin':
        # Set mode to "stuck" and hit the brakes
        Rover.stuck = True
        Rover.throttle = 0
        # Set brake to stored brake value
        Rover.brake = Rover.brake_set
        Rover.steer = 0
        Rover.stuck_time = Rover.total_time
```

```
elif Rover.mode == 'pick' and Rover.stuck: # If stuck while heading towards rock
        # if 0.3 sec passed go back to previous mode
        if Rover.total_time - Rover.stuck_time > 0.3:
            # Set throttle back to stored value
            Rover.throttle = Rover.throttle_set
            Rover.brake = 0 # Release the brake
            Rover.stuck = False
        # Now we're stopped and we have vision data to see if there's a path forward
        else:
            Rover.throttle = 0
            Rover.brake = 0 # Release the brake to allow turning
            Rover.steer = 15
```

```
elif Rover.stuck: # If stuck while moving forward
            Rover.send_pickup = False
            # if 1 sec passed go back to previous mode
            if Rover.total_time - Rover.stuck_time > 1:
                # Set throttle back to stored value
                Rover.throttle = Rover.throttle_set
                Rover.brake = 0 # Release the brake
                Rover.stuck = False
            # Now we're stopped and we have vision data to see if there's a path forward
            else:
                Rover.throttle = 0
                Rover.brake = 0 # Release the brake to allow turning
                Rover.steer = 15
```

Hugging the wall code wise:

In the ‘forward’ mode this controls the steering angle:

```
if (np.min(np.absolute(Rover.obst_angles * 180/np.pi)) > 16): # If there are no obsticales within 16
degrees and 37 pixels away
        Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi)-(0.8 * np.std(Rover.nav_angles *
180/np.pi)), -15, 15)

else:
        Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)
```

So, it allows to switch to our previous mode of steering whenever there’s an obstacle within 16 pixels so that we can avoid crashing as much as possible.

Debugging mode:

The following snippet is altered in debugging mode showing 2 arrows instead of one and highlighting the one used in white while keeping the other one in cyan. The arrows are also completely replaced with a yellow arrow whenever a rock is in the processes of being picked up.

```
if (Rover.mode != 'found') and (Rover.mode != 'pick') and (Rover.mode != 'lockedin'):
    if np.min(np.absolute(oangles * 180/np.pi)) > 16:
        pimg = cv2.line(pimg, (0,160), (int(x_new_arrow), int(y_new_arrow)+160), color=(255,255,255),
thickness=5)
        pimg = cv2.line(pimg, (0,160), (int(x_arrow), int(y_arrow)+160), color=(255,255,0), thickness=5)
    else:
        pimg = cv2.line(pimg, (0,160), (int(x_new_arrow), int(y_new_arrow)+160), color=(255,255,0),
thickness=5)
        pimg = cv2.line(pimg, (0,160), (int(x_arrow), int(y_arrow)+160), color=(255,255,255), thickness=5)
else:
    if len(rangles3) > 0:
        pimg = cv2.line(pimg, (0,160), (int(xr_arrow), int(yr_arrow)+160), color=(0,255,255), thickness=5)
```

Recorded mode:

This can be used in the RDA.py file and it works just like the debugging mode.

Mapping Condition:

To increase the Rover's Fidelity, we gave it a list of conditions that need to be met for it to map. Those include:

1. The rover's pitch being under 0.2 or over 359.8
2. The rover's roll pitch being under 0.2 or over 359.8
3. The steering angle is within 11 degrees left or right
4. The rover's brake is released (brake == 0)
5. The rover is not picking up a rock

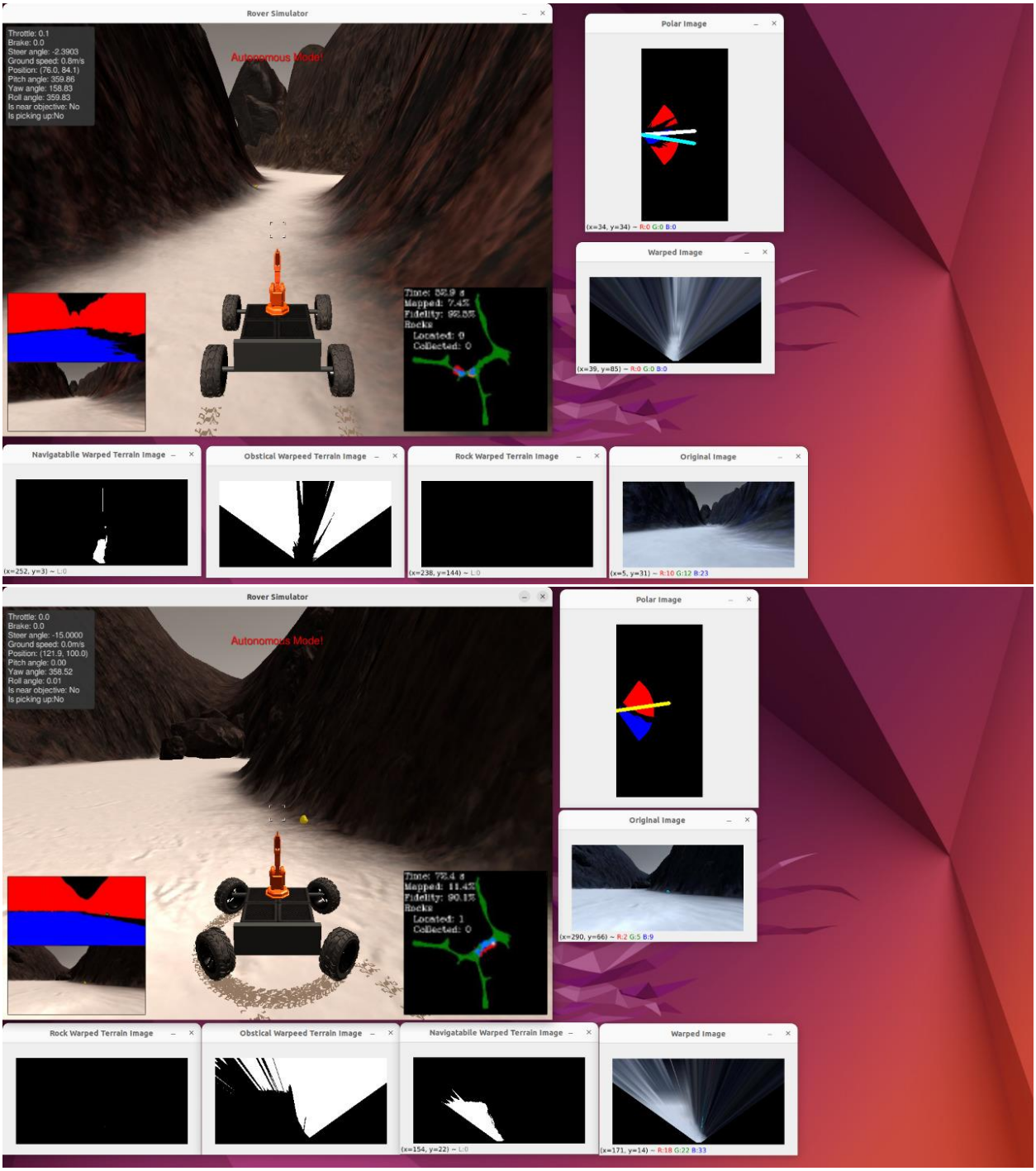
And all these conditions can be bypassed if these conditions were met:

1. The rover sees a pixel that it considers a sample
2. The rover's brake is released (brake == 0)
3. The rover is not picking up a rock

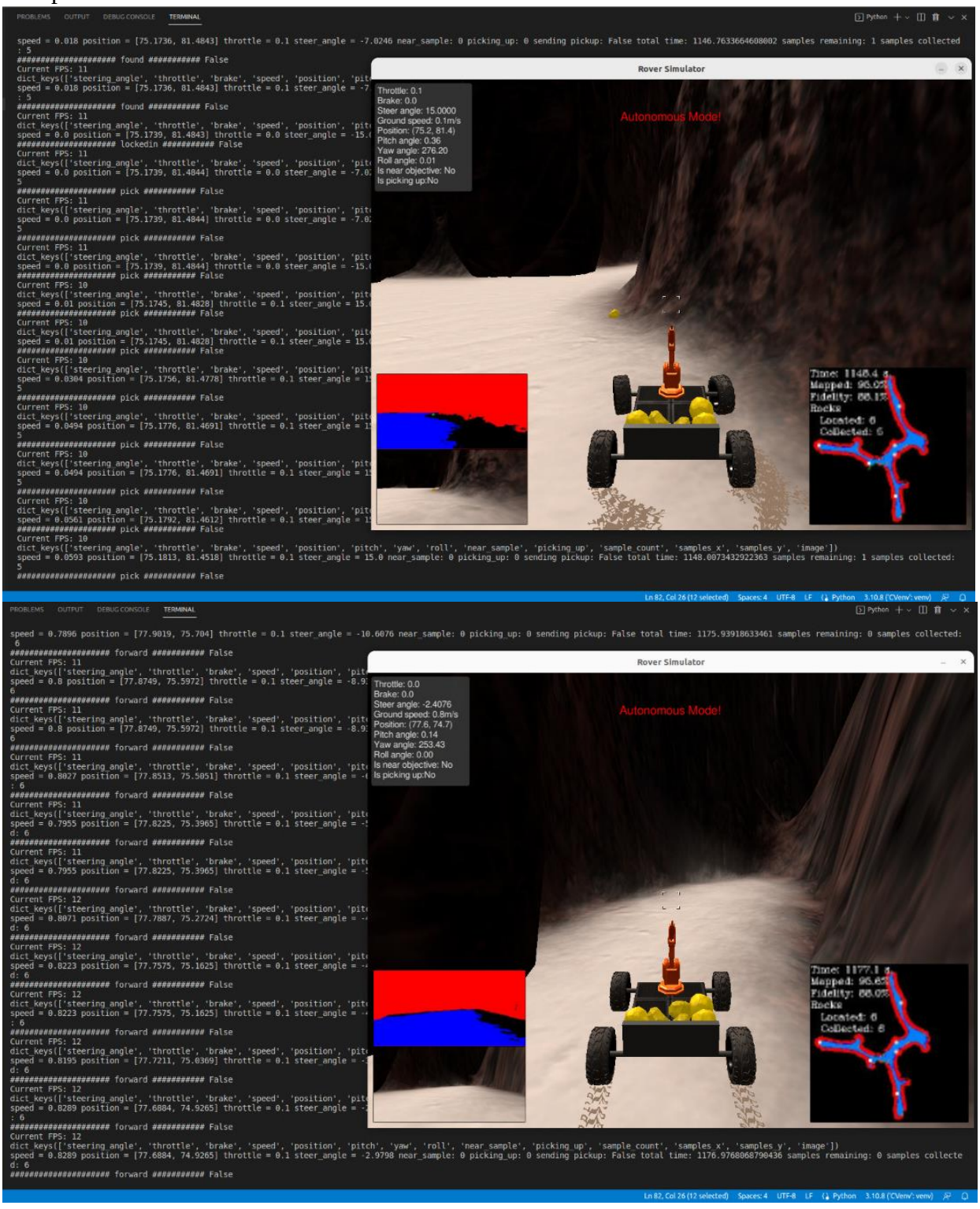
This allows for a better map coverage and decreases the inaccuracies that may occur from the warping being disfigured due to the rover's movement, position, or actions.

Screenshots:

Debugging Mode:



Complete Run:



Time-lapsed recording:
https://drive.google.com/file/d/1rXOvd5BzwLp6wOIZyeJ81OScdZb66bsj/view?usp=sharing