Rover Finds Rock: Exercise in Perception, Decision, and Actuation

February 7, 2018

Navigable terrain, obstacle, and rock identification

The following code defines the functions color_thresh(), color_obstacles(), and color_rocks() that sets a range of possible RGB values which determine what colors are classified to be navigable terrain, obstacles, and rocks respectively.

Color_thresh() is a function that takes in an image and selects all pixels such that for each pixel in the original image all color channels, (red, green, and blue), have values greater than 160. In the selection process, any pixel with this property is labeled as True and represents pixels colored to be classified as navigable terrain. A separate array with the same number of pixels as the original image in a single channel is then initialized to be zero which sets up all pixel values of the original image as a single color. Finally, this single-colored image background takes in as input the selection of pixels labeled as True and sets their values to 1 which highlights navigable terrain in a different color. The first image is the output image of color_thresh() in grayscale where the white region highlights the navigable terrain pixels selected by the function.

Color_obstacles() and color_rocks() are similarly defined as color_thresh() except with different selections of pixel ranges in each of the color channels. Color_obstacles() selects for pixels whose red, green, and blue channels have values that all lie below 160 while color_rocks() selects for pixels whose red and green channels lie above the value 110 and whose blue channel lies above the value 50. In addition, a mask is created to highlight the region that is not within the camera view and is deselected from the color_obstacles() output as a zero in image pixel value. The second image is the output image of color_obstacles() in grayscale where the white region highlights the obstacle terrain pixels selected by the function. Comparing the images produced by color_thresh() and color_obstacles() and disregarding the area not within camera view, the black region of color_thresh() and the white region of color_obstacles() are almost identical which implies that the obstacle areas detected by color_obstacles() are essentially all colored regions that fall outside the threshold for navigable terrain. Finally, the third image displays the original image with a rock pictured within the frame and the fourth image displays a color detection of the yellow in the rock highlighted in white by the function color_rocks(). The range of colors selected for detecting rocks is fairly accurate.

```
1 # Identify pixels above the threshold
```

```
# Threshold of RGB > 160 does a nice job of identifying ground pixels only
def color_thresh(img, rgb_thresh=(160, 160, 160)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "True"
    # where threshold was met
    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 pixels that represent obstacles
# Threshold of RGB < 160 considers as an obstacle a subset of pixels that do not intersect navigable terrain pixels
def color_obstacle(img, rgb_thresh=(160,160,160)):
    color_select = np.zeros_like(img[:,:,0])
    below_thresh = (img[:,:,0] < rgb_thresh[0]) \
                & (img[:,:,1] < rgb_thresh[1]) \
                & (img[:,:,2] < rgb_thresh[2])

    color_select[below_thresh] = 1
    return color_select

# Identify pixels that represent rocks
def color_rocks(img, rgb_thresh=(110,110,50)):
    color_select = np.zeros_like(img[:,:,0])
    between_thresh = (img[:,:,0] > rgb_thresh[0]) \
                  & (img[:,:,1] > rgb_thresh[1]) \
                  & (img[:,:,2] < rgb_thresh[2])

    color_select[between_thresh] = 1
    return color_select

def perspect_transform(img, src, dst):

    M = getPerspetiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))
    mask = cv2.warpPerspective(np.ones_like(img[:,:,0]),M, (img.shape[1], img.shape[0]))

    return warped mask

source = np.float32([[15, 140],
                 [300, 140],
                 [200, 95],
                 [120, 95]])
destination = np.float32([[image.shape[1]/2-5, image.shape[0]-5],
                 [image.shape[1]/2+5, image.shape[0]-5],
                 [image.shape[1]/2+5, image.shape[0]-10-5],
                 [image.shape[1]/2-5, image.shape[0]-10-5]])
warped, mask = perspect_transform(grid_img, source, destination)

threshed = color_thresh(warped)
obstacle = color_obstacle(warped)*mask
```

```
rock = color_rocks(rock_img)
fig = plt.figure(figsize = (12,9))
plt.subplot(221)
plt.imshow(threshed, cmap='gray')
plt.subplot(222)
plt.imshow(obstacle, cmap='gray')
plt.subplot(223)
plt.imshow(rock_img)
plt.subplot(224)
plt.imshow(rock, cmap='gray')
#scipy.misc.imsave('../output/warped_threshed.jpg', threshed*255)
```

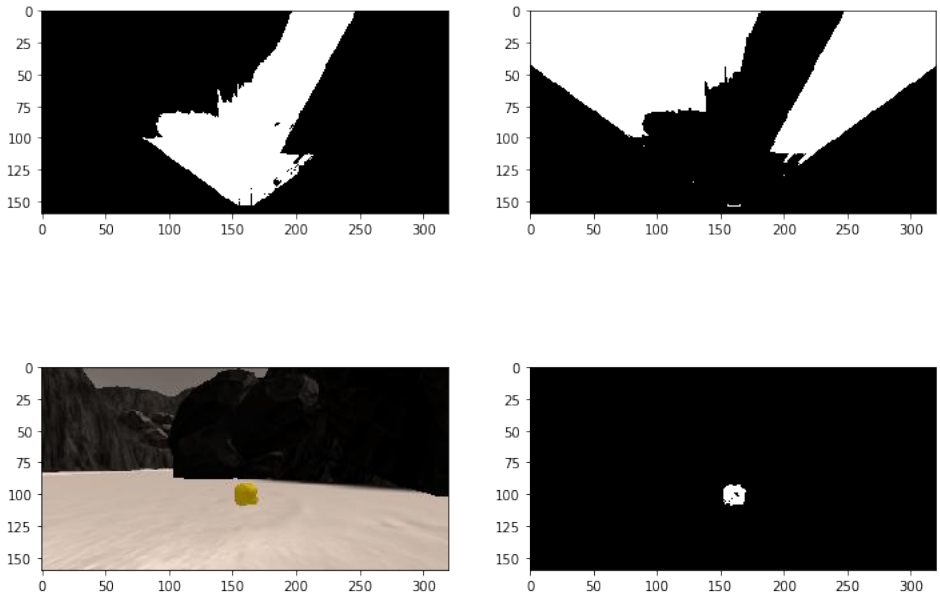

Figure 1

Defining process_image() function

Process_image() is a function that takes in an image and transforms the image to be viewed simultaneously from three different perspectives. The first perspective is the content of the original image. The second perspective is the image viewed after a perspective transform acts on the image to produce a warped top-down view of the rover's data collection of the front view image. The third perspective is obtained by performing a sequence of transformations on three different copies of the image to obtain an overlay of the color distinguished navigable terrain pixels, obstacle pixels, and rock pixels stored in the worldmap and superimposed onto the ground truth map.

The transformations in the third perspective is constructed as follows. First, a perspective transform is performed on the image to change the data collected by the rover's camera into a top-down perspective. Next, the pixels in the warped map are transformed to be centered in rover coordinates. Finally, the pixels recorded in rover coordinates are sent through a rotation and transla-

tion to obtain the pixels in world coordinates, which are pixels ready to be updated on the world map. Each set of transformed coordinates: (obs_x_world, obs_y_world), (nav_x_world, nav_y_world), and (rock_x_world, rock_y_world), is tailored specifically to identify obstacles only, navigable terrain only, or rocks only. The coordinates are updated on the world map by selecting obstacles to be represented by the red channel, navigable terrain represented by the blue channel, and rocks represented by all channels turned on to maximum pixel intensity and therefore appearing as white. Subsequently, an overlay of the worldmap is superimposed onto the ground truth map so what is classified as obstacles, navigable terrain, and rocks by the color thresholding functions can be compared to the actual top-down map of the terrain.

To put everything together, the output image is shaped so the original image is placed in the upper left hand corner of the final image, the perspective transformed image is placed in the upper right hand corner, and the overlay of the worldmap onto the ground truth map is placed in the lower left corner. The process_image() function then outputs this image. Functions from moviepy editor can now be called to take process_image as an input that runs a sequence of images through the function from a data repository storing the images needed to string together a video.

```
1
2  # Define a function to pass stored images to
3  # reading rover position and yaw angle from csv file
4  # This function will be used by moviepy to create an output video
5  def process_image(img):
6      # Example of how to use the Databucket() object defined above
7      # to print the current x, y and yaw values
8      # print(data.xpos[data.count], data.ypos[data.count], data.yaw[
       data.count])
9
10     # TODO:
11     # 1) Define source and destination points for perspective
       transform
12     dst_size = 5
13     # Set a bottom offset to account for the fact that the bottom
       of the image
14     # is not the position of the rover but a bit in front of it
15     bottom_offset = 6
16     source = np.float32([[14, 140], [301 ,140],[200, 96], [118,
       96]])
17     destination = np.float32([[image.shape[1]/2 - dst_size, image.
       shape[0] - bottom_offset],
18                     [image.shape[1]/2 + dst_size, image.shape[0] -
       bottom_offset],
19                     [image.shape[1]/2 + dst_size, image.shape[0] - 2*
       dst_size - bottom_offset],
20                     [image.shape[1]/2 - dst_size, image.shape[0] - 2*
       dst_size - bottom_offset],
21                     ])
22     # 2) Apply perspective transform
23     warped, mask = perspect_transform(img, source, destination)
24     # 3) Apply color threshold to identify navigable terrain/
       obstacles/rock samples
25     nav_terrain = color_thresh(warped, rgb_thresh=(160,160,160))
26     obs_map = color_obstacle(warped, rgb_thresh = (160,160,160))*
       mask
27     rock_map = color_rocks(warped, rgb_thresh=(110,110,50))
28     # 4) Convert thresholded image pixel values to rover-centric
```

```
    coords
    xpix, ypix = rover_coords(nav_terrain)
    obsxpix, obsypix = rover_coords(obs_map)
    rockxpix, rockypix = rover_coords(rock_map)
    # 5) Convert rover-centric pixel values to world coords
    world_size = data.worldmap.shape[0]
    scale = 2*dst_size

    xpos = data.xpos[data.count]
    ypos = data.ypos[data.count]
    yaw = data.yaw[data.count]
    nav_x_world, nav_y_world = pix_to_world(xpix, ypix, xpos, ypos,
     yaw, world_size, scale)

    obs_x_world, obs_y_world = pix_to_world(obsxpix, obsypix, xpos,
     ypos, yaw, world_size, scale)

    rock_x_world, rock_y_world = pix_to_world(rockxpix, rockypix,
    xpos, ypos, yaw, world_size, scale)
    # 6) Update worldmap (to be displayed on right side of screen)
        # Example: data.worldmap[obstacle_y_world, obstacle_x_world
    , 0] += 1
        #          data.worldmap[rock_y_world, rock_x_world, 1] +=
    1
        #          data.worldmap[navigable_y_world,
    navigable_x_world, 2] += 1
    data.worldmap[nav_y_world,nav_x_world, 2]=255
    data.worldmap[obs_y_world, obs_x_world, 0]=255
    if rock_map.any():
        data.worldmap[rock_y_world, rock_x_world, :]=255
    # 7) Make a mosaic image, below is some example code
        # First create a blank image (can be whatever shape you
    like)
    output_image = np.zeros((img.shape[0] + data.worldmap.shape[0],
     img.shape[1]*2, 3))
        # Next you can populate regions of the image with various
    output
        # Here I'm putting the original image in the upper left
    hand corner
    output_image[0:img.shape[0], 0:img.shape[1]] = img

        # Let's create more images to add to the mosaic, first a
    warped image
    warped, mask = perspect_transform(img, source, destination)
        # Add the warped image in the upper right hand corner
    output_image[0:img.shape[0], img.shape[1]:] = warped

        # Overlay worldmap with ground truth map
    map_add = cv2.addWeighted(data.worldmap, 1, data.ground_truth,
    0.5, 0)
        # Flip map overlay so y-axis points upward and add to
    output_image
    output_image[img.shape[0]:, 0:data.worldmap.shape[1]] = np.
    flipud(map_add)


        # Then putting some text over the image
    cv2.putText(output_image,"Video analysis", (20, 20),
                cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
    if data.count < len(data.images) - 1:
        data.count += 1 # Keep track of the index in the Databucket
    ()
```

```
75
76      return output_image
```

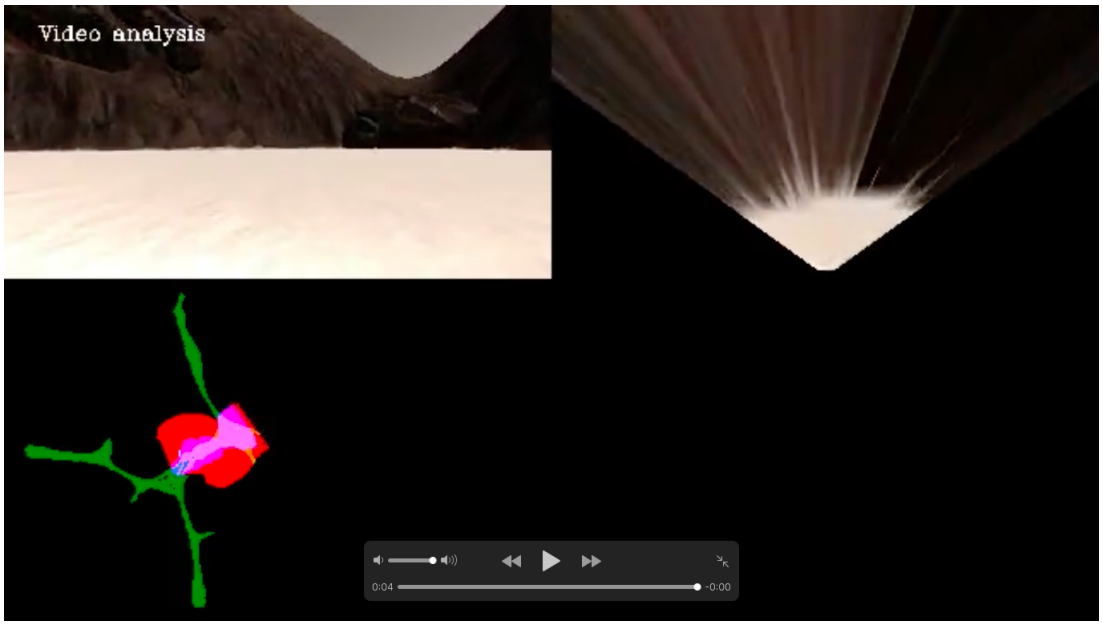

Figure 2

Perception_step() function

The perception_step() function is a function that takes in a class Rover that contains all the interesting data collected on the rover such as camera images from the Rover's front view, steering angle, position, speed, worldmap data, sample collection data, etc. and outputs an update of the data after it has been processed by the function. In particular, the Rover's perspective transformed image is updated to reflect the difference between obstacles and navigable terrain in rover coordinates, the worldmap is updated in transformed coordinates from the original image in world coordinates, and the navigable terrain angles are calculated in polar coordinates.

While autonomously navigating the terrain, the front view camera perspective is transformed by the perception step to give two alternative viewing perspectives of the rover with the intelligence to differentiate between navigable terrain pixels, obstacle pixels, and rock pixels by color region classes for each type of pixel. First, Rover.vision_image stores a map of the Rover in perspective transformed coordinates after the regions distinguishing the color of navigable terrain, obstacles, and rocks has also been selected and differentiated in blue, red, and green respectively. Second, Rover.worldmap stores a map of the Rover similar to the one modified in process_img(). The three types of pixels are again differentiated by three different color channels: blue, red, and green to classify the three different types of regions: navigable terrain, obstacles, and rocks and

are each converted from rover centric coordinates to world coordinates to be updated by their color on the worldmap.

Lastly, the pixels are transformed from rover coordinates to polar coordinates to extract the navigable terrain angle. This information is then fed to the decision_step() function to make updates on the rover's interaction with its environment.

```
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
    source = np.float32([[15, 140],[300, 140], [200, 95], [120,
    95]])
    destination = np.float32([[Rover.img.shape[1]/2 - 5, Rover.img.
    shape[0] - 5],
                              [Rover.img.shape[1]/2+5, Rover.img.
    shape[0]-5],
                              [Rover.img.shape[1]/2 + 5, Rover.img.
    shape[0] -10-5],
                              [Rover.img.shape[1]/2-5, Rover.img.
    shape[0]-10-5]])
    # 2) Apply perspective transform
    warped, mask = perspect_transform(Rover.img, source,
    destination)
    # 3) Apply color threshold to identify navigable terrain/
    obstacles/rock samples
    nav_terrain = color_thresh(warped, rgb_thresh = (160,160,160))
    obs_map = color_obstacle(warped, rgb_thresh = (160,160,160))*
    mask
    rock_map = color_rocks(warped, rgb_thresh = (110,110,50))
    # 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[:,:,2] = nav_terrain*255
    Rover.vision_image[:,:,0] = obs_map*255
    Rover.vision_image[:,:,1] = rock_map*255
    # 5) Convert map image pixel values to rover-centric coords
    xpix, ypix = rover_coords(nav_terrain)
    obsxpix, obsypix = rover_coords(obs_map)
    rocksxpix, rocksypix = rover_coords(rock_map)
    # 6) Convert rover-centric pixel values to world coordinates
    dst_size = 5
    world_size = Rover.worldmap.shape[0]
    scale = 2*dst_size

    nav_x_world, nav_y_world = pix_to_world(xpix, ypix, Rover.pos
    [0], Rover.pos[1],
                                            Rover.yaw, world_size,
    scale)

    obs_x_world, obs_y_world = pix_to_world(obsxpix, obsypix, Rover
    .pos[0], Rover.pos[1],
                                            Rover.yaw, world_size,
    scale)

```

```
    rock_x_world, rock_y_world = pix_to_world(rocksxpix, rocksypix,
     Rover.pos[0], Rover.pos[1],
                                                Rover.yaw, 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[nav_y_world, nav_x_world, 2] += 10
    Rover.worldmap[obs_y_world, obs_x_world, 0] += 1
    Rover.worldmap[rock_y_world, rock_x_world, 1] +=255
    # 8) Convert rover-centric pixel positions to polar coordinates
    # Update Rover pixel distances and angles
        # Rover.nav_dists = rover_centric_pixel_distances
        # Rover.nav_angles = rover_centric_angles
    dist, angles = to_polar_coords(xpix, ypix)
    Rover.nav_angles = angles


    return Rover
```

Decision_step() function

The decision_step() function like the perception_step() function takes in a class Rover which contains all the interesting updated data about the rover after it is modified by the perception_step() and adjusts the action of the rover in its environment by following the direction of a sequence of conditional statements.

The following is an outline of the decision_step() function before modification. The first conditional assumes that there are navigable terrain pixels populated on the map where the nav_angles, initialized to None, have been updated by the perception_step(). On the other hand, if the nav_angles have not deviated from their initialized state, then the simple setting is to make the rover roll forward. Going back to the first case, if the Rover.nav_angles have been updated by the perception_step(), then consider the splitting of two possible states of the rover: 'forward' mode and 'stop' mode. If the rover is in 'forward' mode, then consider another splitting of states: the number of navigable terrain pixels is above or below a certain threshold. If the number of navigable terrain pixels is above this threshold, then consider whether the rover is moving below a maximum velocity or above. In the case that the rover is moving below the maximum velocity, set the throttle to move the rover forward otherwise set the throttle to zero, the brake to zero, and the steer to point in the direction of the average navigable terrain pixel angle. The alternative setting is if the rover's perception step detects mostly obstacles ahead and the number of navigable terrain pixels falls short of a minimum pixel count, in this case, set the throttle to zero, turn on the brake, steer to zero, and the mode to stop. Changing to 'stop' mode, consider whether the rover velocity is greater than .2 or less than .2, when it is greater, then turn on the brake and set throttle and steer values to 0. In the case where the velocity is less than .2, consider whether the rover perceives enough navigable terrain pixels to go forward, if not, then turn around, if so,

then go forward by turning on throttle, setting brake to zero, setting the steer back to the direction of the mean angle of navigable terrain pixels, and resetting mode to go 'forward'.

After modification, we add a setting in which the rover follows the wall on the left but avoids crashing into it and we take into consideration how to maneuver the rover out of situations where it can be in 'stuck' or 'tilt' mode. To recover from being stuck at an obstacle, we consider the length of time the rover stays at velocity zero. There is a count of approximately 10 frames per second for 5 seconds before the rover issues the command to choose whether it is in 'tilt' mode or 'stuck' mode. At zero velocity, if the rover's pitch or roll is out of the normal range, then the rover chooses to be in 'tilt' mode which is a mode that twists the rover's orientation to resurrect the rover to normal pitch and roll range so that it can return to 'forward' mode. If the rover's pitch or roll are in normal range but still stuck at velocity zero, then the rover chooses to be in 'stuck' mode which is a mode that backs-up by counting 10 frames per second for 2 seconds, then re-orients by turning right for 10 frames per second for 4 seconds, and finally returns to 'forward' mode with throttle reset, and brake, steer, and counter set back to zero. In the alternative case where the rover has positive velocity, the rover considers whether any of its navigable terrain pixels are mapped within a certain range on its left hand side, if so, then the rover steers left and throttles forward to stay close to wall-like obstacles on its left. If there are no navigable terrain pixels in the left most region, then the rover steers in the direction of the mean angle of navigable terrain pixels and throttles forward. Besides these add ons, the rest of the code stays the same.

```
def decision_step(Rover):

    if Rover.nav_angles is not None:
        # Check for Rover.mode status

        if Rover.mode == 'forward':
            # 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
                    if Rover.vel == 0:
                        if Rover.counter < 10*5:
                            Rover.throttle = Rover.throttle_set
                            Rover.brake = 0
                            Rover.steer = 0
                            Rover.counter = Rover.counter + 1
                        elif Rover.counter >= 10*5:
                            if (Rover.pitch >= 3 and Rover.pitch
    <=355) or (Rover.roll >=3 and Rover.roll<=355):
                                Rover.mode = 'tilt'
                                Rover.counter = 0
                            else:
                                Rover.mode = 'stuck'
                                Rover.counter = 0
                    elif Rover.vel>0:
                        if any(Rover.nav_angles[i] >= np.pi/6 and
    Rover.nav_dists[i] >=10
                                for i in range(len(Rover.nav_angles)
    )):
                            Rover.steer = 8
```

```
                            Rover.brake = 0
                            Rover.throttle = Rover.throttle_set
                        else:
                            Rover.throttle = Rover.throttle_set
                            Rover.brake = 0
                            Rover.steer = np.clip(np.mean(Rover.
nav_angles *180/np.pi), -15,15)
                            #Rover.steer = np.clip(np.mean(Rover.
nav_angles * 180/np.pi), -15,15)
                elif: # 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), -15, 15)
            # If there's a lack of navigable terrain pixels then go
 to 'stop' mode
            elif len(Rover.nav_angles) < Rover.stop_forward:
                    # 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'

        # If we're already in "stop" mode then make different
decisions
        elif Rover.mode == 'stop':
            # 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 = -15 # 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), -15, 15)
                    Rover.mode = 'forward'

        elif Rover.mode == 'stuck':
            if Rover.counter < 10*2:
                Rover.throttle = -1
```

```
                Rover.brake = 0
                Rover.steer = 0
                Rover.counter = Rover.counter + 1
            elif Rover.counter >= 10*2 and Rover.counter< 10*6:
                Rover.throttle = 0
                Rover.brake = 0
                Rover.steer = -15
                Rover.counter = Rover.counter + 1
            elif Rover.counter>= 10*6:
                Rover.mode = 'forward'
                Rover.throttle = Rover.throttle_set
                Rover.brake =0
                Rover.steer=0
                Rover.counter = 0

        elif Rover.mode == 'tilt':
            if (Rover.pitch >= 3 and Rover.pitch <= 355) or (Rover.
    roll >=3 and Rover.pitch <=355):
                Rover.throttle = 0
                Rover.brake = 0
                Rover.steer = -15
            else:
                Rover.throttle = Rover.throttle_set
                Rover.brake = 0
                Rover.steer = 0
                Rover.mode = 'forward'
    # Just to make the rover do something
    # even if no modifications have been made to the code
    else:
        Rover.throttle = Rover.throttle_set
        Rover.steer = 0
        Rover.brake = 0

    # If in a state where want to pickup a rock send pickup command
    if Rover.near_sample and Rover.vel == 0 and not Rover.
    picking_up:
        Rover.send_pickup = True

    return Rover
```

Results for running drive_rover.py

The following shows a screenshot of the rover running in autonomous mode after having mapped 78.7% of the terrain at 64.2% fidelity and locating 4 rocks. Running the entire course, the rover can map 99.4% of the terrain at fidelity 52.6%, locate 4 rocks and pick up 1 rock. Although fidelity drops as the rover covers more ground and encounters obstacles that deform its map accuracy and increases its time in 'stuck' mode, it can still maneuver out of both 'stuck' mode and 'tilt' mode. Here is a youtube video link of the rover running the entire course: https://youtu.be/I1AYNhu6giw


Untitled
Throttle: 0.2
Brake: 0.0
Steer angle: 8.0000
Ground speed: 1.9m/s
Position: (142.8, 112.3)
Pitch angle: 0.01
Yaw angle: 345.03
Roll angle: 0.92
Is near objective: No
Is picking up:No
Autonomous Mode!
06:24
-12:21
Time: 379.1
Mapped: 78.7%
Fidelity: 64.2%
Rocks
Located: 4
Collected: 0

Figure 3