Faculty of Engineering - Ain Shams University

CSE483 Computer Vision

Project for fall 2022

Submitted to:

Dr. Mahmoud Khalil

Eng. Mahmoud Selim

Submitted by:

Mohamad Ahmed Mohamad Abdelmoniem	19P5170
Karim Ashraf	19P6044
Ahmed Mohsen Ahmed	19P1150
Mohamed Fathy	19P4704
Ziad Assem	19P6363

GitHub repo:

https://github.com/Fathy24/VisionProject

Table of Contents

1. Introduction

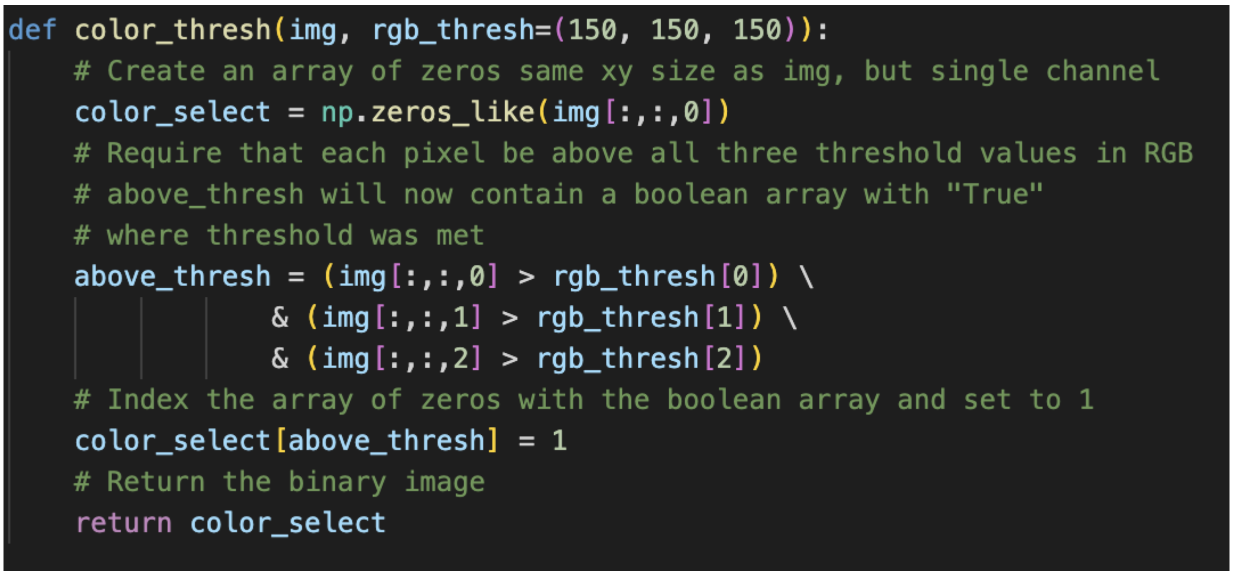
1.1 Packages used:

1. NumPy
2. OpenCV
3. Eventlet
4. Pillow
5. Matplotlib
6. SciPy
7. Imageio
8. Flask
9. Socketio==4.6.1
10. Engineio==3.13.2

2. Perception Module

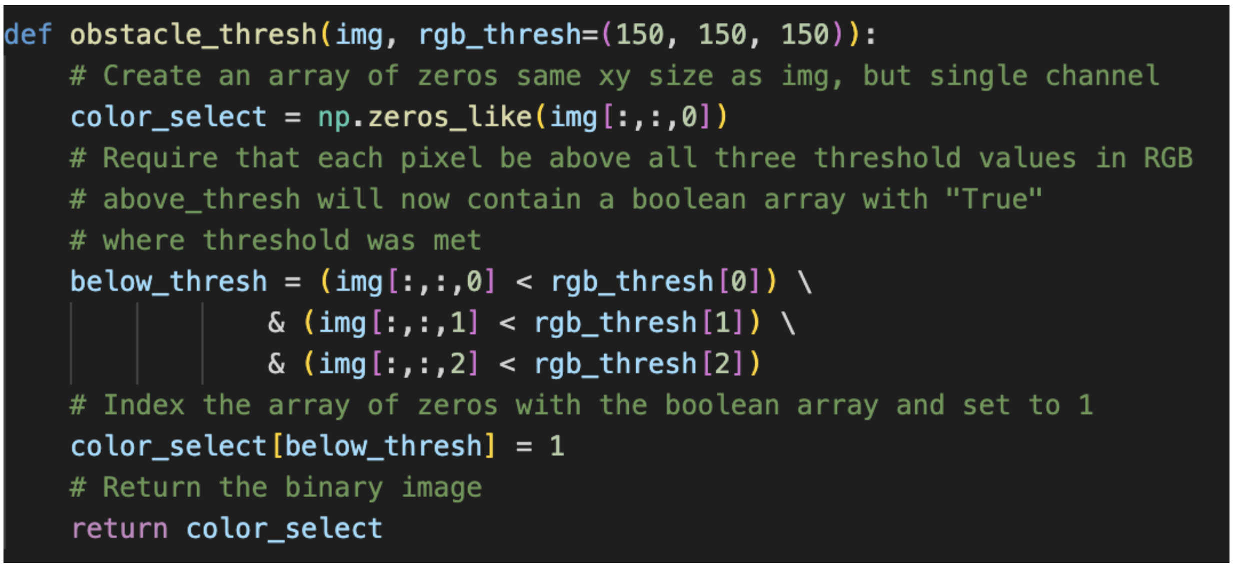
Perception module is responsible for the thresholding of the image from the terrain and we added a small mask in phase 2 to adjust its motion.

2.1 Functions



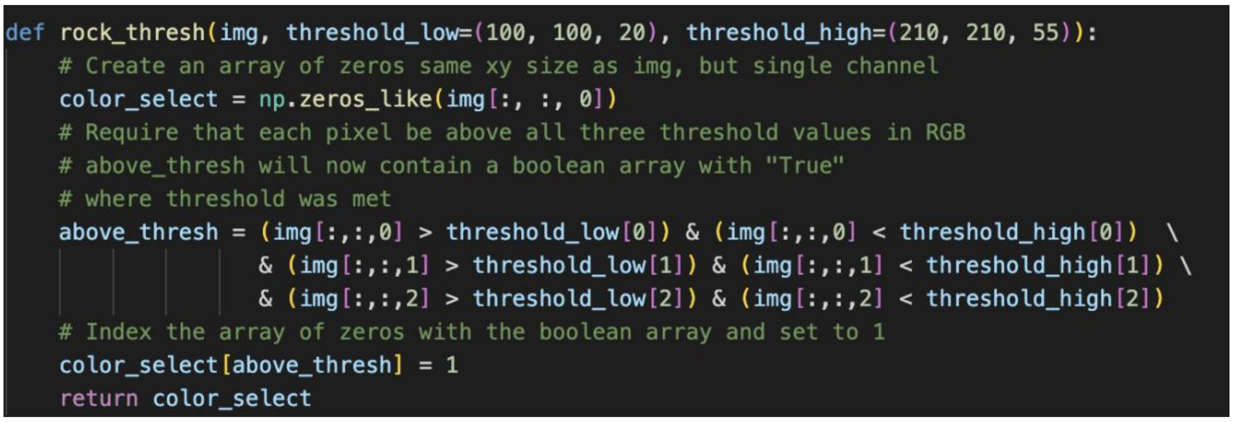
```
def color_thresh(img, rgb_thresh=(150, 150, 150)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "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
```

Function responsible for thresholding the path



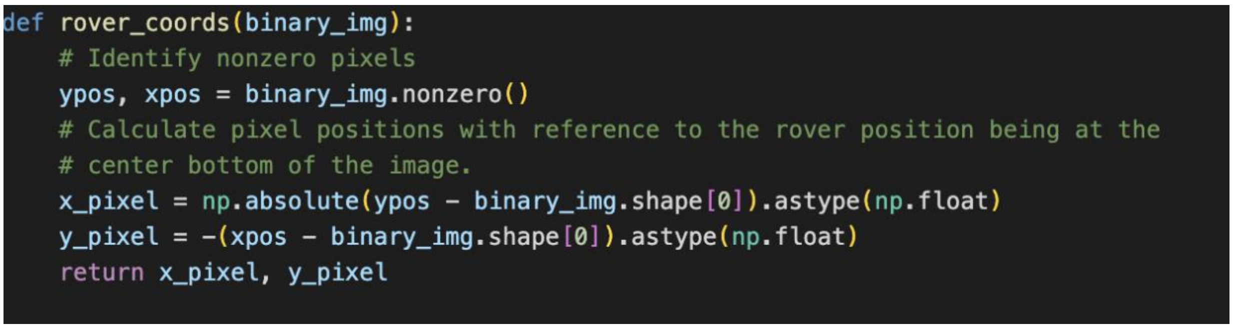
```python
def obstacle_thresh(img, rgb_thresh=(150, 150, 150)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "True"
    # where threshold was met
    below_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[below_thresh] = 1
    # Return the binary image
    return color_select
```

Function responsible for thresholding the obstacles



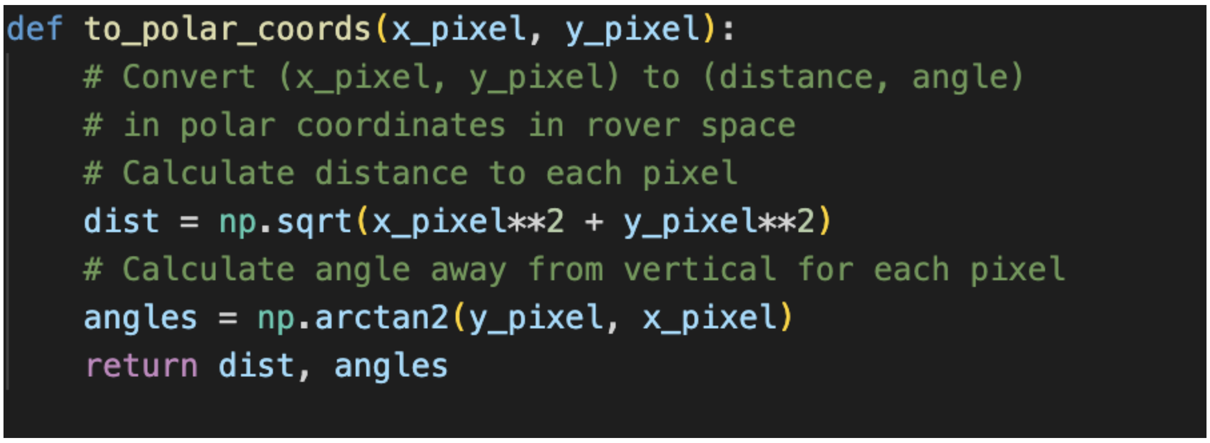
```python
def rock_thresh(img, threshold_low=(100, 100, 20), threshold_high=(210, 210, 55)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:, :, 0])
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "True"
    # where threshold was met
    above_thresh = (img[:,:,0] > threshold_low[0]) & (img[:,:,0] < threshold_high[0])  \
                & (img[:,:,1] > threshold_low[1]) & (img[:,:,1] < threshold_high[1]) \
                & (img[:,:,2] > threshold_low[2]) & (img[:,:,2] < threshold_high[2])
    # Index the array of zeros with the boolean array and set to 1
    color_select[above_thresh] = 1
    return color_select
```

Function responsible for thresholding the rock



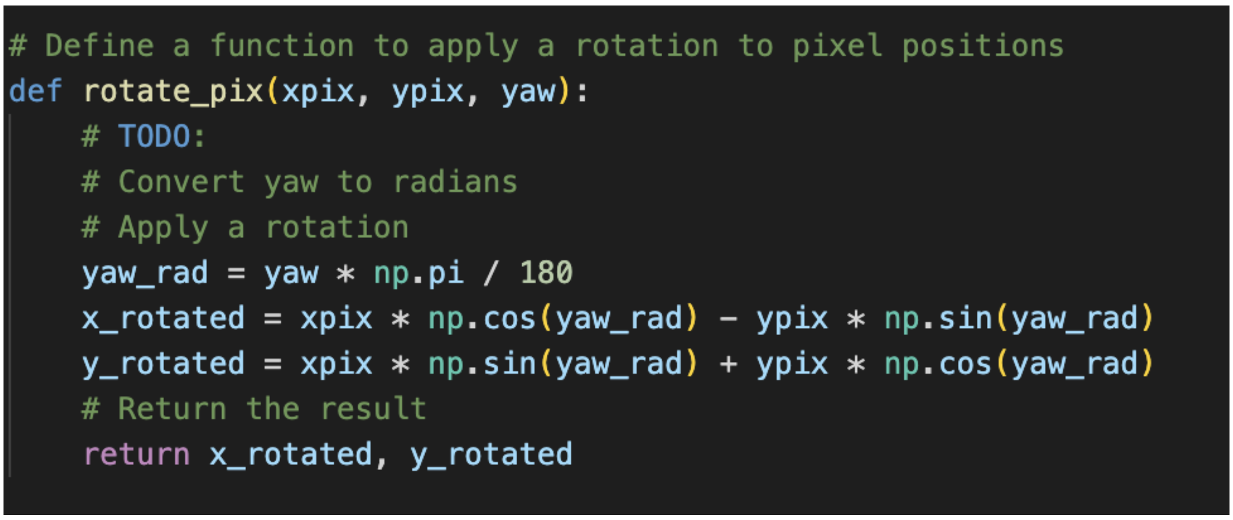
```python
def rover_coords(binary_img):
    # Identify nonzero pixels
    ypos, xpos = binary_img.nonzero()
    # Calculate pixel positions with reference to the rover position being at the
    # center bottom of the image.
    x_pixel = np.absolute(ypos - binary_img.shape[0]).astype(np.float)
    y_pixel = -(xpos - binary_img.shape[0]).astype(np.float)
    return x_pixel, y_pixel
```

Function responsible for converting to rover centric coordinates.



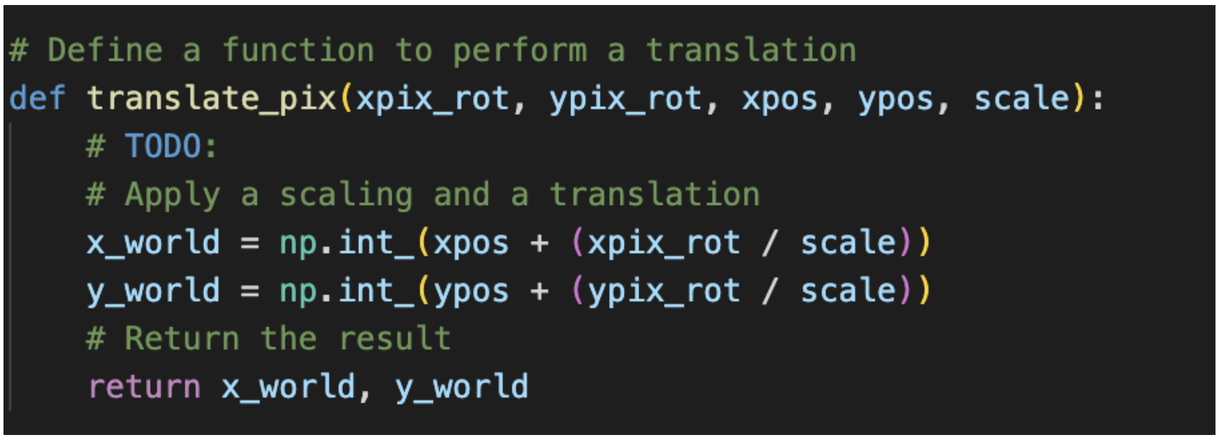
```python
def to_polar_coords(x_pixel, y_pixel):
    # Convert (x_pixel, y_pixel) to (distance, angle)
    # in polar coordinates in rover space
    # Calculate distance to each pixel
    dist = np.sqrt(x_pixel**2 + y_pixel**2)
    # Calculate angle away from vertical for each pixel
    angles = np.arctan2(y_pixel,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

Function responsible to get rover’s navigable angles and navigable distance in polar coordinates



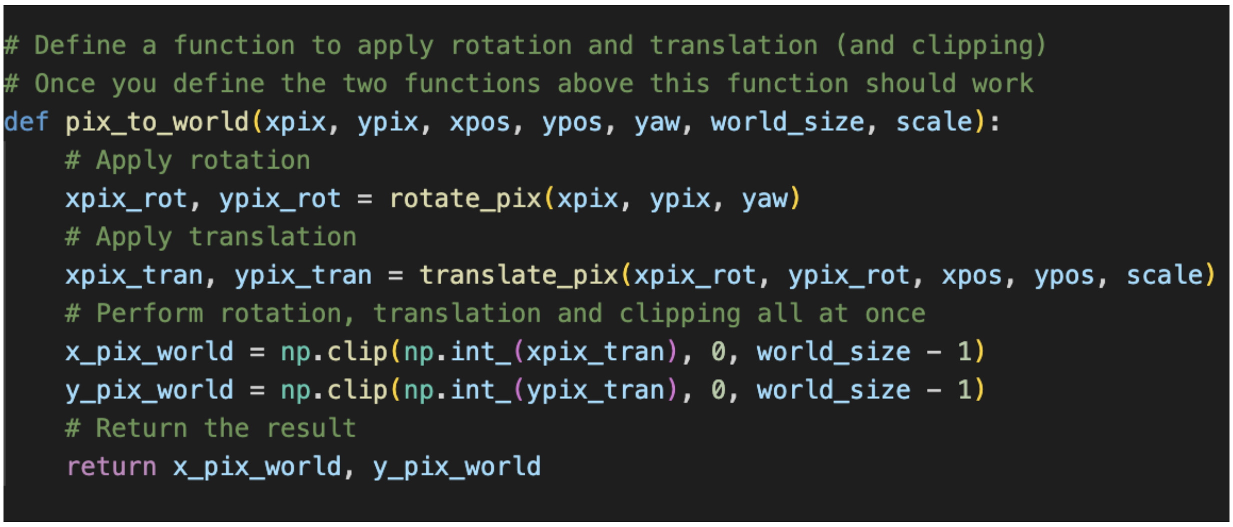
```python
# Define a function to apply a rotation to pixel positions
def rotate_pix(xpix, ypix, yaw):
    # TODO:
    # Convert yaw to radians
    # Apply a rotation
    yaw_rad = yaw * np.pi / 180
    x_rotated = xpix * np.cos(yaw_rad) - ypix * np.sin(yaw_rad)
    y_rotated = xpix * np.sin(yaw_rad) + ypix * np.cos(yaw_rad)
    # Return the result
    return x_rotated, y_rotated
```

Function responsible to rotate axes to adjust rover axis with world axis. Rotating using geometric transformations.

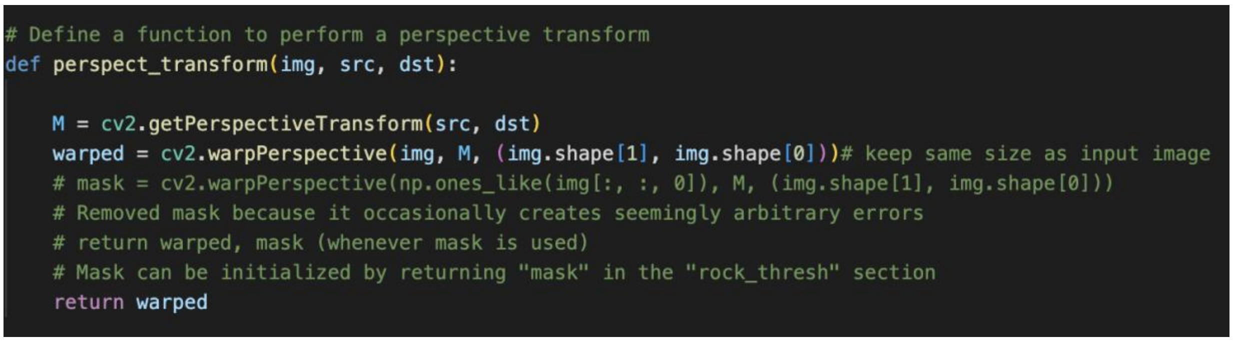


```python
# Define a function to perform a translation
def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # TODO:
    # Apply a scaling and a translation
    x_world = np.int_(xpos + (xpix_rot / scale))
    y_world = np.int_(ypos + (ypix_rot / scale))
    # Return the result
    return x_world, y_world
```

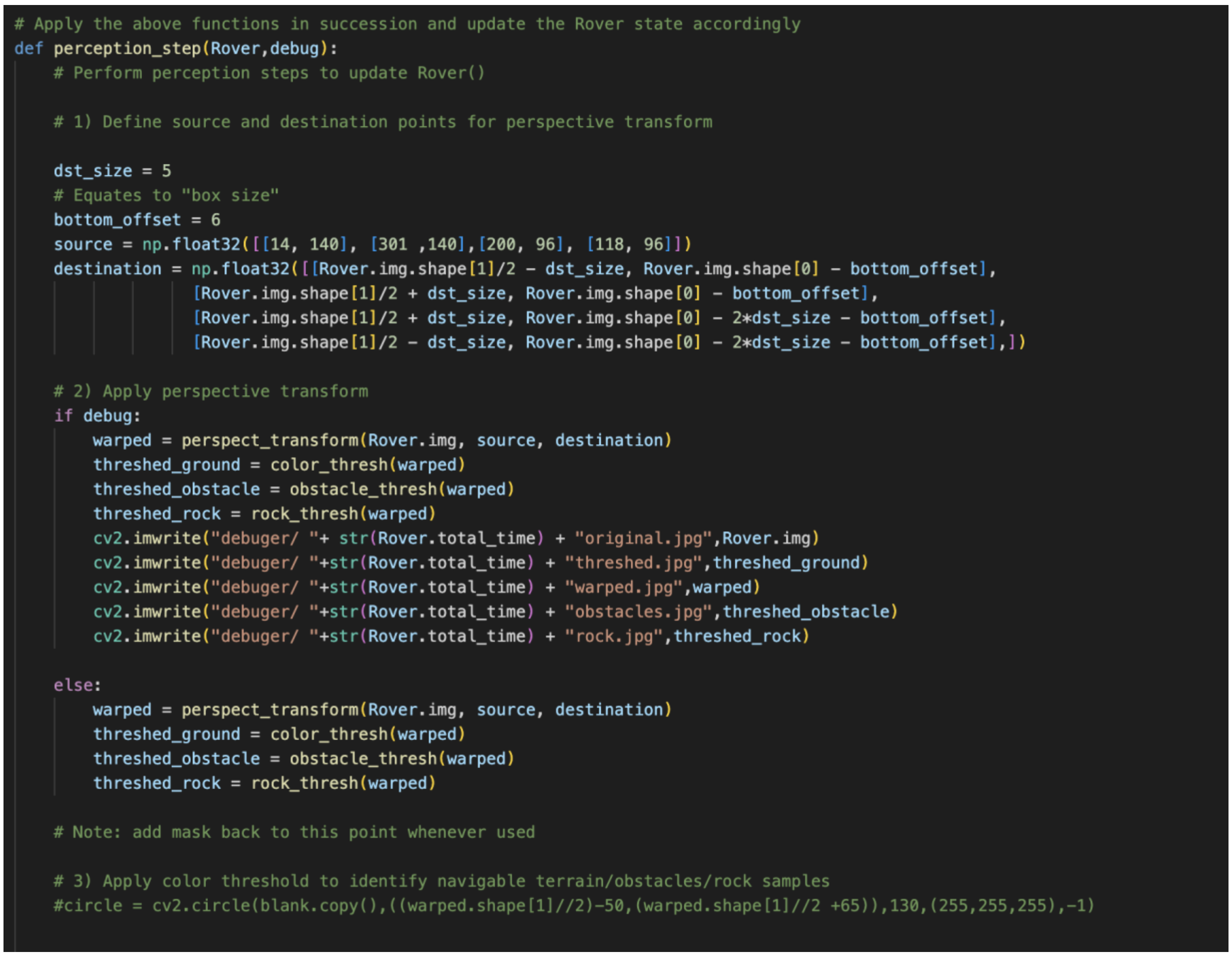
Function responsible to translate pixels with suitable scaling that will be adjusted later.


```python
# 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
```

Using the previously created functions, this function rotates the rover coordinates and axes to match the world's axes using the yaw angle. Then we translate the axes also to start from the same point so that the rover is the origin. We then clip the values to be inside a certain range.

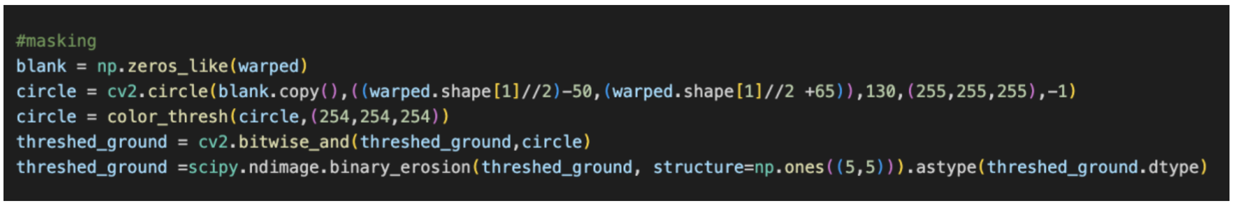

```python
# Define a function to perform a perspective transform
def perspect_transform(img, src, dst):

    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))# keep same size as input image
    # mask = cv2.warpPerspective(np.ones_like(img[:, :, 0]), M, (img.shape[1], img.shape[0]))
    # Removed mask because it occasionally creates seemingly arbitrary errors
    # return warped, mask (whenever mask is used)
    # Mask can be initialized by returning "mask" in the "rock_thresh" section
    return warped
```

Function that gets the bird eye view



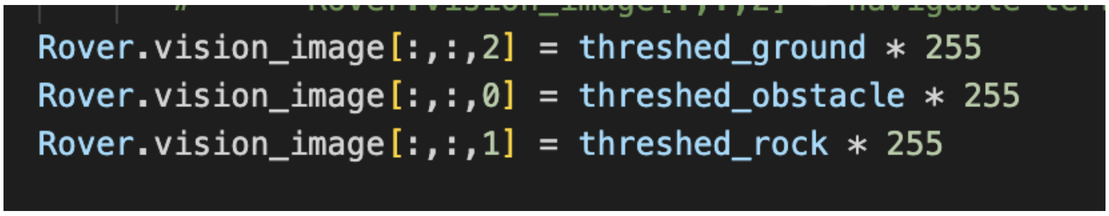
```python
# Apply the above functions in succession and update the Rover state accordingly
def perception_step(Rover,debug):
    # Perform perception steps to update Rover()

    # 1) Define source and destination points for perspective transform

    dst_size = 5
    # Equates to "box size"
    bottom_offset = 6
    source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
    destination = np.float32([[Rover.img.shape[1]/2 - dst_size, Rover.img.shape[0] - bottom_offset],
                    [Rover.img.shape[1]/2 + dst_size, Rover.img.shape[0] - bottom_offset],
                    [Rover.img.shape[1]/2 + dst_size, Rover.img.shape[0] - 2*dst_size - bottom_offset],
                    [Rover.img.shape[1]/2 - dst_size, Rover.img.shape[0] - 2*dst_size - bottom_offset],])

    # 2) Apply perspective transform
    if debug:
        warped = perspect_transform(Rover.img, source, destination)
        threshed_ground = color_thresh(warped)
        threshed_obstacle = obstacle_thresh(warped)
        threshed_rock = rock_thresh(warped)
        cv2.imwrite("debuger/ "+ str(Rover.total_time) + "original.jpg",Rover.img)
        cv2.imwrite("debuger/ "+str(Rover.total_time) + "threshed.jpg",threshed_ground)
        cv2.imwrite("debuger/ "+str(Rover.total_time) + "warped.jpg",warped)
        cv2.imwrite("debuger/ "+str(Rover.total_time) + "obstacles.jpg",threshed_obstacle)
        cv2.imwrite("debuger/ "+str(Rover.total_time) + "rock.jpg",threshed_rock)

    else:
        warped = perspect_transform(Rover.img, source, destination)
        threshed_ground = color_thresh(warped)
        threshed_obstacle = obstacle_thresh(warped)
        threshed_rock = rock_thresh(warped)

    # Note: add mask back to this point whenever used

    # 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
    #circle = cv2.circle(blank.copy(),((warped.shape[1]//2)-50,(warped.shape[1]//2 +65)),130,(255,255,255),-1)
```

Setting bottom offset and setting the debugging mode. Also giving bird eye view to the path, obstacles and rock.



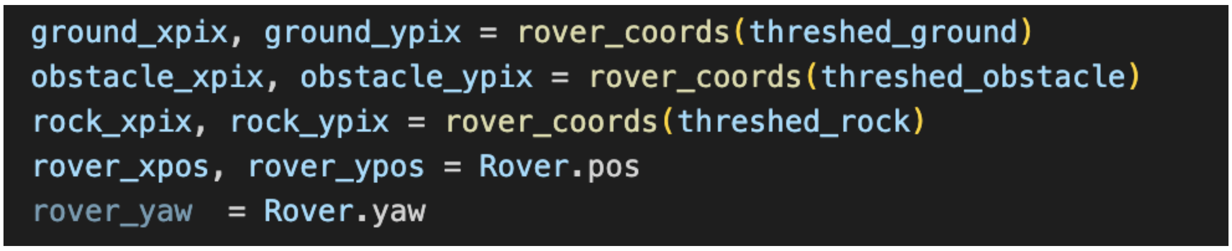
```python
#masking
blank = np.zeros_like(warped)
circle = cv2.circle(blank.copy(),((warped.shape[1]//2)-50,(warped.shape[1]//2 +65)),130,(255,255,255),-1)
circle = color_thresh(circle,(254,254,254))
threshed_ground = cv2.bitwise_and(threshed_ground,circle)
threshed_ground =scipy.ndimage.binary_erosion(threshed_ground, structure=np.ones((5,5))).astype(threshed_ground.dtype)
```

This is an important mask which was implemented in phase 2 to enhance the rover's driving.



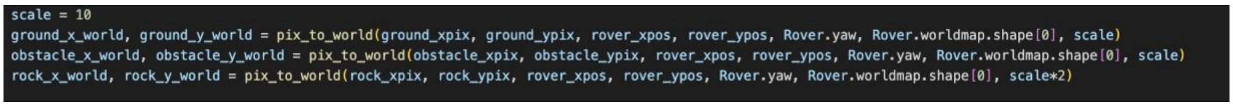
```python
Rover.vision_image[:,:,2] = threshed_ground * 255
Rover.vision_image[:,:,0] = threshed_obstacle * 255
Rover.vision_image[:,:,1] = threshed_rock * 255
```

Coloring each channel in the rover image. Each channel is for the path, obstacles, and rock.



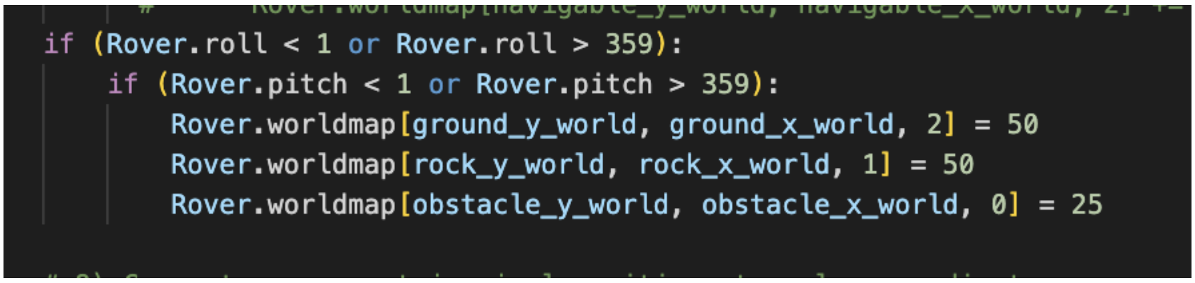
```
ground_xpix, ground_ypix = rover_coords(threshed_ground)
obstacle_xpix, obstacle_ypix = rover_coords(threshed_obstacle)
rock_xpix, rock_ypix = rover_coords(threshed_rock)
rover_xpos, rover_ypos = Rover.pos
rover_yaw  = Rover.yaw
```

Converting map pixels to rover centric coordinates.



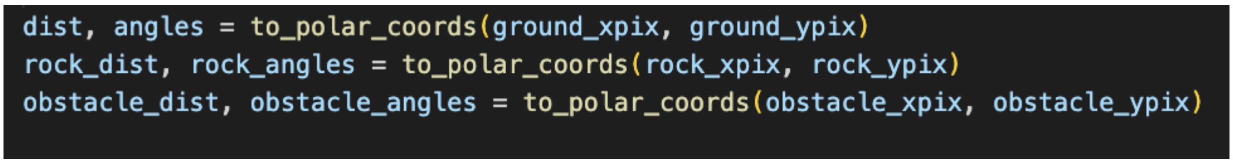
```
scale = 10
ground_x_world, ground_y_world = pix_to_world(ground_xpix, ground_ypix, rover_xpos, rover_ypos, Rover.yaw, Rover.worldmap.shape[0], scale)
obstacle_x_world, obstacle_y_world = pix_to_world(obstacle_xpix, obstacle_ypix, rover_xpos, rover_ypos, Rover.yaw, Rover.worldmap.shape[0], scale)
rock_x_world, rock_y_world = pix_to_world(rock_xpix, rock_ypix, rover_xpos, rover_ypos, Rover.yaw, Rover.worldmap.shape[0], scale*2)
```

Converting rover coordinates to real world.



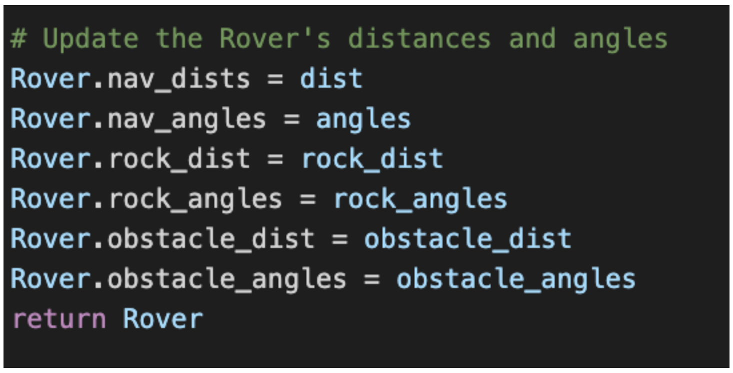
```
        #       Rover.worldmap[navigable_y_world, navigable_x_world, 2] +=
    if (Rover.roll < 1 or Rover.roll > 359):
        if (Rover.pitch < 1 or Rover.pitch > 359):
            Rover.worldmap[ground_y_world, ground_x_world, 2] = 50
            Rover.worldmap[rock_y_world, rock_x_world, 1] = 50
            Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] = 25
```

Updating world map.



```
dist, angles = to_polar_coords(ground_xpix, ground_ypix)
rock_dist, rock_angles = to_polar_coords(rock_xpix, rock_ypix)
obstacle_dist, obstacle_angles = to_polar_coords(obstacle_xpix, obstacle_ypix)
```

Calculating angles and distances for the rover’s navigable angles and navigable distance.

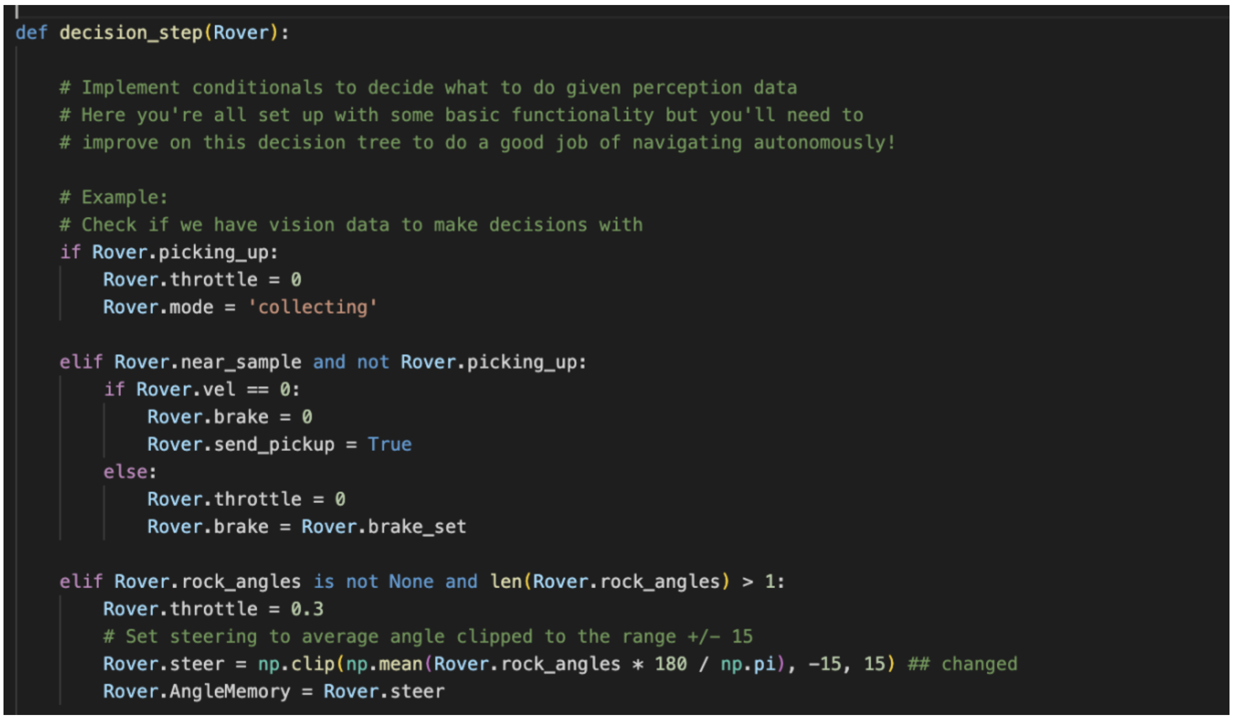


```
# Update the Rover's distances and angles
Rover.nav_dists = dist
Rover.nav_angles = angles
Rover.rock_dist = rock_dist
Rover.rock_angles = rock_angles
Rover.obstacle_dist = obstacle_dist
Rover.obstacle_angles = obstacle_angles
return Rover
```

Assigning the updated values to the rover.

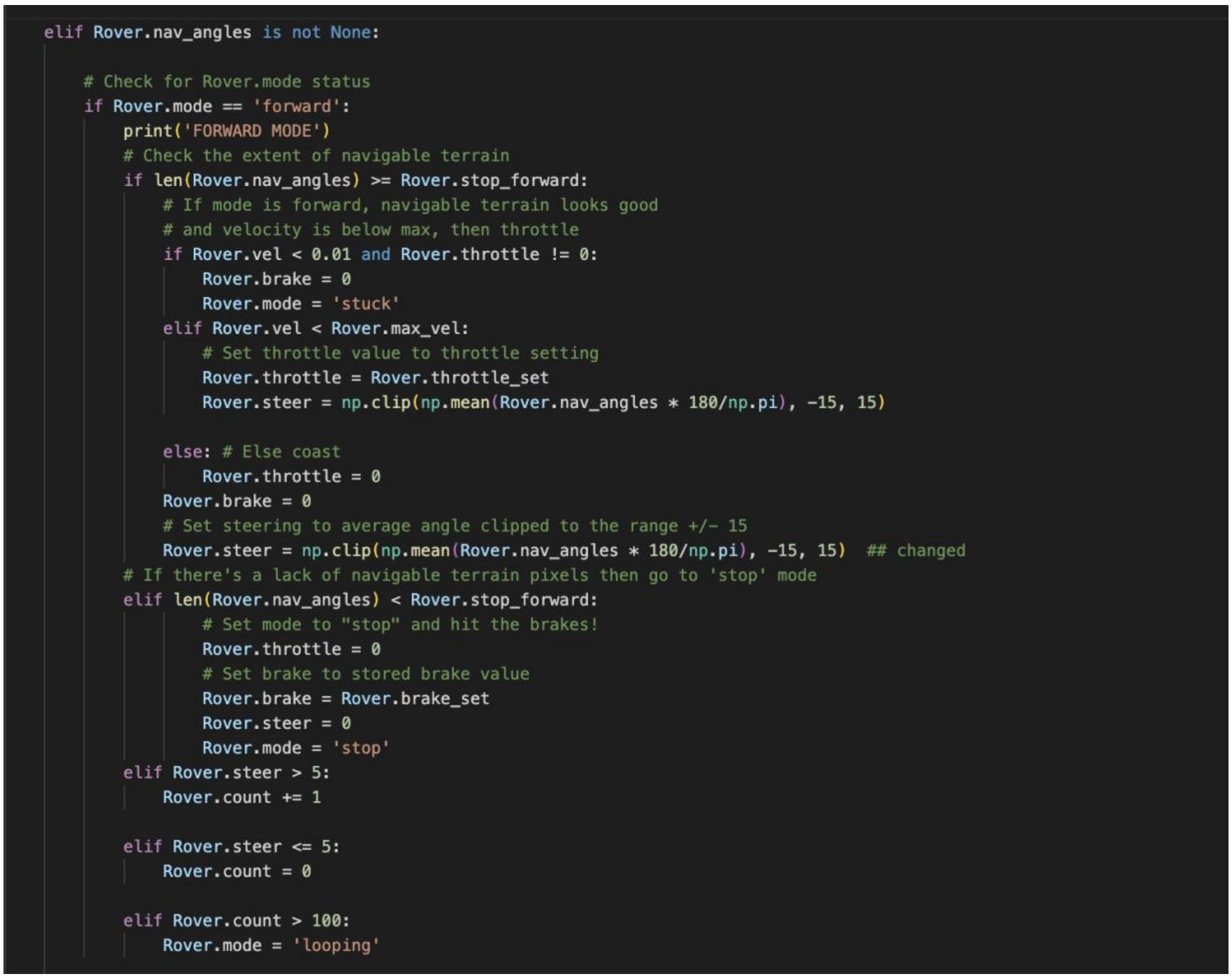
3. Decision Module

Decision is responsible for fetching the rocks.



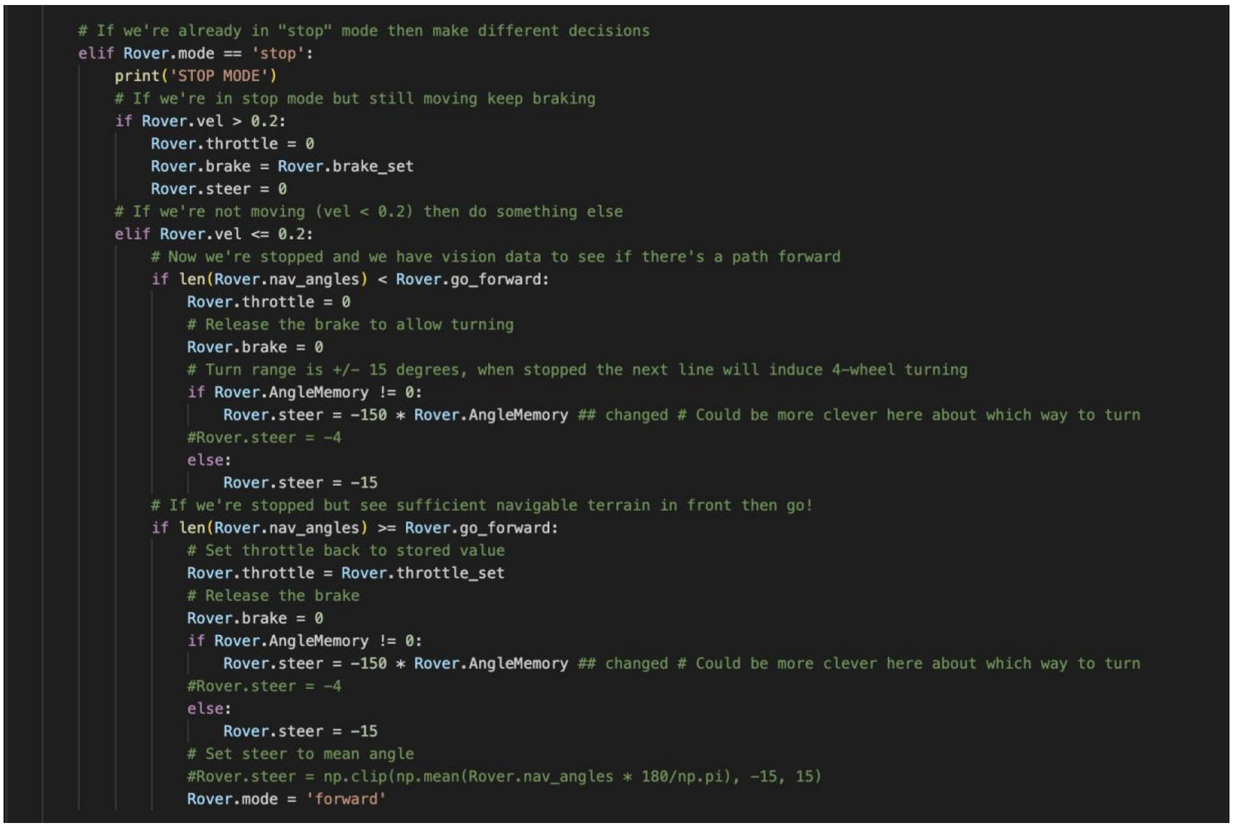
```python
def decision_step(Rover):

    # Implement conditionals to decide what to do given perception data
    # Here you're all set up with some basic functionality but you'll need to
    # improve on this decision tree to do a good job of navigating autonomously!

    # Example:
    # Check if we have vision data to make decisions with
    if Rover.picking_up:
        Rover.throttle = 0
        Rover.mode = 'collecting'

    elif Rover.near_sample and not Rover.picking_up:
        if Rover.vel == 0:
            Rover.brake = 0
            Rover.send_pickup = True
        else:
            Rover.throttle = 0
            Rover.brake = Rover.brake_set

    elif Rover.rock_angles is not None and len(Rover.rock_angles) > 1:
        Rover.throttle = 0.3
        # Set steering to average angle clipped to the range +/- 15
        Rover.steer = np.clip(np.mean(Rover.rock_angles * 180 / np.pi), -15, 15) ## changed
        Rover.AngleMemory = Rover.steer
```

Functions responsible for fetching the ball.



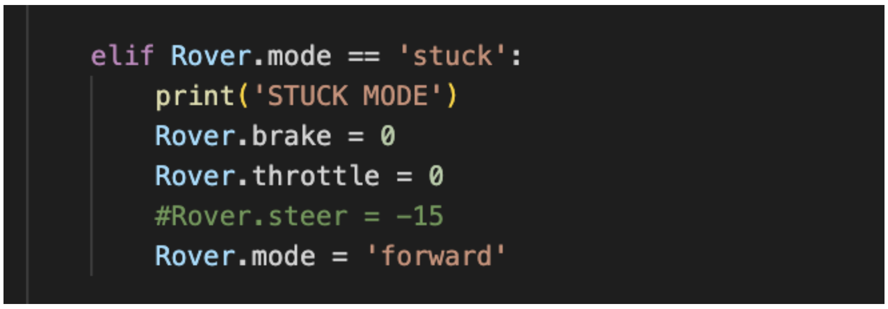
```
elif Rover.nav_angles is not None:

    # Check for Rover.mode status
    if Rover.mode == 'forward':
        print('FORWARD MODE')
        # 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 < 0.01 and Rover.throttle != 0:
                Rover.brake = 0
                Rover.mode = 'stuck'
            elif Rover.vel < Rover.max_vel:
                # Set throttle value to throttle setting
                Rover.throttle = Rover.throttle_set
                Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)

            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), -15, 15)  ## changed
        # 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'
        elif Rover.steer > 5:
            Rover.count += 1

        elif Rover.steer <= 5:
            Rover.count = 0

        elif Rover.count > 100:
            Rover.mode = 'looping'
```

Functions responsible for moving forward and steering in the right direction while moving forward with the guide of the clipped navigable angles that we got from the perception_step() function.



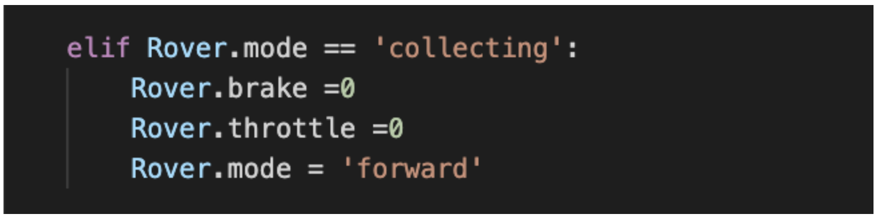
```python
        # If we're already in "stop" mode then make different decisions
        elif Rover.mode == 'stop':
            print('STOP MODE')
            # If we're in stop mode but still moving keep braking
            if Rover.vel > 0.2:
                Rover.throttle = 0
                Rover.brake = 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
                    if Rover.AngleMemory != 0:
                        Rover.steer = -150 * Rover.AngleMemory ## changed # Could be more clever here about which way to turn
                    #Rover.steer = -4
                    else:
                        Rover.steer = -15
                # 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
                    if Rover.AngleMemory != 0:
                        Rover.steer = -150 * Rover.AngleMemory ## changed # Could be more clever here about which way to turn
                    #Rover.steer = -4
                    else:
                        Rover.steer = -15
                    # Set steer to mean angle
                    #Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)
                    Rover.mode = 'forward'
```

Stop conditional responsible for handling the stopping of the rover and adjusting the steering after stopping and collecting the rocks, to be directed back to the correct direction.



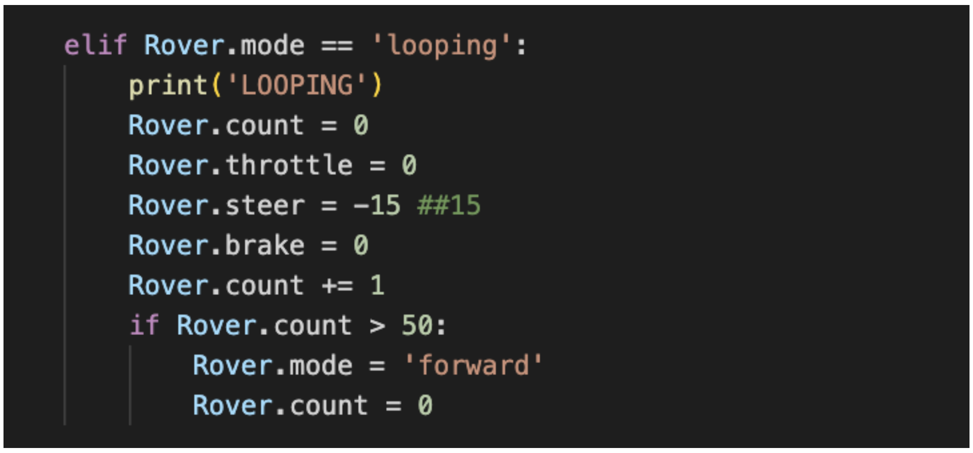
```python
    elif Rover.mode == 'stuck':
        print('STUCK MODE')
        Rover.brake = 0
        Rover.throttle = 0
        #Rover.steer = -15
        Rover.mode = 'forward'
```

This part handles the rover getting stuck.



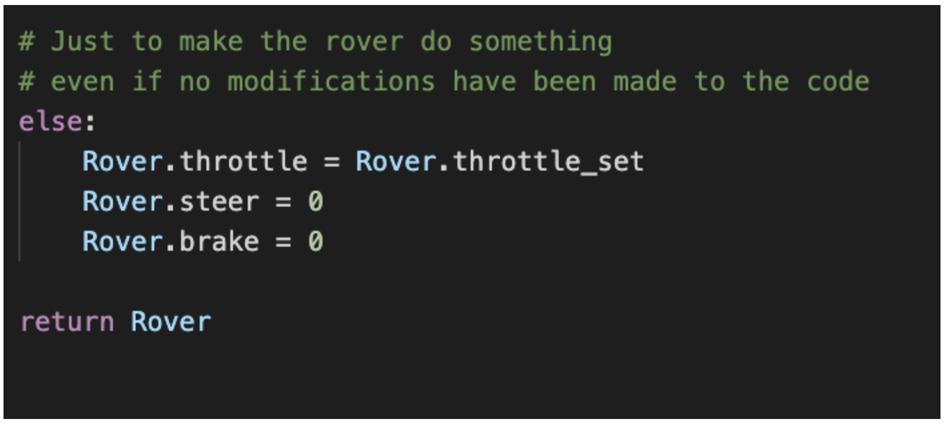
```python
    elif Rover.mode == 'collecting':
        Rover.brake =0
        Rover.throttle =0
        Rover.mode = 'forward'
```

Responsible for moving the rover after collecting the rock and is displayed on the map to inform the user that a rock is being collected.



```python
    elif Rover.mode == 'looping':
        print('LOOPING')
        Rover.count = 0
        Rover.throttle = 0
        Rover.steer = -15 ##15
        Rover.brake = 0
        Rover.count += 1
        if Rover.count > 50:
            Rover.mode = 'forward'
            Rover.count = 0
```

in case the rover entered a loop where it moves around itself for a long time, this part handles the loop. But in our implementation in the previous conditionals, we tried our best to elimante thelooping. Also the mask used in the functin perception_step() fixed the looping of the rover.

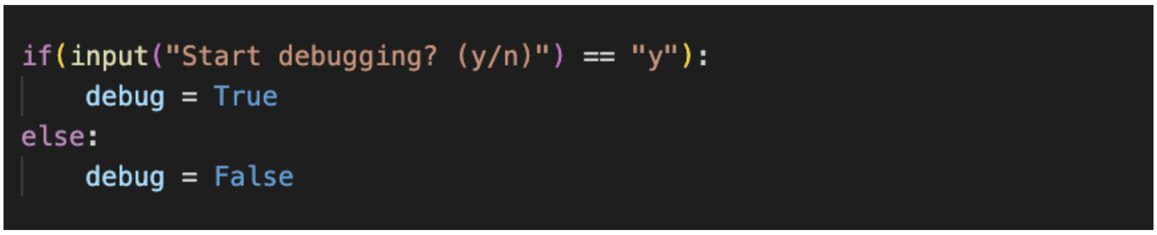


```python
# Just to make the rover do something
# even if no modifications have been made to the code
else:
    Rover.throttle = Rover.throttle_set
    Rover.steer = 0
    Rover.brake = 0

return Rover
```

Setting the acceleration(throttle) of the rover to 0.3 which is the default value and moving the rover.

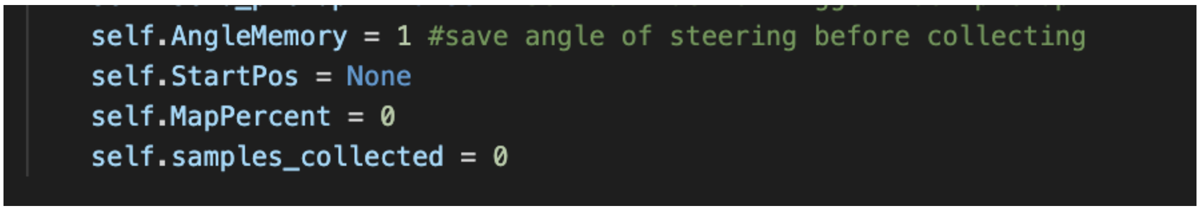
4. Drive_rover module

This is the code’s driver module.



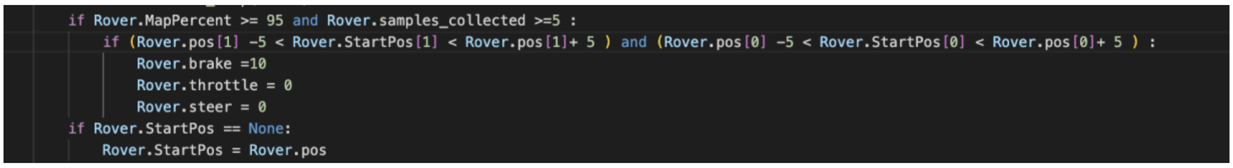
```python
if(input("Start debugging? (y/n)") == "y"):
    debug = True
else:
    debug = False
```

Responsible for the debugging module.



```python
    self.AngleMemory = 1 #save angle of steering before collecting
    self.StartPos = None
    self.MapPercent = 0
    self.samples_collected = 0
```

We added these extra variables because they will be helpful. AngleMemory stores the last steering angle to be used to steer the rover in the opposite direction after picking the rock to ensure that it wont loop. StartPos is the variable that stores the starting positions of the rover so it can help us return it back to the starting point when it finishes. MapPercent gets the percentage of the mapped covered so we can return the rover back to the starting point after it finishes. Samples_collected returns the collected rocks so that we can return the rover back to the starting point after it finishes



```python
    if Rover.MapPercent >= 95 and Rover.samples_collected >=5 :
        if (Rover.pos[1] -5 < Rover.StartPos[1] < Rover.pos[1]+ 5 ) and (Rover.pos[0] -5 < Rover.StartPos[0] < Rover.pos[0]+ 5 ) :
            Rover.brake =10
            Rover.throttle = 0
            Rover.steer = 0
    if Rover.StartPos == None:
        Rover.StartPos = Rover.pos
```

Added this function to support rover returning to starting position after it has collected 5 or more rocks and mapped at least 95% of the map, based on the variables and conditions set that we mentioned above.