COMPUTER VISION

Project

Team 12

Members

Joseph Fakher Farid	**18P2605**
Abdel Rahman Emam Ali Sadek Kassab	**18P3602**
Mostafa El Sherief	**16P8115**
Nour Eldin Talaat Ezzat	**18P3826**
Reda Mohsen Reda	**18P5141**

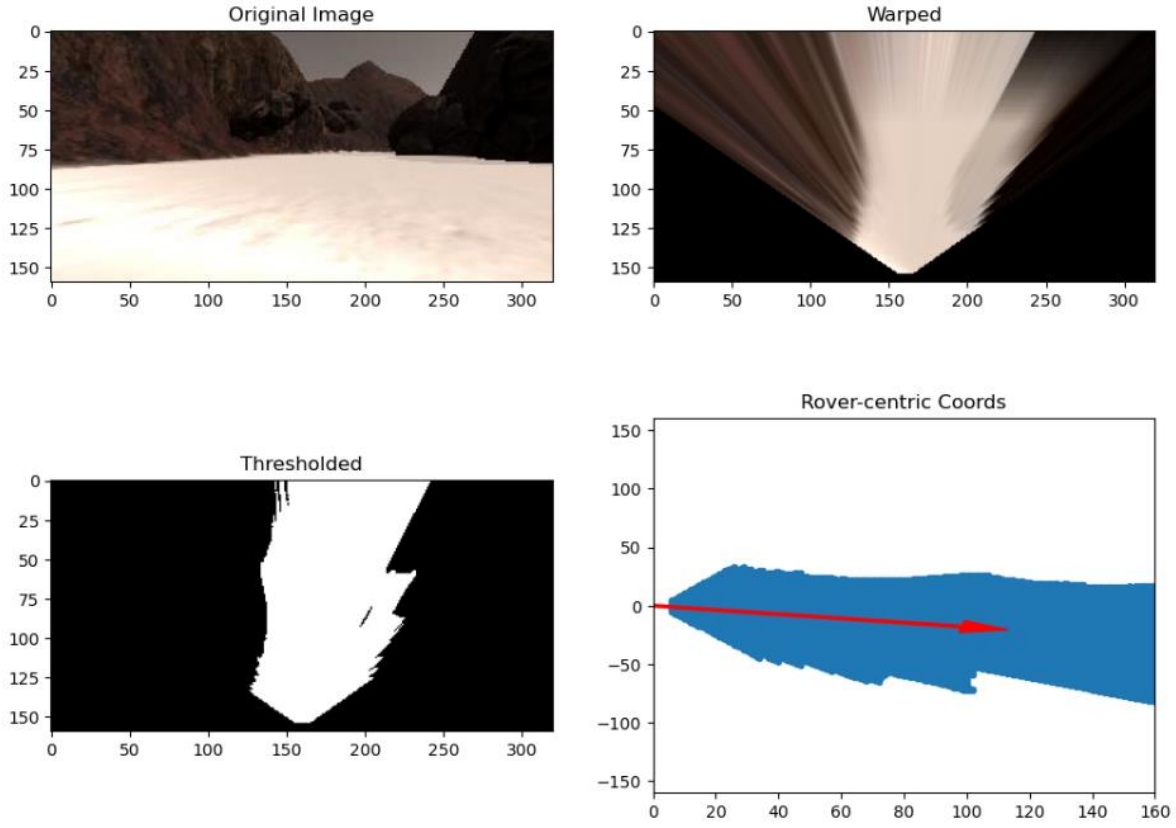
1 Github link:

Abdel-Rahman-Emam/CSE483-project (github.com)

2 Jupyter Notebook Phase 1 Output:

2.1 Mapping pixels:

Rotation and the translation are done to the image to map it to the world map from the rover perspective. And the result is clipped to the world array.



3 Perception.py:

3.1 Perception functions:

```
def color_thresh(img, rgb_thresh=(160,160,160),types=0):
    color_select =np.zeros_like(img[:,:,0])
    if types == 0:
        above_thresh = (img[:,:,0]>rgb_thresh[0])&(img[:,:,1]>rgb_thresh[1])&(img[:,:,2]>rgb_thresh[2])  # for navigable
    elif types == 1:
        above_thresh = (img[:,:,0]>rgb_thresh[0])&(img[:,:,1]>rgb_thresh[1])&(img[:,:,2]<rgb_thresh[2])#for rocks
    else:
        above_thresh = (img[:,:,0]>0)&(img[:,:,1]>0)&(img[:,:,2]>0)#for mask
    color_select[above_thresh] = 1
    return color_select
```

Here is a function for color thresholding to warped images

```python
# Define a function to convert from image coords to rover coords
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
```

Setting the coordinates from of the rover to be in the middle not far on the left.

```python
# Define a function to convert to radial coords in rover space
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
```

Change the coordinates from cartesian coordinates to polar form.

```python
# Define a function to map rover space pixels to world space
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
```

Change the coordinates to be angled with yaw as to be placed on the world map

```python
def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # Apply a scaling and a translation
    xpix_translated = (xpix_rot / scale) + xpos #to get the translated rover centric coordinates, we have to first divide by scale
    ypix_translated = (ypix_rot / scale) + ypos
    # Return the result
    return xpix_translated, ypix_translated
```

Remove the scaling if any is applied to be set on the world map

```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
```

Applying the rotation and translation discussed before in one function which returns the coordinates relative to the world

```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]))
    return warped, mask
```

Perform a perspective transformation on the image (np.array) to change the rover's camera image to a map image.

Step 1:

```python
    # 1) Define source and destination points for perspective transform


    dst_size = 5 #used a destination size of 5, for better fidelity, this does mess with collisions, need optimal number for phase 2
    bottom_offset = 6 #offset of the warped image from the bottom
    scale = 2 * dst_size  # used a scale of destination size*2, the scale does help in improving fidelit, however at the cost of collisions,
                          #  this is the size to be converted from when setting to world map
    image = Rover.img #get the rover image to do transformations on
    source = np.float32([[14, 140], [301, 140], [200, 96], [118, 96]]) #hardcoded numbers for phase 1, phase 2 this has to be automated
    destination = np.float32([[image.shape[1] / 2 - dst_size, image.shape[0] - bottom_offset],
                              [image.shape[1] / 2 + dst_size, image.shape[0] - bottom_offset],
                              [image.shape[1] / 2 + dst_size, image.shape[0] - bottom_offset - 2 * dst_size],
                              [image.shape[1] / 2 - dst_size, image.shape[0] - bottom_offset - 2 * dst_size] ])
```

Set a bottom offset where the rover will be and adding it to all the data

Setting the source by sending the values of the grid hardcoded

Setting the destination size to 5 as it increases the fidelity of the rover but the downside of this is the increase in the collision of the rover.

Passing the image to the rover so operate on it.

Step 2:

```python
    # 2) Apply perspective transform

    warped, mask = perspect_transform(image, source, destination)
    Rover.warped = warped
```

Change the perspective of the image and pass the warped to the rover to be saved

Step 3:

```
# 3) Apply color threshold to identify navigable terrain/obstacles/rock samples

threshed = color_thresh(warped)
#obstacleMap = np.absolute(np.float32(threshed) - 1) * color_thresh(warped, types=2)#had two approaches
#  to get the mask, first is the method of just multiplying any nonzero positive value by 1 manually
obstacleMap = np.absolute(np.float32(threshed) - 1) * mask # 2nd is by using numpy non-zero method
lower_yellow = np.array([24 - 5, 100, 100])
upper_yellow = np.array([24 + 5, 255, 255])
# Convert BGR to HSV
hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
# Threshold the HSV image to get only upper_yellow colors
rockMap = cv2.inRange(hsv, lower_yellow, upper_yellow)
rockMap = perspect_transform2(rockMap, source, destination)
```

Identifying the object using color thresholding.

Moving to the HSV color space and setting the limits in range of lower_yellow and upper_yellow.

Transforming the rockmap to have the ranges of the HSV.

Step 4:

```
# 4) Update Rover.vision_image (this will be displayed on left side of screen)
# Example: Rover.vision_image[:,:,0] =f obstacle color-thresholded binary image
#          Rover.vision_image[:,:,1] = rock_sample color-thresholded binary image
#          Rover.vision_image[:,:,2] = navigable terrain color-thresholded binary imagepitch
#if((Rover.pitch <1 and Rover.pitch > 359) and (Rover.roll < 1 and Rover.roll>359)):

Rover.vision_image[:, :, 2] = threshed * 255 #setting the blue channel to be navigable terrain
#Rover.navigable_thresh_image[:,:,2]= threshed*255 # seperating the navigable for debugging
Rover.vision_image[:, :, 1] = rockMap * 255#setting the green channel to be rocks
#Rover.rock_thresh_image[:,:,1] = rockMap * 255 #seperating the rock for debugging
Rover.vision_image[:, :, 0] = obstacleMap * 255 #setting the red channel to be obstacles, this will be the most dominant in the vision image
#Rover.obstacle_thresh_image[:,:,0] = obstacleMap * 255 #seperating the obstacles for debugging
idx = np.nonzero(Rover.vision_image)
Rover.vision_image[idx] = 255
```

Setting the colors of the objects the path is set to have the blue color as it is passed in the blue channel the rock in the green ad the obstacles in the red color

Step 5:

```
# 5) Convert map image pixel values to rover-centric coords

xpix, ypix = rover_coords(threshed)
rock_x, rock_y = rover_coords(rockMap)  # set the coordinates of the rock
obstaclexpix, obstacleypix = rover_coords(obstacleMap)
xpix, ypix = impose_range(xpix, ypix)
obstaclexpix, obstacleypix = impose_range(obstaclexpix, obstacleypix)
```

Getting coordinates of the objects :

1. Getting the rover coordinates and saving it in xpix,ypix

2. Getting the rock coordinates from rockMap and saving them in rock_x, rock_y
3. Getting the obstacles coordinates facing the rover and storing them in obstaclexpix, obstacleypix from the obstacleMap

```
# 6) Convert rover-centric pixel values to world coordinates

world_size = Rover.worldmap.shape[0]
x_world, y_world = pix_to_world(xpix, ypix, Rover.pos[0], Rover.pos[1], Rover.yaw, world_size, scale)
obstacle_x_world, obstacle_y_world = pix_to_world(obstaclexpix, obstacleypix, Rover.pos[0], Rover.pos[1], Rover.yaw, world_size, scale) #setting obstacles
                                                                                                                                        # in world map
rock_x_world, rock_y_world = pix_to_world(rock_x, rock_y, Rover.pos[0], Rover.pos[1], Rover.yaw, world_size, scale)# transform it to world coordinates
```

Getting the creating the x,y coordinates in the worldmap by passing the x,y of the rover and the yaw and the scale and operating on them as discussed before.

```
# 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] +S= 1
#          Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 1
#rock_xcen = rock_x_world[rock_idx]  # set the centers
#rock_ycen = rock_y_world[rock_idx]
if (Rover.pitch < 0.4 or Rover.pitch > 359.6) and (Rover.roll < 0.4 or Rover.roll > 359.6):
    Rover.worldmap[y_world, x_world, 2] += 1 #set that we found navigable terrain in the world, color is dominant in world map
    Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1  #set that we found obstacles in map, just a bit lighter
    Rover.worldmap[rock_y_world, rock_x_world, 1] = 255  # set the green channel on the worldmap with the location of the rock
Rover.worldmap = np.clip(Rover.worldmap, 0, 255)
```

If the pitch or the roll is not in the range then the rover will not get any input but if in range then set the set the worldmap of the rover with the values of the rocks and obstacles and the terrain in the corresponding channels yellow, red, and green.

```
# 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)
rock_dist, rock_ang = to_polar_coords(rock_x, rock_y)
Rover.nav_angles = angles #set the rover angles
Rover.nav_dists = dist #set the rover distances, this can be seen in the pipeline, and how it works
Rover.samples_angle = rock_ang
Rover.samples_dist = rock_dist
if Rover.start_pos is None:
    Rover.start_pos = (Rover.pos[0], Rover.pos[1])
    print('STARTING POSITION IS: ', Rover.start_pos)

return Rover
```

Get the distance and the angels to polar format and set the navigate angles in rover with the angels and the distance of it with the distance value and do the same with the rocks in the samples_angle and the sample_dist

And save the initial value of the rover before any movement to return to it after it is done.

4 drive_rover:

```
class RoverState():
    def __init__(self):
        self.start_time = None # To record the start time of navigation
        self.total_time = None # To record total duration of navigation
        self.stuck_time = 0
        self.rock_time = 0
        self.start_pos = None # Position (x, y) of the starting location
        self.img = None # Current camera image
        self.pos = None # Current position (x, y)
        self.yaw = None # Current yaw angle
        self.pitch = None # Current pitch angle
        self.roll = None # Current roll angle
        self.vel = None # Current velocity
        self.steer = 0 # Current steering angle
        self.throttle = 0 # Current throttle value
        self.brake = 0 # Current brake value
        self.nav_angles = None # Angles of navigable terrain pixels
        self.nav_dists = None # Distances of navigable terrain pixels
        self.ground_truth = ground_truth_3d # Ground truth worldmap
        self.mode = 'forward' # Current mode (can be forward or stop)
        self.throttle_set = 0.2 # Throttle setting when accelerating
        self.brake_set = 10 # Brake setting when braking
```

some of the state of the rover, start_pos was added to identify the starting position so that the rover can return to it after collecting at least 5 rocks.

Also made a variable for stuck_time so it can escape being stuck in situations, and rock time to be able to escape trying to capture a rock if it takes too long (timeout).

```
    # The stop_forward and go_forward fields below represent total count
    # of navigable terrain pixels.  This is a very crude form of knowing
    # when you can keep going and when you should stop.  Feel free to
    # get creative in adding new fields or modifying these!
    self.stop_forward = 200 # Threshold to initiate stopping
    self.go_forward = 500 # Threshold to go forward again
    self.max_vel = 8 # Maximum velocity (meters/second)
    # Image output from perception step
    # Update this image to display your intermediate analysis steps
    # on screen in autonomous mode
    self.vision_image = np.zeros((160, 320, 3), dtype=np.float)
    #the following images were added to the rover in order to display the debugging mode w
    #according to their names
    #warped view
    self.warped_image = np.zeros((160, 320, 3), dtype = np.float)
    #Map threshold
    self.threshold_image = np.zeros((160, 320, 3), dtype = np.float)
    #Rock threshold
    self.rock_thresh_image = np.zeros((160, 320, 3), dtype = np.float)
    #Obstacle threshold
    self.obstacle_thresh_image = np.zeros((160, 320, 3), dtype = np.float)
    #Navigable threshold
    self.navigable_thresh_image = np.zeros((160, 320, 3), dtype = np.float)
    # Worldmap
```

Increased the maximum velocity allowed for the rover to drive at, and added some states for the debugging mode which will store images corresponding to their names at a file in the code folder.

```
    # Worldmap
    # Update this image with the positions of navigable terrain
    # obstacles and rock samples
    self.worldmap = np.zeros((200, 200, 3), dtype=np.float)
    self.samples_pos = None # To store the actual sample positions
    self.samples_dist = None
    self.samples_angle = None
    self.samples_close = False
    self.samples_to_find = 0 # To store the initial count of samples
    self.samples_located = 0 # To store number of samples located on map
    self.samples_collected = 0 # To count the number of samples collected
    self.near_sample = 0 # Will be set to telemetry value data["near_sample"]
    self.picking_up = 0 # Will be set to telemetry value data["picking_up"]
    self.send_pickup = False # Set to True to trigger rock pickup
```

Initialized a samples angle, dist and close. Samples_close is used to identify whether there's a rock close or not, and if so, samples_dist and samples_angle is calculated to the rock.

Also added samples_collected variable so that it counts, the number of rocks that are collected.

```
# the following lines in the code are used to make new debugging for each run
if debugging:
    [f.unlink() for f in Path(os.getcwd()+"/navthresh").glob("*") if f.is_file()]
    [f.unlink() for f in Path(os.getcwd()+"/obstaclethresh").glob("*") if f.is_file()]
    [f.unlink() for f in Path(os.getcwd()+"/warped").glob("*") if f.is_file()]
    [f.unlink() for f in Path(os.getcwd()+"/rockthresh").glob("*") if f.is_file()]
    [f.unlink() for f in Path(os.getcwd()+"/visimg").glob("*") if f.is_file()]
    [f.unlink() for f in Path(os.getcwd()+"/map").glob("*") if f.is_file()]
# Variables to track frames per second (FPS)
```

For debugging to refresh the folder after every run. Basically, deletes the old run and stores the images of the new run.

4.1 Telemetry function:

```
# Define telemetry function for what to do with incoming data
@sio.on('telemetry')
def telemetry(sid, data):

    global frame_counter, second_counter, fps
    frame_counter+=1
    # Do a rough calculation of frames per second (FPS)
    if (time.time() - second_counter) > 1:
        fps = frame_counter
        frame_counter = 0
        second_counter = time.time()
    #print("Current FPS: {}".format(fps))

    if data:
        global Rover
        # Initialize / update Rover with current telemetry
        Rover, image = update_rover(Rover, data)

        if np.isfinite(Rover.vel):

            # Execute the perception and decision steps to update the Rover's state
            Rover = perception_step(Rover)
            print("STARTING POSITION IS", Rover.start_pos)
            Rover = decision_step(Rover)
```

Checks for the fps to print it at first, then checks if there's an input stream of data from the Rover. If found, the rover and the camera image is then received from the update rover function.

Then update the rover by applying the perception and decision step.

```
    # Create output images to send to server
    out_image_string1, out_image_string2 = create_output_images(Rover)

    # The action step!  Send commands to the rover!

    # Don't send both of these, they both trigger the simulator
    # to send back new telemetry so we must only send one
    # back in respose to the current telemetry data.

    # If in a state where want to pickup a rock send pickup command
    if Rover.send_pickup and not Rover.picking_up:
        send_pickup()
        # Reset Rover flags
        Rover.send_pickup = False
    else:
        # Send commands to the rover!
        commands = (Rover.throttle, Rover.brake, Rover.steer)
        send_control(commands, out_image_string1, out_image_string2)

# In case of invalid telemetry, send null commands
```

Receive the output images from the create output images in supporting functions which produces the weighted map and the color threshed vision image.

And send a pickup command if pick up flag is raised and the robot is not picking up, and send the throttle, brake and steer commands if not picking up.

```
    # In case of invalid telemetry, send null commands
    else:

        # Send zeros for throttle, brake and steer and empty images
        send_control((0, 0, 0), '', '')

    # If you want to save camera images from autonomous driving specify a path
    # Example: $ python drive_rover.py image_folder_path
    # Conditional to save image frame if folder was specified
    if args.image_folder != '':
        timestamp = datetime.utcnow().strftime('%Y_%m_%d_%H_%M_%S_%f')[:-3]
        image_filename = os.path.join(args.image_folder, timestamp)
        image.save('{}.jpg'.format(image_filename))
```

If invalid telemetry is discovered, send 0 and nullstring commands.

Then the 2nd if condition is used to save the images from the rover in case of recording command In autonomous mode.

```python
@sio.on('connect')
def connect(sid, environ):
    print("connect ", sid)
    send_control((0, 0, 0), '', '')
    sample_data = {}
    sio.emit(
        "get_samples",
        sample_data,
        skip_sid=True)
```

When connecting to socket io, initialize data.

```python
def send_control(commands, image_string1, image_string2):
    # Define commands to be sent to the rover
    data={
        'throttle': commands[0].__str__(),
        'brake': commands[1].__str__(),
        'steering_angle': commands[2].__str__(),
        'inset_image1': image_string1,
        'inset_image2': image_string2
        }
    # Send commands via socketIO server
    sio.emit(
        "data",
        data,
        skip_sid=True)
    eventlet.sleep(0)
# Define a function to send the "pickup" command
```

This is the function that sends the control data to the rover in the simulator.

```python
# Define a function to send the "pickup" command
def send_pickup():
    print("Picking up")
    pickup = {}
    sio.emit(
        "pickup",
        pickup,
        skip_sid=True)
    eventlet.sleep(0)
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Remote Driving')
    parser.add_argument(
        'image_folder',
        type=str,
        nargs='?',
        default='',
        help='Path to image folder. This is where the images from the run will be saved.'
    )
    args = parser.parse_args()
```

Send_pickup is the function that sends the pickup command to the rover in the simulator, and the if condition is used to parse arguments to save the image to to the folder.

```python
    args = parser.parse_args()

    #os.system('rm -rf IMG_stream/*')
    if args.image_folder != '':
        print("Creating image folder at {}".format(args.image_folder))
        if not os.path.exists(args.image_folder):
            os.makedirs(args.image_folder)
        else:
            shutil.rmtree(args.image_folder)
            os.makedirs(args.image_folder)
        print("Recording this run ...")
    else:
        print("NOT recording this run ...")

    # wrap Flask application with socketio's middleware
    app = socketio.Middleware(sio, app)

    # deploy as an eventlet WSGI server
    eventlet.wsgi.server(eventlet.listen(('', 4567)), app)
```

check if the image folder doesn't exist to create it when recording, however if user doesn't want to record just print not recording.

5 Decision.py

```
# offset in rad used to hug the left wall.
offset = 0
# Only apply left wall hugging when out of the starting point (after 10s)
# to avoid getting stuck in a circle
if Rover.total_time > 10:
    # Steering proportional to the deviation results in
    # small offsets on straight lines and
    # large values in turns and open areas
    offset = 0.7 * np.std(Rover.nav_angles)
```

Offset done to move the rover near the wall is applied after 10 seconds from running and is done as it changes dynamically not having a static value is changes the value depending on the path if it is a wide path the offset is large if not the offset is decreased.

5.1 Returning to home:

```
# Check if we have vision data to make decisions with
if Rover.samples_collected >= 5:
    print('RETURNING HOME')
    if abs(Rover.pos[0] - Rover.start_pos[0]) < 5 and abs(Rover.pos[1] - Rover.start_pos[1]) < 5:
        Rover.throttle = 0
        Rover.brake = Rover.brake_set
        Rover.steer = 0
        print('RETURNED HOME!!!! BEAM ME UP!!!')
        return Rover
```

If the samples collected is greater than 5 then the rover will set its directions to move back to home.

The rove compares the position 0 (x axis) form the points its in now and the start position and same in the y axis and checks if it is less than 5 then

a. Set the throttle to 0 apply the brakes and set steer to default state (0) to stop the rover.

5.2 Forward Mode:

```
if Rover.mode == 'forward':
    # if sample rock on sight (in the left side only) and relatively close
    if Rover.samples_angle is not None and np.mean(Rover.samples_angle) > -0.3 and np.min(Rover.samples_dist) < 40:
        # Rover.steer = np.clip(np.mean(Rover.samples_angles * 180 / np.pi), -15, 15)
        Rover.rock_time = Rover.total_time
        Rover.mode = 'rock'

    # Check the extent of navigable terrain
    elif len(Rover.nav_angles) >= Rover.stop_forward:
        # If mode is forward, navigable terrain looks good
        # Except for start, if stopped means stuck.
        # Alternates between stuck and forward modes
        if Rover.vel <= 0.1 and Rover.total_time - Rover.stuck_time > 4:
            # Set mode to "stuck" and hit the brakes!
            Rover.throttle = 0
            # Set brake to stored brake value
            Rover.brake = Rover.brake_set
            Rover.steer = 0
            Rover.mode ='stuck'
            Rover.stuck_time = Rover.total_time
        # if velocity is below max, then throttle
        elif Rover.vel < Rover.max_vel:
            # Set throttle value to throttle setting
            Rover.throttle = Rover.throttle_set
        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)
        # Hug left wall by setting the steer angle slightly to the left
        Rover.steer = np.clip(np.mean((Rover.nav_angles + offset) * 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 or Rover.vel <= 0:
        # 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'
```

There are 3 main conditions in the Forward state:

1. Found rock:
    a. If the sample angles from the images is not empty and the mean of them greater than -0.3 and the distance less than 40 which means the angles is of the rock is less than -0.3 and the distance of it and the rover is less than 40, then we set the rock time to total time to record the time of finding a rock and set the state of the rover to rock to begin capturing the rock.
2. Check the extent of the navigable terrain:
    a. Check the path
    b. The path looks good but the rover slows and the stuck time of the rover exceeds 4 sec
        i. That means the rover is stuck so the we set the brake
        ii. Restore the steer to the default state
        iii. Set the mode to Stuck to handle the stuck in the stuck condition
        iv. And update the stuck time to the total time to begin calculating the time of stuck from after the rover is set to stuck

c. Else check if the velocity is less than the maximum velocity so we set the throttle to its default value to increase the speed of the rover to reach the maximum speed.
d. Else get near wall:
    i. Set the throttle to 0
    ii. Release the brakes
    iii. Then begin setting the steer by calculating the mean of the navigation angles and adding the offset to get near the wall on the left as if the rover finds any rock it can collect it and the possibility of finding one would increase.

3. Stop if path not enough to continue:
    a. If the path is not enough then set the throttle to 0
    b. Apply the brakes
    c. Set the steer to the default state and set the rover to stop to handle this case as the rover in the stop state will change its direction and would search for a navigable path.

5.3 Stuck Mode:

```
70
71          # If we're already in "stuck". Stay here for 1 sec
72      elif Rover.mode == 'stuck':
73          # if 1 sec passed go back to previous mode
74          if Rover.total_time - Rover.stuck_time > 1:
75              # Set throttle back to stored value
76              Rover.throttle = Rover.throttle_set
77              # Release the brake
78              Rover.brake = 0
79              # Set steer to mean angle
80              # Hug left wall by setting the steer angle slightly to the left
81              Rover.steer = np.clip(np.mean((Rover.nav_angles + offset) * 180 / np.pi), -15, 15)
82              Rover.mode = 'forward'  # returns to previous mode
83          # Now we're stopped and we have vision data to see if there's a path forward
84          else:
85              Rover.throttle = 0
86              # Release the brake to allow turning
87              Rover.brake = 0
88              # Turn range is +/- 15 degrees, when stopped the next line will induce 4-wheel turning
89              # Since hugging left wall steering should be to the right:
90              Rover.steer = -15
```

In the Stuck state the rover the stuck time value and compares it with the total time to get the actual time that the rover is stuck and not moving the rover sets the stuck time in the forward mode.

1. If the stuck time greater than 1:
    a. The rover is now directed in another position to go from the stuck state to the forward state.
    b. Set the throttle to its default value (0.3)
    c. Release the brake (set to 0)
    d. Setting the rover steer to get the mean of the values of the nav_angles to make the rover calculate the direction which has the actual path that the rover can move through and adding the offset to it.
    e. Lastly setting the rover mode to Forward
2. If the stuck time less:
    a. The state of the is stuck and this is the first second being stuck
    b. The rover sets the throttle to 0
    c. Releases the brake

d. As to move the steer to by -15 degrees as the rover change the direction to change the stuck position of the rover

5.4 Rock Mode:

```
elif Rover.mode == 'rock':
    # Steer torwards the sample
    mean = np.mean(Rover.samples_angle * 180 / np.pi)
    if not np.isnan(mean):
        Rover.steer = np.clip(mean, -15, 15)
    else:
        Rover.mode = 'forward'  # no rock in sight anymore. Go back to previous state

    # if 20 sec passed gives up and goes back to previous mode
    if Rover.total_time - Rover.rock_time > 20:
        Rover.mode = 'forward'  # returns to previous mode

    # if close to the sample stop
    if Rover.near_sample:
        # Set mode to "stop" and hit the brakes!
        Rover.throttle = 0
        # Set brake to stored brake value
        Rover.brake = Rover.brake_set

    # if got stuck go to stuck mode
    elif Rover.vel <= 0 and Rover.total_time - Rover.stuck_time > 10:
        Rover.throttle = 0
        # Set brake to stored brake value
        Rover.brake = Rover.brake_set
        Rover.steer = 0
        Rover.mode = 'stuck'
        Rover.stuck_time = Rover.total_time
    else:
        # Approach slowly
        slow_speed = Rover.max_vel / 2
        if Rover.vel < slow_speed:
            Rover.throttle = 0.2
            Rover.brake = 0
        else:  # Else break
            Rover.throttle = 0
            Rover.brake = Rover.brake_set
```

In the rock mode the rover gets the mean of the sampled data to move towards the rock

1. if the samples is empty the rover change the state to forward if not the steer is set to move by range from -15 to 15 by the mean calculated.
2. If the rover gets more than 20 seconds the rover moves forward and ignores the rock as it could not get to it
3. If the rover gets near the rock the rovers stops by applying the brake and setting the throttle to 0

4. If the rover velocity less than 0 and the stuck time greater than 10 seconds then the rover moves to the stuck state but first it applies the brake and set the steer to 0 and updates the stuck time to the time after setting the rover to the stuck mode.
5. Else check if the velocity of the rover less than half of its value modify the throttle to the default value (0.2) and release the brake.
6. Else then apply the brake and set the throttle to 0.

5.5 Stop State:

```
128
129         # If we're already in "stop" mode then make different decisions
130         elif Rover.mode == 'stop':
131             # If we're in stop mode but still moving keep braking
132             if Rover.vel > 0.2:
133                 Rover.throttle = 0
134                 Rover.brake = Rover.brake_set
135                 Rover.steer = 0
136             # If we're not moving (vel < 0.2) then do something else
137             elif Rover.vel <= 0.2:
138                 # Now we're stopped and we have vision data to see if there's a path forward
139                 if len(Rover.nav_angles) < Rover.go_forward:
140                     Rover.throttle = 0
141                     # Release the brake to allow turning
142                     Rover.brake = 0
143                     # Turn range is +/- 15 degrees, when stopped the next line will induce 4-wheel turning
144                     # Since hugging left wall steering should be to the right:
145                     Rover.steer = -15
146                 # If we're stopped but see sufficient navigable terrain in front then go!
147                 if len(Rover.nav_angles) >= Rover.go_forward:
148                     # Set throttle back to stored value
149                     Rover.throttle = Rover.throttle_set
150                     # Release the brake
151                     Rover.brake = 0
152                     # Set steer to mean angle
153                     # Hug left wall by setting the steer angle slightly to the left
154                     offset = 12
155                     Rover.steer = np.clip(np.mean(Rover.nav_angles * 180 / np.pi) + offset, -15, 15)
156                     Rover.mode = 'forward'  # returns to previous mode
```

The decision making when the rover is in the Stop State:

There are 2 main points in the stop state depending on the velocity

1. Velocity bigger than 2
2. Velocity less than 2

5.5.1 Velocity bigger than 2:

We set the throttle to 0, the steer to 0, and the brake to stop the rover and rest the direction of the wheels to the default position to make the rover stop completely as the rover is ordered to stop.

5.5.2 Velocity less than 2:

The rover is now getting the data to process if there is a path forward or not

This is made by the compression between the length of the nparray nav_angles of the rover and checking if it is less than the go_forward or greater or equals, which means if the vision of the rover appears that the path is not enough for the rover to go forward.

1. If the path is not enough:

a. we release the brake, set the throttle to 0, and make the rover steer by angle -15 (can be +/- value) to make the rover turn in its position and get away from the obstacle in front of the rover.

2. If the path is enough:
    a. then move the rover by setting the throttle to its constant value (0.3) and set brake to 0 as to allow the rover to move.
    b. Setting an offset to the left with (12) to make the rover slightly go to the left as to increase the chance of the rover meeting a rock
    c. Setting the rover steer to get the mean of the values of the nav_angles to make the rover calculate the direction which has the actual path that the rover can move through and adding the offset to it.
    d. Lastly setting the rover state from the stop state to the forward state.

5.6 Picking up the rocks:

```
164
165     # If in a state where want to pickup a rock send pickup command
166     if Rover.near_sample and Rover.vel == 0 and not Rover.picking_up:
167         Rover.send_pickup = True
```

The decision file always checks if the rover is near a sample and it stopped to it and the rover did not order to pick it up yet to order the rover once to pick the rock and collect it.

6 Supporting Functions:

```
x=0
debugging = False
def convert_to_float(string_to_convert):
    if ',' in string_to_convert:
        float_value = np.float(string_to_convert.replace(',', '.'))
    else:
        float_value = np.float(string_to_convert)
    return float_value
```

Initialized an x variable, and a debugging variable for the debugging mode, setting the debugging mode to true will output some images to the folders that will later be discussed in the explanation of this file.

6.1 convert_to_float function

Furthermore, the function convert_to_float is made which basically makes float variables readable in np from strings. This function was premade in the file given.

```
def update_rover(Rover, data):
    # Initialize start time and sample positions
    if Rover.start_time == None:
        Rover.start_time = time.time()
        Rover.total_time = 0
        samples_xpos = np.int_([convert_to_float(pos.strip())
                                for pos in data["samples_x"].split(';')])
        samples_ypos = np.int_([convert_to_float(pos.strip())
                                for pos in data["samples_y"].split(';')])
        Rover.samples_pos = (samples_xpos, samples_ypos)
        Rover.samples_to_find = np.int(data["sample_count"])
    # Or just update elapsed time
```

6.2 update_rover function

This function starts by initializing the start time at which the rover started automating and the total time that is being taken in the run. Further it starts recorder the x and y positions of the rock samples to find.

```
    # Or just update elapsed time
    else:
        tot_time = time.time() - Rover.start_time
        if np.isfinite(tot_time):
            Rover.total_time = tot_time
```

Calculate the total time by subtracting the current time from the start time recorded at the start of the run.

```
# The current speed of the rover in m/s
Rover.vel = convert_to_float(data["speed"])
# The current position of the rover
Rover.pos = [convert_to_float(pos.strip()) for pos in data["position"].split(';')]
# The current yaw angle of the rover
Rover.yaw = convert_to_float(data["yaw"])
# The current yaw angle of the rover
Rover.pitch = convert_to_float(data["pitch"])
# The current yaw angle of the rover
Rover.roll = convert_to_float(data["roll"])
# The current throttle setting
Rover.throttle = convert_to_float(data["throttle"])
# The current steering angle
Rover.steer = convert_to_float(data["steering_angle"])
# Near sample flag
Rover.near_sample = np.int(data["near_sample"])
# Picking up flag
Rover.picking_up = np.int(data["picking_up"])
# Update number of rocks collected
Rover.samples_collected = Rover.samples_to_find - np.int(data["sample_count"])
```

This block of code takes the values from the data taken in real-time from the rover, and then is set inside the rover class with the corresponding values.

```
imgString = data["image"]
image = Image.open(BytesIO(base64.b64decode(imgString)))
Rover.img = np.asarray(image)
```

This block reads the image value taken from the rover, opens it, and saves the values as an np array.

6.3 create_output_images function

```
def create_output_images(Rover):
    global x
    # Create a scaled map for plotting and clean up obs/nav pixels a bit
    if np.max(Rover.worldmap[:, :, 2]) > 0:
        nav_pix = Rover.worldmap[:, :, 2] > 0
        navigable = Rover.worldmap[:, :, 2] * (255 / np.mean(Rover.worldmap[nav_pix, 2]))
    else:
        navigable = Rover.worldmap[:, :, 2]
    if np.max(Rover.worldmap[:, :, 0]) > 0:
        obs_pix = Rover.worldmap[:, :, 0] > 0
        obstacle = Rover.worldmap[:, :, 0] * (255 / np.mean(Rover.worldmap[obs_pix, 0]))
    else:
        obstacle = Rover.worldmap[:, :, 0]
```

This function takes in the rover to provide the output images into the simulator.

First of all, it takes in the navigable if they exist, from the worldmap and sets a navigable variable and does the same for the obstacles.

```
likely_nav = navigable >= obstacle
obstacle[likely_nav] = 0
plotmap = np.zeros_like(Rover.worldmap)
plotmap[:, :, 0] = obstacle
plotmap[:, :, 2] = navigable
plotmap = plotmap.clip(0, 255)
# Overlay obstacle and navigable terrain map with ground truth map
map_add = cv2.addWeighted(plotmap, 1, Rover.ground_truth, 0.5, 0)

# Check whether any rock detections are present in worldmap
```

It then starts to calculate the likely navigable terrain to calculate the fidelity later on, and sets the plot with the obstacles and the navigable terrain, clipping the minimum at 0 and the maximum at 255. And then the map is added using openCV's addWeighted which overlays the images.

```
# Check whether any rock detections are present in worldmap
rock_world_pos = Rover.worldmap[:, :, 1].nonzero()
# If there are, we'll step through the known sample positions
# to confirm whether detections are real
samples_located = 0
if rock_world_pos[0].any():

    rock_size = 2
    for idx in range(len(Rover.samples_pos[0])):
        test_rock_x = Rover.samples_pos[0][idx]
        test_rock_y = Rover.samples_pos[1][idx]
        rock_sample_dists = np.sqrt((test_rock_x - rock_world_pos[1])**2 +
                                    (test_rock_y - rock_world_pos[0])**2)
        # If rocks were detected within 3 meters of known sample positions
        # consider it a success and plot the location of the known
        # sample on the map
        if np.min(rock_sample_dists) < 3:
            samples_located += 1
            map_add[test_rock_y-rock_size:test_rock_y+rock_size,
                    test_rock_x-rock_size:test_rock_x+rock_size, :] = 255
```

This next block, it starts to plot the rocks samples on the world map, with a slightly larger size, it registers its x and y values then calculates the distance to it, if the smallest distance from the rover to the rock is within 3 units, it increments the samples located and adds it to the map.

```
# Calculate some statistics on the map results
# First get the total number of pixels in the navigable terrain map
tot_nav_pix = np.float(len((plotmap[:, :, 2].nonzero()[0])))
# Next figure out how many of those correspond to ground truth pixels
good_nav_pix = np.float(
    len(((plotmap[:, :, 2] > 0) & (Rover.ground_truth[:, :, 1] > 0)).nonzero()[0]))
# Next find how many do not correspond to ground truth pixels
bad_nav_pix = np.float(
    len(((plotmap[:, :, 2] > 0) & (Rover.ground_truth[:, :, 1] == 0)).nonzero()[0]))
# Grab the total number of map pixels
tot_map_pix = np.float(len((Rover.ground_truth[:, :, 1].nonzero()[0])))
# Calculate the percentage of ground truth map that has been successfully found
perc_mapped = round(100*good_nav_pix/tot_map_pix, 1)
# Calculate the number of good map pixel detections divided by total pixels
# found to be navigable terrain
if tot_nav_pix > 0:
    fidelity = round(100*good_nav_pix/(tot_nav_pix), 1)
else:
    fidelity = 0
```

In this block of code, the total navigable pixels are first received from the plot, and same goes for the good nav pix and the bad nav pixels. Finally to calculate the fidelity, the good navigable pixels are divided by the total navigable pixels. And the percentage mapped is calculated by also dividing the good nav pixels by the total map pixels.

```
# Flip the map for plotting so that the y-axis points upward in the display
map_add = np.flipud(map_add).astype(np.float32)
# Add some text about map and rock sample detection results
cv2.putText(map_add, "Time: "+str(np.round(Rover.total_time, 1))+' s', (0, 10),
            cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
cv2.putText(map_add, "Mapped: "+str(perc_mapped)+'%', (0, 25),
            cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
cv2.putText(map_add, "Fidelity: "+str(fidelity)+'%', (0, 40),
            cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
cv2.putText(map_add, "Rocks", (0, 55),
            cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
cv2.putText(map_add, "  Located: "+str(samples_located), (0, 70),
            cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
cv2.putText(map_add, "  Collected: "+str(Rover.samples_collected), (0, 85),
            cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
cv2.putText(map_add, "State: "+str(Rover.mode), (0, 145),
            cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
# Convert map and vision image to base64 strings for sending to server
```

This block just adds text containing statistics to the map image.

```
    global debugging
    pil_img = Image.fromarray(map_add.astype(np.uint8))
    buff = BytesIO()
    if debugging:#if we're in debugging mode, we're gonna start saving images from the pipeline
        pil_img.save("map/map"+str(x)+".jpeg")
    pil_img.save(buff, format="JPEG")
    encoded_string1 = base64.b64encode(buff.getvalue()).decode("utf-8")

    pil_img = Image.fromarray(Rover.vision_image.astype(np.uint8))
    buff = BytesIO()
    if debugging:#if we're in debugging mode, we're gonna start saving images from the pipeline
        pil_img.save("navthresh/visimg"+str(x)+".jpeg")
    pil_img.save(buff, format="JPEG")
    encoded_string2 = base64.b64encode(buff.getvalue()).decode("utf-8")
    if debugging: #if we're in debugging mode, we're gonna start saving images from the pipeline
        pil_img = Image.fromarray(Rover.navigable_thresh_image.astype(np.uint8))
        buff = BytesIO()
        pil_img.save("visimg/navthresh"+str(x)+".jpeg")

        pil_img = Image.fromarray(Rover.warped.astype(np.uint8))
        buff = BytesIO()
        pil_img.save("warped/warped"+str(x)+".jpeg")

        pil_img = Image.fromarray(Rover.rock_thresh_image.astype(np.uint8))
        buff = BytesIO()
```

```
        pil_img = Image.fromarray(Rover.rock_thresh_image.astype(np.uint8))
        buff = BytesIO()
        pil_img.save("rockthresh/rockthresh"+str(x)+".jpeg")

        pil_img = Image.fromarray(Rover.obstacle_thresh_image.astype(np.uint8))
        buff = BytesIO()
        pil_img.save("obstaclethresh/obstaclethresh"+str(x)+".jpeg")
        x+=1


    return encoded_string1, encoded_string2
```

Debugging mode, which just outputs images corresponding to the run if debugging is set to 1. Rock, obstacle and navigable thresholds images are printed, and the warped perspective as well.

7 Screen Shots of the running Rover:

Fidelity: