Faculty of Engineering Ain Shams University

Computer Engineering and Software Systems Program (CESS)

Fall 2022

Course Name: Computer Vision

Course Code: CSE 483

Project Phase 2 Report

Names	ID
Adham Ahmed Mohamed Mohamed Abdelmaksoud	19P1250
AbdulRaouf Monir Kamal Mahmoud	19P4442
Osama Ayman Mokhtar Amin	19P1609

Table of Contents

1.0 Code Explanation

1.1 Perception.py

```
# 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
```

This function sets a threshold over the red, green, and blue colors. It returns an array of all the pixels whose intensity is higher than the mentioned threshold in all of the red, green, and blue channels.

```
# 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
```

This function translates the pixels of a given image to their position with respect to the rover according to the x and y coordinates of the rover position.

```
# 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
    dists = np.sqrt(x_pixel**2 + y_pixel**2)
    # Calculate angle away from vertical for each pixel
    angles = np.arctan2(y_pixel, x_pixel)
    return dists, angles
```

This function calculates the polar coordinates from cartesian coordinates by calculating the magnitude and direction of each pixel.

```
# Define a function to apply a rotation to pixel positions
def rotate_pix(xpix, ypix, yaw):
    # Convert yaw to radians
    yaw_rad = yaw * np.pi / 180
    # Apply a rotation
    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
```

This function rotates the given pixels with a given angle (yaw) about the origin

```
# Define a function to perform a translation
def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # Apply a scaling and a translation
    xpix_translated = (xpix_rot / scale) + xpos
    ypix_translated = (ypix_rot / scale) + ypos
    # Return the result
    return xpix_translated, ypix_translated
```

This function translates and scales the given pixels with a given x value (x coordinate of rover position), y value (y coordinate of rover position), and a scale

```
# 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
```

This function performs geometric transformation (translation, scaling, and rotation) on the given pixels. This function is specific to transform pixels from the point of view of the rover to the world map point of view.

```
# Define a function to perform a perspective transform
def perspect_transform(img, src, dst):
    # obatin a perspective transform matrix
    M = cv2.getPerspectiveTransform(src, dst)
    # transform
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))# keep same
size as input image
    return warped
```

This function gets the bird eye view of a given image according to reference source coordinates and destination coordinates.

1.2 Decision.py:

1) Forward Mode

```
if Rover.mode == 'forward':
            # Check the extent of navigable terrain
            if (len(Rover.nav_angles) >= Rover.stop_forward) and \
                 (np.max(Rover.nav_dists > 20)):
                # If mode is forward, navigable terrain looks good
                # and velocity is below max and not turning, then throttle
                if (Rover.vel < Rover.max_vel) and (Rover.steer < 5):
                    # Set throttle value to throttle setting
                    Rover.steer = np.clip(np.mean(Rover.nav_angles *
180/np.pi), -15, 15)
                    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)
            # If there's a lack of navigable terrain pixels then go to
'stop' mode
            elif (len(Rover.nav_angles) < Rover.stop_forward) or \
                 (np.mean(Rover.nav_dists < 20)): # modified
                    # Set mode to "stop" and hit the brakes!
                    Rover.throttle = 0
                    # Set brake to stored brake value
                    Rover.brake = Rover.brake_set
                    Rover.steer = -15 # modified
                    Rover.mode = 'stop'
            if (np.abs(Rover.vel) <= 0.05):
                Rover.stuck_time += 1
            else:
                Rover.stuck_time = 0
            if Rover.stuck_time > Rover.error_limit:
                Rover.throttle = -Rover.throttle_set
                Rover.steer = -np.clip(np.mean(Rover.nav_angles *
180/np.pi), -15, 15)
                time.sleep(0.5)
```

This mode simply checks if the navigable terrain pixels is greater than the stopping threshold, if yes, it throttles, else it stops. It handles the case where the rover is stuck and the handling is explained in the last section of the document.

2) Stop Mode

```
  elif Rover.mode == 'stop':

            if Rover.vel > 0.5:
                Rover.throttle = 0
                Rover.brake = Rover.brake_set
                Rover.steer = 0
            elif Rover.vel <= 0.5:
                if len(Rover.nav_angles) < Rover.go_forward:
                    Rover.throttle = 0
                    Rover.brake = 0
                    Rover.steer = -15
                if len(Rover.nav_angles) >= Rover.go_forward:
                    Rover.throttle = Rover.throttle_set
                    Rover.brake = 0
                    Rover.steer = np.clip(np.mean(Rover.nav_angles *
180/np.pi), -15, 15)
                    Rover.mode = 'forward'
```

First, it checks if the rover still has velocity and if it has, it brakes.

This mode is trying to look for any navigable terrain pixels by steering the rover, and then check again for any navigable terrain pixels, and if it founds, it throttles and goes to forward mode agaiu.

2.0 Edited Code Explanation

2.1 Perception.py updates

```
def color_thresh_rock(img):
    hsv = cv2.cvtColor(img,cv2.COLOR_RGB2HSV)
    lower_yellow = np.array([20,100,100])
    upper_yellow = np.array([40,255,255])
    mask = cv2.inRange(hsv,lower_yellow,upper_yellow)
    result = cv2.bitwise_and(img,img,mask = mask)
    binary_result = color_thresh(result,(0,0,0))

    return binary_result
```

This function is similar to the color_thresh() function. However, it sets a maximum threshold for the red and green channels, but a minimum threshold for the blue channel. This function is specific for detecting rocks as rocks seem yellow which is a mixture of red and green.

```
 threshed_navigable_crop = np.zeros_like(threshed_navigable)
    threshed_obstacle_crop = np.zeros_like(threshed_obstacle)
    x1 = np.int(threshed_navigable.shape[0]/2)
    x2 = np.int(threshed_navigable.shape[0])
    y1 = np.int(threshed_navigable.shape[1]/3)
    y2 = np.int(threshed_navigable.shape[1]*2/3)

    threshed_navigable_crop[x1:x2, y1:y2] = threshed_navigable[x1:x2,
y1:y2]
    threshed_obstacle_crop[x1:x2, y1:y2]= threshed_obstacle[x1:x2, y1:y2]
```

Perspective transform actually is distorted when the vision of the rover's camera is far away from the rover itself, so only near vision will be added to the world map. Hence, this would increase fidelity.

```
if (np.float(np.abs(Rover.roll) % 360) <= 1) and \
        (np.float(np.abs(Rover.pitch) % 360) <= 1):
        Rover.worldmap[ypix_world_obs, xpix_world_obs, 0] += 1
        Rover.worldmap[ypix_world_rock, xpix_world_rock, 1] += 1
        Rover.worldmap[ypix_world_nav, xpix_world_nav, 2] += 1
```

This means that we only update the world map when the roll or pitch are below certain threshold (<= 1) this value has been calculated using trial and error.

```
if (xpix_rock.any() or Rover.mode == 'goto_rock'):

        if (Rover.mode != 'goto_rock'):
            Rover.mode = 'goto_rock'

        if (xpix_rock.any()):
            Rover.nav_dists, Rover.nav_angles =
to_polar_coords(xpix_rock,ypix_rock)
            Rover.see_rock_error = 0

        else:
            Rover.see_rock_error += 1

        if Rover.see_rock_error > 100:
            Rover.mode = 'stop'

    else:
        Rover.nav_dists, Rover.nav_angles =
to_polar_coords(xpix_nav,ypix_nav)
```

This is to set priority to find the rocks and collect them. Hence, when the rover finds a rock it switches to the mode “goto_rock” and goes to collect rock.

If xpix_rock.any() returns false this means that perspective transform mistakenly transformed an image and denoted as a “rock” (by yellow color).

```
 debugging_mode = False
    if debugging_mode:
        plt.figure(1, figsize=(10,12))
        plt.clf()
        plt.subplot(321)
        plt.imshow(Rover.img)
        plt.title('Rover image')
        plt.subplot(322)
        plt.imshow(warped_navigable)
        plt.title('Bird Eye View')
        plt.subplot(323)
        plt.imshow(threshed_navigable, cmap='gray')
        plt.title('Threshed image for terrain detection')
        plt.subplot(324)
        plt.plot(xpix_nav, ypix_nav, '.', color='blue')
        plt.ylim(-160, 160)
        plt.xlim(0, 160)
        arrow_length = 100
        mean_dir = np.mean(Rover.nav_angles)
        x_arrow = arrow_length * np.cos(mean_dir)
        y_arrow = arrow_length * np.sin(mean_dir)
        plt.arrow(0, 0, x_arrow, y_arrow, color='red', zorder=2,
head_width=10, width=2)
        plt.title('Approximate direction where the Rover should move')
        plt.subplot(325)
        plt.imshow(threshed_obstacle, cmap='gray')
        plt.title('Threshed image for obstacles detection')
        plt.subplot(326)
        plt.plot(xpix_obs, ypix_obs, '.', color='green')
        plt.ylim(-160, 160)
        plt.xlim(0, 160)
        plt.title('Rover-centric obstacle pixels')
        plt.pause(1)
```

This is the debugging mode; you have to set the debugging_mode flag to true to enter debugging mode

```
  generate_video = False

    if generate_video:
        plt.figure(2, figsize=(12,9))
        plt.clf()
        plt.subplot(321)
        fig1 = plt.imshow(Rover.img)
        plt.subplot(322)
        fig2 = plt.imshow(warped_navigable)
        plt.subplot(323)
        fig3 = plt.imshow(threshed_navigable, cmap='gray')
        plt.subplot(324)
        plt.plot(xpix_nav, ypix_nav, '.')
        plt.ylim(-160, 160)
        plt.xlim(0, 160)
        arrow_length = 100
        mean_dir = np.mean(Rover.nav_angles)
        x_arrow = arrow_length * np.cos(mean_dir)
        y_arrow = arrow_length * np.sin(mean_dir)
        fig4 = plt.arrow(0, 0, x_arrow, y_arrow, color='red', zorder=2,
head_width=10, width=2)
        fig1.axes.get_xaxis().set_visible(False)
        fig1.axes.get_yaxis().set_visible(False)
        fig2.axes.get_xaxis().set_visible(False)
        fig2.axes.get_yaxis().set_visible(False)
        fig3.axes.get_xaxis().set_visible(False)
        fig3.axes.get_yaxis().set_visible(False)
        fig4.axes.get_xaxis().set_visible(False)
        fig4.axes.get_yaxis().set_visible(False)
        plt.savefig('./output/pipeline_{}.jpg'.format(Rover.counter),
bbox_inches='tight', pad_inches = 0)
        Rover.counter += 1
        plt.pause(1)
```

Same as debugging mode; you must activate the generate_video flag to true to generate a video of the autonomous mode run.

2.2 Decision.py updates:

```
        elif Rover.mode == 'goto_rock':
            if Rover.near_sample:
                Rover.throttle = 0
                Rover.brake = Rover.brake_set
            elif Rover.vel > np.mean(Rover.nav_dists):
                Rover.throttle = 0
                Rover.brake = Rover.brake_set/2
            elif Rover.vel < Rover.max_vel/2:
                Rover.throttle = Rover.throttle_set/2
                Rover.brake = 0
            elif Rover.vel > Rover.max_vel/2:
                Rover.throttle = 0
                Rover.brake = Rover.throttle_set/3
```

We have created a new mode called goto_rock; this mode is specific to when the rover detects a rock from the xpix_rock.any() flag in the perception_step method in Perception.py, so when the rover to this mode, it slows down to center itself in front of the rock and then collects it.

```
            if (np.abs(Rover.vel) <= 0.05):
                Rover.stuck_time += 1
            else:
                Rover.stuck_time = 0
            if Rover.stuck_time > Rover.error_limit:
                Rover.throttle = -Rover.throttle_set
                Rover.steer = -np.clip(np.mean(Rover.nav_angles *
180/np.pi), -15, 15)
                time.sleep(1)
Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)
```

This part of the code handles the case where the rover is stuck, so it is handled by stuck_time counter which keeps incrementing as long as the velocity is less than 0.05, that is, the rover is trying to throttle, but it doesn't move. If stuck_time value exceeds certain threshold; error_limit, it steers the rover and throttles to try accelerating again.

```
if (len(Rover.nav_angles) >= Rover.stop_forward) and \
                 (np.max(Rover.nav_dists > 20)):
```

This condition was updated to check also if the maximum distance between the rover and the most far pixel so that slam into an obstacle.

3.0 Results:

The best results we could achieve so far are as follows:

Time: 1463 s appx

Mapping: 95.4%

Fidelity: 75.5%

Rocks collected: 6

Figure 1 Result of an autonomous mode run