Luffy Huang
Oct 10. 2017
Assignment for Rock Samples

Jupyter Notebook Analysis

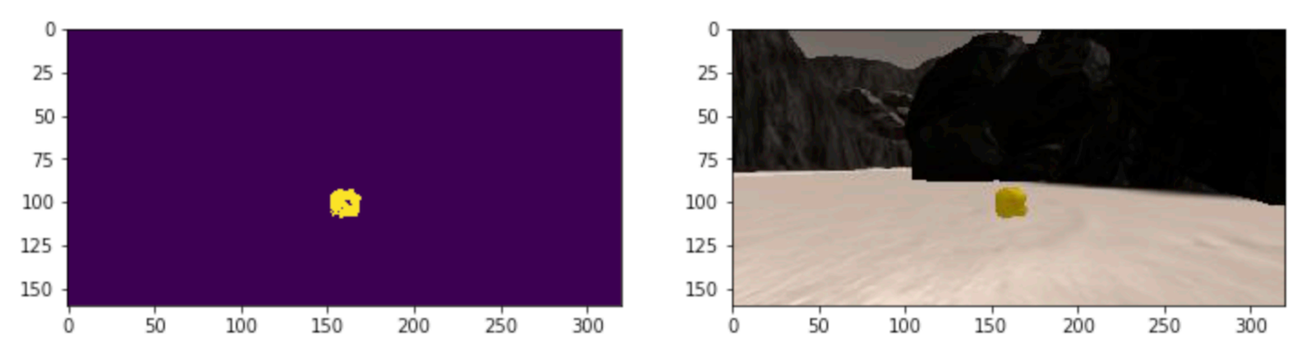
```
def perspect_transform(img, src, dst):

    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
```

In the first part of the assignment, I added a mask variable, which was just a matrix of ones. By transforming the matrix into world-map perspective, I could get a matrix that included both obstacles and navigable areas.

In [7]:
```
def rock_thresh(img, rgb_thresh=(110,110,50)):
    rock_select = np.zeros_like(img[:,:,0])
    rock_thresh = (img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,1] > rgb_thresh[1]) \
                & (img[:,:,2] < rgb_thresh[2])
    rock_select[rock_thresh] = 1
    return rock_select
rock_th = rock_thresh(rock_img)
fig = plt.figure(figsize=(12,3))
plt.subplot(121)
plt.imshow(rock_th)
plt.subplot(122)
plt.imshow(rock_img)
```

Out[7]: <matplotlib.image.AxesImage at 0x11e098710>



Later, I added another function called rock_thresh, which could identify the rock from the image. The yellow color in RGB is with same amount of red and green color and less amount of blue color. So I set color threshold as (110, 110, 50), and pixels that satisfy and threshold were set to ones. As a result, I was able to render the rock in the plot.

```
def process_image(img):
    warped, mask = perspect_transform(img, source, destination)
    threshed = color_thresh(warped)
    obs_map = np.absolute(np.float32(threshed)-1) * mask
    xpix, ypix = rover_coords(threshed)
    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]
    x_world, y_world = pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale)

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

    data.worldmap[y_world, x_world, 2] = 255
    data.worldmap[obs_y_world, obs_x_world, 0] =255
    nav_pix = data.worldmap[:,:,2] > 0
    data.worldmap[nav_pix, 0] = 0

    rock_map = rock_thresh(warped,rgb_thresh=(110,110,50))
    if rock_map.any():
        rock_x, rock_y = rover_coords(rock_map)
        rock_x_world, rock_y_world = pix_to_world(rock_x, rock_y, xpos,
                                                  ypos, yaw, world_size, scale)
        data.worldmap[rock_y_world, rock_x_world, :] = 255
```

In the process_image function, I had navigable areas as threshed variable, and got obstacles map (obs_map) by inverting threshed and getting rid of blind spots of camera through multiplication with mask. Another way to think of the obstacles was that I subtracted navigable areas from mask, which contains both navigable areas and obstacles.
After getting x positions, y positions, yaw angles of the robot, and world map size, I converted rover centric pixel values to world coordinate using pix_to_world function for both navigable areas and obstacles. Then I updated world map in which navigable areas were in blue and obstacles were in red.
Last step, I used rock_thresh to find whether there was a rock in the image, converted it to world coordinate, and updated the map with while spot.

Perception and Decision

```
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
    dst_size = 5
    image = Rover.img
    bottom_offset = 6
    source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
    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] - 2*dst_size - bottom_offset],
                  [image.shape[1]/2 - dst_size, image.shape[0] - 2*dst_size - bottom_offset],
                  ])
```

First, I updated source and destination points for perspective transform, which was the same as we did in previous quiz. The source was the four points of the grid, and the destination was the square area associated with world map.

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

threshed = color_thresh(warped)
obs_map = np.absolute(np.float32(threshed)-1) * mask
Rover.vision_image[:,:,2] = threshed * 255
Rover.vision_image[:,:,0] = obs_map *255

xpix, ypix = rover_coords(threshed)

world_size = Rover.worldmap.shape[0]
scale = 2 * dst_size
xpos = Rover.pos[0]
ypos = Rover.pos[1]
yaw = Rover.yaw
x_world, y_world = pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale)

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

Next, I updated the image (which was shown in the left corner) with obstacles in red and navigable areas in blue in robot's perspective, and read data from the rover, and converted the robot centric pixel values to world coordinate.

```
Rover.worldmap[y_world, x_world, 2] += 10
Rover.worldmap[obs_y_world, obs_x_world, 0] += 1
nav_pix = Rover.worldmap[:,:,2] > 0
Rover.worldmap[nav_pix, 0] = 0

dist, angles = to_polar_coords(xpix, ypix)
Rover.nav_dists = dist
Rover.nav_angles = angles

rock_map = rock_thresh(warped,rgb_thresh=(110,110,50))
if rock_map.any():
    rock_x, rock_y = rover_coords(rock_map)
    rock_x_world, rock_y_world = pix_to_world(rock_x, rock_y, xpos,
                                              ypos, yaw, world_size, scale)
    rock_dist, rock_ang = to_polar_coords(rock_x, rock_y)
    rock_idx = np.argmin(rock_dist)
    rock_xcen = rock_x_world[rock_idx]
    rock_ycen = rock_y_world[rock_idx]

    Rover.worldmap[rock_ycen, rock_xcen, 1] = 255
    Rover.vision_image[:,:,1] = rock_map*255
else:
    Rover.vision_image[:,:,1] = 0
```

In the end of perception step, I modified world map with navigable areas in blue and obstacle as red, updated the distance and angles in polar coordinate, and used if statement to check the existence of rocks. If the rock exists, I would convert the rover perspective to world coordinate, find the position of rock by getting the position of minimum distance, and updated it in green layer (which will overlap with blue layer).

```
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
                    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:
                    # 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.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 forwa
                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
                    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'
```

```python
    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
```

The decision step basically helps the rover to make decisions on its motion, and I would translate the code to plain language.
If the rover is moving forward, we will check if the navigable areas in terms of angle are larger enough to continue rover's motion. If angle is too small, and we will hit brake and make the vehicle stop. Otherwise, the vehicle will continue to move forward in preset throttle in the mean angle.
If the rover starts to stop, meanwhile the velocity is still greater than 0.2 m/s, so we will brake. If the velocity is less than 0.2 m/s and almost fully stops, we will rotate the rover until the angle is bigger enough to continue to move forward.

The result of autonomous driving is shown below.