Project - Search and Sample Return

June 1, 2017

Author: James Goppert

1 Writeup

1.1 Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.?

- The writeup / README should include a statement and supporting figures / images that explain how each rubric item was addressed, and specifically where in the code each step was handled.

YOU ARE READING IT!

2 Notebook Analysis

2.1 Run the functions provided in the notebook on test images (first with the test data provided, next on data you have recorded). Add/modify functions to allow for color selection of obstacles and rock samples.

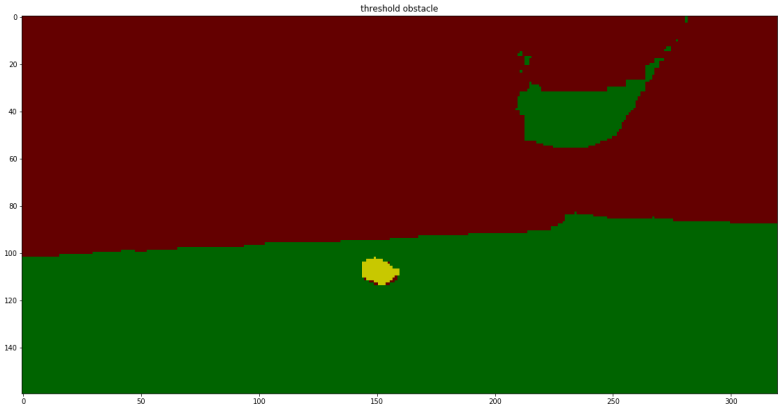
- Describe in your writeup (and identify where in your code) how you modified or added functions to add obstacle and rock sample identification.

2.1.1 Answer

In order to do the color segmentation I used hue saturation and value ranges. The rocks were highly saturated, with low hue, and high value. The ground was higher intensity than the obstacles, but both had low saturation. I also used some open and close operations to remove any dots.

"'python def color_segmenter(img): """ More advanced version of color_thresh that does entire image segmentation

```
@param: img: the input image
@return: dict with images for each type of background
    and a composite for visualization
"""
hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
kernel = np.ones((5,5),np.uint8)
```



image

```
thresh_rock = cv2.inRange(
    hsv,
    np.array([0, 200, 100], dtype=np.float),
    np.array([100, 255, 255], dtype=np.float))
thresh_rock = cv2.morphologyEx(thresh_rock, cv2.MORPH_CLOSE, kernel)
thresh_rock = thresh_rock > 0

thresh_ground = cv2.inRange(hsv,
                            np.array([0, 0, 160], dtype=np.float),
                            np.array([255, 255, 255], dtype=np.float))
thresh_ground[thresh_rock] = 0 # rocks are not ground
thresh_ground = cv2.morphologyEx(thresh_ground, cv2.MORPH_CLOSE, kernel)
thresh_ground = thresh_ground > 0

thresh_obstacle = 1 - thresh_ground

thresh_img = np.zeros_like(img)


ground_x, ground_y = thresh_ground.nonzero()
thresh_img[ground_x, ground_y,:] = [0, 100, 0]

obstacle_x, obstacle_y = thresh_obstacle.nonzero()
thresh_img[obstacle_x, obstacle_y,:] = [100, 0, 0]

rock_x, rock_y = thresh_rock.nonzero()
thresh_img[rock_x, rock_y,:] = [200, 200, 0]

return {
```

```
    'rock': thresh_rock,
    'ground': thresh_ground,
    'obstacle': thresh_obstacle,
    'img': thresh_img
}

   ‘’’
```

2.2 Populate the process_image() function with the appropriate analysis steps to map pixels identifying navigable terrain, obstacles and rock samples into a worldmap. Run process_image() on your test data using the moviepy functions provided to create video output of your result.

- Describe in your writeup how you modified the process_image() to demonstrate your analysis and how you created a worldmap. Include your video output with your submission.

2.2.1 Answer

I compared my calibration image to the test image and didn't find it necessary to modify the source an destination points from what was originally given. I then applied a perspective transform to obtain the warped image. The warped image was sent to the color thresholds to do the segmentation. The pixels corresponding to rock/navigable/and obstacles were then converted into world coordinates. The map at the given world coordinates was then incremented by a fixed amount each time the feature was seen.

Test output video Test Output Video

```
# Define a function to pass stored images to
# reading rover position and yaw angle from csv file
# This function will be used by moviepy to create an output video
def process_image(img):
    # Example of how to use the Databucket() object defined above
    # to print the current x, y and yaw values
    # print(data.xpos[data.count], data.ypos[data.count], data.yaw[data.count])
    xpos = data.xpos[data.count]
    ypos = data.ypos[data.count]
    yaw = data.yaw[data.count]

    # TODO:
    # 1) Define source and destination points for perspective transform


    # Define calibration box in source (actual) and destination (desired) coordinates
    # These source and destination points are defined to warp the image
    # to a grid where each 10x10 pixel square represents 1 square meter
    # The destination box will be 2*dst_size on each side
    dst_size = 5
    # Set a bottom offset to account for the fact that the bottom of the image
```

```
# is not the position of the rover but a bit in front of it
# this is just a rough guess, feel free to change it!
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],
                  ])

# 2) Apply perspective transform
warped = perspect_transform(img, source, destination)

# 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
seg = color_segmenter(warped)

# 4) Convert thresholded image pixel values to rover-centric coords
navigable_pix_x, navigable_pix_y = rover_coords(seg['ground'])
obstacle_pix_x, obstacle_pix_y = rover_coords(seg['obstacle'])
rock_pix_x, rock_pix_y = rover_coords(seg['rock'])


# 5) Convert rover-centric pixel values to world coords
world_size = 200
scale = 5
obstacle_x_world, obstacle_y_world = pix_to_world(
    obstacle_pix_x, obstacle_pix_y,
    xpos, ypos, yaw, world_size, scale)
rock_x_world, rock_y_world = pix_to_world(
    rock_pix_x, rock_pix_y,
    xpos, ypos, yaw, world_size, scale)
navigable_x_world, navigable_y_world = pix_to_world(
    navigable_pix_x, navigable_pix_y,
    xpos, ypos, yaw, world_size, scale)

# 6) Update worldmap (to be displayed on right side of screen)
data.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1
data.worldmap[rock_y_world, rock_x_world, 1] += 255
data.worldmap[navigable_y_world, navigable_x_world, 2] += 1

# 7) Make a mosaic image, below is some example code
    # First create a blank image (can be whatever shape you like)
output_image = np.zeros((img.shape[0] + data.worldmap.shape[0], img.shape[1]*2, 3))
    # Next you can populate regions of the image with various output
    # Here I'm putting the original image in the upper left hand corner
output_image[0:img.shape[0], 0:img.shape[1]] = img

    # Let's create more images to add to the mosaic, first a warped image
```

```
    warped = perspect_transform(img, source, destination)
        # Add the warped image in the upper right hand corner
    output_image[0:img.shape[0], img.shape[1]:] = warped

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

    # output threshhold
    output_image[img.shape[0]:2*img.shape[0], img.shape[1]:2*img.shape[1], :] = seg['img']

        # Then putting some text over the image
    cv2.putText(output_image,"", (20, 20),
                cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
    data.count += 1 # Keep track of the index in the Databucket()


    return output_image
```

3 Autonomous Navigation and Mapping

3.1 Fill in the perception_step() (at the bottom of the perception.py script) and decision_step() (in decision.py) functions in the autonomous mapping scripts and an explanation is provided in the writeup of how and why these functions were modified as they were.

- perception_step() and decision_step() functions have been filled in and their functionality explained in the writeup.

Answer

For the perception step I followed the notebook process_image function closely. I also added calculation of the navigable angles and distances as well as a check if the vehicle was level before publishing the maps due to discrepancy in the perpective transform at large roll and pitch angles.

“‘python # Apply the above functions in succession and update the Rover state accordingly def perception_step(Rover): # Perform perception steps to update Rover() # TODO: # NOTE: camera image is coming to you in Rover.img

```
img = Rover.img
xpos = Rover.pos[0]
ypos = Rover.pos[1]
yaw = Rover.yaw
roll = Rover.roll
pitch = Rover.pitch

# 1) Define source and destination points for perspective transform

# Define calibration box in source (actual) and destination (desired) coordinates
```

```
# These source and destination points are defined to warp the image
# to a grid where each 10x10 pixel square represents 1 square meter
# The destination box will be 2*dst_size on each side
dst_size = 5

# Set a bottom offset to account for the fact that the bottom of the image
# is not the position of the rover but a bit in front of it
# this is just a rough guess, feel free to change it!
bottom_offset = 6

source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
destination = np.float32([[img.shape[1]/2 - dst_size, img.shape[0] - bottom_offset],
                  [img.shape[1]/2 + dst_size, img.shape[0] - bottom_offset],
                  [img.shape[1]/2 + dst_size, img.shape[0] - 2*dst_size - bottom_offset],
                  [img.shape[1]/2 - dst_size, img.shape[0] - 2*dst_size - bottom_offset],
                  ])
# 2) Apply perspective transform
warped = perspect_transform(img, source, destination)

# 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
seg = color_segmenter(warped)

# 4) Update Rover.vision_image (this will be displayed on left side of screen)
    # Example: Rover.vision_image[:,:,0] = obstacle color-thresholded binary image
    #          Rover.vision_image[:,:,1] = rock_sample color-thresholded binary image
    #          Rover.vision_image[:,:,2] = navigable terrain color-thresholded binary image
Rover.vision_image = seg['img']

# 5) Convert map image pixel values to rover-centric coords
navigable_pix_x, navigable_pix_y = rover_coords(seg['ground'])
obstacle_pix_x, obstacle_pix_y = rover_coords(seg['obstacle'])
rock_pix_x, rock_pix_y = rover_coords(seg['rock'])

# 6) Convert rover-centric pixel values to world coordinates
world_size = 200
scale = 10
obstacle_x_world, obstacle_y_world = pix_to_world(
    obstacle_pix_x, obstacle_pix_y,
    xpos, ypos, yaw, world_size, scale)
rock_x_world, rock_y_world = pix_to_world(
    rock_pix_x, rock_pix_y,
    xpos, ypos, yaw, world_size, scale)

if len(rock_x_world) > 0:
    goal_dist, goal_angle = to_polar_coords(np.mean(rock_pix_x), np.mean(rock_pix_y))
    Rover.goal_dist = goal_dist
    Rover.goal_angle = goal_angle
```

```
navigable_x_world, navigable_y_world = pix_to_world(
    navigable_pix_x, navigable_pix_y,
    xpos, ypos, yaw, world_size, scale)

# 7) Update Rover worldmap (to be displayed on right side of screen)

# only update map if we have small roll/pitch so that the
# perspective transform is valid
if roll > 180:
    roll -= 360
if pitch > 180:
    pitch -= 360
if Rover.mode != 'pickup' and np.abs(roll) < 1 and np.abs(pitch) < 1:
    Rover.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1
    Rover.worldmap[rock_y_world, rock_x_world, 1] += 1
    Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 1

# 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(navigable_pix_x, navigable_pix_y)
Rover.nav_dists = dist
Rover.nav_angles = angles
return Rover
```

For the decision function I add the following:

1. I added a function for automatic pickup if the vehicle is near a sample. I also added a pickup mode so it will wait to finish picking up before trying to continue.
2. I implemented a weighted mean for the steering law. The was similar to the example but I wegiht the left steering direction more heavily so it hugs walls more. I also weighted navigable terrain at a greater distance.

```
# This is where you can build a decision tree for determining throttle, brake and steer
# commands based on the output of the perception_step() function
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!

    # pickup rock sample we are near
    if Rover.near_sample:
        Rover.send_pickup = True
        Rover.mode = 'pickup'
```

```
# Example:
# Check if we have vision data to make decisions with
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
            area = 0
            angle_sum = 0
            total_sum = 0
            scale = 10
            right_weight = 0.0
            left_weight = 0.1
            dist_weight = 0.1

            # perform a weighted average over angles
            for angle, dist in \
                    zip(Rover.nav_angles, Rover.nav_dists):
                weight = 1
                if angle > 0:
                    weight += left_weight
                else:
                    weight += right_weight
                # weight further distant points more
                weight += dist_weight*(dist/scale)
                angle_sum += weight*angle
                total_sum += weight
            Rover.steer = np.clip(angle_sum/total_sum * 180/np.pi, -15, 15)
            # Rover.steer += 1*(Rover.goal_angle - np.deg2rad(Rover.yaw))
            # Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)

        # If there's a lack of navigable terrain pixels then go to 'stop' mode
        elif len(Rover.nav_angles) < Rover.stop_forward:
                # Set mode to "stop" and hit the brakes!
                Rover.throttle = 0
                # Set brake to stored brake value
                Rover.brake = Rover.brake_set
```

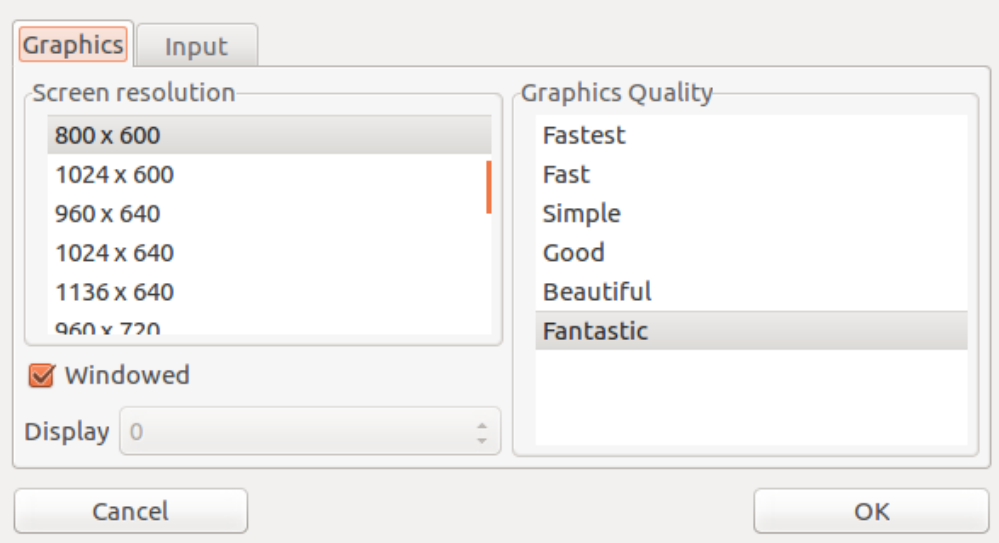
```
                Rover.steer = 0
                Rover.mode = 'stop'

    # If we're already in "stop" mode then make different decisions
    elif Rover.mode == 'stop':
        # If we're in stop mode but still moving keep braking
        if Rover.vel > 0.2:
            Rover.throttle = 0
            Rover.brake = Rover.brake_set
            Rover.steer = 0
        # If we're not moving (vel < 0.2) then do something else
        elif Rover.vel <= 0.2:
            # Now we're stopped and we have vision data to see if there's a path forward
            if len(Rover.nav_angles) < Rover.go_forward:
                Rover.throttle = 0
                # Release the brake to allow turning
                Rover.brake = 0
                # Turn range is +/- 15 degrees, when stopped the next line will induce 4-whe
                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 + 5), -15, 15)
                Rover.mode = 'forward'

    elif Rover.mode == 'pickup':
        # stop and wait until we are finished picking up
        Rover.throttle = 0
        # Set brake to stored brake value
        Rover.brake = Rover.brake_set
        Rover.steer = 0
        # when we are done picking it up, go back to forward mode
        if not Rover.picking_up:
            Rover.mode = 'forward'

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

return Rover
```



graphics settings

3.2 Launching in autonomous mode your rover can navigate and map autonomously. Explain your results and how you might improve them in your writeup.

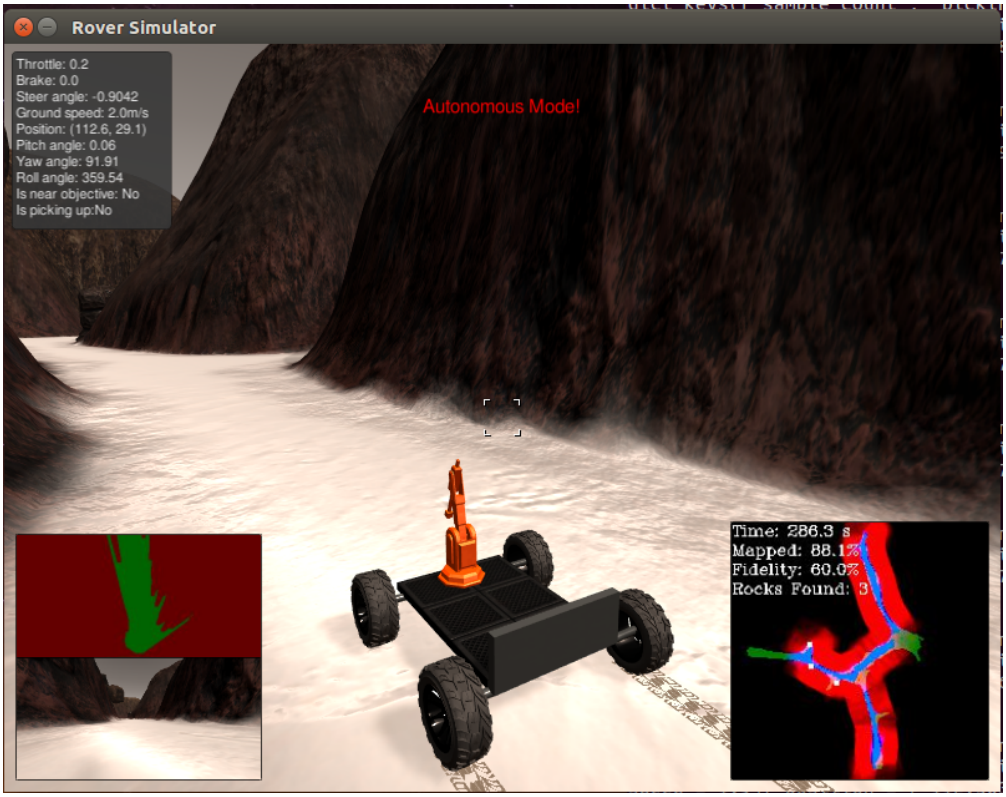
- By running drive_rover.pyand launching the simulator in autonomous mode, your rover does a reasonably good job at mapping the environment.
- The rover must map at least 40% of the environment with 60% fidelity (accuracy) against the ground truth. You must also find (map) the location of at least one rock sample. They don't need to pick any rocks up, just have them appear in the map (should happen automatically if their map pixels in Rover.worldmap[:,:,1] overlap with sample locations.)
- Note: running the simulator with different choices of resolution and graphics quality may produce different results, particularly on different machines! Make a note of your simulator settings (resolution and graphics quality set on launch) and frames per second (FPS output to terminal by drive_rover.py) in your writeup when you submit the project so your reviewer can reproduce your results.

Answer

I was able to get good mapping coverage for most runs, around 80% or better. My fidelity was right around the requirement of 60%.

Things I would do to improve it if I had time.

- Try out VFH obstacle avoidance method instead of mean angle
- Implement a get unstuck mode when it doesn't see a rock in the camera but can't move forward.
- I wanted to add some sort of boundary closure logic where a goal would be initiated on the dge and progressed forward along the edge until the boundary was complete.
- I wanted to have a goal mode, where it wold start tracking the rocks until it reached them.



sim_performance

4 Submission

- Jupyter Notebook with your test code Jupyter Notebook
- Test output video Test Output Video
- Autonomous navigation scripts
- drive_rover.py
- supporting_functions.py
- decision.py
- perception.py
- writeup report (md or pdf file)
- pdf

```
In [1]: !jupyter nbconvert Project\ -\ Search\ and\ Sample\ Return.ipynb --to PDF --output roboN

[NbConvertApp] Converting notebook Project - Search and Sample Return.ipynb to PDF
[NbConvertApp] Writing 21083 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 61001 bytes to roboND_jgoppert_p1.pdf
```