Udacity Robotics Nanodegree
Project 1
Search and Sample Return

Frank Zerofsky
October 13, 2017

Objective:

The objective of the search and sample return project is to receive camera input from a simulated rover and instruct the rover to autonomously navigate its environment. The project will require the use of computer vision techniques for analyzing the image files, and using the analyzed image files to generate actions for the rover.

Process:

Rover perception functions were first developed using Jupyter Notebooks on example data sets. Functions for interpreting pixel color values and coordinate locations were developed using the data. The functions were then tested using locally recorded simulation data within the Roversim environment. The outputs for the simulation data were saved in a video file labeled “my_test_mapping.mp4”. The functions were then implemented into the python project files.

The autonomous mapping python program was then modified in the rover decision.py file and the perception.py file. The decision.py logic is the decision making process for the rover, and the perception.py file is used to analyze the images from the rover’s camera.

Jupyter Notebook Analysis:

Process_image() function incorporates the image processing functions. The function takes one input called “img” which is the view from the rover. A perspective transform in performed on the image to get the resulting plot of top down view on the viewable area of the rover. This is done by scaling known points in the image to known points in the transformed condition.

The color_thresh() function is called on the output from the perspective transform to set a threshold value of RGB pixels in the image for navigable terrain. This threshold converts pixels above the threshold to value of 1 or “true” and the pixels below the threshold to 0 or “false”. The result is binary array representing the transformed image space in black and white where the white represents navigable terrain.

The output from the color threshold function is the navigable terrain in the field of view of the rover. The same image can be used to determine where obstacles are in the environment. Since white navigable pixels are a value of 1 and anything else would be an obstacle. Therefore, by subtracting a value of 1 from each element of the image array results in obstacles having a value of -1. Taking the absolute value of the array results in positive values for obstacles. The only problem is image pixels outside the frame of view of the camera are also included. To remove those pixels, an array of ones is used with the computer vision module function “warp perspective” to get values of 1 in the field of view, and values of 0 outside the camera view. The resulting image is multiplied by the manipulated navigable pixel image to result with the image below. The code for this can be viewed in lines 118, 168, and 172 of the perception.py file.

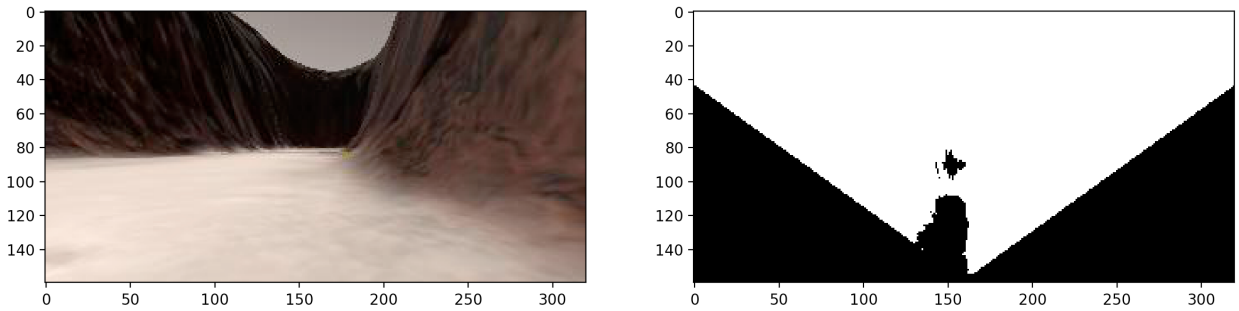

Figure 1: obstacle pixels map

With the binary image of navigable terrain, the following functions can be applied to the pixels to manipulate them for the rover to interpret and plotting on the world map. The rover_coords function converts the navigable terrain from image coordinates to rover coordinates.

```
# get the navigable terrain from the warped image
nav_terrain = color_thresh(warped)
# flip the ones to zeros and the zeros to ones
obs_terrain = np.absolute(np.float32(nav_terrain)-1)*mask

# convert the navigable terrain to rover coordinates using defined function
xpix, ypix = rover_coords(nav_terrain)
# convert the obstacle terrain to rover coordinates
obsxpix, obsypix = rover_coords(obs_terrain)
```

Figure 2: convert to rover coordinates

Data for the current state of the rover is collected from the rover class data. The data class collects the current state of the rover from the .csv file created when recording a simulation.

```
# get the rover data to analyze
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]
```

Figure 3: set variables for current rover state

The output form the rover coordinates function is then passed through the pix_to_world function. This function converts the rover coordinate pixel values to world coordinate values. This is done for both the navigable terrain pixels and the obstruction pixels.

```
# convert navigable terrain to pixels to world coordinates
x_world, y_world = pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale)

# convert obstacle pixels to world coordinates
obs_x_world, obs_y_world = pix_to_world(obsxpix, obsypix, xpos, ypos, yaw, world_size, scale)
```

The current state of the rover is stored in the data class. The world map is updated with the position of the navigable terrain in the blue channel as blue pixels, and the obstruction pixels are stored in the red channel as red pixels. If there is data in the blue channel of the world map

data class, then set the red channel pixels to zero. This results in navigable terrain being recorded and obstructions not.

```
# I need to update the world map with navigable terrain, obstacles, and rocks

# worldmap[] takes inputs of x map points, y map points, and the color channel
# worldmap is previously initialized in the class dataBucket() as an array of zeros (200,200,3)
data.worldmap[y_world, x_world, 2] = 255
data.worldmap[obs_y_world, obs_x_world, 0] = 255

# if blue channel > 0 then set the red channel to 0
# you could try adding a threshold here for the "gray areas"
# don't map the point if within a certain range?
# further investigate
nav_pix = data.worldmap[:,:,2] > 0
data.worldmap[nav_pix,0] = 0
```

The final function added to the process image function is the find_rocks() function. The find rocks functions sets a threshold value for yellow in the rover image similar to the color threshold function. Image pixels within each color channel that meet the threshold are set to a value of 1. The pixels that are outside the threshold range are stay at values of 0. The result below is the image output of a rock in rover image. The code for this function can be found in In[8] of the Jupyter notebook.

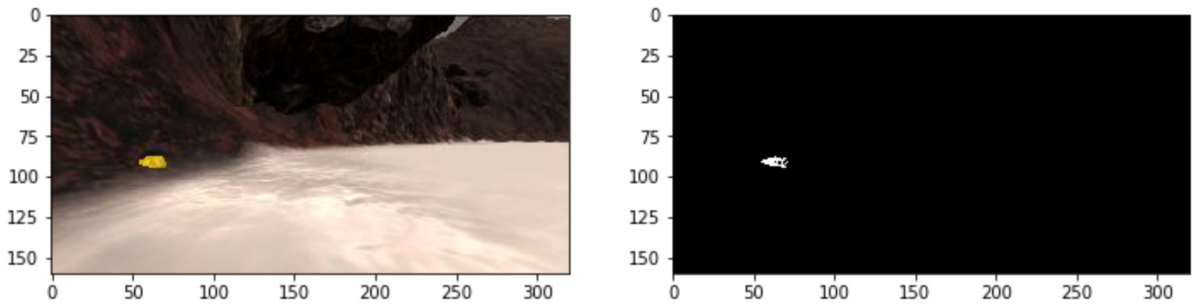

Figure 4: rock identified in the rover image

The process image function calls the find rocks function, if there are any rocks, it records them on the world map by performing the a rover coordinate transform and the world coordinate transform of the positive rock pixel values.

```
# find rocks in the image
rocks = find_rocks(warped, levels=(110, 110, 50))
if rocks.any():
    # there are any values within the rocks array convert the the rock pixels to rover coords
    # then conver the rock pixels from rover coords to world coords
    # update the world map with rock location in R,G,and B channels with 255 (255, 255, 255) = white
    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
```

Autonomous Navigation and Mapping:

The functions developed in the process image function of the Jupyter notebook are adapted to the perception step function in the autonomous navigation code. Modifications were made to enable the rover to follow the environment walls. The following functions were added to the standard perception_step() function.

- color_thresh_snip()
- stbd_frame()
- port_frame()

The first function color_thresh_snip() sets pixel values from a defined area of the image to an array of zeros with the same shape. The result is navigable terrain pixels of the specified area in front of the rover. The coordinates of the range are defined by the constants top_range, bottom_range, left_range, and right_range.

When the function is called in the perception step function of the perception.py file, the variable nav_terrain contains the new image which then converts the true pixels to rover coordinates then world coordinates. The final x_map_world and y_map_world pixel values are recorded on the world map. The reason for the function is to narrow the zone for mapping navigable pixels in the world environment. The result is a smaller area closer to the rover allowing greater fidelity of the rover's map when compared to the actual map of the environment. The code for this function can be viewed below in Figure 5.

```
def color_thresh_snip(color_select):
    # this function is used to create a snip of the color_thresh function output.
    # the purpose is to have a smaller mapped area closer to the rover recorded onto the truth map
    select_snip = np.zeros_like(color_select[:, :])

    # initialize constants for the frame of the picture to be clipped
    view = 50  # pixels in the x direction to keep measured from centerline of image
    bottom_offset = 6
    # initialize ranges of the picture size to trim
    top_range = color_select.shape[0] - view - bottom_offset
    bottom_range = color_select.shape[0] - bottom_offset

    left_range = color_select.shape[1]/2 - view
    left_range = int(left_range)

    right_range = color_select.shape[1]/2 + view
    right_range = int(right_range)

    select_snip[top_range:bottom_range + 1, left_range:right_range + 1] = \
        color_select[top_range:bottom_range + 1, left_range:right_range + 1]

    return select_snip
```

Figure 5: color threshold snipping function within perception.py file

The functions starboard frame (stbd_frame()) and port frame (port_frame()) operate in a similar method only creating images of the right half and left half respectively. The result is the

ability to compare the difference in navigable terrain that the rover has to determine the location of a wall in the world environment and navigate with the wall always to one side.

```
def stbd_frame(image):
    # this function splits the threshold image by 2
    # returns the right side of the image

    snip = np.zeros_like(image[:, :])

    length = image.shape[1]
    mid = length/2
    mid = int(mid)

    snip[:, mid:length + 1] = image[:, mid:length + 1]

    return snip
```

Figure 6: starboard frame function

```
def port_frame(image):
    # this function splits the threshold image by 2
    # returns the left side of the image

    snip = np.zeros_like(image[:, :])

    length = image.shape[1]
    mid = length / 2
    mid = int(mid)

    snip[:, 0:mid + 1] = image[:, 0:mid + 1]

    return snip
```

Figure 7: port frame function

The code in lines 179 through 197 of the perception.py file contains the decision path for the rover in determining which side the wall is on. This is done through the comparison of the number of non-zero values within each half image. If the number of pixels in the starboard is less that in the port frame, then the wall will be on the starboard side of the rover. When the location of the wall is determined, the distance and angle for each pixel in the image is calculated.

```
179     # initialize the starboard and port images
180     stbd = stbd_frame(threshed)
181     port = port_frame(threshed)
182
183     # count the number of true pixels in each image
184     stbd_count = cv2.countNonZero(stbd)
185     port_count = cv2.countNonZero(port)
186
187     # compare the pixel counts to determine where location of the wall
188     # calculate the angles of the pixels, used for steering the rover
189     if stbd_count < port_count:
190         # the wall is on the stbd side
191         xpix, ypix = rover_coords(stbd)
192         dist, angles = to_polar_coords(xpix, ypix)
193
194     else:
195         # the wall is on the port side
196         xpix, ypix = rover_coords(port)
197         dist, angles = to_polar_coords(xpix, ypix)
```

Figure 8: location of the wall using port and starboard image frames

Using the new steering angle values, modifications to the decision.py file were made for the steering logic of the rover. The navigable pixel angles are taken from the perception step function, and a mean angle is calculated. When the mean angle is greater than 35 degrees,the rover will maintain heading. When the mean angle of the navigable pixels in the image are less than 35 degrees, the rover steers to one third the mean angle of the navigable pixels. This allows the rover to correct its steering angle to align with the wall without oversteering into the wall.

Simulation Setup:

The files for the simulation are located on the following github repository.

https//:www.github.com/

1) Clone the github repository
2) Install conda environment located in file “environment.yml” and activate using command “source activate RoboND”
3) In the terminal, navigate to the folder “code”
4) In the terminal, issue the following command.
   python drive_rover.py
5) Open the “Roversim” application.
6) Launch the application with the following settings
   Resolution: 720 x 576
   Graphics Quality: “Good”
   Windowed: checked
7) Press “Play!” and select “Autonomous Mode” when prompted.
8) The autonomous rover simulation will begin once loaded.

Files:

GitHub repository: https://github.com/fzero6/search-sample-return/tree/master

Python files:

drive_rover.py
perception.py
decision.py
supporting_functions.py
Rover_Project_Test_Notebook.ipynb

Other files:

my_test_data.mp4
FRZ_Rover_Report.pdf