Search and Sample Return

Rafael Alfinito Vieira
29/Aug/2017

The below write up will describe how the project`s rubic points were addressed.

A) Notebook Analysis

```
CRITERIA
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.
```

The following material changes were done to relevant parts of the code provided in the scaffolding:

1) in the Databucket() class, I've added the 'roll' and 'pitch' attributes which will later be used to determine if these two variables lay within certain thresholds. If outside of the thresholds, data will not be used in mapping, as the perspective transform function will generate a distorted ground mapping.

```
class Databucket():
    def __init__(self):
        self.images = csv_img_list
        self.xpos = df["X_Position"].values
        self.ypos = df["Y_Position"].values
        self.yaw = df["Yaw"].values
        self.count = 0 # This will be a running index
        self.worldmap = np.zeros((200, 200, 3)).astype(np.float)
        self.ground_truth = ground_truth_3d # Ground truth worldmap
        self.roll = df["Roll"].values #added to be used later
        self.pitch = df["Pitch"].values #added to be used later
```

2) The following four thresholds were applied to the warped image obtained though the perspective transform of the rover's camera images:

    a) Pitch black: threshold for all pure black pixels (RGB[0,0,0]), as these are the pixels that lay in the two opposing triangles outside of the rover's field of view.

```
#------------------------------------------------------------------------
#IF PIXELS ARE PITCH BLACK, THEY ARE PROBABLY OUTSIDE ROVER'S FIELD OF VIEW
rgb_thresh_black=(1, 1, 1)
threshed_black = 1 - color_thresh(warped, rgb_thresh_black)
#------------------------------------------------------------------------
```

b) Obstacles: any terrain below RGB[160,160,160] which is not pitch black.

c) Navigable terrain: all terrain with RGB values above RGB[160,160,160].

d) Sample rocks: a new thresholding function was created to detect rock samples.

This function first converted the image from RGB to HSV, then applied HSV thresholding to detect yellow pixels. This method (HSV thresholding) was chosen, as it seemed more efficient then the RGB thresholding performed beforehand.

Each one of the above four threshold utilizations generated one thresholded image (np array of 0s and 1s) in which the characteristic being detected assumed value True, while all other pixels assumed value False.

The thresholded warped navigation and obstacle images were then clipped by 3/4ths in order to reduce distortion and increase mapping fidelity.

```
#MAKING MASK TO CLIP NAV AND OBSTACLE IMAGES IN ORDER TO REDICE DISTORTION
rowmask = (range(0,int(3*img.shape[0]/4)))
mask = np.zeros_like(img[:,:,0])
mask[rowmask,:] = 1
mask = 1 - mask
```

```
#OBTAINING NAV THRESHOLD
threshed_nav_far = color_thresh(warped, rgb_thresh)
#CLIPPING NAV THRESHOLD WITH MASK
threshed_nav_near = threshed_nav_far * mask
#OBTAINING OBSTACLE THRESHOLD
threshed_obstacles_far = 1 - color_thresh(warped, rgb_thresh) - threshed_black
#CLIPPING OBSTACLE THRESHOLD WITH MASK
threshed_obstacles_near = threshed_obstacles_far * mask
```

Video of test run and mapping is integral part of this project submission.

```
CRITERIA
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 moviepyfunctions provided to create
video output of your result.
```

The below steps outline the process used to transform the rover’ s camera image into the mapped worldmap output, showing the navigable terrain the obstacles, and the sample rocks to be collected.

 # 1) Define source and destination points for perspective transform:
Using the robot’s front camera view, I mapped the four vertices of a 1 square meter grid in front of the robot (source coordinates).
I then determined the coordinates of the vertices of this 1 square meter grid on a birds eye view of the terrain (destination coordinates).

 # 2) Apply perspective transform

With the source and destination coordinates determined above, I applied the 'perspect_transform()' transformation function to the image of the rover's front camera. This transformation warped the front camera view into a bird's eye view perspective (from above).

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

The following four thresholds were applied to the warped image obtained in (2):

a) Pitch black: threshold for all pure black pixels (RGB[0,0,0]), as these are the pixels that lay in the two opposing triangles outside of the rover's field of view.

b) Obstacles: any terrain below RGB[160,160,160] which is not pitch black.

c) Navigable terrain: all terrain with RGB values above RGB[160,160,160].

d) Sample rocks: a new thresholding function was created to detect rock samples.

This function first converted the image from RGB to HSV, then applied HSV thresholding to detect yellow pixels. This method (HSV thresholding) was chosen, as it seemed more efficient then the RGB thresholding performed beforehand.

Each one of the above four threshold utilizations generated one thresholded image (np array of 0s and 1s) in which the characteristic being detected assumed value True, while all other pixels assumed value False.

The thresholded warped navigation and obstacle images were clipped by 3/4ths in order to reduce distortion and increase mapping fidelity.

 # 4) Convert thresholded image pixel values to rover-centric coords

Next, I applied the function 'rover_coords()' to each one of the four thresholded images generated in item (3), above. This function rotated each of the thresholded images 90 degrees clockwise, and adjusted (x, y) coordinates so as to have the rover position being at the origin (0,0) of the new coordinate system.

 # 5) Convert rover-centric pixel values to world coords

Now, I converted the thresholded images from the rover-centric coordinate system to the world coordinate system. This was done through the application of the 'pix_to_world()' function on each of the rover-centric thresholded images. This function uses the rover's (x,y) position in relation to the world coordinate system, as well as the rover's yaw to transform the rover-centric coordinates to the world coordinate system.

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

As the transformation of the rover's front camera view to the bird's eye view is only applicable when the rover's pitch and roll are close to zero, the world map is only updated when both parameters lay inside of acceptable ranges (empirically, +/-2 degrees, in this case).

If rover pitch and roll lay inside the acceptable ranges, all world map pixels which have coordinates set to true in the obstacle, navigation, and sample rock thresholded world-centric coordinates will be updated. This update consists in adding one to the pixel values in each of the world view RGB channels. The red channel is used for obstacles, the blue channel for navigable terrain, and the green channel for rock samples.

As an example, all red channel pixels which have coordinates set to true in the thresholded obstacle image will be increased by one. The same will apply for the blue channel and the navigation thresholded image, and the green channel and the sample rock thresholded image.

B) Autonomous Navigation and Mapping

```
CRITERIA
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.
```

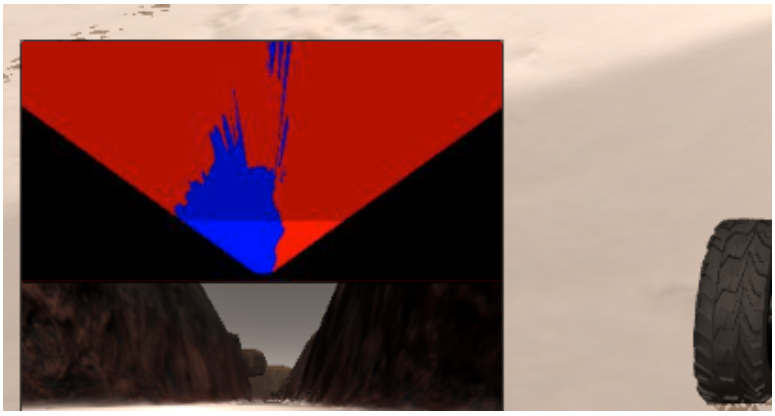
The two main functions that needed to be altered in order to have a functional self-driving rover were the ‘perception_step()’ and the ‘decision_step()’ functions.

1) **perception_step():** in this function, we apply the same 6 steps employed in the Jupyter Notebook (above) to update the rover`s following attributes:

   a) Rover.worldmap: the wordmap, updated with obstacles, rocks, and navigable terrain, will be used by the supporting functions in order for them to compute the percentage of terrain that has been mapped, the mapping fidelity, and the rocks found. Furthermore, this attribute is used to plot the mapped terrain onto the simulator`s lower right-hand corner.

   b) Rover.nav_dists and Rover.nav_angles: these attributes will be used by the `decision_step()` function in order for it to compute the necessary actions to be taken by the rover, such as steering angle, throttle, and breaking.

   In addition to the six steps used earlier in the Notebook, another step was included in order to plot, to the lower left-hand of the simulator screen, the thresholded image of the rover`s warped camera image: Rover.vision_image.

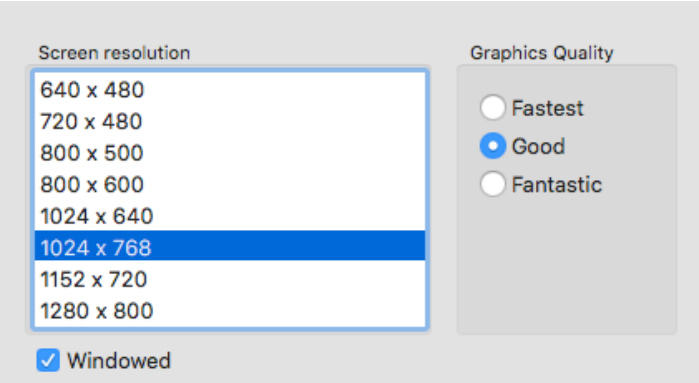


2) **decision_step():** this function uses the telemetry which is stored in the Rover object`s attributes in order to compute, through the use of a decision tree, the next actions to be taken by the rover.

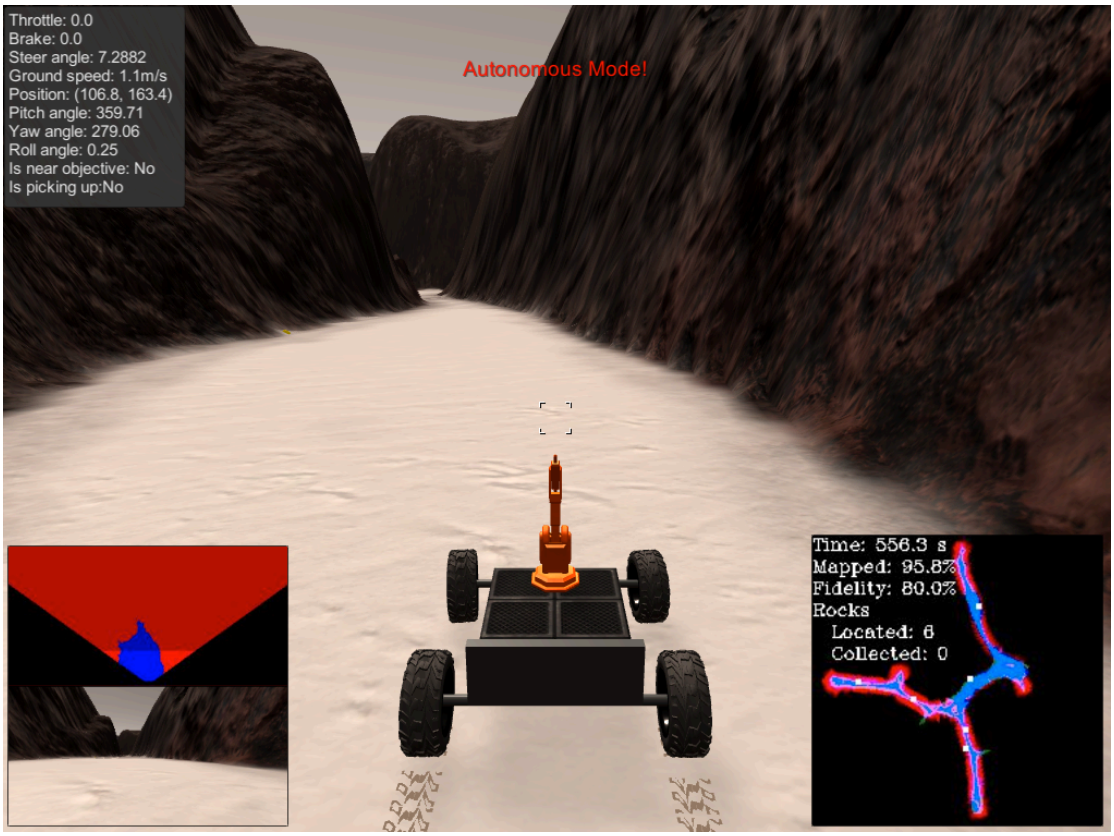
   These actions include:
   - Move forward and steer to vector determined by average angles between rover and navigable pixels.
   - Stop and turn if length of navigable terrain in front of rover is below a set threshold
   - Other scenarios encountered by rover, such as when it gets stuck behind navigable rock, and picking up rock samples (to be implemented)

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

Simulation was run at 1024 x 768 resolution, with `Good` graphics quality. FPS output ranged from 10 to 20.



In autonomous mode, rover was able to map approximately 96% of the terrain, with 80% fidelity. All of the rock samples were found. Results were robust: autonomous tests were run many times, and all yielded similar results with no apparent issues.



Although the rover was highly successful in mapping the terrain and finding the samples, none of them were retrieved. In a later iteration of this project, I hope to have more time to work on a successful sample-retrieving algorithm.