Project: Search and Sample Return

The goals / steps of this project are the following:

Training / Calibration

- Download the simulator and take data in "Training Mode"
- Test out the functions in the Jupyter Notebook provided
- Add functions to detect obstacles and samples of interest (golden rocks)
- Fill in the `process_image()` function with the appropriate image processing steps (perspective transform, color threshold etc.) to get from raw images to a map. The `output_image` you create in this step should demonstrate that your mapping pipeline works.
- Use `moviepy` to process the images in your saved dataset with the `process_image()` function. Include the video you produce as part of your submission.

Autonomous Navigation / Mapping

- Fill in the `perception_step()` function within the `perception.py` script with the appropriate image processing functions to create a map and update `Rover()` data (similar to what you did with `process_image()` in the notebook).
- Fill in the `decision_step()` function within the `decision.py` script with conditional statements that take into consideration the outputs of the `perception_step()` in deciding how to issue throttle, brake and steering commands.
- Iterate on your perception and decision function until your rover does a reasonable (need to define metric) job of navigating and mapping.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

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.

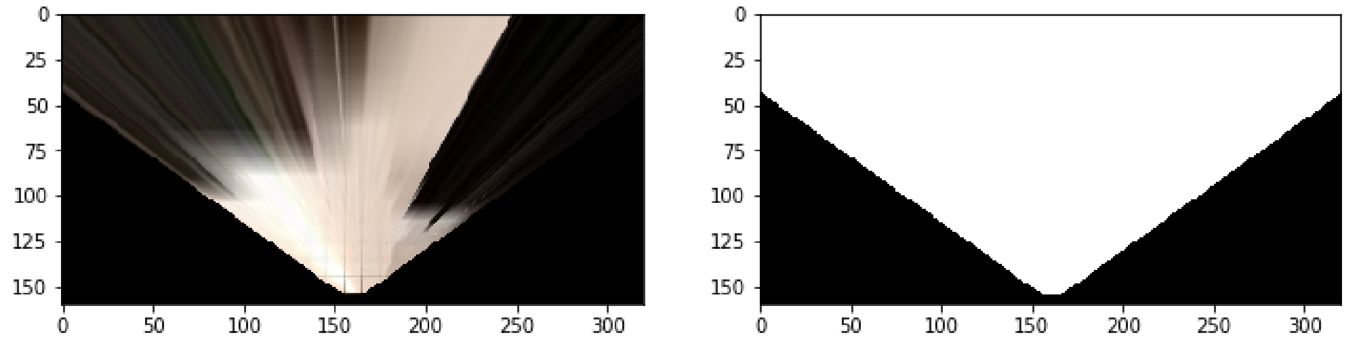
You're reading it!

Notebook Analysis

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.

- Anything not in navigable terrain is considered as obstacle. So, I maintain a mask together with navigable terrain

```
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))
    mask = cv2.warpPerspective(np.ones_like(img[:,:,0]), M, (img.shape[1],
img.shape[0]))
```

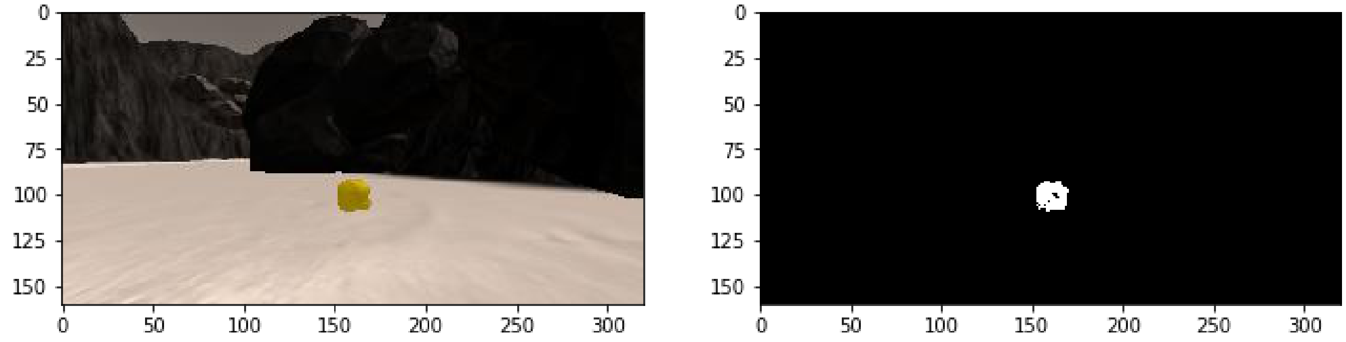


- Rock is identified through color threshold =[110, 110, 50]

```
def find_rocks(img, levels=(110, 110, 50)):
    rockpix = ((img[:,:,0] > levels[0]) \
                & (img[:,:,1] > levels[1]) \
                & (img[:,:,2] < levels[2]))

    color_select = np.zeros_like(img[:,:,0])
    color_select[rockpix] = 1

    return color_select
```



- Identify navigable terrain/obstacles/rocks

```
    threshed = color_thresh(warped)
    rock_map = find_rocks(warped, levels=[110, 110, 50])
    obs_map = np.absolute(np.float32(threshed) - 1) * mask
```

1. 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.

```
def process_image(img):
    # 1) Define source and destination points for perspective transform
    source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
    dst_size = 5
    bottom_offset = 6
    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, mask = perspect_transform(img, source, destination)

    # 3) Apply color threshold to identify navigable
terrain/obstacles/rock samples
    threshed = color_thresh(warped)
    rock_map = find_rocks(warped, levels=[110, 110, 50])
    obs_map = np.absolute(np.float32(threshed) - 1) * mask

    # 4) Convert thresholded image pixel values to rover-centric coords
    xpix, ypix = rover_coords(threshed)
    obsxpix, obsypix = rover_coords(obs_map)
    rockxpix, rockypix = rover_coords(rock_map)

    # 5) Convert rover-centric pixel values to world coords
    scale = 10
    x_pix_world, y_pix_world = pix_to_world(xpix, ypix,
data.xpos[data.count], data.ypos[data.count], data.yaw[data.count],
data.worldmap.shape[0], scale)
    obs_x_world, obs_y_world = pix_to_world(obsxpix, obsypix,
data.xpos[data.count], data.ypos[data.count], data.yaw[data.count],
data.worldmap.shape[0], scale)
    rock_x_world, rock_y_world = pix_to_world(rockxpix, rockypix,
data.xpos[data.count], data.ypos[data.count], data.yaw[data.count],
data.worldmap.shape[0], scale)

    # 6) Update worldmap (to be displayed on right side of screen)
    data.worldmap[obs_y_world, obs_x_world, 0] += 1
    data.worldmap[rock_y_world, rock_x_world, 1] += 1
    data.worldmap[y_pix_world, x_pix_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, mask = 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)

        # Then putting some text over the image
    cv2.putText(output_image,"Populate this image with your analyses to
make a video!", (20, 20),
                cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
    if data.count < len(data.images) - 1:
        data.count += 1 # Keep track of the index in the Databucket()

    return output_image
```

Autonomous Navigation and Mapping

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.

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

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

    # 3) Apply color threshold to identify navigable
```

```
terrain/obstacles/rock samples
    threshed = color_thresh(warped)
    rock_map = find_rocks(warped, levels=[110, 110, 50])
    obs_map = np.absolute(np.float32(threshed) - 1) * mask

    # 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[:,:,2] = threshed * 255
    Rover.vision_image[:,:,1] = rock_map * 255
    Rover.vision_image[:,:,0] = obs_map * 255

    # 5) Convert map image pixel values to rover-centric coords
    xpix, ypix = rover_coords(threshed)
    obsxpix, obsypix = rover_coords(obs_map)
    rockxpix, rockypix = rover_coords(rock_map)

    # 6) Convert rover-centric pixel values to world coordinates
    scale = 10
    x_pix_world, y_pix_world = pix_to_world(xpix, ypix, Rover.pos[0],
Rover.pos[1], Rover.yaw, Rover.worldmap.shape[0], scale)
    obs_x_world, obs_y_world = pix_to_world(obsxpix, obsypix,
Rover.pos[0], Rover.pos[1], Rover.yaw, Rover.worldmap.shape[0], scale)
    rock_x_world, rock_y_world = pix_to_world(rockxpix, rockypix,
Rover.pos[0], Rover.pos[1], Rover.yaw, Rover.worldmap.shape[0], scale)

    # 7) Update Rover worldmap (to be displayed on right side of screen)
        # Example: 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
    Rover.worldmap[y_pix_world, x_pix_world, 2] += 1
    Rover.worldmap[obs_y_world, obs_x_world, 0] += 1
    Rover.worldmap[rock_y_world, rock_x_world, 1] += 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
    Rover.nav_dists, Rover.nav_angles = to_polar_coords(xpix, ypix)

    return Rover
```

```
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!

    # 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
                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-wheel turning
                    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'
    # 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

    # 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
```

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

With the basic of decision_step(), rover can navigate through the map and identify the rocks.

I apply the color_threshold to identify the 'yellow' color which is known as the rock.

The improvement is to do rock pick-up and returning.