Search and Sample Return

1 Notebook Analysis

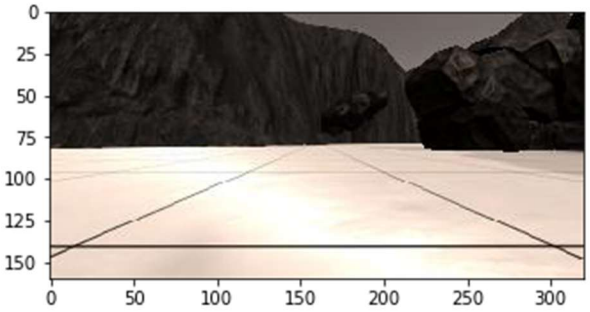
1.1 Obstacle and Sample Identification

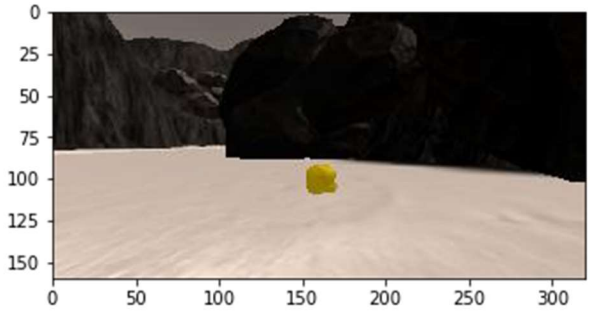
The goal here was to extract the locations of both obstacles and samples from the Rovers camera after completing a perspective transform. For purposes of clarity, I renamed the "color_thresh" function to be drive_thresh as I did not feel the original name was sufficient to differentiate its purpose from the other filters. (Although in retrospect, navigate_thresh may be a better choice). In addition to this trivial change, I added two new functions: obstacle_thresh, and sample_thresh.

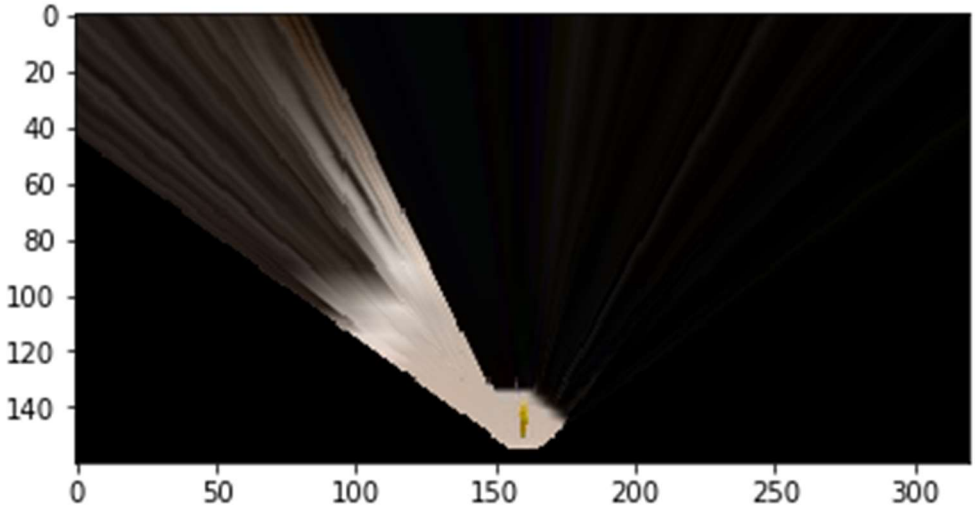
- obstacle_thresh: I assumed (based on clues in the instructions) that obstacles could be extracted from the negative space of the drivable area. (i.e. if you cannot drive on it, it's an obstacle). With that assumption, I simple copied the original "color_thresh" function and inverted all the ">"'s to "<"'s.
- sample_thresh: Here I am exploiting the fact that the samples all appear to be a roughly unique shade of yellow. (I would need a different approach if these were to have random colors, or if the light levels where to be changed during the simulation, or if the material where changed to something shinier / duller). I looked up the RGB values for various shades of yellow and worked by trial and error until I found an upper and lower bound that appeared to work well. My chosen bounds were: (102,102,0) for lower bounds, and (255,255,50) for the upper bounds. Colors where referenced from: http://www.rapidtables.com/web/color/Yellow_Color.htm

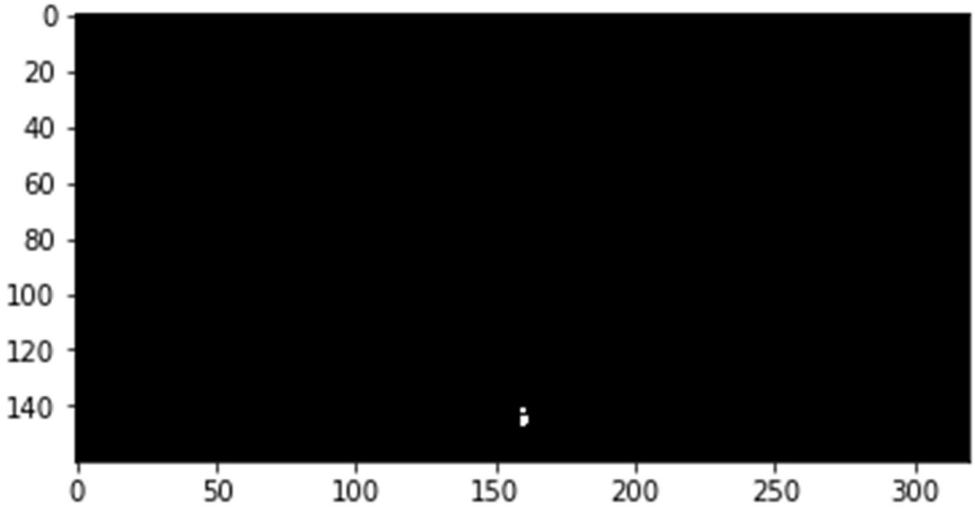
The figures below illustrate the results of the various thresholding functions, starting from the top, these are:

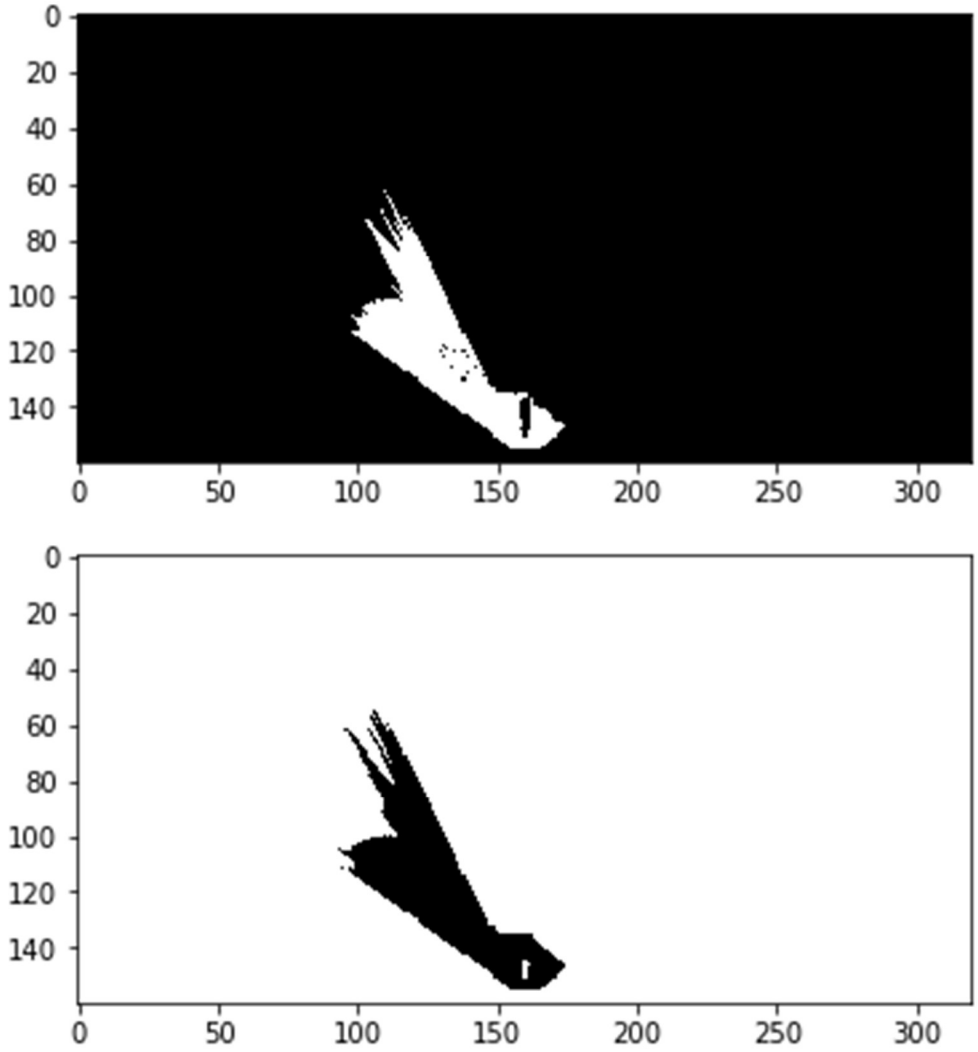
- Original Camera images (I used the right image for this example)
- The input image after a perspective transform
- Transformed image after applying sample_thresh
- Transformed image after applying drive_thresh
- Transformed image after applying obstacle_thresh


0
25
50
75
100
125
150
0
50
100
150
200
250
300


0
25
50
75
100
125
150
0
50
100
150
200
250
300


0
20
40
60
80
100
120
140
0
50
100
150
200
250
300


0
20
40
60
80
100
120
140
0
50
100
150
200
250
300



1.2 PROCESS IMAGE

In this step the relevant features are extracted from the input image fed into the Rover from the camera. These changes were inserted into the process_image function is the attached jupyter notebook. I've listed the additions made to the method below, for the most part they follow the steps illustrated in the pre-existing comments in the notebook.

- I declared a few helper constants and variables. Notably the rovers positions, and some friendly names for the indices corresponding to the various layers in the data.worldmap object.
- Source and Destination points were declared for the OpenCV perspective transform. I kept the original points provided in the examples here as they did not significantly deviate from those points that I had found on my own. Furthermore, I did not see any advantage in changing the size of the destination rectangle for this application.

- The helper function, perspect_transform was called on the input image. This is basically a wrapper for running the warpPerspective method with the correct transformation matrix in OpenCV.
- Thresholding was applied to the transformed image. Each of the three thresholding methods described in the previous section was applied separately to the transformed image.
- The results of each of the thresholds were transformed to rover centric co-ordinates. I made use of the helper function provided in the notebook – rover_coords to achieve this.
- The now rover centric data from above was further transformed into the world reference frame using the pix_to_world helper method provided. As the destination block of the perspective transform was a 10x10 pixel square, the scaling factor passed to this method was 10 in each case.
- All three layers of the world map – Obstacles, Rocks, and Navigable – where updated according to the results of the above world transformations.
- The output image as generated by created a mosaic of the world map, the transformed input, and the original input image provided to the Rover.

2 AUTONOMOUS NAVIGATION & MAPPING

I ran the simulation at 800x600 resolution with highest quality graphics settings in windowed mode.

Some additional changes were made to the Rover class, namely the addition of variables for tracking if a sample was seen in the input image, for tracking the angles to the samples, and for tracking the obstacle angles. I also changed a few of the constants increase the robots maximum sped and to make it a bit more choosy during the wander mode of whether to go forward or keep turning. I will explain these changes further in the decision section.

2.1 PERCEPTION

Similair to the jupytr notebook in the above section, thresholded the input image in three parts, one for the drivable area, one for the obstacles, and one for locating the samples. Like above these methods are named drive_thresh, obstacle_thresh, and sample_thresh. To assist in the removal of potential errors in the results of these methods I also defined a 3x3 kernel (all ones) which will be used in various OpenCV morphology methods.

- drive_thresh – the method is similar to the approach used in the jupytr notebook. Based on trial and error and I increased the threshold from RGB = (160, 160, 160) to be RGB (175, 175, 175). After applying this threshold to the input, I also applied an erode operation using the OpenCV method: cv2.erode using the 3x3 kernel I previously defined. This was applied for only a single iteration and helps with the fidelity of the map.

- obstacle_thresh – this method is practically identical to the drive_thresh with the most notable change being that instead of filtering out everything below the threshold, I am filtering out everything above it.
- sample_thresh – here I changed the thresholding strategy marginally. After filtering out everything above and below the yellow shades of the RBG spectrum I then applied a dilate operation with 3 iterations to try to make any detected samples larger. This was done to make the samples a little easier for the robot to navigate towards.

I experimented with adding a sharpening filter to the image before thresholding, but I found it not only decreased the map's fidelity, but lead to a significant increase in false positives for the sample_thresh method. After experimenting with the step in several permutations I decided the system worked better without this additional filtering.

Much of the image processing is very similar to that done for the jupytr notebook analysis. For the sake of brevity I will not explain again here those processes which exactly match what was already discussed, instead I will focus on the deviations made for the sake of autonomous mode.

- The order of operations was changed for extracting the sample locations. Instead of performing the perspective transform then running the sample_thresh method, I am calling sample_thresh, then transforming the data. This seemed to help with picking out some of the harder to see samples that the robot would occasional just drive by. (Primarily those that had been "sunk" nearly entirely into the ground)
- As the perspective transform is not valid for non-zero pitch and roll, I am not updating the map whenever these are more than 0.5 degrees from 0. This is done to optimize map fidelity (as was suggested in the hints provided in the challenge description)
- I am checking for non-zero entries in the sample_rocks matrix generated by the sample_thresh / perspective transform operation. If any exist, I am setting the "seeSample" flag on the Rover (added by me) and extracting the angles to the sample for navigational purposes. If there are no non-zero entries in the transformed image, then the "seeSample" flag is set to false.

2.2 Decision Making

The decision making process for this Rover is loosely modeled off the soda can collecting Robot from the 1980's described by one of Rodney Brooks grad students (note: fix this later). The basic idea is that the Robot has several "layers" of behaviors where each layer supersedes the one below. The Rovers behaviors are listed below in order of lowest level behavior to highest level:

2.2.1 Wander

The code here is adapted from the starter code provided by Udacity. I have however made a few changes to the behavior which I will walk through here:

First wander checks the mean angle of the obstacles calculated during the perception step. These are used in conjunction with a bias to try to keep the Rover closer to the walls on the map.

```
mean_obstacle_angle = np.mean(Rover.ob_angles)

if not CHANGE_DIR and not Rover.mode == 'stop':
    if mean_obstacle_angle > 0 and DEVIATION_BIAS < 0:
        DEVIATION_BIAS *= -1

    if mean_obstacle_angle < 0 and DEVIATION_BIAS > 0:
        DEVIATION_BIAS *= -1
```

(Note the change direction flag will be explained below). Next, assuming that it has any nav_angles to work with, the Rover enters one of two states: 'forward' or 'stop'. (These are largely derived from the original example code).

In the forward mode, the Rovers behavior is described by the following algorithm:

Rover Forward Algorithm

Given a rover R

Given CHANGE_DIR flag, and DEVIATION_BIAS

Given a list of drivable angles for navigation D

If len(D) > R.change_dir and NOT CHANGE_DIR

    Set throttle to maximum (0.2)

    Multiply the DEVIATION_BIAS by -1

    Set CHANGE_DIR flag to true

    Call determine_forward_direction function

Else if len(D) > R.stop_foward

    Set throttle to maximum

    Set CHANGE_DIR flag to false

    Call determine_forward_direction

Else

Call stop_rover function

I will elaborate on the determine_forward_direction method later in this section. The idea here is that the original forward behavior of the Robot has been split into three regimes. When the angles are the largest, the Rover is assumed to roaming over open terrain far away from other helpful navigational aids. I found that when this occurred the Rover would sometimes get stuck in little loops and never find its way out. To help it get out of said loops, I have added logic to reverse the navigational bias that the determine_forward_direction method uses until such time as it reaches a suitably constrained state.

If there are sufficient available angles to reliable drive forward, the Rover simply resets the CHANGE_DIR flag to true and drives forward as usual, stopping otherwise.

When in the 'stop' state, the Rover uses the following algorithm:

Rover Stop Algorithm

Given a rover R

Given CHANGE_DIR flag, and DEVIATION_BIAS

Given a list of drivable angles for navigation D

Given a list of absolute values of obstacle angles O

If Rover velocity > 0.2

    Call stop_rover function

Else If Rover Velocity <= 0.2

    If len(D) is insufficient to go forward

        Let min_angle = min(O)

        If min_angle < 0

            Steer to a random positive direction

        Else

            Steer to a random negative direction

    Else

        Set Rover state to 'forward'

        Set throttle to max

        Call determine_forward_direction

This behavior is fairly straightforward, if we are still moving, slow down. If we are stopped and can't see a way forward, turn away from the nearest obstacle angle. If we are stopped and there are sufficient angles ahead of us, then we can go forward again.

Key to the wander behaviors functionality is the determine_forward_direction algorithm, which is described below:

Determine Forward Direction

Given a list of drivable angles D

Given the DEVIATION_BIAS

Let mean_navigable = the mean of D in degrees

If there are sufficient angles and the Rover is wandering

Let steerAngle be a random value between [0,1)

Let std be the standard deviation of D

Let mean_navigable = mean_navigatable + DEVIATION_BIAS * std +/- 4 * steerAngle

Clip mean_navigable between -MAX_STEER_ANGLE and MAX_STEER_ANGLE

Set Rover.steer to be mean_navigable

Here I am adding a directional bias to the mean average direction in order to keep the Rover somewhat in touch with the nearest obstacles. Unfortunately, when doing this in a deterministic manner, I found that sometimes the Rover would get stuck in large "macro" loops that it never seemed to break out of. (For example, it would visit the upper right corner of the map, make its way back to the bottom middle, then go to the upper left). To ensure that the Rover fully explores the map, I have added a small random perturbation to the Rovers bias. This allows the Rover to break out of loops without having to keep detailed records of its environment.

(Although a detailed record is of course available thanks to the Rovers mapping functionality)

2.2.2 Go Home

Go Home is triggered when the Rover has collected all the samples (assumed to be six). There are two stages to the go home system.

1. Wander: If the Rover is not within a pre-specified distance from the home location, it will simply wander around as normal. Given the cyclical nature of its environment it will eventually come close enough to the home location to execute the second stage
2. Go Home: Once the wander algorithm has brought the rover within striking distance of home, the Rover changes it strategy. Now rather than biasing towards the average obstacle angle (as wander does), the bias will be applied towards the home location. The Rover will continue to travel until it is within 5 units of the starting position, at which point will stop and continuously print HOME SWEET HOME to the console as a way of celebration.

2.2.3 Avoid

This is essentially the rovers "unstick"mechanism. Despite my best efforts, the rover is not always able to avoid every obstacle on the map, and occasionally finds itself either stuck on rock, or stuck against a wall, or just generally unable to go forward.

To determine if the Rover is stuck I am checking a few conditions:

1. Is the Robot currently trying to get unstuck (determined by the STUCK COUNTER)
2. Is the Robot currently trying to accelerate?
3. Are the brakes on?
4. Does the robot have non zero velocity?

If condition 1 is true, or conditions 2,3, and 4 hold, the Rover is designated to be "stuck", a condition which triggers the avoid algorithm described below:

Avoid

Given STUCK_COUNTER

Given obstacle angles O

Decrement STUCK_COUNTER

Set throttle to 0

Let m = min(O)

If m > 0:

Steer in a random angle in the positive direction

Else

Steer in a random angle in the negative direction

I've found this method to be successful inside the simulation, however if this were a real robot I am afraid this approach would likely have destroyed several hundred million dollars of research equipment by now.

2.2.4 Collect

When the perception system detects a sample in the camera, it sets the "seeSample" flag on the rover, which triggers this behavior. The collect behavior is single minded, it finds the average direction to the sample based on the data provided by the perception module, reduces its maximum speed, and drives towards the sample.

Stopping the rover and grabbing the sample are left to the next higher-level behavior to manage.

2.2.5 Grab

Once the rover is within range of the sample, the simulation sets the Rover.near_sample flag, which is used to trigger this behavior. The Grab behavior simple stops the rover by applying the brakes (fortunately the rover has no organic passengers who might get whiplash from the sudden stops). Once the robot's velocity has reached 0, the grab behavior sets the Rover.send_pickup flag to true to initiate sample collection.

2.2.6 Wait

While not directly named in the code, the wait behavior is initiated whenever the Rover.picking_up flag is true. This ensures that the rover waits for the current sample to be collected before trying to initiate any new actions.

3 RESULTS

Using the approach described above, the Robot can consistently get around 86% of the map “mapped” at roughly 85-88% fidelity. It can generally find all the rocks, and return them to the home base, if given enough time. (I’ve found it can take about 10-15 minutes to complete the task … it’s not exactly fast)

It is not perfect of course, and does have a few weaknesses.

- The Rover is prone to repeating itself, and will sometimes cover the same territory two or three times before making its way to the next section.
- It occasionally doesn’t see a sample fast enough for its navigation system to react and will end up driving right by it. It generally picks these up on a later pass, but this can be inefficient.
- There are a few places on the map where the rover can only get to by pure luck. Notably there is a rather bothersome place behind a patch of rocks that it tends to have trouble with. There is also a small appendix shaped path on the left of the map that it occasionally just misses.
- It can take a long time to get home, which is an artifact of it simply waiting until it gets near home on its normal travels rather than trying to purposefully navigate there.

The efficiency problems the Rover suffers from could likely be mitigated by implementing a more sophisticated planning algorithm which makes use of the information collected already on the map. Unfortunately, this would not work well with the approach I have taken in designing this robot and as such have chosen not to take this path. (Also, that would have been more work which do to personal time constraints I do not have time currently to refactor this project and still meet the given deadlines).

Attached below are a few typical screenshots of the Rover in action using the methods I have described in this report.


Throttle: 0.2
Brake: 0.0
Steer angle: 4.3608
Ground speed: 2.0m/s
Position: (57.7, 99.3)
Pitch angle: 359.96
Yaw angle: 50.98
Roll angle: 0.91
Is near objective: No
Is picking up:No
Autonomous Mode!
Time: 1128.9 s
Mapped: 85.0%
Fidelity: 90.0%
Rocks
Located: 4
Collected: 4


Rover Simulator
Throttle: 0.2
Brake: 0.0
Steer angle: -1.3825
Ground speed: 2.0m/s
Position: (39.2, 96.8)
Pitch angle: 359.98
Yaw angle: 177.95
Roll angle: 359.99
s near objective: No
s picking up:No
Autonomous Mode!
Time: 760.2 s
Mapped: 76.2%
Fidelity: 92.2%
Rocks
Located: 4
Collected: 5


Rover Simulator
Throttle: 0.2
Brake: 0.0
Steer angle: 5.3837
Ground speed: 2.0m/s
Position: (104.1, 89.0)
Pitch angle: 359.91
Yaw angle: 43.05
Roll angle: 359.72
Is near objective: No
Is picking up:No
Autonomous Mode!
Time: 522.9 s
Mapped: 73.0%
Fidelity: 91.4%
Rocks
Located: 3
Collected: 3