Computer vision project

CSE483

|  |  |
| --- | --- |
| **Ahmed Hossam Ahmed Hassan** | **19p5943** |
| **Ziad Mahmoud Mohamed** | **19P6599** |
| **Mohaned Sabry Ghareeb** | **19p6314** |
| **Ahmed Khaled Abdalla Hamed** | **19p3867** |

NASA Mars Sample & Return Rover

(MARS MSR)

In this document we will discuss how to direct the robot

transmitted from a robot-mounted camera by using visuals and computer vision science . The aim of the project is to conduct autonomous mapping and navigation given an initial map of the environment. Also how we managed to choose whether to send orders for the autonomous mode : throttle, brake, and steering with each new image the rover's camera produces.

# The complete code

The complete code is shown and explained in this link :

<https://github.com/Ahmed712441/RoverSim>

# Python notebook analysis

We will discuss some principles covered in this project, such as camera calibration, perspective transform, colour thresholding, and coordinate transforms: camera to robot, robot to world, and euclidian to polar, were demonstrated using a Python Notebook.

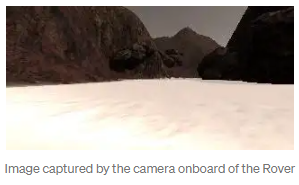
We were asked to build or change features that would enable colour customization of rocks and obstructions. Below, I'll go into more detail :

## Detecting Obstacles / Navigable Terrain

The bright navigable regions and the black barriers provide a pleasing contrast in the environment. So, a straightforward threshold established through drawing this histogram of random images confirming this by trying different value of different threshold . Timeline

Description automatically generated

Filtering the pixels that are higher than a threshold of 160 is all that is required to find the passable terrain (camera values ranges from 0 to 255).



threshed = color\_thresh(warped, rgb\_thresh=(160, 160, 160))  
plt.imshow(threshed, cmap='gray')

# Chart, histogram Description automatically generated

The barriers are just the opposite of travelable ground. However, because We decided to top-down view the image before thresholding, We first needed to ignore the pixels outside of the field of vision (the two dark triangular areas on the bottom left and right). To do that, We made a mask out of a white image that has been warped:

mask = np.ones\_like(threshed)  
mask[:,:] = 255  
mask = perspect\_transform(mask, source, destination)  
plt.imshow(mask, cmap='gray')

Chart

Description automatically generated

Finally, we 've applied the mask to the inverse of the navigable terrain to get the obstacles:

*#identify obstacles*  
obstacles = np.absolute((np.float32(threshed)-1) \* mask)  
plt.imshow(obstacles, cmap='gray')

Chart, histogram

Description automatically generated

## Detecting Rock Samples

As was previously said, the Rover must locate (and, as a bonus, gather) rock samples. The majority of rocks are yellow and look like this:

Graphical user interface, website

Description automatically generated

We will be imposing lower and upper thresholding restrictions to choose the colour yellow. The links below, which illustrates why working with HSV colorspace is more simpler than working with RGB because you just need to set the boundaries for the Hue channel:

<https://dsp.stackexchange.com/questions/27939/why-is-it-easier-to-extract-color-in-hsv-model-than-in-rgb#:~:text=In%20RGB%2C%20all%20the%20three,all%20three%20components%20of%20RGB>.

We discovered that the hue value of the rock's colour is around 24. In order to clearly isolate the rock sample on the image, we transformed the image from RGB (really BGR) to HSV and then took lower and upper boundaries of 10 points below and above that value.

#identify the rock  
lower\_yellow = np.array([24-10,100,100])  
upper\_yellow = np.array([24+10,255,255])  
  
# Convert BGR to HSV  
hsv = cv2.cvtColor(rock\_img, cv2.COLOR\_RGB2HSV)  
# Threshold the HSV image to get only upper\_yellow colors  
mask = cv2.inRange(hsv, lower\_yellow, upper\_yellow)  
  
plt.imshow(mask, cmap='gray') Graphical user interface, application

Description automatically generated

In the part that follows, we 'll go into more detail about how we implemented the Rover Perception Module and then have a similar conversation about the Decision Module.

## Rover Perception Module

The goal here is to map at least 40% of the environment at 60% fidelity and locate at least one of the rock samples.

First, we used the roover\_coords()function to convert the navigable, obstacles, and samples masks from camera coordinates to Rover coordinates, and then we used the pix\_to\_world() method to convert these to world coordinates.

## Strategies Used to Increase Map Fidelity

We 've merged two strategies to get the highest level of map fidelity:

1) to limit camera readings to 8 metres,

2) to only update the map if the Rover Roll and Pitch values are close to zero (plus or minus 1 degree).

Given that camera measurement accuracy declines with increasing distance, we set a distance restriction by using the following technique, to filter out spots farther than 8 metres away from the camera:

**def** impose\_range(xpix, ypix, range=80):  
dist = np.sqrt(xpix\*\*2 + ypix\*\*2)  
**return** xpix[dist < range], ypix[dist < range]

We 've chosen to only utilise the readings where Roll and Pitch values were close to zero because the perspective transform algorithms presuppose that the Rover is parallel to the ground. Anything higher than that would cause noticeable distortion, especially if the Rover ran into an obstacle or broke unexpectedly to grab a rock sample.

Chart, line chart

Description automatically generated

source points 1

Also we cleaned up the obstacle pixels that overlapped with navigable terrain measurements. Here is what the code looks like:

if (Rover.pitch < 1 or Rover.pitch > 359) and (Rover.roll < 1 or Rover.roll > 359):

Rover.worldmap[obstacle\_y\_world, obstacle\_x\_world, 0] = 255

Rover.worldmap[rock\_y\_world, rock\_x\_world,1] = 255

Rover.worldmap[navigable\_y\_world, navigable\_x\_world, 2] = 255

# remove overlap mesurements

nav\_pix = Rover.worldmap[:, :, 2] > 0

Rover.worldmap[nav\_pix, 0] = 0

# clip to avoid overflow

Rover.worldmap = np.clip(Rover.worldmap, 0, 255)

In order to manage the steering of the Rover, the Decision Module will subsequently employ the to\_polar\_coords() method I created by converting the accessible landscape and rocks into:

dist, angles = to\_polar\_coords(xpix\_navigable, ypix\_navigable)

Rover.nav\_dists = dist

Rover.nav\_angles = angles

# Same for rock samples

dist, angles = to\_polar\_coords(xpix\_rocks, ypix\_rocks)

Rover.samples\_dists = dist

Rover.samples\_angles = angles

## Rover Decision Module

The most thrilling aspect of the project was the Rover Decision Module. It involved navigating the terrain with the Rover by using perception information from the Perception Module.

To cover the most ground in the shortest amount of time is the objective. we made the decision to use a wall-following strategy to accomplish that. Hugging the left wall was my decision because it was easier by trial .

The code changes described below where made to the decision\_step() function within the decision.py script.

## Calculating the Steering Offset

Use of the mean angle of the navigable spots to steer the Rover was illustrated Chart

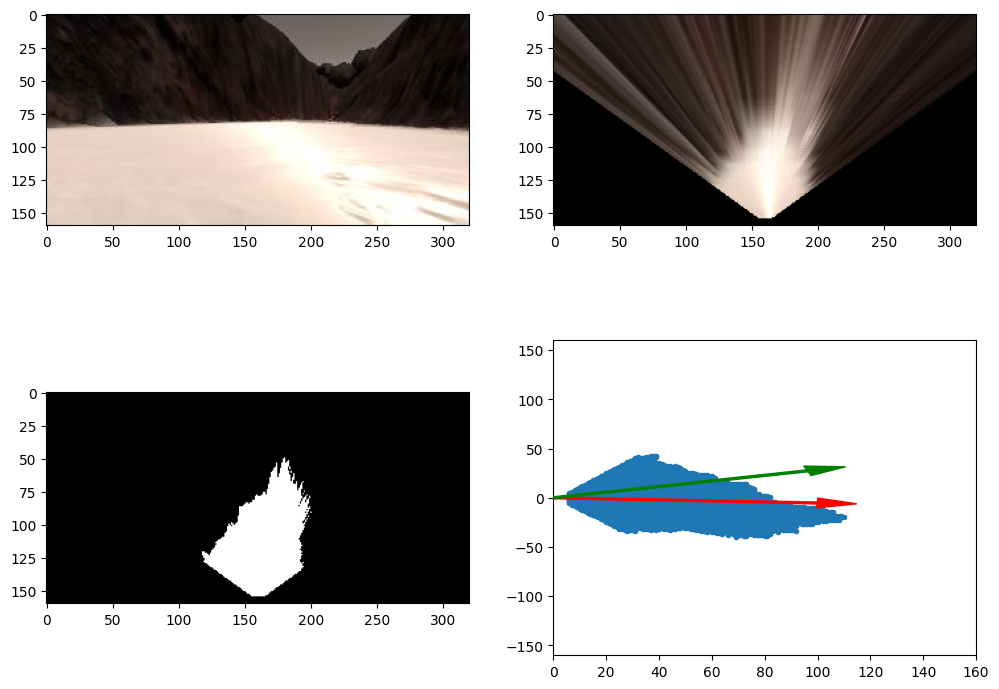
Description automatically generated

Adding a positive offset (to the left in our case) to the mean angle is an easy method to implement the wall-following behaviour (shown above). But we soon realised that a constant offset was insufficient since the Rover would occasionally turn too much and run into a wall, as can happen in a hallway, or it would turn insufficiently and occasionally miss corners entirely.

We made the decision to change the offset in accordance with the standard deviation of the angles as a result. Small offsets on straight lines and big offsets in corners and open spaces were made possible by this:

# Steering proportional to the deviation results in# small offsets on straight lines and# large values in corners and open areasoffset = 0.8 \* np.std(Rover.nav\_angles)

His straightforward method produced extremely robust behaviour. Through trial and error, we decide to scale the standard deviation by 0.8. While maintaining a nearly constant distance from the wall at all times, the Rover continued to steer, leaving the majority of the navigable pixels to the right.



## Stuck state

The terrain is filled with hazardous areas, and the Rover frequently gets stuck. To address this, we've included a new state called "stuck." The rover will attempt to turn to the right for a brief period of time after becoming stuck (this was observed empirically) and then return to the initial state. we changed the code to provide a straightforward state stack that made it simple for the Rover to return to the previous state:

Text, letter

Description automatically generated

Text, letter

Description automatically generatedIf the Rover is in forward mode, is not moving, and has not been in the stuck position for at least 4 seconds, it enters the stuck state. To prevent the Rover from alternating between the forward and stalled modes, the final condition was required. The code appears as follows: