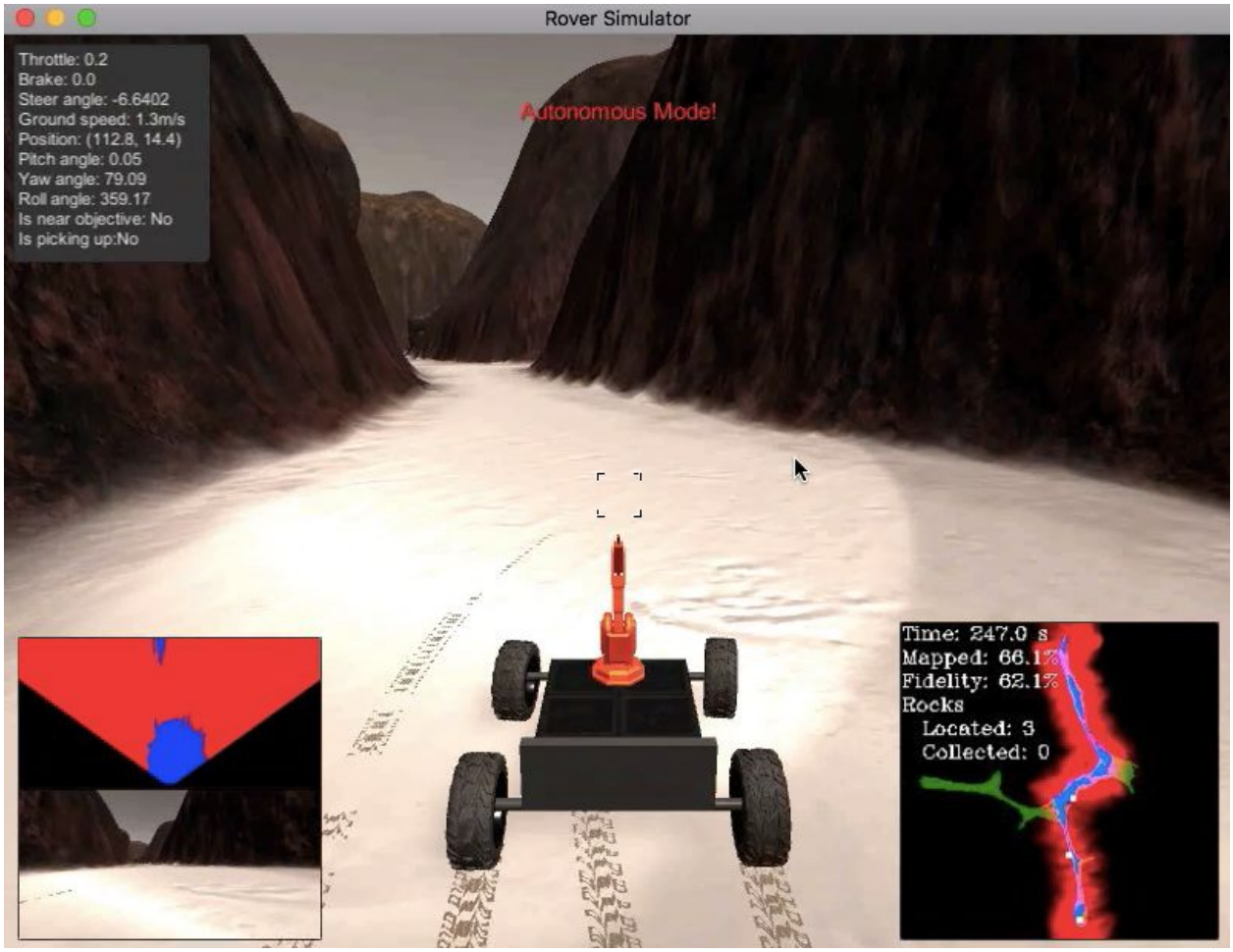
Search and Sample return

Overview



In this project, I learned

1. how to use analysis robot perception using color threshold computer vision algorithm to distinguish navigable terrain, obstacles and identify target object.
2. how to transfer rover viewing image to Rover-Centric coordinate system, and map Rover-Centric image to the ground truth world image
3. how to use decision trees to navigation robot around the terrain based on the rover camera data

Notebook Analysis

In the notebook, I add find_rocks function to identify rock sample.

```
def find_rocks(img, levels=(90, 90, 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;

rock_map = find_rocks(rock_img)
fig = plt.figure(figsize = (12,3))
plt.subplot(121)
plt.imshow(rock_img)
plt.subplot(122)
plt.imshow(rock_map, cmap='gray')
```

The rock identification result shows in Figure 1.

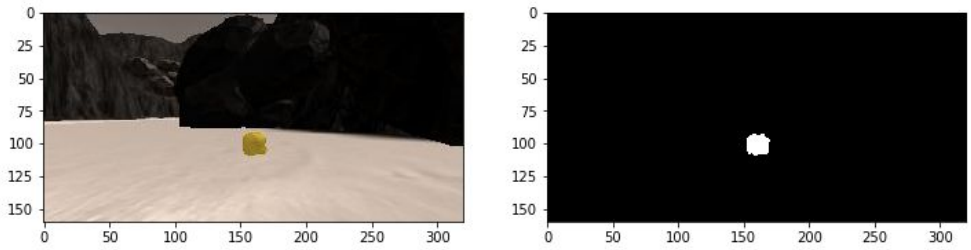
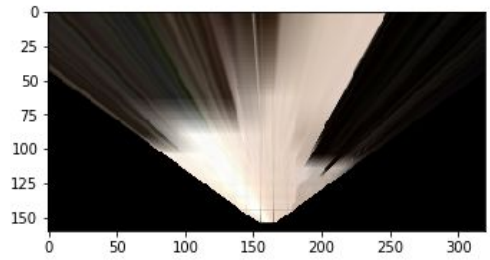
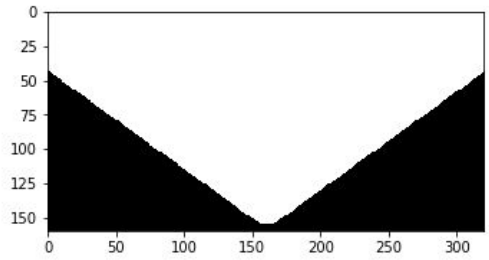


Figure1. Shows the result of color thresholding method for identifying rock sample.
The code uses array masking method to identify rock sample. We first create a binary image(only 0, and 1) with the dimension of img. Rockpix is the RGB color array mask for each color channel, which means that each pixel on the color image meets the threshold values in RGB will contains a boolean array with True. Every true position will be mapped as 1 value in the color_select binary image.

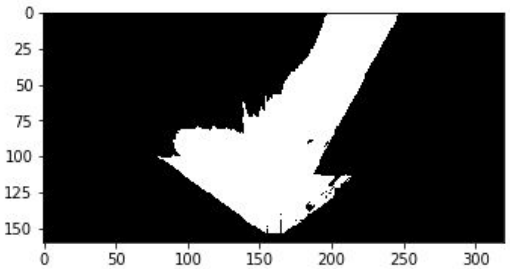
For obstacles, first use persepct_transform function to get the warped and terrain mask image. Then, use mask image to do the oppose selection of obstacles.

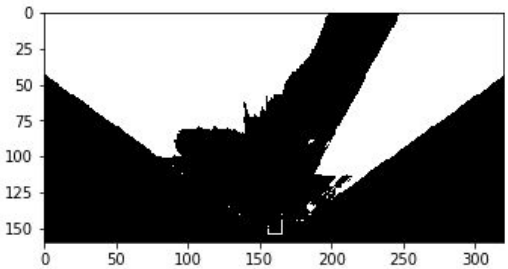
```
def perspect_transform(img, src, dst):

    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))#
keep same size as input image
    mask = cv2.warpPerspective(np.ones_like(img[:,:,0]), M , (img.shape[1],
img.shape[0]))
    return warped, mask
```






```
Then, use color_thresh to get the navigable terrain and use the mask to get
the obstacle area.
def color_thresh(img, rgb_thresh=(160, 160, 160)):
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will now contain a boolean array with "True"
    # where threshold was met
    above_thresh = (img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,1] > rgb_thresh[1]) \
                & (img[:,:,2] > rgb_thresh[2])
    # Index the array of zeros with the boolean array and set to 1
    color_select[above_thresh] = 1
    # Return the binary image
    return color_select

threshed = color_thresh(warped)
fig = plt.figure(figsize=(12,3))
plt.subplot(121)
plt.imshow(threshed, cmap='gray')
obs_map = np.absolute(np.float32(threshed) -1) * mask
plt.subplot(122)
plt.imshow(obs_map, cmap='gray')
```





Autonomous Navigation and Mapping

```
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
    # Define calibration box in source (actual) and destination (desired)
coordinates
    # These source and destination points are defined to warp the image
    # to a grid where each 10x10 pixel square represents 1 square meter
    # The destination box will be 2*dst_size on each side
    dst_size = 5
    # Set a bottom offset to account for the fact that the bottom of the
image
    # is not the position of the rover but a bit in front of it
    # this is just a rough guess, feel free to change it!
    bottom_offset = 6
    image = Rover.img
    source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
    destination = np.float32([[image.shape[1]/2 - dst_size, image.shape[0]
- bottom_offset],
                      [image.shape[1]/2 + dst_size, image.shape[0] -
bottom_offset],
                      [image.shape[1]/2 + dst_size, image.shape[0] -
2*dst_size - bottom_offset],
                      [image.shape[1]/2 - dst_size, image.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);
    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[:,:,0] = obs_map * 255


    # 5) Convert map image pixel values to rover-centric coords
    xpix, ypix = rover_coords(threshed);
    # 6) Convert rover-centric pixel values to world coordinates
    world_size = Rover.worldmap.shape[0];
    scale = 2 * dst_size;
    x_world, y_world = pix_to_world(xpix, ypix, Rover.pos[0], Rover.pos[1],
Rover.yaw, world_size, scale);
    obsxpix, obsypix = rover_coords(obs_map)
    obs_x_world, obs_y_world = pix_to_world(obsxpix, obsypix, Rover.pos[0],
Rover.pos[1], Rover.yaw, world_size, 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_world, x_world, 2] += 10;
    Rover.worldmap[obs_y_world, obs_x_world, 0] += 1

    # 8) Convert rover-centric pixel positions to polar coordinates
    dist, angles = to_polar_coords(xpix, ypix);
    Rover.nav_angles = angles;
    # Rover.nav_dists = dist;

    # Find rocks
    rock_map = find_rocks(warped, levels=(90, 90, 50));
    if rock_map.any():
        rock_x, rock_y = rover_coords(rock_map)

        rock_x_world, rock_y_world = pix_to_world(rock_x, rock_y,
Rover.pos[0],
                                                  Rover.pos[1],
Rover.yaw, world_size, scale)

        rock_dist, rock_ang = to_polar_coords(rock_x, rock_y)
```

```
        rock_idx = np.argmin(rock_dist);
        rock_xcen = rock_x_world[rock_idx];
        rock_ycen = rock_y_world[rock_idx];

        Rover.worldmap[rock_ycen, rock_xcen, 1] = 255
        Rover.vision_image[:,:,1] = rock_map * 255
    else:
        Rover.vision_image[:,:,1] = 0;
```

Applied the functions in the notebook, to update the world map on the right corner of the simulator and update the rover view image on the left corner of the simulator.

```
For decision_steps:
    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), -10, 10)
            # 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 = -20 # 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), -10, 10)
                    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 the robot is in the driving forward mode, if the extent of navigable terrain pixels are bigger than the stop_forward threshold. The Robot will steer towards that direction. During this move forward action, if robot velocity is less than the max velocity it can reach, the robot will accelerate to the max speed. If the robot already drives at max speed, then do coast. If there is a lack of navigable terrain pixels then the robot will got to stop mode.

In the stop mode, we should decrease the speed if the robot speed is bigger than 0.2m/s. If the speed is already less than 0.2m/s, the robot stopped and the robot will look around to find enough navigable terrain pixels. In this stop mode, if the navigable angle is less than the go forward threshold angle, we will let robot turn 20 degree to the left till it finds enough navigable pixels. The drawback of this simple approach is that the robot will spin at the same place. This point will be my future improvement. Then, if the robot find enough navigable terrain pixels, it will go forward.

Video: https://www.dropbox.com/s/nqqrhdob88zv9de/proj1_video.mp4?dl=0

Simulator Configuration: