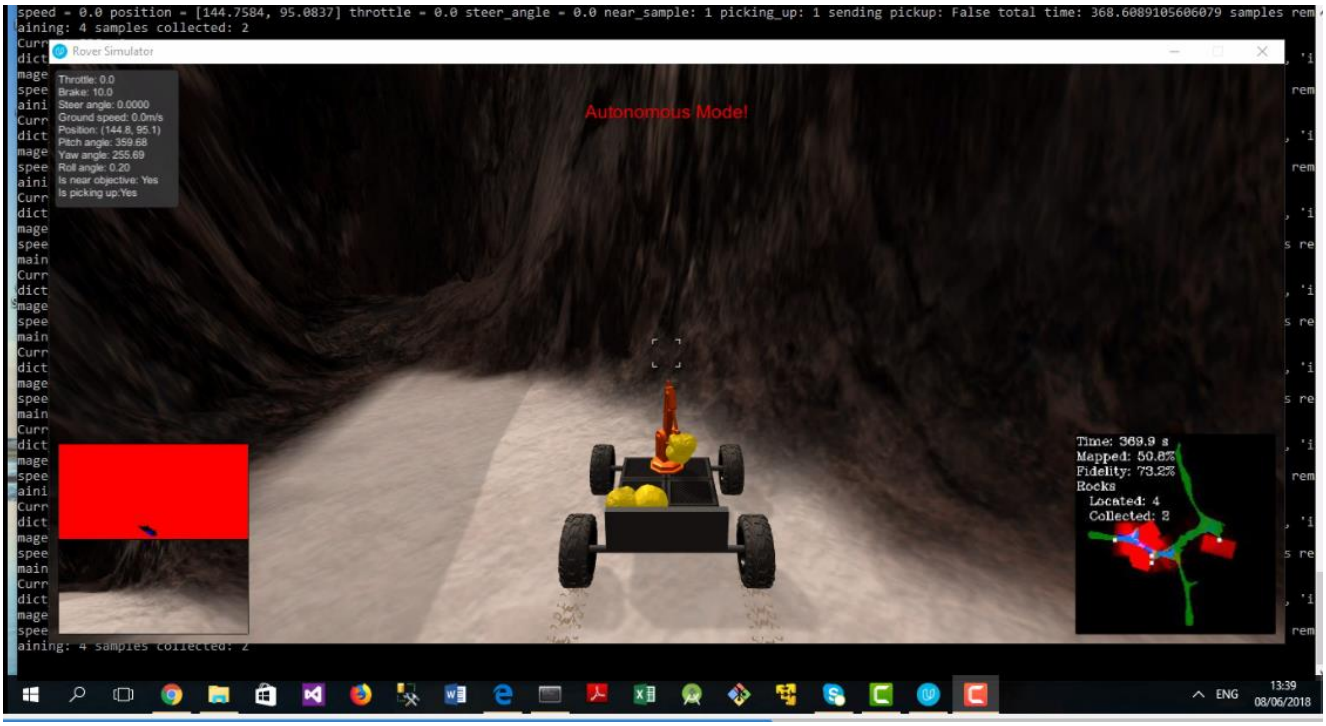
Search and Return Project

Introduction

This is the project one (1) of Nanodegree Robotics Software Engineering at Udacity. The goal of this project is to write codes that autonomously map a simulated environment and search for samples of interest. This provides hands-on experience with three essential elements of robotics which includes perception, decision making and actuation.



Notebook Analysis

This project is provided with a notebook which contains functions that I completed to allow the simulated environment to be mapped. The steps are discussed below:

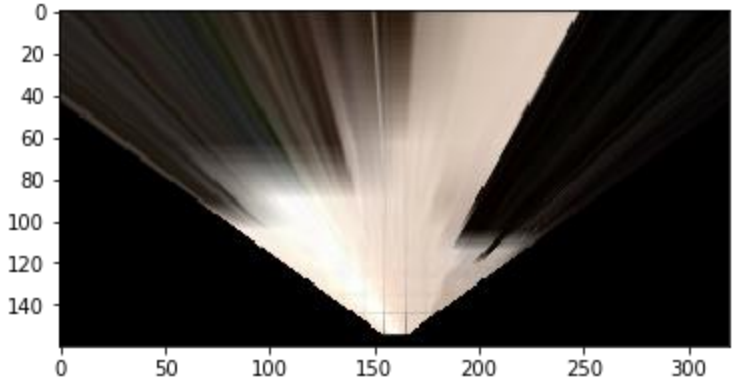
Perspective transform

Perspective transform is applied to the camera images coming from the rover to enable change the perspective seen from the rover's camera to the one where you are looking down on the world from above.

```
Define a function to perform a perspective transform
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
    return warped
```

Color Thresholding

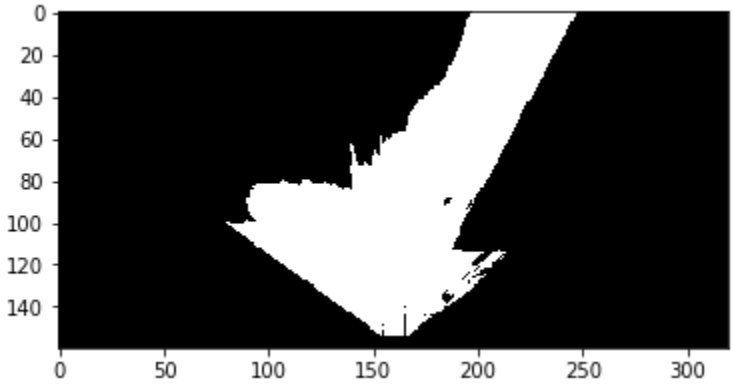
Thresholding methods replace each pixel in the images with white pixel if the intensity of the image is greater than fixed constant to represent navigable terrain while a black pixel if the image intensity is less than fixed constant to represent an obstacle. Throughout the simulated



environment the sand is very light in color and everything else is dark.

Navigable terrain

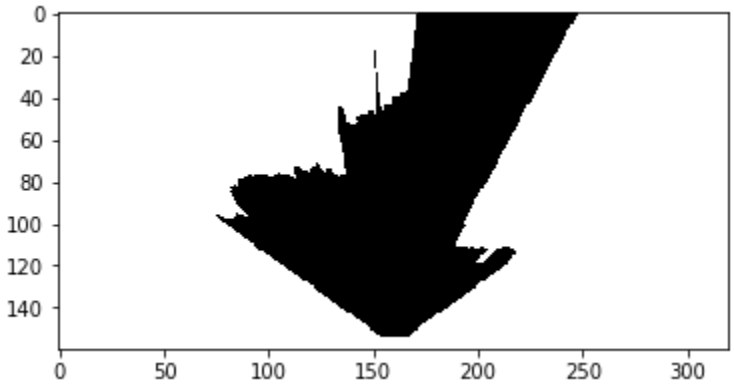
In this case pixel that has intensity greater than 160 in all three channels represents our navigable terrain.




```
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
```

Obstacles

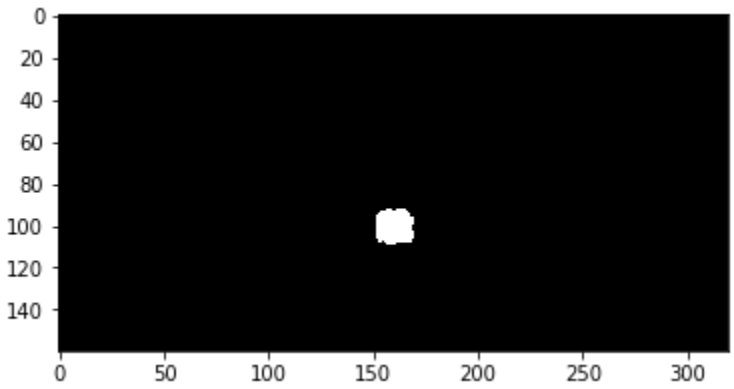
Likewise any pixel with intensity less than 130 in all three channels represents obstacle. Therefore the obstacles are inverse of the navigable terrain.




```
def obstacle_thresh(img, rgb_thresh=(130,130,130)):
 # 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
 below_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[below_thresh] = 1
 # Return the binary image
 return color_select
```

Rock Samples

Finally, the color threshold is also used to identify the rock samples. The rock samples are yellow in color and falls under HSV filter range of [20,150,100] and [50,255,255]. I use OpenCV function **inRange** to perform the color range threshold. And this function requires HSV image, I therefore converted the image from RGB to HSV with upper and lower bounds with finally return mask that isolate the rock sample on the image.

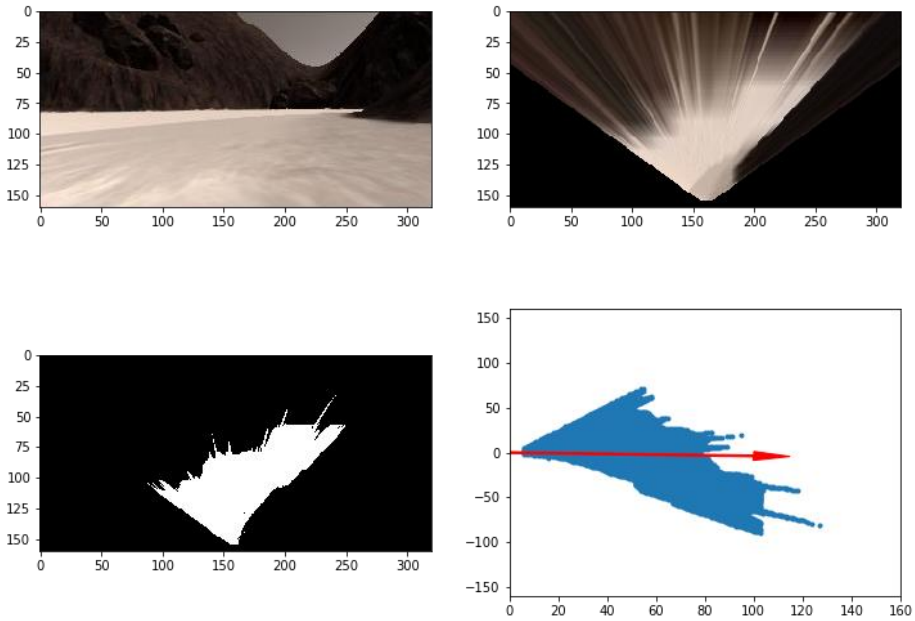



```
def rock_thresh(img):
    # Convert RGB to HSV using openCV
    hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV, 3)

    #Define range of  yellow colors in HSV
    lower_yellow = np.array([20,150, 100], dtype='uint8')
    upper_yellow = np.array([50, 255, 255], dtype='uint8')

    # Threshhold the HSV image to get only yellow colors
    mask = cv2.inRange(hsv,lower_yellow, upper_yellow)
    return mask
```

Coordinate Transformations

Rover-centric coordinate

This change navigable terrain of the image to be relative to the front of the rover. This transformed the coordinate frame to rover centric. These images show transition from rover's camera to rover centric navigable image.



```
# Define a function to convert from image coords to rover coords
def rover_coords(binary_img):
    # Identify nonzero pixels
    ypos, xpos = binary_img.nonzero()
    # Calculate pixel positions with reference to the rover position being at the
    # center bottom of the image.
    x_pixel = -(ypos - binary_img.shape[0]).astype(np.float)
    y_pixel = -(xpos - binary_img.shape[1]/2 ).astype(np.float)
    return x_pixel, y_pixel
```

World coordinates

After map of the navigable terrain in rover centric coordinates is generated, the next step was to map those points to word coordinates to allow you to use rover's position, orientation and camera image to map its environment and compare against the ground truth.

```
# Define a function to apply rotation and translation (and clipping)
# Once you define the two functions above this function should work
```

```
def pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale):
    # Apply rotation
    xpix_rot, ypix_rot = rotate_pix(xpix, ypix, yaw)
    # Apply translation
    xpix_tran, ypix_tran = translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale)
    # Perform rotation, translation and clipping all at once
    x_pix_world = np.clip(np.int_(xpix_tran), 0, world_size - 1)
    y_pix_world = np.clip(np.int_(ypix_tran), 0, world_size - 1)
    # Return the result
    return x_pix_world, y_pix_world
```

Perception module

Most of the code of *perception.py* have been done in the *process_image()* method in the notebook. So I moved the codes from *process_image()* to the *perception_steps()* method in the *perception.py*. In addition it includes a restriction what values are added to the world map. This is performed to improve the fidelity of the data captured. The data is restricted based on pitch and roll of the rover, where each can be 0.5 degrees off the axis.

```
 # 7) Update Rover worldmap (to be displayed on right side of screen)
   if(Rover.pitch <0.5 or Rover.pitch >359.5) and (Rover.roll <0.5 or Rover.roll>359.5):
      Rover.worldmap[obstacles_y_world, obstacles_x_world, 0] += 1
      Rover.worldmap[rocks_y_world, rocks_x_world, 1] =255
      Rover.worldmap[navigable_y_world, navigable_x_world, 2] +=1
```

Finally, I computed the navigable terrain and rock sample into polar coordinate system allow rover to best approach the sample rocks.

```
dists, angles = to_polar_coords(navigable_xpix, navigable_ypix)
```

```
    # Update Rover pixel distances and angles
    Rover.nav_dists = dists
    Rover.nav_angles = angles
if len(rocks_xpix) > 5:
        # if rock is identified, make the rover to navigate it
        rock_dist, rock_angle = to_polar_coords(rocks_xpix, rocks_ypix)
```

Perception module

decision.py contains routine that enable rover to make decisions based on its state. Below described main changes made to the *decision_steps()* in the decision.py file as follows:

forward

This part contains codes for determining where the rover should drive forward. The average angle of the navigable area is used. If there are more than fifty (50) values in the angle list, then increase the throttle to the maximum speed and go the average navigable angle else change to stop mode. This also includes codes for approaching sample rocs. If a sample rock is nearby navigate directly toward the sample rock. If it within the range of -15 to 15 off rover's x-axis. If it is greater than 15 degrees and less 50 degrees off the rover's x-axis rotate near the sample of the rock. When rover is near to the sample rock, it will stop then the arm will pick up the sample and continue to search for more samples.

```
if Rover.mode == 'forward':
```

```
        # Check the extent of navigable terrain
        if Rover.vel < 0.2 and Rover.throttle != 0:
            # if the velocity is still 0 after throttle, it's stuck
            if time.time() - Rover.stuck_time > Rover.max_stuck:
            #initiate stuck mose after 5 seconds of moving
                Rover.mode = 'stuck'
                return Rover
            else:
                # Reset stuck time
                Rover.stuck_time = time.time()

        if Rover.sample_seen:
            if Rover.picking_up !=0
                # Reset sample_seen flag
                Rover.sample_seen = False
                Rover.sample_timer = time.time()
                return Rover

            if time.time() - Rover.sample_timer > Rover.sample_max_search:
                Rover.sample_seen = False
                Rover.sampler_time = time.time()
                return Rover
            avg_rock_angle = np.mean(Rover.rock_angle * 180/np.pi)
            if -15 < avg_rock_angle < 15:
                # Only drive staright for sample if it's within 13 deg
                if max(Rover.rock_dist) < 20:
                    Rover.throttle = 0
                    Rover.brake = Rover.brake_set
                    Rover.steer = avg_rock_angle
                else:
                    # Set throttle at half normal speed during approach
                    Rover.throttle = Rover.throttle_set
                    Rover.steer = avg_rock_angle
            elif -50 < avg_rock_angle < 50:
                if Rover.vel > 0 and max(Rover.rock_dist) < 50:
                    Rover.throttle = 0
                    Rover.brake = Rover.brake_set
```

```
                Rover.steer = 0
            else:
                Rover.throttle = 0
                Rover.brake = 0
                Rover.steer = avg_rock_angle/6
        else:
            # keep logic simple and ignore samples +/- degrees
            Rover.sample_seen = False
    elif len(Rover.nav_angles) > 50:
        # if mode is forward, navigable 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:
            Rover.throttle = 0
        Rover.brake = 0
        # set steering to avearge angle to the range +/-15
        Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)
    else:
        # if there is lack of navigable terrain then to stop mode

        # set mode to stop and hit the brake
        Rover.throttle = 0
        Rover.brake = Rover.brake_set
        Rover.steer = 0
        Rover.mode = 'stop'
```

Stop

The stop section of the code allow the rover to rotate to a new direction when it is against an obstacle where the only

way is to turn around and look for clear path ahead of it and move to forward mode.

```
# if we're already in stop mode then make diffrent decision
        elif Rover.mode == 'stop':
            # if we're in stop mode but still moving keep brakin
            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:
                # Rover is stopped with vision data; see if there's a path forward
                if len(Rover.nav_angles) < 100:
                    Rover.throttle = 0
                    Rover.brake = 0
                    # Turn range +/- 15 deg, when stooped the next line will induce 4-whell turning
                    Rover.steer = -15
                else:
                    # 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'
```

Stuck

Stuck section involves rover reacting to a stuck over. If the rover stuck on an obstacle, it is given throttle but the velocity remain while if it is caught in a circle continually to steer in a circle.

```
if Rover.mode == 'stuck':
                if time.time() - Rover.stuck_time > (Rover.max_stuck + 1):
```

```
            Rover.mode = 'forward'
            Rover.stuck_time = time.time()
        else:
            # Perform evasion to get unstuck
            Rover.throttle = 0
            Rover.barke = 0
            Rover.steer = -15
        return Rover
```

Performance

The code has relatively good performance mapping and navigating the simulated environment. The rover is able to mapped 51% of the area and has a fidelity of around 72%. In addition the rover is able to collect three rocks.

Future Improvement

- ✓ Improve on obstacle identification. Many times the rover by pass rock samples.
- ✓ Improve navigation. Rover should able to navigate all nooks and crannies with the environment.