
# Rover Project

# PROJECT 1

Rover Project

NOVEMBER 10, 2017

## Introduction

In a mars simulated environment, I will be mapping out the map of the universe as it gets discovered by an autonomous rover moving on the surface of mars. This project is for educational purposes to get myself familiarised with OpenCV and other python libraries along with undertadning basic robotics concepts.

*Please check out the python notebook included in the repo for output movie constructed from my own collected data*

# Functions used:

### perspect_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
```

The function above take a FPV (first person view) image and turns it into an eagle eye image.

M is a matrix that maps the source points to destination points. warpPerspective is a function that uses M and apply it to the image to change the perspective of view.

### color_thresh_nav

```
def color_thresh_nav(img, rgb_thresh=(160, 160, 160)):#lower thresh

    # Create an array of zeros same xy size as img, but single channel

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

    mask = np.zeros_like(img[:,:,0])

    mask[int(mask.shape[0]/2):mask.shape[0]-1,:] = 1

    # 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

    navigable = (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[navigable] = 1

    color_select = cv2.bitwise_and(color_select,color_select,mask = mask)

    return color_select
```

The function above is used to create a binary image mask of the elements that correspond to a navigable terrain from the warped image. I have included another rectangular mask in there to ignore points that are far away. I did that because the closer the points to the rover the more accurate the classification is. So ignoring far away points tends to reduce false negative classification of terrain.

What I highlighted shows the lines that pertain to the mask.

---

## color_thresh_obs

```
def color_thresh_obs(img, rgb_thresh=(160, 160, 160)):#lower thresh
    # 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
    navigable = (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[navigable] = 1


    # Return the binary image
    return color_select
```

Literally the oppositve of the previous function, the enequality directions were reversed. If something is not terrain, it is an obstacle.

---

## color_thresh_rock

```
def color_thresh_rock(img, rgb_lower=(60, 60, 0), rgb_upper=(255, 255 ,30)):
    color_select = np.zeros_like(img[:,:,0])
    rock = (img[:,:,0] > rgb_lower[0]) \
            & (img[:,:,1] > rgb_lower[1]) \
            & (img[:,:,2] > rgb_lower[2]) \
            & (img[:,:,0] < rgb_upper[0]) \
            & (img[:,:,1] < rgb_upper[1]) \
            & (img[:,:,2] < rgb_upper[2])
            color_select[rock] = 1
    return color_select
```

This function detect rocks. After observing the colours contained in the rocks I was able to roughly find a min and max limits for the rock. This function thresholds for values between these min and max values.

---

## rover_coords

```
def rover_coords(binary_img):
    # Identify nonzero pixels
    ypos, xpos = binary_img.nonzero()
    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
```

Taking in a binary eagle eye view image and change the coordingated from image coorintaes to rover coordinates.

---

## pix_to_world

```
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
```

Function above takes in the x and y values fom the point of view of the rover and places them in their correct places in the global map view. No changes were made here.  rotate_pix and translate_pix functions does as expected of the name, no changes were done by me here either.

---

## perception_step

```
def perception_step(Rover):

    image = Rover.img

    xpos = Rover.pos[0]

    ypos = Rover.pos[1]

    yaw = Rover.yaw

    dst_size = 5

    bottom_offset = 6

    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 = perspect_transform(image, source, destination)

    # 3) Apply color threshold to identify navigable
terrain/obstacles/rock samples

    navigable = color_thresh_nav(warped) #bin image of eagle eye
```

```
rock = color_thresh_rock(warped)

obs = color_thresh_obs(warped)


# 4) Update Rover.vision_image (this will be displayed on left side of screen)

# Rover.vision_image[:,:,0] = obs*255 #obstacle color-thresholded binary image

# Rover.vision_image[:,:,1] = rock*255 #rock_sample color-thresholded binary image

Rover.vision_image[:,:,2] = navigable*255 #navigable terrain color-thresholded binary image


# 5) Convert map image pixel values to rover-centric coords

nav_rover_x, nav_rover_y= rover_coords(navigable)

rock_rover_x, rock_rover_y = rover_coords(rock)

obs_rover_x, obs_rover_y = rover_coords(obs)


# 6) Convert rover-centric pixel values to world coordinates

nav_world_x, nav_world_y = pix_to_world(nav_rover_x, nav_rover_y, xpos, ypos, yaw, 200, 10)

rock_world_x, rock_world_y = pix_to_world(rock_rover_x, rock_rover_y, xpos, ypos, yaw, 200, 10)

obs_world_x, obs_world_y = pix_to_world(obs_rover_x, obs_rover_y, xpos, ypos, yaw, 200, 10)


# Rover.worldmap[obs_world_y, obs_world_x, 0] += 1

# Rover.worldmap[rock_world_y, rock_world_x, :] = 255

Rover.worldmap[nav_world_y, nav_world_x, 2] += 10

dist, angle = to_polar_coords(nav_rover_x, nav_rover_y)

Rover.nav_dists = dist

Rover.nav_angles = angle
```

The function above combines all the functions from before. It take an FPV image from a rover and updates important rover instance variables. The instance variables that this function updates are:

1. The rover_vision image used to visualize what the rover is seeing.
2. The world map used to visualize map discovery coverage
3. The Rover.nav_dists and Rover.nav_angles which are arrays that store the navigable terrain distances and angles (in polar coordinated). Both are used in the decision step function, they help the rover decide where to head.

---

## decision_step

I am not going to include everything I decision step. I will just highlight my changes and present a state machine of the rover to show my understanding of the code.

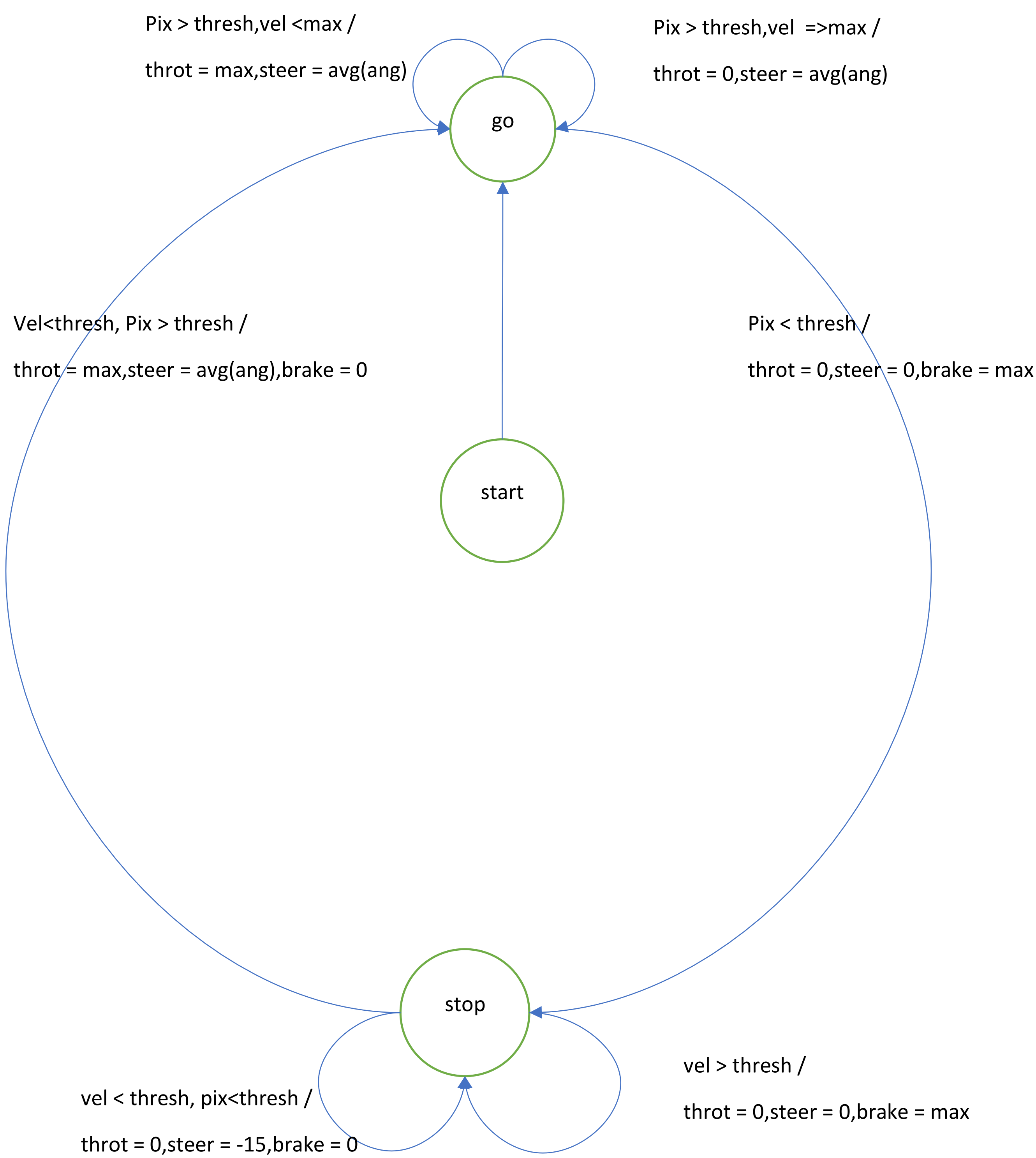
I essentially changed only one line of code:

Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi) + 17, -10, 10)

From

Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi) , -15, 15)

Which has the following effect. The +17 can be thought of as the target angle. If you think of it in terms of control systems, it is the set point. The other clipping values can be thought of as the aggressivent at which we are steering. I realized the more momdetum we have going forward the less I am likely to keep hitting the wall when I am moving next to it, as I lower these values to favour the forward momentum over sharper turns.



Pix > thresh,vel <max /
throt = max,steer = avg(ang)
Pix > thresh,vel =>max /
throt = 0,steer = avg(ang)
go
Vel<thresh, Pix > thresh /
throt = max,steer = avg(ang),brake = 0
Pix < thresh /
throt = 0,steer = 0,brake = max
start
stop
vel < thresh, pix<thresh /
throt = 0,steer = -15,brake = 0
vel > thresh /
throt = 0,steer = 0,brake = max

# Appendinx:

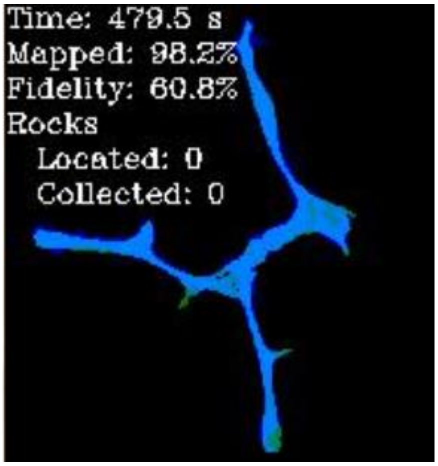



*Figure 1 Output map of a successful run, that doesn't often happen, and it gets stuck frequently.*

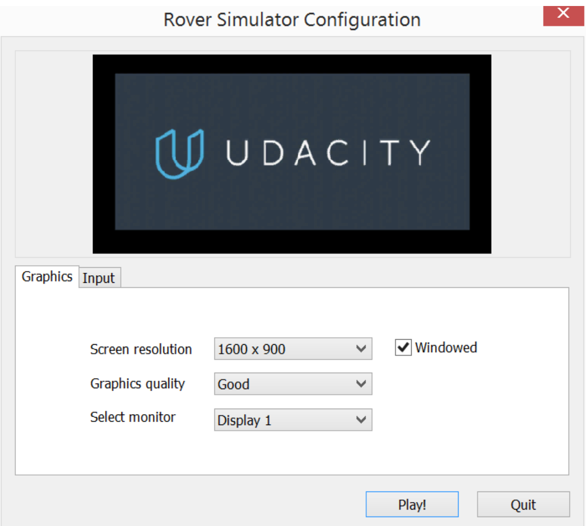



*Figure 2 settings used*