Search & Sample Return Project Report

Notebook Analysis

Obstacle Identification

For obstacles detection a new function ‘obs’ was added to the color thresholding part of the notebook. The function identifies pixel locations below the rgb threshold that is set for detecting ground pixels and returns it as obstacles location. The threshold range was set with RGB values investigated using the matplotlib interactive window.

```
def obs(img, rgb_thresh=(230, 100, 100)):
            obs_select = np.zeros_like(img[:,:,0])
            below_thresh = (img[:,:,0] < rgb_thresh[0]) \
                        & (img[:,:,1] < rgb_thresh[1]) \
                        & (img[:,:,2] < rgb_thresh[2])
            obs_select[below_thresh] = 1
            return obs_select
```

Rock Sample Identification

For rock sample identification, HSV color space values were used. OpenCV’s cv2.cvtColor() was used to convert the rock sample BGR to HSV color space. The BGR values for the rock were investigated using the matplotlib window and were passed on to the cvtColor() function. A lower and upper bound was set to define the gold color range and then cv2.inRange() function was used to threshold the HSV image to get only gold colors. The result was obtained by applying Bitwise_AND operation to merge the mask with the original image in such a way that areas with white pixels in the mask are shown while areas with black pixels in the mask are not shown.

```
# Identify rock pixels and plotting
rock_img = mpimg.imread('../calibration_images/example_rock1.jpg')
gold = np.uint8([[[140,110,0]]])
hsv_gold = cv2.cvtColor(gold,cv2.COLOR_BGR2HSV)
hue = hsv_gold[0][0][0]
hsv = cv2.cvtColor(rock_img, cv2.COLOR_BGR2HSV)
lower_gold = np.array([hue-10,100,100])
upper_gold = np.array([hue+10,255,255])
gmask = cv2.inRange(hsv, lower_gold, upper_gold)
res = cv2.bitwise_and(rock_img, rock_img, mask= gmask)
plt.imshow(res[:,:])
```

Process_Image Function

The process_image function was modified by adding the perception step processes (perspective transformation, color thresholding and coordinate transformation functions). Each individual image is passed on to the process_image function and the resulting output image is stored as one frame of video. In order to make the map of the environment, the input image is first passed through perspective_transform and color_threshold functions. This creates a binary image showing the navigable terrain in front of the rover. However, it provides a top-level view of the world. A robot centric coordinate system is more desirable as it describes the positions of things in the environment with respect to the robot. To convert this view into a rover-centric view, a coordinate transformation is applied to the pixel positions of all navigable terrain pixels using the rover_coords function. The resulting pixels are located in rover-centric x and y coordinate frame. With the map of navigable terrain

in rover-coordinates, the points are then mapped to world-coordinates by rotation and translation operations which account for arbitrary angle and position of rover while it takes a picture. To carry out this work, pix_to_world function is defined which calls two other functions rotate_pix and translate_pix for the above mentioned rotation and translation with scaling operations.

With the navigable terrain pixel positions in rover coordinates, the next step is to determine the direction for rover to move. The rover coordinate pixel positions are first converted to polar coordinates in rover space using to_polar_coords function which provides the distance of each pixel from the origin and the angle away from vertical for each pixel. Then the average angle of direction of all navigable terrain pixels in rovers field of view is computed which is converted to the steering angle that gets sent to the rover.

```
def process_image(img):
    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

    def color_thresh(img, rgb_thresh=(160, 160, 160)):
               color_select = np.zeros_like(img[:,:,0])
               above_thresh = (img[:,:,0] > rgb_thresh[0]) \
                           & (img[:,:,1] > rgb_thresh[1]) \
                           & (img[:,:,2] > rgb_thresh[2])
               color_select[above_thresh] = 1
               return color_select

    def obs(img, rgb_thresh=(230, 100, 100)):
               obs_select = np.zeros_like(img[:,:,0])
               below_thresh = (img[:,:,0] < rgb_thresh[0]) \
                           & (img[:,:,1] < rgb_thresh[1]) \
                           & (img[:,:,2] < rgb_thresh[2])
               obs_select[below_thresh] = 1
               return obs_select

    def rover_coords(binary_img):
               ypos, xpos = binary_img.nonzero()
               x_pixel = 0.35*(-(ypos - binary_img.shape[0]).astype(np.float))
               y_pixel = 0.35*(-(xpos - binary_img.shape[1]/2 ).astype(np.float))
               return x_pixel, y_pixel

    def to_polar_coords(x_pixel, y_pixel):
               dist = np.sqrt(x_pixel**2 + y_pixel**2)
               angles = np.arctan2(y_pixel, x_pixel)
               return dist, angles

    def rotate_pix(xpix, ypix, yaw):
               yaw_rad = yaw * np.pi / 180
               xpix_rotated = (xpix * np.cos(yaw_rad)) - (ypix * np.sin(yaw_rad))
               ypix_rotated = (xpix * np.sin(yaw_rad)) + (ypix * np.cos(yaw_rad))
               return xpix_rotated, ypix_rotated

    def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
               xpix_translated = (xpix_rot / scale) + xpos
               ypix_translated = (ypix_rot / scale) + ypos
               return xpix_translated, ypix_translated

    def pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale):
               xpix_rot, ypix_rot = rotate_pix(xpix, ypix, yaw)
               xpix_tran, ypix_tran = translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale)
               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 x_pix_world, y_pix_world
```

Worldmap

The worldmap is an empty array instantiated in the Databucket() class as 200 x 200 grids corresponding to a 200m x 200m space.. As the rover moves and finds navigable terrain pixels, they are added to the worldmap in the color channel 2 (blue) in the process_image() function. The obstacle pixel locations are added in the color channel 0 (red) and the rock sample pixel locations are added in all three color channel with 255 value (white).

```
# Get navigable pixel positions in world coords
    x_world, y_world = pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale)
    data.worldmap[y_world, x_world, 2] = 255
# Get obstruction pixel positions in world coords
    xo_world, yo_world = pix_to_world(xobs, yobs, xpos, ypos, yaw, world_size, scale)
    data.worldmap[yo_world, xo_world, 0] = 255
# Get rock pixel positions in world coords
    xgw, ygw = pix_to_world(xgol, ygol, xpos, ypos, yaw, world_size, scale)
    data.worldmap[ygw, xgw, :] = 255
```

The output of process_image function is stored as output_image which is instantiated as an array of zeros with an arbitrary shape. First region of this output_image is populated with the original image in the upper left hand corner. The warped image is added to the upper right corner and then the world map is plotted on top of the ground_truth map in the lower left corner using the cv2.addWeighted() function.

```
# Making a mosaic image
# First create a blank image
    output_image = np.zeros((img.shape[0] + data.worldmap.shape[0], img.shape[1]*2, 3))

# Populating regions of the image with various output
# The original image is in the upper left hand corner
    output_image[0:img.shape[0], 0:img.shape[1]] = img

# Creating more images to add to the mosaic, first a warped image
# Adddding the warped image in the upper right hand corner
    warped = perspect_transform(img, source, destination)
    output_image[0:img.shape[0], img.shape[1]:] = warped

# Overlay worldmap with ground truth map
    map_add = cv2.addWeighted(data.worldmap, 1, data.ground_truth, 0.5, 0)

# Flip map overlay so y-axis points upward and add to output_image
    output_image[img.shape[0]:, 0:data.worldmap.shape[1]] = np.flipud(map_add)
```

Autonomous Navigation and Mapping

All the functions defined in process_image() in the notebook were used in the perception_step function in perception.py script where the object Rover is passed from the drive.py script. The Rover.vision_image was updated with thresholded binary image for obstacles in the red channel, navagable terrain in blue channel and rock samples in all three channels. Similarly, worldmap was created using Rover.worldmap following the steps from process_image() function.

```
# Updating the image with navigable terrain, obstacles and rocks
Rover.vision_image[:,:,0] = warpedobs*255
Rover.vision_image[:,:,1] = gmask*255
Rover.vision_image[:,:,2] = warped*255

# Calculating map pixel image values for navigable terrain, obstacles and rock samples in Rover Coordinates
xpix, ypix = rover_coords(warped)
xobs, yobs = rover_coords(warpedobs)
xgol, ygol = rover_coords(gmask)

# Converting Rover Centric Pixel positions to Polar Coordinates
dist, angles = to_polar_coords(xpix, ypix)

# Converting Rover Centric pixel values to World Coordinates
world_size = Rover.worldmap.shape[0]
scale = 2*dst_size

# Navigable pixels
x_world, y_world = pix_to_world(xpix, ypix, Rover.pos[0], Rover.pos[1], Rover.yaw, world_size, scale)

# Obstacle pixels
xo_world, yo_world = pix_to_world(xobs, yobs, Rover.pos[0], Rover.pos[1], Rover.yaw, world_size, scale)

# Rock sample pixels
xgw, ygw = pix_to_world(xgol, ygol, Rover.pos[0], Rover.pos[1], Rover.yaw, world_size, scale)

# Updating Rover Worldmap
Rover.worldmap[y_world, x_world, 2] += 50
Rover.worldmap[yo_world, xo_world, 0] += 100
Rover.worldmap[ygw, xgw, :] = 255
```

The decision_step() in decision.py script was not modified. Two Rover modes were defined as Forward and Stop for the input value of the Rover.nav_angles. Within the modes, throttle, brake and steering decisions were formed and their values were set. These decisions were formed based on the total count of navigable terrain pixels set in the drive.py script in stop_forward and go_forward fields. With no nav angles input, the Rover was set to go in a straight line forward with a throttle value set in drive.py script.

```
# Check if we have vision data to make decisions with
if Rover.nav_angles is not None:
        # Check for Rover.mode status
    if Rover.mode == 'forward':
        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), -15, 15)
        # 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 = -15 # 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), -15, 15)
                Rover.mode = 'forward'
# Just to make the rover do something
else:
    Rover.throttle = Rover.throttle_set
```

Results & Improvements

The autonomous run mapped atleast 40% of the world with more than 73% fidelity while finding rock samples where ever they appeared. Screen Resolution : 1024 x 768. Graphics Quality : Good.

One of the major factors that improved map fidelity was thresholding the x and y pixels in the rover coordinates function. The rover read pixels closer to the camera and ignored farther away points to determine navigable terrain and obstacles. The fidelity score jumped from 58% to an average of 75% in multiple runs.

Another factor that contributed to higher fidelity was using more precise RGB values for color thresholding. Specifically obstacle thresholding was modified with better RGB values to make identification better.

Future Work

There are multiple areas that I would like to work on, in the future, to get improved results. Some of these are:

1. Thresholding applied to Roll and Pitch angle to reduce erroneous pixel reading and identification. Updating the map only when the roll and pitch values are below a threshold.
2. Implementing “wall-hugging” approach by offsetting the mean nav angle for better mapping of the world possibly resulting in higher fidelity score.
3. Reducing the chances of remapping already mapped areas/pixels. This will result in faster run time.
4. Implementing “stuck mode” for better navigation around some obstacles in the map.
5. Better stopping criteria to eliminate the jerk stopping mode.
6. Implement the pick and place functionality