CSE483 Computer Vision

Phase 1 Project Report

Group No. 3

19P4442	AbdulRaouf Monir Kamal Mahmoud
19P1250	Adham Ahmed Mohamed Mohamed Abdelmaksoud
19P1609	Osama Ayman Mokhtar Amin

```python
# Identify pixels above the threshold
# Threshold of RGB > 160 does a nice job of identifying ground pixels only
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
```

color_thresh method is used to detect objects according to the object's color.

For example:

(160, 160, 160) threshold will output true in case of the current value of the pixel is greater than 160 in all 3 channels. This value can be changed to detect different objects such as rocks.

```python
def detect_rocks(img, rgb_thresh=(110, 110, 60)):
    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
```

We detected rocks using RGB threshold (110, 110, 60).

```python
# 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
```

rover_coords method accept image coordinates as a parameter and returns the corresponding rover coordinates.

```python
# Define a function to convert to radial coords in rover space
def to_polar_coords(x_pixel, y_pixel):
    # Convert (x_pixel, y_pixel) to (distance, angle)
    # in polar coordinates in rover space
    # Calculate distance to each pixel
    dist = np.sqrt(x_pixel**2 + y_pixel**2)
    # Calculate angle away from vertical for each pixel
    angles = np.arctan2(y_pixel, x_pixel)
    return dist, angles
```

to_polar_coords method returns the given x, y position in polar form (distance & angle).

```python
# Define a function to map rover space pixels to world space
def rotate_pix(xpix, ypix, yaw):
    # Convert yaw to radians
    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 the result
    return xpix_rotated, ypix_rotated
```

Each pixel taken from rover camera should be rotated to be aligned with the world map, so we use this function.

```python
def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # Apply a scaling and a translation
    xpix_translated = (xpix_rot / scale) + xpos
    ypix_translated = (ypix_rot / scale) + ypos
    # Return the result
    return xpix_translated, ypix_translated
```

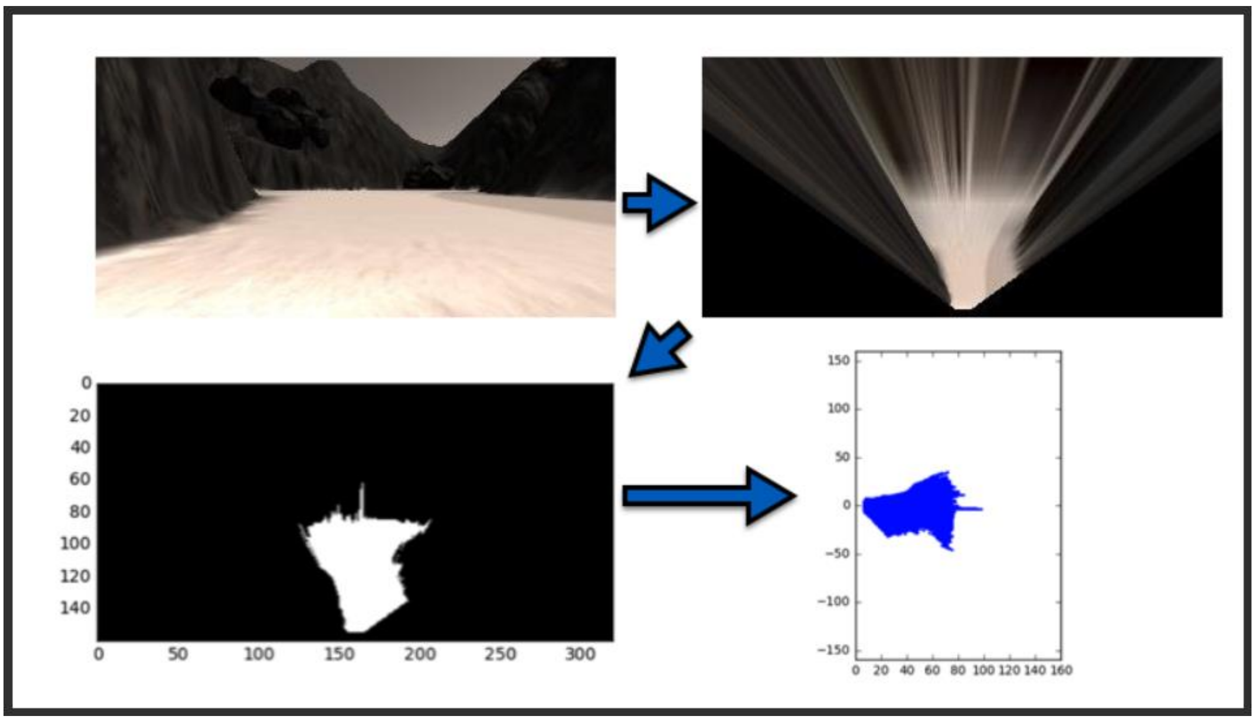
After each pixel is rotated it should be scaled and translated to its new location.

```python
# 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
```

This method calls the above 2 methods to plot the pixel in the world map.

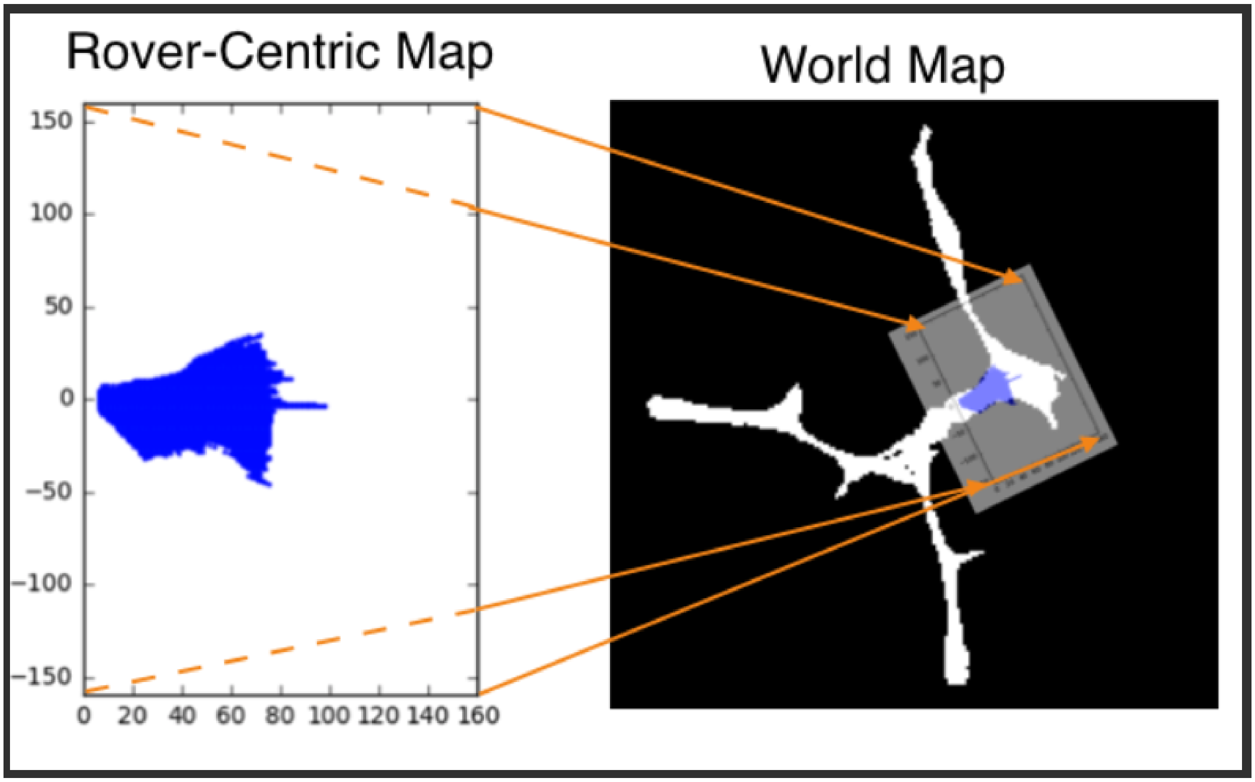
```python
# Define a function to perform a perspective transform
def perspect_transform(img, src, dst):

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

    return warped, mask
```

This method is used to to transform the perspective to bird eye view for processing.



This image illustrates the steps of exploring the map.

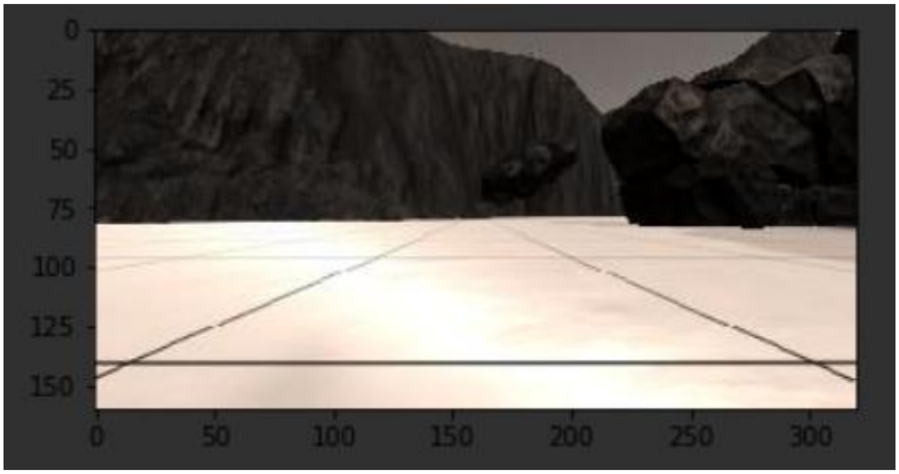
1. The camera mounted on the rover capture an image.
2. This image is transformed to bird eye view perspective.
3. Threshold is applied to focus on important details only.
4. Resulting image is rotated to world coordinates so, it can be mapped properly.



This image shows the mapping between Rover-Centric and World Maps.

```python
# Apply the above functions in succession and update the Rover state accordingly
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
    dst = 5
    bottom_offset = 6
    source = np.float32([[14, 140],
                         [300, 140],
                         [200, 95],
                         [120, 95]])

    destination = np.float32([[Rover.img.shape[1] / 2 - dst, Rover.img.shape[0] - bottom_offset],
                              [Rover.img.shape[1] / 2 + dst, Rover.img.shape[0] - bottom_offset],
                              [Rover.img.shape[1] / 2 + dst, Rover.img.shape[0] - 2*dst - bottom_offset],
                              [Rover.img.shape[1] / 2 - dst, Rover.img.shape[0] - 2*dst - bottom_offset]])
```

Here we define the source and destination points for perspective transform.



It can be seen that the coordinates of the middle grid of the source image are around (14,140), (300,140), (200,95) & (120,95).

To get the x-coordinates destination points, we divide the rover image by 2 and then add/subtract dst (which is half the width of the image).

And for the y-coordinates, we subtract bottom offset to make the image move up, so its borders are visible.

```
    # 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, rgb_thresh=(160, 160, 160))
    tmp_threshed = color_thresh(warped, rgb_thresh=(120, 120, 120))
    obstacles_thresh = (np.ones_like(threshed)-tmp_threshed)*mask
    rocks_thresh = detect_rocks(warped)
```

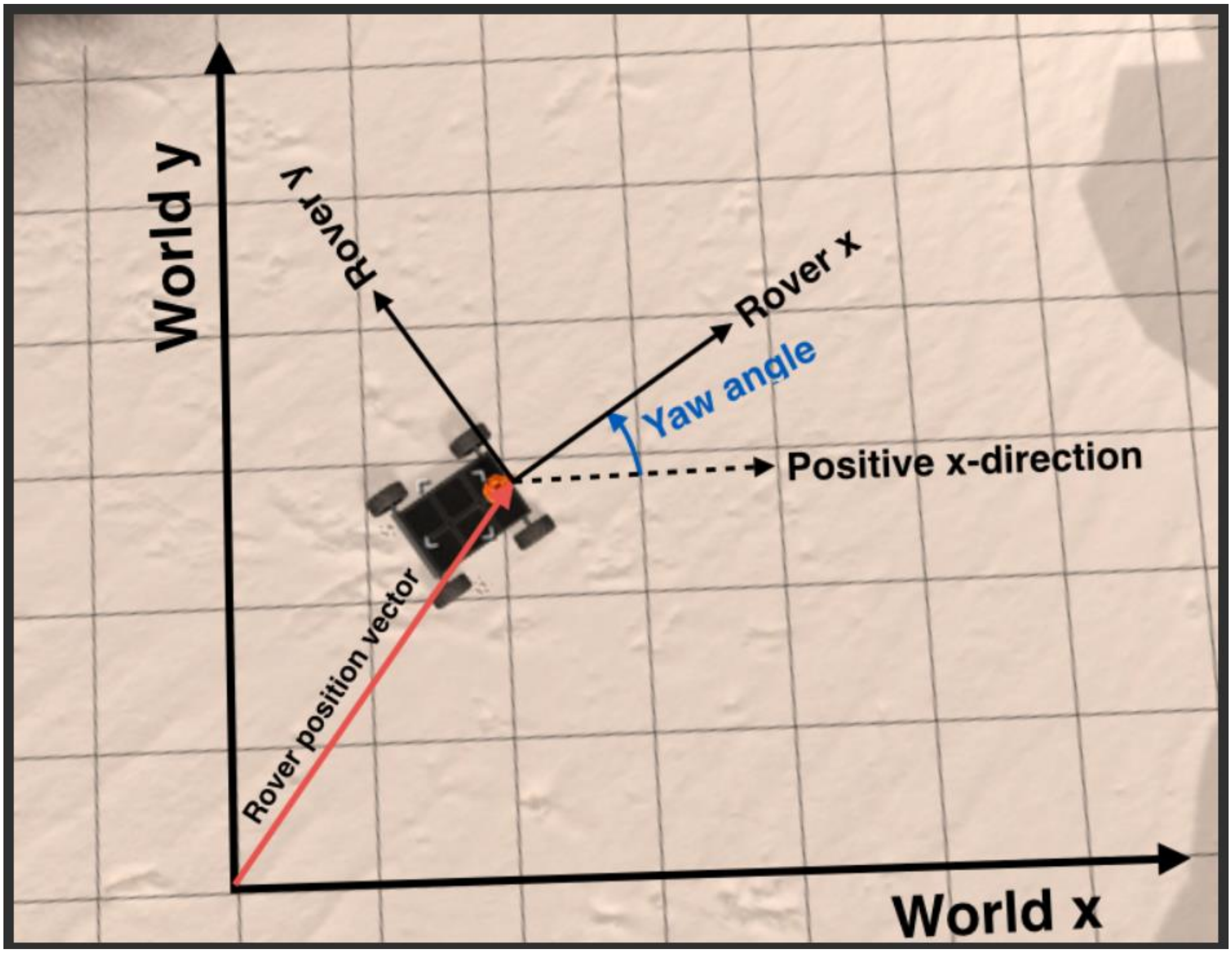
After that we apply perspective transform and threshold to identify navigable terrain, obstacles and rock samples. Images are produced in gray scale value after applying the color threshold.

```
    # 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[:, :, 0] = obstacles_thresh*255
    Rover.vision_image[:, :, 1] = rocks_thresh*255
    Rover.vision_image[:, :, 2] = threshed*255
```

Here we update the vison image that will be displayed on the left side of screen.

```
    # 5) Convert map image pixel values to rover-centric coords
    xpix, ypix = rover_coords(threshed)
    xobstacles, yobstacles = rover_coords(obstacles_thresh)
    xrocks, yrocks = rover_coords(rocks_thresh)

    # 6) Convert rover-centric pixel values to world coordinates
    x_pix_world, y_pix_world = pix_to_world(xpix,
                                            ypix,
                                            Rover.pos[0],
                                            Rover.pos[1],
                                            Rover.yaw,
                                            Rover.worldmap.shape[0],
                                            2*dst)
    obstacle_x_world, obstacle_y_world = pix_to_world(xobstacles,
                                                      yobstacles,
                                                      Rover.pos[0],
                                                      Rover.pos[1],
                                                      Rover.yaw,
                                                      Rover.worldmap.shape[0],
                                                      2*dst)
    rock_x_world, rock_y_world = pix_to_world(xrocks,
                                              yrocks,
                                              Rover.pos[0],
                                              Rover.pos[1],
                                              Rover.yaw,
                                              Rover.worldmap.shape[0],
                                              2*dst)
```

Then, we convert the image pixels to rover's coordinates. After that, it is converted to world coordinates in order to map it to the world map.



This explain the difference between the coordinates of the rover and those of the world.

```
# 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[obstacle_y_world, obstacle_x_world, 0] += 1
Rover.worldmap[rock_y_world, rock_x_world, 1] += 10
Rover.worldmap[y_pix_world, x_pix_world, 2] += 10

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

```

Then, we update rover world map that is displayed on right side of screen.

Finally, we get the polar coordinates of the rover and set the rover’s distance and angle in order to move in the right direction and continue exploring the map according to the decision code.

```python
    debugging_mode = True

    if debugging_mode:
        plt.figure(1, figsize=(12,9))
        plt.clf()
        plt.subplot(221)
        plt.imshow(Rover.img)
        plt.subplot(222)
        plt.imshow(warped)
        plt.subplot(223)
        plt.imshow(threshed, cmap='gray')
        plt.subplot(224)
        plt.plot(xpix, ypix, '.')
        plt.ylim(-160, 160)
        plt.xlim(0, 160)
        arrow_length = 100
        mean_dir = np.mean(Rover.nav_angles)
        x_arrow = arrow_length * np.cos(mean_dir)
        y_arrow = arrow_length * np.sin(mean_dir)
        plt.arrow(0, 0, x_arrow, y_arrow, color='red', zorder=2, head_width=10, width=2)
        plt.pause(1)

    # if rocks_thresh.any():
    #     Rover.nav_dists, Rover.nav_angles = to_polar_coords(xrocks, yrocks)

    return Rover
```

For the debugging mode, we set a flag that when true, all images that are currently being processed are showed to the user on the screen. All steps discussed in this section are displayed.

We calculate the mean direction and draw a red arrow that points to the direction that the rover will continue exploring at.