Project: Search and Sample Return

The goals of the project are to drive the car autonomously and pick up a rock where is in close proximity.

Training / Calibration

Rover_Project_Test_Notebook.ipynb is used to train and calibrate functions.

Fig. 1 shows calibrated images with grid and with a rock.

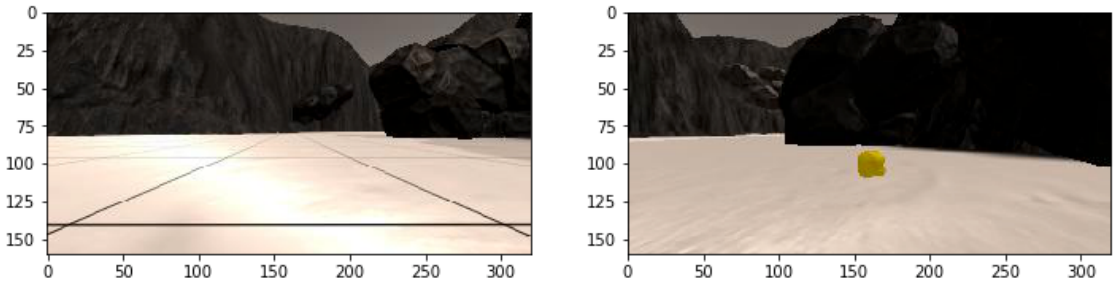


Fig. 1 Screenshots of example images

We use perspective transform from OpenCV to get a bird's eye view for navigation. Fig. 2 shows a warped image along with a mask. Later, we use the mask image to mask out invalid area from obstacles.

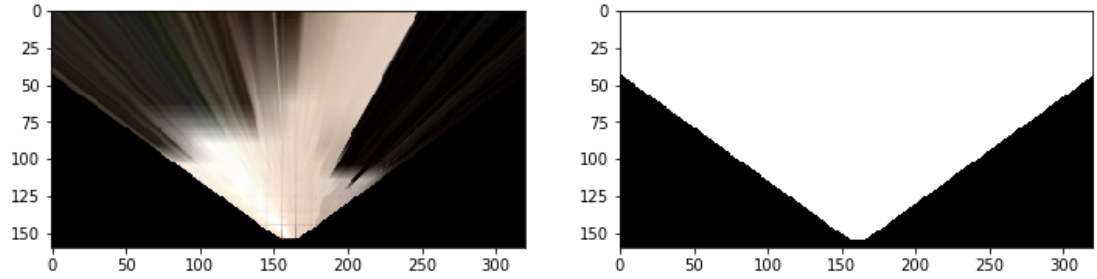


Fig. 2 Warped view and mask after perspective transformation

To look for navigable terrain, we use RGB color thresholds R > 160, G > 160, and B > 160 on the warped image.

```
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
```

For obstacles, we can simply invert warped image and apply the mask.

```
        obstacle = np.absolute(np.float32(warped) - 1) * mask
```

As rock is in yellow color, we can use color threshold with more red (>100) and green (>100), less blue (< 50) to look for rocks.

```
def find_rock(img, rgb_thresh=(100, 100, 50)):
    # 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 between all three threshold values in RGB
    # threshholds 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
```

Process_image() is the primary function to process images from stored images or Robot's camera

1) Define source and destination points for perspective transform

Destination size (dst_size) is set to 5.
A grid with 10x10 pixels represents one square meter.
Bottom offset (bottom_offset) is set to six.

```
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

First, we get the translation matrix from OpenCV function getPerspectiveTransform() by passing source and destination points. Then applying the matrix to another OpenCV function warpPerspective() gets a warped image. When an all one image is transformed, we have a mask.

```
def perspect_transform(img, src, dst)
    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))
    mask = cv2.warpPerspective(np.ones_like(img[:,:,0]), M, (img.shape[1], img.shape[0]))
```

return warped, mask

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

Applying color_thresh() on a warped image, it gives a navigation terrain.
By inverting the terrain and applying mask, we have an obstacle map.
Applying find_rock(), it gives us the rock image.

4) Convert thresholded image pixel values to rover-centric coords

Function rover_coords() converts an image to river-centric coordinates.

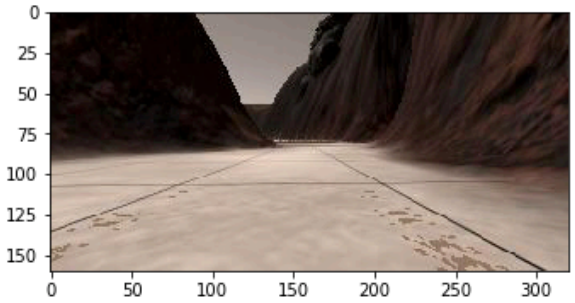
```
def rover_coords(binary_img):
    # Identify nonzero pixels
    ypos, xpos = binary_img.nonzero()
    # Calculate pixel positions with reference to the rover position being at the
    # center bottom of the image.
    x_pixel = -(ypos - binary_img.shape[0]).astype(np.float)
    y_pixel = -(xpos -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

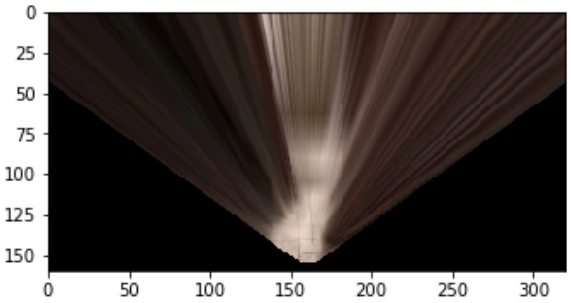
5) Convert rover-centric pixel values to world coords

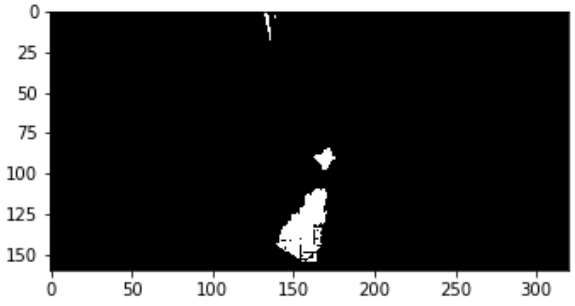
To covert from a rover coordinates to world coordinates. we apply rotation with yaw angle and translation of (xpos, pos) with scale. Then we clip the world coordinates to the world size before returning it.

```
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
```

We update world-map displayed on right side of screen.







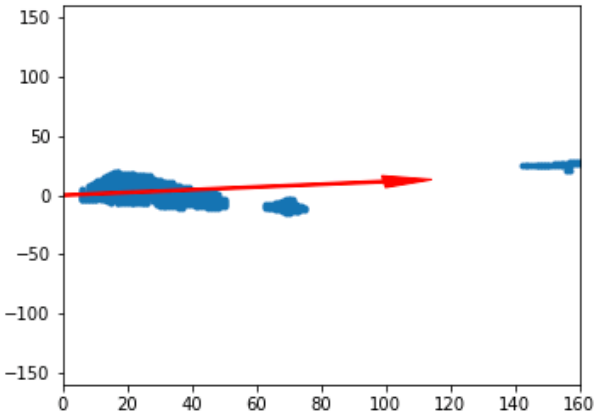


Fig. 3 shows an image, warped image, color thresh-hold, and an overlay with an arrow on rover coordinates.

Movie of processing recorded images is shown in the cell # 12.

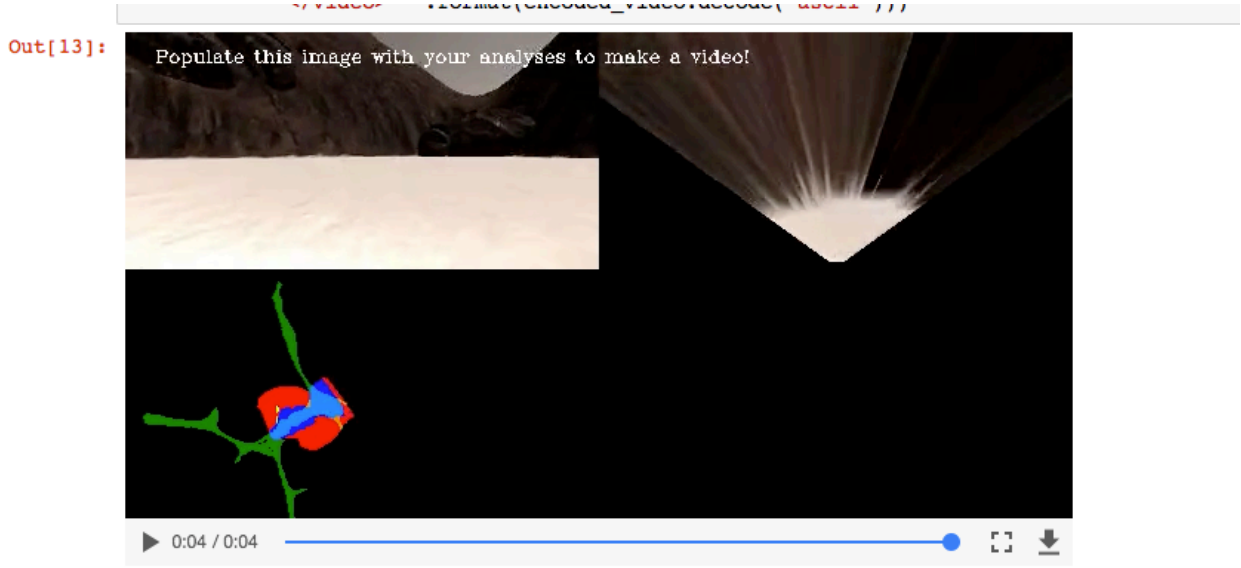


Fig. 4 screen shot of movie

Autonomous Navigation / Mapping

Autonomous navigation applies technics mentioned in process_image() to the perception_step() function of perception.py script.

Rover.vision_image updates camera with a perspective view on the left.
Rover.worldmap shows the world map from top view.
Scale is set to 2*dst_size.
World size is set Rover.worldmap.shape[0] to clip boundary of the world map.

When there is at least a rock found, we calculate the nearest one and update vision map and world map. As our goal is to pick up as many rocks as possible, we want to navigate to the rock. We save the rock angle in Rover.nearest_rock_angle. The checking of thresholds of last rock position and rock polar coordinates are to avoid sticking to a rock.

```
# go after rock
Rover.nearest_rock_angle = None
if np.fabs(rock_ang[rock_idx]) > 0.2 and \
   np.fabs(rock_dist[rock_idx]) > 6.5 and \
   np.fabs(Rover.rock_picked_pos[0] - rock_xcen) > 8.0 and \
   np.fabs(Rover.rock_picked_pos[1] - rock_ycen) > 8.0:
   Rover.nearest_rock_angle = rock_ang[rock_idx]
```

The `decision_step()` function within the `decision.py` script takes into consideration the outputs of the `perception_step()` in deciding how to issue throttle, brake and steering commands.

When Rover is in the forward mode, it tries to drive toward the nearest rock if it found one and hasn't picked up yet. If no rock is close, it drives in the middle of navigation terrain when number of Rover.nav_angles is greater than Rover.stop_forward. Otherwise, it stops the Rover.

If we are in the stop mode with velocity greater than 0.2, we apply break to stop the Rover. When we are stopped, we check to see whether there is a path to go. If there is a sufficient space to navigate, we move the Rover again.

Finally, we need to pick up a rock if there is one. When a Rover is in the near sample state, we check velocity to slow it down if necessary. When a rock is picked up. We reset Rover.nearest_rock_angle and save world coordinates of the Rover position to Rover.rock_picked_pos. We need the position to prevent Rover being recursively circling a rock.

The test has been performed with screen resolution 1024x768 and quality good.

Future Work

Sometimes, Rover gets stuck in a rock and takes long time to get out. We need GPS guide to move more efficiently.