Robotics Nanodegree

Search & Sample Return Project

This project is modeled after the NASA sample return challenge and it will give you first hand experience with the three essential elements of robotics, which are perception, decision making and actuation. You will carry out this project in a simulator environment built with the Unity game engine.

If you need further information, assistance or referral about a project issue, please contact kiang.ng@hotmail.com.

UDACITY

Ng Fang Kiang
Kiang.ng@hotmail.com
Instagram.com/jorcus96
linkedin.com/in/jorcus
fb.com/ngfangkiang
github.com/jorcus

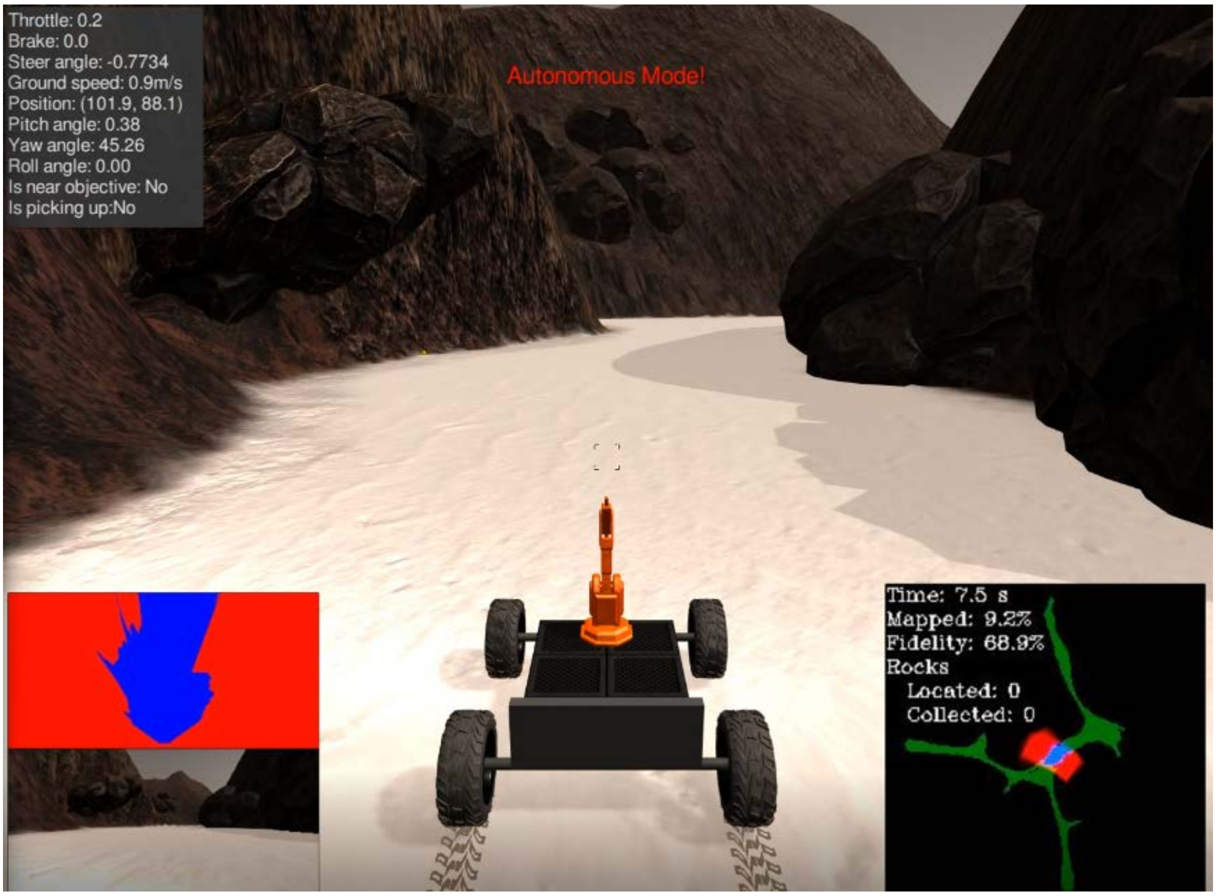
Ng Fang Kiang

Project 1: Follow Me

Prepared/Updated : 11 Nov 2017

Project: Search and Sample Return

This project is modeled after the NASA sample return challenge and it will give you first hand experience with the three essential elements of robotics, which are perception, decision making and actuation. You will carry out this project in a simulator environment built with the Unity game engine.



The goals / steps of this project are the following:

Training / Calibration

- Download the simulator and take data in "Training Mode"
- Test out the functions in the Jupyter Notebook provided
- Add functions to detect obstacles and samples of interest (golden rocks)
- Fill in the process_image() function with the appropriate image processing steps (perspective transform, color threshold etc.) to get from raw images to a map. The output_image you create in this step should demonstrate that your mapping pipeline works.
- Use moviepy to process the images in your saved dataset with the process_image() function. Include the video you produce as part of your submission.

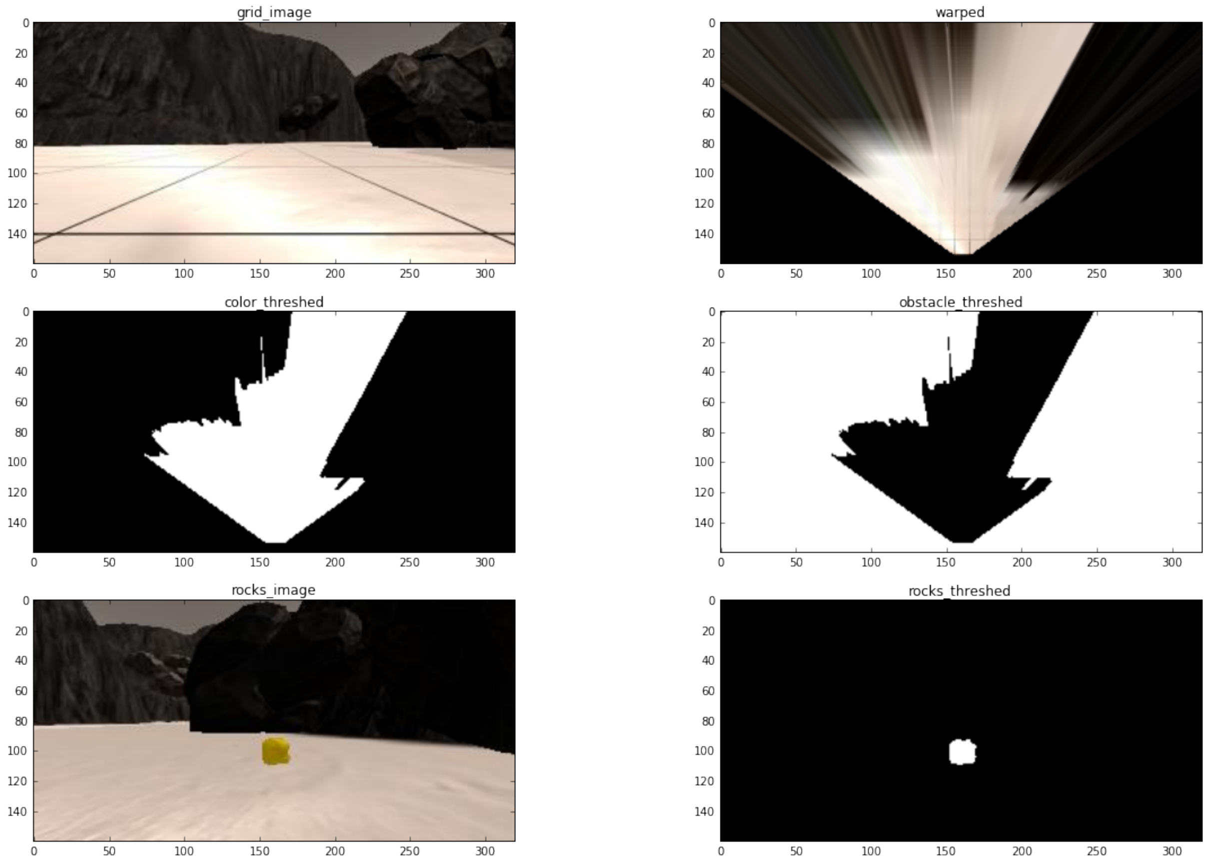
Autonomous Navigation / Mapping

- Fill in the perception_step() function within the perception.py script with the appropriate image processing functions to create a map and update Rover() data (similar to what you did with process_image() in the notebook).
- Fill in the decision_step() function within the decision.py script with conditional statements that take into consideration the outputs of the perception_step() in deciding how to issue throttle, brake and steering commands.
- Iterate on your perception and decision function until your rover does a reasonable (need to define metric) job of navigating and mapping.

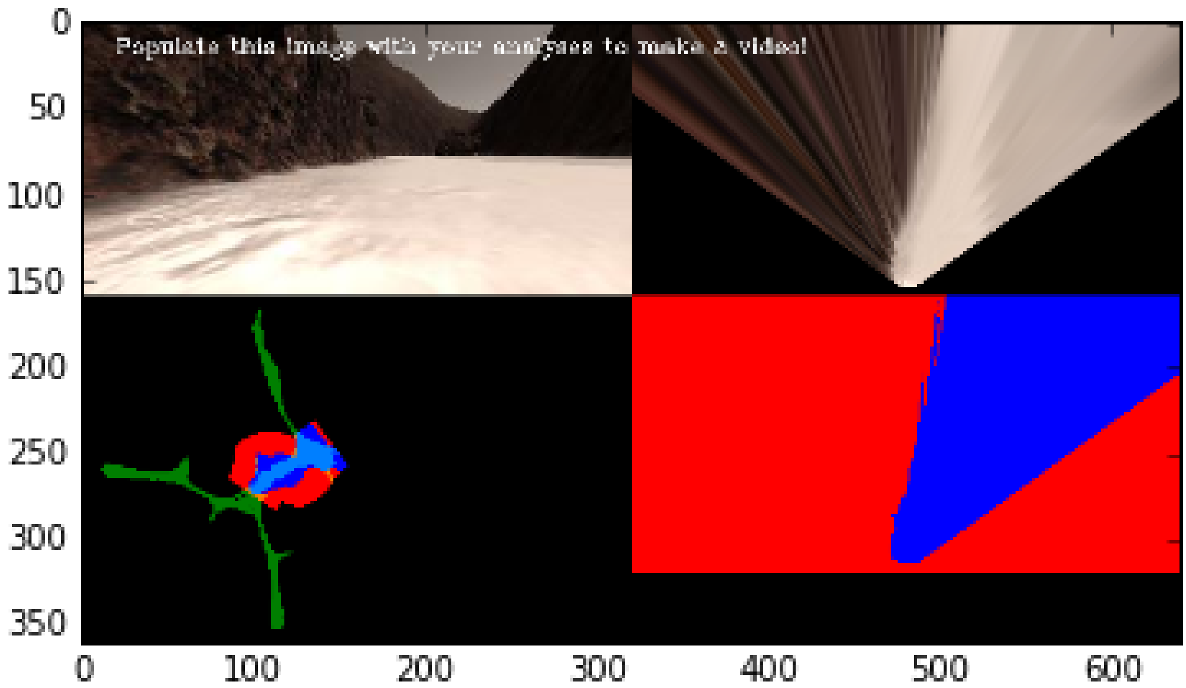
Notebook Analysis

1. Run the functions provided in the notebook on test images (first with the test data provided, next on data you have recorded). Add/modify functions to allow for color selection of obstacles and rock samples.

To find the rock samples, I created color_threshed for navigation, obstacle_threshed for detection of obstacle and rocks_threshed for identifying the rock sample.



Populate the process_image() function with the appropriate analysis steps to map pixels identifying navigable terrain, obstacles and rock samples into a worldmap. Run process_image() on your test data using the moviepyfunctions provided to create video output of your result.



Autonomous Navigation and Mapping

1. Fill in the perception_step() (at the bottom of the perception.py script) and decision_step() (in decision.py) functions in the autonomous mapping scripts and an explanation is provided in the writeup of how and why these functions were modified as they were.

```
# Apply the above functions in succession and update the Rover state accordingly
def perception_step(Rover):
    # Example of how to use the Databucket() object defined above
    # to print the current x, y and yaw values
    # print(data.xpos[data.count], data.ypos[data.count], data.yaw[data.count])

    img = Rover.img
    dst_size = 5
    bottom_offset = 6
    # 1) Define source and destination points for perspective transform
    source = np.float32([[14, 140], [301,140], [200, 96], [118, 96]])
    destination = np.float32([[img.shape[1]/2 - dst_size, img.shape[0] - bottom_offset],
                  [img.shape[1]/2 + dst_size, img.shape[0] - bottom_offset],
                  [img.shape[1]/2 + dst_size, img.shape[0] - 2*dst_size - bottom_offset],
                  [img.shape[1]/2 - dst_size, img.shape[0] - 2*dst_size - bottom_offset],
                  ])

    # 2) Apply perspective transform
    warped = perspect_transform(img, source, destination)

    # 3) Apply color threshold to identify navigable terrain/obstacles/rock samples
    color_threshed = color_thresh(warped)
    obstacle_threshed = obstacle_thresh(warped)
    rocks_threshed = rocks_thresh(warped)
    rocks_threshed = threshold_dilated(rocks_threshed, 5)

    Rover.vision_image[:, :, 2] = color_threshed * 255
    Rover.vision_image[:, :, 0] = obstacle_threshed * 255

    # 4) Convert thresholded image pixel values to rover-centric coords
    nav_xpix, nav_ypix = rover_coords(color_threshed)
    obs_xpix, obs_ypix = rover_coords(obstacle_threshed)

    # 5) Convert rover-centric pixel values to world coords
    dist, angles = to_polar_coords(nav_xpix, nav_ypix)
    mean_dir = np.mean(angles)
    xpos = Rover.pos[0]
    ypos = Rover.pos[1]
    yaw = Rover.yaw
    world_shape = Rover.worldmap.shape[0]
    scale = 2 * dst_size

    # 6) Update worldmap (to be displayed on right side of screen)
    nav_x_world, nav_y_world = pix_to_world(nav_xpix, nav_ypix, xpos, ypos, yaw, world_shape, scale)
    obs_x, obs_y = pix_to_world(obs_xpix, obs_ypix, xpos, ypos, yaw, world_shape, scale)

    Rover.worldmap[nav_y_world, nav_x_world, 2] += 10
    Rover.worldmap[obs_y, obs_x, 0] += 1

    Rover.nav_angles = angles
    Rover.nav_dists = dist
    Rover.rock_nav_angles = None
    Rover.rock_nav_dists = None
    if rocks_threshed.any():
        rock_xpix, rock_ypix = rover_coords(rocks_threshed)
        rock_x, rock_y = pix_to_world(rock_xpix, rock_ypix, xpos, ypos, yaw, world_shape, scale)
        rock_dist, rock_ang = to_polar_coords(rock_x, rock_y)
        rock_dist2, rock_ang2 = to_polar_coords(rock_xpix, rock_ypix)
        rock_idx = np.argmin(rock_dist)

        if not isinstance(rock_x, np.ndarray):
            rock_x = [rock_x]
            rock_y = [rock_y]

        rock_xcen = rock_x[rock_idx]
        rock_ycen = rock_y[rock_idx]
        Rover.worldmap[rock_ycen, rock_xcen, :] = 255
        Rover.vision_image[:, :, 1] = rocks_threshed * 255

        Rover.rock_found = True
        Rover.rock_nav_angles = rock_ang2
        Rover.rock_nav_dists = rock_dist2
    else:
        Rover.rock_found = False
        Rover.vision_image[:, :, 1] = 0

    return Rover
```

```python
def decision_step(Rover):
    ...
    if Rover.nav_angles is not None:
        # Check for Rover.mode status

        if Rover.mode == 'stuck':
            flip_coin = random.randint(0, 1)
            Rover.throttle = 0
            Rover.brake = 0

            if flip_coin == 0:
                if np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15) > 0:
                    Rover.steer = 15
                else:
                    Rover.steer = -15
            else:
                Rover.throttle = 1.0
                Rover.unstuck_turningfrequency = 0
                Rover.mode = 'forward'

        elif Rover.mode == 'forward':
            # Check the extent of navigable terrain

            if Rover.near_sample:
                Rover.brake = Rover.brake_set
                Rover.throttle = 0
                Rover.steer = 0
                Rover.mode = 'stop'

            elif len(Rover.nav_angles) >= Rover.stop_forward:
                # If mode is forward, navigable terrain looks good
                # and velocity is below max, then throttle

                if Rover.stuck is True:
                    Rover.brake = 0
                    Rover.throttle = 0
                    Rover.mode = 'stuck'

                elif 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

                if Rover.rock_found is True and len(Rover.rock_nav_angles) > 1:
                    Rover.steer = np.clip(np.mean(Rover.rock_nav_angles * 180/np.pi), -15, 15)
                else:
                    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

                    if Rover.near_sample:
                        Rover.steer = 0
                    else:
                        # 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 angles
                    Rover.steer = np.clip(np.mean(Rover.nav_angles * 180/np.pi), -15, 15)
                    Rover.mode = 'forward'

    # Just to make the rover do something
    # even if no modifications have been made to the code
    else:
        Rover.throttle = Rover.throttle_set
        Rover.steer = 0
        Rover.brake = 0

    # If in a state where want to pickup a rock send pickup command
    if Rover.near_sample and Rover.vel == 0 and not Rover.picking_up:
        Rover.send_pickup = True
        Rover.mode = 'forward'

    return Rover
```

2. Launching in autonomous mode your rover can navigate and map autonomously. Explain your results and how you might improve them in your writeup.

Note: running the simulator with different choices of resolution and graphics quality may produce different results, particularly on different machines! Make a note of your simulator settings (resolution and graphics quality set on launch) and frames per second (FPS output to terminal by drive_rover.py) in your writeup when you submit the project so your reviewer can reproduce your results.

The rover is running well in the simulations with average 60 FPS but not perfect.

The Failure

1. The rover is having difficulty on navigating around with the big stones and small stones. It makes the rover keep swinging left and right.

2. I've already implement the unstuck scenario, but its still possible get stuck.

Future Enhancement

Nothing is perfect, there's always lots of fun works to improve.

1. Covering all the maps
2. Using deep learning for stone detection to know better about the environment.
3. Develop an algorithm to discover the unknown places.