Project: Search and Sample Return

Robotics Software Engineer Nanodegree

[Rubric](https://review.udacity.com/#!/rubrics/916/view) Points

Required files for project submission:

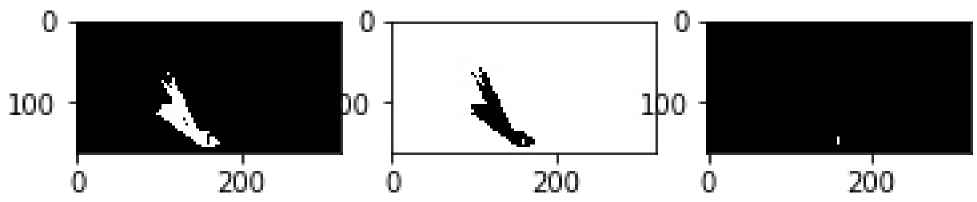
- Jupyter Notebook with test code :
- Autonomous navigation scripts :
- Test output video :

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.

- Modifying the `color_thresh()` function to return thresholds for specified `type_`. You can see that I'm using the `mode` parameter to build specific tresholds depending on the type we want to get - ground, obstacle and sample. The tresholds are gathered depending on the color range that is present on the pixel.

End result - Color Thresholding



2. 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 `moviepy` functions provided to create video output of your result.

As suggested in the project FAQ the final movie file was created using screen capture.

`perception_step()` function:

- A variables that hold transform source and destination values: `python source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]]) destination = np.float32([[Rover.img.shape[1]/2 - dst_size, Rover.img.shape[0] - bottom_offset], [Rover.img.shape[1]/2 + dst_size, Rover.img.shape[0] - bottom_offset], [Rover.img.shape[1]/2 + dst_size, Rover.img.shape[0] - 2*dst_size - bottom_offset], [Rover.img.shape[1]/2 - dst_size, Rover.img.shape[0] - 2*dst_size - bottom_offset],])`
- A perspective transform: `python warped = perspect_transform(Rover.img, source, destination)`
- Thresholds for ground, obstacles and samples: `python ground_thresh = color_thresh(warped, type_="GROUND") obstacle_thresh = color_thresh(warped, type_="OBSTACLE") sample_thresh = color_thresh(warped,type_="SAMPLE",low_thresh=sample_low_thresh, high_thresh=sample_high_thresh)`
- Getting rover-centric coords: `python ground_x, ground_y = rover_coords(ground_thresh) obstacle_x, obstacle_y = rover_coords(obstacle_thresh) sample_x, sample_y = rover_coords(sample_thresh)`
- Getting world coordinates: `python w_ground_x, w_ground_y = pix_to_world(ground_x, ground_y, Rover.pos[0], Rover.pos[1], Rover.yaw, Rover.worldmap.shape[0], map_scale) w_obstacle_x, w_obstacle_y = pix_to_world(obstacle_x, obstacle_y, Rover.pos[0], Rover.pos[1], Rover.yaw, Rover.worldmap.shape[0], map_scale) w_sample_x, w_sample_y = pix_to_world(sample_x, sample_y, Rover.pos[0], Rover.pos[1], Rover.yaw, Rover.worldmap.shape[0], map_scale)`
- And finally create polar coordinates: `python polar = to_polar_coords(ground_x, ground_y) Rover.nav_dists = polar[0] Rover.nav_angles = polar[1]`

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.

`perception_step()` function:

- Same as mention above except an additional step as mentioned below
- Additional step Update worldmap **if** pitch and roll are close to 0, to increase map fidelity:

```
if Rover.pitch < Rover.max_pitch and Rover.roll < Rover.max_roll:
```

```
        Rover.worldmap[w_obstacle_y, w_obstacle_x, 0] += 1
        Rover.worldmap[w_sample_y, w_sample_x, 1] += 1
        Rover.worldmap[w_ground_y, w_ground_x, 2] += 1
```

This is how I modified the `decision_step()`:

- Check if the robot is stuck or not; If it is stuck for a longer time, it should rotate a bit to the right. Then depending on the robot mode it is performing certain tasks.

```
def decision_step(Rover):
 if Rover.p_pos == None:
     Rover.p_pos = Rover.pos
 else:
     if Rover.p_pos != Rover.pos:
         Rover.stop_time = Rover.total_time

 if Rover.total_time - Rover.stop_time > Rover.max_time:
     Rover.throttle = 0
     Rover.brake = 0
     Rover.steer = -15
     time.sleep(1) # wait for the turn

 if Rover.nav_angles is not None:
     if Rover.mode == 'start':
         initial_setup(Rover)
     if Rover.mode == 'sample':
         recover_sample(Rover, nearest_sample)
     if Rover.mode == 'forward':
         move(Rover)
     if Rover.mode == 'tostop':
         stop(Rover)
     if Rover.mode == 'stop':
         find_and_go(Rover)
 return Rover
```

- When in `start` mode the robot turns in a programmed direction and moves to the wall. When it reaches it, he goes into the `stop` mode.

```
def initial_setup(Rover):
 if 90 < Rover.yaw < 95:
     Rover.throttle = Rover.throttle_set
     Rover.brake = 0
     Rover.steer = 0
     if len(Rover.nav_angles) < Rover.go_forward:
         Rover.mode = 'stop'
 else:
     Rover.brake = 0
     Rover.throttle = 0
```

```
     if 90 > Rover.yaw or Rover.yaw >= 270:
         Rover.steer = 10
     else:
         Rover.steer = -10
```

- When in `stop` mode, the robot enters a state, where it looks for possible paths to move to as seen in the `find_and_go()` function. It also checks if a sample is near to pick it up.

```
def find_and_go(Rover):
 if Rover.near_sample and Rover.vel == 0 and not Rover.picking_up:
     Rover.send_pickup = True
 else:
     if len(Rover.nav_angles) < Rover.go_forward:
         Rover.throttle = 0
         Rover.brake = 0
         Rover.steer = -15
     if len(Rover.nav_angles) >= Rover.go_forward:
         Rover.throttle = Rover.throttle_set
         Rover.brake = 0
         Rover.mode = 'forward'
```

- The `forward` mode initializes the wall crawler. It moves next to the wall at a given offset, to accord for the rough terrain near the walls.

```
def move(Rover):
 if Rover.near_sample:
     Rover.mode = 'tostop'

 if len(Rover.nav_angles) >= Rover.stop_forward:
     if Rover.vel < Rover.max_vel:
         Rover.throttle = Rover.throttle_set
     else:
         Rover.throttle = 0
     Rover.brake = 0
     Rover.p_steer = Rover.steer
     Rover.steer = np.max((Rover.nav_angles) * 180 / np.pi) - 30 # minus wall of
 else:
     Rover.mode = 'tostop'
```

- The `stop` mode does exactly that. Stops the rover - to be used for sample picking.

```
def stop(Rover):
 if Rover.vel > 0.2:
     Rover.throttle = 0
```

```
            Rover.brake = Rover.brake_set
            Rover.steer = 0
        elif Rover.vel < 0.2:
            Rover.mode = 'stop'
```

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

The robot maps around ~53% of the map at a fidelity of 70% and higher while locating all samples it sees on the way. The biggest problem right now is to create a mechanism that will move the robot away when it is stuck on a rock.

Simulator settings

Resolution	Graphics quality	FPS
1280x768	Fantastic	60