

Project Search and Sample Return

The goal is to write a code for the rover to be able to map and move in the environment autonomously, detecting obstacles and collect the yellow rock samples.

Tasks and TODOs of the Project:

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.

Rubric Points

Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. I am preparing in pdf.

Notebook Analysis

Jupyter Rover_Project_Test_Notebook Last Checkpoint: 5 minutes ago (autosaved) Logout

File Edit View Insert Cell Kernel Help Trusted Python 3

Rover Project Test Notebook

This notebook contains the functions from the lesson and provides the scaffolding you need to test out your mapping methods. The steps you need to complete in this notebook for the project are the following:

- First just run each of the cells in the notebook, examine the code and the results of each.
- Run the simulator in "Training Mode" and record some data. Note: the simulator may crash if you try to record a large (longer than a few minutes) dataset, but you don't need a ton of data, just some example images to work with.
- Change the data directory path (2 cells below) to be the directory where you saved data.
- Test out the functions provided on your data.
- Write new functions (or modify existing ones) to report and map out detections of obstacles and rock samples (yellow rocks).
- Populate the `process_image()` function with the appropriate steps/functions to go from a raw image to a worldmap.
- Run the cell that calls `process_image()` using `noviipy` functions to create video output.
- Once you have mapping working, move on to modifying `perception.py` and `decision.py` to allow your rover to navigate and map in autonomous mode!

Note: If, at any point, you encounter frozen display windows or other confounding issues, you can always start again with a clean slate by going to the "Kernel" menu above and selecting "Restart & Clear Output".

Run the next cell to get code highlighting in the markdown cells.

```
In [6]: %HTML
<style> code {background-color : orange !important;} </style>
```

```
In [7]: import cv2
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import pandas as pd
from noviipy.editor import ImageSequenceClip
```

Quick Look at the Data

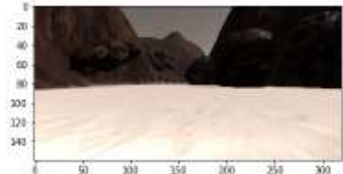
There's some example data provided in the `test_dataset` folder. This basic dataset is enough to get you up and running but if you want to hone your methods more carefully you should record some data of your own to sample various scenarios in the simulator.

Next, read in and display a random image from the `test_dataset` folder.

Perspective Transform

Define the perspective transform function from the lesson and test it on an image.

```
In [8]: path = '../test_dataset/IMG/*'
import glob
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
img_list = glob.glob(path)
# Grab a random image and display it
idx = np.random.randint(0, len(img_list)-1)
image = mpimg.imread(img_list[idx])
plt.imshow(image)
plt.show()
```



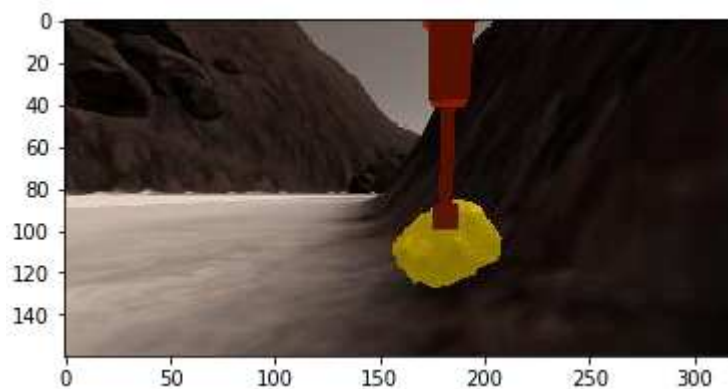
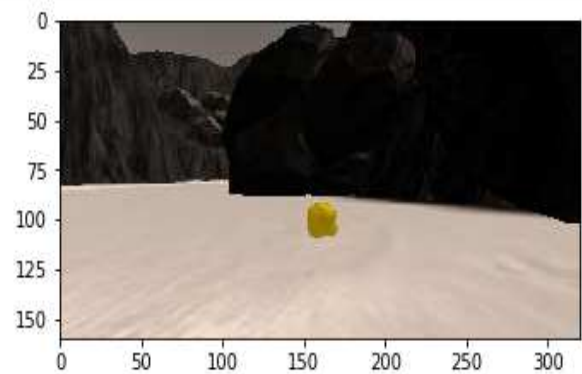
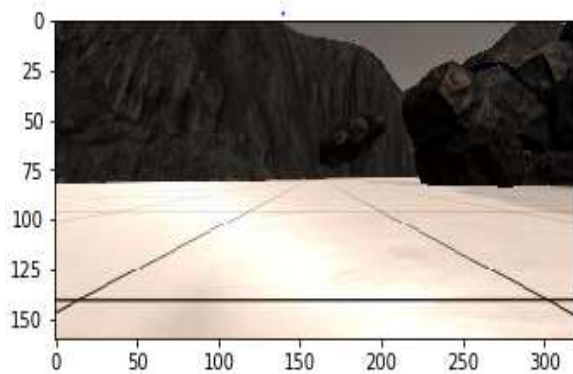
Run the functions provided in the notebook on test images (first with the test data provided, next on data you have recorded). Add/modify functions to allow for color selection of obstacles and rock samples.

The methods were tested in a Jupyter Notebook before applying them in "autonomous mode" in the simulator. The data recorded in "training mode" were uploaded and tested in Jupyter Notebook.

In the notebook the following data and the functions were tested: Perspective Transformation, Calibration Data, Color Thresholding to map and navigate the rover in the terrain, Coordinate Transformations.

The colors were inverted to detect the obstacles and colors of the rocks were imposed to a lower and upper boundary.

```
above_thresh = (img[:,0] > rgb_thresh[0]) \
    & (img[:,1] > rgb_thresh[1]) \
    & (img[:,2] > rgb_thresh[2])
```

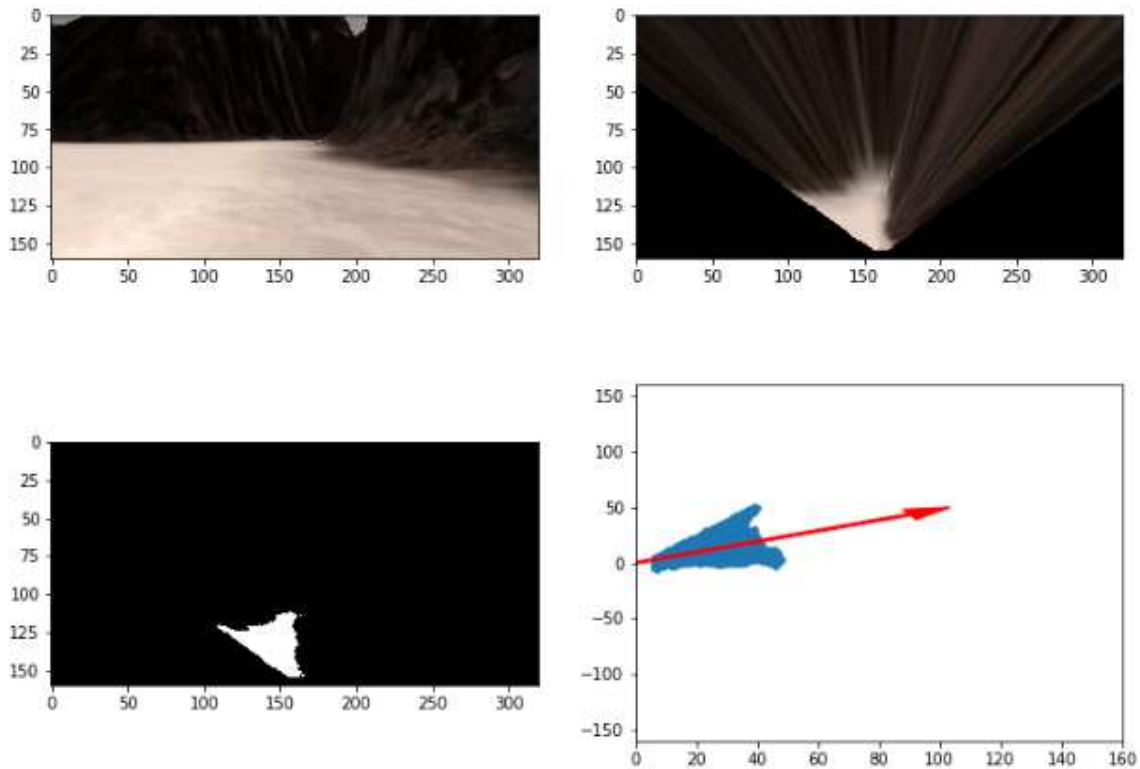


Some tested functions:

Navigable terrain and also obstacles and the positions of the rock samples were mapped. The function was modified to return the pixel locations of obstacles (areas below the threshold) and rock samples (yellow rocks in calibration images). These areas were mapped into world coordinates as well.

```
def obstacle_thresh(img, rgb_thresh=(160, 160, 160)):
    color_select = np.zeros_like(img[:, :, 0])
    thresh = (img[:, :, 0] < rgb_thresh[0]) \
        & (img[:, :, 1] < rgb_thresh[1]) \
        & (img[:, :, 2] < rgb_thresh[2])
    color_select[thresh] = 1
    return color_select
def rock_thresh(img):
    color_select = np.zeros_like(img[:, :, 0])
    threshold_low = (100, 100, 20)
    threshold_high = (255, 255, 30)
    thresh = (img[:, :, 0] > threshold_low[0]) & (img[:, :, 0] < threshold_high[0]) \
        & (img[:, :, 1] > threshold_low[1]) & (img[:, :, 1] < threshold_high[1]) \
        & (img[:, :, 2] > threshold_low[2]) & (img[:, :, 2] < threshold_high[2])
    color_select[thresh] = 1
    return color_select
```

After mapping the terrain, warping and matrix transformation and rotation, the following images were received:



- 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.

The `process_image()` function is used to process the stored images. This function superimposes the map onto the ground truth worldmap. This function was modified by adding perception step process enabling to perform analysis and mapping.

First, we would need the perspective transformation of the image. This data, after converting to world coordinates help to color the map in training mode of the simulator and analyze each image recorded.

The TODOs were:

1. Define source and destination points for perspective transform

```
dst_size = 5
bottom_offset = 6
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 the perspective transformation using the cv2 library functions

```
getPerspectiveTransform and warpPerspective.
```

```
warped = perspect_transform(img, source, destination)
```

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

This allows differentiating between navigable terrain, obstacles and rock samples

```
obstacle_colorsel = color_thresh(warped, rgb_thresh=(40, 40, 40))
rock_colorsel = color_thresh(warped, rgb_thresh=(50, 50, 50))
navigable_colorsel = color_thresh(warped, rgb_thresh=(160, 160, 160))
```

4. Convert thresholded image pixel values to rover-centric cords. This is accomplished by rotating, translating, and finally scaling the image. This is done for each of the three thresholded images.

```
xpix, ypix = rover_coords(navigable_colorsel)
xpix_obstacle, ypix_obstacle = rover_coords(navigable_colorsel)
xpix_rock, ypix_rock = rover_coords(navigable_colorsel)
```

5. Convert rover-centric pixel values to world cords

```
xpix, ypix = pix_to_world(xpix, ypix, data.xpos[data.count-1], data.ypos[data.count-1],
data.yaw[data.count-1], data.worldmap.shape[0], 10)

obstacle_y_world, obstacle_x_world = pix_to_world(xpix_obstacle, ypix_obstacle,
data.xpos[data.count-1], data.ypos[data.count-1], data.yaw[data.count-1],
data.worldmap.shape[0], 10)

rock_y_world, rock_x_world = pix_to_world(xpix_rock, ypix_rock,
data.xpos[data.count-1], data.ypos[data.count-1], data.yaw[data.count-1],
data.worldmap.shape[0], 10)

xpix, ypix = rover_coords(navigable_colorsel)

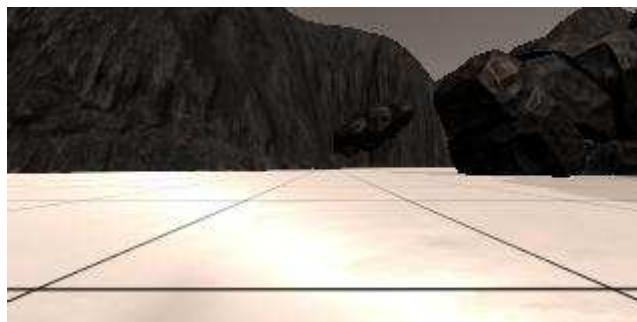
navigable_y_world, navigable_x_world = pix_to_world(xpix, ypix,
data.xpos[data.count-1], data.ypos[data.count-1], data.yaw[data.count-1],
data.worldmap.shape[0], 10)

xpix, ypix = rover_coords(navigable_colorsel)
```

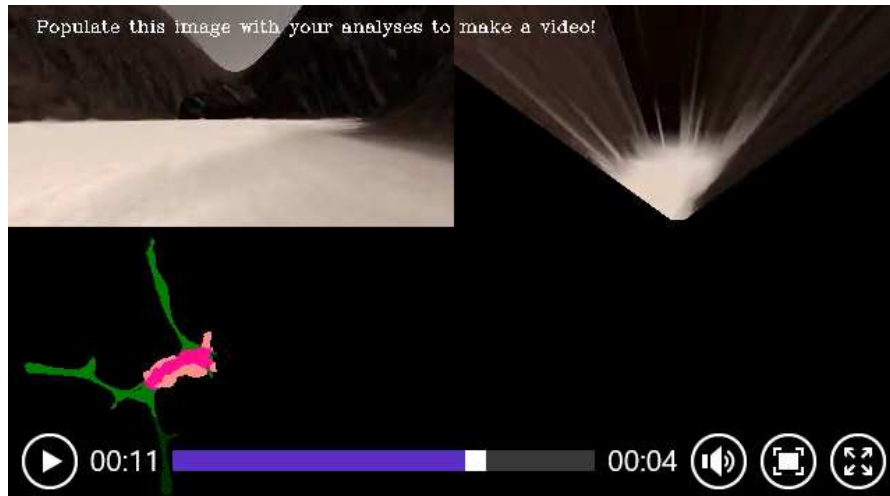
6. Update Rover worldmap (to be displayed on right side of screen)

```
data.worldmap[obstacle_y_world, obstacle_x_world, 0] += 1
data.worldmap[rock_y_world, rock_x_world, 1] += 1
data.worldmap[navigable_y_world, navigable_x_world, 2] += 1
```

7. Make a mosaic image, below is some example code



The output image was stored and used as a frame for the video using **moviepy** library



Autonomous Navigation and Mapping

TODOs:

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.

The `perception.py` script

The `perception_step` function is filled with some new operations. `perception_step()` function works similar as in the notebook for image processing. Instead of interacting with a `Databucket` class, the function interacts with a `Rover` class.

An important thing to note is that the `Rover` angles and distances both for navigable terrain and rocks is updated at the bottom. This information is used by the decision script to know where to move.

The original `color_thresh` function is used only for the terrain coordinates. Other functions were created and used for the rock (`rock_thresh`) and obstacle (`obstacle_thresh`) coordinates.

While the original function checks the above condition, the *obstacle_thresh* checks the below condition, and the *rock_thresh* checks inside a given interval of the *rgb_thresh* values.

In the *perception_step* function the needed steps are processed for the rocks, obstacle and terrain too. In additionally are calculated the distance and angle of the rocks, which are used for rocks detection.

The decision.py script

To the decision module was added the ability of handling rocks. If the rock is not too close, the Rover tries to move slowly, if it is too close, the Rover stops.

The decision tree for the rover mode: forward, stuck and forward were elaborated. It allows to send commands to the rover to change throttle, brake, steering angle and take action. It contained some example conditional statements that demonstrate how you might make decisions about adjusting throttle, brake and steering inputs.

The Rover.nav.angles was updated and Rover now makes more complicated decisions.

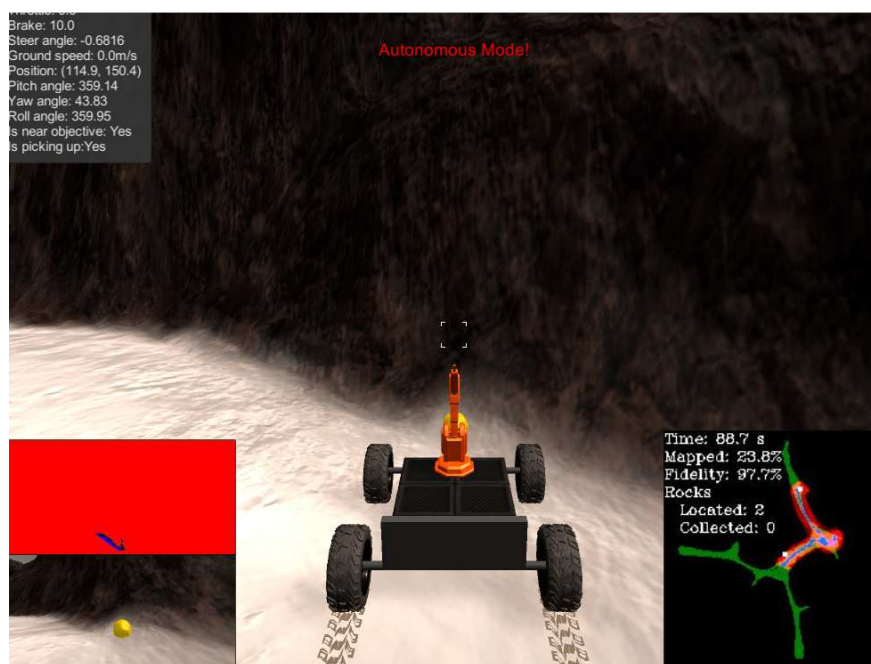
Originally, this script had a forward and a stop mode. I added a stuck mode. When the velocity is low and there is still acceleration, it means that the rover can stuck. When this happens, the rover just steers a bit and tries to move again. When the speed is too low, and gas is not 0.

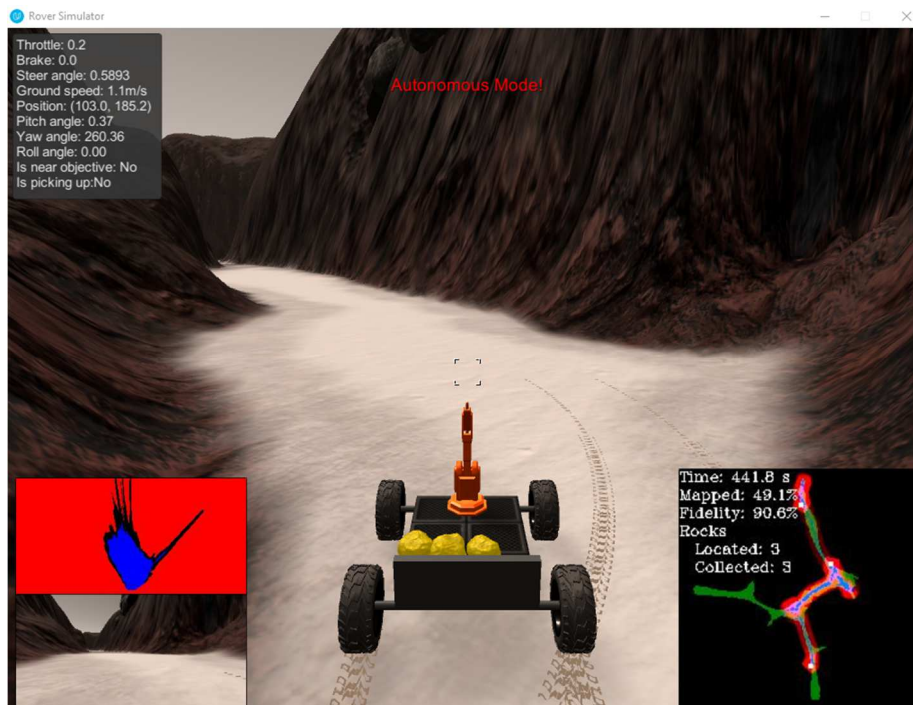
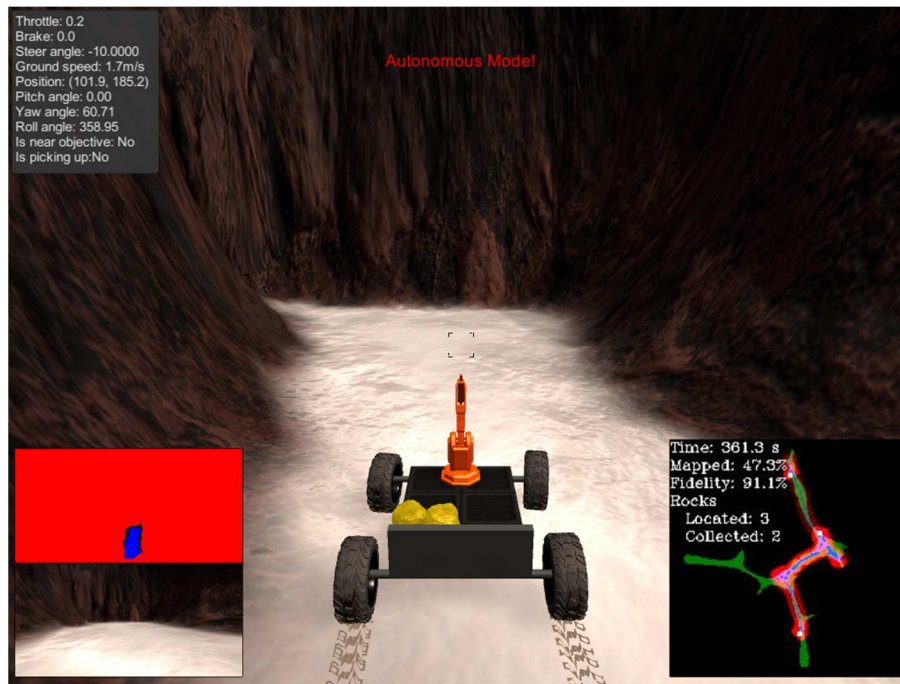
```
Rover.vel < 0.01 and Rover.throttle != 0
the steering angle is -20, and try to move forward
elif Rover.mode == 'stuck':
    # the car has stucked
    Rover.brake = 0
    Rover.throttle = 0
    Rover.steer = -20
    Rover.mode = 'forward'
```

It Checks if there are rocks. Move towards the rock slowly with speed/3; Checks the distance from the rock. If it is not too close, seeing, Steering with angle diapason -20/20, but if it is too close, stops.

Drive_Rover autonomously

For the rock detection there were added two variables (*rocks_angles*, *rocks_dists*) to the Rover state. The missing action step was added following the comment “# The action step! Send commands to the rover!”





1. . 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 screen resolution was chosen 1024x768 with Good Graphics quality

The result. The robots moves quite well, collect rocks and avoid obstacle, but required still some more “smart” decisions to be implemented in the control mechanism.