Cheshta Dhingra

Udacity Robotics

Project 1 – Search and Sample Return

August 18, 2017

Project: Search and Sample Return

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.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. 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.

You're reading it!

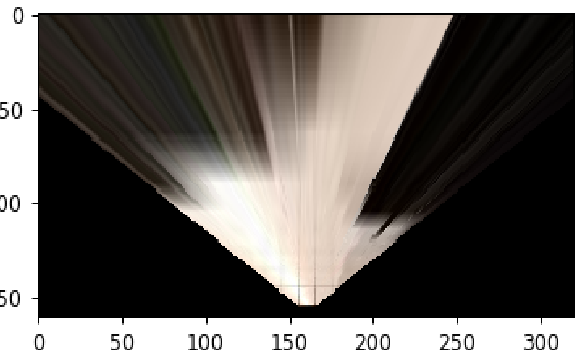
Notebook Analysis

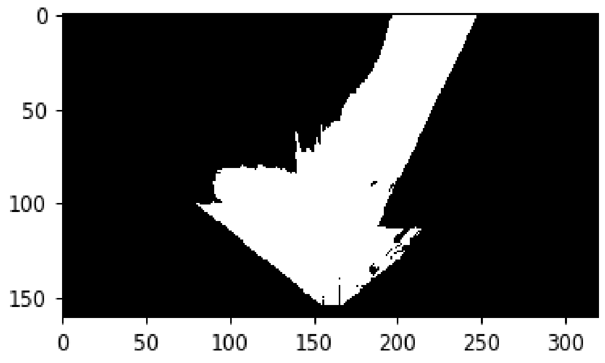
1. Run the functions provided in the notebook on test images (first with the test data provided, next on data you have recorded). Add/modify functions to allow for color selection of obstacles and rock samples.

The first thing I did was to record my own data – I have 503 images in my dataset.

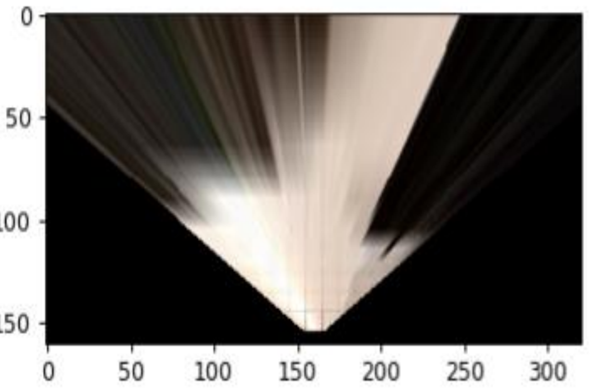
The first change I made to the notebook, was to add a color threshold to identify the obstacles. To do this, I simply reversed the threshold I used to identify terrain, as this seemed to work quite well. I used the values *below* the identified threshold (rgb_thresh=(160, 160, 160)) rather than *above* it. Below are the outputs from both the terrain color threshold (black represents terrain) and the obstacle threshold (black represents obstacles. This code can be found in the notebook in chunks 6 and 7, under Color Thresholding.

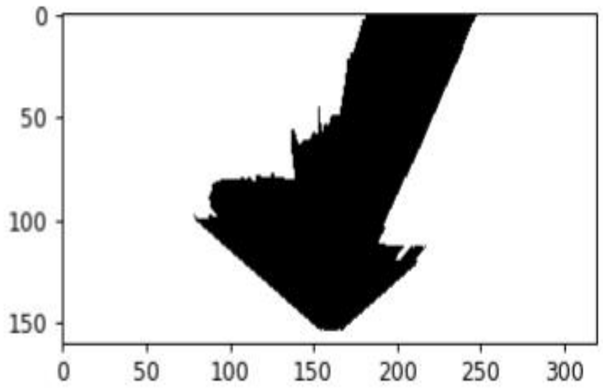
Terrain color threshold



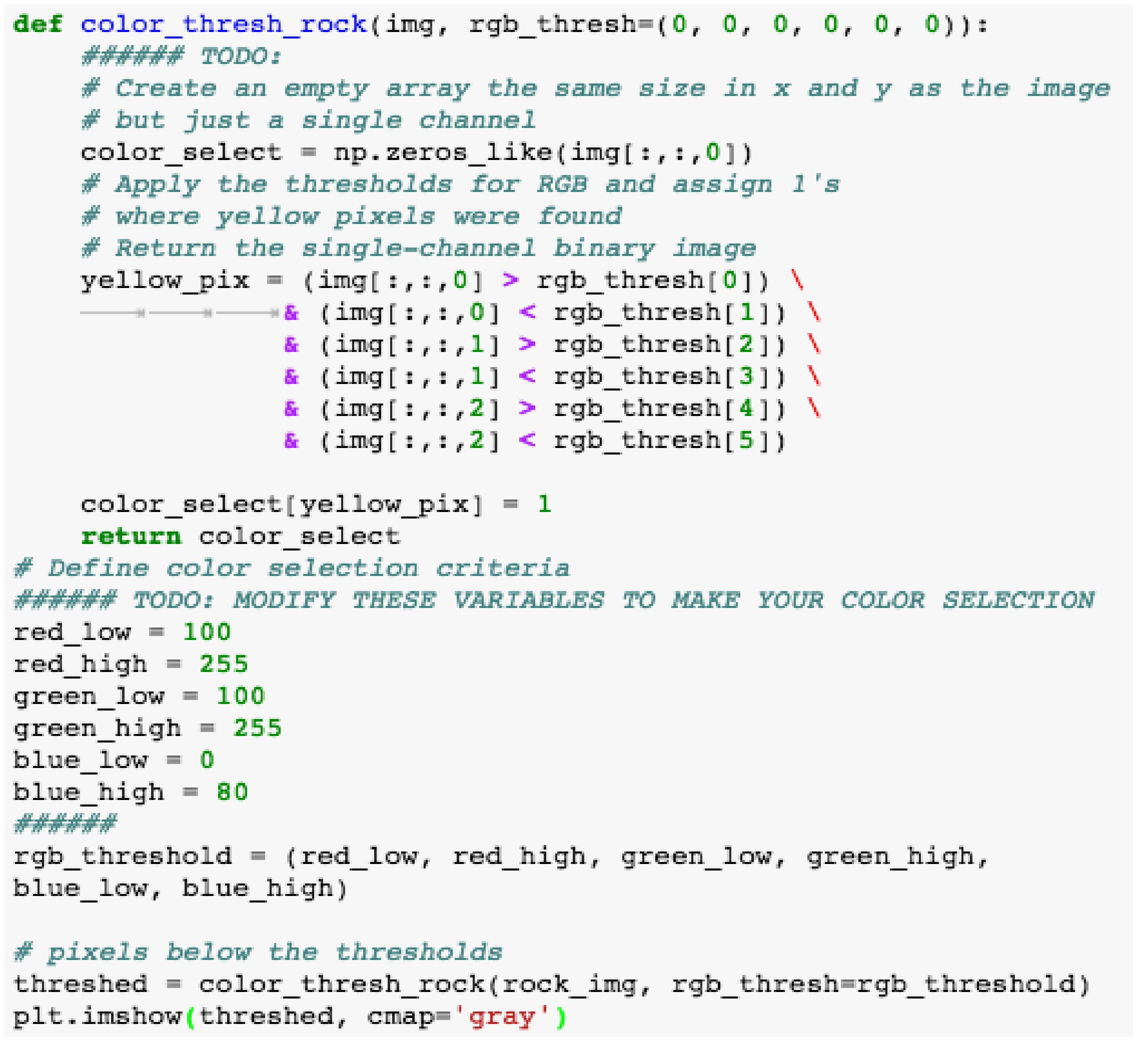


Obstacle color threshold





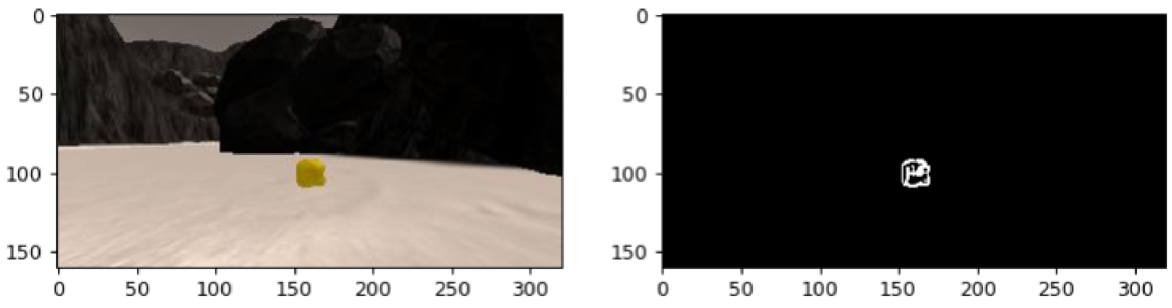
The second modification I made to the code was to identify rocks in the terrain. To do this, I initially spent a lot of time and trial and error to come up with color ranges for all three colors and came up with the following function, which generated the image below it:



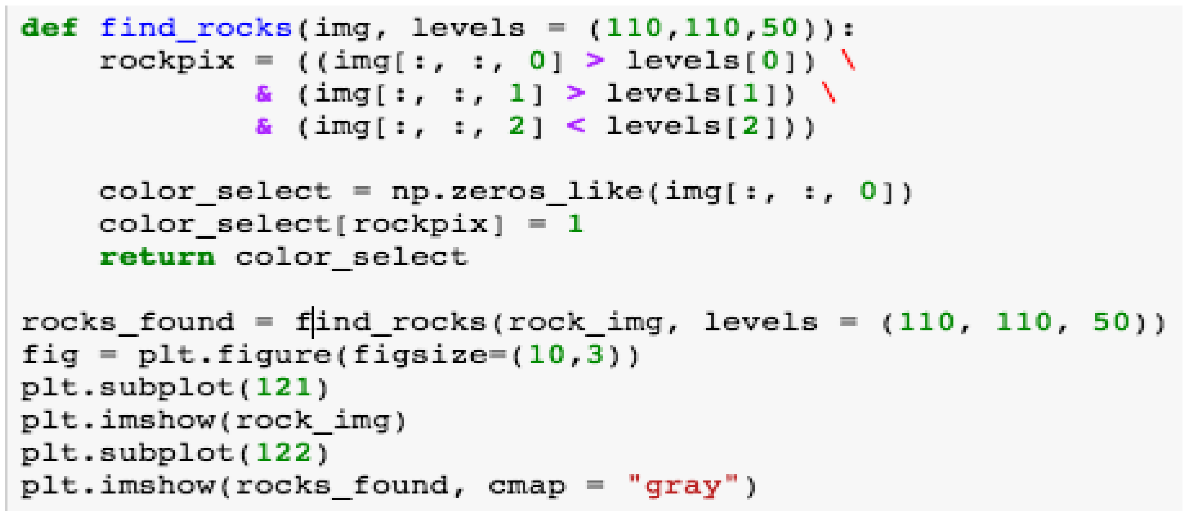
```
def color_thresh_rock(img, rgb_thresh=(0, 0, 0, 0, 0, 0)):
    ###### TODO:
    # Create an empty array the same size in x and y as the image
    # but just a single channel
    color_select = np.zeros_like(img[:,:,0])
    # Apply the thresholds for RGB and assign 1's
    # where yellow pixels were found
    # Return the single-channel binary image
    yellow_pix = (img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,0] < rgb_thresh[1]) \
                & (img[:,:,1] > rgb_thresh[2]) \
                & (img[:,:,1] < rgb_thresh[3]) \
                & (img[:,:,2] > rgb_thresh[4]) \
                & (img[:,:,2] < rgb_thresh[5])

    color_select[yellow_pix] = 1
    return color_select
# Define color selection criteria
###### TODO: MODIFY THESE VARIABLES TO MAKE YOUR COLOR SELECTION
red_low = 100
red_high = 255
green_low = 100
green_high = 255
blue_low = 0
blue_high = 80
######
rgb_threshold = (red_low, red_high, green_low, green_high,
blue_low, blue_high)

# pixels below the thresholds
threshed = color_thresh_rock(rock_img, rgb_thresh=rgb_threshold)
plt.imshow(threshed, cmap='gray')
```



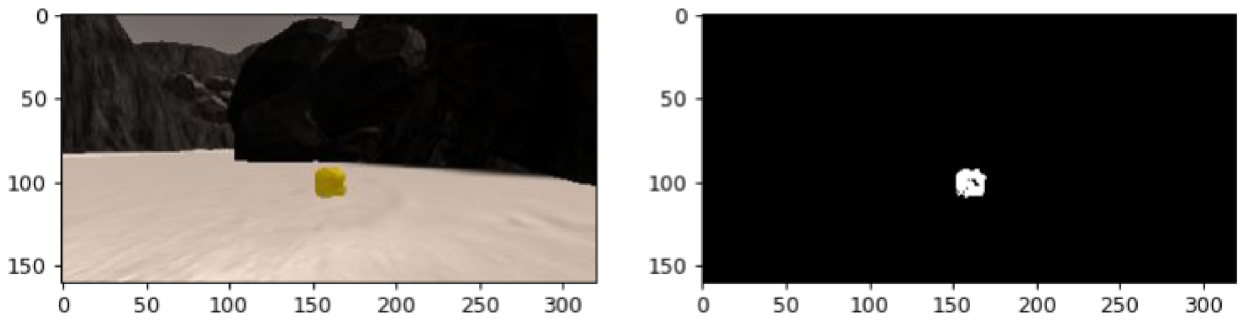
However, then I followed along with the project walkthrough and discovered that there was a better way to do this. So I replaced my function with the following, which gave a better approximation of the rock pixels:



```
def find_rocks(img, levels = (110,110,50)):
    rockpix = ((img[:, :, 0] > levels[0]) \
              & (img[:, :, 1] > levels[1]) \
              & (img[:, :, 2] < levels[2]))

    color_select = np.zeros_like(img[:, :, 0])
    color_select[rockpix] = 1
    return color_select

rocks_found = find_rocks(rock_img, levels = (110, 110, 50))
fig = plt.figure(figsize=(10,3))
plt.subplot(121)
plt.imshow(rock_img)
plt.subplot(122)
plt.imshow(rocks_found, cmap = "gray")
```

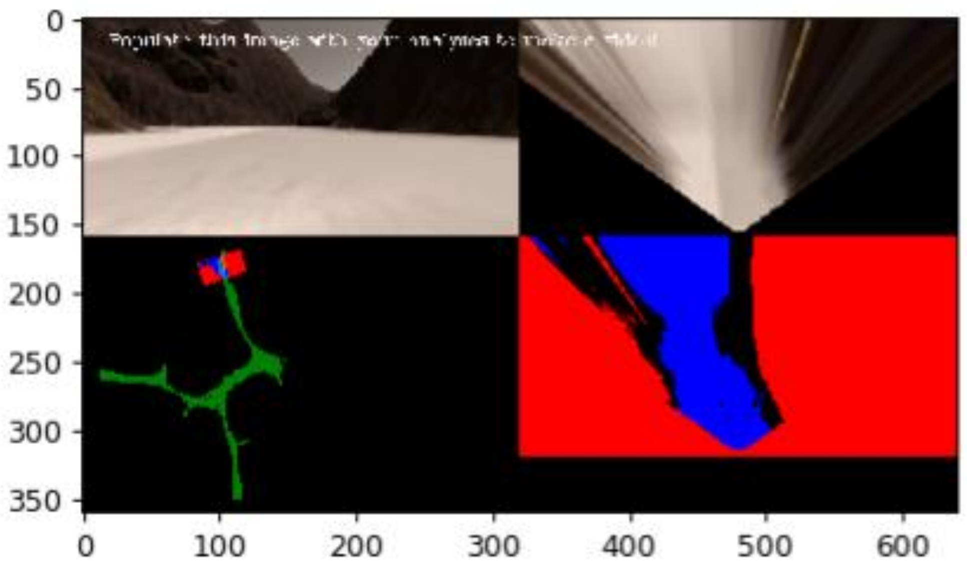


This code can be found in chunk 8 of the notebook.

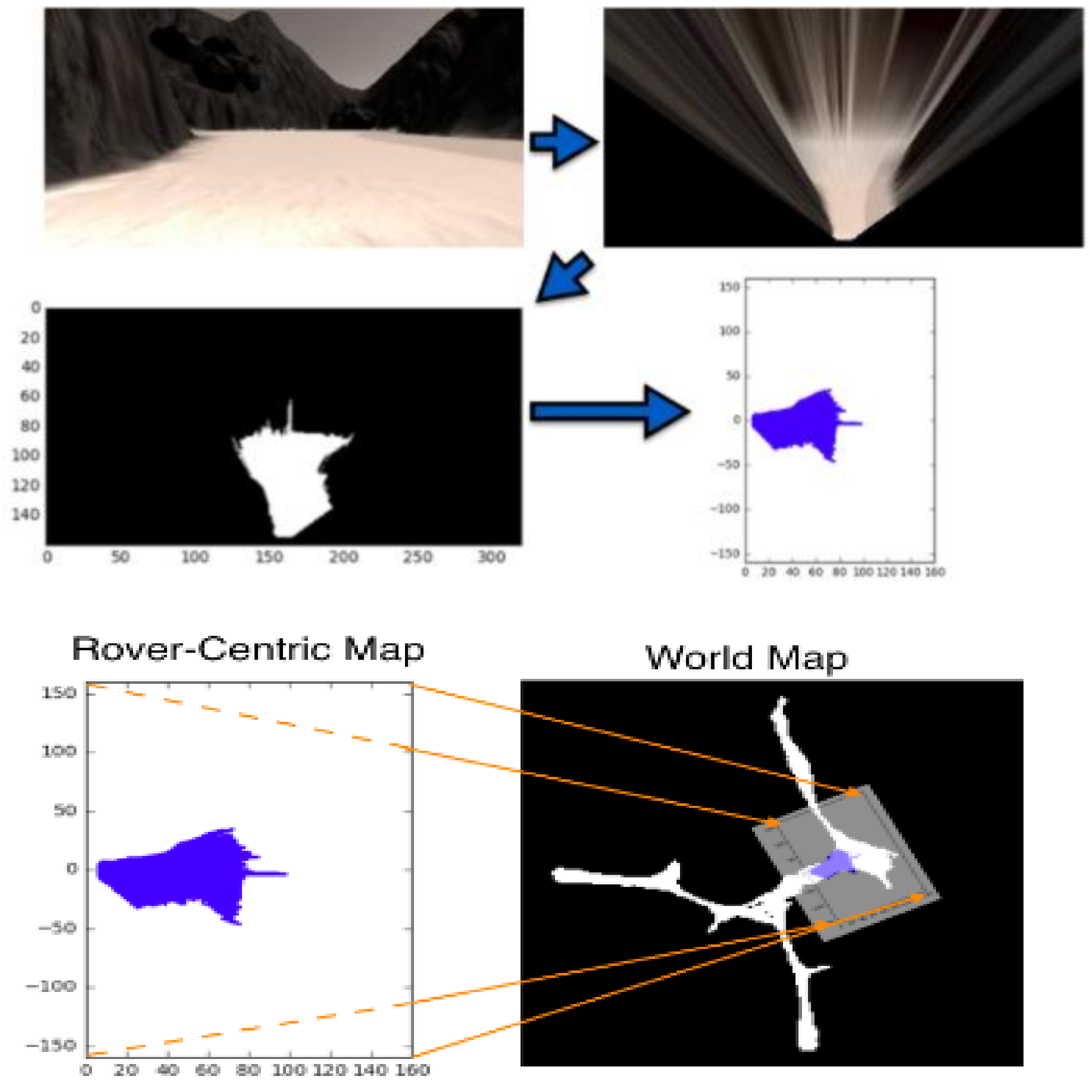
1. Populate the process_image() function with the appropriate analysis steps to map pixels identifying navigable terrain, obstacles and rock samples into a worldmap.
Run process_image() on your test data using the moviepy functions provided to create video output of your result.

Within the process_image() function, I essentially consolidated all of the image processing tasks into a single function. This included the following processing steps:

1. Perspective transform (after defining source and destination points) to gain a birds-eye-view of the Rover's surroundings.
2. Applying a color threshold to identify the navigable terrain and the obstacles.
3. Converting thresholded image pixel values into rover-centric coordinates, such the the rover's coordinate frame exactly matches the world frame.
4. Converting the rover-centric pixel values to world coordinates
    a. Rotate the rover-centric coordinates so that the x and y axes are parallel to the axes in world space.
    b. Translate the rotated positions by the x and y position values given by the rover's location (position vector) in the world.
5. Look for rocks! (once we find a rock, we recolor the pixel of that rocks position to be white)
6. Updating worldmap – in this step, as the rover traverses the terrain I update the colors of the pixels it views to correspond to the feature they represent (TERRAIN, OBSTACLE, SAMPLE)
7. Finally, making a mosaic image to get something that looks like this:



Here is a schematic of the entire process_image() function:



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.**

In perception_step(), we take an inputted camera image from the Rover and process it to update the Rover object. The steps I follow in this step are identical to those used in process_image(), with the addition of a few features that I updated, like Rover.nav_dists and Rover.nav_angles. Thus, I was able to capture images as the rover navigated the terrain autonomously, process them in real time and update the Rover object simultaneously. To recap, these are the steps I followed in perception_step():

1. Perspective transform
2. Color Threshold
3. Converting thresholded image pixel values into rover-centric coordinates
4. Converting the rover-centric pixel values to world coordinates
5. Finding Rocks
6. Updating worldmap

In decision_step(), I did not make any modifications, thereby generating only the basic functionality. The initial condition is controlled by the fact that Rover.nav_angles has been updated in perception_step() to be “not None”, thus setting off the decision tree for the Rover to navigate autonomously (albeit boringly). Here’s the algorithm:

```
If navigable angles =/= 0
    If the Rover mode is ‘forward’
            If navigable terrain looks sufficient
                If the velocity is below max
                    We use throttle
                Otherwise, the Rover coasts (throttle = 0).
            And the brake is set to 0
            Set steering to average angle clipped to the range +/- 15
            If there's a lack of navigable terrain pixels
                Then go to 'stop' mode
    If we’re already in stop mode
        If we're in stop mode but still moving, then keep braking
        If we're not moving (vel < 0.2) then do something else
            If len(Rover.nav_angles) < Rover.go_forward:
                Set throttle and brake to 0 (Release the brake to allow turning)

            If we're stopped but see sufficient navigable terrain in front then go!
                Set throttle back to stored value
                Release the brake
                Set steer to mean angle
                Set Rover mode to forward
Else
    Set the Rover.throttle to prespecifed amount, steer to 0 and brake to 0
```

Given more time for me to dedicate to this course, I would certainly like to experiment with this step more.

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.

For the simulator settings, I used 1024 x 768 and Good graphics quality.

Right now, I am able to achieve the required 60% fidelity and map over 40% of the environment. The Rover is also able to map the location of rocks as they appear in the vision image.

In order to improve the project, I would explore the Rover's ability to pick up rocks, maybe scale the walls, avoid obstacles as soon as they appear, and do a better job of getting out of a sticky situation when it gets stuck in the obstacles. Additionally, I notice that there are some red pixels in the video which are not actually navigated yet, and I would spend more time debugging this issue – which I think has to do with cached versions of the output.