**This document contains most of the write up for the project. Please look at the section to answer the explanations require per the course rubrics.**

**Notebook**

# Define a function to pass stored images to

# reading rover position and yaw angle from csv file

# This function will be used by moviepy to create an output video

**def process\_image(img):**

1. **Databucket() object defined**

**In this function, we are going to use the object Databucket to hold values about the objects (terrain, obstacles, rocks) ‘seen’ by the camera. This data will be used to show on the screen in the simulation.**

# print(data.xpos[0], data.ypos[0], data.yaw[0])

warp = perspect\_transform(img, source, destination)

output\_image = np.zeros((img.shape[0] + data.worldmap.shape[0], img.shape[1]\*2, 3))

**Example Output of what can be done with the incoming images to create pictures then to daisy –chain the images into a complete video. The video application will adjust the rate and output for the individual pictures to be viewed as a single movie.**

output\_image[0:img.shape[0], 0:img.shape[1]] = img

output\_image[0:img.shape[0], img.shape[1]:] = warp

cv2.putText(output\_image,"Populate this image with your analyses to make a video!", (20, 20),

cv2.FONT\_HERSHEY\_COMPLEX, 0.4, (255, 255, 255), 1)

data.count += 1 # This variable keeps track of the index (line items of objects) in the Databucket()

return output\_image

1. **def process\_image(img):**

\*\*Note: see def perception\_step(Rover) BELOW for full parallel explanation. Some modifications were done to run in the notebook. Bur rationale stays the same.

# 1) Define source and destination points for perspective transform

half\_square\_transition = 5

bottom\_offset = 9

source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])

dest = np.float32([[img.shape[1]/2 - half\_square\_transition, img.shape[0] - 2\*half\_square\_transition - bottom\_offset],

[img.shape[1]/2 + half\_square\_transition, img.shape[0] - 2\*half\_square\_transition - bottom\_offset],

[img.shape[1]/2 - half\_square\_transition, img.shape[0] - bottom\_offset],

[img.shape[1]/2 + half\_square\_transition, img.shape[0] - bottom\_offset]])

**# 2) Apply perspective transform**

warped = perspect\_transform(image, source, destination)

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

terrain = color\_thresh(warped)

obstacles = color\_thresh(warped,rgb\_thresh=(0, 0, 0),rgb\_thresh\_above=(160, 160, 160))

rock\_sample = color\_thresh(warped,rgb\_thresh=(150, 100, 0),rgb\_thresh\_above=(255, 200, 50))# (127, 94, 0)(232, 200, 77)

# 4) Update Rover.vision\_image (this will be displayed on left side of screen)

vision\_image = np.zeros((160, 320, 3), dtype=np.float)

vision\_image[:,:,0] = obstacles \* 255 **# Color - Red for obstacles \* 255 to display on left side of screen**

vision\_image[:,:,1] = rock\_sample \* 255 **# Color - Green rock samples**

vision\_image[:,:,2] = terrain \* 255 # **Color - Blue for terrain**

**# 5) Convert map image pixel values to rover-centric coord**s

terrain\_tran\_x, terrain\_tran\_y = rover\_coords(terrain)

obstacles\_tran\_x, obstacles\_tran\_y = rover\_coords(obstacles)

rock\_sample\_tran\_x, rock\_sample\_tran\_y = rover\_coords(rock\_sample)

# 6) Convert rover-centric pixel values to world coordinates Left side

worldmap = np.zeros((200, 200))

scale = 10

world\_size = 200

xpos = data.xpos[0]

ypos = data.ypos[1]

yaw = data.yaw

terrain\_x\_world, terrain\_y\_world = pix\_to\_world(terrain\_tran\_x, terrain\_tran\_y, xpos, ypos, yaw, world\_size, scale)

obstacles\_x\_world, obstacles\_y\_world = pix\_to\_world(obstacles\_tran\_x, obstacles\_tran\_y, xpos, ypos, yaw, world\_size, scale)

rock\_sample\_x\_world, rock\_sample\_y\_world, = pix\_to\_world(rock\_sample\_tran\_x, rock\_sample\_tran\_y, xpos, ypos, yaw, world\_size, scale)

**Simulation Analysis and Overview**

1. **The beginning lines of code was provided as part of the development of this project. No changes were made in the section.**

#### 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.

1. **Please note the rationale behind the changes or modifications to meet the coding criteria for completing the step.**

# Apply the above functions in succession and update the Rover state accordingly

**def perception\_step(Rover):**

# Perform perception steps to update Rover()

# TODO:

# NOTE: camera image is coming to you in Rover.img

**# 1) Define source and destination points for perspective transform**

**# Define a shorter variable (img) to hold the incoming camera image rather than writing the longer variable out (Rover.img). Saves space and easier for a human to read. The remainder of the code under Section 1 setups the destination and location of the rover.img image seen from the camera angle on the canvas or screen.**

img = Rover.img

half\_square\_transition = 5

bottom\_offset = 9

source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])

dest = np.float32([[img.shape[1]/2 - half\_square\_transition, img.shape[0] - 2\*half\_square\_transition - bottom\_offset],

[img.shape[1]/2 + half\_square\_transition, img.shape[0] - 2\*half\_square\_transition - bottom\_offset],

[img.shape[1]/2 - half\_square\_transition, img.shape[0] - bottom\_offset],

[img.shape[1]/2 + half\_square\_transition, img.shape[0] - bottom\_offset]])

**# 2) Apply perspective transform**

**# We are required to do a change from the perception of what the rover sees to what should be displayed on the screen. The perspective change will also allow us to digitize (old term) what is “seen” and where that frame of a picture if from. The transformation can be used to grid out the area as navigable or as an obstacle.**

warped = perspect\_transform(img, source, dest)

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

**# We are use colors to determine wither or not the object in from of the camera is terrain, an obstruction or a rock (sample). We used the variables names as the objects the system will see by using the color differences and perspectives. For those I define 3 variables. We also add the color thresh hold to identify the colors and then which object in the environment we are ‘looking at’.**

terrain = color\_thresh(warped)

obstacles = color\_thresh(warped,rgb\_thresh=(0, 0, 0),rgb\_thresh\_above=(160, 160, 160))

rock\_sample = color\_thresh(warped,rgb\_thresh=(150, 100, 0),rgb\_thresh\_above=(255, 190, 50))# (127, 94, 0)(232, 200, 77)

**# 4) Update Rover.vision\_image (this will be displayed on left side of screen)**

**# Update the Left bottom are of the screen to show the view point of the rover. We do a little conversion for the object seen and then render. The colors (0,1 ,2 ,3) used below represent the colors various objects and will be displayed about the rover camera map ( in the rover’s eyes).**

Rover.vision\_image[:,:,0] = obstacles \* 255 # Color - Red for obstacles \* 255 to display on left side of screen

Rover.vision\_image[:,:,1] = rock\_sample \* 255 # Color - Green rock samples

Rover.vision\_image[:,:,2] = terrain \* 255

**# 5) Convert map image pixel values to rover-centric cords**

**# Update the Left bottom are of the screen to show a flatten terrain and rock samples encountered.**

terrain\_tran\_x, terrain\_tran\_y = rover\_coords(terrain)

obstacles\_tran\_x, obstacles\_tran\_y = rover\_coords(obstacles)

rock\_sample\_tran\_x, rock\_sample\_tran\_y = rover\_coords(rock\_sample)

# 6) Convert rover-centric pixel values to world coordinates

**# Update the screen to show the view point of the rover. We do a little conversion for the object seen and then render. The size window is declared and scale size. All adjustable in the way I set the code up.**

worldmap = np.zeros((200, 200))

scale = 10

world\_size = 200

xcoor = Rover.pos[0]

ycoor = Rover.pos[1]

yaw = Rover.yaw

navigable\_y\_world, navigable\_x\_world = pix\_to\_world(terrain\_tran\_x, terrain\_tran\_y, xcoor, ycoor, yaw, world\_size, scale)

obstacle\_x\_world, obstacle\_y\_world = pix\_to\_world(obstacles\_tran\_x, obstacles\_tran\_y, xcoor, ycoor, yaw, world\_size, scale)

rock\_sample\_y\_world, rock\_sample\_x\_world = pix\_to\_world(rock\_sample\_tran\_x, rock\_sample\_tran\_y, xcoor, ycoor, yaw, world\_size, scale)

**# 7) Update Rover worldmap (to be displayed on right side of screen)**

**# Update the Right bottom are of the screen to show the view point of the rover as a flat area bird view. We do a little conversion for the object seen and then render.**

Rover.worldmap[navigable\_y\_world, navigable\_x\_world, 0] += 1

Rover.worldmap[obstacle\_y\_world, obstacle\_x\_world, 2] += 1

Rover.worldmap[rock\_sample\_y\_world, rock\_sample\_x\_world, 1] += 1

**# 8) Convert rover-centric pixel positions to polar coordinates**

**# Update rover information to be retuned as seen in the excerises.**

rover\_centric\_pixel\_distances, rover\_centric\_angles = to\_polar\_coords(terrain\_tran\_x, terrain\_tran\_y)

# Update Rover pixel distances and angles

Rover.nav\_dists = rover\_centric\_pixel\_distances

Rover.nav\_angles = rover\_centric\_angles

return Rover



**import numpy as np**

**def decision\_step(Rover):**

1. **Decision Tree**
2. **In this code area we are using a decision tree (if-then checks in other languages) to determine throttle, brake and steering from the commands that were output of the perception\_step() function.**
3. **As the conditions are read we then decide what action to take. The decisions of does the condition exist or has changed read from the perception\_step() function will then be used to ‘direct’ the next action of the rover. Subsequently, the next actions or reactions will be based on the next group of data from the rover.**

**# Example:**

**# Check if we have vision data to make decisions with**

**if Rover.nav\_angles is not None:**

**# Check for Rover.mode status**

**if Rover.mode == 'forward':**

**# Check the extent of navigable terrain**

**if len(Rover.nav\_angles) >= Rover.stop\_forward:**

**# If mode is forward, navigable terrain looks good**

**# and velocity is below max, then throttle**

**if 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**

**# Set steering to average angle clipped to the range +/- 15**

**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**

1. **If there is no forward action or terrian runs out then this check will stop and then in the next decision change directions or other action.**

**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**

1. **If the rovers forward action is stopped, then try a different action and finally into 4-wheel spin to look for another direction.**

**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**

**# 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**

1. **If the rover is stopped then check to see if sufficient navigable terrain exists in front of the rover. If the condition of open terrain exists, then move forward.**

**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 angle**

**Rover.steer = np.clip(np.mean(Rover.nav\_angles \* 180/np.pi), -15, 15)**

**Rover.mode = 'forward'**

**# Just to make the rover do something**

**else:**

**Rover.throttle = Rover.throttle\_set**

**Rover.steer = 0**

**Rover.brake = 0**

**return Rover**

**class RoverState():**

#added other rover status objects these will be helpful to keep up with issues such as when to return to the starting point.

self.perception\_count = 0 # Counter to keep track of map cleanup occurrence in XXX frames

self.start\_pos = None # Save starting position for return

self.rock\_pause = False # Pauses to count and stop redundant counts

self.samples\_collected = 0 # Counts till all 6 rocks are collected

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

1. **Results**

**The rover was automated and see to do extremely well in mapping the world. Some of my original code has a few flaws causing the rover to reverse and stay in a spin. I believe that as created by bad coding on recognizing the terrain verse obstacles. I ran the simulation many times for debugging and to see how far along the rover would get in the world. All runs seem to result in 100% mapping but only finding 5 of the rocks (samples). The simulation ran without any interference (intervention) on my behalf except for where I tried to figure out the issue with ‘why’ the rover did not see the last rock. Regardless of how many time, the rover showed object nearby or I could force pick up, the count never incremented. The map did show the location of each rock including the other informational items. The 6th rock was present but the location in a narrow gorge seem to prevent the rover for “seeing” the rock as in perspective. The distance away from the object to match the sample photograph. I could be wrong here but as far as the results, I was very pleased.**

**In Figure 1, below this was an assisted manual with all rocks present but the 6th did not populate as found. Of course, the fidelity is low as I helped search for the rocks. In Figure 2, higher fidelity is present at 61.59% @ 49% of the world mapped. The simulator was running at ~30 FPS with 680x7640 resolution and ''Fastest” graphics quality.**



**Figure 1**



Figure 2.