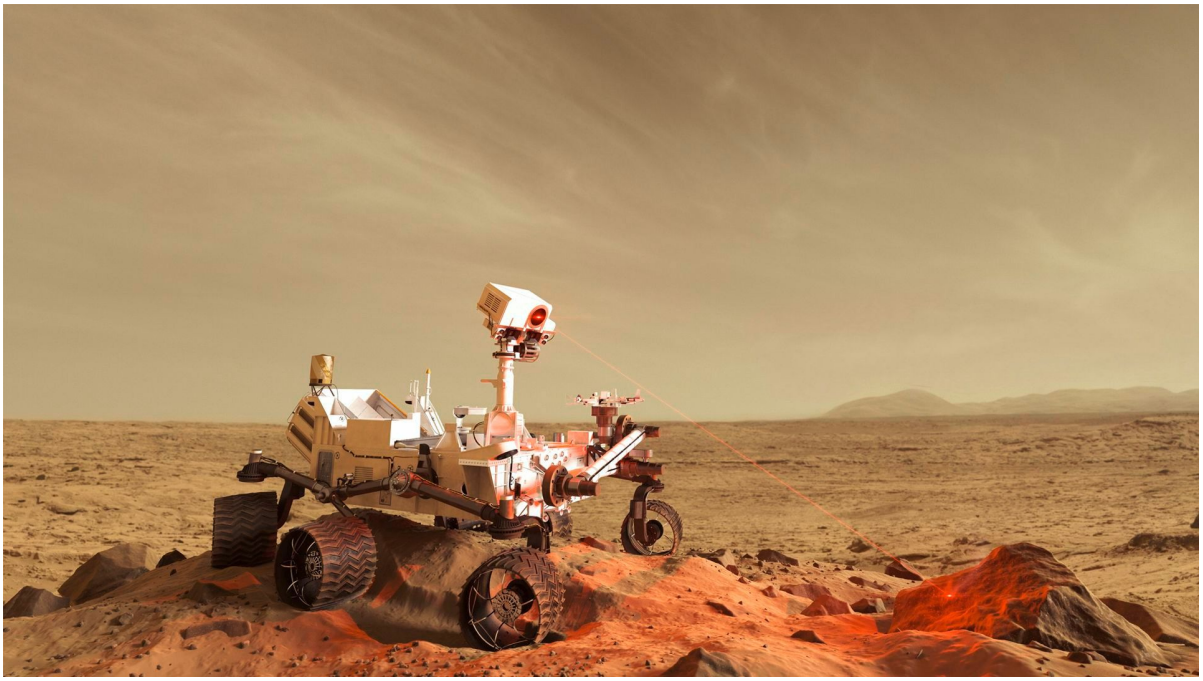
CSE483 Computer Vision

Project Documentation

Fall 2022



Presented to:

Prof. Mahmoud Khalil

Eng. Mahmoud Ahmed Selim

Presented by:

Omar Ayman Ayoub Abdelshafi 1900804

Mohamed Aly Elsayed Matar 19p5238

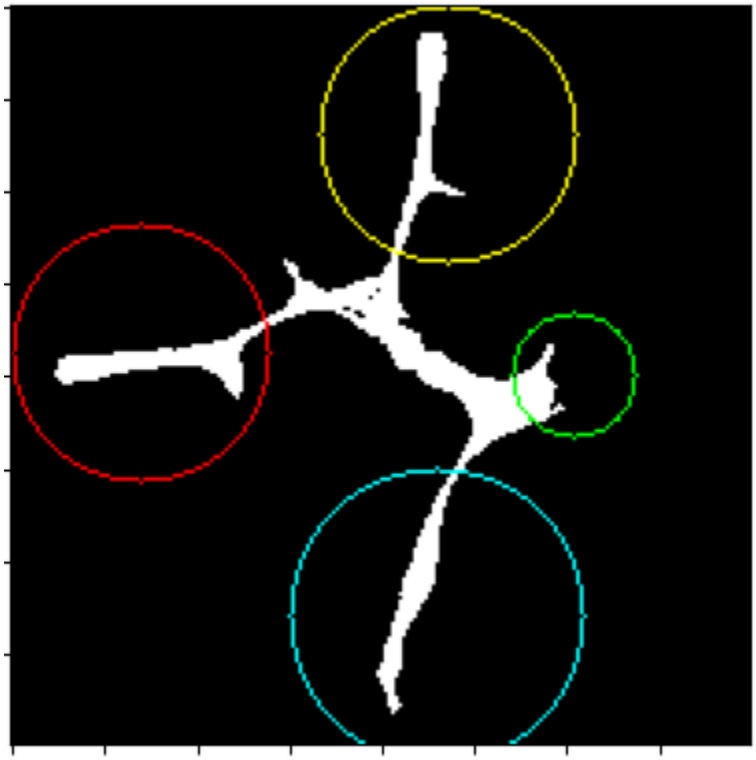
Seifeldin Sameh Mostafa 19p3954

Yara Mostafa Ibrahim Marei 19p1120

VISITED :

We Mark the places the rover visited so it doesn't waste time wondering about visiting the places again and again

The 4 colored regions in the figure below are marked as visited if the rover visits any of them



HOW DOES IT WORKS ?

When the Rover is inside any circle of these circles, it means that the rover position is inside the circle which also means that the distance from Rover's center to this circle center is less than the radius So the following checks are made to check if i'm inside any of them

```
def check_partially_visiting():
    # mark visited
    x, y = Rover.pos
    for place in places:
        upperx = place["center"][0] + place["radius"]
        lowerx = place["center"][0] - place["radius"]
        uppery = place["center"][1] + place["radius"]
        lowery = place["center"][1] - place["radius"]
        if (x >= lowerx and x <= upperx and y >= lowery and y <= uppery):
            place["partially_visited"] = True
            if place["number"] not in Rover.p_vis:
                Rover.p_vis.append(place["number"])
```

WHY PARTIALLY VISITED ?

Because if i marked it as visited and I'm inside it , the rover will get stuck; Once it enters the circle perimeter, it will try to get out

FIX :

```
def check_totally_visited():
    x, y = Rover.pos
    for place in places:
        upperx = place["center"][0] + place["radius"]
        lowerx = place["center"][0] - place["radius"]
        uppery = place["center"][1] + place["radius"]
        lowery = place["center"][1] - place["radius"]
        if not (x >= lowerx and x <= upperx and y >= lowery and y <= uppery) and place["partially_visited"]:
            place["totally_visited"] = True
            if place["number"] not in Rover.t_vis:
                Rover.t_vis.append(place["number"])
                Rover.visited+=1
```

Mark it visited (totally visited) if i partially visited it and now i am out

```python
def check_if_inside_visited():
    # rotate 180 and go forward for 0.5 seconds
    x, y = Rover.pos
    for place in places:
        upperx = place["center"][0] + place["radius"]
        lowerx = place["center"][0] - place["radius"]
        uppery = place["center"][1] + place["radius"]
        lowery = place["center"][1] - place["radius"]
        if (x >= lowerx and x <= upperx and y >= lowery and y <= uppery) and place["totally_visited"]:
            return place["yaw"] + 1.0
    return False
```

This function checks if i'm inside a totally visited circle .

What happens if I visited a totally visited circle ?

The rover will get into jump scare mode

```python
82
83      if Rover.mode == "forward" and check_if_inside_visited():
84          Rover.mode = "jump_scare"
85
```

```python
elif Rover.mode == "jump_scare":  # just saw the devil
    if Rover.vel == 0:
        Rover.throttle = 0
        Rover.mode = "scared"
        Rover.first_yaw = Rover.yaw
        return Rover
    else:
        Rover.brake = 15
        Rover.throttle = 0
        # Release the brake to allow turning
```

What Does this jump_scare mode do ?

If the rover enters the circle, the jump_scare mode will stop it and transfer Rover.mode to scared mode

What Does this scared mode do ?

Scared mode make the Rover either rotate 180 degrees with respect to the yaw it entered the circle with or give it a predefined yaw (there is a predefined yaw for each circle) (both ways works very well)

```
elif Rover.mode == "scared":  # rotate to run
    # Release the brake to allow turning
    r = random.randint(-2, 2)  # if stuck try different moves
    Rover.brake = 0
    Rover.throttle = 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 -1/1
    yaw2 = check_if_inside_visited()
    Rover.brake = 0
    Rover.throttle = 0
    # if (abs(Rover.yaw-yaw) <= 2): #rotate 180 degres till equal yaw
    if (abs((Rover.yaw - Rover.first_yaw)) >= (159 + 10*r)):  # rotate till = place["yaw"]
        Rover.mode = "run"
    return Rover
```

After Rover got scared it will run

Run mode :

```
elif Rover.mode == "run":  # keep going until out

    Rover.steer = np.clip(np.mean(Rover.nav_angles * 180 / np.pi), -8, 8)
    Rover.brake = 0
    Rover.throttle = 0.1
    if not check_if_inside_visited():
        Rover.mode = "forward"
    return Rover
```

In this mode, the rover will run till it’s not inside the visited circle that it visited accidentally

ROTATE MODE

```
elif Rover.mode == "rotate":
    r = random.randint(-2, 2)  # if stuck try different moves
    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 -1/1
    if ( abs(Rover.yaw - Rover.rot_yaw) >= (120 + (-r * (30))) ) :
        Rover.mode = "forward"  ##should be the stop mode where it rotate
```

In this mode the Rover will rotate till it's (180 - 90) degree of it's initial position #mainly used when the rover is stuck

How to know if the rover is stuck ?

If the rover is in the same position for too long, then it's stuck

```
def same_pos():
    if (abs(Rover.pos[0] - Rover.pos_prev[0]) < 0.01) and (
            abs(Rover.pos[1] - Rover.pos_prev[1]) < 0.01) and Rover.mode == "forward":
        return True
    else:
        return False
```

So we check if X,Y of the rover didn't change by more than 0.01 then the rover is stuck

```
if same_pos():
    Rover.pos_count += 1
else:
    Rover.pos_count = 0
```

```
###STUCK #####################################

check_partially_visiting()
check_totally_visited()

if Rover.mode == "forward" and check_if_inside_visited():
    Rover.mode = "jump_scare"

if same_pos():
    Rover.pos_count += 1
else:
    Rover.pos_count = 0

Rover.pos_prev = Rover.pos
if (Rover.pos_count >= Rover.max_pos_count):

    Rover.toggle += 1
    print("fffffff", Rover.toggle)
    print("ffffff22", Rover.toggle%2)
    if Rover.toggle % 2 == 0:
        Rover.mode = "reverse"
    else:
        Rover.rot_yaw = Rover.yaw
        Rover.mode = "rotate"  ##should be the stop mode where it rotate
        # seif change

    Rover.pos_count = 0
    return Rover

##########POS END#####################################
```

The rover may be stuck in another way as it rotate in a circle and keep rotating around it

```
#####STERING FOR TOO LONG #############
upper = Rover.steer_prev + 2
lower = Rover.steer_prev - 2
if (Rover.steer >= lower) and (Rover.steer <= upper) and (
        Rover.steer > 10 or Rover.steer < -10) and not Rover.gold_flag and Rover.mode == "forward":
    Rover.steer_count += 1
else:
    Rover.steer_count = 0

if (Rover.steer_count >= Rover.max_steer_count):  ####better 250 needs to be tried ;;;;;;;;;;
    Rover.steer_count = 0
    Rover.brake = 15
    Rover.steer = -15
    Rover.mode = "reverse"
Rover.steer_prev = Rover.steer

Rover.gold_flag = False

#####STERING END ##############################
```

How are rocks identified ?

```
# Define a function to identify rocks
def find_rocks(img,Rock_thresh_low=(100,100,0),Rock_thresh_high=(255,255,55)):

    color_select = cv2.inRange(img,Rock_thresh_low,Rock_thresh_high)

    return color_select
```

Why (100,100,0) & (255,255,55) ?

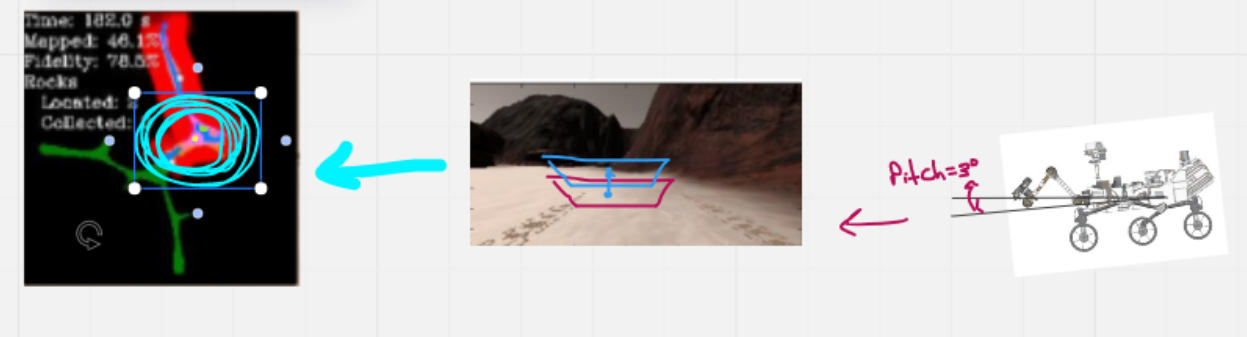
These Numbers works perfectly as 255,255,0 is the brightest yellow and i added 55 offset for caution in case rock is in light

And 100,100,0 is very very dark yellow in case the rock was in shadow

Why Roll and pitch limited ?

```
if (Rover.roll < roll_limit or Rover.roll > 360 - roll_limit) and (Rover.pitch < pitch_limit or Rover.pitch > 360 - pitch_limit):
    Rover.worldmap[rock_y_world, rock_x_world, 1] += 1
    Rover.worldmap[obs_y_world, obs_x_world, 0] += 1
    Rover.worldmap[navigable_y_world, navigable_x_world, 2] += 1
    nav_pix = Rover.worldmap[:, :, 2] > 0
    Rover.worldmap[nav_pix, 0] = 0
    # clip to avoid overflow
    Rover.worldmap = np.clip(Rover.worldmap, 0, 255)
```

Because if there is a roll or a pitch more than 0.001 the place of feasible path will be shifted in map



Where the yellow dot in map represent what the rover think it sees while the yellow dot is what it actually sees

HOW DOES ROVER DETERMINE ITS PATH?

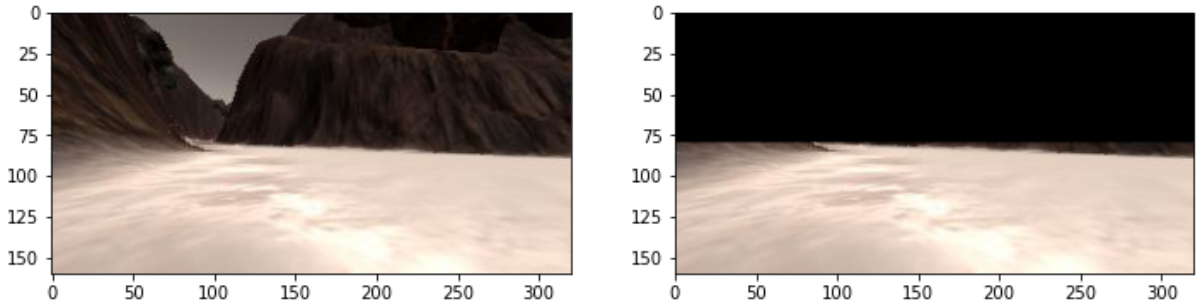
Rover sends its image to perception.py to process it and determine best navigable angles and distance to walk to.

This process consists of several phases that we will dive into.

PREPROCESSING

The top half of the image is set to zero as we need to limit rover vision to get better results and better decisions.

```
Image has shape: (160, 320, 3)
```

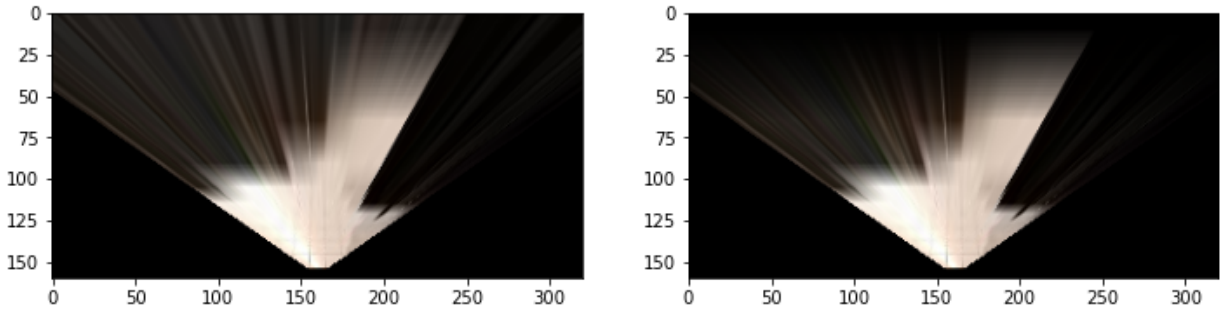


1- PERCEPTION TRANSFORMATION

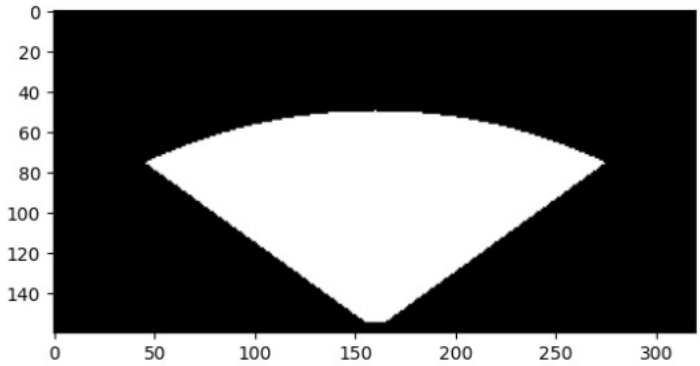
```python
# Define a function to perform a perspective transform
def perspect_transform(img, src, dst):
    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]))  # keep same size as input image
    mask = cv2.warpPerspective(np.ones_like(img[:,:,0]), M, (img.shape[1], img.shape[0]))

    return warped, mask
```

This function is responsible for taking rover images and transforming it into a bird-eye view. This is an essential step in our end goal of determining rover position with respect to our world(mars). Output of this phase is shown in figure below.

Figure below shows the effect of setting top half of given image to 0 then perception transforming(right) as opposed to taking image without any preprocessing done and applying perception transformation(left)



A mask is also returned to limit rover vision, which is shown in figure below.

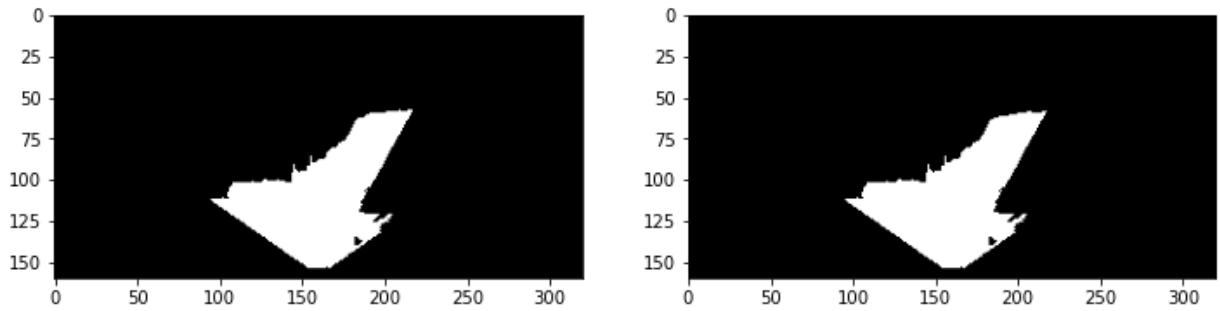


2- COLOR THRESHOLDING

The bird-eye view image is then inputted into a thresholding function that outputs a binary image. A Gaussian filter is used then added to the original image to help remove black dots in the middle of the image that most likely are noise.

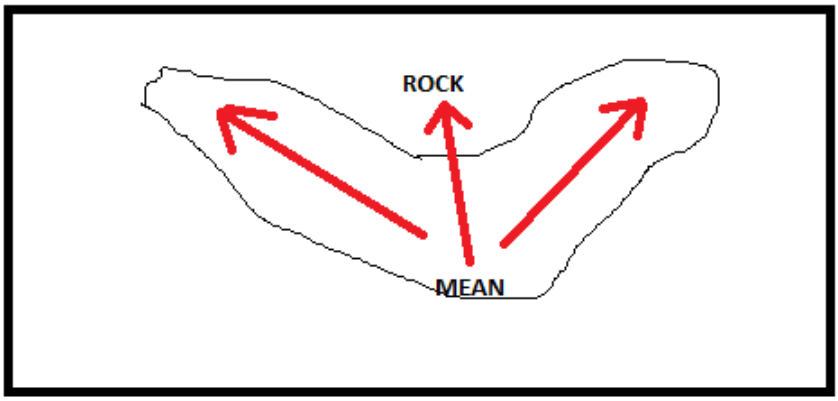
```python
def color_thresh(img, rgb_thresh=(160, 160, 160), kernel_size=3):
    gimg = cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)
    # Create an array of zeros same xy size as img, but single channel
    color_select = np.zeros_like(img[:, :, 0])

    # Require that each pixel be above all three threshold values in RGB
    # above_thresh will contain a boolean array with "True" whenever threshold is met
    above_thresh = (img[:, :, 0] > rgb_thresh[0]) \
                   & (img[:, :, 1] > rgb_thresh[1]) \
                   & (img[:, :, 2] > rgb_thresh[2])

    above_thresh_gaussian = (gimg[:, :, 0] > rgb_thresh[0]) \
                            & (gimg[:, :, 1] > rgb_thresh[1]) \
                            & (gimg[:, :, 2] > rgb_thresh[2])

    # Index the array of zeros with the boolean array and set to 1
    color_select[above_thresh] = 1
    color_select[above_thresh_gaussian] = 1
    # Return the binary image
    return color_select
```

Output of this phase is shown in figure below. Figure below shows the effect of setting top half of given image to 0 (right image) as opposed to taking image without any preprocessing done (left image)



3-DECISION-MAKING

In this step comes the trickiest part, we need to make split decisions on whether to turn right or left sometimes and taking the mean of all angles doesn't quite do the trick. For example, the figure below shows one of these situations.



The rover fails to determine the best path and instead collides with rock. Our function solves this problem by picking the stronger magnitude and going in its direction.

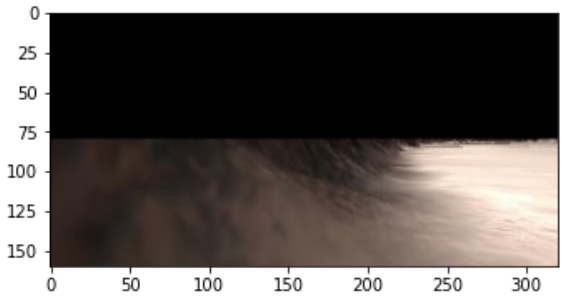
```
def divideConquer(image, source, destination,orgdist,organgle, Rover):
    image1 = np.zeros_like(image[:,:,:])
    image2 = np.zeros_like(image[:,:,:])
    image1[:,:160,:] += image[:,:160,:]
    image2[:,160:,:] += image[:,160:,:]
    warped1, mask = perspect_transform(image1, source, destination) #wraped is the bird-eye view perspective
    threshed1 = color_thresh(warped1)
    warped2, mask = perspect_transform(image2, source, destination) #wraped is the bird-eye view perspective
    threshed2 = color_thresh(warped2)
    xpix1, ypix1 = rover_coords(threshed1)
    dist1, angles1 = to_polar_coords(xpix1, ypix1)

    xpix2, ypix2 = rover_coords(threshed2)
    dist2, angles2 = to_polar_coords(xpix2, ypix2)

    magnitude1 = np.count_nonzero(threshed1)
    magnitude2 = np.count_nonzero(threshed2)

    if abs(np.mean(angles1)) < 0.5 and abs(np.mean(angles2)) < 0.5:
        ret_dist = orgdist
        ret_angle = organgle
        #return orgdist, organgle
    #elif abs(magnitude1-magnitude2) < 100:
     #   ret_dist = Rover.prev_angles[0]
      #  ret_angle = Rover.prev_angles[1]
```

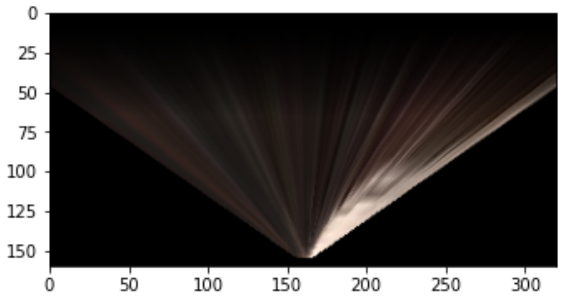
```
    if abs(np.mean(angles1)) < 0.5 and abs(np.mean(angles2)) < 0.5:
        ret_dist = orgdist
        ret_angle = organgle
        #return orgdist, organgle
    #elif abs(magnitude1-magnitude2) < 100:
     #    ret_dist = Rover.prev_angles[0]
      #   ret_angle = Rover.prev_angles[1]
    elif abs(np.mean(angles1)) > 0.6 and abs(np.mean(angles2)) < 0.4:
        ret_dist = orgdist
        ret_angle = organgle
    elif abs(np.mean(angles1)) < 0.4 and abs(np.mean(angles2)) > 0.6:
        ret_dist = orgdist
        ret_angle = organgle
    elif magnitude1 > magnitude2:
        ret_dist = dist1
        ret_angle = angles1
        #return dist1, angles1
    else:
        ret_dist = dist2
        ret_angle = angles2
        #return dist2, angles2

    #Rover.prev_angles = [ret_dist, ret_angle]
    return ret_dist, ret_angle
```

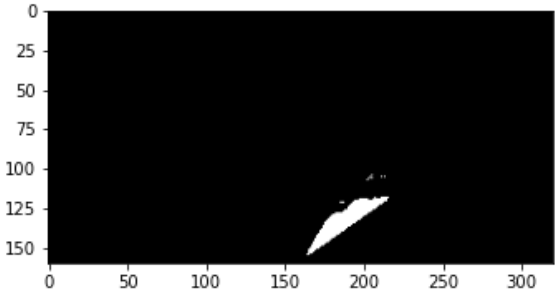
The image is first split horizontally to 2 images. Each half of the image then goes through our original image processing and each angle and distances are determined for each half. The image with stronger magnitude wins and determines the rover's next direction.

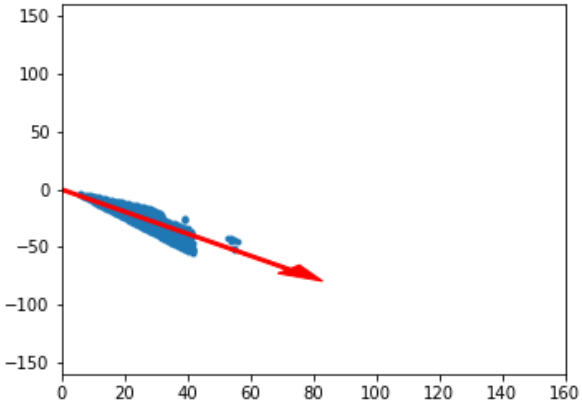
Magnitude is measured by the number of ones present in each respective half of the image.

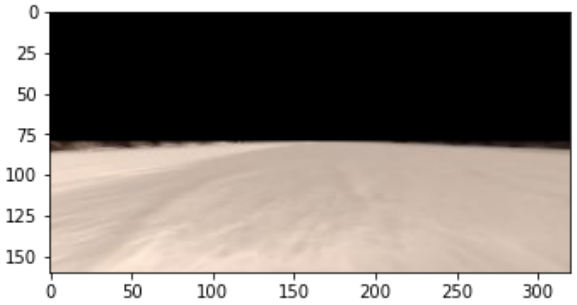
PIPELINE RESULTS

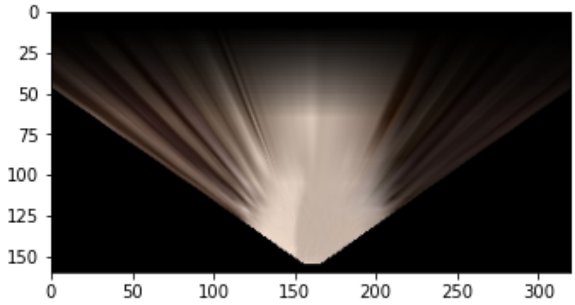


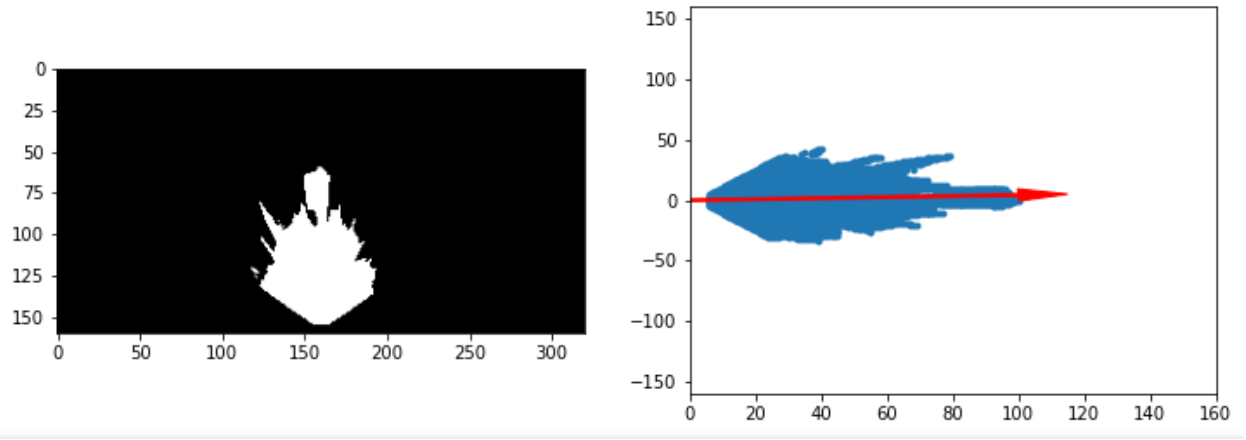












WHAT IS PID?

It's a short for Proportional–Integral–Derivative controller (PID controller or three-term controller)

It's a control loop mechanism employing feedback that is widely used in industrial control systems and other applications requiring continuously modulated control.

HOW DOES PID WORK?

The PID continuously calculates an error value e(t) as the difference between a desired setpoint (SP) and a measured process variable (PV) and applies a correction based on proportional, integral, and derivative terms as the name says it all (P, I, and D respectively)

COMING UP WITH EACH GAIN TERM

Proportional term lets us to steer harder the further away we were from our setpoint we use a cross-track error to measure how far enough we are from the setpoint the value the P gain drastically alters the performance of the rover

but with only proportional control the rover could be crooked when it reaches the center line and will repeatedly overshoot the actual trajectory

To fix this problem we will add an additional error rate and that's the cross-track error

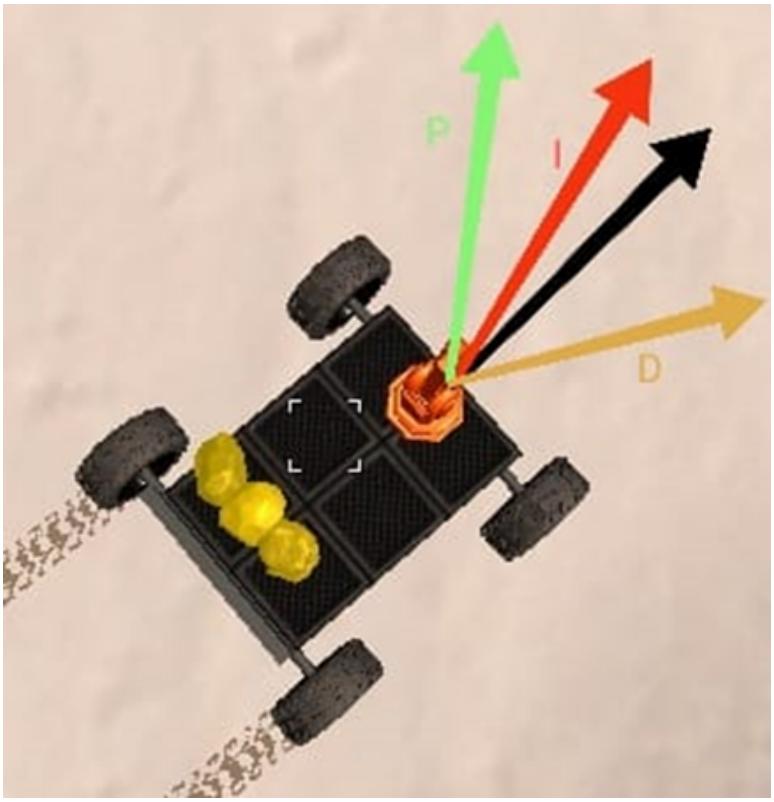
rate or how fast we are moving perpendicularly with respect to the desired trajectory, that's what we call the Derivative term.

The more rapid the change, the greater the controlling or damping effect.

Until now our steering angle is good to go, or is it?

what if our rover was affected by an external activity like heavy winds, rocks, etc… then that will build a lane offset to our desired setpoint

To solve this issue, we will add a third term which is the Integral term, it simply sums the cross-track error to give an indication if the rover is staying too long on one side of the trajectory or the other



"A real snapshot from the working rover"

TUNING

Tuning – The balance of these effects is achieved by loop tuning to produce the optimal control function.

The tuning constants are written as "K" and derived for each control application.

Calculating and tuning these constants could be challenging,

A common approach says that they are not calculated , they are experimentally determined.

Let’s look at each term

The proportional term is simple it is just a differential using error, if a change made to the input to wil be heard immediately in the output

“PTerm = kp * error”

The integral term uses time and adds a portion of the error to the I term and it would be used to help point out the setpoint.

“integral += error * (change in time * .000001)

I = ki * integral”

The derivative term uses the rate of change to temporarily add or subtract from the output to alter the course.

“derivative = (input - last input) / (change in time * .000001)

D = -kd * derivative”

fortunately python have a module that made it simple to us, it’s called “simple_pid”

We simply import PID from it

```
from  simple_pid import PID
```

Then we define a new function the Rover class containing our “k” constants values

Since we made our rover to avoid rocks there was no need for an integral value (no external affects available either), and the setpoint will be set later in the main code

```
self.pid = PID(0.5, 0.1, 0)
```

This is how the function definition looks like, we can see our “k” values

```
class PID(object):
    """A simple PID controller."""

    def __init__(
        self,
        Kp=1.0,
        Ki=0.0,
        Kd=0.0,
        setpoint=0,
        sample_time=0.01,
        output_limits=(None, None),
        auto_mode=True,
        proportional_on_measurement=False,
        error_map=None,
    ):
        """
        Initialize a new PID controller.
```

Here instead of sending our mean angle directly to the steer parameter, we set it as the setpoint first then we set our steer to the output of the PID

```
# Rover.steer = np.clip(np.mean(Rover.nav_angles * 180 / np.pi), -15, 15)
Rover.pid.setpoint =  np.clip(np.mean(Rover.nav_angles * 180 / np.pi), -15, 15)
Rover.steer = Rover.pid(Rover.steer)
```

Github

Final code on this branch

https://github.com/DedRec/Computer_Vision_Project/tree/Phase-2

Repo for phase 1 and branches for phase 2

https://github.com/DedRec/Computer_Vision_Project

Video link of the pipeline

https://drive.google.com/file/d/16L8LJTJJOZX53_H-V_9oV3_2veQT7cgd/view?usp=share_link