EE 3EY4
Lab 9

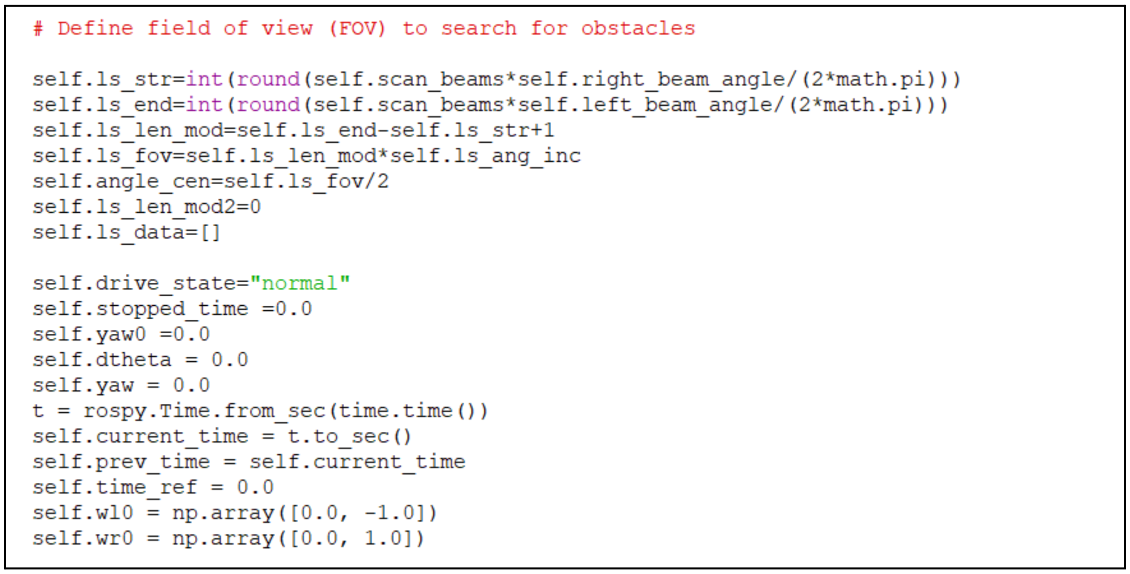
-

Stefan Tosti - Tostis - 400367761 - L05
Luke Hendrikx - Hendrikl - 400388795 - L05
Soham Patel - Pates206 - 400380121 - L05

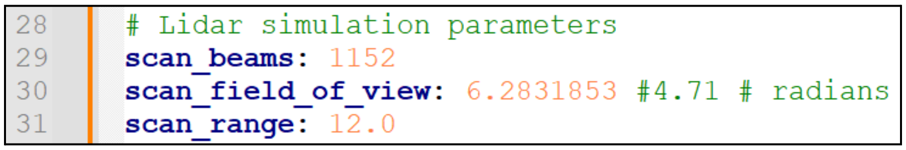
2024 - 03 - 29

Activity 1
Completed

Activity 2
Find below the parts of the code in *navigation_lab9_inc.py* and *params.yaml* that accomplish Step 1…



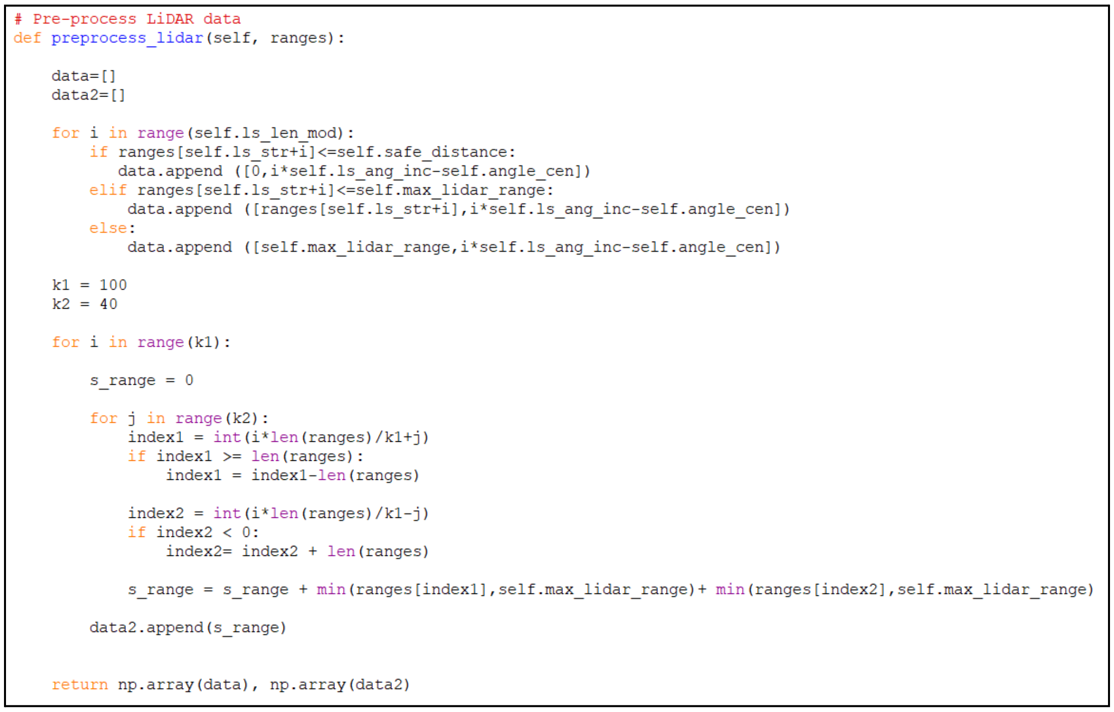
```
# Define field of view (FOV) to search for obstacles

self.ls_str=int(round(self.scan_beams*self.right_beam_angle/(2*math.pi)))
self.ls_end=int(round(self.scan_beams*self.left_beam_angle/(2*math.pi)))
self.ls_len_mod=self.ls_end-self.ls_str+1
self.ls_fov=self.ls_len_mod*self.ls_ang_inc
self.angle_cen=self.ls_fov/2
self.ls_len_mod2=0
self.ls_data=[]

self.drive_state="normal"
self.stopped_time =0.0
self.yaw0 =0.0
self.dtheta = 0.0
self.yaw = 0.0
t = rospy.Time.from_sec(time.time())
self.current_time = t.to_sec()
self.prev_time = self.current_time
self.time_ref = 0.0
self.wl0 = np.array([0.0, -1.0])
self.wr0 = np.array([0.0, 1.0])
```



```
# Lidar simulation parameters
scan_beams: 1152
scan_field_of_view: 6.2831853 #4.71 # radians
scan_range: 12.0
```

Activity 3

The *preprocess_lidar* function can be found below…



```
# Pre-process LiDAR data
def preprocess_lidar(self, ranges):

    data=[]
    data2=[]

    for i in range(self.ls_len_mod):
        if ranges[self.ls_str+i]<=self.safe_distance:
           data.append ([0,i*self.ls_ang_inc-self.angle_cen])
        elif ranges[self.ls_str+i]<=self.max_lidar_range:
            data.append ([ranges[self.ls_str+i],i*self.ls_ang_inc-self.angle_cen])
        else:
            data.append ([self.max_lidar_range,i*self.ls_ang_inc-self.angle_cen])

    k1 = 100
    k2 = 40

    for i in range(k1):

        s_range = 0

        for j in range(k2):
            index1 = int(i*len(ranges)/k1+j)
            if index1 >= len(ranges):
                index1 = index1-len(ranges)

            index2 = int(i*len(ranges)/k1-j)
            if index2 < 0:
                index2= index2 + len(ranges)

            s_range = s_range + min(ranges[index1],self.max_lidar_range)+ min(ranges[index2],self.max_lidar_range)

        data2.append(s_range)


    return np.array(data), np.array(data2)
```

The function operates as follows…

- check each LiDAR scan distance value and compare it to the safe distance for an obstacle

- if the distance value is closer than the safe distance, a 0 is added to a list of distances. This tells us that there is an obstacle at a specified location

- If we are within the safe distance range, we append the distance measurement to another list of distances. This tells us that we have available space at a specified location

- If a distance measurement is beyond the maximum LiDAR scan distance, the max scan distance is appended as a default measurement.

- In order to make this preprocessing less intensive, the total scan is divided into *k1* total values (100 in our case), where each point consists of *k2* individual distance measurements (40 in our case). This allows the program to make decisions based off of average distance values as supposed to exact distance measurements, which substantially decreases computational load on the system.

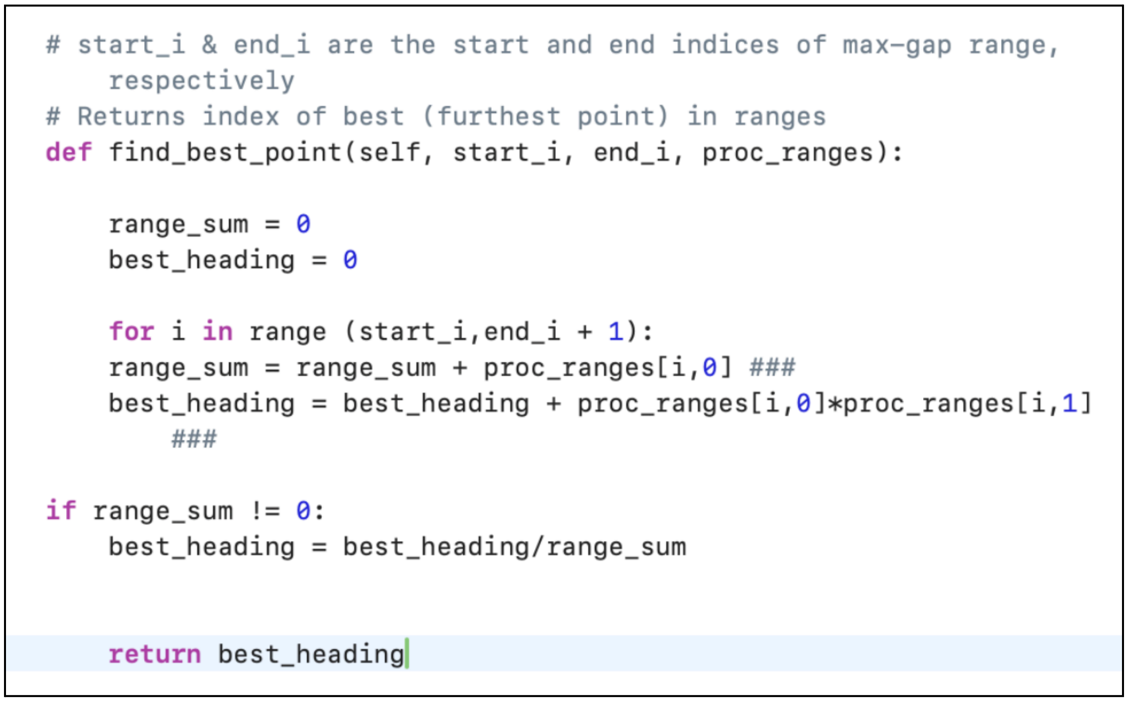
Activity 4

Find below the completed *find_max_gap* function…

```
# Return the start and end indices of the maximum gap in free_space_ranges
def find_max_gap(self, proc_ranges):

    j=0
    str_indx=0;end_indx=0
    str_indx2=0;end_indx2=0

    range_sum = 0
    range_sum_new = 0

    for i in range (self.ls_len_mod):

        if proc_ranges[i,0]!=0:
            if j==0:
                str_indx=i
                range_sum_new = 0
                j=1
            range_sum_new = range_sum_new + proc_ranges[i,0]
            end_indx=i

        if  j==1 and (proc_ranges[i,0]==0 or i==self.ls_len_mod-1):

            j=0

            if  range_sum_new > range_sum:
                    end_indx2= end_indx
                    str_indx2= str_indx
                    range_sum = range_sum_new

    return str_indx2, end_indx2
```

Activity 5

Find below the completed code for *find_best_point…*



```
# start_i & end_i are the start and end indices of max-gap range,
    respectively
# Returns index of best (furthest point) in ranges
def find_best_point(self, start_i, end_i, proc_ranges):

    range_sum = 0
    best_heading = 0

    for i in range (start_i,end_i + 1):
    range_sum = range_sum + proc_ranges[i,0] ###
    best_heading = best_heading + proc_ranges[i,0]*proc_ranges[i,1]
        ###

if range_sum != 0:
    best_heading = best_heading/range_sum


    return best_heading
```

The main issue with the first implementation is that it doesn't take into consideration the angle, and thus we may run into a situation where a further point is identified, but the point is not wide enough for the car to maneuver through. The first approach exclusively considers maximum distance but not the maximum angular width.

The second approach, on the other hand, implements a for loop which is used to sum all of the products of a given points range and angle. This approach also mitigates a division by zero error by checking the denominator, and defaulting any negative values to zero when applicable.

Activity 6

The goal here is to maximize the equation $d_0 = \frac{1}{\sqrt{W^T W}}$

This can be equivalently accomplished by minimizing the denominator, $\sqrt{W^T W}$ which would be the same thing as minimizing $W^T W$ since the square root is a constant scaling factor
This minimization will be solved through the use of Quadratic Programming, which would differentiate this value to find the rate of change of W at various points, and give us a $2W$ term. We can simplify this process by adding a factor of $\frac{1}{2}$ in front of our minimization term to make the derivative easier to deal with, and cancel out this 2.
We thus obtain the final result… $min_w\left(\frac{1}{2}W^T W\right) s.t.\ W^T p_i + 1 \leq 0\, \forall\, i$

Activity 7

The design parameter α is a variable that allows us to balance the two objectives of our optimization problem. One extreme is when $\alpha = 0$. In this case, the first term in *(8)* disappears, and we only have the second term. This would mean that the optimization problem is only focused on the product of the difference between the wall parameter at *k* and at the previous measurement, *k-1*. At the other extreme, $\alpha = 1$, the second term disappears and we are only left with the first term. This means that the optimization problem is solely focused on the wall parameter at step k. When we restrict $0 \leq \alpha \leq 1$ we ensure that neither terms are zero, and thus we can balance both optimization parameters instead of just one.

Activity 8

$$min_{wk}\left(\frac{1}{2}W^T{}_k W_k + (\alpha - 1)W^T{}_{k-1}W_k\right)$$
$$= \frac{1}{2}\alpha W^T{}_k W_k + \frac{1}{2}(1 - \alpha)W^T{}_k W_k - (1 - \alpha)W^T{}_{k-1}W_k + \frac{1}{2}(1 - \alpha)W^T{}_{k-1}W_{k-1}$$
$$= \frac{1}{2}W^T{}_k W_k + (\alpha - 1)W^T{}_{k-1}W_k + \frac{1}{2}(1 - \alpha)W^T{}_{k-1}W_{k-1}$$

To solve the above for W_k we will need to take the derivative, which will in turn make the $W^T{}_{k-1}W_{k-1}$ term negligible

$$= min_{wk}\left(\frac{1}{2}W^T{}_k W_k + (\alpha - 1)W^T{}_{k-1}W_k\right) = (8)$$

Activity 9

$$min_{wk}\left(\frac{1}{2}W^T{}_k W_k + (\alpha - 1)W^T{}_{k-1}W_k\right)$$

We can find the minimum by taking the derivative and setting that to 0…

$$\frac{df(W_k)}{W_k} = \frac{1}{2}2W_k + (\alpha - 1)W_{k-1} = 0$$
$$W_k = (1 - \alpha)W_{k-1}$$

We can then parameterize the right and left barriers as follows…

$Right\ Barrier$: $W^T p + b = 1 \Rightarrow W^T p_i + b - 1 \geq 0$

$Left\,Barrier\colon\ W^T p + b = \ -1 \Rightarrow W^T p_i + b + 1 \leq 0$

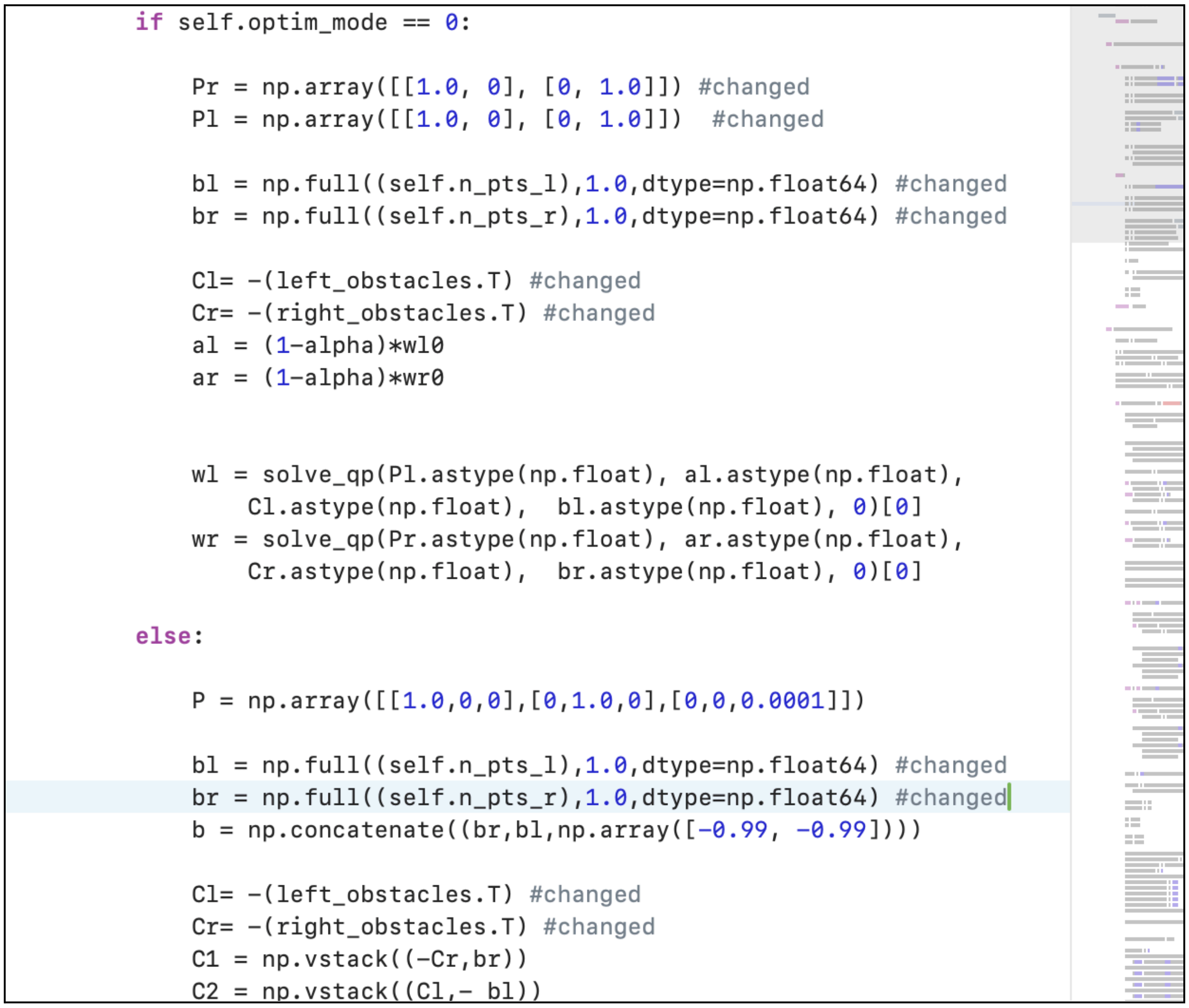
We can then represent the distance between the two barrier lines…
$d = d_r + d_l = \frac{2}{\sqrt{W^T W}}$
We know, from earlier, that we maximize the above through… $min\left(\frac{1}{2}W^T W\right)$

The additional constraint that we have imposed on *b* is to ensure that the origin stays between the two lines that define the walls.

Activity 10
Completed

Activity 11 & 12

Find below the completed code for the *getWalls* function…



```
if self.optim_mode == 0:

    Pr = np.array([[1.0, 0], [0, 1.0]]) #changed
    Pl = np.array([[1.0, 0], [0, 1.0]])  #changed

    bl = np.full((self.n_pts_l),1.0,dtype=np.float64) #changed
    br = np.full((self.n_pts_r),1.0,dtype=np.float64) #changed

    Cl= -(left_obstacles.T) #changed
    Cr= -(right_obstacles.T) #changed
    al = (1-alpha)*wl0
    ar = (1-alpha)*wr0


    wl = solve_qp(Pl.astype(np.float), al.astype(np.float),
        Cl.astype(np.float),  bl.astype(np.float), 0)[0]
    wr = solve_qp(Pr.astype(np.float), ar.astype(np.float),
        Cr.astype(np.float),  br.astype(np.float), 0)[0]

else:

    P = np.array([[1.0,0,0],[0,1.0,0],[0,0,0.0001]])

    bl = np.full((self.n_pts_l),1.0,dtype=np.float64) #changed
    br = np.full((self.n_pts_r),1.0,dtype=np.float64) #changed
    b = np.concatenate((br,bl,np.array([-0.99, -0.99])))

    Cl= -(left_obstacles.T) #changed
    Cr= -(right_obstacles.T) #changed
    C1 = np.vstack((-Cr,br))
    C2 = np.vstack((Cl,- bl))
```

Activity 13

The mathematical derivations can be found below…

$$d = d_r + d_l = \frac{2}{\sqrt{W^T W}} \Rightarrow d_r = \frac{1}{\sqrt{W^T W}},\ d_l = \frac{1}{\sqrt{W^T W}}$$

$$w_r = \frac{W}{b-1}$$

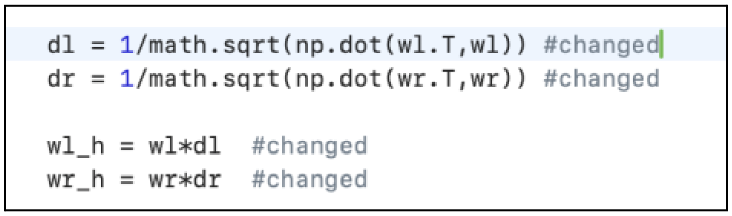
$$w_l = \frac{W}{b+1}$$

$$\widehat{w}_l = \frac{w_l}{||w_l||} = w_l \cdot dl$$

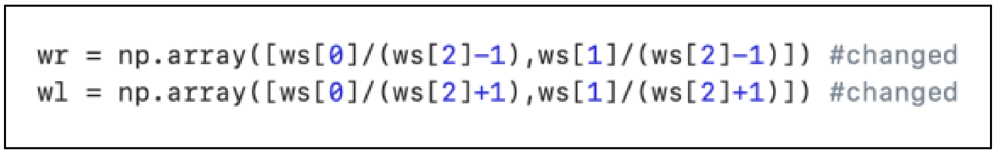
$$\widehat{w}_r = \frac{w_r}{||w_r||} = w_r \cdot dr$$

$$\dot{d}_l = \begin{bmatrix} v_s & 0 \end{bmatrix} \widehat{w}_l$$
$$\dot{d}_r = \begin{bmatrix} v_s & 0 \end{bmatrix} \widehat{w}_r$$
$$cos\left(\alpha_l\right) = [0 \quad -1]\ \widehat{w}_l$$
$$cos\left(\alpha_r\right) = [0 \qquad 1]\ \widehat{w}_r$$

The implementation into the code can be found below…



```
dl = 1/math.sqrt(np.dot(wl.T,wl)) #changed
dr = 1/math.sqrt(np.dot(wr.T,wr)) #changed

wl_h = wl*dl  #changed
wr_h = wr*dr  #changed
```



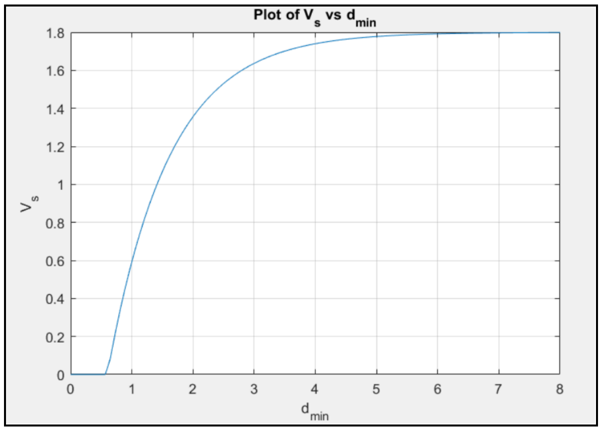
```
wr = np.array([ws[0]/(ws[2]-1),ws[1]/(ws[2]-1)]) #changed
wl = np.array([ws[0]/(ws[2]+1),ws[1]/(ws[2]+1)]) #changed
```

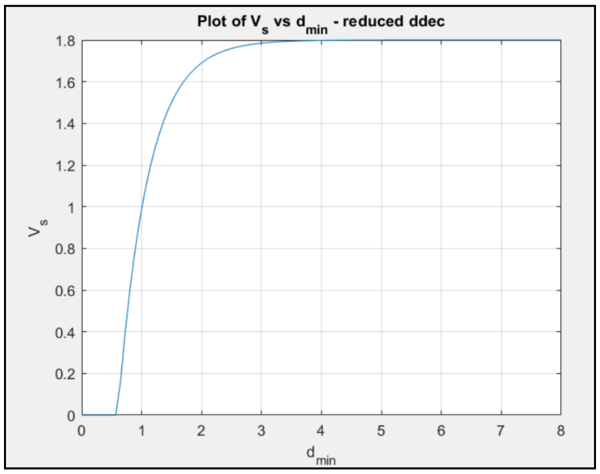
Activity 14

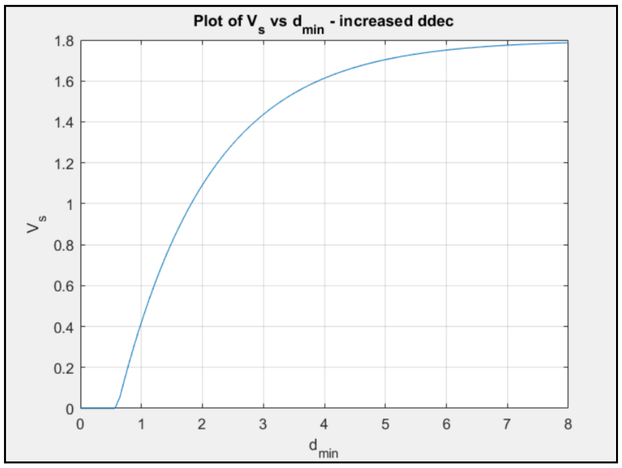
From the plots, we can infer that equation (28) is used to ensure that the vehicle slows down as it approaches an obstacle. We can see a direct proportionality between the vehicles velocity and the value of d_{min} from the plot. This proportionality also seems to be exponential, as you would want the car to react faster if it were approaching at a very high speed as opposed to a very low speed.

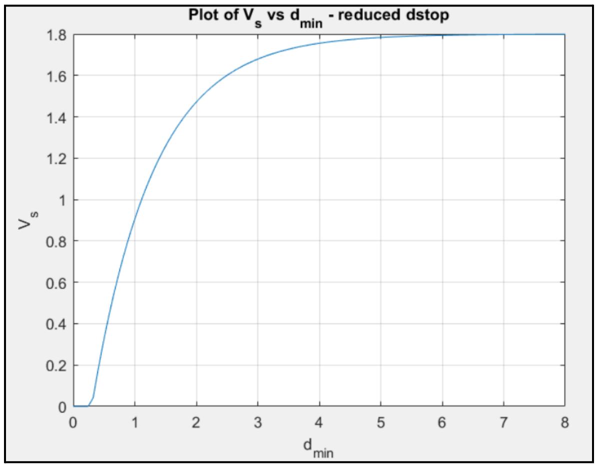
The relevant parameters in the *params.yaml* file are likely *safe_distance, stop_distance_decay, stop_distance, and vehicle_velocity*. The *dmin* parameter detects obstacles and adjusts the vehicle's speed. If an obstacle is detected within the minimum distance, the vehicle slows down. The *dstop* parameter controls the stopping distance. A high *dstop* value causes the vehicle to stop early. The *ddec* parameter determines the deceleration rate as the vehicle approaches *dstop*. Increasing *ddec* increases the deceleration rate.

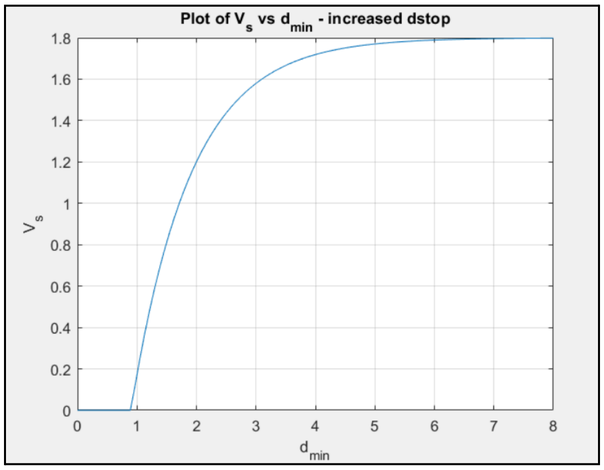
Find the relevant plots below. Additional plots show the vehicle's behavior when *ddec* and *dstop* are adjusted.


Plot of V_s vs d_{min}
V_s
d_{min}

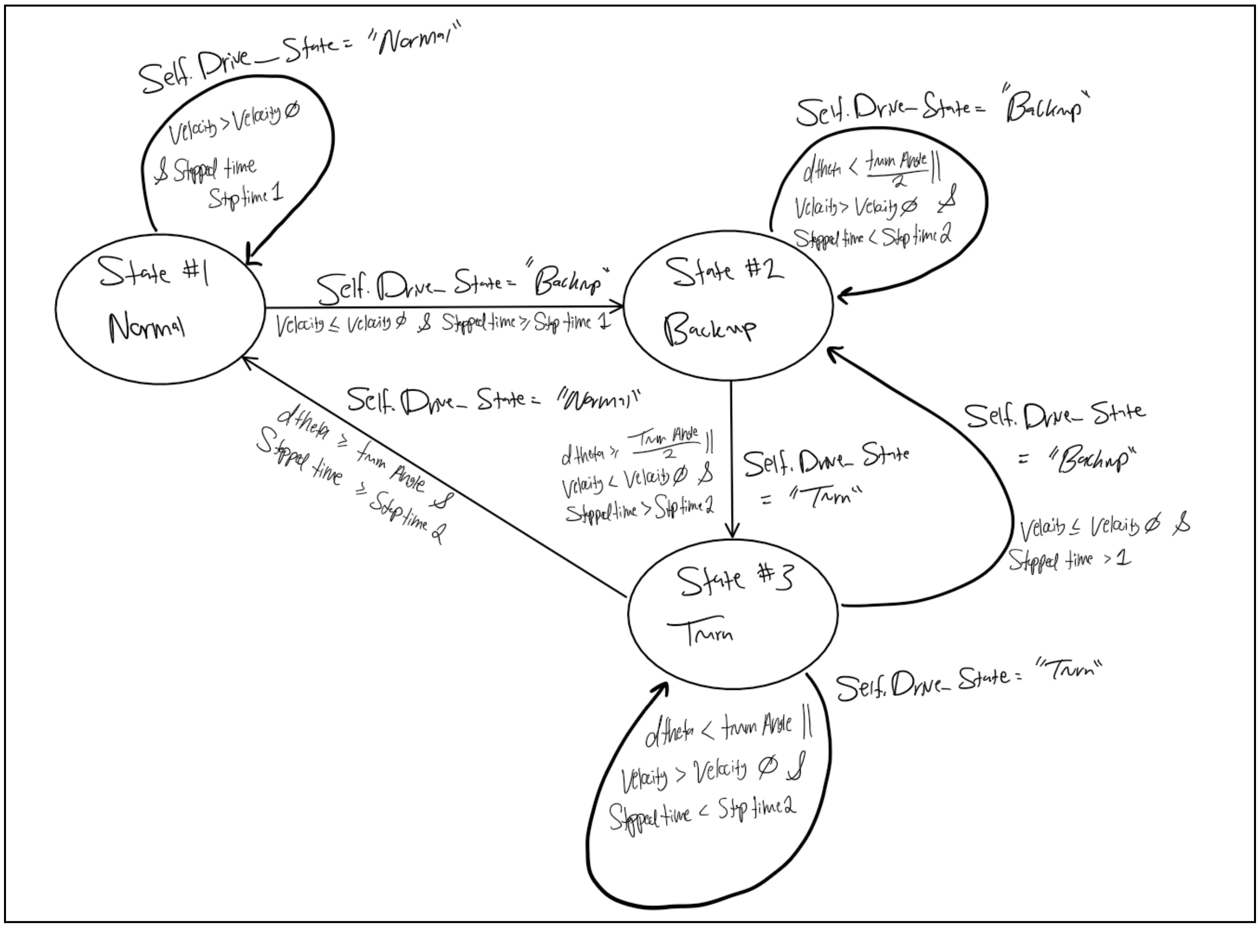

Plot of V_s vs d_{min} - reduced ddec
V_s
d_{min}


Plot of V_s vs d_{min} - increased ddec
V_s
d_{min}


Plot of V_s vs d_{min} - reduced dstop
V_s
d_{min}


Plot of V_s vs d_{min} - increased dstop
V_s
d_{min}

Activity 15
Find below the completed state diagram…


Self. Drive_State = "Normal"
Velocity > Velocity Ø & Stopped time Stop time 1
State #1 Normal
Self. Drive_State = "Backup"
Velocity ≤ Velocity Ø & Stopped time ≥ Stop time 1
State #2 Backup
Self. Drive_State = "Backup"
dtheta < turn Angle / 2 || Velocity > Velocity Ø & Stopped time < Stop time 2
Self. Drive_State = "Normal"
dtheta ≥ turn Angle & Stopped time ≥ Stop time 2
dtheta ≥ Turn Angle / 2 || Velocity < Velocity Ø & Stopped time > Stop time 2
Self. Drive_State = "Turn"
Self. Drive_State = "Backup"
Velocity ≤ Velocity Ø & Stopped time > 1
State #3 Turn
Self. Drive_State = "Turn"
dtheta < turn Angle || Velocity > Velocity Ø & Stopped time < Stop time 2

Activity 16
Mode 0 - Lines are tilted as car drives through the course
Mode 1 - lines are parallel to all objects as car drives through the course

Activity 17
Through changing the parameters, the way the car drove changed. By changing the stop distance, the car would turn either closer or further from obstacles, however if it was increased too much it would stop and refuse to move because it would detect the walls as objects it needed to turn around and there was no path to do so. The safe distance parameter had a similar effect on the car's behavior. Both k_p and k_d had effects on how the car turned. Increasing k_p would increase how quickly (and by extension sharply) the car would turn, however at a point the car could not straighten itself out again properly. To deal with this k_d could be increased to increase the opposing response to the initial turning. The n_pts_r and n_pts_l parameters determined how sensitive the car was to obstacles, and how large of gaps it would drive through. They had to be large enough that they could sense the obstacles and walls, however small enough that the car would not try driving through too small of a gap.

Activity 18
Completed

Activity 19
Overall, the parameters that gave satisfactory self-driving had n_pts_r and n_pts_l at 100, k_p at 4 and k_d at 4.5, and a safe_distance of 1.5. The n_pts parameters were set at 100 because it seemed like a good starting point and they functioned well there. The k_p parameter began at 3 and was increased to 4 to increase the turning speed. The k_d parameter also began at 3 and was slowly increased to 4.5 to account for the increase in k_p. The safe_distance parameter was set to 1.5 because it was reasonable and functioned. The car seemed to function better in mode 0 than it did in mode 1.

Activity 20
The self-driving algorithm which we implemented in this lab seemed to function well for the purposes of this lab. It was able to drive around boxes, drive up a ramp, and avoid walls. The speed at which it could do these things could vary and it could still complete them generally well. The easiest way to improve on this algorithm would be simply better tuning of the parameters. When the parameters of the system change, its response to external input changes, thus the parameters of the system need to be correct to obtain the desired behavior. To improve the algorithm better tuned values for these parameters could be found, instead of using a rough estimate then trial-and-error.