Final Project Report

David Tran (ID: 205-492-874)

February 27, 2021

Introduction

Simulating the behavior of a self-parking car, this program will create an animation that tracks its movement, as it navigates a parking lot randomly populated with other vehicles. The car must obey traffic laws: in other words, it must search for an empty spot and maneuver into it without colliding into any other parked vehicles. To accomplish this task, the use of functions is implemented to compartmentalize the script into various tasks—creating the parking lot, populating the lot with stationary vehicles, drawing the car at a given location, and charting its path from its initial to final location. With this, animations of the car traveling to an empty spot will be generated—one to the top row and one to the bottom row of the lot.

Model and Theory

The self-parking vehicle must follow a trajectory that starts in the white space region to the left of the rows of parking spot. Its initial position must be randomly chosen between $\frac{1}{2}$`w_car` of the top and bottom rows (viz., the whole car must be between the bottom of the top row and the top of the bottom row). The parking spots have dimensions defined by $l_{spot} = 1.25l_c$ and $w_{spot} = 2w_c$. Additionally, there must be at a *minimum* 3 parking spots' width of white space on the left side, but I chose to implement 4 spots' width of white space which is the basis for my formulas. The horizontal length of the lot is `w_lot` and the vertical length is `l_lot`. The centers of the spots that occupy the top row follow equations (1) and (2).

$$x_{center} = 4w_{spot} + w_{spot}k - \frac{w_{spot}}{2}, \quad k \in [1, 10] \tag{1}$$

$$y_{center} = l_{lot} - \frac{l_{spot}}{2} \tag{2}$$

where the factor 4 is the length of white space on the left side and must be at a minimum 3. Similarly, the equation for the centers of the spots in the bottom row abide by equations (3) and (4).

$$x_{center} = 4w_{spot} + w_{spot}(k - 10) - \frac{w_{spot}}{2}, \quad k \in [11, 20] \tag{3}$$

$$y_{center} = \frac{l_{spot}}{2} \tag{4}$$

The path of the car can be broken up into 3 segments: (a) a horizontal line, (b) a quarter circle rotation that aligns the car with its intended spot, and (c) a vertical line. As the car traverses along the path, the car's angle can vary from -90° to 90° with respect to the x axis, depending if it goes to the top or bottom row of spots. Consequently, a formula that incorporates the angle θ that the car makes as it moves along path (b) is necessary.

The vertices of the car at any angle $\theta \in [-90°, 90°]$ are expressed in equations (5), (6), (7), and (8), where they represent the top right, bottom right, top left, and bottom left vertices, respectively.

$$x_{TR} = x_c + \frac{l_{car}}{2}\cos\theta + \frac{w_{car}}{2}\sin\theta, \quad y_{TR} = y_c + \frac{l_{car}}{2}\sin\theta - \frac{w_{car}}{2}\cos\theta \tag{5}$$

$$x_{BR} = x_c - \frac{l_{car}}{2}\cos\theta + \frac{w_{car}}{2}\sin\theta, \quad y_{BR} = y_c - \frac{l_{car}}{2}\sin\theta - \frac{w_{car}}{2}\cos\theta \tag{6}$$

$$x_{TL} = x_c + \frac{l_{car}}{2}\cos\theta - \frac{w_{car}}{2}\sin\theta, \quad y_{TL} = y_c + \frac{l_{car}}{2}\sin\theta + \frac{w_{car}}{2}\cos\theta \tag{7}$$

$$x_{BL} = x_c - \frac{l_{car}}{2}\cos\theta - \frac{w_{car}}{2}\sin\theta, \quad y_{BL} = y_c - \frac{l_{car}}{2}\sin\theta + \frac{w_{car}}{2}\cos\theta \tag{8}$$

x_c and y_c represent the x and y coordinates of the center of the car, respectively. Populating the lot with parked cars, this process is done randomly, with an arbitrary probability $p = 0.5$ of any given spot having a car. Another condition that must be imposed is that at *least* one spot must be empty so that the car can maneuver to it. Additionally, the car will seek out the closest empty spot—if two spots that are in the same column are empty, then the car chooses either spot. To determine the closest empty spot, the distance formula between the points (x_c, y_c) and (x_{spot}, y_{spot}) is utilized:

$$d = \sqrt{(x_c - x_{spot})^2 + (y_c - y_{spot})^2} \tag{9}$$

Another important condition that must be satisfied is that $0 < n_{cars} < 20$; this is to ensure that there is at least one car and 19 cars at most so that the car can find a spot to park in. It is imperative to create a smooth animation that takes into account the lengths of the three paths (a), (b), and (c). To determine the total number of frames that are needed, we use equation (10).

$$\text{total frames} = \text{fps}(t_f - t_0) \tag{10}$$

Then, the lengths of each segment are determined as follows:

$$l_h = x_{spot} - x_c - TR \tag{11}$$

$$l_c = TR\left(\frac{\pi}{2}\right) \tag{12}$$

$$l_v = |y_{spot} - y_c| - TR \tag{13}$$

where l_h, l_c, and l_v denote the horizontal, arclength of the quarter circle, and vertical path lengths, respectively, and TR is the turning radius of the car. Dividing the segments and apportioning the total number of frames, I defined the total number of frames per segment using the formulas below.

$$f_h = \left\lfloor \text{total frames}\left(\frac{l_h}{l_h + l_c + l_v}\right)\right\rfloor \tag{14}$$

$$f_c = \left\lceil \text{total frames}\left(\frac{l_c}{l_h + l_c + l_v}\right)\right\rceil \tag{15}$$

$$f_v = \left\lfloor \text{total frames} \left(\frac{l_v}{l_h + l_c + l_v} \right) \right\rfloor \tag{16}$$

where f_h, f_c, and f_v signify the frames for the horizontal, curved, and vertical paths, respectively. While it may seem arbitrary in the decision that the horizontal and vertical frame counts use floor instead of ceil, I found that if all 3 were the same (i.e. all floor or all ceil), the frame count would not add up to 300. Adjusting the formulas to this mixture yielded 300 frames consistently. Next, the distance traveled per frame needs to be obtained, but for the curved path, the change in θ, $d\theta$, is needed.

$$d_h = \frac{l_h}{f_h} \tag{17}$$

$$d\theta = \frac{\pi/2}{f_c} \tag{18}$$

$$d_v = \frac{l_v}{f_v} \tag{19}$$

One final equation that concludes the model and theory section is the change in x_c and y_c per frame in the curved portion. This change is defined by

$$dx = x_{spot} - TR + TR\sin(k \times d\theta) \tag{20}$$

$$dy = y_{car} \pm (TR - TR\cos(k \times d\theta)) \tag{21}$$

where k iterates from 1 to f_c and the sign of $\pm$ depends if the car is going to the top row (+) or to the bottom row (-).

Methodology and Pseudo-code

For this script, a total of 4 functions will be used: `create_lot`, `create_car`, `populate_spots`, and `find_path`. Respectively, they serve to create the parking lot grid, draw the car at an angle θ with respect to the horizontal axis, populate the empty spots with a random number of parked cars leaving at least one empty spot, and trace the path of the car from its initial location to a final destination in an empty spot.

Algorithm 1 Create the lot using the `create_lot` function with `n_row` spots on both sides

```
Require: Input variables l_lot, w_lot, l_spot, w_spot, and n_row
Require: Output variables x_spots and y_spots
  Perform error checking on the input variables to ensure that they are valid (i.e. numeric values, the lot is the appropriate size and has at least $13 \times n_{row}$ of whitespace, etc.)
  Draw the figure with grid that has dimensions l_lot $+5\times$ w_lot (vertical $\times$ horizontal)
  Draw the actual parking lot with dimensions l_lot $\times$ w_lot
  for $k$ = 1:2$\times$n_row do
    if $k \leq$ n_row then
```

```
        $k$th x_spots element ← $4 \times$ w_spot + (w_spot $\times k$) - w_spot/2
        y_spots elements ← l_lot - l_spot/2
        Draw the x and y arrays that have the coordinates of the vertices of the parking spot centered at the $k$th x_spots and y_spots element
        Generate the $k$th spot on the figure by connecting the vertices
    else
        $k$th x_spots element ← $4 \times$ w_spot + (w_spot $\times (k-10)$) - w_spot/2
        y_spots elements ← l_spot/2
        Draw the x and y arrays that have the coordinates of the vertices of the parking spot centered at the $k$th x_spots and y_spots element
        Generate the $k$th spot on the figure by connecting the vertices
    end if
end for
```

The logic behind this is that a grid will be generated (plus a little extra whitespace to add room to the parking figure window) along with the parking lot with the parameters `l_lot` and `w_lot`. Then, a `for` loop is used that iterates for the total number of rows, which is defined by $2 \times$ `n_row`. It is also important to note that there should be at least 3 parking spots' widths of white space on the left side to allow maneuvering of the vehicle; however, I chose to put 4 spots' widths of white space—hence the factor of 4. The `x_spots` and `y_spots` arrays are populated with the centers of the individual spots that are defined in the pseudocode above. The two rows have the same `x_spots` coordinates, but the `y_spots` values are obviously different. These values vary from script-to-script due to the chosen dimensions of the parking lot. For the 2nd function `create_car`, the following algorithm was implemented.

Algorithm 2 Draw the car centered at (`x_car`, `y_car`) and an angle θ with respect to the x axis in the function `create_car`

```
Require: Input variables x_car, y_car, l_car, w_car, color, and angle
    Perform error checking on the input variables (e.g. ensure that x_car and y_car are positive numbers, check if the variables are numeric, etc.)
    The x and y arrays ← $x$ and $y$ coordinates of the vertices using the input parameters and equations (5), (6), (7), and (8)
    Fill in the box with color
```

Comparatively, this function is much simpler and its sole purpose is to draw the car. The equations that dictate the coordinates of the car's vertices are found using trigonometric relationships. Indeed, the `angle` parameter is necessary because the car's angle changes by $\pm\pi/2$ radians throughout the course of the path. This function is repeatedly called in the main script for all of the frames, as a new car must be generated with its new center position at $(x_{c-1} + \Delta x, y_{c-1} + \Delta y)$. Next, the `populate_spots` function is shown.

Algorithm 3 Fill in the spots with the randomly calculated number of cars in `populate_spots`

```
Require: Input variables x_spots, y_spots, n_cars, parkedColor, l_car, w_car,
    n_row, and filled
    Perform error checking (e.g. if n_cars is an appropriate number, etc.)
    for $k = 1 : 2$n_row do
      if the spot is filled (denoted by a 1 in filled) then
        Call create_car to draw the parked cars at an angle $\pi/2$
      else
        Spot is empty (nothing happens)
      end if
    end for
```

Similar to `create_car`, the `populate_spots` simply draws a parked car. However, it is slightly more sophisticated since it uses a `for` loop to iterate throughout all of the spots and checks the `filled` array which stores either a `0` or `1` to denote whether a spot is empty or occupied, respectively. If the kth value has a `1`, then a car will be drawn centered at that spot's center. To determine the path that the car will take as it moves from its initial, randomly determined spot to the closest empty spot is calculated in `find_path`.

Algorithm 4 Determine all of the x_c and y_c points that the car must follow in the `find_path` function

```
Require: Input variables x_car, y_car, x_spot, y_spot, TR, t0, tf, fps, and top-
    Status
Require: Output variables path_x and path_y
    Perform error checking on the variables (same as the previous functions)
    total frames ← equation (10)
    horizontal length ← equation (11)
    vertical length ← equation (13)
    curve length ← equation (12)
    horizontal change ← equation (17)
    angular change ← equation (18)
    vertical change ← equation (19)
    Frames for each segment ← equations (14), (15), and (16)
    1st position of path_x ← x_car
    1st position of path_y ← y_car
    for $k = 2$ :total frames do
      if $k \leq$ horizontal frames then
        path_x(k) ← path_x(k - 1) + horizontal change
        path_y(k) ← path_y(k-1)
      else if $k >$ horizontal frames AND $k \leq$ horizontal frames + curve
    frames then
        path_x(k) ← equation (20)
```

```
            if spot is in the top row then
                path_y(k) $\leftarrow$ '+' version of equation (21)
            else
                path_y(k) $\leftarrow$ '-' version of equation (21)
            end if
        else if $k >$ horizontal frames + curve frames AND $k \leq$ total frames then
            path_x(k) $\leftarrow$ path_x(k-1)
            if spot is in the top row AND |path_y(k-1) $-$ y_spot| $>$ threshold then
                path_y(k) $\leftarrow$ path_y(k-1) + vertical change
            else if spot is in bottom row AND |path_y(k-1) $-$ y_spot| $>$ threshold then
                path_y(k) $\leftarrow$ path_y(k-1) - vertical change
            end if
            if |path_y(k-1) $-$ y_spot| $\leq$ threshold AND $k \leq$ frames then
                path_y(k) = y_spot
            end if
        end if
    end for
```

The function takes in several parameters that quantify the lengths of each segment, frames per region, and the change in (x_c, y_c) after each frame so that the car travels at a uniform speed. Explanation for use of the equations can be found in the Model and Theory section. I initialized the $k = 1$ positions of the `path` arrays, as MATLAB was generating errors if I did not do so. This explains why the `for` loop iterates from 2 to `total frames`. Then, I divided the loop into 3 regions to account for the different changes in x_c and y_c coordinates. For the horizontal portion, the kth `path_x` value is the same as the previous one plus `horizontal change` that considers the distance. The kth `path_y` value does not vary here. For the curved section, the change in the x_c value is the same regardless if the car of the car's destination. However, y_c is a bit more nuanced: in equation (21), the sign of $\pm$ depends if the vehicle is traveling to the top ('+') or bottom ('-') row. For the vertical path, the x_c coordinate does not vary while y_c varies depending on the parking spot's position. The final `if` statement ensures that the car does not run past the center of the spot.

Algorithm 5 Main script that calls the functions to generate a path in the initial frame and update the car's position in an animation

```
    Open video file and define car dimensions, spot dimensions, lot dimensions, number of rows, colors, initial angles, final angles, turning radius, $t_0$, $t_f$, and frames per second
    Initialize arrays to store spots, filled status, and coordinates of paths
    Initial $x_c \leftarrow 0$
    Initial $y_c \leftarrow$ random number between w_spot + w_car/2 and top of lot - w_spot - w_car/2
    Number of parked cars $\leftarrow$ random number $\in [1, 19]$
    Call create_lot to create the parking lot and populate x_spots and y_spots
    Call create_car to draw the vehicle
```

```
while count $<$ n_cars do
  for $k = 1 : 2\times$n_row do
    if random number $\leq$ n_cars/$(2 \times$ n_row) AND count variable $<$ n_cars AND spot is empty then
      Spot is filled, filled(k) $\leftarrow 1$
      Update count variable by 1
    else if random number $>$ n_cars/$(2 \times$ n_row) AND spot is empty then
      Spot is empty, filled(k) $\leftarrow 0$
    end if
  end for
end while
Call populate_spots to fill the lot with parked cars
Perform error checking to ensure that all delegated spots are filled
for $k = 1 : 2\times$n_row do
  if spot is empty then
    Calculate distance and store it into an array
  else
    Store the distance as an arbitrarily large value
  end if
end for
Obtain the indice of the closest parking spot
if the indice is $< 10$ then
  Spot is in top row
else
  Spot is in bottom row
end if
Define the frames per segment and lengths of each path like in Algorithm 4
Call find_path to obtain the trajectory of the car
Call create_car to draw the car in its final state
Plot path_x and path_y
for $k = 1$ :total frames do
  if $k \leq$ horizontal frames then
    Angle is 0
  else if $k >$ horizontal frames + curved frames then
    Angle is $|\pi/2|$
  else if spot is in top row AND $k \leq$ horizontal frames + curve frames AND $k >$ horizontal frames then
    Angle is ($k$ - horizontal frames)$\times d\theta$
  else if spot is in bottom row AND $k \leq$ horizontal frames + curve frames AND $k >$ horizontal frames then
    Angle is -($k$ - horizontal frames)$\times d\theta$
  end if
  Call create_lot to maintain the lot after each frame
  Call populate_spots to keep the parked cars in the same spot after each frame
```

```
        Call create_car to update the car's position
        Capture and write each frame to the video file
    end for
    Close video file
```

After initializing variables and opening the video file, the `while` loop populates the spots at random and assigns it to `filled`. The quantity `n_cars`/(2×`n_row`) was a factor that I determined gave the spots an equal chance of having a car in it. However, since there is a chance that not all spots will be filled in the first 2×`n_row` iterations, I implemented the `while` loop to keep running this loop until all spots were appropriately filled with `n_cars` cars. The condition in the `elseif` branch seems redundant, but it ensures that the previously occupied spots in `filled` are not overriden. As `n_cars` increases, the probability of any given spot having a parked car increases. Then, `populate_spots` is called to fill the lot with these cars, with `filled` as a parameter that specifies which spots are filled. Error checking is performed on this array to ensure that it meets the required conditions. We once again iterate from 1 to 2×`n_row`. This time, however, we calculate the distance between the initial position of the car's center (x_c, y_c) and its final position using equation (9). However, this calculation is only performed if the spot is empty (denoted by a `0` in `filled`); for instances where the spot is filled, I arbitrarily assigned the distance to a very large value so that it is not accidentally chosen as the closest parking spot. To determine the location of the closest spot, I found the indice of this spot: if this value is > 10, the spot is in the top row (otherwise, it is in the bottom row) and it is assigned a value that denotes the location being in the top (or bottom) row. Similar to the `find_path` function, the frames per segment and lengths of each path are determined using the aforementioned equations. Also, the change in angle is defined here too. I realize that redefining these variables twice is redundant, but I provide more explanation as to why I made this decision in the Discussions and Conclusions section. We then plot the path and final positions of the car using `find_path` and `create_car`. Finally, to generate an animation, we iterate for all of the frames, ensuring that a smooth animation at a constant speed is created. Since the `angle` parameter that is passed to `create_car` varies from segment-to-segment, an inner `if-else` branch is incorporated to check where the current frame is in relation to the path. It also accounts whether the spot in the top or bottom row, as the angle of the curved path varies. Further, this loop maintains the plot, preventing the occupied spots from being wiped out after every frame using `clf`.

Calculations and Results

To first demonstrate that the figure generates a random lot every time with a different number of cars that varies from 1 to 19, figures (1) and (2) are shown. In figure (1), the car chooses a spot in the 3rd position in the bottom row. Obviously, this spot is the closest one to the car's initial (x_c, y_c).

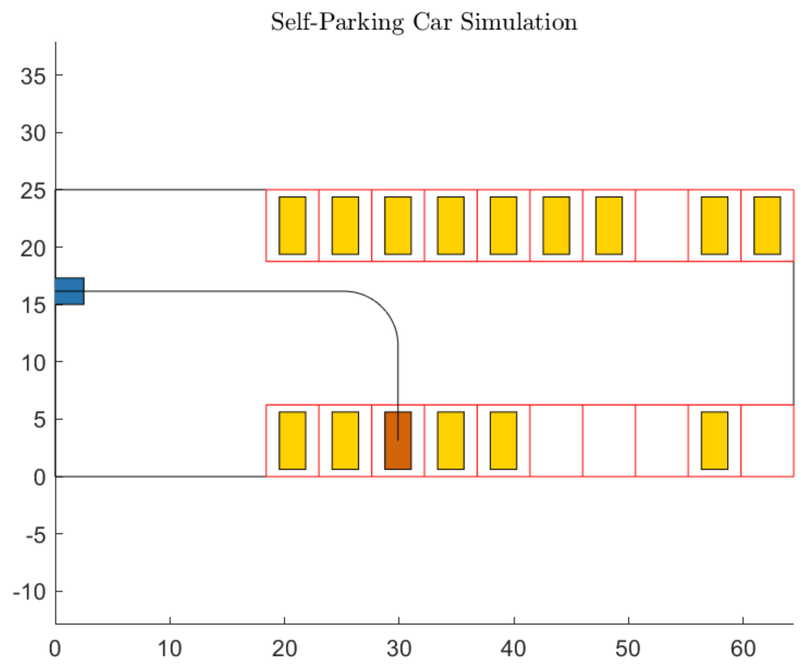

Figure 1: The script determines that the closest spot is the 3rd spot in the bottom row

As shown in figure (2), the car chooses the 1st spot on the top row, with various cars scattered randomly throughout the lot.

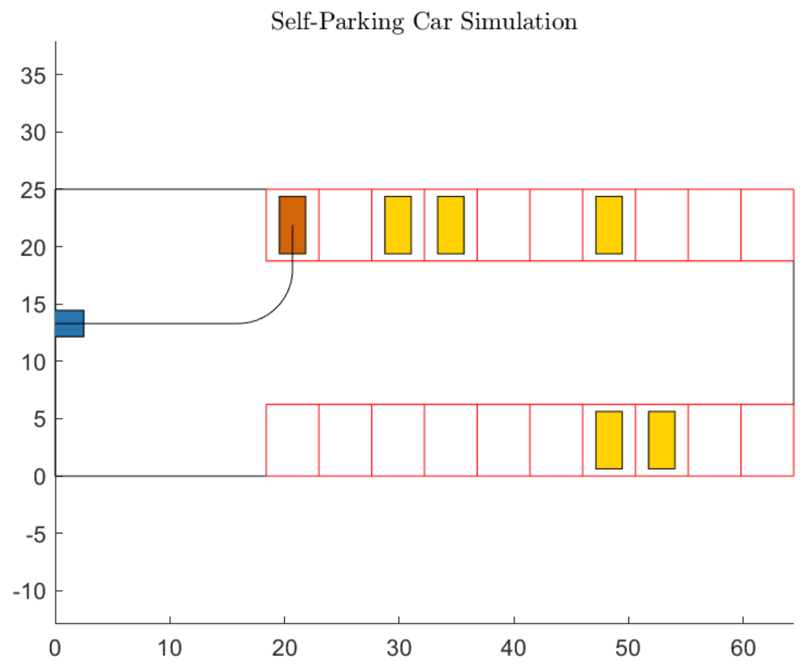

Figure 2: This plot depicts a different number of parked cars compared to figure (1) in other spots and maneuvers into the 1st spot on the top row

Similar to figure (2), the trajectory in figure (3) chooses the 1st spot in its respective row, as the top row's first lot is occupied.

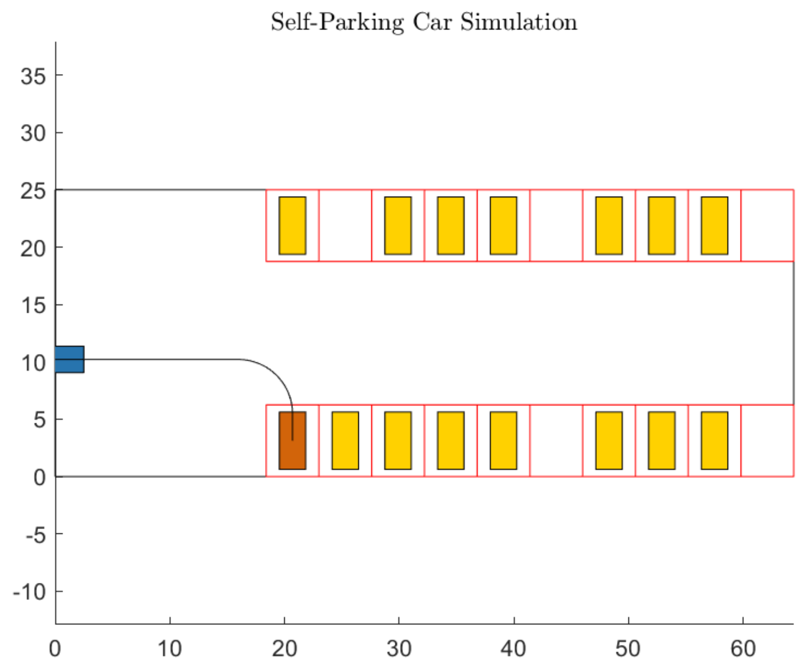

Figure 3: The trajectory is similar to figure (2) in that it chooses the first spot in the bottom row since the first spot in the top row is occupied

Of the six figures, figure (4) has the longest trajectory as shown below and depicts a lot that has the maximum amount of allowed parked vehicles—19.

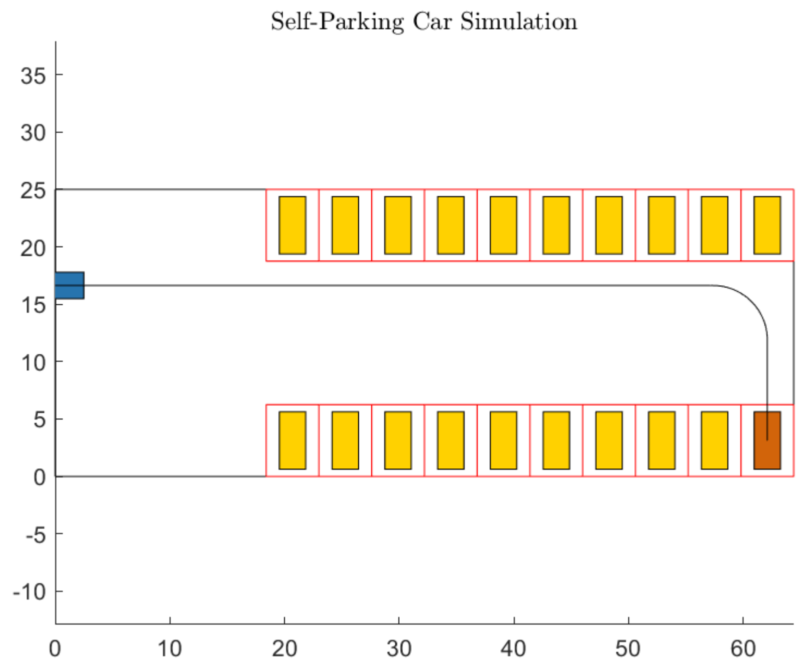

Figure 4: Path of the car is longer than the other figures but moves at a constant speed

Figure (5) illustrates a lot that has no spots in the top row and only three occupied spots in total—two more than the minimum number of spots.

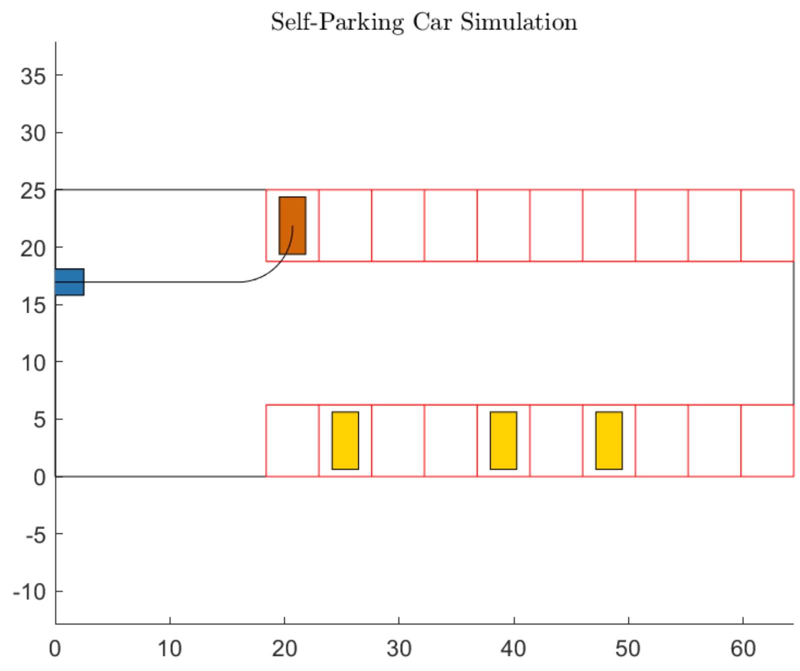

Figure 5: Empty top row with three cars in random positions in the bottom row

Figure (6) shows a path that is between two parked cars, with the spot that it is occupying being the only available one in that row.

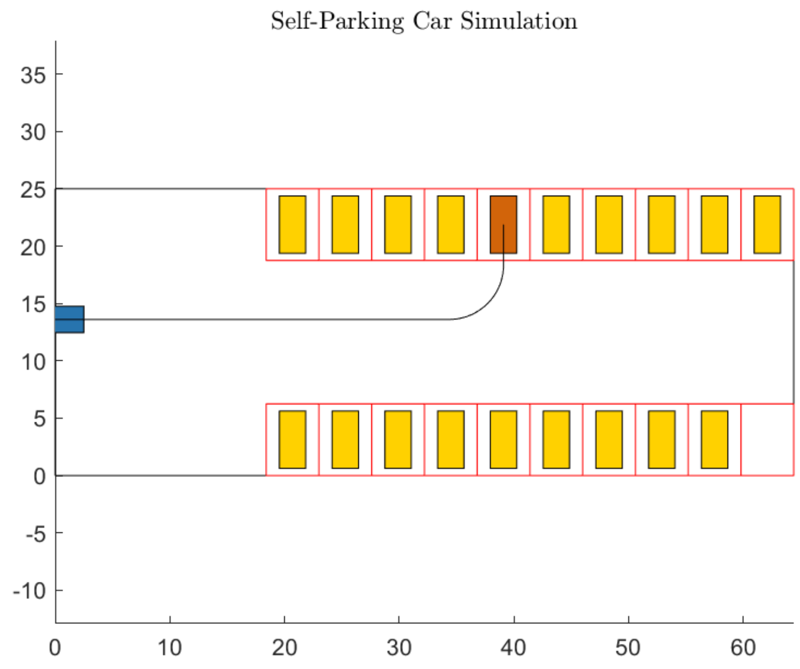

Figure 6: This route shows the car maneuvering in between two UCLA gold cars

Discussions and Conclusions

As technology becomes increasingly pervasive in our lives, the need for a self-parking (and self-driving) vehicle becomes apparent—especially after witnessing the depressing way some Angelenos drive and park. While this script is relatively limited in its applications, it still provides insight into the trajectory and dynamics of a self-parking car. In a real-world scenario, the number of spots may not be known initially and are limited to the capabilities of the sensor. Indeed, pedestrians and unexpected obstacles are future implementations that can enhance the applications of the script—for instance, if the car is going into a path and suddenly finds a shopping cart there, it must reverse safely and find a new empty spot. Nonetheless, this model serves as a strong foundation, upon which future improvements can be implemented to improve its robustness in a vast array of situations. The plots shown in the results section indicate that the car seeks out the closest spot using equation (9). Indeed, the traversed path abides by the required path: a horizontal segment, a quarter circle portion, and a vertical region. The variety of the number of parked cars in differing locations underscore that this script generates these values pseudorandomly as shown in the Calculations and Results section, along with the fact that any of these given scenarios can be encountered in real-life. For instance, figure (1) has 9/10 spots in the top row populated while the bottom row has 8/10 spots occupied, where the spots of these cars are determined randomly. As I worked on my solution, I found that some functions (namely `populate_spots` and `find_path`) needed additional input parameters that would allow certain functions perform tasks such as verifying if a spot is occupied by a car and enhance customizability of the script. However, I did not want to add an excessive amount of parameters as that would make it too "clunky." Another thing to bear in mind is that the scaling of the axes makes the car seem small in its initial position, but it adheres to the correct dimensions everywhere along the path—even if it appears to change size after it passes the curved path. Obviously, future improvements could come in the form of decreasing computational runtime, minimizing the amount of memory used, and augmenting the readability of the code. For instance, I redefined some variables twice in the script like the amount of frames per segment and distance per frame. However, I felt that passing these additional variables to `find_path` would make it overwhelming, as keeping track of all the variables is quite cumbersome—though some variables/arrays like `filled` are necessary, in my opinion at least, to simulate pseudorandom behavior when filling the lot with parked cars. As a result of the apportionment of the frames by section, the car moves at a uniform speed along its path. However, this speed depends on the distance from its initial position to its intended spot. In other words, for longer paths the car moves quicker and for shorter paths the car moves slower. On a final note, I would like to thank the lab sessions by the TAs and lecture videos by Dr. Gao for providing much-needed insight into dissecting this project.