Minimization of parking slots for parallel parking

張問蕖 R11522644 陳珮甄 R11522640 李冠賢 R10522627

December 23, 2022

Problem Description

background

Since the rapid evolution of cars, parking is always the biggest problem for all drivers. We find more and more illegal parking on the road, and we often can't find a parking space. We speculate that the possible reason is that there are too many cars but too few parking spaces and the space that can be divided into parking spaces is limited. Thus, the aim of this report is to minimize parking spaces.

In recent years, parking assistant systems have already become the standard accessory of vehicles on the market. However, because of the development of automatic vehicles, automatic parking system has also attracted more attention all over the world. Though some companies have developed automatic parking systems which can park cars without any interference from drivers, how to optimize the parking trajectory and find the smallest available space for parking is still an important issue.

Therefore, in this project, we'll try to find a trajectory of parallel parking of cars with given parameters and minimize the space available for parking. By using this algorithm, designers who plan parking slots can refer to our calculation results to summarize the minimum length of parking slots currently required in the market, and draw more parking slots in a particular area to optimize the use of space.

literature review

One research from Zips et al. [1] uses two stages to constrain the optimal algorithm. Using the route of the car to leave the parking slot, calculate the route of the car to park. In the first step, the target position is set as the starting point, the switching point is set as the endpoint, and the algorithm is used to plan the path. In the second step, the switching point is set as the starting point, and the parking start position is set as the endpoint. Compared with our project, we only plan the path from the switching point and do not set the target position, as long as the entire vehicle is parked in the parking slot.

In another article from Jing et al. [2], the starting position of the car is fixed, and it can only go backward once, and cannot change direction. Compared with our project, we change the orientation of the car's starting position to find the optimal solution. In addition, we allow the car to change directions and explore the relationship between the number of times of changing directions and the length of the parking slot.

Another article from Li et al. [3] is to fix the starting position of the car and plan the path to find the shortest time for parking the car into the parking slot. Our project is mainly to plan the starting position and path of the car to find the shortest length of the parking slot, regardless of the length of parking time.

Problem Formulation

The optimization problem is formulated as equation (1) with explanation in the following paragraphs.

$$
\begin{aligned}
&\min_{\mathbf{P},L} L \\
\text{subject to } &\Gamma - \Gamma_{\max} \leq 0 \\
&\phi - \phi_{\max} \leq 0 \\
&d_r + d_f - L \leq 0 \\
&-\theta_0 \leq 0 \\
&\theta_0 - \frac{\pi}{2} \leq 0 \\
&F(C) \cap F(S) = \emptyset, \quad \forall P_i \in \mathbf{P}
\end{aligned} \tag{1}
$$

The shape of the car is a rectangle, whose dimension is a tuple $D = (w, d_f, d_r, b, \phi_{\max})$, where w (m) is the width of the car, d_f (m) and d_r (m) are the distance from rear axle center to the front and the tail of the car, respectively, b (m) is the distance between the front and rear axle, and $\phi_{\max}$ (rad) is the maximal steering angle, as the figure 1.

The shape of the parking slot is also a rectangle, one side of its long side is road, and we define the dimension as a tuple $S = (W, L, p)$, where W (m) and L (m) are the width and the length of the parking slot respectively, and p is the upper right corner of the parking slot, by which point the car will pass, as the figure 1.

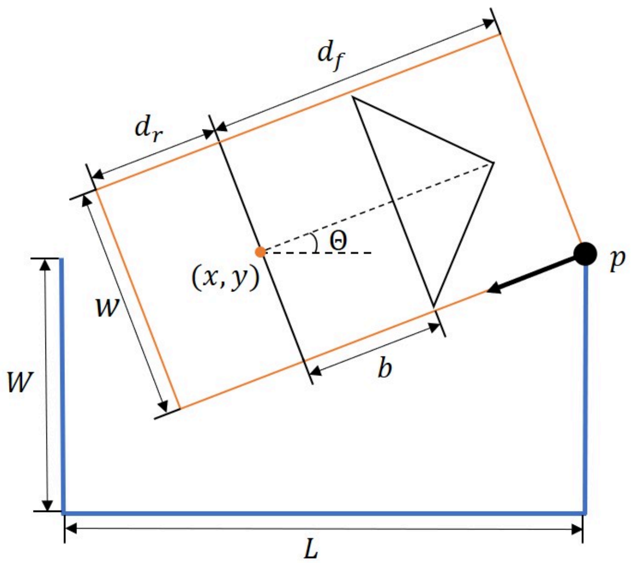


Figure 1: The figure shows the symbols of the car and the parking slot. The rectangle with the orange outline represents the frame of the car, while the lines in blue represents the parking slots.

In our project, we would like to minimize the length of the parking slot within a certain number of direction change. Thus, our objective function is

$$\min L \tag{2}$$

and we set W, w, d_f, d_r, b, ϕ_{max}, Γ_{max}, Δ as our design parameters. Where the Γ_{max} (times) means the maximum of times the car moves forward and backward, referred as "times of direction change" hereinafter, and the Δ is the step length used in the discrete car turning model, we set it as a constant equal to 0.001.

Equation (3) represents the whole trajectory of the parallel parking, **P**, which is also the variable of this project.

$$\begin{aligned} \mathbf{P} &= \langle P_i \rangle = \{P_0, P_1, \ldots, P_n\} \\ P_i &= (x_i, y_i, \theta_i), \quad \forall i \in [0, n] \end{aligned} \tag{3}$$

where x, y are cartesian coordinates of the rear axle center, θ is car heading angle, shown in figure 1. Note that we consider the longer side of the slot, its length, located on the x-axis of the cartesion coordinate used in this project.

The trajectory **P** is composed of the position at each instant. The car position at each instant is related to the previous instant $P_{k+1} = f(P_k)$, where function f, shown as equation (4) [4], consists of the equations of current x-coordinate, y-coordinate, and heading angle of the car.

$$P_{k+1} = \begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k + s_k \cdot \Delta \cdot \cos(\theta_k) \\ y_k + s_k \cdot \Delta \cdot \sin(\theta_k) \\ \theta_k + \frac{s_k \cdot \Delta}{b} \cdot \tan(\phi) \end{bmatrix} = f\left(\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} \right) = f(P_k; \phi, s_k, \Delta) \tag{4}$$

where ϕ is the steering angle at instant k, Δ is the moving step length mentioned above, $s_k \in \{-1, +1\}$ is the moving direction of backward and forward, respectively.

The constraints in the optimization problem are listed and explained as following.

- The vehicle trajectory shouldn't collide with the parking slot.

  $$F(C) \cap F(S) = \emptyset, \quad \forall P_i = (x, y, \theta) \in P \tag{5}$$

  where $F(C)$ is the frame of car, while $F(S)$ is the frame of the parking slot. In figure 1, $F(C)$ is drawn in orange, and $F(S)$ is drawn in blue.

- The radius of curvature shouldn't less than its minimum, which is also referred to that the steering angle should always less than its maximum.

  $$\phi \leq \phi_{\text{max}} \tag{6}$$

- The times of direction change shouldn't be more than the given limitation.

  $$\Gamma \leq \Gamma_{\text{max}} \tag{7}$$

  This constraint is the representation to people's patience and efficiency for parking cars. We wouldn't like to wait someone to park their car for a long time and spend lots of time parking our own cars.

- The dimensions, length and width, of the parking slot should be less than the dimensions of the vehicle, otherwise it's impossible to park the vehicle into the slot.

$$\begin{aligned} L &\geq d_r + d_f \\ W &\geq w \end{aligned} \tag{8}$$

- The angle of initial orientation of the vehicle should always be in the first quadrant. Because the driving convention in Taiwan is right-hand traffic, the parallel parking slots is basically at the right-hand side of driving direction. This constraint represent this situation.

$$0 \leq \theta_0 \leq \frac{\pi}{2} \tag{9}$$

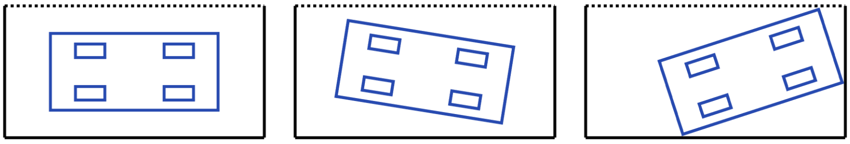
Because of some limitations, we make these assumptions in the project:

1. When the car is steering, the trajectory is always circle on which the middle of rear axle of the car is located.
2. The car will perfectly follow the trajectory models.
3. Except for the car and the parking slot, no other static and dynamic obstacles exist in the environment.

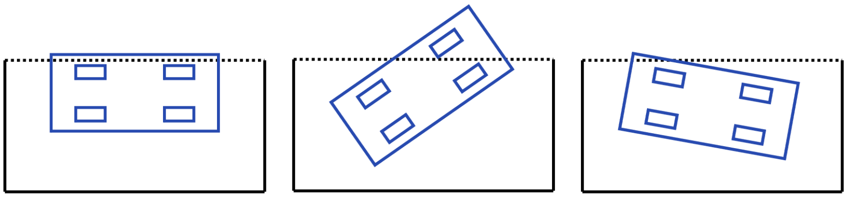
We think that the constraints we set are enough, and the results of optimization for each time are the same, so our formulation is well-bounded.

Problem Solution

To find the solution of this problem, the most difficult part is searching for viable paths satisfying the kinematic properties of the car to park the car into the given slot. For a slot with given dimensions, if the whole projected area of a car from its top is inside the range of the slot, no matter which direction it is heading, it is considered being successfully parked. Figure 2 shows some examples of successfully parked cars and unsuccessfully parked cars. With this definition, next step to solve this problem is to find paths along which the car moves from outside the slot to inside the slot.



(a) Parking succeeded



(b) Parking failed

Figure 2: Example of parking behavior

Next, we have to find a proper way to determine where the car start to move. Since the car can start moving from everywhere outside the slot, there are infinite possible initial configurations (x, y, θ) for a car to start moving from. We should decrease the amount of initial configurations so that the path finding procedure able to be implemented.

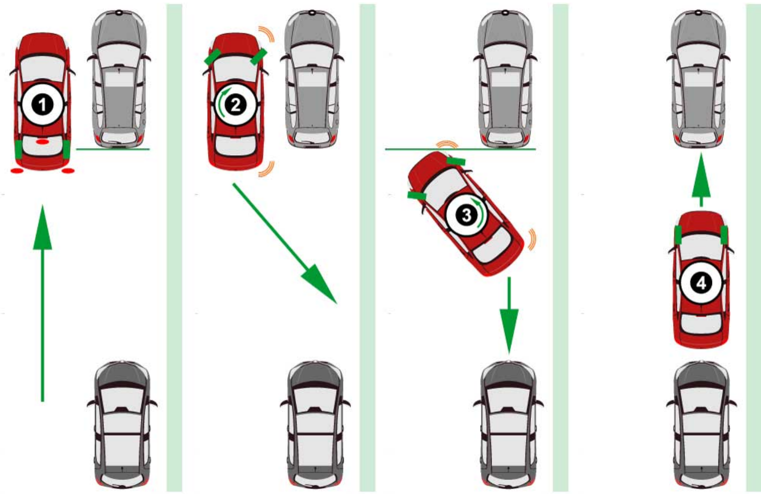
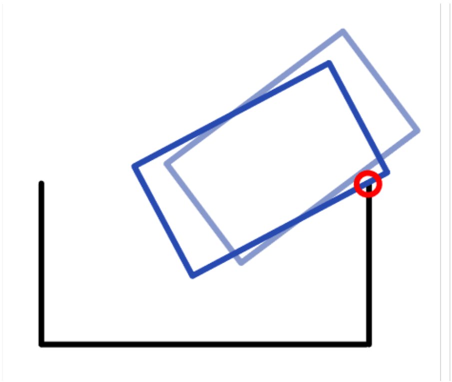
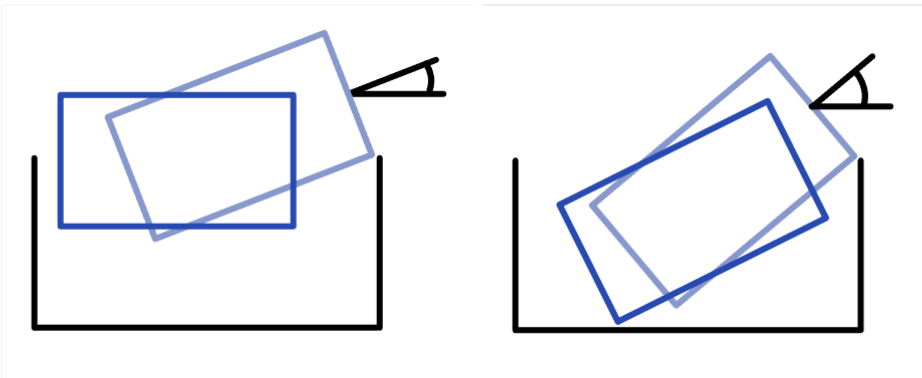


Figure 3: steps of parallel parking [5]



(a) Too early (b) Too late

Figure 4: Example of timing of switching to step 3



(a) Too small (b) Too large

Figure 5: Example of angles when switching to step 3

Parallel parking can be basically separated into 4 steps shown as figure 3. In our own driving experience, step 2 and step 3 are the most important steps when we park a car. If the car switch to step 3 too late or too early, the car will probably hit the slot on its right back corner or right front corner, respectively. Figure 4 shows the situation of these collisions. In addition, the angle between the heading direction of the car and the side of the slot when we switch from step 2 to 3 is also important. If the angle is too small, the back wheel of the car might still not be in the slot so we fail to park. If the angle is too large instead, there might be no more space to move backward in step 3. Figure 5 shows the situation of these conditions. Also models of parallel parking are mainly focus on these two steps [6] [7].

Refering to [4], we decide to make the initial configuration, the place from which the car start moving, located at where the right front corner of the car contacts with the corner of the slot. We consider the car is already at the begining of step 3 in figure 3 at this initial configuration so it will first move backward trying to fit itself into the slot. If it cannot be parked at this step, it will change its moving direction to forward and the turning direction to right, then try to make its left front corner get into the slot. If still not fit in, it'll change the moving direction and turning direction again and do this process repetitively until it's successfully parked.

We think this initial configuration is reasonable based on the following analysis. First, we consider the purpose of doing step 2 and step 3 in figure 3. The purpose of step 2 is to get the car tail into the slot and also reach a location from which the car won't hit its right front corner in step 3. The purpose

Algorithm 1 Parking process

```
function Parking Process(C_0 : (x_0, y_0, θ_0), S : (L, W))
    C ← C_0
    s ← −1                                  ▷ Moving backward first
    φ ← φ_max                               ▷ Always turns with max steering angle
    Γ ← 0                                   ▷ Times of direction change
    while A(C) ⊄ A(S) do                    ▷ Car isn't parked successfully
        NC ← move(C, s, φ)                  ▷ Move a step
        if F(NC) ∩ F(S) ≠ ∅ then            ▷ Car hit the slot
            if Γ == Γ_max then
                return Failed
            end if
            s ← −s, φ ← −φ                  ▷ Change moving direction
            Γ ← Γ + 1
            continue
        end if
        C ← NC
    end while
    return Success
end function
```

of step 3 is to move the car head into the slot. Next, Combining this purpose and our optimization goal, to minimize the length of slot, we think that the right front corner of the car should be as near the corner of slot as possible otherwise the space at the front are wasted.

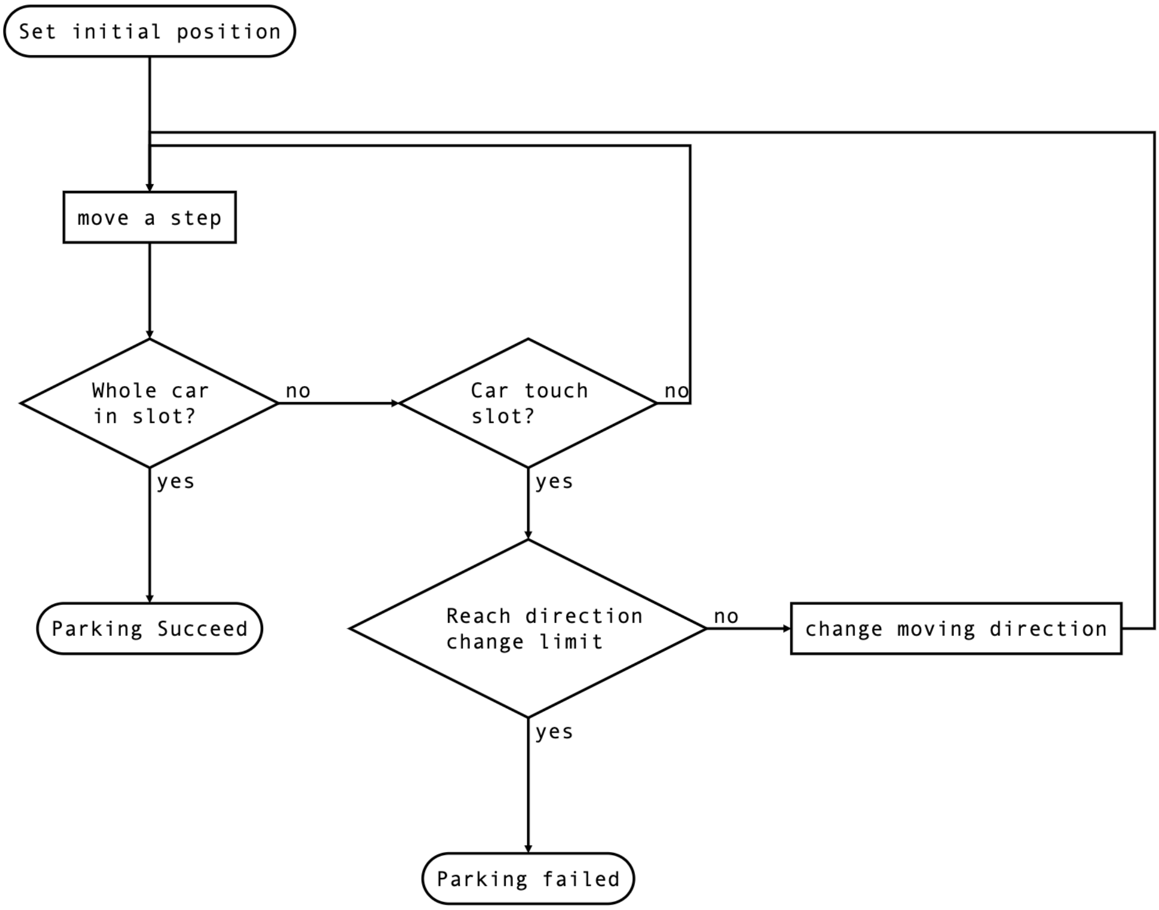


Figure 6: Flowchart of parking process

After deciding the initial configuration, we have to find viable paths of parking the car and minimize the length of slot. Basicallly, since the path is hard to find analytically, we decide to use the similar method as the one in [4] which is simulating the parking process from different initial configurations to determine whether the car can be parked into the slot with given dimensions. The flowchart of our parking process is shown as figure 6, and the pseudo codes are listed as Algorithm 1

Algorithm 2 Minimize length

function Length Minimal(W)
  $L \leftarrow L_0$
  while do True
    $\mathbf{O} \leftarrow \emptyset$
    $\mathbf{C}_0 \leftarrow \{C_0 = (x, y, \theta) | \theta = m \cdot \Delta\theta, m \in \mathbb{N}\}$
    $\mathbf{O} \leftarrow \{\text{Parking Process}(C_0, S : (L, W)) | C_0 \in \mathbf{C}_0\}$
    if All($\mathbf{O}$ $==$ Failed) **then**
      Break
    end if
    $L \leftarrow L - \Delta L$
  end while
  return L
end function

If there exist any viable path to park into a slot with given length l, we subtract it by Δl and run the process again. After few round of iteration, when the car cannot be parked into the slot with the direction change times lower than the limitation we set, the optimum is found. This optimization process is shown as figure 7, and the pseudo codes are listed as Algorithm 2.

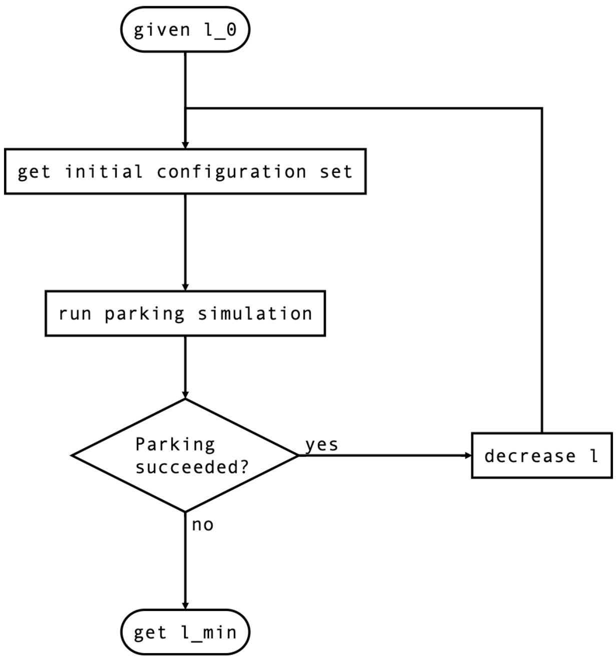


Figure 7: Flowchart of optimization process

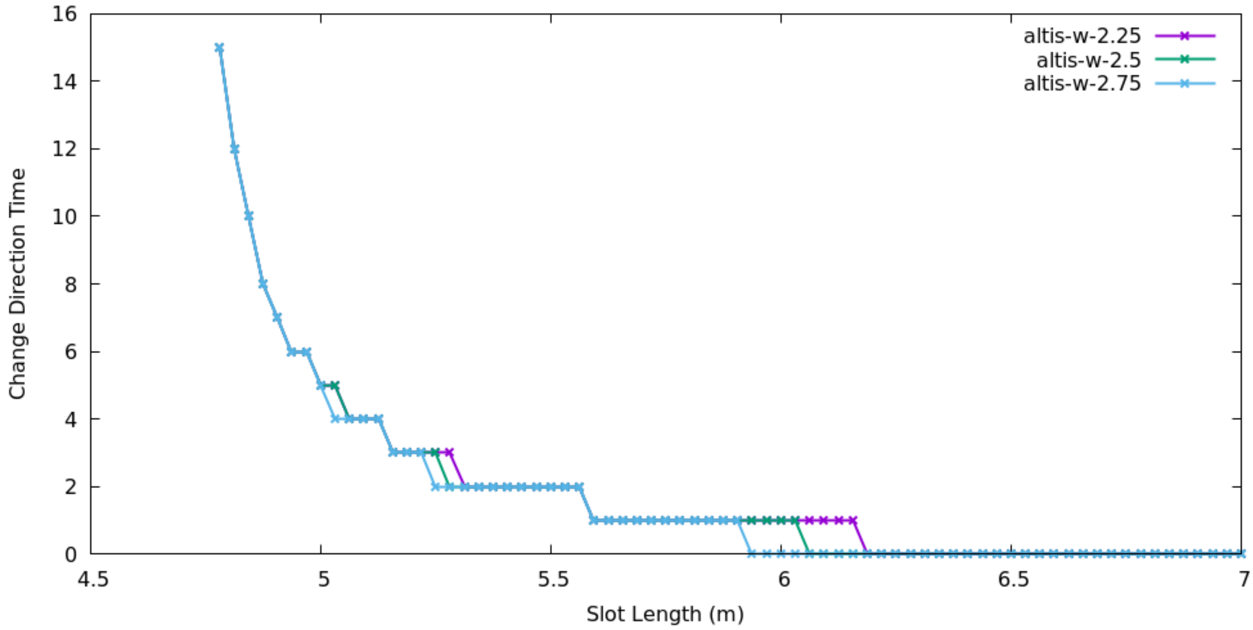


Figure 8: Relation between times of direction change needed and the length of the slot with different slot widths (The title of each line “altis-w-2.5” means that this simulation is conducted under the width of slot W set to 2.5m)

Table 1: Parameters of Toyota Corolla Altis [8]

w	d_f	d_r	b	ϕ_{max}
1.780	3.515	0.815	2.700	28

By ploting out the relation between times of direction change needed and the length of the slot, we can easily found the minimized length of slot with any limitation to times of direction change. The result is ploted and shown as figure 8. Note that in this result, the car parameters are set to ones of Toyota Corolla Altis, the most bought car model in Taiwan, and its parameters are listed in table 1. The max steering angle is calculated by equation (10) which can be found by figure 9, where b is the distance between front and rear wheel axes, r_{min} is the minimal turning curvature radius.

$$\phi_{\text{max}} = \tan^{-1}\left(\frac{b}{r_{\text{min}}}\right) \tag{10}$$

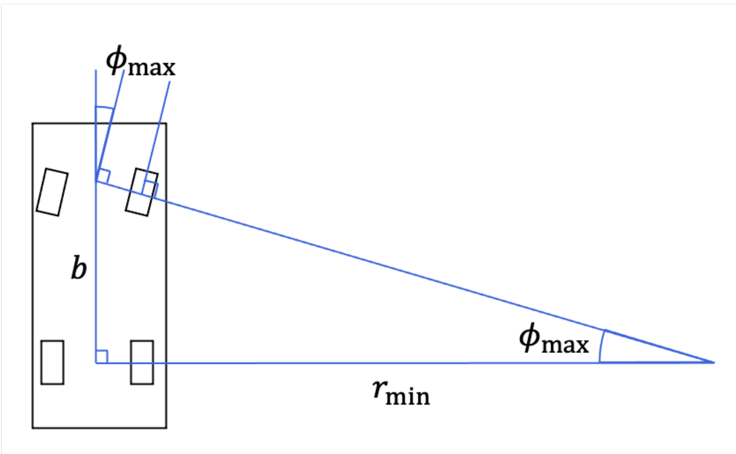


Figure 9: Proof of equation (10)

In the result we can find that the times of direction change increase faster when the length of slot is less than a certain number. Besides, by comparing the results of different setting to width of the slot, we found that for wider slot the times of direction change is much less than the thiner one when the length is large, but after we decrease the length to smaller number the key factor to times of direction change becomes the length of slots, so the lines on figure 8 overlap.

From figure 8, we found that when we set the limit of times of direction change to five, the minimal length of slot is 5 meters, 1.08 times of the length of the car. If we set the limit to one so that only a backward movement and a forward movement are needed, the minimal length of slot becomes 5.6 meters, about 1.2 times of the length of the car.

It is obvious that by this way, the length of slot, which theoretically can be every real number with infinite amount, is discretized into finite options so this solution might not be really “optimal”. However, when people determine the dimensions of real slots, we also choose only discretized numbers so this result is still good enough for situation in reality.

Discussion and Conclusion

In this project, given a car dimension and width of parking slot, we try to find the shortest parking slot, which the car can be parked into, by limiting the number of direction changes in the path. We also change the initial orientation of the car at the moment before it enters the parking slot and try to find the optimal solution by changing the initial state in many ways. We further limited the times of direction change and then optimize the width of the parking slot using the most popular car model in Taiwan to examine whether it's still available to release some space in the common parking spaces.

In our simulation, we set the steering angle to its maximum in the whole process, because the time complexity of simulation will grow exponentially if we add other options, which will cause that we cannot find any viable path within acceptable time of simulation. However, we also discuss the difference between different steering angle set with other parameters are the same, the results are shown as figure 10. From the results, we found that the times of direction change increase when the steering angle are smaller (means that the turning radius is longer), so we assume that with the steering angle set to its maximum we can get the lowest time of direction change and the shortest length of slot.

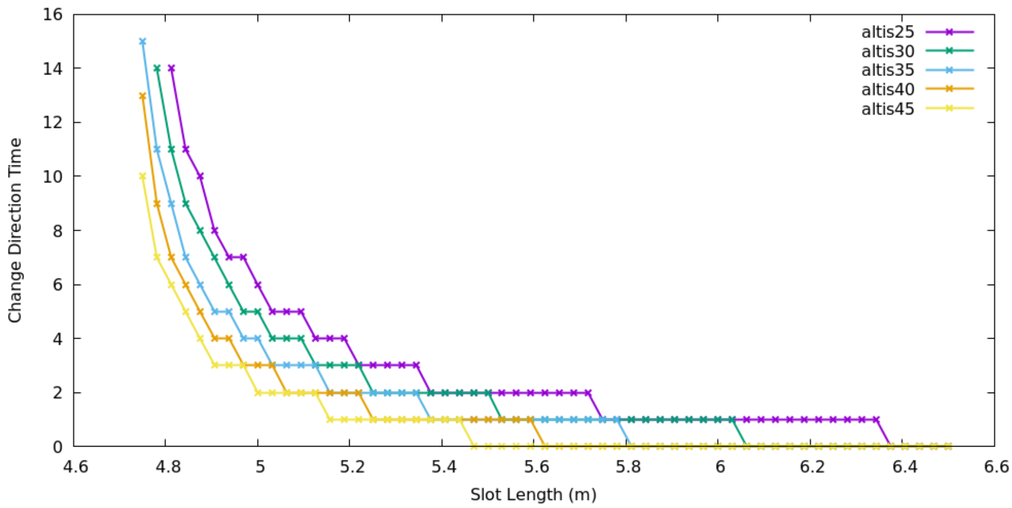


Figure 10: Comparison to times of direction change between different steering angle

Besides, we also do the comparison between different car models try to determine whether the times of direction change is related to the ratio of slot length to car length. From the result shown as figure 11, we find that all the cars can be parked with only one direction change (only move backward and forward once for each) with the slot length 1.3 times of its car length, and most of those cars can be parked into the slot whose length is 1.2 times of the car length within 2 times of direction change.

We also conduct our simulation on Mercedes Benz V300, one of the longest cars in Taiwan, with the result shown as figure 12. It shows that this car can be parked into a 6.2 meters long slot with 3 times of direction change.

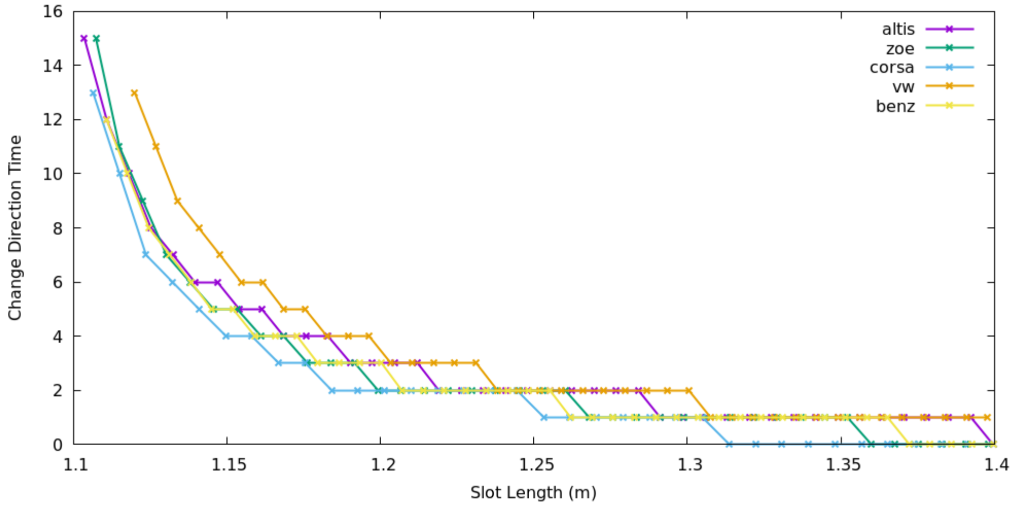


Figure 11: Comparison to times of direction change between different steering angle (Car models are got from [4])

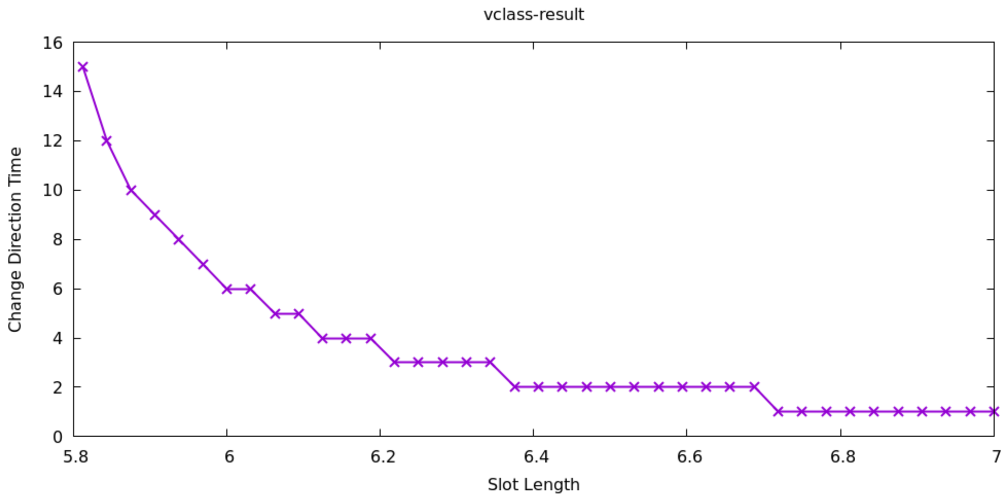


Figure 12: Simulation result of Mercedes Benz V300

From all this results, we found that the proper length of slots for people to park their car within one direction change is about 5.7 meters which is longer than the official length of slots, 5.5 meters in Taiwan. The reason might be that there are some unused space in the slot at the front and back in the real parking situation so the usable parking area are bigger than the official data, and that most of slots in Taiwan are not connected one by one but only two slots are connected so there are more usable space when parking. Finally, there are still several cars which is much longer than the common ones. To make sure most of the drivers including who drives longer cars and who drives normal cars can use parking slots without too much effort, 5.5 meters is a proper length for currently use. If someday the parking space is not enough, connecting all the slots one by one or even shorten their length is still a good way to solve this problem. In this situation, since the times of direction change which can be considered as a kind of hardness of parking cars grows very fast after 5 when shorten the length, we can set the length of slot to a number that most of the cars can be parked into within 3 to 5 times of direction change.

References

[1] Patrik Zips, Martin Bock, and Andreas Kugi. A fast motion planning algorithm for car parking based on static optimization. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2392–2397. IEEE, 2013.

[2] Wei Jing, Dudong Feng, Pinchao Zhang, Shijun Zhang, Sihan Lin, and Bowei Tang. A multi-objective optimization-based path planning method for parallel parking of autonomous vehicle via nonlinear programming. In *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1665–1670. IEEE, 2018.

[3] Bai Li, Kexin Wang, and Zhijiang Shao. Time-optimal maneuver planning in automatic parallel parking using a simultaneous dynamic optimization approach. *IEEE Transactions on Intelligent Transportation Systems*, 17(11):3263–3274, 2016.

[4] Jiri Vlasak, Michal Sojka, and Zdeněk Hanzálek. Parallel parking: Optimal entry and minimum slot dimensions. *arXiv preprint arXiv:2205.02523*, 2022.

[5] OFFICIAL Driving School an Easy Method Company. Parallel parking. Available at `https://officialdrivingschool.com/instruction/parallel-parking/` (2022/12/23).

[6] jeromeawhite. Mathematics of parking. Available at `http://www.talljerome.com/NOLA/parallelparking/index.html` (2022/12/23).

[7] jeromeawhite. Parallel parking optimization. Available at `https://www.geogebra.org/m/bAs4CNMq#material/mhe5dtr8` (2022/12/23).

[8] LTD. HOTAI MOTOR CO. Corolla altis. Available at `https://www.toyota.com.tw/showroom/ALTIS/` (2022/12/23).

Appendix

Codes

lib/car/car_param.ex

```
defmodule Car.CarParam do
  alias Util.Math
  alias Car.CarParam, as: Param

  @msv {1.8, 3.7, 1.0, 2.7, 45}
  @zoe {1.771, 3.427, 0.657, 2.588, 33}
  @corsa {1.532, 3.212, 0.410, 2.343, 32}
  @vw {1.994, 3.308, 1.192, 3.400, 36}
  @benz {1.939, 3.528, 1.016, 2.630, 32}
  @altis {1.780, 3.515, 0.815, 2.700, 28}
  @vclass {1.928, 4.400, 0.970, 3.430, 29}

  @enforce_keys [:w, :df, :dr, :b, :phi_max]
  defstruct [:w, :df, :dr, :b, :phi_max]

  def new(w \\ 1.8, df \\ 3.7, dr \\ 1.0, b \\ 2.7, phi_max \\ 45) do
    %Param{w: w, df: df, dr: dr, b: b, phi_max: Math.d2r(phi_max)}
  end

  def predefined(type) when is_binary(type) do
    case type do
      "msv" -> @msv
      "zoe" -> @zoe
      "corsa" -> @corsa
      "vw" -> @vw
      "benz" -> @benz
      "altis" -> @altis
      "vclass" -> @vclass
    end
    |> (fn x ->
          new(elem(x, 0), elem(x, 1), elem(x, 2), elem(x, 3), elem(x, 4))
        end).()
  end
end
```

lib/car/car_pos.ex

```
alias Util.Vector, as: Point

defmodule Car.CarPos do
  alias Car.CarPos, as: Pos

  # @enforce_keys [:p, :theta, :s, :phi]
  # defstruct [:p, :theta, :s, :phi]
  @enforce_keys [:p, :theta, :s]
  defstruct [:p, :theta, :s]

  def new() do
    Point.new() |> new
  end

  def new(p = %Point{}, theta \\ 0, s \\ 1) do
```

```
    # def new(p = %Point{}, theta \\ 0, s \\ 1, phi \\ 0) do
    # %Pos{p: p, theta: theta, s: s, phi: phi}
    %Pos{p: p, theta: theta, s: s}
  end
end
```

lib/car.ex

```
alias Util.{Vector, Line, Math}
alias Util.Vector, as: Point

defmodule Car do
  alias Car.CarParam, as: Param
  alias Car.CarPos, as: Pos

  @enforce_keys [:param, :pos]
  defstruct [:param, :pos]

  def new do
    new(Param.new(), Pos.new())
  end

  def new(type) when is_binary(type) do
    type
    |> Param.predefined()
    |> new(Pos.new())
  end

  def new(param = %Param{}, pos = %Pos{}) do
    %Car{param: param, pos: pos}
  end

  def vertices(c = %Car{}) do
    ps = c.pos.p
    theta = c.pos.theta
    df = c.param.df
    dr = c.param.dr
    w = c.param.w

    vf = theta |> Vector.unit()
    vl = (theta + Math.d2r(90)) |> Vector.unit() |> Vector.mul(w)

    p1 = vf |> Vector.mul(df) |> Vector.add(ps) |> Vector.add(vl |> Vector.mul
   (0.5))

    p2 =
      vf |> Vector.mul(dr) |> Vector.neg() |> Vector.add(ps) |> Vector.add(vl |>
     Vector.mul(0.5))

    p3 = p2 |> Vector.sub(vl)
    p4 = p1 |> Vector.sub(vl)
    [p1, p2, p3, p4, p1]
  end

  def frames(c = %Car{}) do
    c
    |> vertices
    |> Line.connect_points()
  end
```

```

  def locate_rh(_, _, theta \\ 0)

  def locate_rh(c = %Car{}, s = %Slot{}, theta) do
    c |> Car.locate_rh(s |> Slot.vertices() |> List.last(), theta)
  end

  def locate_rh(c = %Car{}, p = %Point{}, theta) do
    v1 = (theta + Math.d2r(180)) |> Vector.unit() |> Vector.mul(c.param.df)
    v2 = (theta + Math.d2r(90)) |> Vector.unit() |> Vector.mul(c.param.w / 2)
    p = p |> Vector.add(v1) |> Vector.add(v2)
    %{c | pos: %{c.pos | p: p, theta: theta}}
  end

  def move(c = %Car{}, dist \\ 1.0), do: move(c, dist, c.param.phi_max)

  def move(c = %Car{}, dist, phi) when phi <= c.param.phi_max do
    theta = c.pos.theta + dist * c.pos.s * :math.tan(phi) / c.param.b
    v_move = Vector.unit(theta) |> Vector.mul(dist * c.pos.s)
    p = c.pos.p |> Vector.add(v_move)
    %{c | pos: %{c.pos | theta: theta, p: p}}
  end

  def print(c = %Car{}, stream \\ :stdio) do
    c
    |> vertices
    |> Point.print(stream)

    c
  end
end
```

lib/min_slot/config.ex

```
defmodule MinSlot.Config do
  alias MinSlot.Config

  @enforce_keys [:car, :slot]
  defstruct [:car, :slot]

  def new(c = %Car{}, s = %Slot{}, theta \\ 0) do
    c = c |> Car.locate_rh(s, theta)
    %Config{car: c, slot: s}
  end
end

defimpl Inspect, for: MinSlot.Config do
  def inspect(cfg, _ \\ []) do
    dir = if cfg.car.pos.s > 0, do: "Front", else: "Back"
    "#{dir} #{cfg.car.pos.p} #{cfg.car.pos.theta}"
  end
end
```

lib/min_slot/movement.ex

```
defmodule MinSlot.Movement do
  alias MinSlot.{Movement, Config}

```

```
  # ct: times of direction change
  @enforce_keys [:cs, :ce, :s, :phi, :ct]
  # ct: times of direction change
  defstruct [:cs, :ce, :s, :phi, :ct]

  def new(cs = %Config{}, ce = %Config{}, s \\ -1, phi \\ nil, ct \\ 0) do
    phi =
      if phi == nil do
        ce.car.param.phi_max
      end

    h = %Movement{cs: cs, ce: ce, s: s, phi: phi, ct: ct}
    update_in(h.ce.car.pos.s, fn _ -> s end)
  end

  def next(h = %Movement{}) do
    nh = %{h | s: -h.s, phi: -h.phi, ct: h.ct + 1}
    update_in(nh.ce.car.pos.s, fn _ -> nh.s end)
  end

  defp move_to_end(c = %Car{}, s = %Slot{}, dist, phi, record_func) do
    nc = c |> record_func.() |> Car.move(dist, phi)

    cond do
      Slot.inside?(s, nc) -> nc
      Slot.intersect?(s, nc) -> c
      true -> move_to_end(nc, s, dist, phi, record_func)
    end
  end

  def proc(h, ct_max, dist \\ 0.01, record_func)
  def proc(h = %Movement{}, ct_max, _, _) when h.ct > ct_max, do: {:over, h}

  def proc(h = %Movement{}, _, dist, record_func) do
    func = fn car -> move_to_end(car, h.ce.slot, dist, h.phi, record_func) end
    h = update_in(h.ce.car, func)
    # |> IO.inspect
    if Slot.inside?(h.ce.slot, h.ce.car) do
      {:complete, h}
    else
      {:next, h}
    end
  end
end
```

lib/min_slot.ex

```
defmodule MinSlot do
  alias MinSlot.{Config, Movement}
  @enforce_keys [:h, :i, :ct_max]
  defstruct [:h, :i, :ct_max]

  @moduledoc """
  Documentation for `MinSlot`.
  """

  @doc """
  Hello world.

```

```
## Examples

    iex> MinSlot.hello()
    :world

"""

def new(h \\ [], ct_max \\ 0, i \\ []) do
  %MinSlot{h: h, i: i, ct_max: ct_max}
end

def init(car, slot, n_init \\ 0.01, ct_max \\ 0)

def init(%Car{param: %Car.CarParam{dr: dr, df: df, w: cw}}, %Slot{l: sl, w: sw}, _, ct_max)
    when dr + df >= sl or cw > sw,
    do: %MinSlot{h: [], i: [], ct_max: ct_max}

def init(car = %Car{}, slot = %Slot{}, n_init, ct_max) do
  hs =
    thetas(car, slot, n_init)
    |> Enum.map(fn theta -> Config.new(car, slot, theta) end)
    |> Enum.map(fn c -> Movement.new(c, c) end)

  new(hs, ct_max)
end

defp thetas(car, slot, n_part_or_d_deg)

defp thetas(%Car{param: %{dr: dr, df: df}}, %Slot{w: w}, n) when is_integer(n) do
  theta_end =
    (w / (dr + df + 0.001))
    |> :math.asin()

  # init thetas
  0..n |> Enum.map(fn x -> x * theta_end / n end)
end

defp thetas(c = %Car{}, s = %Slot{}, d_deg) when is_float(d_deg) do
  judge_fn = fn theta ->
    c = c |> Car.locate_rh(s, theta)
    c = update_in(c.pos.p.x, &(&1 - 0.0001))
    s |> Slot.intersect?(c)
  end

  theta_recursion(0, judge_fn, d_deg)
end

defp theta_recursion(theta, func, d_theta, out \\ []) do
  case func.(theta) do
    false -> theta_recursion(theta + d_theta, func, d_theta, [theta | out])
    true -> out
  end
end

def run(data, record_func \\ fn car -> car end)
def run(data = %MinSlot{h: []}, _), do: data

def run(data = %MinSlot{h: [h | t]}, record_func) do
```

```
    case Movement.proc(h, data.ct_max, record_func) do
      {:complete, h} ->
        %{
          data
          | h: t,
            ct_max: min(h.ct, data.ct_max),
            i: if(data.ct_max > h.ct, do: [{h.cs, h.ce}], else: [{h.cs, h.ce} | data.i])
        }

      {:next, h} ->
        %{data | h: [Movement.next(h) | t]}

      {:over, _} ->
        %{data | h: t}
    end
    |> run(record_func)
  end

  def min_ct(data = %MinSlot{}) do
    res = data |> run
    {!(res.i == []), res.ct_max, Enum.map(res.i, &elem(&1, 0))}
  end

  def parkable(data = %MinSlot{h: [], i: []}), do: {false, data.ct_max, []}

  def parkable(data = %MinSlot{h: [h | t]}) do
    case Movement.proc(h, data.ct_max, fn c -> c end) do
      {:complete, h} ->
        {true, h.ct, [h.cs]}

      {:next, h} ->
        %{data | h: [Movement.next(h) | t]}
        |> parkable

      {:over, _} ->
        %{data | h: t}
        |> parkable
    end
  end

  def record_proc(cs = %Config{}, filename) when is_binary(filename) do
    {:ok, path_file} =
      Util.Path.append_name(filename, "path")
      |> File.open([:write])

    {:ok, frame_file} =
      Util.Path.append_name(filename, "frame")
      |> File.open([:write])

    record_func = fn car ->
      IO.puts(path_file, "#{car.pos.p.x} #{car.pos.p.y}")
      car |> Car.print(frame_file)
    end

    cs.slot |> Slot.print(frame_file)

    [Movement.new(cs, cs)]
    |> new(1000)
    |> run(record_func)
```

```

    File.close(path_file)
    File.close(frame_file)
  end
end
```

lib/slot.ex

```
defmodule Slot do
  alias Util.{Vector, Line}
  alias Util.Vector, as: Point

  @enforce_keys [:w, :l]
  defstruct [:w, :l]

  def new(l \\ 6.0, w \\ 3.0), do: %Slot{w: w, l: l}

  def vertices(%Slot{w: w, l: l}) do
    [{0, w}, {0, 0}, {l, 0}, {l, w}]
    |> Enum.map(fn p -> Point.new(p) end)
  end

  def frames(s = %Slot{}) do
    s
    |> vertices
    |> Line.connect_points()
  end

  def intersect?(s = %Slot{}, c = %Car{}) do
    s |> intersect?(Car.frames(c))
  end

  def intersect?(s = %Slot{}, lines) do
    s
    |> frames
    |> Line.intersect?(lines)
  end

  def inside?(s = %Slot{}, c = %Car{}) do
    c
    |> Car.vertices()
    |> Enum.all?(&inside?(s, &1))
  end

  def inside?(s = %Slot{}, p = %Point{}) do
    s
    |> vertices
    |> (fn x -> x ++ [List.first(x)] end).()
    |> Line.connect_points()
    |> same_side?(p)
  end

  defp same_side?(lines, point, greater \\ nil)
  defp same_side?([], _, _), do: true

  defp same_side?([l = %Line{} | t], p = %Point{}, greater) do
    vp = p |> Point.sub(l.ps)
    side = l |> Line.vec() |> Vector.cross(vp)
    out = side > 0
```

```

    cond do
      side == 0 -> false
      greater == nil -> same_side?(t, p, out)
      out == greater -> same_side?(t, p, out)
      true -> false
    end
  end

  def print(s = %Slot{}, stream \\ :stdio) do
    s
    |> vertices
    |> Point.print(stream)

    s
  end
end
```

lib/util/line.ex

```
defmodule Util.Line do
  alias Util.{Vector, Line}
  alias Util.Vector, as: Point

  @enforce_keys [:ps, :pe]
  defstruct [:ps, :pe]

  def new({ps, pe}), do: new(ps, pe)

  def new(ps = %Vector{}, pe = %Vector{} \\ %Vector{x: 0, y: 0}) do
    %Line{ps: ps, pe: pe}
  end

  def vec(%Line{ps: ps, pe: pe}) do
    Vector.sub(pe, ps)
  end

  def intersect?(l1 = %Line{ps: ps1, pe: pe1}, l2 = %Line{ps: ps2, pe: pe2}) do
    v1 = l1 |> Line.vec()
    v2 = l2 |> Line.vec()
    c1s = v1 |> Vector.cross(Vector.sub(ps2, ps1))
    c1e = v1 |> Vector.cross(Vector.sub(pe2, ps1))
    c2s = v2 |> Vector.cross(Vector.sub(ps1, ps2))
    c2e = v2 |> Vector.cross(Vector.sub(pe1, ps2))

    (c1s * c1e < 0 && c2s * c2e < 0) ||
      Point.point_on_line?(l1.ps, l2) ||
      Point.point_on_line?(l1.pe, l2) ||
      Point.point_on_line?(l2.ps, l1) ||
      Point.point_on_line?(l2.pe, l1)
  end

  def intersect?(_, []), do: false

  def intersect?(l = %Line{}, [h = %Line{} | t]) do
    intersect?(l, h) || intersect?(l, t)
  end

  def intersect?([], _), do: false
```

```

  def intersect?([h | t], g = [_ | _]) do
    intersect?(h, g) || intersect?(t, g)
  end

  def connect_points([]), do: []

  def connect_points(points = [_ | t]) do
    points
    |> Enum.drop(-1)
    |> Enum.zip_reduce(t, [], fn x, y, acc -> [Line.new(x, y) | acc] end)
    |> Enum.reverse()
  end
end

defimpl Inspect, for: Util.Line do
  def inspect(l, _ \\ []) do
    "Line{#{Inspect.Util.Vector.inspect(l.ps)} -> #{Inspect.Util.Vector.inspect(
   l.pe)}}"
  end
end
```

lib/util/math.ex

```
defmodule Util.Math do
  def r2d(rad) do
    rad * 180 / :math.pi()
  end

  def d2r(deg) do
    deg * :math.pi() / 180
  end
end
```

lib/util/path.ex

```
defmodule Util.Path do
  def append_name(filename, append_text) do
    [Path.rootname(filename), "-", append_text, Path.extname(filename)]
    |> Enum.join()
  end

  def write_file(data, filename, func \\ fn data, _ -> data end) when is_binary(
   filename) do
    file =
      case filename do
        "" ->
          :stdio

        _ ->
          {:ok, file} = filename |> File.open([:write])
          file
      end

    data
    |> func.(file)

    File.close(file)
```

```
  end
end
```

lib/util/vector.ex

```
defmodule Util.Vector do
  alias Util.{Vector, Line}

  @enforce_keys [:x, :y]
  defstruct [:x, :y]

  def new({x, y}), do: new(x, y)
  def new(x \\ 0, y \\ 0), do: %Vector{x: x, y: y}

  def unit(rad \\ 0) do
    %Vector{x: :math.cos(rad), y: :math.sin(rad)}
  end

  def rotate(%Vector{x: x, y: y}, rad \\ 0) do
    c = :math.cos(rad)
    s = :math.sin(rad)
    %Vector{x: x * c - y * s, y: x * s + y * c}
  end

  def dot(%Vector{x: x1, y: y1}, %Vector{x: x2, y: y2}) do
    x1 * x2 + y1 * y2
  end

  def cross(%Vector{x: x1, y: y1}, %Vector{x: x2, y: y2}) do
    x1 * y2 - y1 * x2
  end

  def add(%Vector{x: x1, y: y1}, %Vector{x: x2, y: y2}) do
    %Vector{x: x1 + x2, y: y1 + y2}
  end

  def sub(%Vector{x: x1, y: y1}, %Vector{x: x2, y: y2}) do
    %Vector{x: x1 - x2, y: y1 - y2}
  end

  def mul(%Vector{x: x, y: y}, t) do
    %Vector{x: t * x, y: t * y}
  end

  def neg(%Vector{x: x, y: y}) do
    %Vector{x: -x, y: -y}
  end

  def abs(%Vector{x: x, y: y}) do
    :math.sqrt(x * x + y * y)
  end

  def point_on_line?(p = %Vector{}, l = %Line{}) do
    vs = l.ps |> sub(p)
    ve = l.pe |> sub(p)
    cross(vs, ve) == 0 && dot(vs, ve) <= 0
  end

  def print(vecs, stream \\ :stdio)
```

```elixir

  def print(v = %Vector{}, stream) do
    IO.puts(stream, "#{v.x}\t#{v.y}")
  end

  def print([], stream), do: IO.puts(stream, "")

  def print([v = %Vector{} | t], stream) do
    print(v, stream)
    print(t, stream)
  end
end

defimpl Inspect, for: Util.Vector do
  def inspect(v, _ \\ []) do
    "(#{v.x}, #{v.y})"
  end
end

defimpl String.Chars, for: Util.Vector do
  def to_string(v) do
    "(#{v.x}, #{v.y})"
  end
end
```

main.exs

```elixir
defmodule Main do
  def test do
    slot = Slot.new(4.5, 2) |> Slot.print
    res = Car.new("zoe")
    |> MinSlot.init(slot, 0.001, 100)
    |> MinSlot.run
    res
    |> Map.fetch!(:i)
    |> Enum.map(fn {x, y} ->
      x.car |> Car.print
      y.car |> Car.print
    end)
    IO.puts(:stderr, "ctmax: #{res.ct_max}")
  end

  def min_length(car, config \\ %{}, filename \\ "") do
    car
    |> get_ctmax(config)
    |> List.keysort(0)
    |> res_print(filename)
  end

  def res_print(data, filename \\ nil)
  def res_print(data, nil), do: data
  def res_print(data, filename) when is_binary(filename) do
    write_result = fn data, file ->
      data
      |> Enum.map(&("#{elem(&1, 0)}\t#{elem(&1, 1)}"))
      |> Enum.map(fn str -> IO.puts(file, str) end)
      data
    end

```

```
  write_case = fn data, file ->
    data
    |> Enum.map(fn d ->
      ["l = #{elem(d, 0)}",
        elem(d, 2)
        |> Enum.map(fn cs ->
          "#{cs.car.pos.theta}"
        end)
      ]
      |> List.flatten
      |> Enum.join("\n")
    end)
    |> Enum.map(fn str -> IO.puts(file, str) end)
  end

  result_filename = filename
                    |> Util.Path.append_name("result")
  case_filename = filename
                  |> Util.Path.append_name("case")
  Util.Path.write_file(data, result_filename, write_result)
  Util.Path.write_file(data, case_filename, write_case)

  data
end
def res_print(data, _), do: res_print(data)

@doc """
Get the max change direction by decreasing the length of slot

## Parameters

- car: A %Car{} structure
- parkable_only: [Bool] if true, the result will only correct for the
  available length
- l: [float] the starting length
- config: [Map] optional configuration
  - l_end: the end of length
  - d_l: difference between each length
  - w: the width of the slot
  - ct_max: the maximum direction changing times
  - dist: the distance of each car movement
  - concur_max: max number of parallel processes
"""

def get_ctmax(car, config \\ %{})
def get_ctmax(car = %Car{}, config = %{}) do
  # Config_fetch
  parkable_only = config |> Map.get(:parkable_only, false)
  l = config |> Map.get(:l, 6)
  d_l = config |> Map.get(:d_l, 0.01)
  w = config |> Map.get(:w, 2)
  ct_max = config |> Map.get(:ct_max, 10)
  dist = config |> Map.get(:dist, 0.001)
  concur_max = config |> Map.get(:concur_max, 1)

  car_length = trunc(car.param.dr + car.param.df)
  l_end = config |> Map.get(:l_end, car_length)
          |> (fn x -> if x < car_length, do: car_length, else: x end).()

  IO.puts(:stderr, "w = #{w} start")
```

```

  #Process function selection
  proc_func = if parkable_only,
    do: &MinSlot.parkable/1,
    else: &MinSlot.min_ct/1

  l_proc =  fn l ->
    MinSlot.init(car, Slot.new(l, w), dist, ct_max) |> proc_func.()
  end

  (l - l_end) / d_l
  |> trunc
  |> (fn nl -> 0..nl end).()
  |> Enum.map(fn x -> l - d_l * x end) # length list
  |> Enum.chunk_every(concur_max)
  |> ctmax_chunk_concur(l_proc)
end

@doc """
## Parameters

- l_chunks: list containing chunks of lengths. e.g. [[6,5,4], [3,2,1]]
- func: function take length as the only argument and return the result
"""
def ctmax_chunk_concur(l_chunks, func, out \\ [])
def ctmax_chunk_concur([], _, out), do: out
def ctmax_chunk_concur([chunk | t], l_func, out) do
  s = self()
  res = chunk
    # create processes for every length in the chunk
    |> Enum.map(fn l ->
      spawn(fn ->
        IO.puts(:stderr, "l = #{l} start")
        # do the length process
        res = l_func.(l)
        # send back the result
        send(s, {res, l})
        IO.puts(:stderr, "l = #{l} end")
      end)
    end)
    |> Enum.map(fn _ -> # collect results
      receive do
        {{successed, ct, i}, l} -> {successed, {l, ct, i}}
      end
    end)

  out = (res
        # filter out failed case
        |> Enum.filter(&(elem(&1, 0)))
        # fetch {l, ct_max} data
        |> Enum.map(&(elem(&1, 1)))
  ) ++ out

  # stop if there are any failed case in chunk
  case Enum.all?(res, &(elem(&1, 0))) do
    true -> ctmax_chunk_concur(t, l_func, out)
    false -> out
  end
end

```

```
  def make_record(car_name, sl, sw, theta, record_name) do
    Car.new(car_name)
    |> MinSlot.Config.new(Slot.new(sl, sw), theta)
    |> MinSlot.record_proc("plot/data/#{record_name}.txt")
  end

  defmodule CLI do
    def run do
    end

    def help do
    end
  end
end


car_length = fn car_name -> Car.CarParam.predefined(car_name) |> (&(&1.dr + &1.
    df)).() end

run_width_muls = fn car_name, width_muls ->
  width_muls
  |> Enum.map(fn m ->
    {%{l: car_length.(car_name) * 1.4 |> Float.floor(1), w: Car.CarParam.
    predefined(car_name).w * m, ct_max: 10, d_l: 0.03125, concur_max: 8} ,
      m}
  end)
  |> Enum.map(fn {cfg, m} -> {Main.min_length(Car.new(car_name), cfg, nil), m}
    end)
  # Generalize car length
  |> Enum.map(fn {data, m} ->
    updated_data = data
    |> Enum.map(fn d ->
      val = elem(d, 0) / car_length.(car_name)
      put_elem(d, 0, val)
    end)
    {updated_data, m}
  end)
  |> Enum.map(fn {data, m} ->
    Main.res_print(data, "plot/data/#{car_name}-w-#{m}-generalized.txt")
  end)
end

main1 = fn ->
  wms = [
    1.2,
    1.3,
    1.4,
    1.5
  ]

  _car_models = [
    "msv",
    "zoe",
    "corsa",
    "vw",
    "benz",
    "altis"
  ]
  |> Enum.map()
  |> Enum.map(fn name ->
```

```
    IO.puts(:stderr, "\nstart #{name}")
    run_width_muls.(name, wms)
  end
  )
end

#main1.()

# ---------------------------------------------------------------------

car_length = &(&1.param.dr + &1.param.df)

run_width_muls = fn car, car_name ->
  cfg = %{
    l: car_length.(car) * 1.4 |> Float.floor(2),
    w: 2.5,
    ct_max: 15,
    d_l: 0.03125,
    concur_max: 7
  }
  Main.min_length(car, cfg, nil)
  |> Enum.map(fn d ->
    val = elem(d, 0) / car_length.(car)
    put_elem(d, 0, val)
  end)
  |> (fn data ->
    Main.res_print(data, "plot/data/#{car_name}.txt")
  end).()
end


main2 = fn ->
  [
    "msv",
    "zoe",
    "corsa",
    "vw",
    "benz",
    "altis"
  ]
  |> Enum.map(fn name ->
    car = Car.new(name)
    {car, name}
  end)
  |> Enum.map(fn {car, name} ->
    IO.puts(:stderr, "\nstart #{name}")
    run_width_muls.(car, name)
  end
  )
end

#main2.()

(fn ->
  [2.5]
  |> Enum.map(fn w -> %{
    l: 7,
    w: w,
    ct_max: 20,
    d_l: 0.03125,
```

```
    concur_max: 8
  } end)
  |> Enum.map(fn cfg ->
    Main.min_length(Car.new("vclass"), cfg, "plot/data/vclass.txt")
  end)
end).()

#Main.make_record("altis", 6.25, 2.25, 0.44, "altis-w-2.25-l-6.25")
```