A DIRECT METHOD TO FIND OPTIMAL TRAJECTORIES FOR MOBILE ROBOTS USING INVERSE KINEMATICS

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of
Master of Science
in
Computer Science
by
Wenyu Lu
DARTMOUTH COLLEGE
Hanover, New Hampshire
August 2011

Ex offending Committee:

(Chair) Devin Balkcom

Scot Drysdale

Chris Bailey-Kellogg

Brian W. Pogue, Ph.D.
Dean of Graduate Studies
Abstract

Robot motion planning has applications in a wide range of areas such as robot navigation and medical surgery. Among the topics that are related to motion planning, computing time-optimal trajectories is challenging and of great importance. We want to move a robot from the start to the goal in the shortest time. There are four different vehicle designs that are considered in our work: Dubins cars, Reeds-Shepp cars, differential drive cars and omni-directional vehicles. The objective is to help us better understand structures of optimal trajectories of some certain kinematic vehicles, as well as to form a good basis for comparison with the indirect method, which applies Pontryagin’s principle to search rather than searching directly. In the proposed work, we build a direct method to explore time-optimal trajectories for mobile robots by using inverse kinematics: search free parameters (durations) to reach an intermediate configuration, then compute the last three durations from the intermediate configuration to the goal configuration by using inverse kinematics. The property of inverse kinematics guarantees the system to reach the goal precisely. Moreover, the inverse kinematics approach is a very simple algorithm for computing trajectories for systems with a small number of switches in their optimal trajectories. We have implemented the inverse kinematics solver for three-segment trajectories and a brute-force search planner in the absence of obstacles using naive uniform sampling algorithm. The results show that our method can compute optimal trajectories for Dubins cars and differential drive cars. It also produces good trajectories for omni-directional vehicles in most of our test cases. We also re-implemented the planner in C and applied optimization strategies. The search time turned out to be largely reduced.
# Contents

1 Introduction .................................................. 1
   1.1 Related work ................................................. 4
       1.1.1 Kinematics models ....................................... 4
       1.1.2 Searching and optimization approaches .................. 6
   1.2 Model and kinematics ....................................... 9
       1.2.1 Forward kinematics ....................................... 10
       1.2.2 Vehicle types ........................................... 11

2 Inverse kinematics ............................................ 12
   2.1 CCC .......................................................... 13
   2.2 LLC, CLL and LCL ............................................ 14
   2.3 LCC, CCL and CLC ............................................ 15
   2.4 LLL .......................................................... 16

3 Searching for optimal trajectories with fixed number of segments .... 19
   3.1 Three-segment planner ....................................... 19
   3.2 Searching trajectories with four actions or more .............. 20

4 Computing control lines ....................................... 21
   4.1 What is a control line ....................................... 21
   4.2 How to compute a control line ................................ 22
       4.2.1 Numeric approach to compute control lines .......... 22
       4.2.2 Geometric approach to compute control lines .......... 24
4.3 Determine the optimality of a trajectory ........................................... 25

5 Optimize the search process ................................................................. 28
  5.1 Updating the upper bound for search ................................................. 28
  5.2 Caching the three-segment trajectories ............................................. 29
  5.3 Planning with pre-computation ....................................................... 30

6 Implementation and results ................................................................. 33
  6.1 Implementation ............................................................................. 33
  6.2 Computing Dubins curves ............................................................. 33
  6.3 Comparison with the indirect approach ........................................... 34
  6.4 Time optimal trajectories for differential drive cars ....................... 35

7 Conclusion and future work ................................................................. 38
  7.1 Conclusion ..................................................................................... 38
  7.2 Future work ................................................................................. 39
    7.2.1 Speeding up the direct search by using optimization algorithms .... 39
    7.2.2 Computing optimal trajectories in the presence of obstacles ...... 40
# List of Figures

1.1 Left: a robot arm system; Right: a trajectory of a Dubin’s car. ................. 2
1.2 kinematic mobile models ................................................................. 11

2.1 CCC trajectory shown from two points of reference. .............................. 13
2.2 LLC trajectory shown from two points of reference. .............................. 15
2.3 left: original LCC trajectory; right: simplified LCC trajectory, \( p_s \) is the start position of the chosen reference point and \( p_g \) is the goal position of the chosen reference point; \( p_1 \) is the intersection. ........................................ 16
2.4 left: a sample CLC trajectory, \( p_1 \) and \( p_2 \) are switch points; right: a sample LLL trajectory ................................................................. 16
2.5 Convex hull of \( V \), \( v_i \) and \( v_j \) are adjacent vertices of \( p \). ................. 18

3.1 Pseudocode of the four-action planner. .................................................. 20

4.1 Switching points of a three-segment trajectory ................................. 23
4.2 Switch points of a three-segment trajectory ........................................ 25
4.3 A non-optimal trajectory: the two control lines do not agree with each other. 27
4.4 This trajectory is believed to be optimal. The two control lines almost overlap each other. ................................................................. 27

5.1 Computing trajectories with updated upper bound. ............................... 29
5.2 Computing trajectories with pre-computation. ..................................... 32

6.1 Optimal trajectories for Dubins cars. Left: the start configuration is (-3, -1, 0). Right: the start configuration is (-3, -4, 2). ................................. 34
6.2 Optimal trajectories for omni-directional cars. Left: a three-segment trajectory. The start configuration is (-2.34597872821, -4, -1.27073145627), the goal configuration is (2, 3, π/4); Right: a four-segment trajectory. The start configuration is (-2.5, -1, -2), the goal configuration is (0, 0, 0).

6.3 A three-segment optimal trajectory for differential-drive cars, the start configuration is (2,1, π/4); A four-segment optimal trajectory for differential-drive cars, the start configuration is (1,-1, π/4).

6.4 Optimal control for differential drive cars.
List of Tables

1.1 Control sets for different types of vehicles .............................. 11
Chapter 1

Introduction

Robot motion planning has applications in a wide range of areas such as robot navigation [15] and medical surgery [3]. Among the topics that are related to motion planning, computing time-optimal trajectories is challenging and of great importance. We want to move a mobile from the start to the goal in the shortest time avoiding any collision with obstacles. However, what we know about time optimal trajectories is still limited despite the extensive research on this topic. For example, Balkcom and Mason [5] has studied optimal trajectories for differential-drive cars. But this method can only be applied to environments without obstacles. Rapidly-exploring random trees [25] and Probabilistic Roadmaps [24] are widely used in motion planning problems. But these two methods only explore feasible trajectories rather than optimal trajectories.

In this work, we explore time-optimal trajectories for mobile cars by using a direct search approach. We define the configuration of a robot as $(x, y, \theta)$, which indicates the position and the orientation of the robot. The control $u = (\dot{x}, \dot{y}, \dot{\theta})$ describes the velocity in the local frame of the robot. Given the start configuration $q_0$ and the goal configuration $q_g$, the problem is to find the control function $u(t)$ such that

$$q_g = q_0 + \int_0^T \mathcal{R}(\theta(t))u(t)dt$$

(1.1)

and $T$ is minimized. Although the controls of a system are continuous, Furtuna and Balkcom [16] have proved that optimal trajectories only require a finite set of controls. Thus,
the problem is also stated as to find the sequence of controls \((u_1, u_2, \ldots, u_n)\) and the durations of each control \((t_1, t_2, \ldots, t_n)\), such that the system reaches the goal by applying control \(u_i\) for time \(t_i\) sequentially, and \(T = \sum_{i=1}^{n} t_i\) is minimized. Suppose we know the sequence of controls that is applied to the optimal trajectory. Durations are the parameters we need to calculate for the given goal configuration. Such a system can be viewed as a planar robot arm—a trajectory with a translation action and a rotation action is equivalent to a robot arm with a prismatic joint and a revolute joint. The original problem thus becomes an inverse kinematics problem: calculating all the joint angles in order to make the end effector of the robot arm to reach a specified position and orientation. If the trajectory

![Figure 1.1: Left: a robot arm system; Right: a trajectory of a Dubin’s car.](image)

is composed of two segments or less, the configurations that the system can reach is very limited. If three actions are applied, we have at most two solutions to reach a specified goal. If more than three controls are applied, there are an infinite number of possible ways to reach the goal configuration due to the extra degrees of freedom.

Based on the analysis above, we build a direct search method to compute optimal trajectories in the unobstructed plane. If the number of controls is fixed to \(n\), use the first \(n - 3\) durations as free parameters to reach an intermediate configuration. Then compute the last three durations from the intermediate configuration to the goal configuration by using inverse kinematics. There are a couple of reasons for choosing this method. First, the property of inverse kinematics guarantees the system to reach the goal precisely. Second, the inverse kinematics approach is a very simple algorithm for computing trajectories for systems with a small number of switches in their optimal trajectories. For example,
LaValle [26] has stated that the inverse kinematics method is probably the simplest method to compute optimal trajectories for Dubins cars. Moreover, the direct method forms a good basis for comparison with the indirect method which is developed by Furtuna [16].

We have implemented a brute-force search planner using Python. The results show that the direct method finds a good solution most of the time when the trajectories have no more than four segments. However, a naive uniform sampling is computationally expensive – computing a trajectory with more than five actions is almost infeasible. To increase the speed of search, we have also re-written the code using C. The new planner is almost a hundred times faster than the original one. Some optimization strategies are also applied to further reduce search time. These approaches include: updating upper bound for search in each iteration; caching trajectories between intermediate configurations and goal configuration to avoid redundant work; uniformly sampling the configuration space and precomputing all the three-segment trajectories.

Our planner fixes the number of actions when searching trajectories. Unless the number of actions for optimal trajectories is known in advance, the optimality of the trajectories produced by the planner remains a mystery. In order to determine if the trajectories are optimal or not, we compute control lines for different sections of a trajectory. The results can tell us the trajectories are not optimal when control lines of the trajectory do not agree with each other.

In the remainder of chapter 1, we discuss work related to robotics motion planning in terms of different kinematic models and optimization approaches. We also define problem model, kinematics as well as the symbols we will use in the following chapters.

Chapter 2 studies the inverse kinematics problems by digging into the eight cases of three-segment trajectories. We also give some sample trajectories for each case.

Chapter 3 explains how to search a three-segment trajectory using inverse kinematics when the structure of the trajectory is unknown. A four-segment planner is implemented by uniformly sampling the free parameter.

In chapter 4, we use control lines to determine the optimality of a trajectory for the given controls and durations. A control line is derived from the Hamiltonian which satisfies Pontryagin Maximum Principle. The distance between the reference point and the
control line determine the optimal control. We discuss two ways to compute control lines: a numeric approach and a geometric approach, as well as how to determine the optimality of a trajectory by computing control lines for different set of controls. The disagreement of these control lines indicates the found trajectory is not optimal, while the opposite result is only a hint that the trajectory is optimal.

Chapter 5 presents some strategies to reduce search time for the direct planner. One approach is to update upper bound for search after each iteration. With a tighter upper bound, the number of iterations can be reduced. Another method is to cache the three-segment trajectories for the intermediate configurations that is reached by the system. Those trajectories that have been cached can be used directly without extra computation. We also precompute all the three-segment trajectories within a certain range of configurations and store the results in a file. These trajectories can be reused for searching trajectories with more than three actions.

Chapter 6 discusses some experiments results obtained from applying the direct planner to different types of vehicles. We apply our new method to differential drive cars with bounded angular velocity. Balkcom and Mason [5] have solved the same problem by applying Pontryagin’s Maximum Principle to derive necessary conditions for time-optimal trajectories. We compare the results generated by the direct method with the results shown in Balkcom and Mason’s work. We also applied our planner to compute trajectories for omni-directional vehicles and compare our approach with the indirect method.

1.1 Related work

Most of the previous work has been focused on four kinematic models: Dubins cars, Reeds-Shepp cars, differential-drive cars and omni-directional vehicles.

1.1.1 Kinematics models

Dubins [13] characterized the structure of the shortest path between two configurations for a simple car. This model moves at a constant forward speed, with a maximum steering angle and a minimum turning radius. Dubins proved that these curves are composed of
no more than three segments. Each segment is either a line or circular arc, and there are only six combinations of these segments that are possibly optimal. As we stated in the introduction, the direct method is a very simple way to compute Dubins curves since there are only a few switches in the optimal trajectories. We just need to iterate over six possible structures and solve the inverse kinematics problem for each structure. Dubins curves arise in many applications of robotics motion planning. Alterovitz [3] [14] models a steerable medical needle which follows a Dubin’s curve during the insertion procedure and soon adapted this model to a dynamic environment [2]. Reeds and Shepp [30] extended Dubins’ work by allowing a steered car to move backwards as well as forwards. They showed that there are 48 candidate optimal paths; but this number was reduced to 46 by Sussmann and Tang [34].

In order to explore optimal trajectories for generalized models, Souères and Boissonnat [33] proposed to determine necessary conditions for time-minimum trajectories by using Pontryagin’s Maximum Principle. The PMP method was also adopted in [18], [16], [5] and [4]. Furtuna et al [18] generalized the Dubins and Reeds-Shepp curves. They proposed a polynomial-time algorithm to compute the minimum-time sequence of rotations and translations for two given configurations. Furtuna and Balkcom [16] explored optimal trajectories for generalized parameterized mobile models. They applied Pontryagin’s Maximum Principle to derive necessary conditions for time-optimal trajectories and provides geometric interpretations of the conditions.

Balkcom and Mason [5] solved the problem to find the time optimal trajectory for differential drive with bounded wheel angular velocities. The optimal trajectories contain rotations in place and straight lines. Chitsaz et al [12] also characterized the minimum wheel rotation trajectories for differential drive-vehicles in an unobstructed plane. Balkcom et al [4] studied the time-optimal trajectories for omni-directional vehicles. Unlike the mobile vehicles which are mentioned above, an omni-directional robot can slip in a direction parallel to the axle. The time-optimal trajectories are composed of circular arcs, straight lines as well as spins in place. However, they have not solved the problem of determining which trajectory is optimal.

In addition, Salaris [32] translated the optimal synthesis to the image plane and pro-
posed the optimal control words for a unicycle with a limited field-of-view.

Obstacles have also been considered in the motion planning problem. Agarwal et al [1] propose an algorithm to compute the shortest path in a convex polygon with the constraint that the curvature of the path is at most one. Vendittelli derived a geometric algorithm to compute the nonholonomic distance between a robot and polygonal obstacles. Hayet [20] presented a motion planner to compute collision-free path for a differential drive system that maintains visibility of a static landmark. Chitsaz and LaValle [11] proposed an algorithm to compute minimum wheel-rotation trajectories for differential-drive mobile robots among piecewise smooth and convex obstacles.

1.1.2 Searching and optimization approaches

Hart and Nilsson [19] proposed the $A^*$ algorithm to compute the minimum cost path through a graph by using heuristic. $A^*$ is a complete and accurate graph-searching algorithm but not applicable to continuous configuration space. The potential field path planning treats the robot as a point which is attracted by the goal and repulsed by the obstacles. Compared to $A^*$, the potential field approach is relatively low-cost but not complete – it probably fails to find a solution due to local minima. The randomized potential field method [6] avoids local minima by using random walk – the tradeoff is a large amount of parameter tuning.

In recent years, sampling-based motion planning [26] has been widely used for problems in high dimensional configuration spaces. Kavraki et al [24] presented a two-phase motion planning method – the learning phase constructs a Probabilistic RoadMap (PRM) which is composed of randomly-selected collision-free configurations and feasible paths that connect these configurations; the query phase then performs path planning in the roadmap by using a heuristic evaluator which is developed in the learning phase. The PRM method can be applied to generalized holonomic robots, and run more efficiently with customized components for some family of problems. The learning process significantly affects the efficiency and accuracy of the algorithm, since building a roadmap is computationally expensive and the roadmap is the basis for any further search. Once
the roadmap is established, it is usually easy and quick to process path planning queries. In addition, the original random sampling strategy has weaknesses in exploring narrow passages. Thus, a lot of work has been done to improve the methods for constructing roadmaps and enhance the performance of the PRM planners when exploring narrow passages.

Boor [9] introduced a strategy to sample the free configuration space by using Gaussian distribution. This technique largely reduces the number of samples compared to random sampling. Pisula et al [27] used a bounded error approximation of the generalized Voronoi diagram to improve sampling configurations in narrow regions. Holleman and Kavraki [21] used the medial axis as the heuristic for sampling configurations in order to find configurations of maximal clearance. The planner with this improved sampling strategy is more error-tolerant and performs better when computing narrow passages. Hsu et al [22] presented a bridge test strategy to find paths through narrow passages. A bridge is a line segment which is formed by connecting three sample configurations. This method builds short bridges inside narrow passages to enhance the density of samples; and use random sampling to achieve appropriate sampling density in wide open regions. A variant of PRM method proposed by Siméon et al [35] is the visibility roadmap approach. While the PRM approach constructs a roadmap with all collision-free configurations, the visibility roadmap method integrates only the configurations that connect two connected components of the roadmap, and the configurations that are not visible to the guard configurations. Compared to other PRM algorithms, the visibility roadmap approach produces a smaller roadmap and thus performs more efficiently in particular for capturing narrow passages.

The PRM approach is probabilistically complete, which means the probability of finding a solution gets close to one as the number of iterations increases [10]. Our method is different from PRM because the number of iterations is fixed. If a solution exists, our direct search planner is guaranteed to find the solution as long as the time resolution is small enough.

LaValle [25] proposed the concept of Rapidly-exploring Random Tree (RRT) for motion planning problems with nonholonomic constraints and high degrees of freedom. The
planner extends towards a randomly-selected point in each iterative step. By iteratively applying control inputs, an RRT spreads out across the whole configuration space until it reaches the goal. RRT has many nice properties which lead to its popularity in planning applications. For example, the RRT method is considered to be complete under very general conditions. It can also be embedded into other planning systems.

However, Yershova [37] pointed out that, due to the Voronoi-biased exploration of RRT, this algorithm has difficulties in computing trajectories when the obstacles are not taken into consideration, or the sampling domain is not properly chosen. Yershova’s work proposed an improved RRT algorithm with a new sampling strategy that is based on visibility regions. Compared to the original RRT, this new planner solves more classes of problems and runs much faster in many cases. Rodríguez [31] et al proposed some new strategies for growing the exploring tree by using the obstacle information and C-space information. For example, the shape of the obstacle and the boundary of the C-space can both be used for deciding which direction the tree grows. These new strategies improve the overall performance of the RRT algorithm, especially when exploring narrow passages and other difficult regions. Xu et al [36] applied RRT to motion planning of bevel-tip steerable needle in three-dimensional environment with obstacles. This method searched in randomly sampled control space to boost exploration completeness, and uses RRT with backchaining to find feasible entry points.

PRM and RRT are probably the most two popular sampling-based planning algorithms. The main difference between these two methods is that, PRM is categorized as a multiple-query method while RRT is a single-query method. The PRM method can answer multiple queries for the same environment with the trade-off for pre-computational cost. In contrast, an RRT planner is more efficient but only solves single-query problems. These two algorithms are combined by Bekris [7] in order to process multiple queries more efficiently. Our direct approach will also involve pre-computation. The difference is, the PRM method precomputes the probabilistic roadmap as the basis for future search; our method precomputes the optimal trajectories with a small number of switches, thus trajectories with more switches can be solved efficiently. Compared to the PRM and RRT approaches, the direct method is difficult to applied to high-dimensional problems. But it always finds the op-
timal trajectories (or relatively optimal trajectories, it depends on the search resolution), while the PRM and RRT planners usually find feasible trajectories rather than optimal trajectories.

There are some other planning algorithms that are worth being mentioned. The Expansive Space Tree (EST) method is a tree-based algorithm proposed by Hsu [23]. EST only samples the relevant portions of configuration space based on the expansiveness of configuration spaces. This approach can be recognized as a single-query solver because it has no pre-computation process. The EST planner has been combined with a PRM planner by Plaku et al [28]. The new planner is faster than PRM and more robust than the tree planners such as EST and RRT. Another sampling-based planning approach proposed by Ratliff [29] is Covariant Hamiltonian Optimization for Motion Planning (CHOMP). The CHOMP algorithm uses covariant gradient techniques to improve an input path to a collision-free trajectory, which is guaranteed to be locally optimal but not necessarily globally optimal. Bhatia and Kavraki [8] presented a multi-layered approach to solve motion planning problems with temporal goals. A high-level planner provides feasible high-level plans by using a discrete abstraction of the system. A low-level planner uses the high-level plans to explore feasible solutions. The discrete abstraction is constructed using the geometry of the obstacles and the subsets of the workspace.

1.2 Model and kinematics

The following models and kinematics are largely based on Furtuna’s related work. If the configuration is \( q = (x, y, \theta) \), the controls \( u \) represent the generalized velocity in the local frame of the robot. The transformation matrix that describes the frame of the robot with respect to the world frame is

\[
T_{WR} = \begin{bmatrix}
\cos \theta & -\sin \theta & x \\
\sin \theta & \cos \theta & y \\
0 & 0 & 1
\end{bmatrix}
\]
and $R$ is the matrix that transforms the velocity into the world frame

$$R = \begin{bmatrix} \cos(\theta(t)) & -\sin(\theta(t)) \\ \sin(\theta(t)) & \cos(\theta(t)) \end{bmatrix}$$ \hspace{1cm} (1.3)$$

then the system equations are

$$q(T) = q(0) + \int_0^T R(\theta(t))u(t)dt.$$ \hspace{1cm} (1.4)

### 1.2.1 Forward kinematics

The cardinal sine function is defined as:

$$\text{sinc}(x) = \begin{cases} 
\sin x / x, & x \neq 0 \\
1, & x = 0
\end{cases}$$ \hspace{1cm} (1.5)$$

Furtuna has defined a similar function, the *cardinal versine* by analogy with the cardinal sine:

$$\text{verc}(x) = \begin{cases} 
1 - \cos x / x, & x \neq 0 \\
0, & x = 0
\end{cases}$$ \hspace{1cm} (1.6)$$

Furtuna has also defined the unified translation-rotation matrix describing motion using the $i^{th}$ control for duration $t$:

$$T(i, t) = \begin{bmatrix} 
\cos \dot{\theta}_i t & -\sin \dot{\theta}_i t & \dot{x}_i t \text{sinc}(\dot{\theta}_i t) - \dot{y}_i t \text{verc}(\dot{\theta}_i t) \\
\sin \dot{\theta}_i t & \cos \dot{\theta}_i t & \dot{x}_i t \text{verc}(\dot{\theta}_i t) + \dot{y}_i t \text{sinc}(\dot{\theta}_i t) \\
0 & 0 & 1
\end{bmatrix}$$ \hspace{1cm} (1.7)$$

We define $U$ as the set of all controls and the $i^{th}$ element in $U$ is $u_i = (\dot{x}_i, \dot{y}_i, \dot{\theta}_i)$. Furtuna [16] has shown that optimal trajectories only require a finite set of constant controls, as long as the bounds on the generalized velocities are polyhedral. We also denote $D$ as the sequence of indices of the controls that are applied to the trajectory and the $i^{th}$ element in $D$ is $d_i$. A trajectory is defined as $[(d_1, t_1), (d_2, t_2), \ldots (d_n, t_n)]$, which indicates the application of
Table 1.1: Control sets for different types of vehicles

<table>
<thead>
<tr>
<th>Kinematic Model</th>
<th>Controls $(\dot{x}, \dot{y}, \dot{\theta})$, in robot frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dubins</td>
<td>$(1, 0, 0), (1, 0, \pm 1)$</td>
</tr>
<tr>
<td>Reed-Sheep</td>
<td>$(\pm 1, 0, 0), (\pm 1, 0, \pm 1)$</td>
</tr>
<tr>
<td>Diff-Drive</td>
<td>$(\pm 1, 0, 0), (0, 0, \pm 1)$</td>
</tr>
<tr>
<td>Omnidrive</td>
<td>$(\pm \sqrt{3}/3, \pm 1, 0), (\pm 2\sqrt{3}/3, 0, 0)$</td>
</tr>
<tr>
<td></td>
<td>$(0, 0, \pm 1), (0, \pm 4/3, \pm 1/3)$</td>
</tr>
<tr>
<td></td>
<td>$(\pm 2\sqrt{3}/3, 2/3, 1/3)$</td>
</tr>
<tr>
<td></td>
<td>$(\pm 2\sqrt{3}/3, -2/3, -1/3)$</td>
</tr>
</tbody>
</table>

control $u_{d_1}$ for time $t_1$, followed by control $u_{d_2}$ for time $t_2$, etc. The state after applying the $k^{th}$ control is:

$$T(t) = T_0T(d_1,t_1)T(d_2,t_2)\ldots T(d_k,t_k).$$ (1.8)

Given the starting configuration matrix as $T_0$ and the goal configuration matrix as $T_g$, The problem is to find the sequence of controls $D$ and the durations for each control $t_1, t_2, \ldots, t_n$, such that

$$T_g = T_0T(d_1,t_1)T(d_2,t_2)\ldots T(d_n,t_n),$$ (1.9)

and $t' = \sum_{i=1}^{n} t_i$ is minimized.

### 1.2.2 Vehicle types

In our work, we mainly consider four kinematic models: Dubins cars [13], Reeds and Sheep cars [30], differential-drive cars studied by Balkcom and Mason [5], omni-directional vehicles [4]. We choose different control sets for different kinematic designs, as shown in Table 1.1.
Chapter 2

Inverse kinematics

As we discussed in the first section, a three-segment trajectory can be computed using inverse kinematics. We denote a rotational action as ‘C’ and a translational action as ‘L’. There are eight cases for a three-segment trajectory: CCC, CLC, CLL, LLC, LCL, LCC, CCL and LLL. These cases are grouped into 4 categories by the number of rotation controls. For example, CCL, LCC and CLC fall into the same category since they all have two ‘C’s. Generally speaking, cases with the same number of rotation controls can be solved in a similar way. Define the start configuration as $q_0$ and goal configuration as $q_g$, and also define the start rotation matrix as $R_0$ and the goal rotation matrix $R_g$; $s$ and $g$ are the coordinates of the start position and the goal position respectively. The start configuration matrix is $T_0$ and the goal configuration matrix is $T_g$. Without loss of generality, re-index the controls so that the first three controls in $U$ are the controls for the trajectory structure currently being considered; for example, $u_1 = (\dot{x}_1, \dot{y}_1, \dot{\theta}_1)$ will be the first control applied, as well as the first control in the set $U$.

If the $i^{th}$ control is a rotation, the rotation center in the car’s local frame is:

$$c_i = \begin{bmatrix} -\dot{y}_i/\dot{\theta}_i \\ \dot{x}_i/\dot{\theta}_i \end{bmatrix}$$  \hspace{1cm} (2.1)$$

$c_i$ remains constant as long as the corresponding control is not changed. The basic idea of inverse kinematics is to calculate the coordinates of switch points. We will explain CCC in

12
The other cases can be solved in a similar way. To draw the trajectory of a mobile car, it is common and straightforward to choose the center of the mobile car as the reference point. However, this is not necessarily a good way. For the CCC case, the trajectory is composed of three circular arcs, which is quite complicated structure to study. Thus, we will use a different reference point to facilitate the geometric analysis. This approach will be applied to other cases as well. We call the trajectory of the center of the car original trajectory, and the trajectory with the chosen reference point simplified trajectory. If the simplified trajectory does not contain any circular arcs, we can easily get the solution by solving a linear equation.

2.1 CCC

In the original CCC trajectory, the centers of the circles $c_A$ and $c_B$ are the locations of the first rotation center in the initial state, and the third rotation center in the final state respectively:

$$c_A = R_0c_1 + s, \quad (2.2)$$

$$c_B = R_gc_3 + g. \quad (2.3)$$

Choose the second rotation center as the reference point; the trajectory structure is simplified to two circular arcs (figure 2.1). $p_s$ is the start position of the chosen reference point and $p_g$ is the goal position of the chosen reference point. The radii are

$$r_A = ||c_1 - c_2||, \quad (2.4)$$
\[ r_B = ||c_2 - c_3||. \] (2.5)

Compute the intersections of the circles, \( p_1 \) and \( p_2 \). For each intersection, compute the corresponding trajectory, and choose the faster. Define \( \Delta \theta_1, \Delta \theta_2 \) and \( \Delta \theta_3 \) as the angles that the vehicle rotates for the three controls respectively. For \( p_1 \),

\[
\cos(\Delta \theta_1) = \frac{(p_1 - c_A)^T(p_s - c_A)}{||p_1 - c_A|| ||p_s - c_A||} \] \hspace{1cm} (2.6)

\[
t_1 = \Delta \theta_1 / \hat{\theta}_1 \] \hspace{1cm} (2.7)

\[
\cos(\Delta \theta_3) = \frac{(p_1 - c_B)^T(p_g - c_B)}{||p_1 - c_B|| ||p_g - c_B||} \] \hspace{1cm} (2.8)

\[
t_3 = \Delta \theta_3 / \hat{\theta}_3 \] \hspace{1cm} (2.9)

Recall the forward kinematics equation

\[ T_g = T_0 T(1,t_1) T(2,t_2) T(3,t_3). \] (2.10)

Thus, we can infer that

\[ T(2,t_2) = (T_0 T(1,t_1))^{-1} T(3,t_3)^{-1} T_g. \] (2.11)

\( \Delta \theta_2 \) (and thus \( t_2 \)) may be computed using a two-argument arctangent of the first two elements in the first column of \( T(2,t_2) \). Computation for the second intersection \( p_2 \) is analogous:

\[
t_2 = \frac{\arctan(T(2,t_2)[1][1])}{\hat{\theta}_2} / T(2,t_2)[2][1]. \] (2.12)

### 2.2 LLC, CLL and LCL

The original trajectories of these three cases all contain a circular arc and two line segments. Choose the rotation center as the reference point; the simplified trajectory is composed of two line segments without any circular arcs. Figure 2.2 shows an example of an LLC.
Figure 2.2: LLC trajectory shown from two points of reference.

trajectory, for which the solution is:

\[
\begin{bmatrix}
\dot{x}_1 & \dot{x}_2 \\
\dot{y}_1 & \dot{y}_2
\end{bmatrix}
\begin{bmatrix}
t_1 \\
t_2
\end{bmatrix}
= p_g - p_s
\]  
(2.13)

\[T(3,t_3) = (T_0 T(1,t_1) T(2,t_2))^{-1} T_g\]  
(2.14)

\(\Delta \theta_3\) (and thus \(t_3\)) may be computed using a two-argument arctangent of the first two elements in the first column of \(T(3,t_3)\):

\[t_3 = \frac{\arctan(T(3,t_3)[1][1])}{\theta_3} \frac{T(3,t_3)[2][1]}{\theta_3}.\]  
(2.15)

2.3 LCC, CCL and CLC

For LCC and CCL, choose the second rotation center as the reference point and get a simplified trajectory which is composed of a line segment (the velocity vector in world frame) and a circular arc. The intersections of the line and the circle are possible switches.

However, we solve the CLC case in a different way since the velocity vector in the world frame cannot be computed directly by applying either the start configuration or the goal configuration to the velocity vector in the local frame. Define the two switch points \(p_1\) and \(p_2\). Suppose \(v\) is the velocity vector in the local frame, then the CLC case can be solved as followings:
Figure 2.3: left: original LCC trajectory; right: simplified LCC trajectory, \( p_s \) is the start position of the chosen reference point and \( p_g \) is the goal position of the chosen reference point; \( p_1 \) is the intersection.

Figure 2.4: left: a sample CLC trajectory, \( p_1 \) and \( p_2 \) are switch points; right: a sample LLL trajectory

\[
p_2 - p_1 = R_0 R(\dot{\theta}_1 t_1) v
\]

(2.16)

\[
p_2 - p_1 = R_g R(\dot{\theta}_3 t_3) v
\]

(2.17)

\[
t_2 = \frac{||p_2 - p_1||}{||v||}
\]

(2.18)

2.4 LLL

LLL trajectories can only be applied when the start and goal configurations have the same orientation. We will prove that an optimal trajectory can always be constructed using at most two translation controls.

**Theorem 1:** Consider any two configurations that have the same orientation. If the controls that are applied to trajectories between these two configurations are all translations, then we can construct the optimal trajectory with at most two translation controls.

**Proof:** Define the sequence of controls that is applied to the optimal trajectories as \( U = \)}
\{u_1, u_2, \ldots, u_n\}. The start position and goal position are \(q_0\) and \(q_g\) respectively. Since the controls in the applied sequence have no angular velocities, each control can be described as a velocity vector
\[
u = \begin{bmatrix}
\dot{x} \\
\dot{y}
\end{bmatrix}.
\tag{2.19}
\]
Our goal is to find the times \(t = (t_1, t_2, \ldots, t_n)\) such that
\[
q_g = q_0 + u_1 t_1 + u_2 t_2 + \cdots + u_n t_n.
\tag{2.20}
\]
Suppose \([t_1, t_2, \ldots, t_n]\) is a normalized vector, then \(q_g - q_0\) should be in a convex set of \(U\). As we know, convex hull is the smallest convex set of a vector space. This reminds us to use the nice properties of convex hull to prove our theory. Let \(\lambda_i = \frac{t_i}{T}, T = t_1 + t_2 + \cdots + t_n, \)
h = \(\frac{q_g - q_0}{T}\), then
\[
h = \lambda_1 u_1 + \lambda_2 u_2 + \cdots + \lambda_n u_n
\tag{2.21}
\]
with \(\lambda_1 + \lambda_2 + \cdots + \lambda_n = 1\).

Thus, we know that \(h\) is in the convex hull of \(U\). Suppose \(T'\) is the optimal time, \(h' = q_g - q_0\), scale \(u_1, u_2 \ldots u_n\) by \(T'\),
\[
v_i = T' u_i,
\tag{2.22}
\]
\[
h' = \lambda_1 v_i + \lambda_2 v_j + \cdots + \lambda_n v_n.
\tag{2.23}
\]
Define \(V\) as the vector set \(v_1, v_2, \ldots, v_n\), the above analysis indicates that \(h'\) is in the convex hull of \(V\). Choose the two adjacent vertices of \(h', v_i\) and \(v_j\). Since the convex hull is a polygon, \(h'\) can be described as the linear combination of \(v_i\) and \(v_j\). Choose \(\alpha\) and \(\beta\) such that \(\alpha + \beta = 1\), we have
\[
h' = \alpha v_i + \beta v_j.
\tag{2.24}\]
Let $a'$ equal to $T'a$ and $b'$ equal to $T'b$,

$$h' = a'u_i + b'u_j,$$

(2.25)

$$a' + b' = T',$$

(2.26)

which means $[a', b']'$ is the optimal solution. Thus, an optimal trajectory can always be constructed by using at most two translation controls.

![Convex hull of V, $v_i$ and $v_j$ are adjacent vertices of p.](image)

Figure 2.5: Convex hull of V, $v_i$ and $v_j$ are adjacent vertices of p.

We have proved that two translational controls are sufficient for the LLL case. But we don’t know which two controls yield the optimal trajectory. The way to solve this problem is quite simple - for each pair of controls, compute the solution by solving a linear function

$$
\begin{bmatrix}
\dot{x}_i & \dot{x}_j \\
\dot{y}_i & \dot{y}_j
\end{bmatrix}
\begin{bmatrix}
t_i \\
t_j
\end{bmatrix} = g - s.
$$

(2.27)

It is possible that we get negative solutions for the linear equation. This means the particular pair of controls is feasible. Then we just choose the fastest from the feasible solutions.
Chapter 3

Searching for optimal trajectories with fixed number of segments

3.1 Three-segment planner

We have solved the inverse kinematics problem for each class of three-action trajectory. To find the optimal three-action trajectory, we simply loop over all the possible structures with three controls and pick the fastest. Suppose the control list contains $n$ controls, then the number of all possible structures is $n^3$. For example, the control list for differential drive cars is composed of four controls, which derives 64 possible combinations of three controls.

However, if two consecutive controls are the same, we should ignore this combination since it equals to a two-control structure. And this situation is covered by any three-control set which contains these two controls-if the solution only requires two controls, then the time for the third one is 0. Consider a vehicle that has $n$ controls, there are $n^3 - 2n(n - 1)$ possible combinations of three controls excluding the situation that the first two controls or the last two controls are the same.
3.2 Searching trajectories with four actions or more

We have implemented the planner to find optimal four-action trajectories. Iterate over all possible trajectory structures with four segments, and sample the one-dimensional time space for each trajectory structure. There are two issues which need to be considered. First, what is a good resolution for sampling? To balance the accuracy and performance of our planner, we choose 0.1 as the sampling resolution.

Second, what is the lower bound and the upper bound of the sampling space? It is obvious that the lower bound of time is 0. Suppose there exists a solution \((u_1, t_1), (u_2, t_2), \ldots, (u_n, t_n), t' = \sum_{i=1}^{n} t_i\). The upper bound for sampling should be smaller than or equal to \(T\) if the duration for the first control of the final solution is greater than \(t'\), then there’s no way that this solution is the fastest. Thus, in our implementation, we simply get a solution by running our inverse kinematics solver and use the summation of the times as the upper bound.

```
FOUR_ACTION_TRAJECTORY(T0, Tg, U)
  upperbound = Get upperbound from IK solver
  t1=0
  for i=1:LENGTH_OF_U
    while(time<upperbound)
      T = configuration reached by apply t_i
      ans = Get solution from three-seg planner
      sol = [(U(i),time),ans]
      add sol to the list of solutions
    pick the fastest solution from the list
```

Figure 3.1: Pseudocode of the four-action planner.

A planner that searches trajectories with \(n\) actions \((n > 4)\) can be implemented in the same way – sample the first \(n-3\) durations and compute the last three durations using the three-segment planner. Our new planner that is written in C can search five-segment trajectories within a reasonable amount of time.
Chapter 4

Computing control lines

The direct planner searches optimal trajectories with a fixed number of actions. The optimality of the trajectory is a mystery unless the number of actions for optimal trajectories is known – for example, we already know that a Dubins car has three actions in its optimal trajectories and a differential-drive vehicle has four actions in its optimal trajectories. In this chapter, we will introduce the concept of control lines and how to use control lines to determine the optimality of a trajectory with more than three actions. More detailed information about control lines can be found in Furtuna’s doctoral dissertation [17].

4.1 What is a control line

Time optimal trajectories are supposed to satisfy Pontryagin’s Maximum Principle. The Hamiltonian to be maximized is

\[ H = k_1 \dot{x} + k_2 \dot{y} + \theta(k_1 y - k_2 x + k_3). \]  (4.1)

H is a constant value along the time-optimal trajectory.

The control line is defined as a line with heading \((k_1,k_2)\), and the signed distance \(k_3\) from the origin. The term \(-k_2 x + k_1 y + k_3\) is the distance from the reference point of the vehicle to the control line. The control line and the current state determine the optimal controls for the vehicle.
4.2 How to compute a control line

There are two ways to compute control lines: one is a numeric approach; the other one is a geometric approach. We will talk about both the methods in the following sections.

4.2.1 Numeric approach to compute control lines

Recall that the Hamiltonian is constant along a time-optimal trajectory. Consider a three-segment trajectory with two switching points A and B. The three controls can be described as the generalized velocities in the vehicle’s local frame: \((\dot{x}_{i1}, \dot{y}_{i1}, \dot{\theta}_{i1}), (\dot{x}_{i2}, \dot{y}_{i2}, \dot{\theta}_{i2}), (\dot{x}_{i3}, \dot{y}_{i3}, \dot{\theta}_{i3})\). The durations for each control are \(t_1, t_2, t_3\) respectively. \(v_1, v_2, v_3\) and \(v_4\) are translational velocities in world frame:

\[ v_1 = [\dot{x}_1, \dot{y}_1]^T, \quad (4.2) \]
\[ v_2 = [\dot{x}_2, \dot{y}_2]^T, \quad (4.3) \]
\[ v_3 = [\dot{x}_3, \dot{y}_3]^T, \quad (4.4) \]
\[ v_4 = [\dot{x}_4, \dot{y}_4]^T, \quad (4.5) \]
The rotational velocities in world frame $\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}$ are the same the angular velocities in local frame. At point A, the system switches from the first control to the second control; at point B, the system switches from the second control to the third control. The numerical approach solves $k_1, k_2, k_3$ and $H$ from linear functions shown as following:

$$
\begin{align*}
H &= k_1 \dot{x}_1 + k_2 \dot{y}_1 + \dot{\theta}(k_1 y_A - k_2 x_A + k_3) \\
H &= k_1 \dot{x}_2 + k_2 \dot{y}_2 + \dot{\theta}(k_1 y_A - k_2 x_A + k_3) \\
H &= k_1 \dot{x}_3 + k_2 \dot{y}_3 + \dot{\theta}(k_1 y_B - k_2 x_B + k_3) \\
H &= k_1 \dot{x}_4 + k_2 \dot{y}_4 + \dot{\theta}(k_1 y_B - k_2 x_B + k_3).
\end{align*}
$$

The first function indicate the state that the vehicle reaches point A and is about to switch to the second control; the second function indicates that the vehicle is still at point A and has already switched to the second control, etc. The functions can be solved by computing the null space of the matrix

$$
\begin{bmatrix}
\dot{x}_1 + \dot{\theta}_1 y_A & y_1 - \dot{\theta} x_A & \dot{\theta}_1 & -1 \\
\dot{x}_2 + \dot{\theta}_2 y_A & y_2 - \dot{\theta} x_A & \dot{\theta}_2 & -1 \\
\dot{x}_3 + \dot{\theta}_3 y_B & y_3 - \dot{\theta} x_B & \dot{\theta}_3 & -1 \\
\dot{x}_4 + \dot{\theta}_4 y_B & y_4 - \dot{\theta} x_B & \dot{\theta}_4 & -1
\end{bmatrix},
$$

and the result has to be normalized by the factor $\sqrt{k_1^2 + k_2^2}$. 

Figure 4.1: Switching points of a three-segment trajectory
4.2.2 Geometric approach to compute control lines

Define the “control line frame” L as a frame that is attached to the control line with its x-axis aligned with the control line. The geometric interpretation of a Hamiltonian is

\[ H = \dot{x}_L + y_L \dot{\theta}, \]  

where \( y_L \) is the distance from the vehicle to the control line; \( \theta_L \) is the angle of the vehicle with respect to the control line; \( x_L \) is the vehicle’s velocity along the control line.

Furtuna has proved that the same value of the Hamiltonian is obtained for any reference point on the vehicle when the trajectory obeys the Pontryagin Principle. Choosing a rotation center as the reference point, for which \( \dot{x}_L = 0 \). If the angular velocity is \( \dot{\theta} \) at time \( t \), then the distance from the reference point to the control line is

\[ y_L = H/\dot{\theta}. \]  

A switch point \( s_{ij} \) is defined as

\[ s_{ij} = \begin{bmatrix} \dot{y}_i - \dot{y}_j \\ \dot{x}_i - \dot{x}_j \\ \dot{\theta}_i - \dot{\theta}y_j \end{bmatrix}. \]  

A rotation center \( c_i \) for control \( u_i \) can be described as

\[ c_i = \begin{bmatrix} -\dot{y}_i \\ \dot{x}_i \\ \dot{\theta}_i \end{bmatrix}. \]  

\( s_{ij} \) can also be represented as

\[ s_{ij} = c_j - c_i. \]  

Consider a trajectory with controls \( u_1, u_2 \) and \( u_3 \). The Hamiltonians of \( u_1 \) and \( u_2 \) in the reference frame are

\[ H_1 = x_{1L} + y_L \dot{\theta}_1 \]  

24
Figure 4.2: Switch points of a three-segment trajectory

\[ H_2 = x_{2L} + y_L \dot{\theta}_2. \]  \hspace{1cm} (4.18)

\( H_1 \) and \( H_2 \) are equal only if \( y_L(\dot{\theta}_2 - \dot{\theta}_1) = 0 \), which indicates that \( s_{12} \) is on the control line. We can prove the switch point \( s_{23} \) is also on the control line. Based on the discussion above, we construct a linear system of \( k_1, k_2 \) and \( k_3 \):

\[
\begin{bmatrix}
  s_{12} \\
  s_{23}
\end{bmatrix}^T \begin{bmatrix}
  k_1 \\
  k_2 \\
  k_3
\end{bmatrix} = \begin{bmatrix}
  0 \\
  0
\end{bmatrix}.
\]  \hspace{1cm} (4.19)

### 4.3 Determine the optimality of a trajectory

From previous two sections, we know a trajectory with three controls has a unique control line. If a trajectory satisfies the Pontryagin Principle, there is only one control line along the trajectory. Otherwise we would get different Hamiltonians for the same switching point. However, even if these control lines all agree with each other, we still cannot say this trajectory is optimal. Thus, this method only provides limited information about the optimality of the trajectory.
From the two approaches for computing control lines, we conclude that three controls can determines a control line. Thus, for a trajectory with $n$ controls, we compute control lines for two control sets of three controls, where the controls appear in the same order as they are in the trajectory; the control indices in the two set are not all the same. If the two control lines agree with each other, it is a hint that the trajectory is optimal; otherwise the result proves that the trajectory is not optimal.

Take a four-segment trajectory as an example. Consider a trajectory with controls $u_1, u_2, u_3, u_4$, and durations $t_1, t_2, t_3, t_4$. To determine the optimality of this trajectory,

- Compute the control line for controls $u_1, u_2, u_3$ with corresponding durations $t_1, t_2, t_3$.
- Compute the control line for controls $u_1, u_2, u_4$ with corresponding durations $t_1, t_2, t_4$.
- Compare the two control lines. If they do not agree with each other, the trajectory is not optimal.
Figure 4.3: A non-optimal trajectory: the two control lines do not agree with each other.

Figure 4.4: This trajectory is believed to be optimal. The two control lines almost overlap each other.
Chapter 5

Optimize the search process

Although we have improved the speed of the direct planner greatly by rewriting the code in C, searching a five-action or six-action trajectory is still a relatively difficult task. There are a couple of reasons why the algorithm is inefficient. First, we have to uniformly sample the first $n - 3$ durations. When the number of actions is increased linearly, the time for search an optimal trajectory is increased exponentially. Assume we search optimal trajectories for a vehicle which has $n$ controls with a time limit of $t$ and a resolution of $r$, the number of iterations to search a four-action trajectory is $\lfloor nt/r \rfloor$; the number of iterations to search a five-action trajectory is $\lfloor (nt/r)^2 \rfloor$. The planner is slowed down not only by the number of iterations, but also the large amount of computation in each iteration – we have to loop over all the possible three-action trajectories between each intermediate configuration reached by applying the first $n - 3$ sample durations and the goal configuration. Thus, to speed up searching optimal trajectories, we can either reduce the number of iterations or the amount of computation in each iteration. In the remaining part of the section, we will talk about the strategies to improve the performance of the planner.

5.1 Updating the upper bound for search

One approach to reduce the number of iterations is to update the upper bound for searching free parameters after each iteration. Our current algorithm computes a three-segment trajectory by the inverse kinematics solver and uses the sum of the three durations as the
upper bound. As the searching process is going on, faster trajectories are produced by the planner and the current upper bound can be replaced by a tighter one.

5.2 Caching the three-segment trajectories

When the planner searches optimal trajectories, it is possible that different iterations reach the same or intermediate configuration. However, the current planner computes the last three durations in each iteration, which involves a great amount of redundant work. This method caches the trajectories between the intermediate configurations and goal configuration. If an intermediate configuration is reached again, we can use the trajectory in the

Figure 5.1: Computing trajectories with updated upper bound.
cache without calling the three-segment planner.

This method affects the accuracy of the planner because it is difficult to reach exactly the same configuration twice. We have to round the configuration to some precision, for example, 0.01, and find a corresponding trajectory in the cache approximately.

5.3 Planning with pre-computation

The brute-force search we’ve implemented is computationally expensive not only because the naive uniform sampling method is inefficient, but also because we do too much redundant work during the search process. For example, we apply control $u_i$ for time $t_i$ and get an intermediate configuration $T_i$. Then we compute the solution for optimal three-action trajectory by using inverse kinematics. In the following work, we apply control $Zu_j$ for time $t_j$. It is highly possible that we get an intermediate configuration $T_j$ that is $T_i = T_j$. Thus, we solve the same inverse kinematics problem again.

Assume we construct a grid with uniform sampled configurations. If optimal three-action trajectories for all the sampled configuration are precomputed and stored, we can get durations for the last three controls just by a single query if the intermediate configuration is sampled. Even if the configuration is not sampled, the pre-computation result still yields a good heuristic if the sampling resolution is reasonable. The performance of our direct search method will be thus greatly improved. Similarly, if we precompute the optimal $n$-action trajectories and store the result, then the computation cost for computing $(n + 1)$-action trajectories can be reduced.

We precompute the three-action trajectories and stored the results in a file. At the beginning of searching an optimal trajectory, these three-action trajectories are read into a hash table. Whenever we want to compute the three-segment trajectory between an intermediate configuration and goal configuration, we first look up in the hash table without calling the three-segment planner. There are a couple of issues we should pay attention to. First, the $(x, y, \theta)$ space should be gridded densely so we have a great chance to find the trajectories in the hash table. But a high resolution would result in a huge hash table, which is extremely time-consuming and would finally slow down the planner. Second, this opt-
timization approach inevitably affects the accuracy of the algorithm since we can hardly find exactly the same configuration in the hash table for an intermediate configuration. To get a tradeoff of the efficiency and accuracy of the planner, we sample the configuration space with a resolution of 0.1 and round each intermediate configuration to a precision of 0.1. This resolution guarantees the planner to search a trajectory with a reasonable error.

The pre-computation strategy has largely improved the speed of the planner. While it takes 20 minutes to search a five-segment trajectory for the original planner written in C, the search time is reduced to 150 seconds by applying the pre-computation strategy.
Figure 5.2: Computing trajectories with pre-computation.
Chapter 6

Implementation and results

We have implemented the direct search method with naive uniform sampling. The planner is applied to several kinematic models such as Dubins cars, differential-drive cars and omni-directional vehicles. We will compare the planning results that are generated by the direct planner and indirect planner. For differential-drive cars, we will also compare our planner with Balkcom and Mason’s previous work [5].

6.1 Implementation

We implemented the first version of the direct planner using Python with Numpy and tried all the new algorithms on this platform. The planner in Python is slow for problems with relatively high dimension – it takes about half an hour to search a four-segment trajectory and is almost infeasible for searching a five-segment trajectory. We have thus re-implemented the planner using C, which improved the speed of the planner about one hundred times. It only takes 30 seconds to search a four-segment trajectory and about 20 minutes to search a five-segment trajectory without any optimization strategy.

6.2 Computing Dubins curves

As LaValle [26] stated, the inverse kinematics method is probably the simplest way to compute optimal trajectories for Dubins cars. We used the direct method to compute Dubins
Curves. Dubins [13] has proved that the optimal trajectories of Dubins car only contain three segments. And there are only six possible optimal structures. Thus, no search process is required to compute Dubins curves - we only need to iterate over the six possible structures and solve each case by the inverse kinematics solver. The structure with the minimum total time is the final solution. Figure 6.2 shows some sample Dubins curves. The goal configuration is always $(0, 0, 0)$.

6.3 Comparison with the indirect approach

We have implemented the direct method using both Python and C. We also run a large number of test cases and compared the results with the solutions which are generated by the indirect search approach. As expected, the direct approach always finds a feasible solution which guarantees to reach the goal configuration precisely. When the distance is relatively short, the direct method produces the farthest trajectory 90% of the time, and this percentage falls down as the distance between the start and goal increases. The results are easy to explain: an optimal trajectory within a short distance needs only a small number of controls – three or four is enough in most cases. Thus, the direct approach works well when the goal is not far from the start, although insufficient sampling may slightly affect the performance. When the distance between the start and the goal becomes larger, the optimal trajectory probably needs more controls. The direct method fails in this situation since it cannot be applied to high-dimension problems due to the computation capability.
Although the direct method becomes less accurate when the optimal trajectory has more than four actions, it still produces trajectories with relatively low time cost, which is good enough for some path planning problems. On the other hand, the solution acquired by our method could also be an upper bound for the search processes of many other optimization approaches. Figure 6.3 shows sample optimal trajectories generated by the direct method for omni-directional cars.

### 6.4 Time optimal trajectories for differential drive cars

Balkcom and Mason [5] have solved the problem to find time optimal trajectories for differential drive vehicles with bounded angular velocity. They also proved that four actions are sufficient for an optimal trajectory. According to their previous work, our direct method is able to generate optimal trajectories for any given start and goal configurations. Thus, we have searched for optimal trajectories for bounded velocity differential drive cars. The results would test the correctness of our method, which forms a good basis for applying the same approach to other kinematic models.

We choose the goal configuration as $(0, 0, 0)$ and the start configuration as $(x, y, \theta)$, where $x \in [-3, 3], y \in [-3, 3], \text{ and } \theta = \frac{\pi}{4}$. The $(x, y)$ coordinates are sampled across...
the configuration space with a resolution of 0.1. The metric is composed of twelve regions. Each region defines the structure of optimal trajectories for the points that fall in the region. \( \sim \) indicates a turn-left action; similarly \( \simeq \) means turn-right; then \( \uparrow \) and \( \downarrow \) indicate go-forward and go-backward respectively. For example, given the start configuration \((2,1,\frac{\pi}{4})\) and goal configuration \((0,0,0)\), the structure of the optimal trajectory is \( \sim\downarrow\sim \). Configurations that fall in the four leaf-like regions all derive four-segment structures. For example, given the start configuration \((1,-1,\frac{\pi}{4})\) and goal configuration \((0,0,0)\), the structure of the optimal trajectory is \( \sim\uparrow\sim\downarrow\sim \). The result produced by our method agrees very well with Balkcom and Mason’s previous work.

Figure 6.3: A three-segment optimal trajectory for differential-drive cars, the start configuration is \((2,1,\frac{\pi}{4})\); A four-segment optimal trajectory for differential-drive cars, the start configuration is \((1,-1,\frac{\pi}{4})\).
Figure 6.4: Optimal control for differential drive cars.
Chapter 7

Conclusion and future work

In this chapter, we conclude the contribution of our work as well as its limitations. The directions for future work based on the current status are also discussed.

7.1 Conclusion

We have solved the inverse kinematics problems for the eight cases of a three-segment trajectory. The inverse kinematics solver guarantees the system to reach goal if the three-segment trajectories exist. For vehicles such as Dubins cars, the inverse kinematics approach is almost the simplest way to compute optimal trajectories since it has been proved that the optimal trajectories for a Dubins car have at most three actions.

Planner that searches optimal trajectories with fixed number of actions are also implemented. Since we have re-written the code in C, the search time of the planner has been reduced at least a hundred times. This allows for searching a five-action trajectory within a reasonable amount of time. To further increase the speed of the planner, we also use some optimization strategies which either reduce the number of iterations or the amount of computations in each iteration.

Our method is simple in theory but performs well for type of vehicles such as Dubins car and Differential-drive cars. If the planner explores sufficiently in the time space for the first \( n - 3 \) free parameters, then the planner guarantees to find the optimal trajectory as long as the number of actions is enough. Even though the planner fails to find the optimal
trajectory due to the insufficient number of actions, we are still able get a good solution with the inverse kinematics solver.

However, limitations also arise in our work. First, the number of actions has to be fixed when the planner searches trajectories. This lack of flexibility may result in a non-optimal solution when the number of actions for an optimal trajectory is unknown. Moreover, as the number of actions increases, the computation time grows rapidly due to the large number of possible structure and sample durations.

7.2 Future work

7.2.1 Speeding up the direct search by using optimization algorithms

The direct search approach we have accomplished is a brute-force method. We do a naive sampling for the duration time of the first control, and iterate over all the three-segment cases for each sample. Even though we have re-written the code in C and applied some optimization strategies, this approach is still almost computationally infeasible if the dimension of the problem is greater than five.

We could combine our inverse kinematics planner with some optimization algorithms rather than a brute-force search. We hope this improved method would search for optimal trajectories with more segments, and thus solve the problem of find optimal trajectories for kinematic models such as omnidirectional cars.

Since our inverse kinematics solver guarantees the system to reach the goal configuration precisely, no error function is required to compute the optimal trajectory – the cost function is simply the sum of the durations for each control. Assume the sequence of controls that we apply to the optimal trajectory is \((u_1, u_2, \ldots, u_n)\). Define \(h(t_1, t_2, \ldots, t_n)\) as the forward kinematics function that returns the configuration after we apply \(u_i\) for \(t_i\) sequentially. \(g^{-1}(X, Y)\) returns the total time cost for the optimal three-action trajectory between configuration \(X\) and configuration \(Y\). Given the start configuration \(T_0\) and the goal configuration \(T_g\), the cost function we want to minimize is:

\[
f(t_1, t_2, \ldots, t_n) = t_1 + t_2 + \cdots + t_n + g^{-1}(h(t_1, t_2, \ldots, t_n), T_g). \tag{7.1}
\]
The purpose is to minimize the above function

A lot of optimization methods are possibly feasible, such as gradient descent, simulated annealing, and Newton’s method. However, some of these methods requires computing analytical gradients of the cost function explicitly, which is not an easy task because the inverse kinematics solver is too complex.

7.2.2 Computing optimal trajectories in the presence of obstacles

By now, our method only searches for optimal trajectories in an unobstructed plane, so does the indirect method which is developed by Furtuna. A more interesting but challenging topic is to search for trajectories in an environment with obstacles. It is feasible, although not an easy task, to solve this problem with the direct search method. In each iteration step, if there are any obstacles that collide with the candidate trajectory, just discard this invalid solution. Collision detection would be easy if we view the vehicle and obstacles as points – a trajectory is composed of circular arcs and line segments, we just need to check if the points (obstacles) lie on those arcs or segments with the coordinates of the points known in advance. Computation becomes more complicated if the obstacles are polygons or even have irregular shapes. We have discussed motion planning with obstacles in the section of related work. It is possible try to combine the algorithms that previously developed such as RRT, probabilistic roadmap, etc, with our inverse kinematics planner. The issues we need to consider about are:

- What is the collision detection method for our planner?
  This problem largely depends on what shapes we choose for obstacles and the vehicle. A simple way is to model all the obstacles and the vehicle as circles. To detect collisions, we just need to check if the circles intersect with each other. A more general but complicated way is to model all the objects as polygons or even some irregular shapes, and use the bounding box method to detect collisions.

- How to search the optimal trajectory?
  The main idea of our direct search method is to computing many candidate trajectories by iterate over all possible structures and times, and pick the one with the
minimum time cost. An intuitive way for planning in presence of obstacles is to do collision detection in every search step. If any collision happens, then discard the invalid trajectory. But this method is not guaranteed to be complete - we have to fix the number of segments when computing optimal trajectories. It is possible that there are no feasible trajectories with a certain number of segments. In order to avoid this problem, a good start point is to explore optimal trajectories with only a few obstacles. The result can still help us understand more about the structure of optimal trajectories.
Bibliography


