FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Motion Planning for Car-like Robots

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Lukas Krammer
Matrikelnummer 0525332

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Wolfgang Granzer

Wien, 23.07.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Lukas Krammer
Simon-Denk-Gasse 2/4
1090 WIEN

Erklärung

”Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen - , die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.”

Wien, 23.07.2010

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Abstract

Motion planning is one of the most challenging tasks in robotics. Dedicated algorithms are used in many different applications starting from CNC machines to human-like robots. An interesting research area within this field is motion planning for car-like robots. In the last few years, car-like robots became increasingly important, because precise positioning systems like GPS or GLONASS and modern sensor technologies allowed navigating in rural or even in urban terrains. These car-like robots can be used for different kinds of purposes. In the automotive domain, they may pave the way for driving in urban terrain. In the agriculture area, applications for harvesting, fertilizing or lawn mowing can be realized. This thesis focuses on motion planning for car-like robots and particularly on applications where an arbitrary working area should be covered as efficient as possible.

The aim of this thesis is to develop a path planning application which is able to compute a feasible path between two arbitrary points. Furthermore, the path shall fully cover a predefined area avoiding obstacles. Moreover, it is desired that the predefined working area is cruised in parallel lanes (e.g., for mowing a soccer field). The path shall be calculated based on lists of geodetic data which represent arbitrary but simple polygons.

At the beginning, basic concepts of system theory and system modeling will be studied. Next, different approaches for solving the basic motion planning problem will be discussed. Then, system models for car-like robots will be addressed. Based on such a system model, motion planning concepts are examined which are suitable for car-like robots. Considering the benefits and the drawbacks of all investigated algorithms, a solution based on randomized trees is proposed. Finally, an algorithm for motion planning covering predefined areas in parallel lanes is introduced.

A proof-of-concept implementation in combination with a simulation framework allows evaluating the quality and the feasibility of the computed path. The simulation is based on a simple path following controller and a realistic system model for a car-like robot which considers errors of the system (e.g., GPS and heading errors) close to reality. Analyses of the simulation show that the car-like robot is able to follow the computed path even in presence of system errors.

Kurzfassung

Routenplanung bzw. Bewegungsplanung sind aktuell spannende Forschungsgebiete der Robotertechnik. Angefangen von CNC Maschinen bis hin zu humanoiden Robotern gibt es verschiedenste Anwendungsbereiche für Roboter. Ein wichtiger Anwendungsbereich ist der automotive Sektor. Hier nehmen Arbeiten an autonomen Fahrzeugen, die sich im städtischen Bereich frei bewegen können, einen bedeutenden Stellenwert ein. Darüber hinaus gibt es noch weitere, weniger bekannte, jedoch ebenfalls anspruchsvolle Anwendungsgebiete für mobile Roboter. Großes Potential haben dabei mobile Roboter im landwirtschaftlichen Bereich. Genau diesem Anwendungsgebiet ist diese Diplomarbeit gewidmet. Sie beschäftigt sich hauptsächlich mit der Routenplanung für mobile Roboter im landwirtschaftlichen Kontext und geht dabei speziell auf die Anfordernisse dieses Bereichs ein. Ein Beispiel für eine Anwendung ist das Mähen einer bestimmten Fläche auf einem Feld.

Das Ziel dieser Arbeit ist es, einen Routenplanungsalgorithmus für automobilähnliche Roboter zu entwickeln und zu implementieren. Die Entwicklung ist an ein bereits existierendes Fahrzeug angelehnt. Dieses Fahrzeug ist mit einem präzisen satelliten-gestützten Navigationssystem und digitalem Kompass ausgestattet. Der genannte Algorithmus soll es ermöglichen, einen Pfad zwischen zwei beliebigen Punkten zu finden, sowie einen definierten Bereich, wie z.B. eine Arbeitsfläche, in parallelen Bahnen abzufahren.

Um dieses Ziel zu erreichen, werden im ersten Schritt einige vielversprechende Planungsstrategien und Ansätze untersucht. Im Anschluss daran werden Algorithmen diskutiert, die speziell für mobile Roboter geeignet sind. Diese Ansätze gehen von dem grundlegenden Problem aus, unter Berücksichtigung von Hindernissen und spezifischen Systemeigenschaften, einen zulässigen Pfad zwischen zwei beliebigen Punkten zu finden. Bei der Untersuchung dieser Algorithmen wird auch auf die praktische Umsetzbarkeit geachtet.

Basierend auf den zuvor vorgestellten Ansätzen wird ein Algorithmus entwickelt und diskutiert, der es ermöglicht, ein beliebiges Feld in parallelen Bahnen abzufahren, wobei größter Wert auf Effizienz und Realisierbarkeit gelegt wird. Um zu verifizieren, ob der generierte Pfad für das Fahrzeug fahrbar ist, wird das Verhalten eines solchen Fahrzeugs simuliert. Dabei wird ein realistisches Modell eines automobilähnlichen Fahrzeugs angenommen. Weiters werden die Fehler der Sensoren (Positionsfehler und Orientierungsfehler) realtitsnahe berücksichtigt. Um das Fahrzeug entlang eines Pfades zu bewegen, wird ein geeigneter Steuer- bzw. Regelalgorithmus verwendet. Analysen zeigen, dass der neu entwickelte Algorithmus alle gewünschten Anforderungen erfüllt.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction and motivation

1.1 Problem environment

In the past two decades, robots became more and more present in our daily life. Primarily, robots were invented to save human resources in industry. Since a few years the use of robots is not only limited to the industrial domain. In the following, the use of robots in the three popular domains is presented:

- Industry
- Urban traffic
- Agriculture

1.1.1 Robots in the industry

Manufacturing steps where normally a lot of manpower is necessary is nowadays done by different types of robots. This automation saves a lot of human resources and increases the quality of the products because robots work precisely around the clock. In the most cases, robot arms are used to replace the work of humans. However, movements which are easy to handle for humans (e.g., moving along a contour of a component) are difficult tasks for robots. Many challenging aspects like mechanical uncertainty or dynamic forces (e.g., inertia) have to be considered. It is often desired that the robot moves as fast as possible which obviously increases the dynamic forces. This simple example provides a first insight into the complexity of planning and controlling tasks.

However, not only simple robot arms are used in the industry. Automated guided transport vehicles are used in factory halls. Consider, for example, large assembly lines where objects are transported from one machine to another. These robot systems can also save lots of human resources. Robots which are used in factory halls can possibly be guided by a colored line on the floor or by a radio signal in a wire under the floor. They are often controlled by centralized

entities which plan and supervise the actions of the system. The realization of such systems is not a trivial task. For instance, it must be guaranteed that the robot stops if there is an obstacle in front of it (e.g., an employee). It must be guaranteed that the robot keeps on a predefined path. If no predefined path exists, the robot must navigate based on an indoor positioning system. Such a system is necessary to determine its position. In case of a factory robot, this could be done by laser beams or triangulation techniques. These two examples show that robots have revolutionized the industry. However, controlling a robot or planning its actions are still challenging tasks.

1.1.2 Robots in urban terrain

Another very interesting application of robots are Automated Guided Vehicle (AGV)s which are able to navigate in urban environments in presence of traffic lights, road signs, and traffic participants. Since a lot of unpredictable events (e.g., obstacles in front of the car) are possible, no technically mature autonomous car exists at present. Currently, some prototypes (e.g., [11, 12]) are developed which use laser distance measuring to construct a 3D surface of the environment. Due to the relative high speed of the vehicle, it is immanent to have a dependable long range obstacle detection. Additionally, precise satellite based navigation systems can be used. Many systems rely on image processing technologies to recognize signal lights or road markings (e.g., for lane keeping). However, many of these technologies are still under development or too expensive for practical use. Therefore, it will take further time to be ready for market.

1.1.3 Robots in agriculture

Besides industry, robots are more and more used in agriculture. However, robots in this area are not widespread. One reason is that agricultural robots need precise positioning systems to navigate in rural terrains. In this case, techniques which are well suited for indoor applications can not be used. The key-technology for this application is a Global Navigation Satellite System (GNSS). This term denotes positioning systems that use satellites to determine the actual position of an entity. The use of this technology was impractical for a long period of time due to economical and political reasons. Since navigation in rural terrain is less complex than in urban terrain, agricultural robots do not depend on expensive sensor technologies. Moreover, the speed of agricultural robots is slower than the speed of urban AGVs Thus, the reaction time can be longer and less expensive hardware is necessary.

Due to these facts, the agricultural area provides a widespread area for research and development at present. Currently, a few companies (e.g., CLAAS GPS Pilot[1]) provide Global Positioning System (GPS)-based tractors and harvesters. However, a lot of challenges still remain for agricultural robots.

[1] More Information at http://www.claas.com/

1.2 Aim of the thesis

This thesis mainly deals with robots for agricultural purposes. Since precise navigation systems like GPS are available for everyone and well established sensor technologies (e.g., for obstacle detection) are sufficient for navigation in rural terrain, robots for agricultural purpose seem to be realizable at the moment.

Thus, the thesis focuses on Car-Like Robot (CLR) which are able to navigate in rural terrain. For many applications in the agricultural field, it is necessary to cover a predefined area (e.g., a field). If a robot should be used to mow or harvest, it is desired and necessary that it visits each point of the field at least once. However, it is not desired that the robot visits one part of the field several times and other parts of the fields only once. For many applications, it is also necessary to cruise the field in parallel lanes.

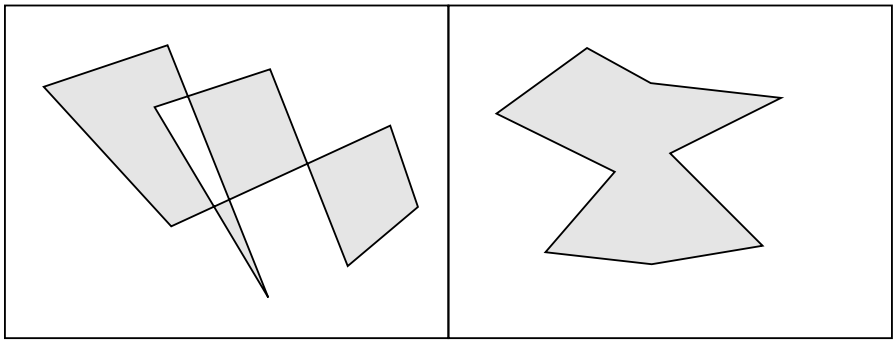
In the context of this thesis, different motion planning strategies will be discussed particularly useful for CLRs. Furthermore, a practical motion planning algorithm will be designed and implemented which fulfills the mentioned claims. This algorithm shall compute a complete path based on a predefined environment in an achievable way. The computed paths have to be feasible for CLRs.

1.3 Problem statement

The aim of this thesis is to find a feasible path for a CLR which covers a predefined area. The most common positioning system (GPS) shall provide the field coordinates. An area can be defined by an ordered set of GPS coordinates. This ordered set of points represents a polygon on the plane. In addition to the plane coordinates (latitude, longitude), GPS provides also the height w.r.t. the sea level. Therefore, it could even be possible to define the field in a 3D space. However, it is assumed that the field is flat, i.e., differences between the representation in 3D space and in 2D space can be neglected. Hence, a polygon will be represented by a list of two dimensional plane coordinates. Moreover, it is mandatory that the polygon is simple. This means that the polygon delimits exactly one area. This definition is equivalent with the claim that two pairwise disjoint line segments must not intersect. Figure 1.1 illustrates the difference between a simple and a complex polygon. However, it is not necessary that the polygon is convex. This means that it is allowed that the polygon consists of internal angles greater than 180 degree. To illustrate the difference between these two types of polygons, Figure 1.2 shows an example of each one.

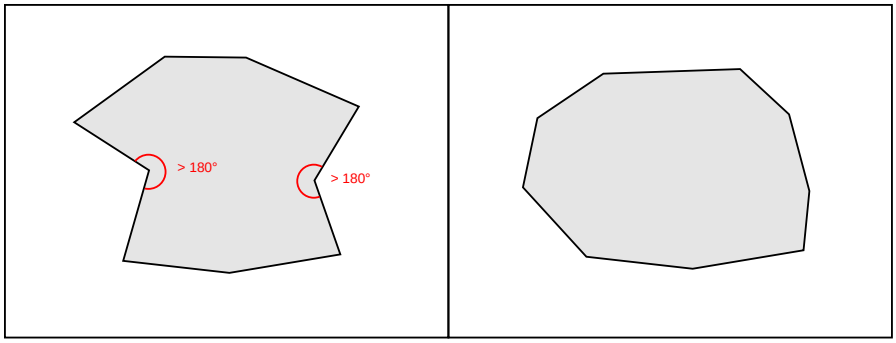
Based on such polygons a complete *definition area* can be defined as shown in Figure 1.3. The term definition area denotes an area on the plane where a robot is allowed to work independently with a few restrictions. In the figure, the definition area is represented by the red polygon. It is assumed that nothing outside this area is considered. If the robot gets out of this area (e.g., due to an uncertain event), it has to stop immediately. Moreover, it is assumed that all static obstacles within the definition area are known. Thus, the robot can safely operate within this area.

Furthermore, the definition area contains the *working area*. This area should be covered by the robot as shown in Figure 1.3 to be suitable for special applications. It is aimed, that



(a) Complex Polygon (b) Simple Polygon

Figure 1.1: Example of a complex and a simple polygon



(a) Non-convex Polygon (b) Convex Polygon

Figure 1.2: Example of a non-convex and a convex polygon

the working area is cruised in parallel lanes. Since robots can have different working breadth, the distance between two adjacent lanes must be variable. Moreover, it is necessary that the direction of the lanes is variable, too. This property can reduce the number of turn maneuvers dramatically. Since it is not required that the working area is convex, it is possible that very short lanes appear. These short lane segments can be neglected, if they are shorter than a predefined length.

Finally, all polygonal static obstacles in the definition area must be defined and known in advance. These obstacles represent dangerous areas for the robot (e.g., trees, lakes, fences). In Figure 1.3, these obstacles are marked blue. To simplify the routing considerations in the following chapters, it is assumed w.l.o.g. that obstacles are either completely within the working area or outside the working area.

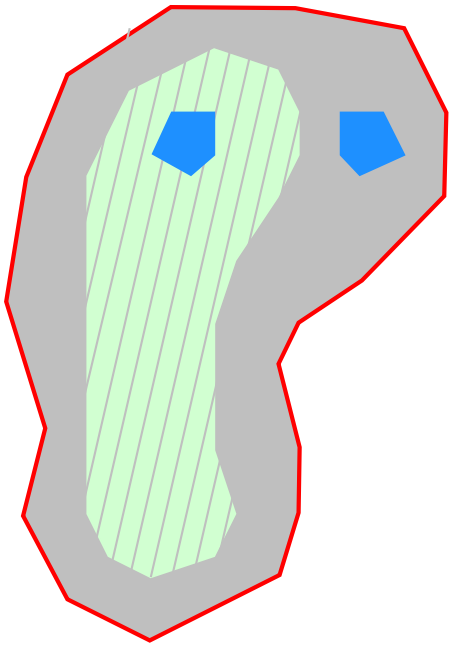


Figure 1.3: Example of a definition area including the working area and two obstacles

1.4 Outline

As mentioned in the introduction, this thesis primarily deals with motion planning for GPS-based AGVs. Thus, Chapter 2, provides a short introduction to satellite based navigation systems and presents basic concepts of system theory and system modeling. In Section 2.1, an overview about GNSS is given. Furthermore, GPS is introduced and approaches to improve the precision of GPS are discussed. Additionally, methods are presented how to transform GPS coordinates to planar coordinates. Additionally different types of real systems (e.g., robots) are shown. Since real systems have some constraints in general, this chapter also describes methods to model different constraints. Moreover, the difference between a trajectory or a physical path is presented.

After the basic concepts of system theory were introduced, Chapter 3 presents a selection of different motion planning approaches. Moreover, this chapter also works out the advantages and disadvantages of each approach and the possibility of practical use.

As this thesis especially deals with CLRs, Chapter 4 presents a simplified system model (including the kinematic constraints) of CLRs . Based on this system model, motion planning algorithms are considered which are well suited for CLRs. Finally, a comprehensive algorithm is presented which completely covers a predefined field.

In additional to the theoretical work, Chapter 5 describes a *proof-of-concept* implementation of the designed motion planning approach for CLR which fulfills the requirements. In this chapter, the implementation of data structures and geometric algorithms (e.g., collision detection) are discussed, too. Furthermore, important computational optimizations are discussed and finally a method is described which can be used to avoid dynamically appearing obstacles.

To prove the feasibility of the computed path, the behavior of a real CLR was simulated. Chapter 6 discusses the considered uncertainties of the simulation (e.g., error models). Furthermore, different path tracking controllers are discussed. Finally, one controller is used to simulate the behavior of the system.

Finally, Chapter 7 summarizes the main results of this thesis. On one hand, the results of the motion planning application are discussed (i.e., the performance and the quality of the path). On the other hand, the results of the simulation are presented which show that the computed paths are feasible.

The thesis is concluded with Chapter 8 which contains a summary and an outlook on further research activities.

CHAPTER 2

Basic concepts

2.1 Positioning systems

In the previous chapter, it was claimed that AGVs commonly navigate with the help of GNSSs. Thus, this section introduces the principles of GNSS. Since a GNSS basically provides geodetic data, an overview about the geometry of geodetic data is given, too.

2.1.1 Global navigation satellite systems

GNSSs are basically used to determine the absolute position of an entity allover the world. In addition to the position information, these systems also provide the height w.r.t. the sea level which is important for aerospace applications. Currently, a few GNSS are available from which exactly one is fully operational, namely GPS. All these systems are based on the same principle. A complete GNSS consists of a few base stations (four base stations and one control center in case of GPS) around the equator. These base stations communicate with a set of satellites (at least 24 for GPS) to determine and to correct the position of the satellites. To determine the position and the height of a point, a receiver is necessary. Based on the information of at least three satellites, a receiver is able to determine the position of a point.

Primarily, GNSS were developed for military use. Nevertheless, GPS provides a low accurate version for civilian use. The GPS system uses two different code sequences (P/Y and C/A) which are transmitted over two different carrier frequencies `L1` and `L2` for position determination as shown in Figure 2.1.

- The P/Y code uses a high *chiprate* for the transmitted signal and therefore it provides high accuracy. Since it is encrypted, it is only usable for military purposes. The P/Y code is transmitted over `L1` and over `L2` to correct atmospheric disturbance. The reason is that atmospheric errors behave different for different frequencies and therefore errors can be estimated and corrected.

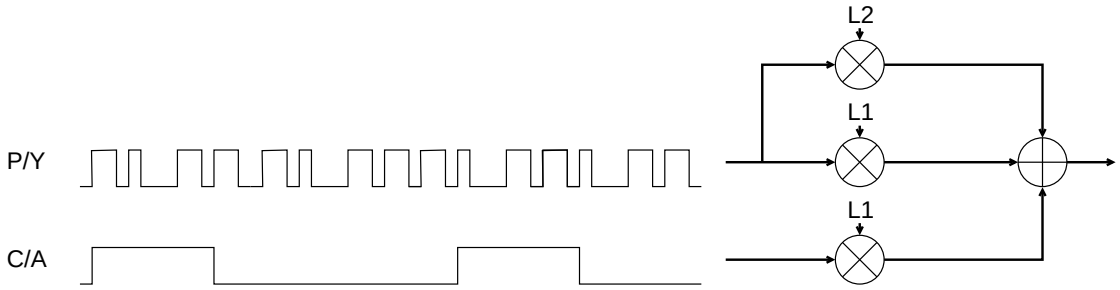


Figure 2.1: Generation of a GPS signal

- The C/A code uses a ten times lower *chiprate*. Thus, the position determination is less accurate. However, the C/A code is not encrypted and can be used free of charge. However, the C/A code is only transmitted over the `L1` carrier frequency. Error estimation and correction as mentioned above are not possible. Furthermore, the accuracy of the C/A code was artificially degraded by a technique called Selective Availability (SA). Therefore, the position accuracy of GPS was about 100 meters [13] . In 2000, the US turned of the SA and therefore the precision increased to 30 meters. Since the accuracy is still too low for dedicated application fields, Differential GPS (DGPS) is a common alternative to increase the accuracy below 30 meters.

2.1.2 Differential global positioning system

DGPS makes it possible to increase the relative accuracy for civilian use by merging the results of two receivers. This approach is based on the idea that most of the natural errors (e.g., stratospheric errors) are strongly correlated for receivers which are not too far away from each other. Before SA was turned off, DGPS was able to correct the artificial errors, too because these errors are also correlated for different receivers.

With this system, the relative accuracy (from one receiver to the other) was increased up to 3 meters. If DGPS is used, two receivers are necessary which have to be connected to each other. In general, one of these receivers is called the *base-station* which is located on a fixed position The base station transmits correction data via a radio connection to the other mobile receiver called *rover*.

Since the atmospheric errors only change over a long period of time (i.e., a few minutes) and are quite similar for large regions, it is possible to use broadcast correction data. These data can either be transmitted via radio signals or over satellites. These systems are called Satellite Based Augmentation System (SBAS). One example of such a system is the European Geostationary Navigation Overlay Service (EGNOS). This system uses geostationary communication satellites to transfer the correction data to the receivers. Hence, receivers which can receive EGNOS correction information in addition to the normal signals of GPS satellites provide higher position accuracy. Since the correction information of such a wide range system is based on interpolation techniques, it is not as precise as the information of a local *base-station*.

2.1.3 Real time kinematic

Another approach called Real Time Kinematic (RTK) provides a much more accurate positioning method. Instead of evaluating the phase of the code, RTK receivers are able to evaluate the carrier phase of the signal. Since the code is modulated with a carrier frequency, this frequency (approx. 1.500 MHz) must be much higher than the code frequency (approx. 1MBaud) called *chiprate*. Thus, the phase difference between the received carrier signal and the local reference signal provides higher resolution than the phase difference of the code itself. Since two receivers are used, the influence of atmospheric errors can be reduced and an accuracy of 0,2 meters can be provided. If the carrier phase of both `L1` and `L2` frequency is evaluated, the position accuracy increases up to 0,02 meters. As mentioned in the previous section, it is not possible to decrypt the code, which is modulated by the `L2` frequency but it is possible to compare the carrier phase independently from the modulated data. At the moment, RTK systems are primarily used for geodetic purposes. The main reasons are the costs of such systems. Thus, this approach is not suitable for low cost applications.

2.2 Position determination

If one specifies a point anywhere on earth using geodetic data (i.e., latitude, longitude, and height), this point is not determined exactly. To specify the exact position of a point, the geodetic model and the coordinate system has to be defined, too. Hence, this section introduces the geodetic models to determine GPS coordinates. Furthermore, methods to compute the distance between two points are presented.

2.2.1 Coordinate systems

Since the earth has no exact geometric form (e.g., ellipsoid), a few slightly different geometric models (i.e., coordinate systems) exist. The most commonly used coordinate system is the World Geodetic System 1984 (WGS84). The WGS84 is based on a three-dimensional Cartesian coordinate system as shown in Figure 2.2. Furthermore, the origin and the directions of at least two axes have to be defined. For the origin of WGS84, the *center of mass* of the earth is taken. The direction of the Z-axis is defined as the north pole of the earth. The direction of the X-axis is determined by the intersection between the meridian plane and the equator. The meridian plane has no physical matter and is defined arbitrarily.[1] The direction of the Y-axis follows from the origin and the two other axis.

To specify the position of the north pole and the equator, the earth is approximated by an ellipsoid where the center of it is equal to the *center of mass* of the earth. The ellipsoid used by WGS84 is defined by the parameters given in Table 2.1.

The semi-major axis represents the distance from the center of the ellipsoid to the farthest point of the ellipsoid (i.e., longest diameter of the ellipsoid). In contrast, the semi-minor axis represents the shortest diameter diameter of the ellipsoid. To illustrate these parameters, the ellipsoid can be interpreted as rotating ellipse. Figure 2.3 shows an ellipse with the semi-major

[1] The Prime Meridian historically passes through Greenwich (UK)

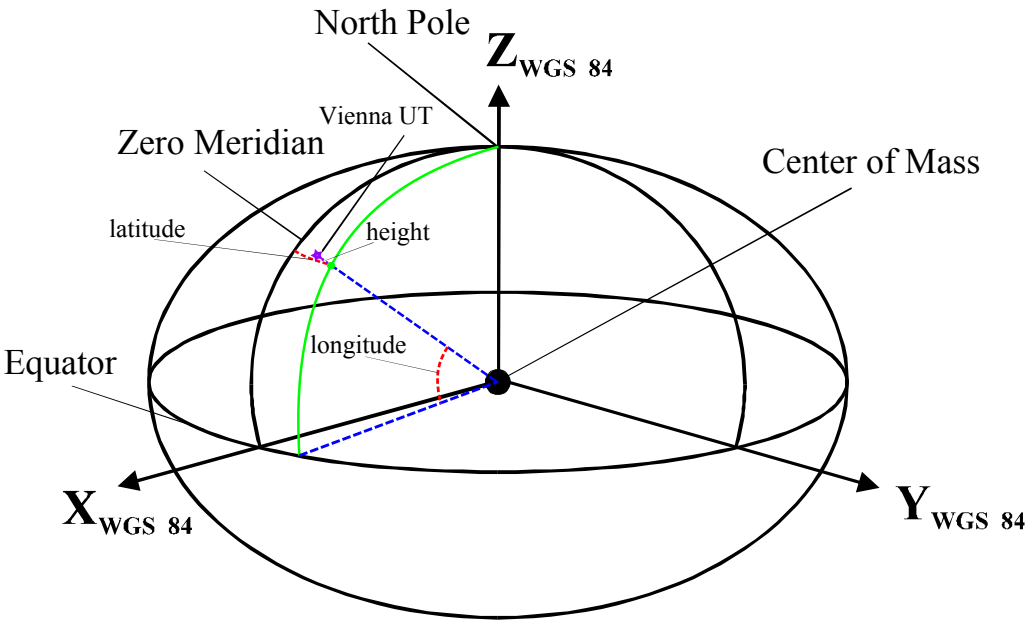


Figure 2.2: Definition of the WGS84 coordinate system (taken from [1])

Parameter	Name	Value (WGS84)	Unit
Semi-major axis	a	$6.378.137,000$	m
Semi-minor axis	b	$6.356.752,314$	m
Angular velocity	ω	$7,292115E-5$	$\frac{rad}{s}$
Geocentric gravitational constant	$G \cdot M$	$3.986.004,418E8$	$\frac{m^3}{s^2}$

Table 2.1: Parameters of the WGS84 ellipsoid

and semi-minor axis. The further parameters mentioned in Table 2.1 can be neglected for the geometric coordinate system.

If this cartesian coordinate system is used, a point can be determined by specifying cartesian coordinates $\sigma_x, \sigma_y, \sigma_z$ based on the tree axes X, Y, Z. However, an even better coordinate system exists to specify a point. By using spheric coordinates a point can be determined by two angles and the Euclidean distance from the origin. Since the reference ellipsoid (i.e., the shape of the earth) is defined by WGS84, the third parameter can be expressed as the distance from the desired point to the perpendicular point of the ellipsoid. This height can approximately be interpreted as the height over sea level.

Consider, for example, the position of the Vienna University of Technology. If spheric coor-

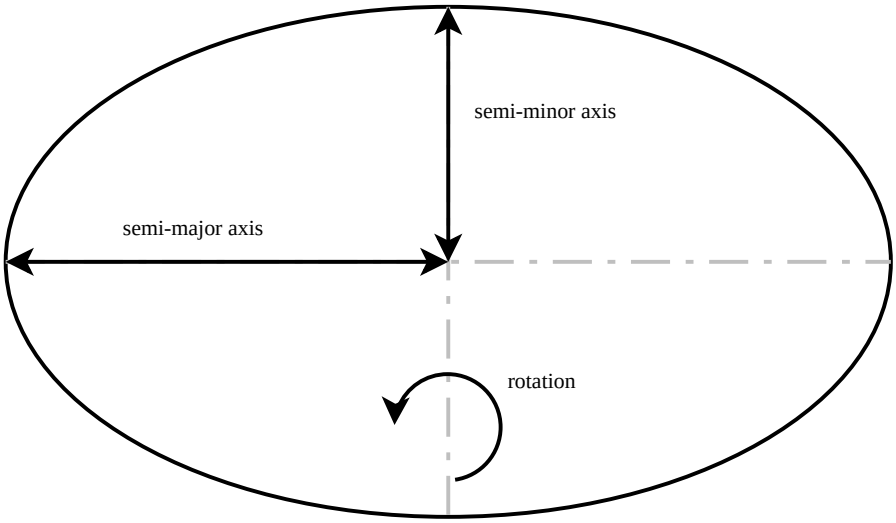


Figure 2.3: Parameters of a rotating ellipse

dinates are used, the position can be determined as

$$\begin{aligned} latitude &= 48.1989206° North \\ longitude &= 16.369913° East \\ height &= 170m \end{aligned}$$

The three coordinates represent the position w.r.t. the WGS84 reference ellipsoid. In contrast to this representation, the same point can be expressed by cartesian coordinates as follows:

$$\begin{aligned} \sigma_x &= 4086702,43588491m \\ \sigma_y &= 1200450,40780295m \\ \sigma_z &= 4731774,6142551m \end{aligned}$$

Based on these coordinates it can be computed, that the Vienna University of Technology is $6.366.467,543m$ away from the center of the earth.

2.2.2 Distance estimation

If points on the earth are determined by GPS modules, they are represented by spheric coordinates based on WGS84 in general. If one assumes that all points are near the ellipsoid, the height can be neglected for distance calculation because the latitude and longitude specify a point on the ellipsoid.

Since these two coordinates are only angles, it is not immediately possible to calculate the distance between two different points. However, this calculation and the calculation of the heading between two points are fundamental for motion planning.

There are several possibilities to calculate the distance and the heading between two points. The simplest method is to ignore that the spheric coordinates are based on an ellipsoid and to

assume that the coordinates specify a point on a sphere. Further, it can be assumed that the points are close together and so the difference between the Euclidean distance and the distance on the surface of the ellipsoid (i.e., the arc length) can be neglected. In this case, the distance can be evaluated by simple trigonometric functions.

Another precise possibility to evaluate the distance and the heading is presented in [14]. This method considers the exact figure of the WGS84 ellipsoid. However, one drawback of this method is that the computational effort is very high in contrast to the simple approach mentioned before. This drawback can lead to serious problems in practice, since motion control algorithms use this calculation frequently.

2.3 System models and constraints

Before a motion planning strategy can be applied, a system model of the target system is necessary. A system model is used to describe the behavior and the constraints of a system. Every feasible path or trajectory has to suffice constraints of the system. Consider the case of a CLR where not every path is feasible. Since system models determine the behavior of a system, they implicitly determine the constraints of a system, too. Thus, it is not necessary to specify the system constraints explicitly.

However, system models and system constraints for generic problems can be described in different ways.

- It is possible to describe a system with a set of rules of the form $if(\cdots)then\{\cdots\}else\{\cdots\}$.
- Another possibility is to describe a system in fuzzy logic [15].
- Furthermore, a system model can be specified by a set of difference equations (i.e., a time discrete system).
- Moreover, system models can be determined by a set of differential equations.

The most common method in robotics is the description of a system using a set of difference or differential equations. Thus, these methods are described in the next part of this section.

2.3.1 Time discrete models

In the following, a common form of difference equations is used to describe a time discrete, not necessarily linear time invariant system.

$$x_{k+1} = f(x_k, u_k) \tag{2.1}$$

In this equation, x_k represents the *state vector* that has the dimension n. Furthermore, u_k represents the input vector with dimension m. The *state vector* x_k represents the actual state of the system at a discrete time k and the input vector u_k represents the inputs at a discrete time k. In literature, u is also called *action space*. The function $f()$ is called *state transition function* and computes the actual state in the so called State Space (SS) $\mathcal{X}$ at time $k+1$ based on the previous state x_k and the input vector u_k at time k. The *state vectors* x_k and x_{k+1} are elements of the SS ($x_k, x_{k+1} \in \mathcal{X}$).

2.3.2 Differential models

Parametric model

Similarly to the time discrete system model, time continuous system can be defined. A time invariant not necessarily linear system model can be specified by a set of differential equations. Generally, such a set of equations can be written in the following form:

$$\dot{x}(t) = f(u(t), x(t)) \tag{2.2}$$

In this equation, $x(t)$ represents a state in the SS $\mathcal{X}$. The SS contains *state variables* and/or *velocity variables*. A *state variable* represents for example the position of an object. In contrast a *velocity variable* represents for example the velocity of an object. If the SS only consists of *state variables*, the SS is also called Configuration Space (CS) $\mathcal{C}$. If the SS contains both *state variables* and *velocity variables*, a function can be defined which maps the SS to the CS. This function is simply a projection from the SS to the CS. Furthermore, the *tangent space* can be defined as the deviation of the SS. Thus, the *state transition function* maps a state in the SS and a state in the *action space* to a state in the *tangent space*.

Furthermore, the Workspace (WS) $\mathcal{W}$ is defined as the physical space of the system. Consider for example an arbitrary object in the 3D Euclidean space. Obviously, the CS is defined as $\mathcal{C} = \mathbb{R}^3 \times [0, 2\pi[^3$ (where the last vector components represent the headings of the entity). The WS only considers the position of the system (i.e., $\mathcal{W} = \mathbb{R}^3$).

Moreover, the SS $\mathcal{X}$ and the *tangent space* $\dot{\mathcal{X}}$ can be combined to the Phase Space (PS) $\mathcal{X} \times \dot{\mathcal{X}}$. A state in the PS represents a complete state including its tangents in each dimension.

Finally, $u(t) \in U(x(t))$ represents a state in the *action space* of the system. In general, it is possible that every state $x(t)$ has a different set of possible actions $U(x(t))$. The *action space* can be defined as $U = \bigcup_{\forall x(t) \in X} U(x(t))$, the union of all possible actions.

Since the transition function is only able to consider the velocities, it is not possible to express accelerations or higher moments of a system with this model. In this case, the system model is expressed by

$$\ddot{x}(t) = f(u(t), x(t), \dot{x}(t)) \tag{2.3}$$

However, a method to express systems with higher order deviations by systems with only first-order deviations exists. It is possible to reduce the degree of a system by increasing the dimension of the system. Consider for example a two dimensional SS. Assume that a system with constraints on the velocity and the acceleration exists. The following SS model describes this system:

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \end{pmatrix} = f\left(\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}, \begin{pmatrix} x \\ y \end{pmatrix}, u \right) \tag{2.4}$$

Considering Equation 2.3 and Equation 2.4, note that $x(t) \neq x$. To reduce the degree of this system, the SS of the model can be extended by the first-order deviations of the states via $\chi := \dot{x}$ and $\gamma := \dot{y}$. The system model can now be written as follows.

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\chi} \\ \dot{\gamma} \end{pmatrix} = f' \left(\begin{pmatrix} x \\ y \\ \chi \\ \gamma \end{pmatrix}, u \right) \tag{2.5}$$

In this example, $(x, y, \chi, \gamma)^T \in \mathcal{X}$ represents the SS where $(x, y)^T \in \mathcal{C}$ represents the CS. Moreover, $(x, y, \chi, \gamma)^T \times (\dot{x}, \dot{y}, \dot{\chi}, \dot{\gamma})^T$ represents the PS.

Note that it is always possible to reduce a higher-order system to a first-order system. The resulting SS consists of state variables and velocity variables. Thus, a system with an n-dimensional CS and kth-order derivations can be reduced to a first-order system with a $(n \cdot k)$ - dimensional SS.

Implicit model

In contrast to the parametric model of Equation 2.2, a system model can also be expressed in another way. In this case, the system behavior itself is specified without considering an input vector. An implicit system model can be specified as

$$g(q, \dot{q}) = 0 \tag{2.6}$$

Under certain restrictions as mentioned in [7], it is possible to transform an implicit model to a parametric model. Note that the implicit model can also be described by a set of equalities and inequalities as shown in the following equation.

$$g(q, \dot{q}) \bowtie 0 \tag{2.7}$$

The $\bowtie$-operator is a placeholder for $=, <, >, \leq, \geq$. Since the system behavior can be expressed by a set of inequalities, it is clear that they are more powerful than the parametric model. However, this is out of the focus of this thesis because this kind of constraints is not necessary for the aimed systems.

2.3.3 Classification of constraints

In the previous section, system models and their constraints were introduced. In this section, a few systems with different kinds of constraints are examined. Nearly every physical system has any kind of constraints. Only in some rare cases, it is assumed that all constraints can be neglected. Constraints can be divided into kinematic constraints and dynamic constraints.

- Kinematic constraints are constraints on the CS or the first-order model. They only depend on the velocities of the CS or on the CS itself, i.e., a system model in the form of Equation 2.2 with no *phase variables* (i.e., $\mathcal{C} = \mathcal{X}$) is a kinematic model.

- If a system is defined as
$$\ddot{X}(t) = f(U(t), X(t), \dot{X}(t)), \tag{2.8}$$
the system has dynamic second-order constraints. Even if the order of the system is reduced by increasing the SS, the system remains dynamic.

System models can be divided based on the type of their constraints, too. These types have high impact on the complexity of motion planning approaches.

- **Unconstraint systems:**
  As mentioned before, completely unconstraint systems are very rare. Consider for example a free flying spheric object with zero mass. In this case, the action vector can be mapped directly to the SS ($x(t) = u(t)$). Since every dimension is independent from each other, the system is unconstraint.

- **Systems with holonomic constraints:**
  A holonomic constraint is a constraint which only depends on the time and on one variable of the CS. The constraint must not depend on any variable of the *tangent space*. If the constraints of a system are given in the implicit form as described in Equation 2.7, a holonomic system looks like:
  $$g(q) = 0. \tag{2.9}$$
  If it is possible to integrate the equation
  $$g(q, \dot{q}) = 0 \tag{2.10}$$
  such that $\dot{q}$ can be eliminated, the system is called absolute integrable (i.e., the constraint is holonomic).

  Consider again a spheric flying object with zero math (e.g., no dynamic forces) but restrict the movement of this object to the surface of a sphere with radius r (e.g., a satellite). In this case, the constraint can defined as
  $$(x^2 + y^2 + z^2) - r^2 = 0 \tag{2.11}$$
  where x, y, z are parameters of a three-dimensional cartesian coordinate system, i.e., the CS. Since this does not contain any parameter from the tangent space, the system suffices the claim above and therefore it is holonomic. Otherwise, the system is non-holonomic.

  One can show that every holonomic system in $\mathcal{C}'$ can be transformed to an unconstraint system based on a transformed CS ($\mathcal{C}''$) [7]. Thus, holonomic constraints do not increase the complexity for motion planning tasks but change the definition of the CS of the system. However, holonomic systems are not unconstraint systems.

- **Systems with non-holonomic constraints:**
  As mentioned before, non-holonomic constraints not only depend on values of the CS. They further depend on values of the tangent space. If a constraint is expressed by the following equation
  $$g(q, \dot{q}) = 0 \tag{2.12}$$
  and it is not possible to integrate this equation (i.e., to eliminate $\dot{q}$), the constraint is non-holonomic. In contrast to holonomic constraints, non-holonomic constraints increase the complexity for motion planning approaches dramatically.

Consider for example a rolling coin on a plane surface as shown in Figure 2.4. This system is better known as *unicycle model* in literature. The parametric model can be defined as

$$\dot{q} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}. \tag{2.13}$$

The corresponding implicit model of this system can be defined as

$$\dot{x} \cdot \sin(\theta) - \dot{y} \cdot \cos(\theta) = 0. \tag{2.14}$$

Obviously, the equation can not be integrated such that $\dot{x}$ and $\dot{y}$ can be eliminated. The CS for this system is $\mathcal{C} = \mathbb{R}^2 \times [0, 2\pi[$, (x, y, θ) where the first two dimensions represent the position on the plane and the last dimension represents the orientation of the disk w.r.t. the inertial system. It is clear that the object is able to move in two directions (forward, backward) but it is not possible that the disc moves sidewards. This is called a non-holonomic constraint.

If the system has non-holonomic constraints, it is much more difficult to find a feasible path or even a trajectory and therefore the complexity of the planning task is more difficult than in the holonomic case. However, most real-world systems have non-holonomic constraints. Hence, the following chapters of this thesis deal with non-holonomic systems.

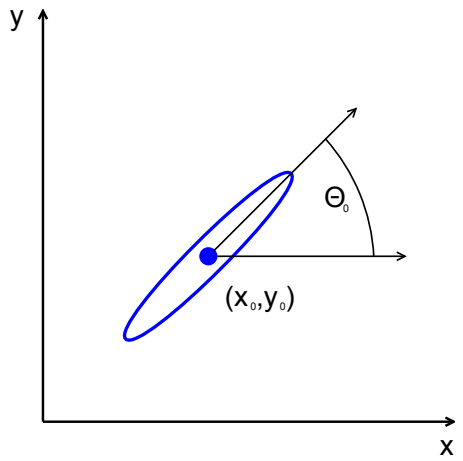


Figure 2.4: System configuration of a unicycle

2.4 Objectives of motion planning

Before motion planning approaches can be discussed, the desired result of a motion planning application has to be determined. The following objectives of motion planning are supposable:

- A geometric path in the WS or in the CS can be aimed.

- If a geometric path is not sufficient, a trajectory in the CS can be required.
- Another possibility is that an action trajectory in combination with a trajectory in the CS is aimed.
- In some rare cases, it is possible that only an action trajectory is necessary to control a system.
- However, it is possible that (in case of a discrete system) a consecutive sequence of states (i.e., including input and SS) depending on the time is desired.

Since in general either a geometric path or a trajectory in the CS is aimed, the differences between these two curves are presented as follows.

2.4.1 Trajectory vs. geometric path

- A trajectory is an integral curve $g(t)$ on an arbitrary SS (e.g., for a CLR) which suffices the *state transition function*. As mentioned before, a trajectory can be computed by integrating the state equation function based on an action trajectory and an initial state in the SS ($x_0 = x(t_0)$).

  If an action trajectory $u(t)$ (i.e., an arbitrary time dependent curve in the *action space*) and an initial state $x(t_0)$ are given, the state trajectory can be calculated as follows:

$$x(t) = x(t_0) + \int_{t_0}^{t} f(x(t'), u(t'))dt' \tag{2.15}$$

- A geometric path is a curve in 2D or 3D Euclidean space. This means that the WS is equal to the Euclidean space (i.e., $\mathbb{R}^2 = \mathcal{W} \subseteq \mathcal{C}$ or $\mathbb{R}^3 = \mathcal{W} \subseteq \mathcal{C}$). In contrast to a trajectory which is a time dependent curve, a geometric path $g(u)$ depends not on the time t but only on a monotonic increasing variable u.

2.4.2 Conversion of curves

- **From trajectory to a geometric path**
  Since a trajectory contains more information than a geometric path, it is possible to generate a geometric path based on a trajectory. This can be done by a function which maps the SS to the two- or three-dimensional Euclidean space ($p : \mathcal{C} \mapsto \mathbb{R}^3$). In many cases, the mapping can be done by a simple projection of the higher dimensional trajectory to the Euclidean space. Since a trajectory depends on a time variable which is obviously monotonic increasing, the time variable of a trajectory can be mapped directly to a geometric path ($t \mapsto u$).
- **From a geometric path to a trajectory**
  In general, it is not possible to derive a trajectory from a geometric path. In some cases, it is even possible to derive a trajectory from a geometric path. In Section 4.2.1 an example is presented, where this is possible. The challenge in this case is to map the variable u to

the time variable t. A wrong mapping $u \mapsto t$ can lead to an infeasible trajectory, even if the geometric path is feasible.

2.5 Complexity of motion planning

While in the previous section the desired result of a motion planning process was discussed, this section deals with the complexity of motion planning problems. In general, a lot of different motion planning problems exist. To determine the quality of a motion planning approach, it is very helpful to know lower bounds on execution time or memory consumption for a given motion planning problem.

Due to the amount of different motion planning problems, providing a proof for the lower bound for each planning problem is not possible. Therefore, a few classes of motion planning problems are presented for which lower bounds were already specified and proved.

- One of the most famous motion planning problem is the *piano mover's problem* [16]. The challenge of this problem is to move a polyhedral object (e.g., a piano) from an initial position (i.e., a state in the CS) to a target position in an unbounded CS in presence of polyhedral obstacles (e.g., walls or tables). Note that if a bounded CS is desired, then it can be modeled with a number of polyhedral obstacles which limit the CS around the initial and the goal point. Moreover, this problem asks for an arbitrary path and not for the shortest path.

  The *piano mover's problem* can be generalized by adding free flying polyhedral rigid objects to the robot. An example for such a system is a robot arm where every part can be moved freely. This problem is called the *generalized mover's problem* ([16]). This specific problem seems to be very difficult to solve. In fact a lower bound proof is presented in [16] which shows that the *generalized mover's problem* is PSPACE-hard.

- Another motion planning problem which is relevant for this thesis deals with the motion of a robot with non-holonomic constraints. The challenge of this problem is to find a *curvature constrained shortest path* in a two dimensional WS. Furthermore, the WS is restricted by arbitrary many polygonal obstacles. Intuitively, this problem is quite hard to solve. In [17], a lower-bound proof for this problem is provided with the result that this problem is NP-hard. This problem is very similar to the motion planning problem for CLRs. The challenge in this case is to navigate a CLR meeting all kinematic constraints from an arbitrary initial position to a goal position in presence of arbitrary obstacles. These obstacles are represented by rigid polygons. In Chapter 4, a few motion planning algorithms are presented to solve this problem. Since it was shown that the described problem is NP-hard, no complete solution exists which solves the problem efficiently.

CHAPTER 3

Motion planning algorithms

3.1 Overview

In the previous chapter, the basic concepts of system theory were introduced. This chapter presents a few basic motion planning algorithms. These algorithms are used to solve the basic motion planning problem (i.e., finding a path between two arbitrary states). However, these algorithms are described in a generic way and they are not applied to a specific system.

At the beginning, a discrete naive approach is presented to solve the motion planning problem. Next, a few roadmap based motion planning approaches are examined. To overcome the complexity of motion planning, the last part of this chapter introduces probabilistic approaches to solve the motion planning problem.

3.2 Discrete motion planning algorithms

The following section deals with the basic aspects of discrete motion planning. Before motion planning strategies are described, the basic problem is formalized as follows:

- Let $\mathcal{X}$ be an n-dimensional discrete SS where the dimension is finite or countable infinite.
- Furthermore, define for each state $x \in \mathcal{X}$ in the discrete SS a specific input vector u from the so called action state $u \in U(x)$. Note that in general the *action space* can be different for each element $x \in \mathcal{X}$. Furthermore, define $U = \bigcup_{\forall x \in \mathcal{X}} U(x)$ as the complete *action space*.
- The state transition function $x' = f(x, u)$ (as described in Section 2.3) defines the transition from a state $x \in \mathcal{X}$ to a state $x' \in \mathcal{X}$ depending on the input vector $u \in U(x)$ where $U(x) \subseteq U$.
- Finally, define $x_{init} \in \mathcal{X}$ as the initial state and $X_{goal} \subseteq \mathcal{X}$ as the set of goals of the motion planning problem.

In addition to the previously mentioned definitions, a few more definitions are necessary, if an optimal path is desired. A path is called optimal if and only if no other feasible path exists with lower costs (i.e., a shorter path).

- Let l be a function $l : (\mathcal{X}, \mathcal{X}) \mapsto \mathbb{R}^+$ which represents the length (or the costs) between two states in $\mathcal{X}$. Since the system is deterministic, the function l can also be defined as $l : (\mathcal{X}, U) \mapsto \mathbb{R}^+$.
- Subsequently, it is assumed that the result of the planning process is a sequence of input vectors for the system $\pi = (u_0, u_1, ..., u_k)$. If the initial state x_{init} is given, the complete sequence of states in $\mathcal{X}$ can be derived.
- Furthermore, let L be the cost function which represents the costs for a plan π and define it as $L = \sum_{k=0}^{K-1} l(x_k, u_k)$ for a plan with length K.

In some cases, it is not necessary to know the sequence of action vectors of a system because a low level feedback controller generates the action vector during runtime depending on the current state and the estimated state. However, this is described in a later chapter.

The given formulation can also be described by a weighted undirected graph $G(V, E, l, v_{init}, V_{goal})$. In addition to the common definition of the graph, a distinct vertex $v_{init} \in V$ exists which denotes the initial node of the graph. Moreover, a state v_{goal} or even a set $V_{goal} \subseteq V$ represents the goal vertices. Furthermore, the function l is defined similarly to the above definition: $l : E = (V, V) \mapsto \mathbb{R}^+$. If a feasible path between two vertices v_k and v'_k exists, l is defined as $0 \leq l(v_k, v_{k'}) < \infty$ and if no path exists $l(v_k, v_{k'}) = \infty$.

In the following section, an introduction about basic search strategies is given and furthermore an overview on the most popular graph search algorithms in context of motion planning is presented.

3.2.1 Graph search algorithms

Graph search algorithms offer a systematic way to find a not necessary optimal path from a state x_{init} to a set of goal states X_{goal}. Graph search algorithms are a special case of greedy algorithms and have a common scheme as described in [7]. Algorithm 1 shows an approximate scheme of a graph search algorithm. Most of the graph search algorithms are based on this concept. Basically, this search approach separates all vertices in three distinct sets.

- **Unvisited nodes**
  This set contains nodes which were not considered yet. Thus, they are neither in P nor in Q.
- **Dead nodes**
  All the dead nodes are in the set P. These nodes were still processed by the algorithm and therefore $P \cap Q = \emptyset$.
- **Alive nodes**
  These nodes are represented by the sequence Q of Algorithm 1. This means that they are

not processed but they were appended to the queue Q. A node can only be in Q if one of its neighbors is dead (i.e., in P).

At the beginning, the set P is empty and the sequence Q only contains the initial node. The algorithm pops the first element from Q. Note that Q is a sequence of unprocessed nodes. Depending on the concrete search algorithm, the order within this queue is defined differently. If one node is removed from Q it is added to P and furthermore all possible successor nodes are checked. Since for every node x a set of actions $U(x)$ exists, every possible action is applied to reach a possible successor x'. If such a successor node is neither in Q nor in P (i.e., not visited yet), it is added to Q. This step is performed for every possible successor x' of x.

Algorithm 1 FORWARD_SEARCH (taken from [7])

```
1:  P ← ∅
2:  Q.Insert(x_I)
3:  while Q ≠ ∅ do
4:     x ← Q.popFirst()
5:     P ← P ∪ x
6:     if x ∈ X_G then
7:        return SUCCESS
8:     end if
9:     for all u ∈ U(x) do
10:       x' ← f(x, u)
11:       if x' ∉ Q ∧ x' ∉ P then
12:          Q.Insert(x')
13:       end if
14:    end for
15: end while
```

The presented algorithm shows the basic scheme of graph search approaches by means of a forward search algorithm. However, a few slightly different concepts exist. These concepts can be distinguished on the direction of the search. Depending on the motion planning task, some may have more benefits than others.

- **Forward search**
  This method is the most common one. The search algorithm starts in the initial state x_{init} and explores the search space based on the system model until a state within the set of goal states X_{goal} is reached.

- **Backward search**
  This method starts from a goal state $x_{goal} \in X_{goal}$ and explores the SS backwards. However, in this case it is necessary to transform the system model because a function $g = f^{-1}$ is needed which calculates the predecessor x of x' based on x' and $u \in U(x)$. Note that f represents the state transition function as introduced in Section 2.3.1.

- **Bidirectional search**
  In some cases, it is useful to combine forward search with backward search. If the two trees meet each other the algorithm terminates.

In the previous part of this section, the basic properties and distinctions of search algorithms were considered. In the following part of the section, a few popular search algorithms are presented. These algorithms are able to find a path in an efficient way. Furthermore, the results of these algorithms are optimal and complete in the discrete space which means that the algorithm finds an optimal solution, if one exists. Otherwise, it fails.

Dijkstra's Algorithm

One of the most famous algorithms in graph theory is the so called Dijkstra algorithm. Normally, this algorithm is applied on an undirected, weighted graph $G = (V, E, s, t)$. However, in this case the algorithm is adapted to solve a discrete motion planning problem as described before. The only difference is the selection of the neighbor nodes (states). In a normal graph, these nodes can be selected directly but in this case all possible actions $U(x)$ have to be applied on a state x to get all successor states x' according to the discrete transition function $x' = f(x, u),\ \forall u \in U(x)$.

In contrast to the basic search scheme, Dijkstra's algorithm does not need a set P. This set is replaced by using the $dist$ attribute of each state. Every discrete state x has an attribute $x.dist$ which denotes the distance to the initial state x_{init}. If a state is unprocessed (i.e., no path to x_{init} exists yet) this attribute is set to infinity. Furthermore, every state has an attribute $x.pred$ which stores the way back to its predecessor (i.e., a pointer to the parent node). If the node is still unprocessed, $x.pred$ is set to NIL. Algorithm 2 illustrates Dijkstra's algorithm in context of discrete motion planning.

A* Algorithm

An extension to Dijkstra's algorithm is the A* algorithm. The basic concept of the A* algorithm is quite similar. In addition, a heuristic is used by the A* algorithm to estimate the distance from an arbitrary state to the goal state (or a set of states). Based on the motion planning problem and the SS, there are many possibilities to realize such a heuristic. However, the quality of the heuristic is immanent for the performance of the algorithm. In case of an Euclidean SS without obstacles, the Euclidean distance to the goal state(s) can be used as heuristic. Algorithm 3 shows the concept of the A* algorithm. The only difference in contrast to Dijkstra's algorithm can be found in Line 16. The decision about the next state is based on the distance to the root and further on the estimated distance to the goal.

3.2.2 From continuous systems to discrete systems

If a discrete motion planning approach is used in practice, a strategy is necessary to discretize the continuous system to a discrete system. Obviously, the SS has to be quantized but it is also necessary to adapt the state transition function which is the more challenging task.

Algorithm 2 Dijkstra's Algorithm

1: $\forall x \in \mathcal{X} \backslash x_{init} : x.dist \leftarrow \infty$
2: $x_{init}.dist \leftarrow 0$
3: $\forall x \in \mathcal{X} : x.pred \leftarrow NIL$
4: $Q \leftarrow \{x_{init}\}$
5: **while** $Q \neq \emptyset$ **do**
6: $x \leftarrow \{q' \in Q | q'.dist \leq q''.dist \, \forall q'' \in Q\}$
7: $Q \leftarrow Q \backslash x$
8: **if** $x.dist = \infty$ **then**
9: return FAILURE
10: **end if**
11: **if** $x \in X_G$ **then**
12: return SUCCESS
13: **end if**
14: **for all** $u \in U(x)$ **do**
15: $x' \leftarrow f(x, u)$
16: $d \leftarrow x.dist + dist(x, x')$
17: **if** $d < dist(x')$ **then**
18: $Q \leftarrow Q \cup x'$
19: $x'.dist \leftarrow d$
20: $x'.pred \leftarrow x$
21: **end if**
22: **end for**
23: **end while**

Transformation of the CS

Independent from the strategy, a discretized motion planning algorithm can never be complete or optimal in general. To illustrate this fact, consider a WS with a narrow passage. Furthermore, assume that the WS is transformed to a discrete WS such that there is no state within this passage. Thus, a path in the original system exists but no path in the discrete system. However, if the density of the states is increased, the possibility that a path is found increases but there is no limit for the narrowness.

In practice, the choice of the strategy is very important for the quality of the transformation and the performance of the search algorithm. Consider the case where lots of obstacles exist in one part of the CS but no obstacle is located in the rest of the CS. Obviously, the density of states in the constrained portion of the space should be higher than in the free space. A uniform dense state lattice will lead to a bad performance of the search algorithm but a coarse state lattice can lead to an incomplete SS.

Transformation of the transition function

Another challenging task is the transformation from a parametric differential transition function to a discrete transition function. The problem is that such a discretization can lead to integral

Algorithm 3 A* Algorithm

1: $\forall x \in \mathcal{X} \backslash x_{init} : x.dist \leftarrow \infty$
2: $x_{init}.dist \leftarrow 0$
3: $\forall x \in \mathcal{X} : x.pred \leftarrow NIL$
4: $Q \leftarrow \{x_{init}\}$
5: **while** $Q \neq \emptyset$ **do**
6: $x \leftarrow \{q' \in Q | q'.dist \leq q''.dist \, \forall q'' \in Q\}$
7: $Q \leftarrow Q \backslash x$
8: **if** $x.dist = \infty$ **then**
9: return FAILURE
10: **end if**
11: **if** $x \in X_G$ **then**
12: return SUCCESS
13: **end if**
14: **for all** $u \in U(x)$ **do**
15: $x' \leftarrow f(x, u)$
16: $d \leftarrow x.dist + dist(x, x') + h(x, X_G)$
17: **if** $d < dist(x')$ **then**
18: $Q \leftarrow Q \cup x'$
19: $x'.dist \leftarrow d$
20: $x'.pred \leftarrow x$
21: **end if**
22: **end for**
23: **end while**

errors which change the system behavior immanently. Thus, it is possible that this behavior leads to infeasible paths.

However, these problems are out of the scope of this thesis since they are avoided by all further mentioned motion planning approaches. In Chapter 4, a few special methods are presented where discrete methods are used to solve continuous motion planning problems.

3.3 Motion planning based on roadmaps

Roadmap based motion planning approaches belong to the most common motion planning approaches. Intuitively, a roadmap is constructed considering the environment including obstacles. After a roadmap is constructed, it is easy to find a path using common methods (e.g., graph search).

As mentioned in the previous section, discretized motion planning approaches are not complete or optimal in general. However, roadmap based approaches can be complete and possibly optimal. The reason is that if a CS is discretized, arbitrary configurations are sampled (e.g., state lattice). In case of roadmap based approaches, the CS is decomposed and the configurations are sampled using special approaches.

3.3.1 Visibility graph method

Consider a two-dimensional WS with arbitrary rigid obstacles. Assume that the WS equals the CS. All of the obstacles are represented by simple polygons. Note that a finite CS can be expressed by a set of obstacles surrounding the CS. A naive approach is used to connect every vertex (including the start and goal vertex) with each other if and only if there is no obstacle between them. This method is called *visibility graph method*. An example of such a visibility graph is shown in Figure 3.1. Finally, a graph search algorithm can be used to find a possibly optimal path between the initial state and the goal state.

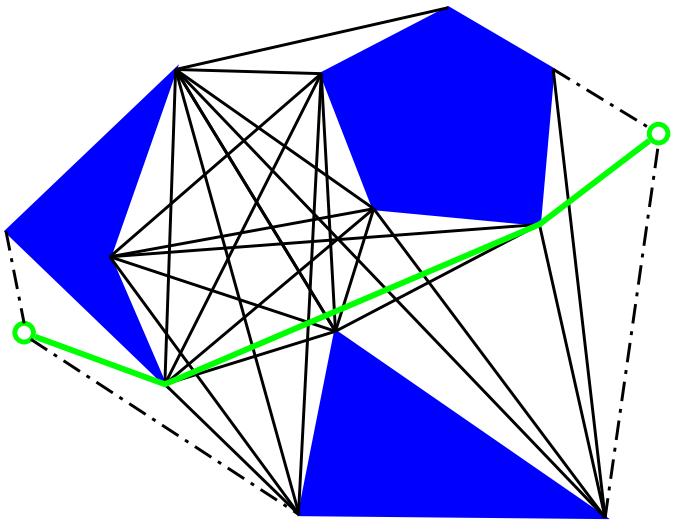


Figure 3.1: Example of visibility graph (taken from [2])

This algorithm can also be used for higher dimensional WS and CS. In this case, the algorithm does not provide an optimal solution. Hence, it is not suitable for higher dimensions as mentioned in [7].

If the target system has non-holonomic constraints (i.e., $\mathcal{W} \neq \mathcal{C}$), these constraints have to be considered and therefore the visibility graph has to be extended. For example, if the CS is defined as $\mathcal{C} = \mathbb{R}^2 \times [0, \pi[$ and $\mathcal{W} = \mathbb{R}^2$ then the third dimension must be discretized to n different values. For every node in the visibility graph, n nodes are added to the search graph. Finally, an edge between two states in the search graph is added, if and only if an edge between the two corresponding nodes exists in the visibility graph and the edge is feasible w.r.t. the non-holonomic constraints. Since the third dimension of the CS is discretized, no complete algorithm exists as mentioned in Section 3.2.2. Thus, it is not guaranteed that a solution can be found even if one exists.

3.3.2 Retraction approach

Another roadmap based approach uses *Voronoi diagrams* ([3]) to construct a roadmap. Given a set of n states in an Euclidean space, a Voronoi graph is a partition of the space into $n+1$ different

regions. The regions are limited by straight line segments. Every given state is the center point of one Voronoi region or Voronoi cell and so all line segments have the maximum possible distance to the center points. This kind of Voronoi diagram determines regions defined by straight line segments around points. An example of a Voronoi diagram is shown in Figure 3.2. However, Voronoi diagrams can also be extended to *generalized Voronoi diagrams*. In contrast to simple Voronoi diagrams, the regions are based on convex polygons which are represented by vertices and edges. In this case, the Voronoi regions are limited by straight lines and parabolic arcs. Based on a *generalized Voronoi diagram*, a path graph can be computed as shown in Figure 3.3. Further details about Voronoi diagrams can be found in [3, 18].

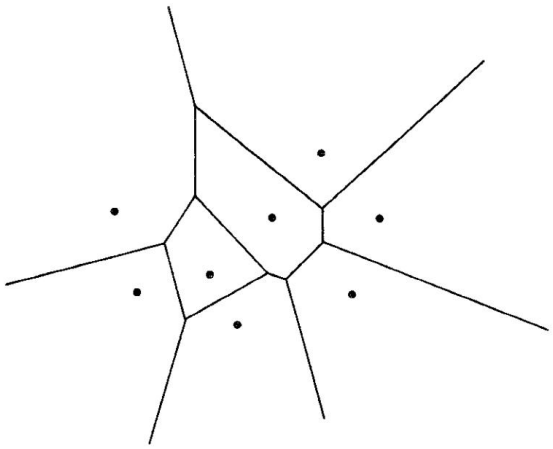


Figure 3.2: Example of a simple Voronoi diagram (taken from [3])

After constructing the Voronoi graph, a path from the initial state to the closest element of the Voronoi graph and a path from the Voronoi graph to the goal state has to be found as described in [2]. If the WS does not change, the Voronoi diagram has to be calculated only once. Thus, this algorithm is well suited for multiple query applications.

3.3.3 Cell decomposition

Another motion planning concept is cell decomposition. Assume that a finite CS exists and all obstacles are represented by simple polygons. The idea is to partition the free space $\mathcal{C}_{free}$ in convex polygons. Based on these polygons, a roadmap can be constructed and a free path can be found by simple graph search.

The CS is limited to $\mathcal{C} = \mathbb{R}^2$ for the following motion planning approaches for simplicity but it is possible to use these approaches also for higher dimensional CS (e.g., $\mathcal{C} = \mathbb{R}^3$). In this case, polygonal obstacles are replaced by polyhedral obstacles. Furthermore, it is assumed that only point-like robots are considered. If polygonal robots are considered, the CS must be extended.

However, two different types of methods exist. One possibility is to partition the CS in exact cells. Thus, this method is called *exact cell decomposition*. The other method is to partition the CS in predefined segments (e.g., rectangles) depending on the CS. The latter method is called *approximate cell decomposition*.

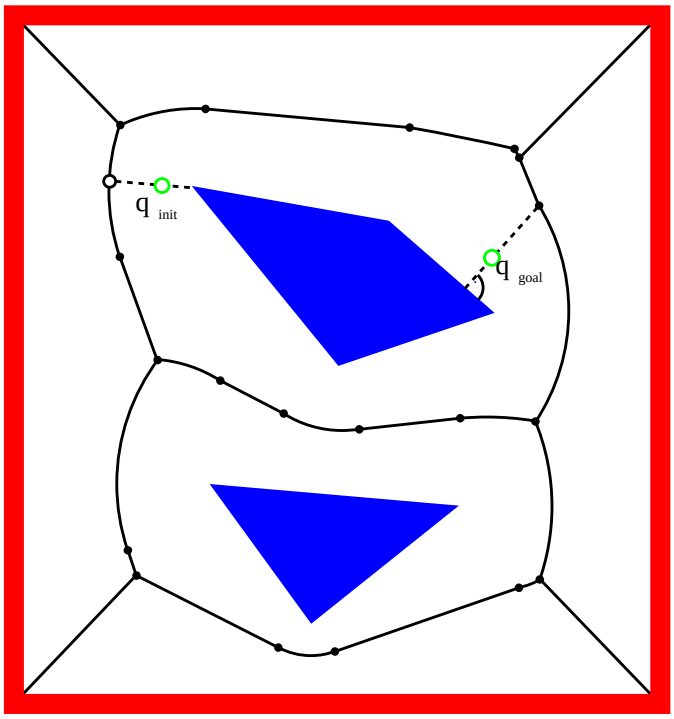


Figure 3.3: Example of a Voronoi based path graph (taken from [2])

Exact Cell Decomposition

As mentioned before, the CS is divided into convex polygons as shown in Figure 3.4. This method is basically intended for $\mathcal{C} = \mathbb{R}^2$ but it is possible to generalize this method to higher dimensional spaces. However, the following subsection concentrates on a two dimensional CS. Based on this cell-partition, a sequence of adjacent cells has to be found from the initial cell to the goal cell. Obviously, the initial cell contains the initial configuration q_{init} and the goal cell contains the configuration q_{goal}. This can be done by constructing a graph where every vertex represents a cell and two vertices are connected with an edge if and only if the corresponding cells are adjacent. Based on this graph, a graph search algorithm can be used to find a sequence of cells. If an optimal path is desired the edges have to be weighted (e.g., with the Euclidean distance). In addition an optimal graph search algorithm has to be used (e.g., A*) to determine a sequence of nodes.

Based on such a sequence, a path can be found as follows. The cut between two adjacent cells is a non-zero line segment which is also called borderline or face. A feasible path can be generated, if all borderlines of the consecutive cells are bisected and these states are connected in the same order as the corresponding graph nodes. Finally, the initial state and the goal state have to be connected. Since all cells are convex by assumption and the path starts and ends at the border of the appropriate cell, no obstacle can be in between. One possible but not necessary optimal path is shown in Figure 3.4.

Since this partition of the WS is very inefficient, another approach can be used to partition

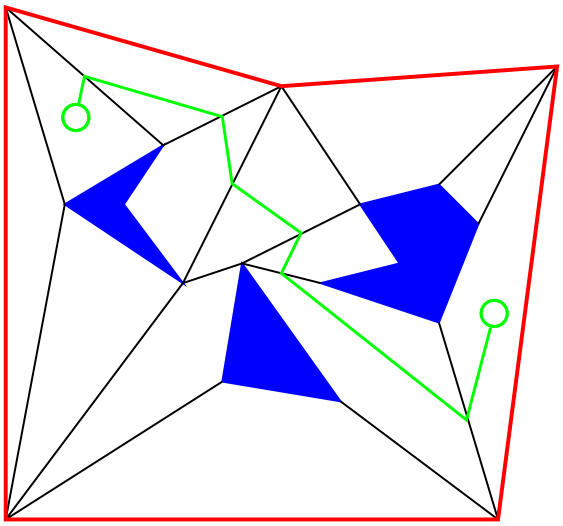


Figure 3.4: Example of path generated by exact cell decomposition (taken from [2])

a WS. This approach is computational efficient and can be used in practice. One drawback of this approach is that the partition is not optimal. The approach is based on a sweep line line algorithm. The decomposition of a WS is shown in Figure 3.5.

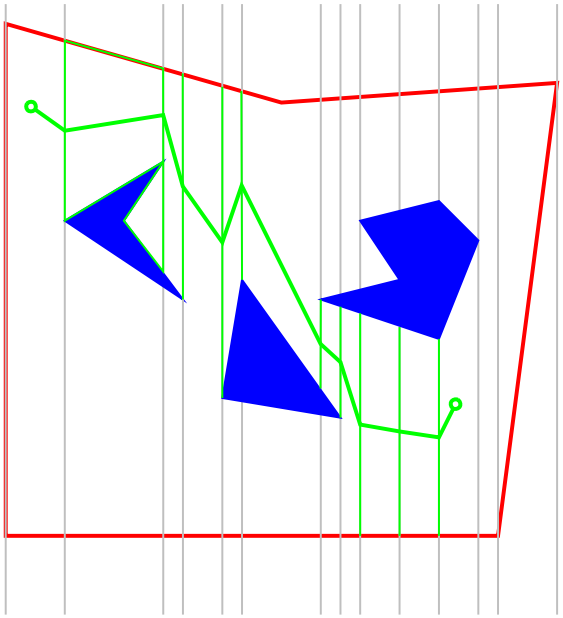


Figure 3.5: Exact cell decomposition using sweep line algorithm (taken from [2])

This algorithm uses a sweeping line which moves over the WS with a straight frontier and stops at each vertex of an obstacle polygon. The resulting polygons are either triangles or trape-

zoids in case of a two dimensional space. In Figure 3.5, these segments are marked green. Similar to the above method, a search graph is computed. Furthermore, a path can be created by bisecting the intersecting lines between two consecutive cells.

Approximate Cell Decomposition

In contrast to the exact cell decomposition, the approximate cell decomposition divides the WS into rectangloid subsegments. Since it is not possible to delimit the free space of the WS from the obstacles exactly, three different types of cells can be distinguished:

- **EMPTY cell**
  An empty cell lies completely in the free space of the WS.
- **FULL cell**
  A full cell is a cell which lies completely within an obstacle.
- **MIXED cell**
  A part of a mixed cell lies in the free space of the WS and obviously the other part of the cell is a part of an obstacle.

If it possible to find a sequence of adjacent EMPTY cells (i.e., from the cell containing the initial state to a cell that contains the goal state) such that only free cells are passed, a feasible path can be generated and the algorithm succeeds. If no such sequence can be found, mixed cells have to be decomposed to smaller rectangloid segments. These segments are characterized as enumerated above. Subsequently, it is checked again if a sequence of empty cells from the initial state to the goal state exists and so on. Thus, the algorithm starts from a single mixed cell and iteratively decomposes mixed cells until a path is found.

The method described above constructs a hierarchical structure of the CS. This structure can be expressed by a hierarchy tree as shown in Figure 3.6 (the corresponding WS is shown in Figure 3.7). Since it is not necessary to decompose empty or full cells, only mixed cells are decomposed and therefore only these cells can have children in the hierarchy tree. Note that by convention every mixed cell is decomposed to a fixed number of rectangloid segments. In general, it is not possible to exactly decompose a cell such that only full and empty cells remain. Thus, the resolution of the decomposition is limited. This limitation corresponds with the height of the hierarchy tree. Nevertheless, it is assumed that most of the cells of Figure 3.6 can be decomposed exactly and only two mixed cells reach the resolution limit.

3.4 Motion planning using potential fields

Most of the previous mentioned motion planning approaches tried to decompose the WS and deduce a search graph. In this section, another approach is presented which maps the WS to a potential field to find a feasible path based on a gradient field as introduced in [2]. To simplify the explanation of this method, the CS is restricted to $\mathcal{C} = \mathbb{R}^2$. However, it is possible to use this method also in higher dimensional CS. Furthermore, it is assumed that a point-like robot or a robot with fixed orientation is considered.

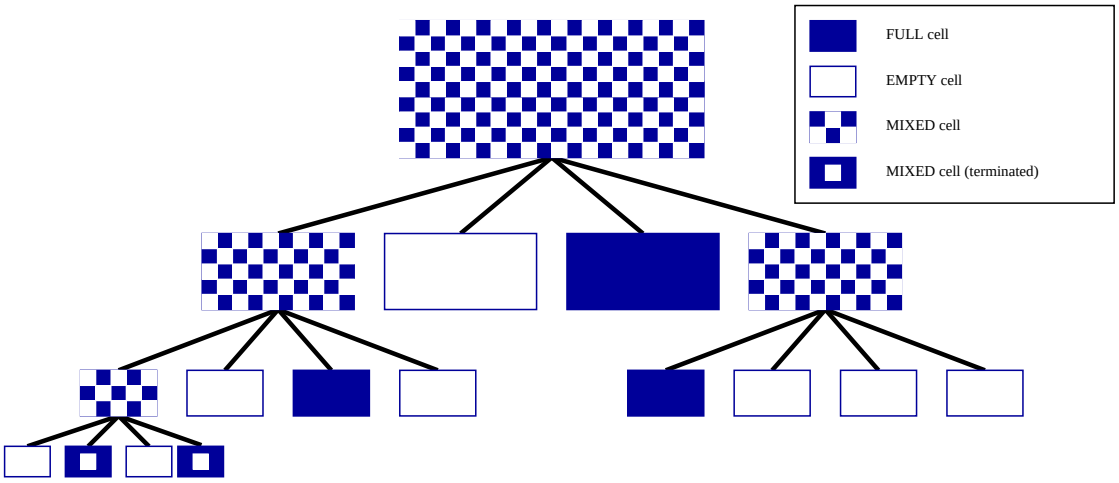


Figure 3.6: Decomposition tree

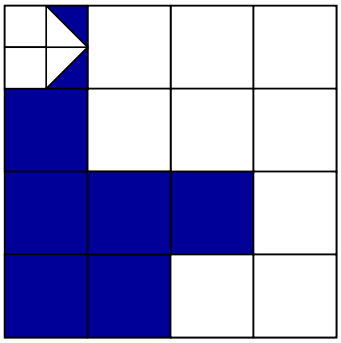


Figure 3.7: Decomposition of a two-dimensional WS

Based on a scalar field (i.e., a potential field $U : \mathbb{R}^n \mapsto \mathbb{R}$), a gradient vector can be defined for each configuration in the CS. This vector determines the slope of the actual configuration in the scalar field as length of the vector. Furthermore, the vector directs to the maximum slope of the scalar field. Thus, a gradient field is a function on the scalar field and defines a gradient vector for every point in the field. Formally, a gradient field is denoted by ∇ which is also known as the nabla operator. Consider a scalar field based on $\mathbb{R}^2$ that is defined as $U(x, y)$. The gradient field which is denoted by $\nabla U(x, y)$ is defined as

$$\nabla U(x, y) = \begin{pmatrix} \frac{\partial U(x,y)}{\partial x} \\ \frac{\partial U(x,y)}{\partial y} \end{pmatrix}. \tag{3.1}$$

To use this concept for motion planning, the WS has to be mapped to a scalar field fulfilling the following constraints:

1. The goal configuration must have the lowest value in the WS i.e., $U(x_{goal}, y_{goal}) < min(U(x, y)) \ \forall x, y : \ x \neq x_{goal} \wedge y \neq y_{goal}$.

2. Furthermore, the obstacle regions must have much higher potential than the free regions.

3. Finally, $U(x, y)$ must decrease towards the goal state.

Moreover, it is necessary that the resulting function is piecewise differentiable.

Until now, only a two dimensional CS was considered for simplicity. However, in the following paragraphs the CS is generalized. Therefore, a state within the CS is denoted by $q \in \mathcal{C}$. To fulfill these conditions, the resulting scalar field can be computed as the combination of two different scalar fields.

- **Attractive potential**
  The attractive potential field does not consider the obstacle regions. It simply reduces the potential towards the goal configuration.

$$U_{att}(q) = \frac{1}{2} \cdot \xi \cdot \rho_g^2(q) \tag{3.2}$$

  In Equation 3.2, $\rho_g(q)$ represents the Euclidean distance between q and q_{goal}, i.e., $\rho_g(q) = \|q - q_{goal}\|$. Furthermore, ξ represents a positive scaling factor. An example of an attractive potential function is shown in Figure 3.8. In this example, the configuration $q = (10, 40)$ represents the goal configuration. Note that the potential function fulfills condition (1) and (3) as desired above.

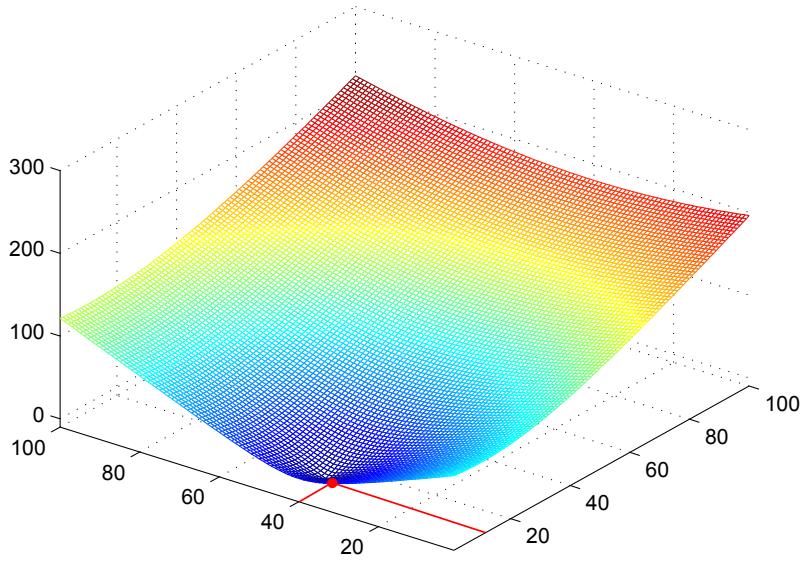


Figure 3.8: Example of an attractive potential function

- **Repulsive potential**
  In contrast to the attractive function, the repulsive function does not consider the goal configuration and is only based on the obstacles. Formally, this function is defined as:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{\rho_{ob}(q)} - \frac{1}{\rho_0})^2 & \text{if } \rho(q) \leq \rho_0 \\ 0 & \text{if } \rho(q) > \rho_0 \end{cases} \tag{3.3}$$

  In this equation, ρ_{ob}(q) represents the minimum distance from q to any obstacle, i.e., $\rho_{ob}(q) = min_{q' \in \mathcal{O}} \|q - q'\|$ where $\mathcal{O}$ represents the obstacle regions of a CS. Furthermore, η represents a positive scaling factor and ρ_0 represents a constant threshold.

  This function evaluates to zero, if the distance to the next obstacle is greater than a given threshold. Otherwise the function increases quadratically. If the distance is zero (i.e., a configuration within an obstacle), the function evaluates to a constant value. Figure 3.9 shows an example of a repulsive potential function with three circular obstacles. Note that this function suffices condition (2).

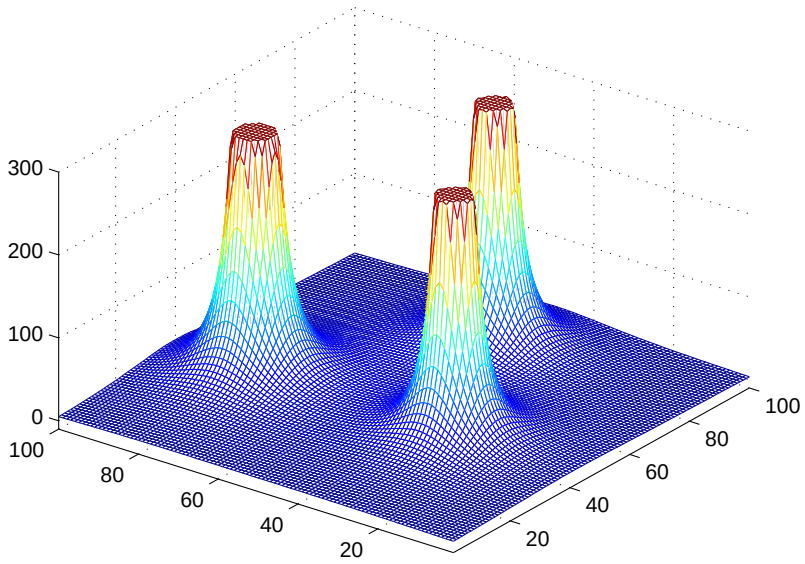


Figure 3.9: Example of a repulsive potential function

Finally, these two potential fields are combined to a resulting potential field $U = U_{rep} + U_{att}$ fulfilling all conditions ((1)-(3)) as mentioned before. Based on Figure 3.8 and Figure 3.9, the resulting potential function is shown in Figure 3.10.

Based on the definition of $U(q)$, a vector function F can be defined as the inverse gradient field of U: $F(q) = -\nabla U(q)$. The inversion is necessary, since the vector of a gradient field points to the highest slope in general.

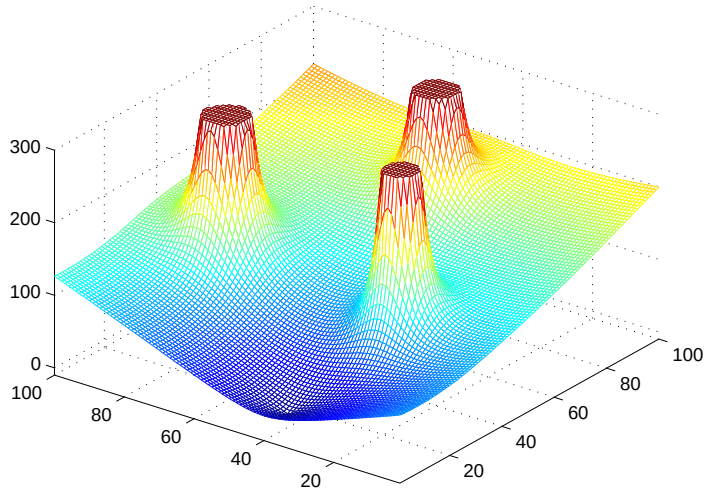


Figure 3.10: Example of a complete potential function base on an attractive and a repulsive function

After computing the gradient field based on a potential field as described above, a motion planning approach is desired that uses this gradient field to find a path to the goal configuration q_{goal}. In the following paragraph, two basic approaches are presented to find a path from an arbitrary initial configuration q_{init} to a goal configuration defined by the potential field q_{goal}.

- **Depth-first planning**
  Since the inverse gradient field provides a vector for each configuration which points to the state with the lowest slope, intuitively a path can be created by following the direction of these vectors.

  Starting from any given state q_{init}, the gradient vector $\vec{t}$ of any state q can be defined as

$$\vec{t} = \frac{\vec{F}(q)}{\left\|\vec{F}(q)\right\|} = \begin{pmatrix} t_1 \\ \cdots \\ t_j \\ \cdots \\ t_m \end{pmatrix} \tag{3.4}$$

  where $\vec{t}$ represents the normalized unit vector, pointing to the lowest slope of U. Consequently, the next state q_{i+1} can be determined by

$$x_j(q_{i+1}) = x_j(q_i) + \delta t_j(q_i) \tag{3.5}$$

  where t_j, $1 \leq j \leq m$ is the j'th component of the m-dimensional vector $\vec{t}$ and $x_j(q)$ represents the j'th component of the state q. Since $\vec{t}$ has unit length (i.e., $\|\vec{t}\| = 1$), the

variable δ represents the size of a step, i.e., the distance between two states q_i and q_{i+1}. If this approach is continued, q_i successively converges to q_{goal}. However, since δ is not infinitely small, the goal configuration will not be exactly reached. Therefore, a set $X_{goal} = \{X \subseteq \mathcal{C} | x \in X \wedge \|q - q_{goal}\| < \epsilon\}$ has to be defined which specifies a subset of the CS around the goal configuration. If $q_i \in X_{goal}$, the algorithm terminates.

This method is quite feasible, if there are only convex obstacles in the CS. However, if there are non-convex obstacles in the CS, it is possible that the gradient field yields into a trap. This is the case when a local minimum is reached where the planner possibly rotates around a local minimum. However, a systematic approach to overcome this problem is quite difficult. Therefore, another strategy is presented which solves this problem.

- **Best-first planning**
  In contrast to the previous approach, this method discretizes the potential field by a grid such that the initial state and the goal state are elements of the discrete SS. Furthermore, a p-neighborhood is defined such that two states only diverge in p different dimensions where $0 \leq p \leq m$ and m denotes the number of dimensions. Hence, it is clear that a state q has $2m$ 1-neighbors and further $3^m - 1$ m-neighbors. Normally, the algorithm only considers only 1-neighbors. Due to the discretization of the CS, it is assumed that if two adjacent states are in free space, the path between them is also in free space. Furthermore, it is assumed that the CS is mapped to a finite subset. The algorithm constructs a tree starting at q_{init}. It explores all unexplored neighbors of the leaf with the least potential in the discrete potential field. In case of a non-convex obstacle, it first explores all states within the trap. If all states within the trap (i.e., around the local minimum) are explored, it continues with the next state outside the trap with the least potential. Finally, the algorithm terminates if the goal state is reached. Since the algorithm stores the parent node in each node, it is easy to track the path back to the initial state. The overall complexity of this algorithm is $O(m \cdot r^m \cdot log(r))$ where r is the number of discrete points in one dimension. Due to this upper bound, the algorithm is only practical for CS with less dimensions (e.g., up to $r = 4$).

Note that a few other motion planning approaches based on potential fields exist. Since this section only presents the basic concepts of potential field planning, the reader is referred to [2] for further information.

3.5 Probabilistic motion planning

In Section 3.3, a few roadmap based motion planning strategies were presented. These strategies provide systematic and deterministic methods to solve motion planning problems. This section introduces probabilistic approaches to solve motion planning problems. These approaches are not complete in general but they provide a pretty good average performance in practice.

3.5.1 Probabilistic roadmaps

In this section, a probabilistic roadmap based approach is presented which can also be used in practice for high dimensional CS. The method called Probabilistic Roadmap (PRM) can be divided into two different phases. In the *preprocessing phase*, the roadmap is established. This step has to be performed only once for a given WS. In the second phase, the previously computed roadmap is used to find a feasible path from an initial state to a goal state. This can be done by using graph search algorithms. In this phase (*query phase*), a path can be computed very fast.

Preprocessing phase

The preprocessing phase generates a roadmap based on the CS and the obstacles as shown in Figure 3.11. However, the initial and the goal state are not considered in this phase. The roadmap is represented by an undirected graph similar to Section 3.3. The nodes represent valid configurations in the CS and the edges between them represent feasible paths between two adjacent states. Informally, the roadmap is constructed by iteratively adding random configurations within the free space of the CS. Furthermore, it is tried to connect every random configuration with its neighbors by a local planner. More formally, Algorithm 4 describes the construction strategy of the roadmap. In this case, the roadmap is represented by an undirected graph $G = (V, E)$ where the weight of each edge E has to be defined in advance. In general, the Euclidean distance can be used in case of an Euclidean CS. Algorithms to construct a roadmap are presented in [19, 4]. The more illustrative and detailed algorithm of [4] is presented in Algorithm 4 (with a few modifications).

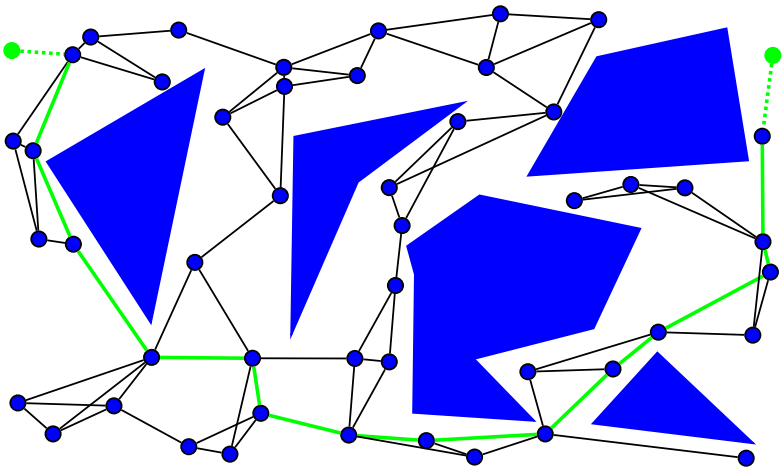


Figure 3.11: Example of a PRM with a possible path between two configurations (taken from [4])

The function $D(q, q')$ represents the distance function between two configurations which is defined as $C_{free} \times C_{free} \mapsto \mathbb{R}^+$. This function can either represent the Euclidean distance between two configurations or the result of a local planner connecting these configurations. Moreover, the function $\Delta(q, q') : V \times V \mapsto \{true, false\}$ evaluates to true if the local planner

Algorithm 4 Constructing $G = (V, E)$

```
1:  V ← ∅
2:  E ← ∅
3:  while |V| < n do
4:      q ← random configuration ∈ C_free
5:      V ← V ∪ q
6:  end while
7:  for all q ∈ V do
8:      V_q ← the k closest neighbor of q ∈ V
9:      for all q' ∈ V_q in order of increasing D(q, q') do
10:         if ¬(Π(q, q') ⊆ E) ∧ Δ(q, q') then
11:             E ← E ∪ Π(q, q')
12:         end if
13:     end for
14: end for
```

has found a feasible path between two nodes and false otherwise. Furthermore, the function $\Pi(q, q')$ represents a set of edges computed by the local planner. In some cases, local planners use existing edges for computing a path and therefore the planner does not necessarily return only one edge. Hence, the algorithm from [4] was adapted in Line 10 and 11.

The performance of this algorithm strongly depends on the local planner which is used to connect two configurations q and q'. To realize such a planner, different methods exist which can be distinguished between complete deterministic or incomplete local planners. Complete local planners always find a solution, if one exists and otherwise they return a failure. However, most of these planners are computational expensive. On the other hand, simple local planners exist that possibly do not find a solution even if one exists. However, these simple local planners can be implemented very efficient.

Since a complete deterministic planner finds a solution in each case, the number of random points can be reduced and therefore also the number of executions of this planner. If a non-deterministic but fast planner is used, it is obvious that much more random points are necessary to find a feasible path. In this case, the planner is also executed more frequently. Thus, a trade off has to be found between these extrema. Consider, for example, that a quite simple local planner connects two configurations by a straight line. This planner can be implemented very fast but in many situations this approach will fail.

The strategy presented above has a drawback, if the CS contains narrow passages. In this case, only less random points or even no one is generated within such a passage. Hence, it could be possible that two unconnected subgraphs arise even if the CS is connected. To overcome this drawback many strategies exist that increase the density of random points within narrow passages. One possible strategy uses a post-processing step after the roadmap construction.

In general, it is very difficult to determine the narrowness of a region in the CS. One possibility to approximate this probability function is to consider the number of neighbors within a given distance of a node. This is a metric for the narrowness because a node in a narrow passage

has less neighbors than a node in a free region of the CS. Another possibility is to consider the failures of the local planner as an indicator for narrowness. In this case, the failure ratio of a node q can be defined as

$$r_f = \frac{f(q)}{n(q)+1}$$

where $f(q)$ denotes the failures of the local planner and $n(q)$ denotes the number of neighbors ($|V_q|$). Further the probability of a node can be defined as

$$\omega(q) = \frac{r_f(q)}{\sum_{a \in V} r_f(a)}. \tag{3.6}$$

where $\omega(q)$ represents the relative narrowness of a configuration. Thus, the density of random configurations is increased in narrow regions according to $\omega(q)$. This extension mechanism is called *node enhancement*.

Query phase

If a roadmap was created, queries can be executed which ask for a path between the initial configuration q_{init} and the goal configuration q_{goal}. Since the roadmap uses only random configurations, it is nearly impossible that any random configuration q is equal to q_{init} or q_{goal}. Hence, a path between q_{init} and the closest state in the roadmap $q'_{init} \in V$ and further between q_{goal} and $q'_{goal} \in V$ has to be computed. If these paths were found, a graph search algorithm can be used to find a feasible path.

First, the two nodes $q'_{init}, q'_{goal} \in V$ have to be specified. One possibility is to choose the node $q_i \in V$ with the smallest Euclidean distance to q'_{init} or q'_{goal}. To compute the path between q_{goal} and $q'_{goal} \in V$ the local planner as described in the previous paragraph can be used. However, if a complete and slow local planner is used, it is possible that a query takes a long time. If a fast and incomplete local planner is used, it is possible that no path is found and therefore the next nearest node $q \in V$ is declared as q'_{init} or q'_{goal} respectively and the local planner is executed once again.

Finally, the path can be computed by the concatenation of the following segments:

$$\Pi(q_{init}, q'_{init}) \circ \text{shortest\_path}(q'_{init}, q'_{goal}) \circ \Pi(q'_{goal}, q_{goal})$$

Lazy PRM

If the CS often changes (e.g., in case of moving obstacles) or only single queries are performed, the approach described above has an enormous drawback. In some cases, the preprocessing phase has to be performed for every query phase and since the preprocessing phase is computational expensive, the approach is quite inefficient for such purposes.

The reason for the possibly high computational effort is the collision check which has to be performed for every node $v \in V$. A Lazy Probabilistic Roadmap (LPRM) which is introduced in [5] is an adaption of the PRM approach which reduces the number of collision checks. The basic idea is to compute random configurations q without considering the obstacles. Similar to the basic PRM approach, a roadmap is constructed. In this step, the obstacles are also neglected.

After computing the roadmap, a shortest path algorithm is performed (e.g., A* as described in Section 3.2) and finally the nodes of the resulting path are checked for collisions. If colliding nodes appear, these nodes are removed from the graph $V := V \backslash V_c$ where V_c is the set of colliding nodes of the path. Afterwards, the shortest path algorithm is performed again.

If no path can be found either in the original roadmap or due to the removal of a node, new nodes have to be inserted. This method is called *node enhancement*. To avoid confusion, this technique is different to the one of the previous paragraph. The idea of this concept is to add a set of nodes N_{enh} around a given state called *seeds*. These seeds should basically be in narrow passages or difficult regions of the CS. One possibility is to use a state which is the intersection between an obstacle and a previously removed edge of the shortest path. This strategy guarantees that the seed state is close to the obstacles and near to the shortest path. To guarantee probabilistic completeness, it is necessary to add uniformly distributed random points over the whole CS. If this step is performed iteratively, a cluster of nodes around the seeds arises and the other regions are neglected. To overcome this drawback only uniformly generated random points can be used as basis for the seeds. The new random points around the seeds can be distributed in many different ways. An overview about this approach is given by the flow chart in Figure 3.12.

Since collision checks are only performed for complete paths and not for every random point, it turns out that this method is more practical for single queries or for CS with moving obstacles. However, this approach can also be used for multiple queries instead of basic PRM. Another PRM based approach is presented in Section 4.4 where non-holonomic systems are considered, especially for CLRs.

3.5.2 Motion planning based on rapidly-exploring random trees

Another probabilistic motion planning approach uses a special kind of random trees exploring the CS called Rapidly-exploring Random Tree (RRT). The special property of RRTs in contrast to conventional random trees is that the RRT rapidly discovers the whole CS. This property is very useful to find a path over large distances. The difference between a conventional tree and an RRT is shown in Figure 3.13. The left tree is constructed using a conventional approach where a local cluster arises. On the right side of this figure, the tree rapidly explores the whole space. Despite from the previous mentioned approaches, RRT can be used for high dimensional CSs. As described in Section 3.5.1, PRM based approaches are only suitable for 4-5 dimensions. In contrast to PRM based approaches, RRT approaches do not need any preprocessing and therefore they are well suited for single query purposes. Another interesting property is that either a path in the WS or even the whole trajectory can be computed.

The construction algorithm of an RRT is quite easy to implement. Starting from an arbitrary configuration in the CS, the algorithm selects a random state within the free portion of the space. Subsequently, it searches for the closest state in the current tree using an arbitrary metric. Furthermore, the algorithm uses a local planner to get towards the random state. In a predefined or random distance from the nearest node, the algorithm generates a new node, if the path and the new state do not collide with obstacles.

Algorithm 5 shows the construction of an RRT with a size of k nodes. It uses the function `RANDOM_STATE()` to get a random point in the free portion of the space. It is very important

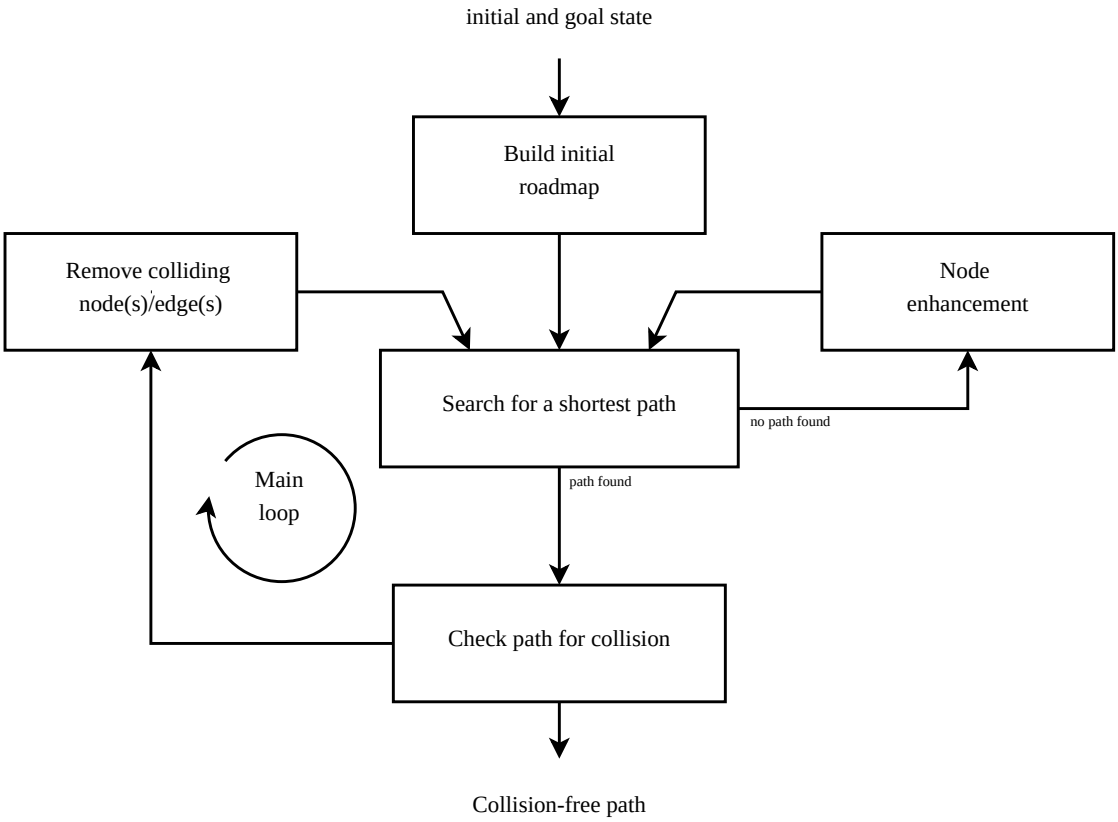


Figure 3.12: Flow chart of the LPRM approach (taken from [5])

that the random states are uniformly distributed to guarantee probabilistic completeness of the motion planning algorithm. Since it is quite difficult to generate random states from the free portion of the space in general, random numbers are chosen from the whole space and colliding nodes are rejected.

Furthermore, the function `NEAREST_NEIGHBOR(x,`$\mathcal{T}$`)` is used to find the nearest neighbor of `x` in the tree $\mathcal{T}$. This function needs a metric to estimate the distance or the costs between two states. For example, in the Euclidean space without obstacles, the Euclidean distance can be used. However, if there are obstacles within the space or the space is high-dimensional, it is quite difficult to find a good metric. Since this step is executed frequently as described in the algorithm, it is very important to find a fast metric. In a high dimensional CS including obstacles, the problem of finding an exact metric can be as hard as the motion planning problem itself [6]. Therefore, in most cases heuristics are used to estimate the costs between two states.

Finally, the function `NEW_STATE(...)` generates a new node x_{new} between the nearest node x_{near} and the actual random node x_{rand} connected to x_{near}. The new point must be reachable from the nearest point considering all constraints of the system. Therefore, a local planner can be used to move towards x_{rand}. However, it is not necessary that the planner computes the whole path to x_{rand}. It is sufficient that it computes a feasible path towards the random state

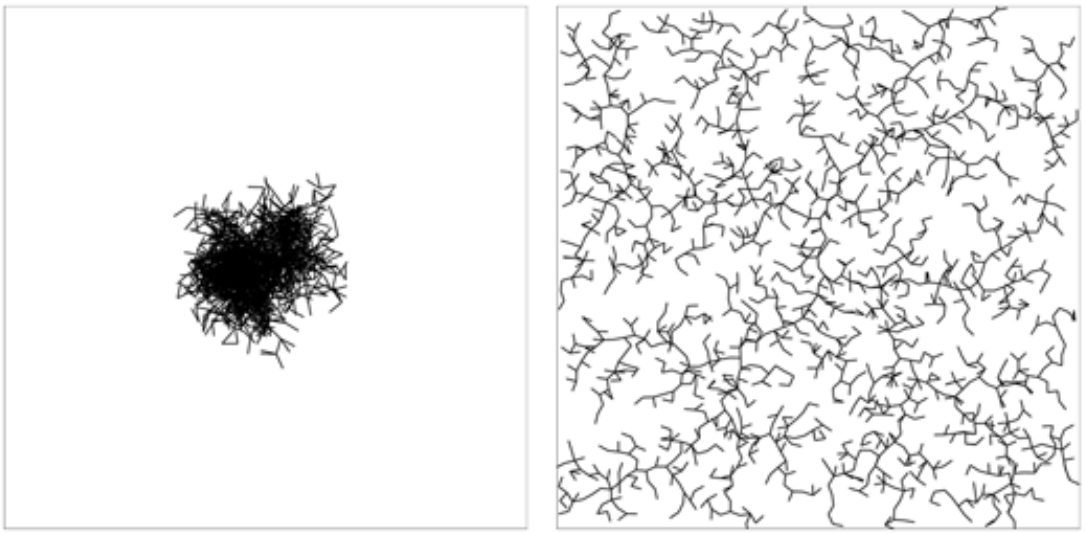


Figure 3.13: Comparison between a conventional tree and an RRT in two-dimensional space (taken from [6])

(with a fixed or random length). If the new state is within an obstacle area the state is rejected and a new random state is chosen. The steps described above are visualized in Figure 3.14.

Since collision checking is necessary for `RANDOM_STATE()` and for `NEW_STATE(...)`, it is very important that this step is executed very fast. The quality and the computation time immanently depend on the distance metric and the local planner. Obviously, it is nearly impossible that Line 5 of the function `EXTEND()` in Algorithm 5 is satisfied, especially in the case of a continuous CS. Thus, it is practical to define a function which checks if $\|x_{new} - x_{goal}\| \leq \epsilon$ for a constant ϵ. However, if ϵ is chosen too large, it has to be determined if the path between x_{new} and x_{goal} is feasible and whether a path between these nodes has to be computed. This step has to be implemented very efficient (e.g., using a heuristic), since it has to be executed for every new node.

Algorithm 6 shows a simple adaptation of the basic RRT algorithm. In additional to Algorithm 5, it is checked if a new node is connectable to the goal node. In this case, the algorithm returns the final path using the function `PATH(`$\mathcal{T}$`,`x`)` which simply iterates from the matching node $x_{new} \in X_{goal}$ back to the root x_{init}. If the tree reaches a size of k nodes without finding a path, it returns a failure.

Bidirectional RRT

In the previous part of this section, an RRT was described which starts from x_{init} and terminates if one node of the tree is close to x_{goal}. However, this approach can be improved, if two trees are computed simultaneously, where one tree starts at x_{init} and the other tree starts at x_{goal}. If a new node is added to one tree, it is checked if any node of the other tree is close enough to the new node. Algorithm 7 describes this approach. Note that the function `EXTEND(`$\mathcal{T}$`,`x`)` is

Algorithm 5 Basic algorithm for RRT generation (taken from [6])

BULID_RRT (x_{init})

1: $\mathcal{T}.init(x_{init})$
2: **for** $k = 1$ to K **do**
3: $x_{rand} \leftarrow$ RANDOM_STATE ()
4: EXTEND ($\mathcal{T}$, x_{rand})
5: **end for**
6: Return *T*

EXTEND($\mathcal{T}$, x)

1: $x_{near} \leftarrow$ NEAREST_NEIGHBOR (x,$\mathcal{T}$)
2: **if** NEW_STATE($x, x_{near}, x_{new}, u_{new}$) **then**
3: $\mathcal{T}$.add_vertex (x_{new})
4: $\mathcal{T}$.add_edge ($x_{near}, x_{new}, u_{new}$)
5: **if** $x_{new} = x$ **then**
6: Return *Reached*
7: **else**
8: Return *Advanced*
9: **end if**
10: **end if**
11: Return *Trapped*

Algorithm 6 Planning algorithm using an RRT (taken from [6])

RRT_PLANNER (x_{init}, x_{goal})

1: $\mathcal{T}.init(x_{init})$
2: **for** $k = 1$ to K **do**
3: $x_{rand} \leftarrow$ RANDOM_STATE ()
4: **if** not EXTEND ($\mathcal{T}$, x_{rand}) = *Trapped* **then**
5: **if** EXTEND ($\mathcal{T}$, x_{new}) = *Reached* **then**
6: Return PATH($\mathcal{T}$)
7: **end if**
8: **end if**
9: **end for**
10: Return *Failure*

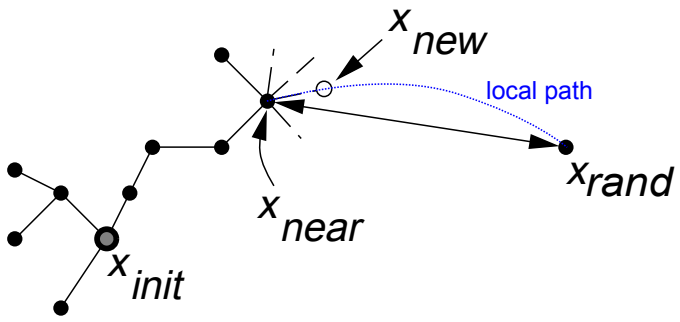


Figure 3.14: Visualization of function `EXTEND()` (taken from [6])

the same as in Algorithm 5.

The function `CONNECT` $(\mathcal{T}_b, x)$ uses a greedy approach to connect the new node with the other tree. For simplicity it is also possible to use the `EXTEND` $(\mathcal{T}_b, x)$ function instead of `CONNECT` $(\mathcal{T}_b, x)$ in Line 6 of `RRT_CONNECT_PLANNER` from Algorithm 8.

Algorithm 7 Bidirectional planning algorithm using an RRT (taken from [6])

RRT_CONNECT_PLANNER (x_{init},x_{goal})

1: $\mathcal{T}_a.init(x_{init})$
2: $\mathcal{T}_b.init(x_{goal})$
3: **for** $k = 1$ to K **do**
4: $x_{rand} \leftarrow$ RANDOM_STATE ()
5: **if** not EXTEND ($\mathcal{T}_a$, x_{rand}) = *Trapped* **then**
6: **if** CONNECT ($\mathcal{T}_b$, x_{new}) = *Reached* **then**
7: Return PATH($\mathcal{T}_a$, $\mathcal{T}_b$)
8: **end if**
9: **end if**
10: SWAP ($\mathcal{T}_a$, $\mathcal{T}_b$)
11: **end for**
12: Return *Failure*

CONNECT($\mathcal{T}$, x)

1: **repeat**
2: $S \leftarrow$ EXTEND ($\mathcal{T}$, x)
3: **until** not (S = *Advanced*)

Guided RRT

In [10], an interesting approach is presented which can increase the performance and quality of RRT based planning algorithm dramatically. Especially in presence of narrow passages in the CS, this method provides much better and faster solutions.

The basic idea is to use an auxiliary path to guide the RRT planner. This auxiliary path is based on other motion planning approaches e.g., roadmap based approaches as described in

the beginning of this chapter. The question is: why is it desirable to use two different planning approaches? Most of the described approaches are only suitable for low-dimensional spaces without differential constraints but RRT approaches are well suited for these problems.

Therefore, an auxiliary path is computed in the CS without considering any constraints. Furthermore, many of these algorithms provide a high quality path. This path is used to guide the RRT search as described in Algorithm 8. It is assumed that the auxiliary path is represented by a list of states which is stored in $P[\,]$. Moreover, the distance from the nearest node in the tree to the corresponding state in $P[i]$ is stored in $D[i]$. Note that this list has to be updated after each iteration. Furthermore, the temporal goal t is initialized to the first element of the path. The algorithm checks in each iteration, if any node of the tree has reached the temporal goal t and if so, a new temporal goal is elected based on the distance of elements of $P[\,]$ by the function `NEXT()`. In contrast to the basic RRT approach, the *random node* is set to the temporal goal t for every n_{bias} iteration in Line 9 to 13 of the algorithm. It is also necessary to select random states as in the basic RRT to guarantee probabilistic completeness. If an auxiliary path exists which is not feasible for a constrained system, the performance decreases. However, after escaping this trap, the algorithm tends to go back to the auxiliary path.

In Line 20, the algorithm updates the distances from each point of the auxiliary path to the tree. The rest of the algorithm is quite similar to the basic RRT approach.

Algorithm 8 RRT-Path (taken from [10])

RRT_PATH (x_{init}, x_{goal})

1: $\mathcal{T}.add(x_{start})$
2: $P[\,] \leftarrow$ prepare auxiliary path from x_{init} to x_{goal}
3: $D[\,] \leftarrow$ distances between points in $P[\,]$ to the nearest point in tree T
4: $t \leftarrow$ FIRST($P[\,]$)
5: **for** k to K **do**
6: **if** EXTEND ($\mathcal{T}$), t = *Reached* **then** {Evaluates to true if t is reached by a node of $\mathcal{T}$}
7: $t \leftarrow$ NEXT ($P[\,]$,$D[\,]$)
8: **end if**
9: **if** $k \mod n_{bias} \neq 0$ **then**
10: $x_{rand} \leftarrow t$
11: **else**
12: $x_{rand} \leftarrow$ RANDOM_STATE ()
13: **end if**
14: {Similar to RRT_PLANNER}
15: **if** not EXTEND ($\mathcal{T}$, x_{rand}) = *Trapped* **then**
16: **if** EXTEND ($\mathcal{T}$, x_{new}) = *Reached* **then**
17: Return PATH ($\mathcal{T}$)
18: **end if**
19: **end if**
20: UPDATE_DISTANCE($D[i]$) $\forall i$
21: **end for**
22: Return *Failure*

Analysis

Since the RRT approach is a non-deterministic algorithm, it is obvious that it cannot be complete. This means that it is not guaranteed that the algorithm finds a solution even if one exists. Moreover, the algorithm cannot determine if a solution exists or not. In practice, the algorithm provides a quite good average performance and it can be assumed that no path exists if the algorithm does not find one after a sufficient number of steps.

This interesting property leads to a term called probabilistic completeness. This means that if the size of the tree increases, the chance to find a solution increases proportionally. If the tree size converges to infinity, a solution will eventually be found if one exists. In [20], a proof is presented which shows that the RRT approach is probabilistic complete.

3.6 Feedback motion planning

In classical control theory, trajectory planning and system control are different tasks. The classical approach in control theory is to plan a feasible trajectory in the first step. In the second step a state controller is used to keep the system on the desired trajectory.

This approach has many drawbacks. A state controller is able to keep the system on the desired trajectory but the controller is only able to lead the system back to the trajectory if it is within an ϵ-environment around the trajectory. In the most cases, ϵ can be denoted as $d = \|x(t) - x_d(t)\| \leq \epsilon$. The control loop is only stable, if the system does not leave the ϵ-environment. Due to uncertain events (e.g., sensor errors of real systems), it is possible that the system leaves this environment and gets uncontrollable.

To overcome this drawback, another approach is presented. This approach merges the planning and the control task to one single closed loop task. Assume that the actual state of an arbitrary system is x_{init}. Further let x_{goal} be the desired goal position. In the first step, a motion planning algorithm is used to plan a feasible trajectory from x_{init} to x_{goal}. If the motion planning algorithm also computes the input vector of the system, this vector can directly be used as input for the system. After the system was executed a given period of time, the system reaches the state x_1. Not that x_1 can be arbitrary. From this state, the motion planning algorithm is used again to plan a feasible trajectory form x_1 to x_{goal}. These steps can be continued for states $x_2, ..., x_{n-1}$ until $x_n \in X_{goal}$ is reached. As one can see, no trajectory tracking controller is necessary. Thus, uncertain behavior can not destabilize the system controller.

Due to these properties, one will ask why this approach is not used for every task in practice. The main drawback of this approach is the performance. As mentioned before, the motion planning algorithm has to be performed periodically during runtime. Since these algorithms are inefficient or even non-deterministic, it is not possible to execute these algorithms in the desired interval. However, if the time interval between two sampling states x_i and x_{i+1} is too long, this can lead to an inefficient path or a collision.

In the case of a predefined CS, motion planning approaches can be used which are optimized for multiple queries (e.g., roadmap based approaches or potential field approaches) to reduce the effort of a query. In the following part of this section, concrete feedback planning approaches either for discrete or for continuous systems are introduced.

3.6.1 Discrete approach

In this paragraph, only motion planning problems on discrete CSs are considered. Since real systems are continuous in general, the continuous system has to be discretized. In Section 3.2, the formal definition of a discrete motion planning problem was presented. The definition for a feedback motion planning problem is quite similar with the exception that the discrete CS and also the input space are finite. An example of a finite discrete CS is given in Figure 3.15.

The idea is to define a not necessary optimal but feasible plan π from every state $x_i \in \mathcal{C}$ to $x_{goal} \in X_{goal} \subseteq \mathcal{X}$. This plan consists of a sequence of actions vectors $\pi_i = (u_i^0, u_i^1, ..., u_i^k)$. Since the CS is finite and known in advance, it is possible to compute a plan π_i for every state $x_i \in X$ in advance. Hence, it is not necessary to execute a motion planning algorithm during runtime. If the system starts in an arbitrary state x_i of the CS $\mathcal{X}$, it executes the first action u_i^0 of the plan π_i. Due to the state transition function, this results in a state $x_j = f(x_i, u_i^0)$. However, if an uncertain event arises, the system resides in state x_l instead of state x_j. Fortunately, this is no problem for the algorithm. Since all states consist of a feasible plan to the goal, the system executes u_j^0 or u_l^0 respectively. If the system reaches x_{goal} the algorithm stops and holds the system.

As one can see from the above description, only the first action u_i^0 of a plan π_i is executed. Therefore, the subsequent actions $u_i^1, ..., u_i^k$ are not necessary for the algorithm and so a lot of memory can be saved.

Hence, a feedback motion plan can be reduced to a function which maps every state x_i to one action u_i. Thus, the function π can be redefined as $\pi : X \mapsto U$ in contrast to Section 3.2.

Until now, only not necessary optimal paths were considered. However, this approach can also be used, if optimal paths are desired. In this context, Dijkstra's algorithm is reconsidered. In Section 3.2, this algorithm was used to find a shortest path between two nodes of a weighted undirected graph. However, this algorithm has a few side effects which are very useful for feedback motion planning. Assume that x_{goal} is the start node for Dijkstra's algorithm and that the goal node can never be reached (i.e., the algorithm terminates if all nodes are processed). In this case, Dijkstra's algorithm calculates for every node (state) a cost value which represents the costs to x_{goal}. Furthermore, every state (node) contains a pointer to the next state on the path to the start node (x_{goal}). This exactly represents the behavior of the function π_i for a state x_i.

Figure 3.15 shows an example of a grid based two-dimensional CS. In this case, each node represents a state in the CS. The red nodes denote the states which are within obstacles – therefore they are not connected. The target state x_{goal} is represented by the green node in the figure. Furthermore, it is assumed that the distance between two adjacent nodes (states) is one. The value within a node represents the costs-to-go functions which denotes the costs to the goal node. The arrows point to adjacent nodes with the least costs to the goal state. The arrow represents the action which the system has to execute next. Hence, this arrow can also be interpreted as the function π_i as described above. The presented example is very similar to the example in [7].

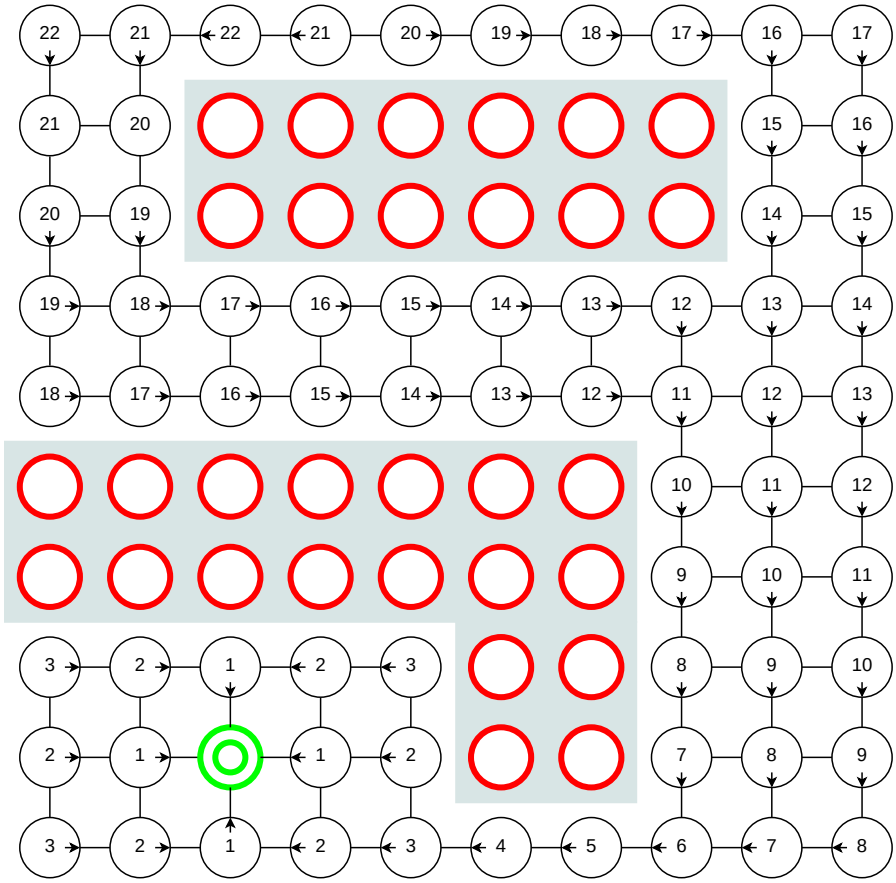


Figure 3.15: Example for a feedback plan over a 2D CS with obstacles (taken from [7])

3.6.2 Continuous approach

Due to the complexity of the theory behind this concept, only the basic ideas of continuous feedback motion planning are presented. The interested reader is referred to [7]. One concept is based on a hierarchical approach combining the potential field method as introduced in Section 3.4 and the method of cell decomposition as described in Section 3.3.

Preprocessing Phase

Consider a continuous but finite CS. This CS can possibly contain polyhedral obstacles.

- The first step is to decompose the CS into a cell complex. This means that the CS is divided into polyhedral obstacles. In case of $\mathcal{C} = \mathbb{R}^2$, these objects are triangles ([18]). These cells are denoted as $W = \{w_0, ..., w_{m-1}\}$. One distinguished cell w_{goal} exists which contains the goal configuration x_{goal}. Based on this cell decomposition, an undirected weighted graph $G = (V, E, c)$ where c is the cost function is constructed as follows. Every node in the graph represents a cell in the CS ($W \mapsto V$), the corresponding node of the cell w_{goal}

which contains the goal state x_{goal} is denoted as goal node v_{goal}. Furthermore, two nodes of the graph are connected if and only if the corresponding cells are adjacent (i.e., they share the same edge). The costs of an edge can be defined differently. One possibility is to define the weight as the Euclidean distance between the center of two adjacent cells. In case of w_{goal}, the goal configuration x_{goal} is considered instead of the center point. The resulting partition is shown in Figure 3.16.

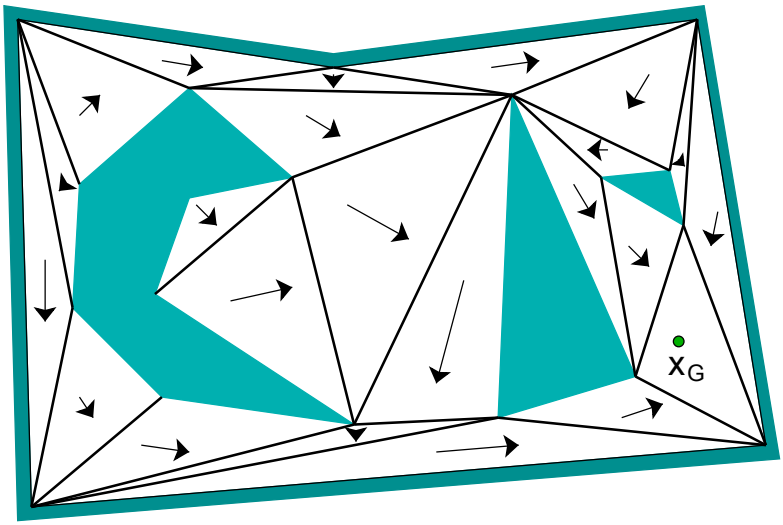


Figure 3.16: Example of a triangle cell decomposition (taken from [7])

Once such a graph is computed, the algorithm of discrete feedback motion planning is applied on the graph. This means that Dijkstra's algorithm is applied to this graph starting from v_{goal}. This results in a graph where every node (or cell) consists of a cost value. Furthermore, every node contains an action which directs the system to an adjacent cell (or node). In case of $\mathcal{C} = \mathbb{R}^2$, this can be illustrated as shown in Figure 3.16. The arrows in every cell denote the action. The action of each cell specifies one face or edge (depending on the dimension) to an adjacent cell. Informally, one can say that the system must exit the actual cell over this face. Hence, this face or edge is called exit face. This step describes the global feedback plan of the system. It must be guaranteed that a system which resides in a cell exits this cell over the exit face.

- To define a local plan for each cell which guides the system over the exit face, a gradient field is defined for every cell. This method is quite similar to the potential field approach as described in Section 3.4. The gradient field should be defined such that every integral curve (i.e., trajectory) ends at the exit face. Figure 3.17 shows a cell where the vectors point to the exit face (or edge). Since the goal cell w_{goal} does not have an exit face, the potential field must be defined such that every trajectory ends in the goal configuration x_{goal}.

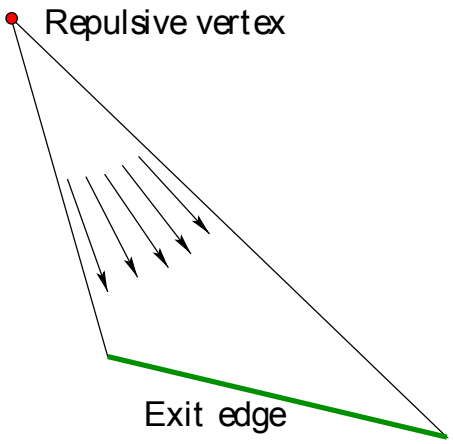


Figure 3.17: Gradient field within a triangle cell (taken from [7])

Execution Phase

Once the gradient fields were computed, the preprocessing is done and the algorithm can be executed as follows. Starting from an arbitrary configuration $x_i \in w_a$, the corresponding action trajectory derived from the gradient field is applied to the system for a given time interval. After this execution step the system resides in the configuration x_j. It is not necessary that x_j exactly follows the aimed trajectory because of the basic properties of feedback motion planning. Subsequently, the action trajectory w.r.t. the gradient field is applied to the system starting from x_j.

The time interval in which the system executes without any feedback can lead to a problematic behavior. Ideally, the time interval should be near to zero. In this case, it is nearly impossible that the system leaves X_{free}. However, this is not possible in practice because the feedback planner needs a few time to calculate the next input curve for the system and therefore a compromise has to be found.

A few motion planning concepts for continuous systems exist but the above described method is only one illustrative approach. Note that the above approach is complete but not optimal. In [7], algorithms are presented which are optimal or possibly probabilistic.

3.7 Comparison of motion planning approaches

Table 3.1 summarizes the presented approaches based on the properties: completeness, optimality, and performance. Additionally, the table provides an overview about the benefits and drawbacks of each approach.

Motion planning approach		Benefits	Drawbacks	Suitable for CLR	Properties		Performance
Category	Idea				Completeness	Optimality	
Discrete	Grid based approach	Easy to compute	Enormous amount of graph nodes	Yes	Yes, w.r.t. the discrete space	Yes, w.r.t. the discrete space	Poor
Roadmap based	Visibility graph	Feasible for multiple queries, optimal path in the Euclidean space	Computes a path near the obstacles, not suitable for non-holonomic systems in general	No	Yes	Yes, in the Euclidean space	In a space with n vertices at most n(n-1) edges have to be computed
	Exact cell decomposition	Feasible for multiple queries, the path is far away from the obstacles	Not suitable for non-holonomic systems in general	No	Yes	No	Limited to the partition of the space (e.g., triangulation, sweep line)
	Approximate cell decomposition	Feasible for multiple queries, good performance due to the hierarchical approach, the path is far away from the obstacles			Yes	No	Good
	Retraction based	Feasible for multiple queries, the path has a maximum distance to the obstacle	Generalized coronoid diagram has to be computed	No	Yes	No	Depends on the efficiency of the coronoid diagram computation
Continuous	Potential field	Fast query phase	Local minima in case of non-convex obstacles	No	Yes	No	Poor
Probabilistic	Probabilistic roadmap	Suitable for spaces up to 4 dimensions, suitable for multiple queries	Enormous amount of graph nodes in large areas	Yes, with modifications	No	No	Good
	Rapidly-exploring random tree	Suitable for higher-dimension spaces, no preprocessing, no graph search	Unpredictable runtime and path-quality	Yes	Probabilistic complete	No	Good

Table 3.1: Comparison of the presented motion planning approaches

CHAPTER 4

Motion planning for car-like robots

4.1 Overview

In the previous chapter, a few basic motion planning approaches were presented. Until now, no system specific behavior was considered discussing these approaches. This chapter focuses on special systems called CLR starting with the system model. Afterwards, a few motion planning algorithms are introduced which are adapted from Chapter 3 to satisfy the requirements of CLRs. The end of this chapter deals with an application specific problem for CLRs. Based on the basic motion planning algorithms, strategies were developed to cover special application scenarios for CLRs (e.g., for mowing and harvesting).

4.2 System model of a car-like robot

This section introduces simplified system models for CLRs. While the system model for a CLR is a continuous system model, discrete models for CLRs which are more or less discretizations of the appropriate continuous models are examined, too. The basic models only consider the kinematic properties of a CLR. However, more complex models which consider dynamic properties (e.g., side acceleration, slip of the tires) exist, too. These models are very important, if the speed of the CLR is high and the dynamic properties strongly influence the behavior of the CLR. If the speed of the robot is low enough, these properties and forces can be neglected and the system model can be reduced to the kinematic *single-track model*. Figure 4.1 shows the geometric model of a CLR. The model reduces the car to a single track which behaves as a bicycle. Thus, this model is also called *bicycle model*.

- The coordinates (x, y) represent the two-dimensional position of the CLR w.r.t. the coordinate system.
- The angle θ denotes the course of the CLR. It is also referred to as heading.
- The angle ϕ represents the steering angle of the CLR w.r.t. its main axis.

- L denotes the length of the CLR.

The CLR follows a circle with radius ρ, if the steering angle ϕ is constant. Furthermore, it is obvious that the steering angle is limited to predefined boundaries $\phi \in [-\phi_{max}, \phi_{max}]$. The parametric system model for CLR is defined in Equation 4.1.

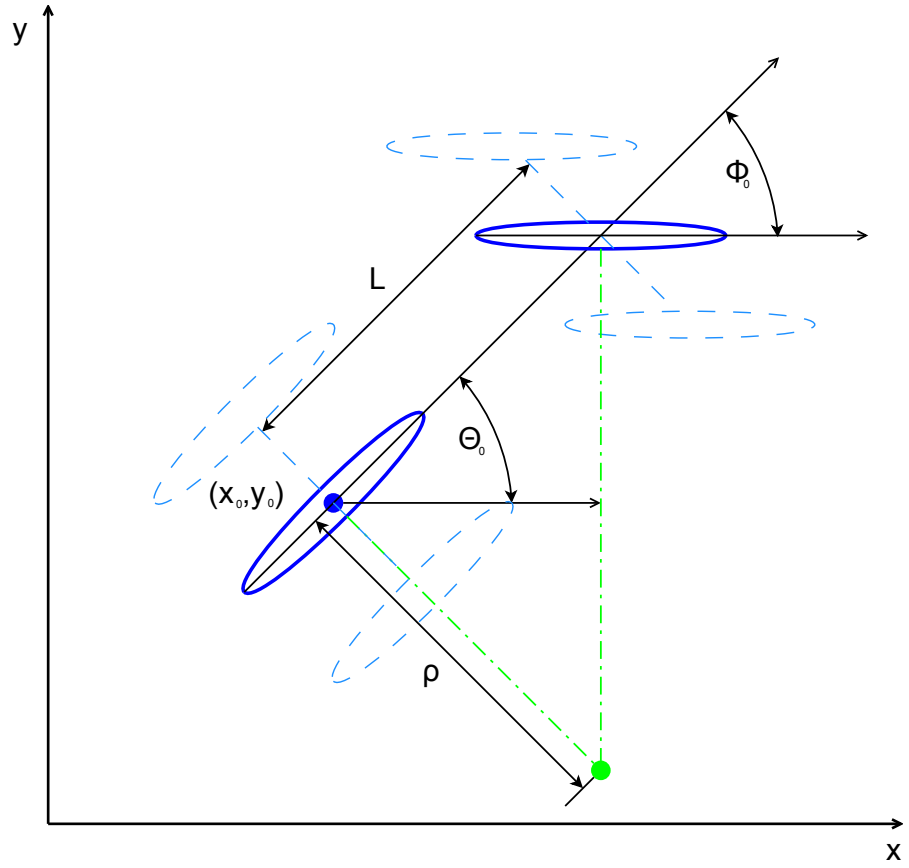


Figure 4.1: Visualization of the singe-track model for a CLR

$$\dot{q} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{pmatrix} = \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \\ \frac{\tan(\phi)}{L} \\ 0 \end{pmatrix} \cdot v_1 + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \cdot v_2 = \begin{pmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ \frac{\tan(\phi)}{L} & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \tag{4.1}$$

The input vector $[v_1, v_2]^T$ in Equation 4.1 contains the speed of the CLR along the main axis of the CLR (v_1) and the angular velocity v_2. The CS[1] consists of the cartesian coordinates x, y and the heading θ of the main axis w.r.t. the coordinate system. The last dimension of the CS represents the steering angle ϕ of the CLR w.r.t. the main axis of it. Obviously, the steering

[1] Since the system model of a CLR has only kinematic constraints, CS = SS.

angle ϕ is limited by $|\phi| \leq \phi_{max} < \pi$. Based on this description, the CS can be defined as follows.

$$C = \mathbb{R} \times \mathbb{R} \times [0, 2\pi[\times [-\phi_{max}, \phi_{max}]$$

This model considers the change rate of the steering angle ($\dot{\phi}$). If the speed of the CLR is low compared to the steering velocity, it can be assumed that the steering angle changes fast enough. Thus, the constraint on the steering velocity can be neglected and the system model reduces to a three dimensional CS. Equation 4.4 specifies a simplified model for a CLR. In this model, the input vector contains the speed of the CLR along the main axis and the steering angle of the CLR in contrast to Equation 4.1.

$$\dot{q} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \nu \\ \omega \end{pmatrix} \tag{4.2}$$

Note that the input variable of Equation 4.2 is defined as follows.

$$\omega = \frac{v}{L} \cdot \tan(\phi) \tag{4.3}$$

.

For simplicity, this model can also be expressed as shown in Equation 4.4.

$$\begin{aligned} \dot{x} &= \nu \cdot \cos(\theta) \\ \dot{y} &= \nu \cdot \sin(\theta) \\ \dot{\theta} &= \tfrac{v}{L} \cdot \tan(\phi) \end{aligned} \tag{4.4}$$

4.2.1 Trajectory of a car-like robot

In the case of a CLR, a two-dimensional geometric path is sufficient to describe the movement of the CLR. It is also interesting that the third dimension of the CS (the heading) is determined by the geometric path. The heading of the CLR is exactly the tangent angle of the geometric path. The only information which is not specified by the geometric path is the time information i.e., the speed of the CLR. Thus, in general, it is sufficient to plan the geometric path for the CLR and to choose a constant speed.

4.2.2 Optimal paths for car-like robots

Since two arbitrary configurations of the CS of a CLR can not be connected by a straight line because of the non-holonomic constraints, the question for a shortest path between two configurations comes up. The non-holonomic constraint restricts the path to a curvature constraint path. In general, a few models exist which are able to connect two arbitrary configurations by a shortest path but only in an infinite CS without any obstacles. In the following paragraphs, two similar models are presented which describe a path between two configurations as a concatenation of line and arc segments.

Furthermore, complete algorithms exist to determine the optimal path using these models. Since no real CS is unbounded and without obstacles, these algorithms are not for practical use.

Dubins' car

This model provides a method to connect two arbitrary configurations of a curvature constraint system by a shortest path. [21] proved that such an optimal path can be found by a concatenation of at most three different line or arc segments. The arc segments have constant curvature. However, it is assumed that the CS is unbounded and contains no obstacles.

An important property of this model is that it is based on a unicycle model as described previously. Therefore, a *Dubins' curve* is geometric smooth but not curvature smooth i.e., the car has to stop after each motion.

A Dubins' curve can be denoted by three different primitives. The first possible primitive is a straight line which is denoted by S. Furthermore, arcs with a constant curvature (e.g., the highest possible curvature of the robot) exist. Such an arc can either turn right R or turn left L. Based on these three types of primitives, a Dubins' curve can have one of the following structure.

$$\{LRL, RLR, LSL, LSR, RSL, RSR\}$$

Since the radius of the arc segments is defined, a Dubins' curve can be exactly defined by specifying the length of the straight segment S or the angle of the arcs in case of either R or L. Furthermore, C can be defined as $C = \{R, L\}$. Thus, a Dubins' curve can be generalized as follows:

$$\{(C_\alpha, C_\beta, C_\gamma), (C_\alpha, S_d, C_\gamma)\} \tag{4.5}$$

An additional restriction is necessary to guarantee that the specified path is optimal. If and only if $\alpha, \gamma \in [0, 2\pi[$, $\beta \in]\pi, 2\pi[$ and $d \geq 0$ then the definition of Equation 4.5 describes an optimal and unique Dubins' curve [21]. An example of such a curve is shown in Figure 4.2.

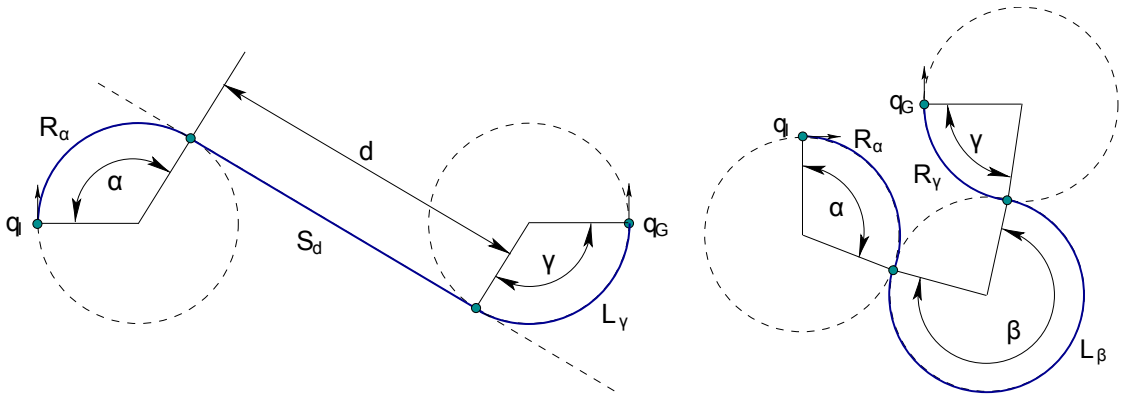


Figure 4.2: Example of two Dubins' curves (taken from [7])

Once the basic structure is defined, the question is how to find such a path. A naive approach is to try all seven possible combinations and select the shortest one. A more feasible approach is to partition the environment of a configuration to seven distinct areas according to the different motion combinations (cf. Figure 4.3). Once a goal configuration is in an area, the basic combi-

nation is determined. Furthermore, it is necessary to determine the parameters $(\alpha, \beta, \gamma, d)$ of a path. This can be done by linear programming approaches.

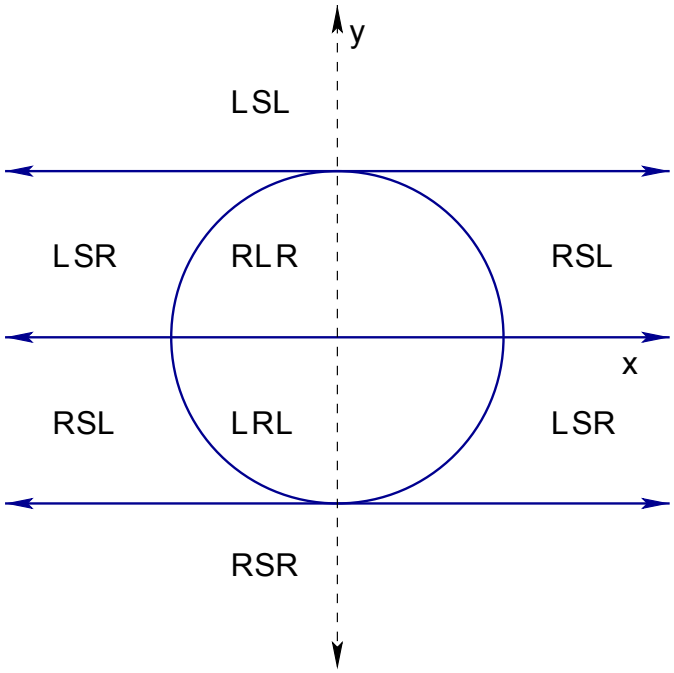


Figure 4.3: Partition of the space according to the seven basic motions (taken from [7])

Based on such a path, a simple control set for a CLR can be computed. Since only three different types of motions are available, obviously only three different controls $\phi = \{0, \phi_{max}, -\phi_{max}\}$ exist. Since the steering angle can not be changed in zero time, a CLR has to stop after each segment to change its steering angle. Consider for example the segments L_α, S_d, R_γ. In this case, the control set for the steering angle can be defined as shown in Equation 4.6.

$$u(t) = \begin{cases} -\phi_{max} & \text{if } t_0 < t \leq t_1 \\ 0 & \text{if } t_1 < t \leq t_2 \\ -\phi_{max} & \text{if } t_2 < t \leq t_3 \end{cases} \tag{4.6}$$

Reeds and Shepp Car

The Reeds and Shepp Car (RSC) ([22]) is an extension of Dubins' car. In contrast, the RSC allows also backward motion. Similar to Dubins' car a lot of basic combinations exist. Since forward and backward motions are allowed, the number of basic combinations increases to 48 in contrast to 7 in case of Dubins' car. For a complete description of the basic combinations, the reader is referred to [22]. An example of such a path is shown in Figure 4.4. This example shows a $R_\alpha|L_\beta|R_\gamma$ path. Note that the constraints of the paragraph mentioned before are not applicable in this case.

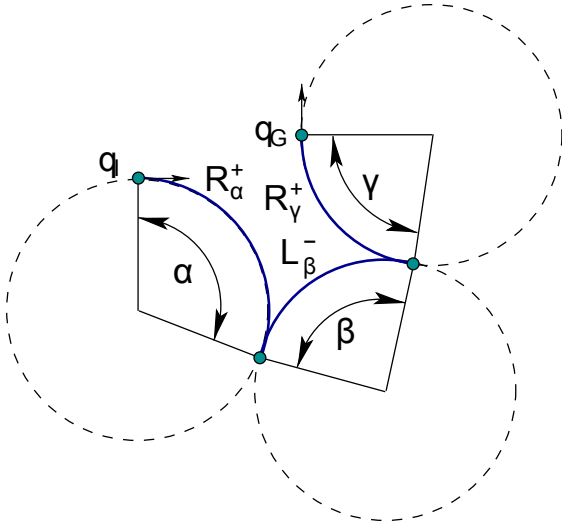


Figure 4.4: Example of a Reeds and Shepps curve (taken from [7])

4.3 Discrete motion planning for car-like robots

The approach of discrete motion planning was introduced in Section 3.2. Since a CLR is a continuous system, the system has to be discretized. A naive method of discretion is based on regular rectangular grid. All discrete states are arranged in a grid as shown in Figure 4.5(a). Furthermore, it is possible to arrange the states in a diamond lattice as shown in Figure 4.5(b). Another method is to choose uniform random configurations of states as shown in Figure 4.5(c).

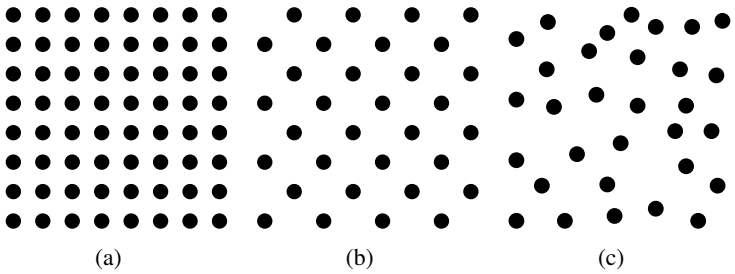


Figure 4.5: Different types of state lattice

Since the CS for a CLR is a three- or four-dimensional space, all dimensions (including the angles) must be discretized.

Naive Approach

Once a discrete CS is constructed, the states have to be connected, if they are feasible. Starting from a central point of the CS, all immediate neighbors are checked for feasibility. Note that the term *point* denotes all states with equal positions but different headings or steering angles in this

context. If a node of the center point can feasibly be connected to an adjacent node with arbitrary heading, the two states are connected by an edge. Two nodes can be connected, if the appropriate configurations can be connected by a concatenation of a straight line and an arc with a maximum curvature as shown in Figure 4.7. Note that a Dubins' curve is not suitable in this case, since a Dubins' curve computes a path with a constant curvature for every two configurations. Thus, this method can not be used for a feasibility check. If all possible neighbor configurations are connected, they are marked as `done`. Furthermore, they are connected to all their neighbors and so on.

This approach explores all reachable neighbors w.r.t. the initial configuration. Obviously, this approach requires a function which checks the feasibility between two states. Furthermore, it has to perform a collision check.

After the creation of the search graph as described before, the edges can be weighted either by the Euclidean distance between the corresponding nodes or by the path length generated by a local planner. Furthermore, a graph search algorithm can be used to find a feasible or even optimal path (w.r.t. the discretized CS) between two arbitrary states in the CS.

The drawback of this approach is the large number of states. If a fine resolution of states is required to guarantee a pretty optimal and complete solution, the number of states increases dramatically. If a coarse resolution is used, it is possible that a path can not be found, even if one exists.

Decomposition Approach

Another approach presented in [23] tries to find an optimal path in a state lattice as shown in Figure 4.5(a), by decomposition of paths starting from an initial point. Similarly to the previous approach, the immediate neighbor (i.e., 1-neighborhood) is checked if a feasible connection exists. In this case, the two nodes are connected.

In the second step, nodes in 2-neighborhood are considered. If the central point can be connected to a nodes in distance two, the computed path is checked if it is sufficiently close to any adjacent node of the central point. In this case, the node is obviously connected to the central point in the previous step. Thus, the new path is decomposed to two different segments. After this step, the 3-neighborhood is considered and so on.

Every path is decomposed to shorter segments as long as it is possible. After exploring and decomposing the k-neighborhood, the algorithm explores the $(k+1)$-neighborhood in a circle around the central point.

Note that this approach assumes that a local planner exists which connects two different configurations. Furthermore, a function is necessary which checks, if a path passes any state sufficiently close. In contrast to the naive approach, less edges are necessary. However, the possibly high amount of states remain.

4.4 Motion planning for car-like robots based on probabilistic roadmaps

In Section 3.5.1, a motion planning approach called PRM was introduced. However, the described approach is well suited for holonomic systems. Hence, non-holonomic systems were

not considered in those section.

In this section, motion planning concepts were introduced for non-holonomic systems particulalary for CLR which are based on PRM. As described in the previous sections, the challenge of motion planning for non-holonomic systems is to meet the non-holonomic constraints.

In [24], an approach is presented how to adapt the basic PRM-concept for non-holonomic system. This algorithm consists of three different phases as described below.

- **Control roadmap**
  The control roadmap is constructed similarly to the basic PRM without considering the kinematic constraints of the system. Figure 4.6 shows an example of a control roadmap (i.e., the black graph). The control roadmap is constructed in a two dimensional space $\mathcal{W} = \mathbb{R}^2$ without considering the course of the robot. It is assumed that the robot can be modeled as a disc and therefore the heading can be neglected. Thus, random nodes in the free space of the WS are computed. Furthermore, each node is connected by a local planner to its neighborhood, if there is no collision between them. This step is similar to the basic PRM, too.

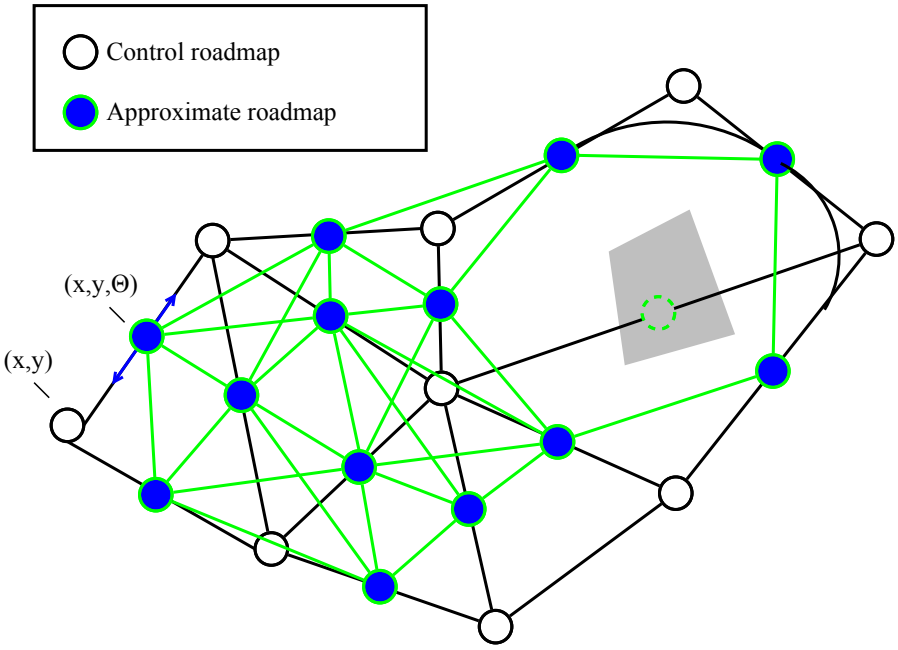


Figure 4.6: Computation of the approximate roadmap based on the control roadmap

The resulting control roadmap is represented by a graph $G = (V, E)$ which stores the connectivity of the free space but it does not consider any constraints.

- **Approximate roadmap**
  Based on the control roadmap, the approximate roadmap represented by a graph $G' = (V', E')$ is constructed as follows. Figure 4.6 shows an example of an approximate roadmap (i.e., the green graph). In contrast to the control roadmap, the approximate

roadmap considers the heading of the CLR. Thus, each node represents a state in $\mathcal{C} = \mathbb{R}^2 \times [0, 2\pi[$. For every edge $e \in E$ of the control roadmap, a node $v' \in V'$ exists in the approximate roadmap. This mapping can be described by a bijective function $l : E \mapsto V'$ which is defined as follows. The plane coordinates of a node v' are based on the bisection of the corresponding edge in the control roadmap. The third dimension of a node in the approximate roadmap represents the course of the robot which is equal to the geometric angle of the edge of the control roadmap.

Two nodes $v'_1 = l(e_1), v'_2 = l(e_2) \in V'$ are connected by an edge if and only if the corresponding edges $e_1, e_2 \in E$ are adjacent. Therefore, every node in the control roadmap G is mapped to an edge in the approximate roadmap which is denoted by the following function $l' : V \mapsto E'$. In graph theory, this mapping $G' = L(G)$ is called *line graph* of G.

Since the kinematic constraints were not considered until now, it is clear that not every connection $e' \in E' = (v'_1, v'_2)$ between two configurations v'_1 and v'_2 is feasible for a CLR. Therefore, a local planner is used to check the feasibility of an edge considering the curvature constraints. If an edge is infeasible for a CLR, it is removed from the roadmap G'.

The local planner can use a simple geometric model to connect two configurations of G'. A connection consists of a straight line segment in combination with an arc as shown in Figure 4.7. This figure shows the projection of two points of the roadmap on the plane where the third dimension of the configuration is represented by the angle of each state (similar to a vector field). Every node can be written as $v' = (r, \theta)$ where $r = (x, y)^T$ represents the point in the WS and θ the orientation of the robot. For every two nodes such a connection can be defined uniquely. To check the feasibility of such a connection, only the curvature of the arc has to be considered. The relation between the radius of the arc and the curvature can be defined as

$$\kappa = \frac{cot(\frac{\alpha}{2})}{\min(a, b)} \tag{4.7}$$

where $a = \|r_1 - r_x\|$ and $b = \|r_2 - r_x\|$. α denotes the angle difference between the two configurations. Note that v'_x is not a node of the graph.

Based on the maximum curvature of the CLR $\kappa_{\max}$, edges which do not fulfill this requirement can be removed. After this step the resulting graph $G'' = (V', E' \backslash \{e \in E' | \kappa_e > \kappa_{\max}\})$ represents a feasible roadmap for a CLR.

Based on the reduced approximate roadmap, a graph search algorithm (e.g., A*) can be performed to find a path from q_{init} to q_{goal}. If another CLR with another maximum curvature is used, only the last step has to be repeated.

However, this path is not yet feasible for practical use since curvature of the computed path is not continuous. If a CLR follows this path it has to stop after each arc or line segment and to change its steering angle.

To generate a curvature continuous path based on a sequence of arcs and lines, path smoothing approaches like cubic B-Splines can be used.

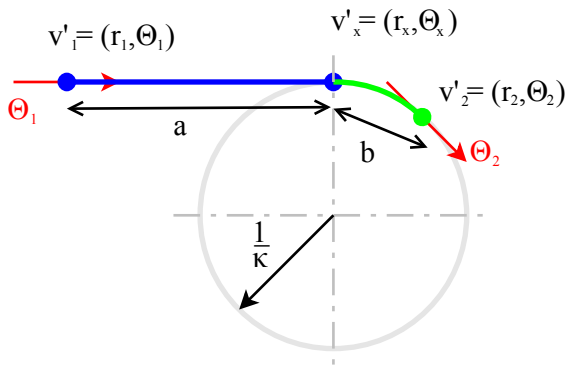


Figure 4.7: Geometric connection between two nodes of the approximate roadmap

4.5 Motion planning for car-like robots based on rapidly-exploring random trees

RRT based motion planning approaches can be used for different purposes. Famous participants of the DARPA Urban Challenge[2] use this approach to plan a route for their CLRs [25]. Sometimes the RRT approach is used in combination with feedback motion planning as described in Section 3.6.

This section examines the RRT approach in combination with the forward simulation of the system model of a CLR. Starting from the configuration q_{init} and q_{goal}, two trees are constructed, where every path between the root and any other node of the tree is feasible.

Remember the RRT algorithm of Section 3.5.2 (cf. Algorithm 9). The RRT approach for CLRs is based on the bidirectional RRT approach. Since this algorithm is a high level description, it is necessary to specify the functions of the algorithm as follows.

- `RANDOM_STATE()`
  This function is used to generate a random state out of the CS. The CS for a CLR depends on the used system model. The simple system model only considers the position and the heading of the car and therefore it can be defined as $\mathcal{C} = \mathbb{R}^2 \times [0, 2\pi[$. If in addition to the simple system the steering angle is considered as state variable, the CS looks like $\mathcal{C} = \mathbb{R}^2 \times [0, 2\pi[\times [-\phi_{max}, \phi_{max}]$.

  It is important that the random function generates a uniform random number in every dimension to guarantee probabilistic completeness. Furthermore, this function must also perform a collision check, to avoid that the random state is inside obstacles.

- `EXTEND` $(\mathcal{T}, x_{rand})$
  This function extends the tree by a node based on a given random configuration. At the beginning, the function searches for the nearest neighbor of the given random state. Therefore, the distance of every node of the tree to the random state has to be computed

[2] More information available at http://www.darpa.mil/grandchallenge/

Algorithm 9 Bidirectional planning algorithm using an RRT (taken from [6])

RRT_CONNECT_PLANNER (x_{init},x_{goal})

1: $\mathcal{T}_a.init(x_{init})$
2: $\mathcal{T}_b.init(x_{goal})$
3: **for** $k = 1$ to K **do**
4: $x_{rand} \leftarrow$ RANDOM_STATE ()
5: **if** not EXTEND ($\mathcal{T}_a$, x_{rand}) = *Trapped* **then**
6: **if** CONNECT ($\mathcal{T}_b$, x_{new}) = *Reached* **then**
7: Return PATH($\mathcal{T}_a$, $\mathcal{T}_b$)
8: **end if**
9: **end if**
10: SWAP ($\mathcal{T}_a$, $\mathcal{T}_b$)
11: **end for**
12: Return *Failure*

CONNECT($\mathcal{T}$, x)

1: **repeat**
2: $S \leftarrow$ EXTEND ($\mathcal{T}$, x)
3: **until** not (S = *Advanced*)

EXTEND($\mathcal{T}$, x)

1: $x_{near} \leftarrow$ NEAREST_NEIGHBOR (x,$\mathcal{T}$)
2: **if** NEW_STATE($x, x_{near}, x_{new}, u_{new}$) **then**
3: $\mathcal{T}$.add_vertex (x_{new})
4: $\mathcal{T}$.add_edge (x_{near}, x_{new}, u_{new})
5: **if** $x_{new} = x$ **then**
6: Return *Reached*
7: **else**
8: Return *Advanced*
9: **end if**
10: **end if**
11: Return *Trapped*

and the shortest one is chosen. However, the exact computation of the real distance (or costs) between two configurations can be as hard as the motion planning problem itself. Since it is not necessary to know the exact distance, it is sufficient to estimate the distance using a heuristic or simply the Euclidean distance.

One fast method is to consider only the Euclidean distance of the projection in $\mathbb{R}^2$ but this is not optimal. Consider the case of two configurations with nearly the same coordinates in $\mathbb{R}^2$ but two different headings. The Euclidean distance between the two configurations is nearly zero but the real distance is much higher since the CLR has to turn.

To overcome this drawback, a local planner can be used. However, a local planner is more complex than the calculation of the Euclidean distance.

If the nearest neighbor is determined, a path has to be generated towards the random configuration. Note that it is not necessary that the new configuration reaches the random configuration. It is only necessary that the path from the nearest node to the new node is feasible. It is possible to use the result of a local planner and to compute the new node in a fixed or random distance from the nearest node. Additionally, a collision check along the whole path has to be performed to avoid collisions.

Another possibility is to simulate the system model in combination with a system controller. In this case, the system model as mentioned before can be used and possibly discretized. Furthermore, a controller is used to move the system towards the random configuration. In this case, the system can also stop in a fixed or random distance from the nearest node. Obviously, a collision check for the whole path is necessary. The advantage of this approach is, that an action trajectory is generated implicitly. Furthermore, the path must be feasible, if the system model behaves as the real system. Details about this approach are presented in Chapter 5.

If this function reached the random state directly, it returns *reached*. If it computed a new node *advanced* is returned.

- `CONNECT` ($\mathcal{T}, x_{new}$)

  This function is used to check if the new node computed by `EXTEND` can be feasibly connected to some node of the rest of the tree. The function has to decide whether two configurations are feasible considering their heading and the obstacles of CS. Furthermore, the system model of the CLR has to be considered (i.e., the curvature constraint). For this purpose, it is possible that a local planner is used to connect two configurations. Next, a collision check has to be performed.

- `PATH` ($\mathcal{T}, \mathcal{T}$)
  Based on two distinct configurations of the two trees, the `PATH` function computes a path from q_{init} to q_{goal}. If every node of a tree stores the path to its predecessor, it is easy to compute a path from the connecting nodes to the root. Furthermore, both paths must be combined with the result of the local planner to return the complete path.

4.6 Hybrid approach based on rapidly-exploring random trees

Since the above approach may take a large number of nodes in presence of narrow passages or large CSs, a new approach developed in the context of this thesis can be used to overcome this drawback. Remember the guided RRT approach from Section 3.5.2. This approach uses an auxiliary coarse grained path to guide the random search.

This method also uses an auxiliary path which does not suffice the constraints of the system. In contrast to the guided RRT, this approach uses a special distribution of the random points. In addition to the uniform distribution of the random points which guarantees probabilistic completeness, a normally distributed channel around the auxiliary path is generated. Within this channel the density of random configurations is much higher than in other regions. However, the normally distributed function guarantees that the density in the center of the channel is higher than in the outer regions of the channel.

Figure 4.8 shows a sketch of a WS. In this figure, the auxiliary path is marked red. Furthermore, the density around this path is high. The orange areas in the figure denote a medium density and the yellow areas denote a low density. Finally, the white region denotes the uniformly distributed area where the density is lower than in all other regions. The blue curves along the path sketch the normal distribution of the random points along the path. Note that the density decreases continuously.

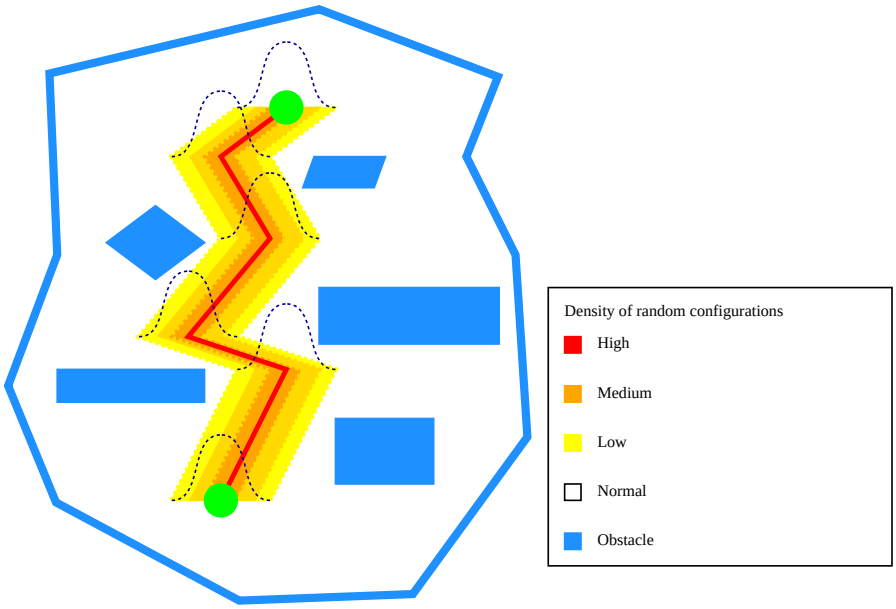


Figure 4.8: Visualization of the density of random configurations in a CS

In the most cases, this approach computes a path very fast and of high quality in contrast to the basic RRT approach. However, there are some CS in which this approach takes quite longer than the conventional approach. Consider for example a CS with a long narrow and infeasible passage. This approach will try to find a path along this passage, even if no one exists. Since the

density of random configurations in other regions is quite lower, this approach will take a longer time to find a feasible path.

4.7 Application-specific problems for car-like robots

In the previous sections, the basic motion planning problem was discussed and different solutions were presented. In this section, another problem is addressed based on the solutions of the previous sections. The problem is to find a path which completely covers a predefined area within the WS as claimed in Section 1.3. To simplify this problem, it can be assumed that a motion planning algorithm exists, which is able to find a path for the CLR from any initial configuration to an arbitrary goal configuration. There are many different ways of covering a field depending on its contour. However in practice, a field is covered in parallel lanes where turn maneuvers are performed at the end of each lane.

4.7.1 Covering an area by solving the traveling salesman problem

One possibility to cover a field is to discretize the CS and create a feasible search graph as described in Section 4.3. Furthermore, all states outside the area are deleted from the graph. Thus, the graph results in a discrete representation of the field in the WS. If all states of the graph are visited, it can be assumed that the whole area was completely covered. The formulation of this problem is similar to the Traveling Salesman Problem (TSP) problem. In contrast to the TSP problem, not every node has to be visited. In the graph a lot of nodes with the same planar coordinates but different heading exist. Thus, it is sufficient to visit at least one node of such a group. If such a node is visited, all other nodes of the group are set visited. Solving this problem does not guarantee that the whole area is covered. Since the TSP cannot be solved in polynomial time, this concrete problem can not be solved efficiently, too. However, heuristic algorithms exist which are able to solve the TSP problem very efficiently but they are not complete. Further information about this approach can be found in [26, 27].

If the motion planning problem in this section is solved with such an algorithm, it is possible that the CLR has to perform permanent turning maneuvers. This fact can lead to possible instabilities and cause a lot of load for the mechanical system.

4.7.2 Covering an area in parallel lanes

A more feasible solution is to cover the working area in parallel lanes is shown in Figure 1.3 of Section 1.3. Since the CLR has a limited turn radius it is not practical to pass one lane after the other. Assume for example that the CLR has a turn radius of 5 meters and the distance between two lanes is 2 meters. If the CLR should pass two subsequent lanes, it has to perform complicated maneuvers. In many cases, it is necessary to change the direction of the CLR a few times. These maneuvers consume much time and cause a lot of load for the mechanical parts of the system, too. If the working area is covered in parallel lanes, it is possible that the path is longer than a TSP approach. However, it is possible that a real CLR takes less time to cover the area by using parallel lanes. One reason can be that a CLR is able to drive faster, if it drives straight forward.

To calculate an efficient path for the CLR it is necessary to consider the turn radius and the distance between two lanes. The distance between two subsequent lanes on the path of the CLR should be at least twice the turn radius of the CLR.

To simplify the problem, it is assumed that the working area has the form of a rectangle where exactly n lanes with equal length are covering the working area. The lanes within the field are numbered from 0 to $(n-1)$. One possible representation of a working area is shown in Figure 4.9. Depending on the physical constraints of the CLR and the distance of the lane, the CLR can not pass the first k neighbors on its left and right side without moving backwards. Furthermore, assume that the costs of each lane are equal to the length of the lane. The costs of a maneuver from one lane to another equals also the distance between these two endpoints multiplied with a constant factor which represents the effort of the maneuver. A maneuver from one lane to another within the k-neighborhood has constant but high costs.

In many agricultural tasks, it is necessary that two subsequent lanes are passed in alternating directions. Consider for example that a soccer field should be mowed. In this case, obviously the lanes are mowed in alternating directions. A scheme of such a simplified field is shown in Figure 4.10.

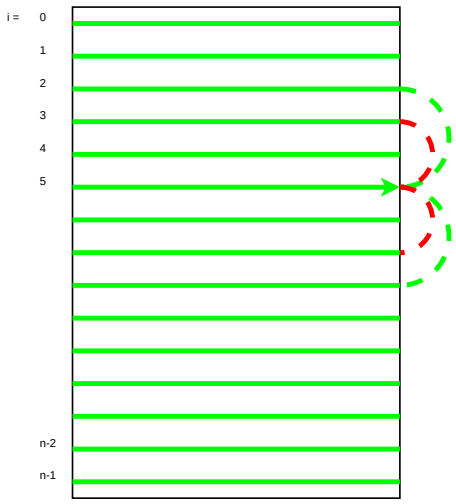


Figure 4.9: Simplification of a working area as rectangle. The maneuvers with low costs are marked green, where the maneuvers with high costs are marked red.

Finding a strategy using a decision tree

Based on the definition as mentioned before, a strategy is desired which computes a sequence of lanes. One possibility to find an optimal path in the described environment, is to build a decision tree. Assume that the field contains n lanes, then the root node of the decision tree has at most $n-1$ children. Hence, each node in distance l from the root has at most $n-l$ children and the tree has a depth of n. Since subtrees with costs greater than the best found solution are neglected during graph search, the number of iterations can be reduced. However, the number of nodes in the decision tree remains high. If an optimal solution is found, it can

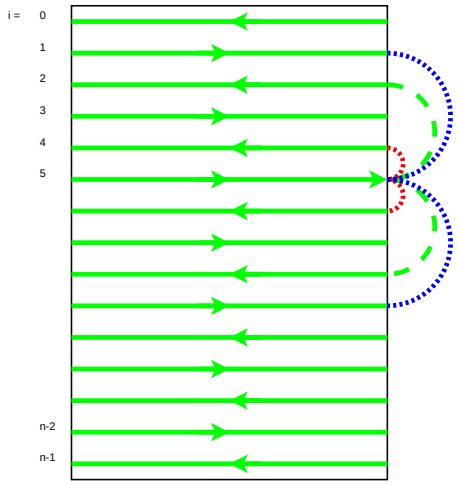


Figure 4.10: Simplification of a working area as rectangle. Each lane should be passed in a predefined direction denoted by the arrows. The maneuvers with low costs are marked green, where the maneuvers with high costs are marked red. The blue maneuvers are not allowed.

not be immediately determined as optimal until the whole tree is explored. Since the tree size is exponential in general and the number of lanes in a field is quite high, this approach is not feasible for practical use.

In case of a strategy where the directions of the lanes are considered, the decision tree can be built quite similar. In contrast to an undirected strategy, the size of the tree reduces, since half of the possible decisions are not allowed. Thus, the number of children of the root is $(n/2) - 1$ and the depth of the root is $n/2$. However, the tree has exponential size and therefore it can not be used in practice.

Finding a strategy using a set of rules

In contrast to a complete decision tree, another quite easy approach can be used to find a sequence of lanes. This approach is based on a set of rules which specify a local approach. In contrast to global approaches, the decision about the next lane is only based on a set of rules. Furthermore, once a decision is made, it can not be changed (i.e., no backstepping).

This approach obviously does not compute an optimal solution in general but the difference between the computed solution and the optimal solution decreases, if the amount of lanes increases. These rules can be defined either for a directed strategy or for an undirected strategy. In case of a directed strategy, a few more rules are necessary.

- One possible set of rules is listed below. These rules neglect the direction of the lanes. In this example, it is assumed that the amount of lanes is denoted by n. The current position of the CLR is denoted by i where $0 \leq i < n$.

  - If lanes $j_0, j_1, ..., j_l < i - k$ that have not been visited exist, choose the lowest j as next lane.

  - Otherwise choose the lowest following line $j > i + k$ which was not yet visited.
  - If the first two rules do not match, choose the predecessor $j < i$ with the lowest index as next lane.
  - If the first 3 rules do not match, take the successor $j > i$ with the lowest following as next lane.

- In contrast to the rules mentioned before, the following set of rules provides the possibility for covering directed lines. The additional rules guarantee the completeness of this approach. However, the application of the two last rules indicate a very poor strategy. Normally, these rules are only applied in the last few steps and therefore they can be neglected for the overall performance in case of a large number of lanes.

  Since the direction is considered in this case, the definition of the problem has to be extended by a function $d(i) \mapsto \{-1, 1\}, 0 \leq i < n$ which represents the different directions of each lane.

  - If lanes $j < i - k$ and $d(j) \neq d(i)$ that have not been visited yet exist, choose the lowest j as next lane.
  - Otherwise choose the lowest following line $j > i + k$ that has not been visited yet where the condition $d(j) \neq d(i)$ holds.
  - If the first two rules do not match, choose the predecessor $j = i - k - 1$ as next lane if it exists.
  - If the first three rules do not match, choose the predecessor $j < i$ fulfilling condition $d(j) \neq d(i)$ with the lowest index as next lane.
  - If all other rules do not match, take the successor $j > i$ fulfilling condition $d(j) \neq d(i)$ as next lane.
  - If no rule matches, take the lane with the smallest index.

The result of such an algorithm is a sequence of lanes and the desired direction of each lane. However, this result can not directly be used for a real area. Therefore, an approach has to be found which uses this result to compute an overall path covering the whole polygonal area.

Covering an arbitrary polygonal area

The result of the algorithm mentioned before is a sequence of lanes and the desired direction of each lane. Based on this algorithm, a feasible plan for a CLR has to be computed which covers an arbitrary polygonal field which possibly contains obstacles. Before a strategy can be computed the amount of lanes has to be determined. This can be done by counting the intersections of parallel lanes with the border of the working area and the obstacles. The number of intersection points of a lane can be interpreted as follows:

- If a lane has no intersection point, it is outside the working area and therefore it is not considered anymore.

- If a lane has exactly two intersections it is within the working area and crosses no obstacle. Thus, the number of lanes is increased by one.
- If a lane has exactly $2k$ intersections, it is inside the working area and crosses $k-1$ obstacles. Thus, $k-1$ paths have to be computed to avoid the obstacles. Obviously, the number of lanes is increased, too.

Once a strategy is computed (either directed or undirected), the first lane is retrieved from the result list and is added to the overall path. If the lane crosses one or more obstacles, a path is computed between two adjacent intersection points. Next, the second lane is retrieved from the result list and a path between the first and the second lane is computed. If the lane crosses obstacles, paths are computed to avoid them and so on. To illustrate the concept of this algorithm, Figure 4.11 shows the computation of the overall path. Details about surrounding of obstacles are presented in Section 5.5.

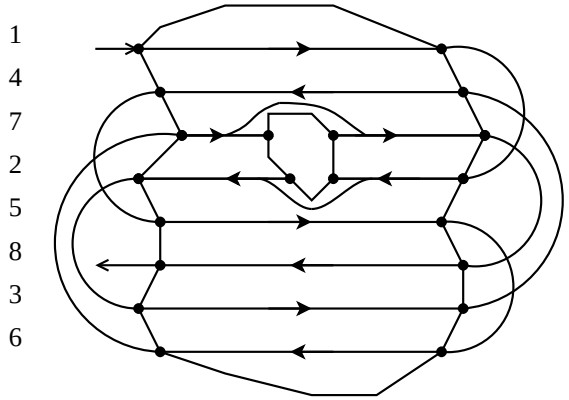


Figure 4.11: Construction of the overall path

CHAPTER 5

Implementation

5.1 Requirements

In the previous chapters, basic concepts of motion planning were introduced and motion planning approaches were presented especially for CLRs. This chapter describes a *proof-of-concept* implementation of such a motion planning algorithm for a CLR as claimed in Section 1.3. The implementation provides a Graphical User Interface (GUI) to edit the elements of the WS and to visualize the motion planning process.

5.1.1 Development environment

The implementation is realized in Java, since it provides a comfortable environment for developing platform independent GUIs. Another advantage of Java is that programs can be easily parallelized using threads. Thus, the following implementation tries to parallelize the path computation as far as possible. Moreover, the system is implemented in a way that it can be easily ported to a real system. However, this is out of focus of this thesis.

5.1.2 Target system

Since it is desired to implement the motion planning application for practical use, the whole implementation is based on a real-world lawn mower[1]. This vehicle has the steering mechanism on the rear axis. Thus, it is similar to a backward driving CLR. However, the CLR is able to drive forward, too. The distance between front and rear axis is about 2 meters (i.e., the length of the CLR w.r.t. the system model). The maximum speed of the desired vehicle is 10 km/h. The steering angle can be directly set w.r.t. the main axis of the CLR. The maximum steering angle is about +/-20 degree. This results in a turn radius of about 5 meters. It is assumed that the steering velocity is high enough to control the CLR at maximum speed. The desired speed

[1] e.g., Gianni Ferrari GT Range cf. `http://www.gianniferrari.com/`

of the car can be directly set using a low-level speed controller. Thus, the inputs of the system are the speed (including the direction) and the steering angle.

The vehicle is equipped with a precise single frequency RTK-GPS[2]. Thus, the position of the vehicle can be determined with an accuracy of less than 0,2 meters, i.e., at least 95% of all measured values are within a distance of 0,2 meters from the actual point. Furthermore, the heading of the vehicle is estimated with a digital compass[3]. This compass has an accuracy of +/-1 degree.

5.1.3 Working environment

As mentioned in Section 1.3, the working environment consists of three different types of polygons. The definition area limits the operating area of the vehicle. It is the outmost polygon and contains the working area and the obstacles. The size of this area is about $500 \times 500m$ but also larger areas are allowed. Thus, an ordinary definition area is about $200.000 - 300.000m^2$. Note that in case of a larger area, the computation time for a path increases dramatically.

5.1.4 Assumptions

For simplicity, it is assumed that the robot is a single point object. Thus, it is not necessary to consider the shape of the CLR. To ensure that no part of the robot gets out of the definition area or touches any obstacle, safety areas have to be defined as shown in Figure 5.1. Such a safety area can be realized by an offset shape of the corresponding polygons. Section 5.3.5 introduces an algorithm which computes an offset shape of a polygon based on *straight skeleton*.

5.1.5 Motion planning approach

Since the definition area is large and an accuracy of less than one meter is aimed, a conventional motion planning approach is not suitable. Consider for example a definition area with a size of 500×500 meters. Furthermore, assume that a grid of 1 meter is desired. The resulting 2D grid contains 250.000 nodes. However, this is not enough since the heading of the vehicle has to be considered, too. If the heading dimension is discretized into 60 different angles the number of nodes increases by a factor of 60. The search graph of this example would contain about 15 million nodes. Obviously, neither the construction of this graph nor a search in this graph is realizable in feasible time.

Since an RRT based motion planning algorithm promises good performance, this approach is used for the path computations of this implementation. To increase the quality of the path and the performance of the computation, a *hybrid RRT* approach (as introduced in Section 4.6) is used to find a feasible path between two configurations within the definition area.

Since the hybrid RRT approach is used, an auxiliary path is necessary. A method to find such a path is described in Section 5.4.1.

[2]Septentrio AsteRx1 PRO Sensor with RTK extension
[3]Honeywell HMR3300

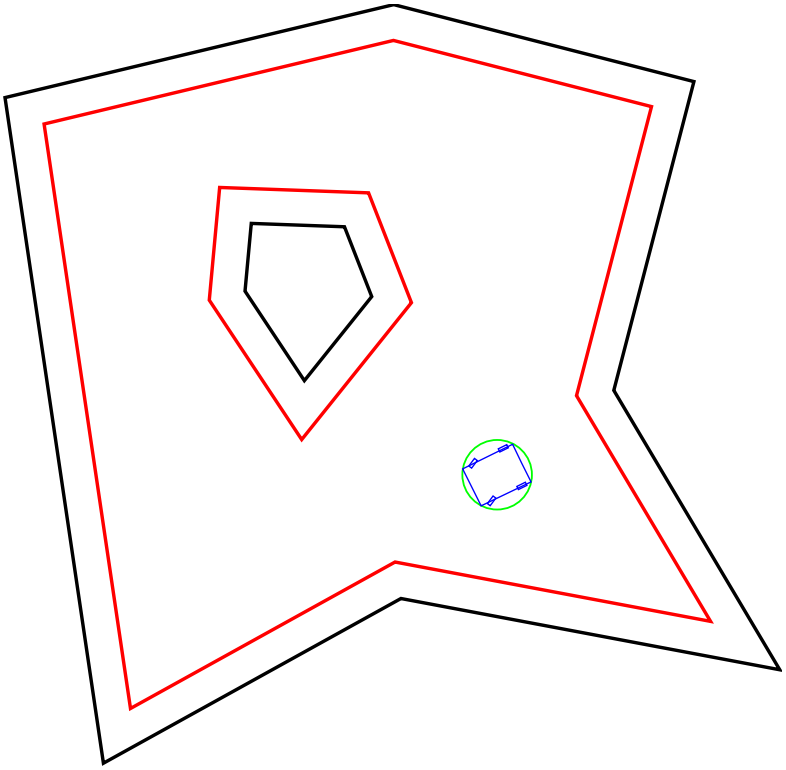


Figure 5.1: Definition area with obstacles

5.2 Data structures

Before the implementation of the motion planning algorithm is described, the basic data structures have to be specified which are used to map the physical environment to the motion planning application. First, the basic geometric data structures are addressed. Next, based on these definitions the representation of the workspace is shown. Since an object oriented approach is used, these data structures contain basic methods, too.

5.2.1 Geometric data representation

Point

The basic unit for geodetic computations is the representation of a point. As described in Section 2.1, a point can be defined by specifying the longitude, latitude and height. This definition is unique if it is assumed that the WGS84 ellipsoid model is used. Thus, a class `GPSPoint` as shown in Figure 5.2 can be used to represent a point using its geodetic coordinates. As mentioned in Section 2.2.2, it is assumed that the desired area (i.e., definition area) is flat enough and so the height w.r.t. the sea level is neglected. Thus, it is not stored in the class `PlanePoint`.

The basic operation between two points is the computation of the distance. Furthermore, another important operation is the computation of the heading between two points. The definition

GPSPoint
-MinutePoint: double [] -ReferencePoint: static GPSPoints
+GPSPoint(LatitudeMinute:double,LongitudeMinute:double) +GPSPoint(ThisPoint:PlanePoint) +GPSPoint(RefPoint:GPSPoint,XDistance:double,YDistance:double) +GPSPoint(Latitude:String,Longitude:String,EastWest:String,NorthSouth:String) +getCourseDistance(PointA:GPSPoint,PointB:GPSPoint): static double [] +getCourseDistance(PointA:GPSPoint): double [] +getPlanePoint(): PlanePoint +getNewPoint(distance:double,course:double): GPSPoint +getCoordinates(): double [] +setReferencePoint(latitude:double,longitude:double) +getDistanceXY(): double []

Figure 5.2: Class diagram of the `GPSPoint` class

of the heading is shown in Figure 5.3. Note that the heading angle is computed w.r.t. the y-axis (i.e., the north-south axis). To compute the distance and the heading between two points, the class `GPSPoint` contains appropriate methods. These methods use the approach presented in [14] to compute the distance and the heading. Figure 5.2 provides an overview of the class.

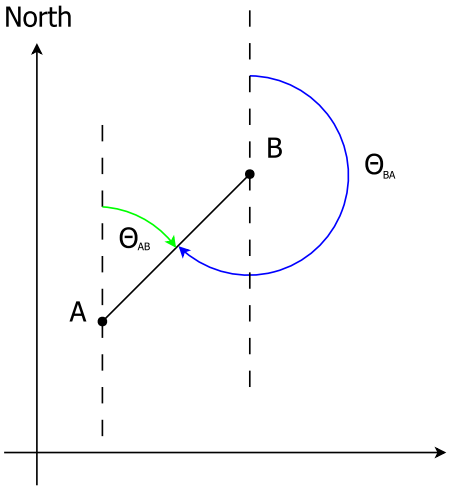


Figure 5.3: Definition of the heading between two GPS Points

As the distance computation is the most used function in this motion planning application, the poor performance of the approach presented in [14] would compromise the overall performance of the motion planning application and the real-time capability of a motion controller.

To overcome this problem, the representation of all points is changed. Based on an arbitrary but fixed reference point, all points can be defined by specifying the heading and the distance to the reference point. Since the reference point is known, this representation is unique, too. However, it is possible to use the distance along the north-south axis and the distance along the west-east axis as a unique representation of the point. For simplicity, the north-south axis

is denoted as the y-axis and the west-east axis is denoted as the x-axis. Thus, every point can be defined in a cartesian coordinate system. Note that the reference point is in the origin of the coordinate system. To store a plane point in the application, the class `PlanePoint` (cf. Figure 5.4) is used to store its x-y-coordinates w.r.t. the reference point. To determine a unique point, the class contains a static attribute which specifies the position of the reference point. This reference point is represented by an instance of the class `GPSPoint`. If a point is imported which is represented by geodetic parameters, it is converted to a planar representation w.r.t. the reference point by the constructor of the class `PlanePoint`. An overview about this class is provided in Figure 5.4 where the class diagram is shown.

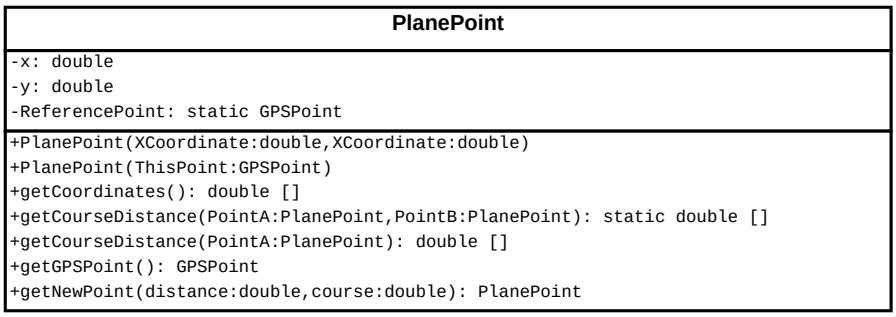


Figure 5.4: Class diagram of the `PlanePoint` class

In some cases, it is necessary to know the geodetic coordinates of a point. Thus, the class `PlanePoint` contains a method which is able to convert every instance to an instance of `GPSPoint`. Based on the statically stored reference point and the plane coordinates, the geodetic coordinates of the desired point can be computed. To achieve this computation, an approach presented in [14] is used. This approach is able to compute a desired geodetic point based on distance and the heading from the reference point. Since the position of the desired point is determined by cartesian coordinates w.r.t. the reference point, the distance and the heading from the reference point to the desired point can be determined. Thus, the geodetic coordinates of the desired point can be determined by converting the cartesian coordinates to polar coordinates using the approach of [14].

Note that the conversion between `PlanePoint` and `GPSPoint` has no influence on the desired point in theory. However, due to the limited precision of the data representation, conversions between these representations decrease the accuracy of a point. Thus, they are only converted if it is absolutely necessary.

Line

If lines are considered precisely, it turns out that a few different types of lines exist. An overview about these different types is shown in Figure 5.5.

- A straight line can simply be defined by specifying two points (the *definition points* A and B) but these points do not delimit the line. Thus, a straight line is infinite in both

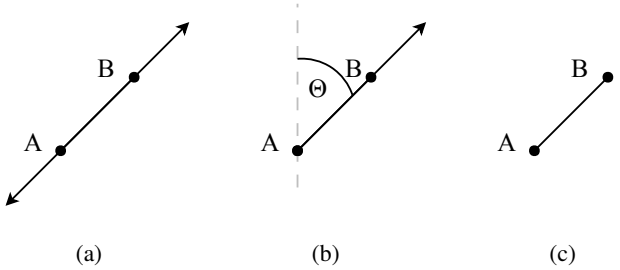


Figure 5.5: Different types of lines

directions w.r.t. the definition points. An intersection between two lines exists, if they are not parallel. Figure 5.5(a) shows an example of a straight line.

- In contrast to a straight line, a ray is only infinite in one direction. In Figure 5.5(b), an example of a ray is presented. If a ray is defined by two points similarly to a straight line, only one of them delimits the ray and therefore it is called the *endpoint* (A). The other point is only used to determine the direction of the ray and is called the *definition point* (B). A ray can also be defined by specifying the *endpoint* (A) and the direction (θ) of it.

- As shown in Figure 5.5(c), a line segment is usually defined by two points similarly to the straight line. These points delimit the line segment and therefore they are called *endpoints*. Note that a line segment can be specified by other parameters, too. However, this requires more than two parameters and therefore it is impractical.

A class `Line` represents all different types of lines. In Figure 5.6, the corresponding class diagram is presented. Since every line-type can be specified by two points, the class contains two attributes of type `PlanePoint` to specify a line. To determine the type of the line, the class contains also an attribute which specifies the line type. The type of the line is important especially for the method which intersects two different types of lines. More details about the implementation of intersections are presented in Section 5.3.1.

Polygon

A polygon is represented by the class `Polygon` as shown in Figure 5.7. The main attribute of this class is a linked list of `PlanePoint` objects. This list contains the vertices of the polygon. Due to this data structure, it is easy to insert or remove points. The points in this list are stored either in clockwise or counter-clockwise order. This order is very important for further computations (e.g., *straight skeleton*). Since either polylines or polygons are represented by this class, an attribute is used which determines whether the polygon is ”open” or ”closed”.

One of the most important operations in this context is collision checking. Thus, this class contains the method `isInside()` to determine whether a point is inside a polygon or not. The approach used by this method is described in Section 5.3.4.

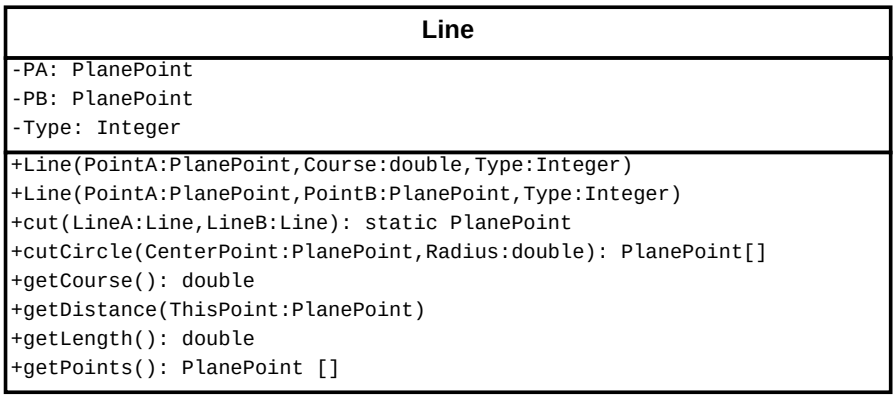


Figure 5.6: Class diagram of the line class

Furthermore, this class contains the method `cut()` to compute the intersection with lines represented by the class `Line`. This method is used by the line intersection method of `Line` to successively check every segment of the polygon for intersection.

Moreover, the method `getDirection()` is provided which determines the direction of the stored polygon. This method is based on the computation of the cross product between two adjacent segments (i.e., three consecutive points). A detailed description is presented in [18].

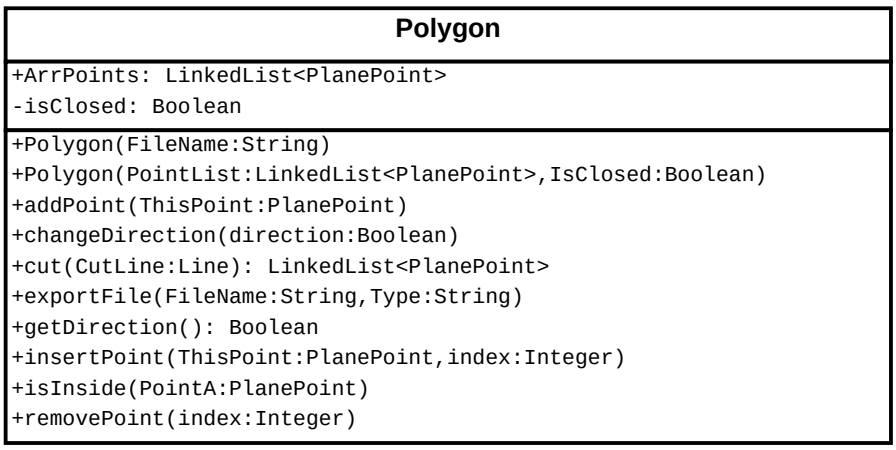


Figure 5.7: Class diagram of the polygon class

5.2.2 Representation of the workspace

As described in Section 1.3, the workspace contains the definition area, the working area and a set of obstacles. These areas are represented by simple polygons, i.e., instances of the class `Polygon`. Since the definition area contains an undefined amount of obstacles, these obstacles are stored in a linked list of polygons.

As safety areas have to be defined to avoid violations of forbidden areas (e.g., inside an obstacle or outside the definition area), a method is used to compute an offset shape of the definition area and the obstacles.

This class also contains the methods `setDynamicObstacle()` and `resetDynamicObstacle()` to add and remove a dynamic obstacle, respectively. These methods are used to model an obstacle which is detected by the safety sensors of the vehicle during runtime. Since the contour of this object can not be determined, it is assumed that the dynamic obstacle is a rectangloid object. Based on this obstacle, a bypass route is computed which surrounds this obstacle and goes back to the original path. If the vehicle is back on the original path, the dynamic obstacle is removed. Details about this method are described in Section 5.6.

Since an RRT approach is used and the computation of random points is computational expensive, a set of random configurations is computed before a path can be computed by the method `generateRandomPoints()`. This method is a preprocessing step and has to be done only once for the workspace.

Furthermore, the method `getPath()` is used to compute a path based on two points in the plane with corresponding headings. This method is the most challenging part of this implementation. Details about the implementation of this method are presented in Section 5.4.

The method `calcLanes` computes the lanes within the working area. As aimed in the introduction, the distance between two lanes and the direction of the lanes can be defined. Based on this method, a complete path can be computed.

The method `calcCompletePath` computes a path which covers the whole working area. Based on the previously called method `calcLanes` and the method `getPath`, a feasible path is computed. A detailed description of this method is presented in Section 5.5.

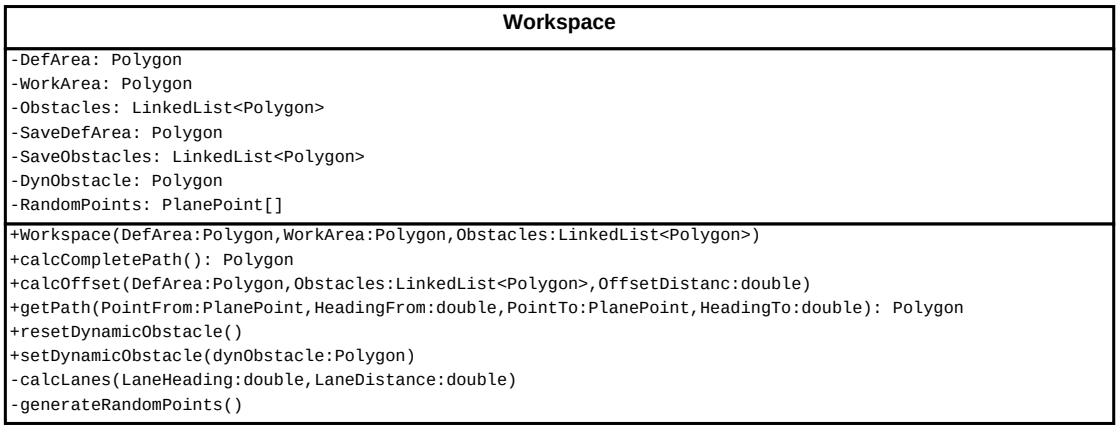


Figure 5.8: Class diagram of the workspace class

5.3 Geometric operations

5.3.1 Line intersection

Intersection is one of the most important operations between two line segments. Therefore two different approaches are presented to compute the intersection. Note that it is assumed that a line (any type) is presented by two points (i.e., class `Point`) and that all of these three presented line types can be represented by this class.

A naive method to compute the intersection between two lines is to use a linear equation. Based on

$$\begin{aligned} y_a &= k_a x_a + d_a \\ y_b &= k_b x_b + d_b \end{aligned} \tag{5.1}$$

where the slopes k_a, k_b and the y-deviations d_a, d_b are given, the intersection point can be simply computed by setting $y_a = y_b$ and $x_a = x_b$. However, this method fails if one of the lines is parallel to the y-axis.

A quite better approach uses trigonometric functions to compute the intersection [28]. The first step is to compute the distance between a *definition point* of a line and the intersection point. In the second step, the relative position of the intersection point w.r.t. the definition is calculated and finally the absolute position is computed based on the position of the definition point. The following equations show how the intersection point can be calculated using four definition points. Figure 5.9 illustrates the variables of Equation 5.2. Note that the variable s denotes the distance between the two points in the subscript.

$$\begin{aligned} s_{a_1 p} &= s_{a_1 b_1} \cdot \tfrac{\sin(\gamma-\beta)}{\sin(\gamma-\alpha)} \\ y_p &= y_{a_1} + s_{a_1 p} \cdot \cos(\alpha) \\ x_p &= x_{a_1} + s_{a_1 p} \cdot \sin(\alpha) \end{aligned} \tag{5.2}$$

To check the intersection between line segments or rays, it has to be checked if the computed intersection point is on the line or ray. This can be done by computing the heading of the endpoint(s) to the intersection point. If the heading is equal with the heading from the appropriate endpoint to the other *endpoint* or the *definition point*, the point is on the line. Note that in case of a line segment, this check has to be performed for each endpoint.

5.3.2 Line-circle intersection

Another important operation is the intersection of a line with a circle with given center point and radius. In general, a line has either no intersection point or two intersections with a circle. The case that the line is a tangent of the circle is neglected because of the computational inaccuracy. Note that if a line segment or a ray is considered instead of a line, it is possible that only one intersection exists.

According to Figure 5.10, the point H is computed by the intersection of the given line and a ray starting in the center point of the circle and orthogonal to the given line. Since the distances s_{CH} and the radius $r = s_{CQ} = s_{CP}$ are known, the distance $s_{HP} = s_{HQ}$ can be calculated. Finally, the two points can be calculated by specifying the distance and the angle between H and the desired points P and Q.

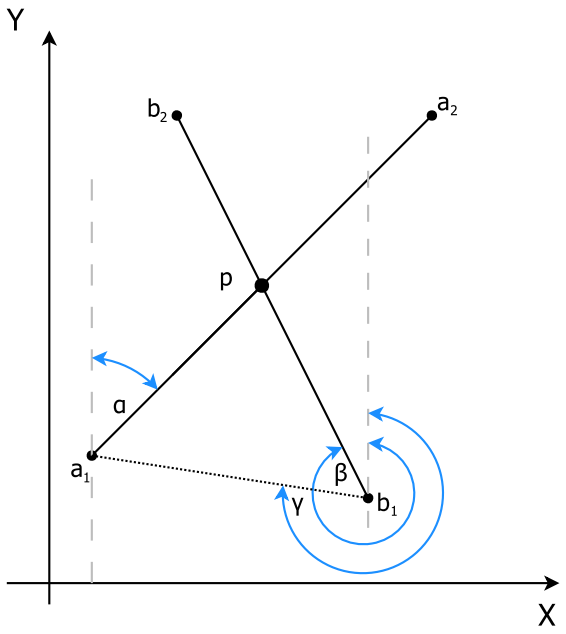


Figure 5.9: Intersection between two lines

5.3.3 Polygon intersection

Another basic geometric operation is the intersection between two polygons. Before intersection algorithms are examined, it has to be determined whether the polygons are convex or not. If the polygons are convex, it is possible to determine the intersection points in linear time.

Convex polygons

Let P and Q be two convex polygons where $P = (p_0, ..., p_{n-1})$ and $Q = (q_0, ..., q_{m-1})$. The determination of the intersection points between two polygons P and Q takes at least $\mathcal{O}((n+m))$ [29]. In [30], an algorithm is presented which computes the intersection points between two polygons P and Q. This approach fails if at least one polygon is not convex.

Arbitrary polygons

In case of arbitrary polygons, no efficient algorithm exists. Thus, the simplest method to intersect two polygons P and Q is to test every edge of P with every edge of Q for intersection. Obviously, the complexity of this approach is $\mathcal{O}(nm)$ [29].

5.3.4 Point in polygon

To check whether the CLR is within the definition area or within an obstacle, it is necessary to determine if a given point is within a polygon. Based on the line intersection algorithm of Section 5.3.1, it can be determined if a point is within a polygon or not.

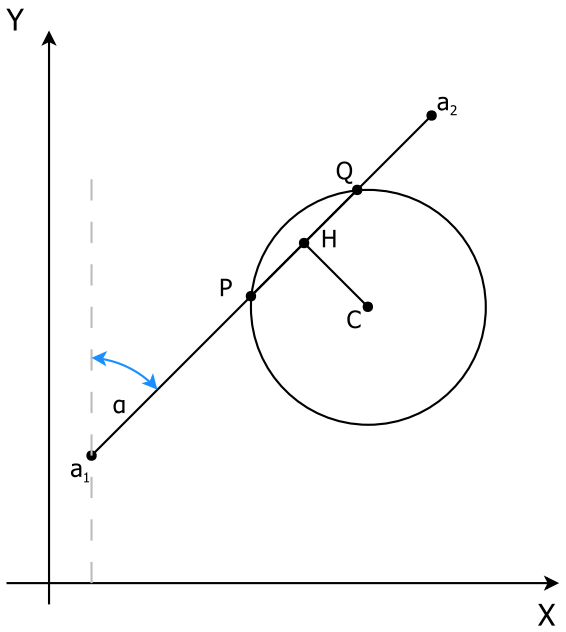


Figure 5.10: Intersection between a line and a circle

Consider a ray starting from the desired point with an arbitrary direction. Furthermore, assume that only simple polygons are considered. Obviously, the ray intersects the polygon if it is inside. In case of a convex polygon, this is a sufficient condition. In general, (i.e., in case of a non-convex polygon), this is a necessary but not a sufficient condition.

A sufficient condition for the general case is found by counting the number of intersections of the polygon with the ray. If the number of intersections is even, the point is outside the polygon. The point is inside the polygon if the number of intersections is odd. To illustrate this method, Figure 5.11 shows two examples of points. The red point (A) is outside the polygon and intersects 4 times. In contrast, the green point (B) is within the polygon and has three intersection points.

5.3.5 Polygon offsetting

Another important geometric method is *polygon offsetting*. This means that a shape of a simple polygon is computed in a given distance. This can be used to realize safety areas for the definition area (inside offset) or for obstacles (outer offset). An example of such a contour is shown in Figure 5.1. To compute an offset contour, a *straight skeleton* of a polygon has to be computed [3].

In [8], a detailed algorithm is presented which constructs a straight skeleton of simple polygons. The presented algorithm can be used to compute a straight skeleton of a simple polygon including holes. However, the considered polygons used by this implementation are only simple

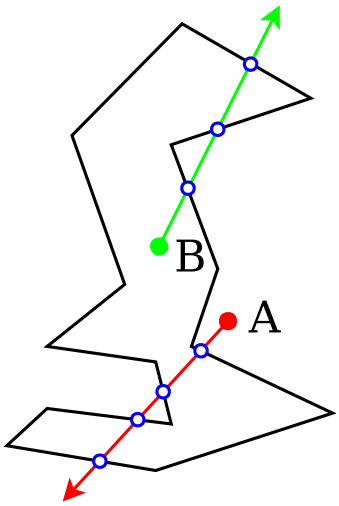


Figure 5.11: Example of points in polygons

polygons without holes but the definition area may contain simple polygons (i.e., the obstacles). Thus, the obstacles within the definition area are interpreted as the holes of the polygon. Figure 5.12 shows the straight skeleton of a simple polygon containing a hole.

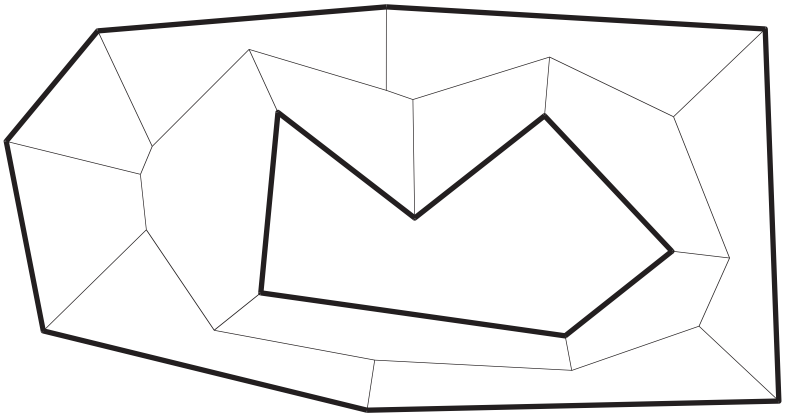


Figure 5.12: Example of a straight skeleton for a polygon with holes (taken from [8])

5.4 Path computation using a rapidly-exploring random tree approach

Based on the previously described environment, the RRT motion planning approach was implemented. This motion planning approach uses random configurations to construct a tree of feasible paths. To compute a path between two points, two trees are constructed (one from

the goal configuration and one from the initial configuration). Since Java supports threads, the construction of these trees can be parallelized. If the construction is parallelized, the different threads have to be synchronized because each thread accesses the tree of the other thread.

Before an RRT algorithm can be implemented, a random configuration generator has to be implemented. Furthermore, a distance metric has to be introduced which estimates the distance between two configurations. Finally, the construction of the random tree has to be specified.

To take advantage of multi threading, it is also possible to construct different pairs of trees with different strategies in parallel. Thus, for example, it is possible that a conventional RRT algorithm is used beside a hybrid RRT algorithm. Finally, a faster or even the better result can be used. Details of the multi threading approach and the synchronization mechanism are presented in Section 5.4.5.

5.4.1 Auxiliary path

Since the hybrid RRT approach was implemented an auxiliary path is necessary. The auxiliary path only guides the RRT search. It is not necessary to meet the constraints of the CLR or provide a complete solution. Thus, a simple grid based approach is used as shown in Figure 5.13. This grid only considers the planar coordinates of the system and neglects the heading information. A state is only sampled if it is inside the definition area and not within an obstacle. After the computation of the nodes, they are connected to their adjacent neighbors as illustrated by the blue points in Figure 5.13.

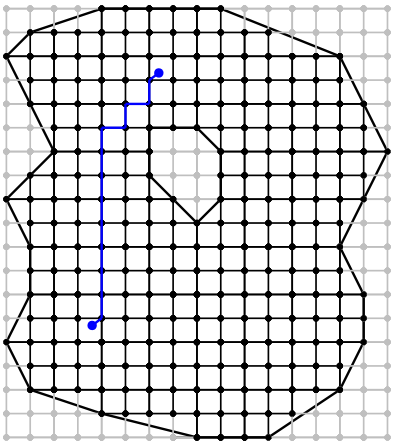


Figure 5.13: Example of an auxiliary based on a state lattice

The graph is constructed by using the class `Node` which contains the plane coordinates of the node (i.e., an instance of the class `PlanePoint`). Furthermore, the class contains four static pointers to the adjacent neighbors. All nodes are stored in a two dimensional linked list according to their position in the grid.

To find a path between two arbitrary configurations within the definition area, the nearest node in the graph of the initial configuration and the goal configuration has to be determined. The nearest node is found by simply checking the Euclidean distance of all nodes. Obviously,

this is not an optimal solution but the effort can be neglected, compared to the effort of the whole task.

If a path is computed, the nearest node in the graph w.r.t. the initial and goal state is determined. Based on these two nodes in the graph, an implementation of Dijkstra's algorithm is used to find a path between them. It is assumed that the distance between all adjacent nodes is equal to simplify the search.

Based on the resulting path of this process (cf. Figure 5.13), a normally distributed channel can be computed.

5.4.2 Random point generation

This section presents methods to generate random configurations. As it is shown in the following section, the distance metric uses only the plane coordinates. Therefore, it is sufficient to compute a random plane point and so the heading of the CS can be neglected.

Since a hybrid RRT approach is used in the described implementation, it is necessary to compute two different types of random points. On one hand, uniformly distributed random configurations have to be computed to cover the whole definition area. On the other hand, normally distributed random configurations along a given auxiliary path have to be computed.

Uniformly distributed random points

Since it is difficult to compute random points within a polygon (i.e., a random number has to be mapped to an element in an arbitrary set), random numbers are generated over the whole *bounding area*. The *bounding area* is delimited by the maximum horizontal and vertical expansion of the definition area as shown in Figure 5.14.

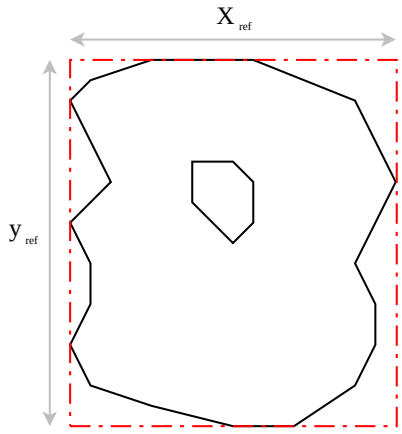


Figure 5.14: Bounding area of a definition area

Based on the horizontal (x_{ref}) and the vertical size (y_{ref}) of the bounding area, two independent uniformly distributed random numbers are generated using a Java standard library ($x_{rand} = \mathcal{U}(0, x_{ref})$, $y_{rand} = \mathcal{U}(0, y_{ref})$). Based on these two values, an instance of `PlanePoint` is computed which represents a random state.

However, it is possible that this random state is either within an obstacle or outside the definition area. Thus, it is has to be verified, if the plane point is within the free portion of the definition area. To achieve this, the method presented in Section 5.3.4 is used.

If a point is not valid, it is discarded and another random state is computed until a valid random state is found. If a random sample of the whole CS is desired, it is possible to extend this method by computing a random number between 0 and 2π. This random number represents the third dimension (i.e., the heading) of the CS.

Normally distributed random points along the auxiliary path

In contrast to the computation of a uniformly distributed random state, this method requires an auxiliary path in addition to the definition of the WS.

First, the length L of the auxiliary path is determined. Furthermore, a uniformly distributed random number $l = \mathcal{U}(0, L)$ is computed. In addition to this random number, another uniformly distributed random number α is defined as $\alpha = \mathcal{U}(0, 2\pi)$. Subsequently, a normally distributed random number ($d = \mathcal{N}(\mu, \sigma^2)$) is computed. The mean μ is chosen as zero and the variance is chosen as $\sigma^2 = (\frac{D}{2})^2$ where D represents the aimed width of the channel.

The computation of a random configuration on the channel can be computed by the following steps. Figure 5.15 illustrates this construction phase.

1. First, a point on the auxiliary path in distance l from an endpoint has to be found. Denote this point as Q. Since l is uniformly distributed, it does not matter which endpoint is chosen.

2. Next, starting from Q a ray with angle α is constructed.

3. The desired random point Q' is defined as a point on the ray in distance d from Q.

Note that this approach does not compute an exact normally distributed channel along the auxiliary path. However, this approach is quite sufficient for this purpose and can be implemented efficiently.

5.4.3 Distance metric

One of the most important functions used by the RRT approach is the distance metric. This function is used to estimate the distance (or even the costs) between two configurations in the CS. Since the performance of the whole algorithm strongly depends on this function, it has to be implemented efficiently. There are lots of possibilities to estimate the distance. A few methods are described in the following:

- **Local planner**
  As mentioned in Section 4.2.2, a local planner based on *Dubins' car* can be used. This approach provides a method to compute the optimal distance between two configurations. If obstacles between these configurations occur, the planner fails.

  Since the effort for computing a *Dubins' curve* is high, this algorithm is not used as distance metric for the RRT approach.

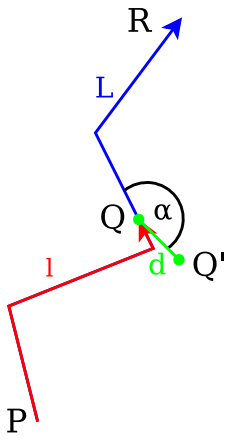


Figure 5.15: Computation of a random point on the auxiliary channel

- **3D Euclidean distance**
  Another approach uses the Euclidean distance. This metric can be computed efficiently. Since the third dimension of the CS represents the heading of the vehicle in an interval $[0, 2\pi]$, it is necessary to scale the value according to the other dimensions (i.e., plane coordinates). The easiest method is to use the size of the average value of the bounding area (Figure 5.14) as reference value for the direction. Thus, the adapted direction z can be computed as $z = \frac{\theta}{2\pi} \cdot \frac{x_{ref}+y_{ref}}{2}$ where x_{ref} and y_{ref} represent the size of the bounding area.

- **2D Euclidean distance**
  Furthermore, it is possible to simply use the Euclidean distance of the plane coordinates. Obviously, the heading of the CLR is neglected. Since a precise estimation is not absolutely necessary, this method can be used in practice.

To evaluate the impact of the distance metric, the latter two methods were tested. It turned out that the distance metric using the heading has an average runtime, poorer than the distance metric using only plane coordinates. The tree which uses the 2D distance metric explores quite faster than the other distance metric.

It also was tried to vary the scaling factor of the heading dimension. It turned out that the speed of the path computation decreases if the scaling factor of the heading is increased.

5.4.4 Tree expansion

To successively extend the tree, a function is necessary which enlarges the tree.Based on the random configuration and the distance metric, the nearest node in the tree is determined. To achieve this the distance between the random configuration and every node of the tree is successively checked. After every node was checked, the nearest one is selected. Starting from the nearest configuration of the tree, a feasible path has to be computed which moves the vehicle

towards the random configuration. Figure 5.16 illustrates the expansion of a tree node. In this case, the tree node has two successors (X^1_{new} and X^2_{new}) which are computed based on the two random configurations X^1_{rand} and X^2_{rand}. Note that the path between X^i_{new} and X^i_{rand} is not a part of the tree because the tree is only expanded a constant or random distance towards the random configuration.

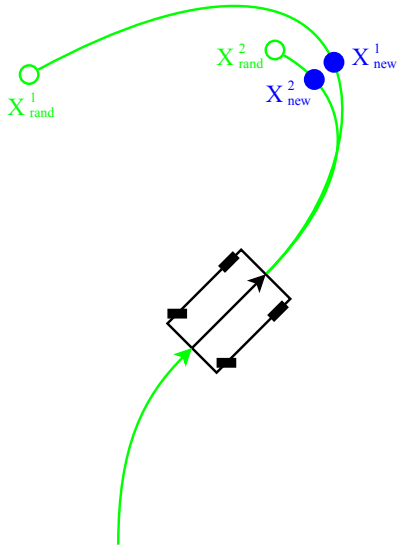


Figure 5.16: Expansion of the tree based on a local planner

To realize a method which computes a path towards the random configuration, different approaches can be used:

- One possibility is the use of a local planner based on *Dubins' car* or *reeds and sheps car*.
- Another possibility is to use straight line segments with restricted angles to expand the tree.
- Finally, a simulation of the desired system can be used to compute a feasible path. In this case, a combination of a system model and a controller is used to compute a path.

Since the last method is very flexible and can be efficiently implemented, it is used to compute the path within the described application. Figure 5.17 shows the concept of this approach. The system starts simulating from the nearest configuration in the tree. The input of the controller is the desired random point. A feedback loop between the controller and the virtual system is used to move the vehicle towards the random point. The most important property of this process is that the virtual system model generates a feasible trajectory.

Since time discrete models are used, the control process is iterated a given number of steps. The last output of the system model is defined as the new node of the tree. The following two paragraphs describe the system model and the controller as shown in Figure 5.17.

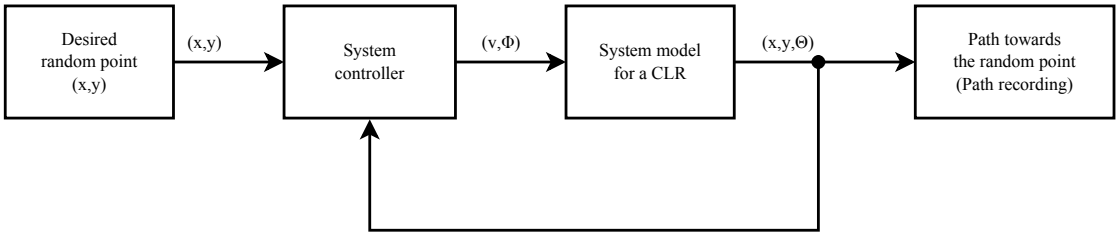


Figure 5.17: Block diagram of the trajectory generator

Simulation model

Basically, the simulation model is a discretization of the simplified *single track model* as described in Section 4.2. Thus, the time discrete model is defined as follows:

$$\begin{aligned} X_{n+1} &= X_n + (v \cdot \Delta t) \cdot \cos(\theta_n + \tfrac{\psi \cdot \Delta t}{2}) \\ Y_{n+1} &= Y_n + (v \cdot \Delta t) \cdot \sin(\theta_n + \tfrac{\psi \cdot \Delta t}{2}) \\ \theta_{n+1} &= \theta_n + (\psi \cdot \Delta t) \end{aligned} \tag{5.3}$$

Obviously, this system of equations computes the next state (i.e., $X_{n+1}, Y_{n+1}, \theta_{n+1}$) based on the speed v, the steering angle ϕ, and the previous state (i.e., X_n, Y_n, θ_n). Note that ψ instead of ϕ is used in the equation. However, ψ can be determined as

$$\psi = \frac{v}{L} \tan(\phi). \tag{5.4}$$

Since this model describes a rear wheel driven car and the vehicle normally moves backwards, the speed variable is set to a negative value in general.

Depending on the time interval Δt, the model behaves differently. If the time interval is too large, the system may gets unstable.

Controller

While the system model simulates a real system (i.e., the CLR), the system controller is used to move the vehicle towards the desired point (i.e., the random point). Generally, lots of motion controllers for CLR exist. A detailed overview about system controllers for CLRs is presented in Section 6.3. For this purpose a controller called *pure pursuit controller* is used. Figure 5.18 shows the geometric model of the vehicle and the concept of the controller. Based on the angle η (which is the angle between the line segment and the main axis of the CLR), the steering angle is computed. However, the distance between the point on the path and the vehicle depends on the speed. Note that this controller does not control the speed of the vehicle. It requires only that the speed is constant.

To use the *pure pursuit controller* for this purpose, it has to be adapted since no path or trajectory is given. Thus, the line segment is computed between the CLR and the desired static point. Similar to the conventional *pure pursuit controller*, the steering angle is computed. Equation 5.5 defines the corresponding control rule for the system. Note that only the steering angle is controlled, the speed of the vehicle has to be constant.

$$\phi = -\arctan(\frac{L \sin \eta}{\frac{L_{rv}}{2} + l_{rv} \cos \eta}) \tag{5.5}$$

The length l_{rv} represents the distance covered by the vehicle until a steering command changes the heading of the vehicle. This value corresponds to the response time of the system. Since it can be assumed that the speed of the car is slow this parameter can be neglected.

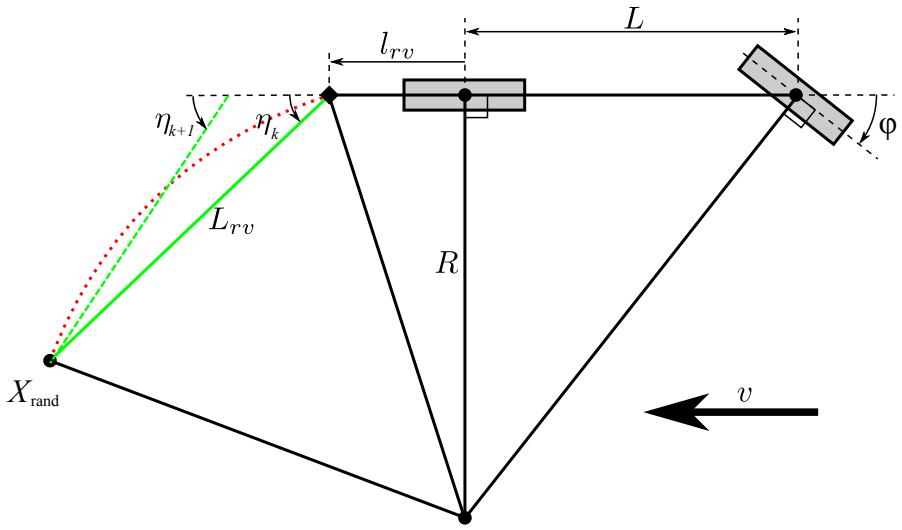


Figure 5.18: Pure pursuit controller which moves the vehicle towards X_{rand} (taken from [9])

5.4.5 Optimizations

Since state-of-the-art personal computers have at least two processor cores and the Java Virtual Machine (JVM) executes threads as native threads, it is possible to parallelize the path computation. This increases the utilization of resources and the performance of the execution. The following optimizations are possible and have been implemented:

- Since the path computation uses at least two random trees, each of them can be executed as one thread. To avoid race conditions, it must be ensured that one tree does not change while the other tree checks for intersection.
- Another possibility to increase the performance of the algorithm is to compute a number of tree-pairs simultaneously. Since these are independent instances of the algorithm (possibly with different parameters), it is not necessary to synchronize them. However, it is necessary to terminate all threads, if a feasible path was found by one tree-pair.

In the proposed solution, the following approach is used: First, a method is used which initializes and starts all threads. Additionally, it initializes an instance of a sync class. This so called *sync object* is responsible for synchronizing all threads. Thus, the sync object contains all running

tree-threads. Furthermore, this object contains flags which indicate that a feasible path was found. Before any thread starts with the expansion phase of its tree, it uses a method to check if another tree-pair has found a path. In this case, the thread terminates.

If a tree-pair has found a path, it stores the result in the sync object and calls a method which indicates that a path was found. Finally, this method wakes up the main thread and terminates its own thread.

Since the resulting path is stored in the sync object, it can not be overwritten by another thread which was not terminated yet.

The method which initialized and started all threads (including the sync object) sleeps until a path is found. If a path was found, this thread is waken up and it is able to read the resulting path from the sync object.

It is also possible to compute a number of possible paths which are stored in the sync object. This means the computing threads are only terminated if a given number of paths were found or if a given time threshold was reached. Obviously, this variation of the sync object calculates a better path (w.r.t. the length) but also results in a longer runtime.

5.5 Construction of the overall path

In the previous section, the implementation of an algorithm was described which is able to find a path between two arbitrary points in the definition area. This section describes an extension to this implementation which is able to cover a working area as claimed in Section 1.3. In Section 4.7, a strategy was presented to compute an ordered set of lanes which cover predefined area. However, this section describes the implementation of this strategy.

Given a desired heading and a desired lane distance, a method was implemented which simply computes an ordered set of points and headings. Figure 5.19 illustrates the construction of this list.

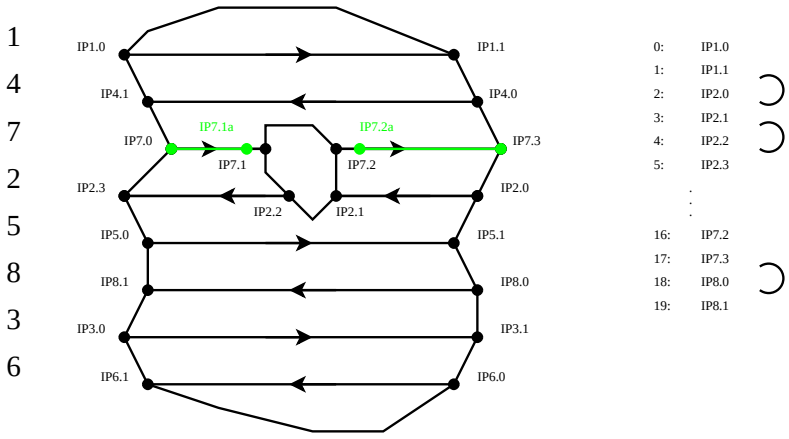


Figure 5.19: Construction of a path covering the working area

- Before a strategy can be computed, the number of lanes has to be determined which are necessary to cover the area. Starting from an arbitrary point within the working area, an infinite lane is computed with a given heading. Based on this lane, offset lanes are computed in both directions. This is continued until an offset lane does not intersect with the working area.
- Once the number of lanes is determined, a sequence of lanes is computed based on the strategy of Section 4.7. Since this algorithm is only based on a set of rules, it can be implemented in a simple way. The result of this algorithm is a strategy representing an order of lanes. In addition to the order, the aimed direction is specified. In Figure 5.19, this sequence is denoted by the numbers of the next lanes. Furthermore, the direction of each lane is denoted by the arrow.
- Based on this strategy, the intersection points of the first lane are computed w.r.t. the desired direction. Subsequently, the next lane (defined by the strategy) is selected and the intersections are computed iteratively. All these intersections are stored in an ordered list of points. Furthermore, the heading of each intersection point is determined by the heading of the lane and the desired direction. If this process is continued for all lanes (specified by the strategy), a list of points and headings is computed. In Figure 5.19, every intersection point is denoted by $IP\ x.y$ where x denotes the lane number and y denotes the order of intersections of one lane. All these intersection points are stored in an ordered list according to their names.
- As be seen see in the figure, a path is only necessary from an odd item to an even item of the ordered list. The path from the even items to the odd items is determined by the lane. Note that this is independent from the amount and the shape of the working area or obstacles. Thus, based on the these points and the appropriate headings, the method `getPath()` is used to determine a path.
- One can see, that it is not always possible to compute a feasible path around obstacles (e.g., $IP7.1 - IP7.2$ in the figure). Thus, a simple *try and error* approach is used to compute a feasible path. After the computation of a path failed, the initial and the goal points are redefined. Since every computed path starts at an end of a lane and ends at the beginning of a lane, these lanes are simply shortened. The intersection points $IP7.1a$ and $IP7.2a$ in Figure 5.19 illustrate this approach. The distance between the original and the new intersection points depends on the length of the whole lane. However, this distance has a predefined minimum and maximum length. If the path computation fails again, this step is repeated.

5.6 Dynamic collision avoidance

At the beginning of this section, it was mentioned that methods were implemented to avoid collisions with dynamic obstacles (e.g., people, vehicles, equipment). If an obstacle occurs in front of the vehicle, a mechanism is necessary that computes a path which surrounds the unexpected obstacle and continues driving on the original path.

Since the vehicle does not know the shape of the obstacle, it has to estimate the size and the contour of it. In the presented approach, it is assumed that the obstacle looks like a rectangle with a predefined size. Furthermore, it is assumed that the obstacle is immediately in front of the vehicle. Based on the position and the heading of the vehicle, the corresponding polygon is computed and stored in the `Workspace` object.

In contrast to the normal path computation, this method has to be executed during runtime. Thus, the performance of the chosen method is very important. In contrast to the normal planning method, this approach must allow backward motion because the obstacle is in front of the vehicle. Since the change of the driving direction is a time consuming process (i.e., change gears), the amount of direction changes has to be kept low.

Since the RRT approach is also suitable for this task, the implementation of this method is based on the RRT implementation described in Section 5.4. Note that the expansion step based on a simulation is used for this dynamic case, too. However, it is not possible to use the previously described implementation directly. The adaptions and extensions of the previously described implementation are described as follows:

- The used method constructs only one tree because no explicit goal position is desired. If a path from the actual position to the original path is found, the tree expansion terminates. To avoid that the computed bypass route directly leads to the end of the original path, a cost function is introduced. This cost function is based on the length of the bypass route and the path length between the actual position and the position where the bypass route meets the original path. Every path with a length above a given threshold is dropped. As mentioned in the previous section, threads can be used to compute multiple instances of the algorithm to decrease the computation time.

- Since forward and backward motions are allowed, it has to be decided whether the vehicle moves forward or backward towards the random configurations. This decision can simply be done by determining the heading of the random point w.r.t. the heading of the vehicle. Figure 5.20 shows the actual position of the vehicle and four random positions where the algorithm decides to move forward for the upper points and backwards for the lower points. However, it is possible that one direction is not allowed as mentioned below. In this case, the CLR can only move in the allowed direction.

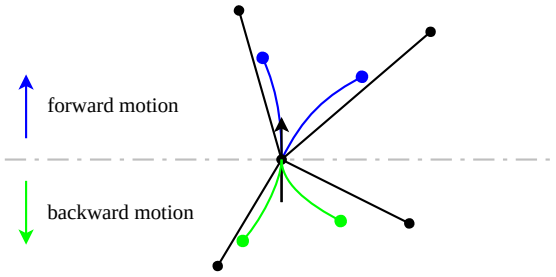


Figure 5.20: Direction of the motion based on the random point

- If the expansion of the tree is not restricted, it is possible that a path changes the direction after each node. To avoid this behavior, an additional restriction is necessary. This restriction ensures that the direction is only changed after a predefined number of nodes. This can be realized by storing the direction and a counter in each node. This counter contains the number of consecutive predecessor nodes with the same direction. The direction can only be changed if the counter is above a given threshold.

CHAPTER 6

Simulation

6.1 Simulation overview

Based on the results of the motion planning application, a simulation is used to verify the feasibility of the computed path. This simulation is also implemented in Java and is integrated in the motion planning application as described in the previous chapter.

Before details about the simulation are described, the basic structure of the framework has to be determined. Figure 6.1 shows a block diagram of the simulation process. The input of the simulation is a simple geometric path. In contrast to a trajectory tracking controller, the controller uses a static path as input. Based on the actual state of the system, the controller has to estimate the desired position on the path. Furthermore, the controller has to set the input values of the system to keep the system on the path. To prevent the use of an Ordinary Differential Equation (ODE) solver, a time discrete system model is used.

The system is controlled by a lateral controller which only actuates the steering angle. The speed value is not handled by the controller but it is used as input variable to determine the steering angle. Thus, it is assumed that the speed value is constant.

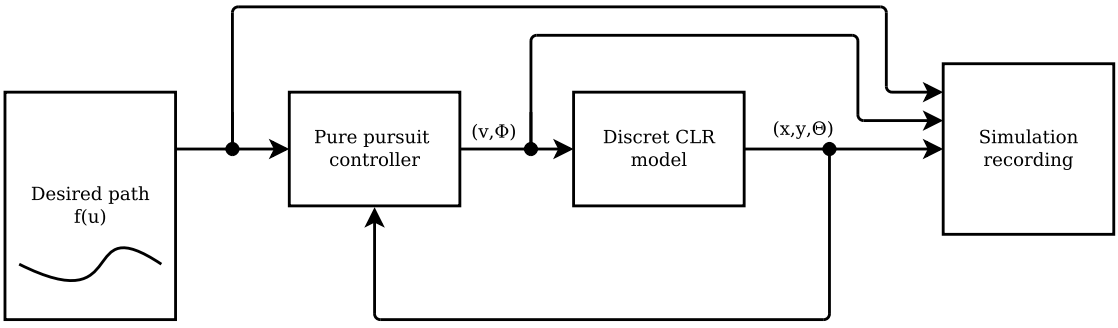


Figure 6.1: Block diagram of the simulation process

6.2 System model

Based on the simplified system model as described in Equation (4.4) of Section 4.2, a time discrete system model can be derived as follows:

$$\begin{aligned} X_{n+1} &= X_n + (v \cdot \Delta t) \cdot \cos(\theta_n + \tfrac{\psi \cdot \Delta t}{2}) \\ Y_{n+1} &= Y_n + (v \cdot \Delta t) \cdot \sin(\theta_n + \tfrac{\psi \cdot \Delta t}{2}) \\ \theta_{n+1} &= \theta_n + (\psi \cdot \Delta t) \end{aligned} \tag{6.1}$$

Note that ψ is defined as:

$$\psi = \frac{v}{L} \tan \phi \tag{6.2}$$

The step size (i.e., Δt) can be chosen arbitrarily. However, if the step size is too large, the simulated system diverges from the continuous system model. Hence, it is possible that the overall system gets instable. A feasible step size for the desired system is about 50 milliseconds. This model is an idealized system model which does not consider any uncertain events or inaccuracies. As real-world reference vehicle, the target system described in Section 5.1.2 is considered. Thus, the presented system model is extended to consider positioning errors, heading errors, and errors of the steering angle .

Positioning errors

Since a high accurate positioning system is used, the positioning error is small in contrast to low cost positioning systems. However, the used GPS-RTK system (cf. Section 5.1.2) has an accuracy of $d = \|p_m - p_{ref}\| = 0,05 - 0,2m$ where p_m represents the measured value and p_{ref} represents the reference value. The accuracy of a positioning system is defined such that the distance of 95% of all measured values are smaller than d. Moreover, it is assumed that the measured values are normal distributed, since the supplier does not provide further information. Remember that a normal distributed random number is defined as $x = \mathcal{N}(\mu, \sigma^2)$, where μ represents the mean and σ^2 represents the variance. Note that the standard deviation is defined as $\sigma = \sqrt{\sigma^2}$. Furthermore, the density function can be defined as $f(x, \mu, \sigma)$. The parameter σ^2 of a normal distributed random number is defined in a way that the distribution function suffices the following equation:

$$F(-d; 0, \sigma^2) = 0,025 \tag{6.3}$$

This claim is equivalent with the following demand based on the density function:

$$\int_{-d}^{d} f(x; 0, \sigma^2) dx = 0.95 \tag{6.4}$$

Since the result can not be computed analytically, tables are used to determine the desired result. Thus, σ is determined as $\sigma \approx 0.5 \cdot d$. Considering these facts, positioning errors can be defined as

$$p_e = \begin{pmatrix} x_e \\ y_e \end{pmatrix} = \begin{pmatrix} x_{ref} \\ y_{ref} \end{pmatrix} + \begin{pmatrix} A \cdot \sin \alpha \\ A \cdot \cos \alpha \end{pmatrix} \tag{6.5}$$

where A is a random variable which represents the distance error and is defined as $A = \mathcal{N}(0, 0.5d)$ and α is a uniform distributed random variable which represents the heading and is defined as $\alpha = \mathcal{U}(0, 2\pi)$.

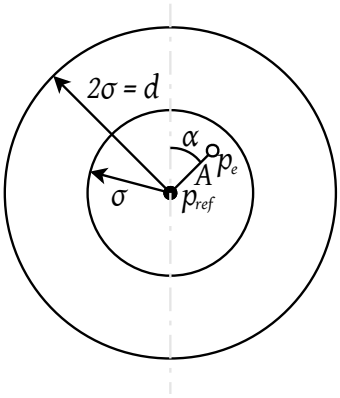


Figure 6.2: Error simulation of a plane point

Heading errors

Since the accuracy of the used electronic compass (cf. Section 5.1.2) is only specified by one value without specifying the error model, it is assumed that this value determines the standard deviation of a normal distributed random number. Thus, the error function of the heading can be specified as

$$\theta_e = \theta_{ref} + \mathcal{N}(0, \frac{\pi}{180}). \tag{6.6}$$

Steering angle errors

In contrast to the previous mentioned errors, this error can not be modeled by a normal distributed or a uniform distributed error function. The error of the steering angle depends on the difference between the last steering angle ϕ_n and the new steering angle ϕ_{n+1}.

6.3 System controller

In the previous chapters of this thesis, only motion planning problems were considered. In general, a controller is necessary to keep the system on the desired path or trajectory. Since this chapter deals with the simulation of a CLR moving along a path, a controller is necessary which keeps the vehicle on the path.

6.3.1 Survey of control approaches

In literature, a lot of approaches for controlling a CLR can be found in [31]. In [32], a survey is presented which informally explains the benefits and drawbacks of a number of control approaches.

Considering control approaches for CLRs, it can be distinguished between two different types of controllers based on the type of their input:

- **Trajectory tracking controller**
  The input of such a controller is a trajectory of the SS. It is possible that a controller

only needs a state trajectory to control the system (e.g., $(x, y, \theta)^T$). Furthermore, it is possible that a controller depends on the whole trajectory of a system i.e., the input- and state trajectory (e.g.$(x, y, \theta, \phi)^T$).

- **Path following controller**
  A path following controller uses only the geometric path instead of a trajectory to control the system. Thus, it considers only the deviation between the desired path and the actual position of the vehicle. Note that a path tracking controller is only a lateral controller. Speed control has to be done by another controller or is even set to a constant value. However, in general a path tracking controller is not independent of the speed of the CLR. Thus, fast changes of speed can destabilize the behavior of the path tracking controller.

The *pure pursuit controller* is one example of a feasible path following controller. It is used within the simulation framework. Details about this controller are described next.

6.3.2 Pure pursuit controller

The pure pursuit controller as introduced in [9, 32], is a very simple and robust *path following controller*. Moreover, it requires only a non-smooth geometric path.

The controller intuitively acts as a human driver. It visualizes a point on the path in a given distance depending on the speed of the vehicle. The steering command is based on the angle between the main axis of the vehicle and the angle of the point w.r.t. the vehicle. The geometric model of the controller is shown in Figure 6.3.

The computation of the control rule requires only one trigonometric function and, thus, it is computational efficient. One challenge of this controller is to track the desired Look-Ahead Point (LAP). Furthermore, the angle η has to be determined based on the position of the vehicle and the LAP. This computation requires also the use of a few trigonometric functions.

To determine the point on the path in front of the vehicle, a geometric function is used which computes the intersection between a line (i.e., a subset of the desired path) and a circle where the radius represents the *look-ahead distance*. This method is described in Section 5.3.2. The approach is feasible if only one intersection is found. In general, a lot of intersections are possible and the point can not be determined exactly. However, this problem can be solved by intersecting only a subset of the path with the circle around the car. If this subset is chosen such that only one intersection occurs, the LAP can be determined exactly.

Figure 6.4 shows the determination of the LAP. Since the simulation is time discrete, a LAP is computed in every step. Thus, the new LAP is in front of the previous LAP w.r.t. the path. Since it is assumed that the vehicle moves with a known and constant speed, the range of the vehicle between two steps can be estimated. Considering these two facts, a subset of the path can be defined such that it starts at the previous LAP. Furthermore, the length of this segment can be determined by the range of the CLR between two steps (including an uncertainty factor). The red line in Figure 6.4 denotes the segment which is intersected with the circle around the CLR.

Once the LAP is determined, the motion rule as defined in Equation 6.7 can be used to compute the steering angle. Figure 6.3 illustrates the meaning of the variables of the motion rule.

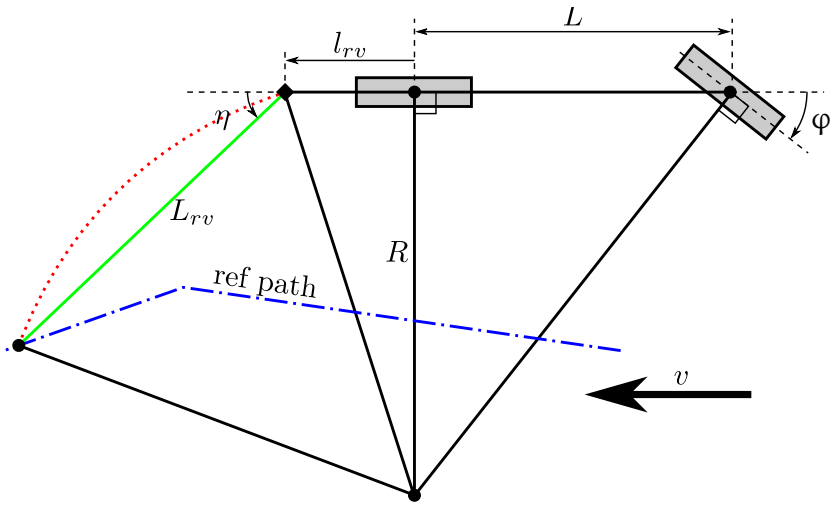


Figure 6.3: Geometric representation of a pure pursuit path tracking controller (taken from [9])

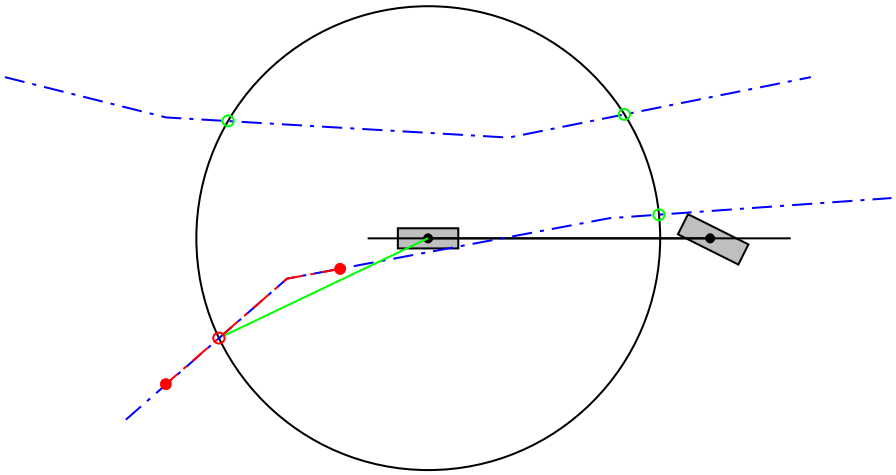


Figure 6.4: Determination of the LAP using the pure pursuit approach

$$\phi = -\arctan\left(\frac{L \sin \eta}{\frac{L_{rv}}{2} + l_{rv} \cos \eta}\right) \tag{6.7}$$

Note that it is not feasible that the controller is called after each simulation step, since the real-world system is not able to change the steering angle within 50 milliseconds. Thus, the steering angle can only be changed in an interval of at least one second. If it is assumed that the step size is 50ms, then the steering angle can be updated after 20 steps.

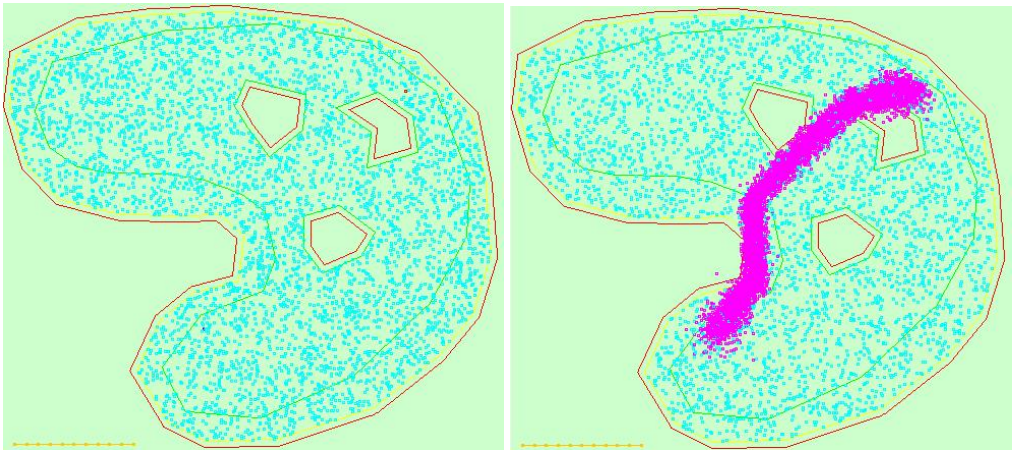
CHAPTER 7

Results

7.1 Path computation results

To evaluate the path computation, an example WS with three obstacles was defined as shown in Figure 7.1. The definition area of this WS is about $350 \times 300m$. To visualize the size of the WS, an orange reference line is drawn in the left bottom corner. The line is exactly 100m long and is divided into 10 segments.

Figure 7.1: Example of a definition area created with the motion planning application



(a) Conventional RRT (b) Hybrid RRT

Figure 7.2: Comparison of the random point generation between the conventional and the new hybrid RRT approach

Since the conventional RRT approach and the hybrid RRT approach were implemented, these two approaches are compared in the following sections. To evaluate the described test series, the test application was executed on a *Sun Java* JVM_x64 with Windows 7 64bit operating system. The hardware platform is based on a dual core CPU (Intel®Core ™2 Duo E8400, 2x3.0GHz, 4GB RAM).

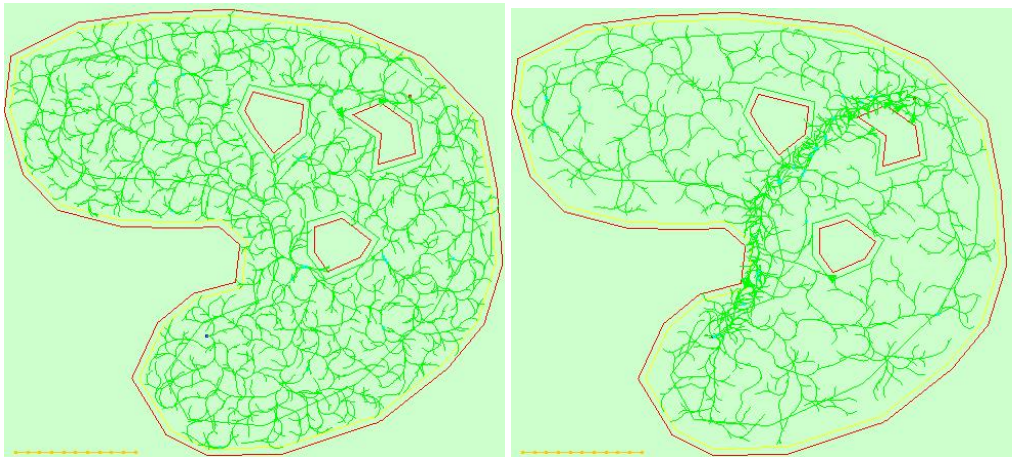
7.1.1 Exploration of the definition area

Random point generation

As described in Section 5.4, the *hybrid RRT* approach despites from the conventional RRT approach in the generation of the random points. The conventional RRT approach distributes 2.000 random points over the whole definition area as shown in Figure 7.2(a). In contrast to the conventional approach, the hybrid approach computes a channel of normally distributed random points in addition to the uniformly distributed random points. Figure 7.2(b) shows an example of a WS with 2.000 random points. Note that two of three points are normally distributed along the channel and the others are uniformly distributed over the whole area. The magenta points represent the random points along the channel (i.e., 1.333 points) and the cyan points illustrate the uniformly distributed random points (i.e., 667 points).

Tree exploration

In the previous paragraph, the result of the random point generation was illustrated. However, this is not sufficient to guarantee a rapidly exploring tree. Thus, in Figure 7.3(a) it is shown that the conventional RRT approach explores the whole definition area except the obstacles. As shown in Figure 7.3(b), the hybrid approach explores the whole definition area area, too. However, it is also possible to continue the tree expansion in order to get shorter paths.



(a) Conventional RRT (b) Hybrid RRT

Figure 7.3: Comparison of the tree exploration between the conventional and the hybrid RRT approach

Note that the trees in both figures contain exactly 2.000 points.Obviously, a path can be found with less than 2.000 nodes but these figures illustrate the coverage of the tree.

7.1.2 Performance and quality of the path computation

To show the quality and the performance of the new hybrid RRT approach, a quantitative analysis of the runtime and the length of the computed path was made. This analysis is based on a fixed WS and fixed initial and goal configurations. To increase the significance of this analysis, 50 executions of the same but independent problem instance were examined. Thereby once the conventional RRT approach was used and once the hybrid RRT approach was taken. Note that the Euclidean distance between the initial and the goal state is $197m$.

Based on this test series, the mean value and the standard deviation are listed in Table 7.1. The *hybrid RRT* approach provides a much shorter path length than the conventional RRT. Another important advantage is that the standard deviation of the length is much smaller using the hybrid approach. This means that the hybrid RRT approach computes paths with constant quality. The hybrid RRT approach has also a better performance than the conventional RRT despite the fact that the hybrid has to compute a guideline in advance. Obviously, the runtime of the algorithm strongly differs for every run using the conventional approach.

To visualize the path computation, Figure 7.4 shows the computation of 50 paths with the same initial and goal configuration. The left figure shows the computation using the conventional approach and the right figure shows the paths computed by the hybrid approach.

7.1.3 Overall path

Based on the implementation of the RRT approach, a path can be computed which covers the whole working area. Figure 7.5 shows the construction phase of a path which covers the green area in the figure. The distance between two lanes in the figure is three meters. Note that for special application scenarios (e.g., mowing) the vehicle is able to lift the tools for a turning maneuver and therefore it is allowed to use the green area for turning maneuvers, too.

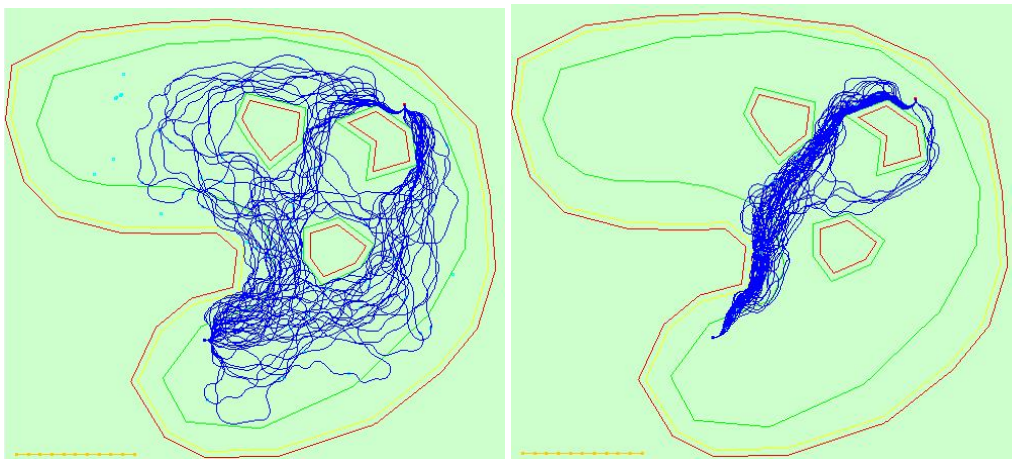
7.2 Simulation results

As described in the previous chapter, a simulation is used to evaluate the feasibility of a path. To evaluate the quality of the path and the robustness of the controller, the distance of the simulated vehicle to the computed path is determined. Since different schemes to measure the distance between the vehicle and the desired path exist, Figure 7.6 illustrates the used method. To visualize the analysis, Figure 7.7 shows the simulated path. In this case, a typical path was simulated which was created by the RRT planner.

The simulated CLR has a length of 3 meters. Furthermore, the speed is 2m/s. The distance to the LAP as described in Section 6.3 was set to $7m$. The step time is 50 milliseconds and the controller updates the steering angle every second. Furthermore, the LAP of the *pure pursuit controller*, is set to 7 meters. However, the quality of the path can possibly be improved by varying this parameter.

To visualize the behavior of the controller, Figure 7.8 shows a diagram of the distance error (i.e., the distance between the desired path and the CLR). In fact, the distance error remains always under $1m$. In addition to the distance error, the slope of the heading is shown in the figure. Thus, it can be observed that the distance error is only high, after the heading of the CLR was changed fast.

Note that the simulation of the overall path reduces the average distance error due to the straight lanes. However, since the positioning and heading errors are also simulated, the distance error can not be eliminated completely.



(a) Conventional RRT (b) Hybrid RRT

Figure 7.4: Comparison of the conventional and the hybrid RRT approach computing 50 paths

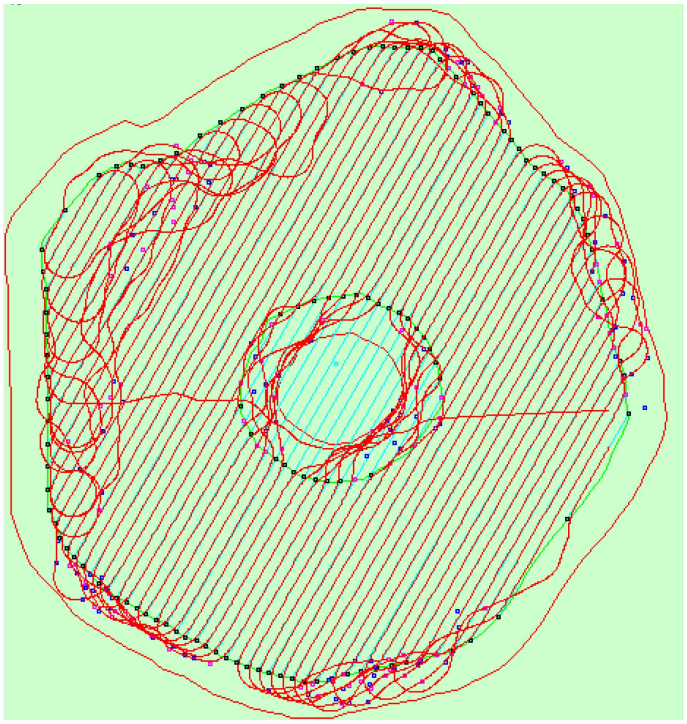


Figure 7.5: Visualization of the overall path covering the green area

*	*	*	Conventional RRT	Hybrid RRT
Duration	Average	ms	9030,86	8465,46
Duration	Standard deviation	ms	1532,59	624,94
Path length	Average	m	426,36	338,02
Path length	Standard deviation	m	66,71	26,65

Table 7.1: Comparison of the conventional RRT approach with the new hybrid RRT approach

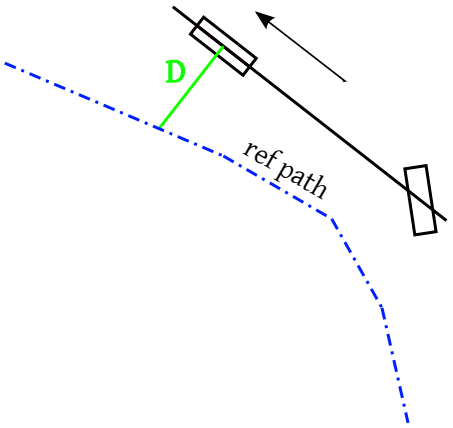


Figure 7.6: Visualization of the distance determination

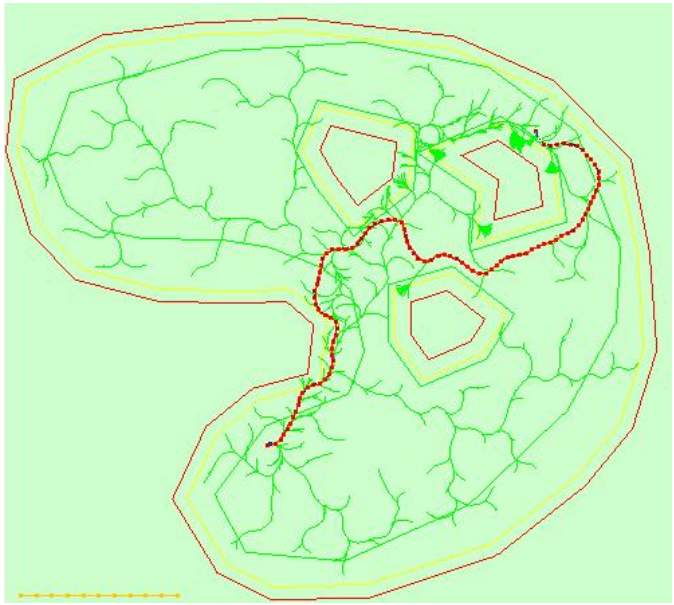


Figure 7.7: Visualization of the simulation process

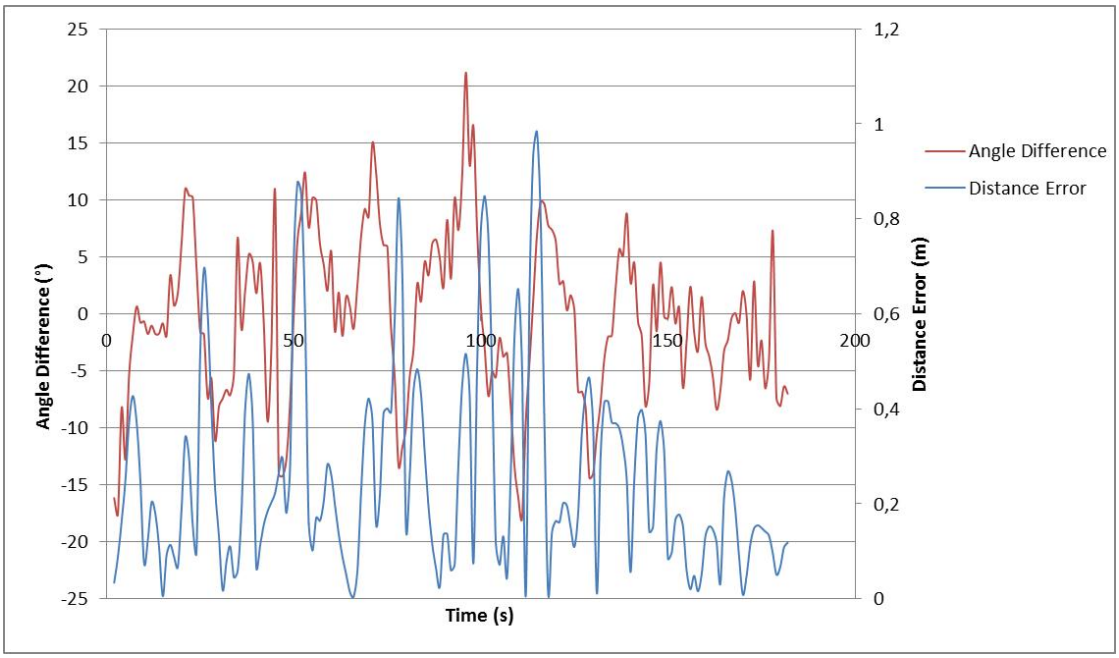


Figure 7.8: Diagram of the distance error depending on the time

CHAPTER 8

Conclusion and outlook

Given the problem statement of the introduction of this thesis, one could assume that this problem can be solved in an easy way. It seems that everything that is necessary is to calculate a few lines, intersect them, and generate a path. However, if motion planning is studied intensively it turns out that motion planning is one of the most challenging tasks in robotics. This is particularly true for finding paths in arbitrary workspaces which may contain arbitrary obstacles. A number of well studied motion planning problems exist but in general no optimal and efficient approach can be given.

Another quite challenging task is motion planning for CLRs. Especially, if an optimal path is claimed, this problem is not solvable in polynomial time. Thus, practical approaches can not be optimal, complete, and fast at the same time. Most of these planning approaches take samples of the WS and use well know graph theoretic methods to solve the problem. Another important group of motion planning algorithms is based on probabilistic methods. These methods provide a good average performance but they are not complete in general.

Since the optimal motion planning problem for CLRs is not solvable in polynomial time, the problem to cover a predefined area by an optimal path is quite harder. If one considers the well known TSP problem, this problem itself is not solvable in polynomial time. If this problem is restricted to curvature constraints path, the problem obviously gets harder.

At the beginning of this thesis, the problem statement was formulated and the basic terms used in this thesis were defined. Summarizing, the problem is to find a feasible path for a CLR which covers a predefined area in parallel lanes. Thus, a method to find a feasible path between two arbitrary points is the basis to find a solution for the given problem.

Before motion planning approaches were examined, basic concepts of positioning systems were introduced. Since this thesis also considers real applications the use of such systems is immanent. As the low cost positioning systems are not sufficient to provide the required position accuracy to control a real vehicle, high precision positioning systems were introduced. In addition to the positioning systems, geodetic models were presented. These models are the basis for all further computations.

Since it is not possible to examine motion planning approaches without considering the system properties, an introduction to system theory was given. This overview includes methods to describe the system behavior and the constraints of the system. To evaluate the performance of a motion planning approach, the lower bounds of a few fundamental motion planning approaches were presented.

Once the basic concepts were introduced, a selection of motion planning approaches was presented. These approaches are based on very different ideas. The simplest approach is the discretization of the CS. Thus, graph theoretic methods can be used to solve the appropriate motion planning problem. An important fact is that if a CS is discretized, no complete solution exists to solve this problem in general. Another intuitive approach is the use of roadmaps. However the challenge is the computation of the roadmap. The search within the roadmap can be implemented efficiently. Since the complexity of motion planning problems is high, probabilistic methods provide alternative concepts to solve hard problem instances. These approaches provide a good average performance but they are not complete in general. In contrast to the conventional approaches, another concept was presented which combines the motion planning algorithm with the system controller. This approach provides a very robust method to control a system.

Based on the introduced motion planning approaches, few of them which can be applied to CLRs were examined. First, a simple system model of a CLR was introduced. It turned out that a CLR has non-holonomic constraints which complicates the motion planning task. A naive approach was given which simply discretizes the CS and uses a graph search algorithm to find a feasible path. Furthermore, roadmap based and probabilistic methods were introduced to find feasible paths for a CLR. Especially one of these approaches called RRT promised a quite good performance. This method is based on the concept of random rapidly exploring trees. In addition to this approach, important improvements which made RRT feasible for practical use were presented and developed in the context of this thesis.

Since the RRT approach promises a good performance, a *proof-of-concept* implementation was performed. To improve the quality and the performance of this implementation, an extension to the basic RRT approach was developed in the context of this thesis. This *hybrid RRT approach* uses an auxiliary path to modify the distribution of the random configurations. The described implementation is able to find a path between two arbitrary configurations in the CS. Furthermore, it is able to find a path which covers a whole area.

To evaluate the results of the system, the behavior of a real CLR was simulated. The simulation model considers positioning and heading errors. Furthermore, it is considered that the steering angle can not be set exactly. To keep the virtual vehicle on the path, a controller was necessary. Thus, a few different controllers were examined. Since the speed of the desired system is low, dynamic constraints can be neglected and therefore a simple *pure pursuit controller* was chosen to control the virtual vehicle.

Finally, the performance of the implemented motion planning approach was analyzed by examining the runtime and the length of the path. The results showed that is possible to compute a feasible high quality path for CLRs in an acceptable time. Furthermore, the simulation results were analyzed. The quality of the simulation was determined by analyzing the distance between the desired path and the position of the virtual vehicle. Results showed that the vehicle is able to follow the path whereas overshooting is limited within acceptable bounds in case of narrow

curves.

In a future step, another motion planning approach (e.g., PRM-based) could be implemented to compare it to the RRT approaches. In contrast to them, PRM-based approaches can be expected to have a quite long preprocessing phase but queries will probably be executed fast in general.

It would also be interesting to implement a feedback motion planning algorithm for CLR. This approach makes the path tracking controller obsolete. However, due to the resource requirement of the necessary algorithm, it will require a high performance computer to achieve realtime capabilities.

It is also possible to realize a complete approach based on Dubins' car especially for smaller WS. If the coverage of the area in parallel lanes is not required, an algorithm based in the curvature constraint TSP can be implemented. However, the use of this approach will only be usable for small workspaces.

Based on the currently implemented application (including the simulation), the basic methods of this application can be ported to a real system. In this case, the real system replaces the simulated system model. However, a real vehicle will require a precise positioning system and low level controller to control the speed of the vehicle and the steering angle. Furthermore, the brake and the gearshift have to be controlled based on the speed set-value.

Bibliography

[1] EUROCONTROL Brussels-Belgium and IfEN University FAF Munich-Germany , *WGS 84 Implementation Manual*, 2.4 ed., 1998.

[2] J. C. Latombe, *Robot Motion Planning*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[3] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys*, vol. 23, no. 3, pp. 345–405, 1991.

[4] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[5] R. Bohlin and L. E. Kavraki, "Path Planning Using Lazy PRM," in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 521–528, 2000.

[6] S. M. L. Valle and J. J. Kuffner, "Randomized kinodynamic planning," *Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[7] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006. Available at http://planning.cs.uiuc.edu/.

[8] P. Felkel and S. Obdrzalek, "Straight skeleton implementation," in *Proceedings of Spring Conference on Computer Graphics*, pp. 210–218, 1998.

[9] Y. Kuwata, J. Teo, S. Karaman, G. Fiore, E. Frazzoli, and J. P. How, "Motion planning in complex environments using closed-loop prediction," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference and Exhibit*.

[10] V. Vonàsek, J. Faigl, T. Krajnìk, and L. Přeučil, "RRT-path - A Guided Rapidly Exploring Random Tree," in *Robot Motion and Control 2009* (K. R. Kozlowski, ed.), Springer, 2009.

[11] S. Kammel, J. Ziegler, B. Pitzer, M. Werling, T. Gindele, D. Jagzent, J. Schröder, M. Thuy, M. Goebl, F. v. Hundelshausen, O. Pink, C. Frese, and C. Stiller, "Team AnnieWAY's autonomous system for the 2007 DARPA Urban Challenge," *Journal of Field Robotics*, vol. 25, no. 9, pp. 615–639, 2008.

[12] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney, "Stanley: The robot that won the DARPA Grand Challenge: Research Articles," *Journal of Field Robotics*, vol. 23, no. 9, pp. 661–692, 2006.

[13] H. Dodel and D. Häupler, *Satellitennavigation*. Springer, 2009.

[14] T. Vincenty, "Direct and inverse solution of geodesics on the ellipsoid with application of nested equations," *Survey Review*, vol. 23, no. 176, pp. 88 – 93, 1975.

[15] H. J. Zimmermann, *Practical applications of fuzzy technologies*, ch. Using fuzzy logic for mobile robot control, pp. 185 – 205. Kluwer Academic Publisher, 1999.

[16] J. H. Reif, "Complexity of the mover's problem and generalizations," *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, pp. 421–427, 1979.

[17] J. Reif and H. Wang, "The complexity of the two dimensional curvature-constrained shortest-path problem," in *Third International Workshop on the Algorithmic Foundations of Robotics*, pp. 49–57, 1998.

[18] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry: algorithms and applications*. Springer, 2008.

[19] L. Kavraki, P. Svestka, J. C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[20] J. J. Kuffner Jr. and S. M. Lavalle, "RRT-Connect: An efficient approach to single-query path planning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 995–1001, 2000.

[21] L. E. Dubins, "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents," *American Journal of Mathematics*, vol. 79, pp. 497–516, 1957.

[22] J. A. Reeds and L. A. Shepp, "Optimal path for a car that goes both forwards and backwards," *Pacific Journal of Mathematics*, vol. 145, no. 2, pp. 367–393, 1990.

[23] M. Pivtoraiko, R. Knepper, and A. Kelly, "Optimal, smooth, nonholonomic mobile robot motion planning in state lattices," tech. rep., Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, 2007.

[24] G. Song and N. M. Amato, "Randomized Motion Planning for Car-like Robots with C-PRM," tech. rep., Department of Computer Science, Texas A&M University, 2001.

[25] Y. Kuwata, G. A. Fiore, J. Teo, E. Frazzoli, and J. P. How, "Motion planning for urban driving using RRT," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1681–1686, 2008.

[26] K. Savla, E. Frazzoli, and F. Bullo, "On the point-to-point and traveling salesperson problems for dubins vehicle," in *Proceedings of the 2005 American Control Conference*, pp. 786–791, 2005.

[27] K. Savla, E. Frazzoli, and F. Bullo, "Traveling Salesperson Problems for the Dubins Vehicle," *IEEE Transactions on Automatic Control*, vol. 53, no. 6, pp. 1378–1391, 2008.

[28] F. J. Gruber and R. Joeckel, "Vermessungstechnische Grundaufgaben," in *Formelsammlung für das Vermessungswesen* , Vieweg+Teubner, 2009.

[29] J. O'Rourke, *Computational Geometry in C*. Cambridge University Press, 1997.

[30] G. T. Toussaint, "A simple linear algorithm for intersecting convex polygons," *The Visual Computer*, vol. 1, no. 2, pp. 118–123, 1985.

[31] A. DeLuca, G. Oriolo, and C. Samson, *Robot Motion Planning*, ch. Feedback Control of a Nonholonomic Car-Like Robot. Springer, 1998.

[32] J. M. Snider, "Automatic steering methods for autonomous automobile path tracking," tech. rep., Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2009.

[33] J. P. Laumond, S. Sekhavat, and F. Lamiraux, *Robot Motion Planning*, ch. Guidelines in Nonholonomic Motion Planning for Mobile Robots. Springer, 1998.

[34] P. Svestka and M. H. Overmars, *Robot Motion Planning*, ch. Probabilistic Path Planning. Springer, 1998.

[35] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," tech. rep., Iowa State University, 1998.

APPENDIX A

Acronyms and abbreviations

AGV Automated Guided Vehicle

CLR Car-Like Robot

CS Configuration Space

DARPA Defense Advanced Research Projects Agency

EGNOS European Geostationary Navigation Overlay Service

DGPS Differential GPS

GNSS Global Navigation Satellite System

GPS Global Positioning System

GUI Graphical User Interface

HLR Human-Like Robot

JVM Java Virtual Machine

LAP Look-Ahead Point

LPRM Lazy Probabilistic Roadmap

MSAS Multi-functional Satellite Augmentation System

ODE Ordinary Differential Equation

PRM Probabilistic Roadmap

PS Phase Space

RRT Rapidly-exploring Random Tree

RSC Reeds and Shepp Car

RTK Real Time Kinematic

SA Selective Availability

SBAS Satellite Based Augmentation System

SS State Space

TSP Traveling Salesman Problem

WAAS Wide Area Augmentation System

WGS84 World Geodetic System 1984

WS Workspace