CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR THESIS

Lukáš Niedoba

Smooth trajectory generation in 2D

Department of Cybernetics

Thesis supervisor: **RNDr. Miroslav Kulich, Ph.D.**

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Lukáš N i e d o b a

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Bachelor Project: Smooth Trajectory Generation in 2D

Guidelines:

1. Get acquainted with the library for generation of Voronoi diagrams in 2D *the boost.polygon Voronoi library* and with the library for vizualization and manipulation of scientific data VTK.
2. Get acquainted with representations of smooth curves in 2D.
3. Implement an algorithm for approximation of the shortest path in Voronoi diagram in a polygonal scene with a smooth curve.
4. Design and implement an algorithm for optimization of an initial trajectory based on effective computation of a distance function.
5. Verify functionality of the implemented algorithms experimentally. Describe the obtained results with respect to the algorithms' speed and robustness as well as quality of the generated outputs.

Bibliography/Sources:

[1] M. Kulich: Vyhlazování cesty pro autonomní vozítko, diplomová práce, MFF UK Praha, 1996.
[2] W.J. Schroeder, K. Martin, W. Lorensen:The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, Third Edition. Kitware, Inc. 2006.
[3] E.G. Gilbert, D. W. Johnson: Distance functions and their application to robot path planning in the presence of obstacles, IEEE Journal of Robotics and Automation, vol.1, no.1, pp.21,30, Mar 1985.
[4] THE BOOST.POLYGON VORONOI LIBRARY, http://www.boost.org/doc/libs/1_55_0/libs/polygon/doc/voronoi_main.htm [online]
[5] Escande, A.; Miossec, S.; Benallegue, M.; Kheddar, A. "A Strictly Convex Hull for Computing Proximity Distances With Continuous Gradients", Robotics, IEEE Transactions on, On page(s): 666 - 678 Volume: 30, Issue: 3, June 2014.

Bachelor Project Supervisor: RNDr. Miroslav Kulich, Ph.D.

Valid until: the end of the summer semester of academic year 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 20, 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Lukáš N i e d o b a

Studijní program: Kybernetika a robotika (bakalářský)

Obor: Robotika

Název tématu: Generování hladkých trajektorií ve 2D

Pokyny pro vypracování:

1. Seznamte se s knihovnou pro generování Voroného diagramů ve 2D *the boost.polygon Voronoi library* a s knihovnou pro vizualizaci a manipulaci s vědeckými daty VTK.
2. Seznamte se s metodami reprezentace hladkých křivek ve 2D.
3. Implementujte algoritmus pro aproximaci nejkratší cesty ve Voroného diagramu dané polygonální scény hladkou křivkou.
4. Navrhněte a implementujte algoritmus pro optimalizaci počáteční trajektorie založený na efektivním výpočtu funkce vzdálenosti od překážek.
5. Funkčnost implementovaných algoritmů ověřte experimenty. Experimentální výsledky popište se zaměřením na rychlost a robustnost algoritmů a kvalitu generovaných výsledků.

Seznam odborné literatury:

[1] M. Kulich: Vyhlazování cesty pro autonomní vozítko, diplomová práce, MFF UK Praha, 1996.
[2] W.J. Schroeder, K. Martin, W. Lorensen:The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, Third Edition. Kitware, Inc. 2006.
[3] E.G. Gilbert, D. W. Johnson: Distance functions and their application to robot path planning in the presence of obstacles, IEEE Journal of Robotics and Automation, vol.1, no.1, pp.21,30, Mar 1985.
[4] THE BOOST.POLYGON VORONOI LIBRARY, http://www.boost.org/doc/libs/1_55_0/libs/polygon/doc/voronoi_main.htm [online]
[5] Escande, A.; Miossec, S.; Benallegue, M.; Kheddar, A. "A Strictly Convex Hull for Computing Proximity Distances With Continuous Gradients", Robotics, IEEE Transactions on, On page(s): 666 - 678 Volume: 30, Issue: 3, June 2014.

Vedoucí bakalářské práce: RNDr. Miroslav Kulich, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 20. 1. 2015

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne............................ ....

Podpis autora práce

Acknowledgements

I would like to thank my supervisor RNDr. Miroslav Kulich, Ph.D. for his guidance and patience. His advices always helped me to choose the right movement and overcome difficulties in this thesis. I would also like to thank my family and friends for their huge support during my studies and solving this thesis.

Abstrakt

Bakalářská práce se zaměřuje na plánování cesty robotického vozítka ve známém prostředí. Plánovaná cesta je určena pro robota aproximovaného bodem, popřípadě kruhem. Cílem je najít efektivní cestu s ohledem na energetickou náročnost a její bezpečnost, což má minimalizovat možnost kolizí s překážkami. Nalezená cesta má být hladká, aby byl zajištěn rychlý a plynulý pohyb robota. Mapa prostředí je jeho rovinným zjednodušením a je reprezentována polygonálně. Hlavním cílem této práce je navrhnout a implementovat algoritmus využívající Voroného diagram pro nalezení a optimalizaci cesty a otestovat jej na několika mapách různého charakteru. Dosažené výsledky jsou prezentovány.

Abstract

This work is focused on the planning trajectory of a robotic vehicle at already known environment. Planned path is assigned to the robot approximated by a point or a circle. The goal is to find the most optimal path at its energy consumption and safetiness to eliminate possible colisions. The path must be smooth to ensure fast and fluent robot movement. For representation of the environment is used the set of polygonal obstacles in the plane. The main goal of this thesis is to design and implement the algorithm taking in advantage the Voronoi diagram for purpose of path finding and its optimization then test it on several maps of different character. Finally, the achieved results are presented.

Contents

1 Introduction **1**

2 Algorithm **3**

2.1 Voronoi diagram 5

2.2 Path planning 7

2.2.1 Simplification 7

2.3 Coons cubic spline 8

2.3.1 Manipulation 9

2.3.2 Intersection with an edge 10

2.4 Optimization 11

2.4.1 Penalty function 12

2.4.2 Energetic function 12

2.4.3 Safety function 12

3 Implementation **17**

3.1 Voronoi diagram 17

3.2 A* search 18

3.2.1 Simplification 19

3.3 B-spline 19

3.4 Penalty function 19

3.5 Optimization 21

3.6 Visualization 22

4 Experiments **23**

4.1 α variation 23

4.2 Tolerance variation 25

4.3 Final paths 27

5 Conclusion **32**

List of Figures

2.1 Process overview 4
2.2 Map divided by Voronoi diagram 6
2.3 Simplification process ilustration, picture taken from [de Konig, 2011] 15
2.4 Coons cubic segment 15

4.1 α scaling 24
4.2 Tolerance impact 26
4.3 Final paths in different environment 28
4.4 Final paths in different environment 2 29
4.5 Final paths in different environment 3 30
4.6 Final paths in different environment 4 31

List of algorithms

1 Whole process 3
2 The A* algorithm 14
3 Ramer-Douglas-Peucker 14
4 Optimization 16
5 Safety function implementation 20

Chapter 1

Introduction

The problem of path planning is widely spread into several domains. From planning the most optimal trajectory of a crane carrying some cargo through planning trajectory of a robotic manipulator in a factory assembling any kind of parts to finding trajectory of a robotic vehicle which is the major goal of this thesis. There are generally offering a lot of possible trajectories connecting requested points, but most of that trajectories hardly fulfils the demanding criteria. These criterias come principally from the energy costs or time estimation minimizing and from inaccuracies produced by the control system, inexact measurements or infiltrated from external influences which could inflict collisions in case of the trajectory leaded closely to the obstacles.

Several techniques dealing with this problem are described in [Escande et al., 2014] and [Gilbert and Johnson, 1985]. Generally it is possible to plan a path in the 3D space representing the real world or a part of some environment. It is known that planning is the NP-HARD problem in general. Due to this fact a number of methods was already invented working with some sort of environment simplification, see [LaValle, 2006]. We could mention two geometric methods using representation of the location as the set of polygonal obstacles in the plane.

First of them called Visibility graph means the endpoints of a polygon‘s edges as nodes in the graph i.e. two nodes are connected with an edge if they can be connected with a straight line that does not intersects obstacles. Then with collaboration of any graph shortest path finding algorithm the path could be planned here. The path is passed through vertices of obstacles and it could cause collisions with them. Also it could be easily imagined that this way will never be the most effective path if we are looking on its energy consumption.

The second widely used approach is to divide the plane with Voronoi diagrams and then plan on the graph built from the Voronoi edges and vertices. It guarantees the maximum distance from the obstacles, but also the path will not be the most optimal at its energy consumption and will have a lot of sharp corners which is not suitable for the nice, fast and fluent robot movement.

The plane can be divided into the grid of equivalent parts and search the shortest path there which would give the smoother and more efficient path to ensure smooth trajectories the grid must be divided fine enough and it brings the big memory consumption.

On the other hand, the path planing method described in [Gilbert and Johnson, 1985] uses the distance functions to compute the distance to every obstacle plus proceeding function of the rotation. This gives the smooth path solution continuous at C^2 and effective to the energy consumption but if the environment is complex with a large number of obstacles it could be hard to compute.

The problem of smooth curves generation belongs to path planning at already known enviroment which is relevant, because planning in an undiscovered enviroment is very different. This thesis works with a planar simplification of some real enviroment so it is the most useful for mobile robots. The approach implemented in this thesis should restrict the evaluation of the distance only to the closest obstacles with taking the Voronoi graph in advantage.

The paper is organised as follows. The theoretical foundations and mathematical theory is presented in Chapter 2. The methods of implementation and its problems are described in Chapter 3. The experiments performed with variation of setting, its comparison and conclusion are in Chapter 4.

Chapter 2

Algorithm

In this chapter the whole process will be described first followed by explanation of the theory underlying each part of the process. The theory (mathematical theory) and idea introduced here profits from the diploma thesis [Kulich, 1996].

The enviroment is described by a plane containing polygonal obstacles. This is all represented by the set of points and line segments and surrounded by a boundary represented the same way. The robot actual position is assigned into the beginning point and the desired location where the path is planned is assigned into the end point. A robot control system requires the smooth[1] C^2 curve expressed as the parametric formula depending on its time.

Algorithm 1: Whole process

input : the set of source segments and points
actual position, desired position

output: path expressed as the parametric formula

1 Build Voronoi diagram (VD);
2 Find the shortest path through VD edges with A* algorithm;
3 Simplify the found path;
4 Create initial B-Spline;
5 Optimize spline;

An overview of the process is described in Algorithm 1. First the plane is partitioned into regions applying Voronoi diagram computational geometry concept onto an input set of elements (line 1). The robot begins searching for the shortest path between the initial and desired position through the graph constructed from Voronoi diagram edges. For that purpose a heuristic A star algorithm is employed (line 2). It can often happen that the generated path contains a lot of stages which could be aproximated without any loose of information by one straight line, so they are merged by a simplification algorithm (line 3). Then the first draft of a path represented

[1] C^2 differentability class i.e. the derivatives f‘ and f“ exist and are continuous.

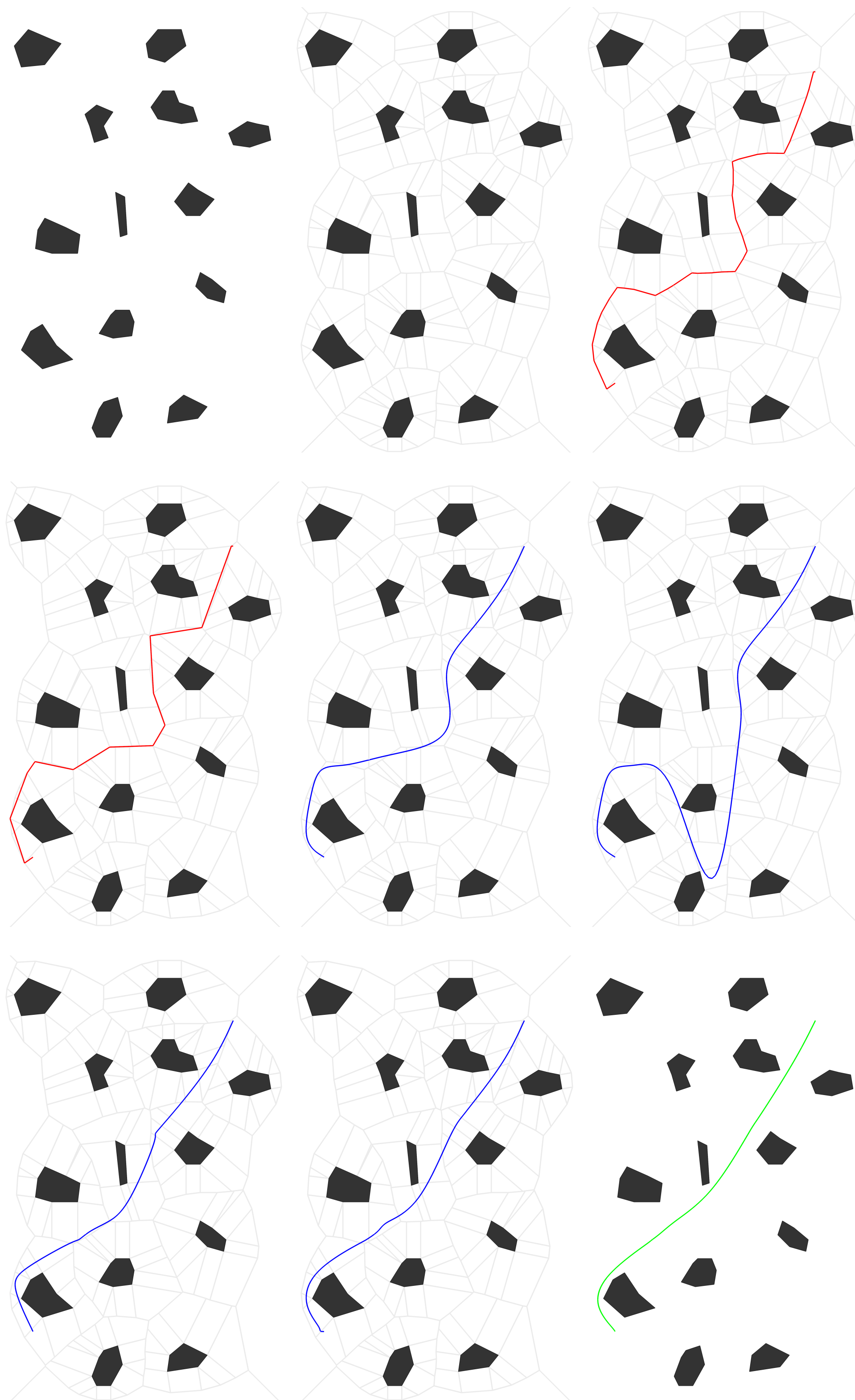

Figure 2.1: Process overview

by a cubic B-spline is generated (line 4). Finally, the drafted spline is optimized with regard to a path length and distance from obstacles (line 5).

The next sections concern about the particular steps of the algorithm parts in detail and are formed in the same order as the algorithm goes. Mathematical formulas and necessary definitions will be introduced together with their theoretical basement.

2.1 Voronoi diagram

Voronoi diagram (VD) presented in [de Berg et al., 1997] is the geometry concept that divides a given space $\mathbb{R}^n$ into parts called **cells**. This division depends on given objects - **source elements** placed into space. Every cell represents a bounded area where all its points are closer to one corresponding input element than to all the others input elements, which is useful for path planning at least for two reasons. If the path is planned through edges of Voronoi diagram, then the found path is at every time in ideal position in consideration to obstacles because the distance to the nearest obstacle is maximal. Moreover if the cell in which the point lays is known, the closest source element is obtained easily. This helps with evaluating a distance to the nearest obstacle. Rigorous definition follows:

Definition 1 *Let $O = \{O_1, O_2, ..., O_n\}$ be the set of source elements laying in $\mathbb{R}^n$. For each two points $o_i, o_j \in O; o_i \neq o_j; i, j \in 1, 2, ..., n$ define*

$$
\begin{aligned}
B(o_i, o_j) &= \{z \in \mathbb{R}^n; d(o_i, z) = d(o_j, z)\} \\
D(o_i, o_j) &= \{z \in \mathbb{R}^n; d(o_i, z) < d(o_j, z)\} \\
D(o_j, o_i) &= \{z \in \mathbb{R}^n; d(o_i, z) > d(o_j, z)\}
\end{aligned}
$$

Then the Voronoi cell $\nu(o_i, O)$ of point o_i in set O is defined as intersection of sets:

$$
\nu(o_i, O) = \bigcap_{o_j \in O \setminus o_i} D(o_i, o_j)
$$

and Voronoi diagram $V(O)$ is defined as union of the boundaries of all the cells in O. Because Voronoi cells are open sets it is:

$$
V(O) = \mathbb{R}^n \setminus \left(\bigcup_{o_i \in O} \nu(o_i, O) \right)
$$

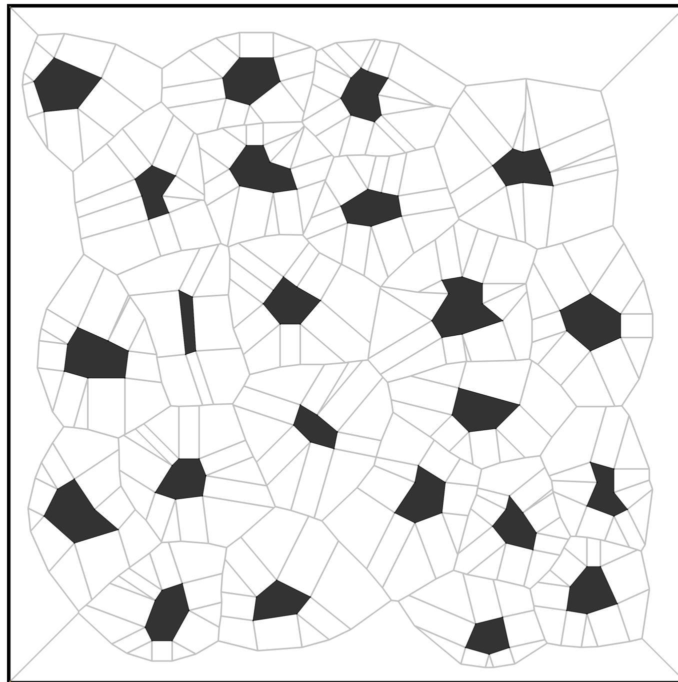

Figure 2.2: Map divided by Voronoi diagram

In this work the VD will be needed only in $\mathbb{R}^2$ plane, so the input elements are in this case lines and points because they are sufficient to represent planar simplification of obstacles when formed as polygons. The Voronoi cells are then also polygons moreover convex polygons. In the following figure 2.2 is shown the example division of the given environment by the VD.

The algoriths ensuring the construction of the VD exist in more varians and they vary in its time complexity or dimensions of the divided space. Two of the algorithms are described below.

Divide and conquer The given set of source elements is recursively divided by a line into two roughly same sized subsets until there is a set with three points where it is easy to determine Voronoi diagram. Then by backtracing the parts of the diagram are merged. The algorithm's running time is $O(n \cdot log(n))$. This method has been presented at [Shamos and Hoey, 1975].

A Sweepline algorithm The sweepline principle sweeps a horizontal line through the plane and notes the temporary bound intersections. The sweepline is a line functioning as the reference element to count a temporary bound with the input elements. This temporary bound is composed of pieces of parabolas. It simply successively builds a Voronoi diagram in the part of the plane between beginning position and actual position of sweepline. More could be found at [Fortune, 1986].

With a rising distance between a source element to the sweepline the corresponding parabola

is gaped more. Voronoi edges arise where two of parabolas intersects and Voronoi vertex arises where at least three of them intersect. The algorithm‘s running time is also $O(n \cdot log(n))$.

2.2 Path planning

Now the plane is divided by a VD presented at previous section and as the initial and desired positions have been obtained at the beginning of the process, it is needed to connect these with the shortest path. Once the plane was divided into Voronoi regions its edges could be used as a graph well covering the whole plane so the path connecting given positions can be searched by a planning algorith on a graph. Though positions mostly don‘t lay on graph edges or vertices, they could be connected with a graph structure by adding an edge between them and their closest vertice in the graph. The fact that this graph well covers the plane comes out from definition of the Voronoi diagram it is obvious that there must be graph edges in the middle of distance between every two source elements.

A star presented in [Delling et al., 2009] is the best algorithm for this purpose as the well known graph algorithm focused on finding optimal ways. This algorithm is based on greedy Dijkstra algorithm with addition of heuristic computation. The search is guided by a heuristic function $h(v)$, which estimates the cost from the vertex v to the goal vertex. In this case, nodes represents vertices, so heuristic function is definned as $h(v) = d(v, goal)$. The algorithm uses two sets datastructures named *openset and* closedset. *Openset stores nodes prepared to be processed and* closedset contains nodes already processed. The algorithm iterates repetitely while the *openset contains some nodes. At the beginning of every iteration the node from the* openset is choosen. This choice of the node v depends on the best value of $f(v) = h(v) + g(v)$ where $g(v)$ is cost from $start$ to the v along the best known path. The pseudocode is introduced in Algorithm 2.

2.2.1 Simplification

Simplification is the process of merging the edges of a polyline which is used to decreace this count of segments then the whole dimension which mainly affects time consumption of that process. In our case this polyline represents a path found by the graph algorithm and for optimization.

Many approaches solving this task were developed and use wide representation of a simplification ratio. For the best suiting point-to-edge distance tolerance it has been choosen the Ramer-Douglas-Peucker algorithm. This algorithm has a worst case running time of $O(n^2)$ but the average running time is $O(n.log(n))$. The principle of this algorithm is that firstly there is only one edge from the beginning to the end point. Then the algorithm counts a distance to each point between these points. The shortest perpendicular distance from point to the edge is meant by distance. The furthest point which has bigger distance than the tolerance is used as a division point of this edge. All points having a smaller distance than defined tolerance are

thrown, so this causes that both corresponding edges of the thrown point are merged to one connecting their outlying endpoints. So now there are two edges the both beginning at a new division point and each of them ends at one of the endpoints. If there are some control points remained between endpoints, whole algorithm is called recursively to those edges.

2.3 Coons cubic spline

Spline is a polynomial function piecewise definned. The cubic B-spline will be presented at this section, coordinates of the points laying on the curve are counted from polynomials defined here and its piece shape unequivocally outgoes from position of points figurating at polynomials and called **control points**. The parts of the curve are called **segments** and are connected and forms whole curve.

Splines are widely used and the best known from the computer graphics. For the final smooth path representing it is the best way of representation at least because Coons cubic spline is C^2 continuous function. Next advantage is low memory consumption because for storing a whole curve only several points called control points are needed. There are always exactly four points for one segment of a spline and by adding a next point a new segment arises, which is continuously connected to the previous one. Every change of control points is local, it means that a change affects only segments defined by these points so maximally at both directions of curve. Every segment lays within a convex set defined by its corresponding control points and thus the whole curve lays within a convex set determined by the whole set of the control points. The definition of the Coons cubic spline presented in [Shikin and Plis, 1995] will be written in the following part.

Definition 2 *Let the $\vec{P} = \{P_0,\ P_1,\ ...,\ P_n\}$ be $(n+1)$ points in $\mathbb{R}^2; n \geq 3$. Then the Coons cubic spline segment is*

$$S_i(\vec{P}, t) = \frac{1}{6}\sum_{j=0}^{3} P_{j+i}C_j(t), \qquad for\ t \in \ \langle 0,\ 1\rangle\,, \tag{2.1}$$

where P_i are above definned Coons cubic control points and C_i are Coons cubic polynomials

$$\begin{aligned}
C_0\,(t) &= (1-t)^3\,,\\
C_1(t) &= 3t^3 - 6t^2 + 4,\\
C_2(t) &= -3t^3 + 3t^2 + 3t + 1,\\
C_3(t) &= t^3.
\end{aligned}$$

The whole spline curve is definned as:

$$C(\vec{P},t) = S_{\lfloor t \rfloor}\left(\vec{P},\ t - \lfloor t \rfloor\right); \qquad t \in \langle 0,\ n-2 \rangle, where\ \lfloor \cdot \rfloor\ isthefloorfunction$$

So once a spline is created i.e. its control points are generated it is easy to determine an arbitary point on the curve by simply specifying the desired time t into Equation 2.1 . This also gives the possibility of the representation a path curve spline as the vector depending on the time.

2.3.1 Manipulation

The spline curve will be used for the smooth aproximation of the path represented as the sequence of the line segments found by the A star algorithm and simplified. In this section the formulas helpful for the approximation will be introduced. Approximation is the initial shape of a smooth path which will be at the next step formed into a final shape by moving the control points.

The first step is to place the beginning and the endpoint of the path curve at its position. By instating additional poins from the time interval $\langle 0,\ 1 \rangle$ (where the Coons segment is definned) into Equation 2.2 statements which will be helpful when building an initial draft spline are obtained. By placing first three control points on the same coordinates it is possible to initiate the whole curve here, the same idea is used for the endpoint.

$$S_i(0) = \frac{P_0 + 4P_1 + P_2}{6}$$
$$S_i(1) = \frac{P_1 + 4P_2 + P_3}{6}$$

The aproximation of the polygonal path is done by placing control points near the middle of each line segment alternately to its left and right side.

When the spline is shaped at the next step, which involves moving control points, it is necessary to hold the beginning and the endpoint of the spline at the same coordinates. So all the control points could be moved except the ultra points, their coordinates are counted from following statments for holding the beginning and endpoind on the same position.

$$P_0 = \frac{6S_0(0) - 4P_1 - P_2}{6}$$
$$P_n = \frac{6S_n(1) - 4P_{n-1} - P_{n-2}}{6}$$

2.3.2 Intersection with an edge

In the next step - shaping the curve, it will be necessary to know the Voronoi cells which the curve passes through. Knowing the intersection points of the curve with the Voronoi diagram edges will be helpful in findin such a cells. So in this section we will concern about theory of computing the intersections of a coons segment with a line segment (edge).

The mathematical parametrical representation of the edge is:

$$\begin{aligned} E_i : x &= X + s.dx; \qquad s \in \langle 0,\ 1 \rangle\,, \\ y &= Y + s.dy. \end{aligned} \tag{2.2}$$

Then we got the system of equations by instating the Equation 2.1 into $x,\ y$ from the Equation 2.2

$$\begin{aligned} S_{i,x}\left(\vec{P}, t\right) &= X + s.dx; \qquad t, s \in \langle 0,\ 1 \rangle\,, \\ S_{i,y}\left(\vec{P}, t\right) &= Y + s.dy. \end{aligned} \tag{2.3}$$

The intersection of a cubic spline part with a line segment could be solved as a cubic equation with Cardans formulas after the following adjustment. By substituting to the Equation 2.3 we get four coefficients of cubic formula equations

$$at^3 + bt^2 + ct + d = 0,$$

where

$$\begin{aligned} a &= \left(3\left(P_{2,x} - P_{1,x}\right) + P_{0,x} - P_{3,x}\right)dy - \left(3(P_{2,y} - P_{1,y}) + P_{0,y} - P_{3,y}\right)dx \\ b &= 3((2P_{1,x} - P_{0,x} - P_{2,x})dy - (2P_{1,y} - P_{0,y} - P_{2,y})dx) \\ c &= 3((P_{0,x} - P_{2,x})dy - (P_{0,y} - P_{2,y})dx) \\ d &= (6X - P_{2,x} - P_{0,x} - 4P_{1,x})dy - (6Y - P_{2,y} - P_{0,y} - 4P_{1,y})dx \end{aligned}$$

Solving this equation generally gives six roots, where we are interested only at real roots

$$t_1, t_2, t_3 \leftarrow Real\{t_1, t_2, t_3, t_4, t_5, t_6\}$$

At first, we can throw all solutions where time doesn't lay in the interval $\langle 0, 1\rangle$. Than it is needed to check whether s also lays in this interval. So by adding together both lines of formula 2.3 and adjusting them for getting s we got

$$s = \frac{C_x\left(\vec{P}, t\right) + C_y\left(\vec{P}, t\right) - (X + Y)}{dx + dy}$$

The problem occours with dividing by zero when $dx \stackrel{\varepsilon}{=} -dy$.[4] In this case s will be checked just for one dimension of coordinates according to the following formula which comes also from the Equation 2.3.

$$s = \frac{C_x\left(\vec{P}, t\right) - X}{dx}$$

Due to the fact that we obtained solutions which even didn't lie on an intersectioned edge, it turned out there were still some numerical inacuracies. This issue was fully fixed by matching coordinates obtained from Coons cubic definition with coordinates obtained by the formula adjusted from the parametric formula of an edge and testing its equality.

$$\begin{aligned} C_x\left(\vec{P}, t\right) &\stackrel{\varepsilon}{=} X + dx.s \\ C_y\left(\vec{P}, t\right) &\stackrel{\varepsilon}{=} Y + dy.s \end{aligned} \tag{2.4}$$

2.4 Optimization

Generally, optimization aims to find such a set of input variables for which some cost function is evaluated as minimal or maximal and this could be well fitted for our current problem. Our goal is to find the shape of a spline fullfiling requirements for safety of path and its shortest path. These requirements are already defined by some mathematical formula called **Penalty function** which gives the best solution exactly at the minima. Penalty function will be more specifically shown in 2.4.1. The input are spline control points coordinates which represent input variables.

Spline shape is based totally on n control points. As it was described at 2.3.1 the first and last of them must be reserved for placing endpoints to correct coordinates. Now there are $n-2$ points entering into the optimization process. The number of points multiplied by the number of coordinates at plane gives dimension of optimization $2 * (n-2)$

[4] By equivalency relation here it is meant $\left(x \stackrel{\varepsilon}{=} y\right) \iff ((x + \varepsilon >= y)\ and\ (x - \varepsilon <= y))$ This epsilon equivalence is because of numerical inaccuracies

2.4.1 Penalty function

The penalty function gives global information about safetyness and energy consumption through the whole curve. Given these requirements the function could be also divided into two subfunctions each of them dealing with one problem: Safetyness as $F_S(C)$ and the function concerned at energy consumption $F_E(C)$. Finally both these functions will be summed up each of them multiplied by a coeficient definning its relevance. It is also necessary that the penalty function and all its subfunctions are continuous at all points where the function is definned. So the penalty function is:

$$F_P\left(C\left(\vec{P},t\right)\right) = \alpha F_S\left(C\left(\vec{P},t\right)\right) + (1-\alpha)\,F_E\left(C\left(\vec{P},t\right)\right) \qquad ;\alpha \in \langle 0,\ 1\rangle \tag{2.5}$$

2.4.2 Energetic function

Energy consumption is unequivocally linked with a spline length so the energetic function is defined as the sum of lenghts of the particular coons cubics:

$$F_E\left(C\left(\vec{P},t\right)\right) = \sum_{i=0}^{n} \int_0^1 \sqrt{\left(\frac{\partial S_{i,x}(t)}{\partial t}\right)^2 + \left(\frac{\partial S_{i,y}(t)}{t}\right)^2} dt \tag{2.6}$$

2.4.3 Safety function

The safety function is based on evaluating a distance from obstacles to a **current point** on the path curve. As the plane was divided into Voronoi cells the distance is evaluated always to the source element of an actual cell - i.e. the cell in which the current point lays. There are two types of elements corresponding to each cell. The first type is an edge of an obstacle and the second one is a point - i.e. a vertex of a polygon describing an obstacle corner. So it is needed to split these situations and count by different formulas which are the following:

Distance from a point This formula gives a distance from the obstacle point to the point laying on a Coons cubic segment at time t.

$$d\left(C\left(\vec{P},t\right),O_i\right) = \sqrt{\left(C_x\left(\vec{P},t\right) - O_{i,x}\right)^2 + \left(C_y\left(\vec{P},t\right) - O_{i,y}\right)^2} \tag{2.7}$$

Distance from an edge This formula gives the shortest perpendicular distance from the parametrically expressed edge to the point laying on the Coons cubic segment at a time t.

$$d(C(\vec{P},t),O_i) = \sqrt{\overline{dy}C_x\left(\vec{P},t\right) - \overline{dx}C_y\left(\vec{P},t\right) + \overline{dx}Y - \overline{dy}X},$$

$$where$$

$$\overline{dx} = \frac{dx}{\sqrt{dx^2+dy^2}}$$

$$\overline{dy} = \frac{dy}{\sqrt{dx^2+dy^2}}$$

χ **function** The aim of this function is to specify some zone arround obstacles, where a value given by a distance will be several times increased compared to a distant space. Finally it should ensure that the curve is repelled away from this zone and at other space out of this zone will not be restricted to shaping as it is needed for reaching the shortest length of the whole curve path. So according to these considerations development of the function should follow high outcome for small values and then rapidly descent to give multiple time lower outcome. Four variants of this function were laboured each with slightly different course.

$$\chi_1(x) = \begin{cases} \frac{20000}{1+x+offset} & \text{if } x < offset \\ 20000 & \text{if } x >= offset \end{cases} \tag{2.8}$$

$$\chi_2(x) = \frac{20000}{(1+x)} \tag{2.9}$$

$$\chi_3(x) = \frac{10000}{(1+x)^2} \tag{2.10}$$

$$\chi_4(x) = \frac{10000}{1.5^{(x+1)}} \tag{2.11}$$

Intersections Let the $\tau_0,\ \tau_1,\ ...,\tau_n;\quad \tau_0 < \tau_1 < \ ... \ < \tau_n$ be the ascending sequence of intersections times of a cubic spline with Voronoi edges. Function $S_{rc}(\tau_i)$ gives an source element $o \in O$ linked to a cell containing points $C(\vec{P},t_p);\ t_p \in \langle \tau_i,\ \tau_{i+1} \rangle$

Finally all parts needed to build the safety function are defined. So the safety function is defined as an integral through every part of the spline each time delimited by a pair of intersections times:

$$F_s(C(\vec{P},t)) = \sum_{i=0}^{n} \int_{\tau_i}^{\tau_{i+1}} \chi\left(d(C(\vec{P},t),S_{rc}(\tau_i))\right) dt \tag{2.12}$$

Algorithm 2: The A* algorithm

input : $start$ - actual position, $goal$ - desired position, graph
output: shortest path from the $start$ to the $goal$ represented as the sequence of line segments

1 $closedset \leftarrow \{\}$;
2 $openset \leftarrow \{start\}$;
3 $g_val(start)$;
4 $f_val(start) = g_val(start) + h_val(start,\ goal)$;
5 **while** $openset$ *is not empty* **do**
6 $X \leftarrow$ the node from openset having lowest f_val;
7 **if** $X = goal$ **then**
8 R
9 eturn backtrack path;
10 $openset = openset \setminus X$;
11 $closedset = closedset \cup X$;
12 **for** *each* $n \leftarrow$ *neighbor node of* x **do**
13 **if** $closedset$ *contains* n **then**
14 continue;
15 $tmp_g_val = g_val(x) + |x - n|$;
16 **if** (*openset not contains*) *or* $(tmp_g_val < g_val(n))$ **then**
17 $g_val(n) = tmp_g_val$;
18 $f_val(n) = g_val(n) + h_val(n,\ goal)$;
19 Return error - this path doesn't exist;

Algorithm 3: Ramer-Douglas-Peucker

input : $PointList[]$ - the sequence of points along the path, $tolerance$
output: Simplified $PointList[]$

1 $edge \leftarrow$ Create edge from PointList[first] and PointList[last];
2 throw all points which are closer to the $edge$ than $tolerance$;
3 $furthestPointIndex \leftarrow$ at PointList[] find index of furthest point from $edge$;
4 **if** *Some points remained between PointList[first] and PointList[furthestPointIndex]* **then**
5 CALL DouglasPuecker($PointList[first, .., furthestPointIndex]], tolerance$);
6 **if** *Some points remained between PointList[furthestPointIndex] and PointList[last]* **then**
7 CALL DouglasPuecker($PointList[furthestPointIndex], .., last], tolerance$);
8 Return PointList[];

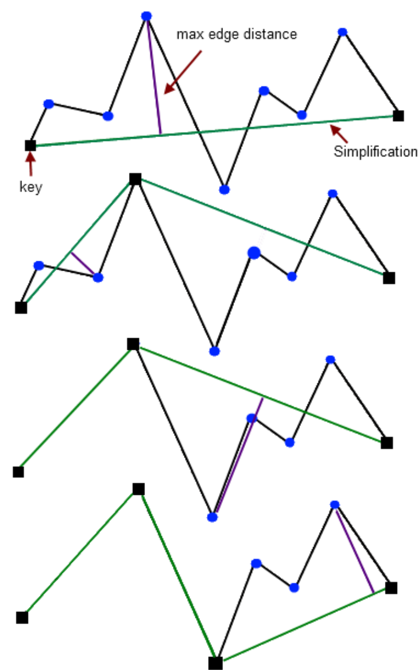

Figure 2.3: Simplification process ilustration, picture taken from [de Konig, 2011]

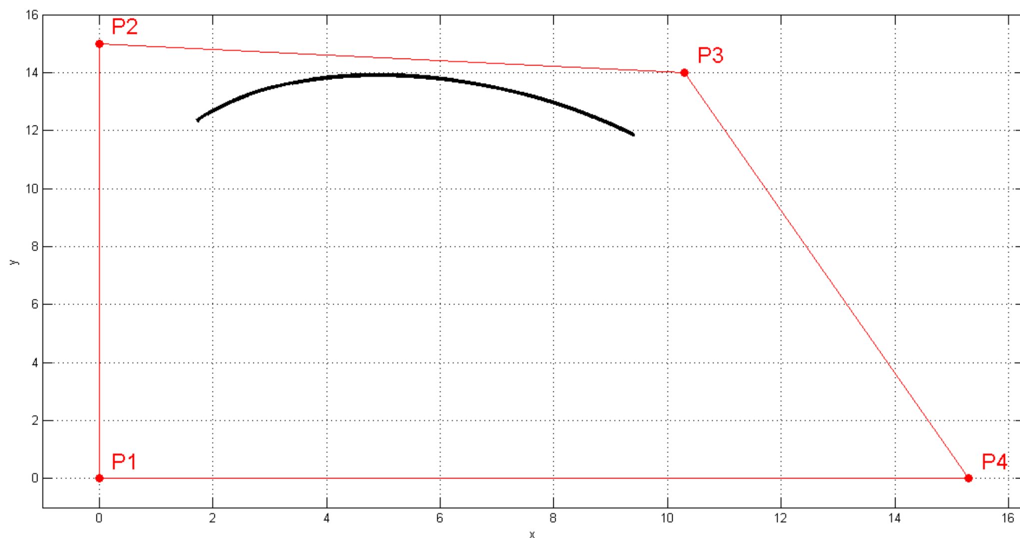

Figure 2.4: Coons cubic segment

Algorithm 4: Optimization

input : initial $SplinePointsSet$
output: optimized $SplinePointsSet$

1 **repeat**
2 Determine how to generate new set of control points;
3 $SplinePointsSet \leftarrow$ Generate new set of control points;
4 $length \leftarrow$ Measure spline;
5 Find intersections;
6 $safetyVal \leftarrow$ Evaluate safety function integral;
7 $optFunction \leftarrow \alpha * safetyVal + \beta * length$;
8 **until** $optFunction$ *is not minima*;
9 Return $SplinePointsSet$;

Chapter 3

Implementation

The main goal of the process of smooth curve generation is to implement a library in C++ programming language on a Linux system. The whole process is visualised thanks to `Visualisation Toolkit library`. If there is not need for visualisation, library should work separately only assuming input data and returning a processed result. C++ was chosen for both because of demanding calculations which this process requires and due to the fact that this kind of robots are usually built from modules controlled by some computer or embedded device running on Linux sytem which is very adaptable and it makes it the most suitable for mobile robots usage purposes.

3.1 Voronoi diagram

The free open-source `Boost C++ library` was chosen for VD creation and processing implementation. This set of libraries could be used at a wide range of C++ application domains for example it provides support and structures for tasks as multithreading, image processing, geometry, linear algebra, unit testing, graph theory, various system routines and variety of math or numeric problems.

VD algorithms belongs to the `Polygon` part of `Boost library` providing algorithms focused on manipulation planar geometry data. It implements the generic sweepline algorithm together with an interface to construct VD of points and line segments. It must be just met the limitations of input elements coordinations represented by integer type and input line segments should not overlap except their endpoints. The library fully supports metaprograming based input so it is possible to work with custom defined input data types.

When the source elements are prepared they are passed to Voronoi creation through iterator interfaces. The algorithm creates a structure of VD allowing to iterate through its elements and connects corresponding(related) components by pointers.

A Voronoi vertex represents a point of VD. The Voronoi output point structure stores vertex coordinates and pointers to the connected edges. It gives us methods to access and iterate

through incident edges arround this point.

Voronoi edge represents the bounds of Voronoi Cell i.e. points equidistant to the two closest source elements. Boost implements this structure as half-edge data structure more could be found at [McGuire, 2000]. The structure implements methods for the acces to related vertices and other edges with a common cell or a connected half-edge, also provides methods determining the type of the edge due to it could be a standard line segment, curved as a parabolic arc or infinite which occours at the borders of the input map. Edges whose endpoint lays on the source element are called primary edges, testing whether an edge is primary can be done by provided methods.

A Voronoi cell represents a VD region bounded by the Voronoi edges. As it outcomes from the definition of VD the source element could be a point or a segment, the structure ensures the methods for determining which type of the source is and the attached source element could be accessed by obtaining a unique ID from the provided method. The boundary edges could be accessed by obtaining a pointer to one of them followed by iteration through them.

The algorithm doesn‘t resolve whether the diagram is created outside obstacles or within any obstacle it simply creates VD through the whole given plane. This situation is needed to identify and then to filter the vertices inside the obstacles.

The main connection is estabilished after the creation of VD which allows to easily connect Boost voronoi output graph structure with its copies or variants through the rest of the algorithms or visualisation structures. A unique ID to each element is given determining its position at its container which could be paired through associative array with any structure. Once if we are able to get voronoi element it is possible to delegate any of its above mentioned built in methods. This gives us ability to work with diagram with no limitations and no need of copying all its traits and links to the other structures.

3.2 A* search

Boost well served also for implementing the A* algorithm by supported graphs structures and A* finding through them belonging to the Boost Graph part of library.

The structure of the graph is different to the VD output structure so it is converted by iterating through Voroni edges and building the weighted graph. Weights had been set to the value of a distance between two its endpoints. Only edges whose endpoint donesn‘t lay on the source element have been added.

The integer IDs of the beginning and end point are passed to the algorithm then when the way has been succesfully found it is returned as the sequence of points IDs along the found shortest path.

3.2.1 Simplification

The polyline simplification algorithm has been implemented thanks to the `psimpl` library. It is a lightweight header-only type library. `psimpl` implements all algorithms using templates. It is posible to use the library at any dimension at space with defined floating point or signed data types.

Input of the algorithm is the sequence of points IDs along the path found by A*. The implementation of the simplification algorithm with `psimpl` required to create serialized linear points coordinates sequence. The same kind of sequence outcomes from the simplification algorithm.

3.3 B-spline

The B-spline curve structure has been implemented as the class working with the set of control points stored in a container. The methods for obtaining a point laying on the path at specific time or solving the intersections have been implemented straight forwardly according the theoretical basis denoted at chapter 2.3.

The whole access to the coonse spline has been implemented from the input side with methods allowing adding or modyfying the coordinates of the control points and from the output side by the method returning coordinates of the point laying on the curve at a given point.

The implementation of the path measuring has been implemented within the framework of the Coonse spline itself. For optimization evaluation there is a need to evaluate a length of the whole curve with every iteration. It is inapropriate to count the length analyticaly because this leads to eliptic integrals. The length could be estimated with an aproximating curve with several segments and summing up their length using euclidean metric to theirs endpoints. This approximation could be also used for visualization purposes. Since the method for obtaining a point at the time has been implemented, it is easy to divide time interval of the path, create a segmential path approximation by connecting adjacent points obtained from the division and summing up the length of all that segments.

3.4 Penalty function

The penalty function as was said at the chapter 2.4.1 consists of the path-length part and safety function part. Path length implementation has been done described in the 3.3. The safety function must pass through whole path and compute the integral as was written at the chapter 2.4.3 this process has been divided into two subprocesses:

- The algorithm dealing with division of the path and assigning these parts to the corresponding Voronoi cell where they belongs to.

- Once the part is specified it could be worked out the value of its safety integral and this is what the second separated process do.

The process of the curve division is described at this paragraph and following pseudocode. It is needed to determine to which Voroni cell the beginning point of the path belongs at first then it is computed the intersections with each the line segment bounding the cell. If there were any intersections, the one of them having laying at the lowest time on the path is used to switch the actual cell through intersected edge to the next cell. If there wasn't any intersection the intersections are counted again with the same cell but the next segment of the path curve. Every time before the curve segment is switched or the actual cell has been moved the integral is counted at this part and summed up to the others. This method is repetively applied since the whole path is passed.

Algorithm 5: Safety function implementation

input : path stored as $CoonsCubic$, VD
output: $SafetySum$

1. $actualCell \leftarrow$ Determine the cell where the point $CoonsCubic(0)$ lays;
2. $segments \leftarrow CoonsCubic.numberOfSegments$;
3. $lastIntersectionTime = 0$;
4. **for** $c = 0; c < segments; switchCell?c = c : (c+1)$ **do**
5. $\quad switchCell \leftarrow$ **false**;
6. $\quad$ **for** $i = 0; (i < actualCell.size)$ **and** $switchCell; i = i+1$ **do**
7. $\quad\quad$ **if** *any intersection with* $actualCell.source$ **then**
8. $\quad\quad\quad SafetySum = SafetySum + penalty$;
9. $\quad\quad\quad$ BREAK Safety Function evaluation;
10. $\quad\quad intersectionTimesTmp \leftarrow$ find intersections with $actualCell.segment[i]$;
11. $\quad\quad$ **if** *any intersections at* $intersectionTimesTmp$ **then**
12. $\quad\quad\quad$ **for** *each* $intrsctn$ *from* $intersectionTimesTmp$ **do**
13. $\quad\quad\quad\quad$ if($CoonsCubic(intrsctn - \epsilon)$. lays inside $actualCell$)**and**($CoonsCubic(intrsctn + \epsilon)$. lays outside $actualCell$) $SafetySum = SafetySum + SafetyIntegral(lastIntersectionTime, intrsctn, actualCell)$;
14. $\quad\quad\quad\quad actualCell \leftarrow$ switch cell through the intersected segment;
15. $\quad\quad\quad\quad switchCell \leftarrow$ **true**;

As the calculation of the intersections brings the inaccuracies with itself, it could happen that the path passes through a segment nearby its endpoint common to one or more neighboring segments and a cell is switched to the incorrect one. This situation has been treated with a correction mechanism based on a simple check after every switch of the cell. It is verified whether the point laying closely to the intersection point actually lays in the new switched

actual cell. If not, the new actual cell is searched into group of cells laying arround the actual misdetermined cell by the same kind of check. Then the actual cell is updated again.

The integral as it has been said at the section 2.4.1 couldn't be counted analytically so it has been counted numerically with a solid step which could be configured because it has huge impact on whole algorithm speed and its outcome. This impact will be compared and discusset in chapter three.

3.5 Optimization

Algorithms from the `OPT++ library` for the optimization purposes were utilized. `OPT++` is a library written in `C++` dealing with nonlinear optimization. The library provides the Newton method, a nonlinear inferior-point method, parallel direct search a trust region - parallel direct search hybrid and a generation set search. Most of these methods can be solved as constrained or not constrained problems, with or without analytic gradients.

We have choosen to use unconstrained parallel direct search and generation set search methods for the path optimization, these methods no need derivative information. Both of them are based on standard Direct search method. The parallel direct search could be easily extended for processing on multiply processors, which is intended to implement in future.

The both optimization methods only need to specify a dimension of the optimized problem, give the initial situation and provide the function evaluating the given situation of generated optimized variables accessible at every iteration of the optimization process. The dimension of the optimization in this case is the number of control points multiplied by constant 2 because points lays at the plane. The initial situation of the variables is when created permisible B-spline path. By permissible here it is meant that spline doesn't intersects any of obstacles, if the spline would cross any of obstacles the function will be evaluated by a high value and then the optimization hardly finds the minima. The evaluation of the generated states is ensured by a given penalty function where the path is set to the one created from the set of coordinates generated by the optimization algorithm at its iteration. As the evaluation function the penalty function described at section 3.4 has been given.

The optimization algorithm implementation provides several parameters affecting the process:

- The number of dimensions of optimized problema.
- Maximal allowed value given by the penalty function.
- The tolerance of the change of evaluation function between iterations before algorithm stops.
- Maximal number of algorithm iterations.

3.6 Visualization

VTK - The Visualization toolkit is an open-source, object oriented, freely available software system for 3D computer graphics, visualization and image processing. VTK consists of a C++ class library. VTK methods are also available wrapped by several interpreted languages such as Java, Python or Tcl and could be runned by them. VTK is multiplatform, could be runned on Unix platforms, Linux, Mac and Windows.

Chapter 4

Experiments

4.1 α variation

In this section the α parameter determinating between safety and length of the path will be varied for showing its affection. The higher the alpha parameter is the more the safetiness of the path should be dominant over its length(energy consumption). Several paths with fixed beginning and endpoint was generated with changing this parameter.

Parameter was choosen at the sequence:

$$\alpha = \left\{ \frac{1}{200}, \frac{2}{200}, \frac{4}{200}, \frac{16}{200}, \frac{32}{200}, \frac{64}{200}, \frac{96}{200}, \frac{128}{200}, \frac{150}{200}, \frac{180}{200}, \frac{185}{200} \right\}$$

It could be observed in Figure 4.1 in color spectrum changing from blue to red.

In the Figure 4.1(left) can be observed how the path with low α (low safetiness) approaches to the obstacles and with incrasing its value the path is getting larger spacing from the obstacles. In the Figure 4.1(right) was moved the endpoint. Here for higher values of α the path durning optimization jumps besides where the polygonal path wasn't even passed through. It happens, because durning the optimization process the shape of curve oscilates wide and when the safetiness requirement is more strict the if there are wider spaces between obstacles, safety function gets lower evaluation and minimum is found here.

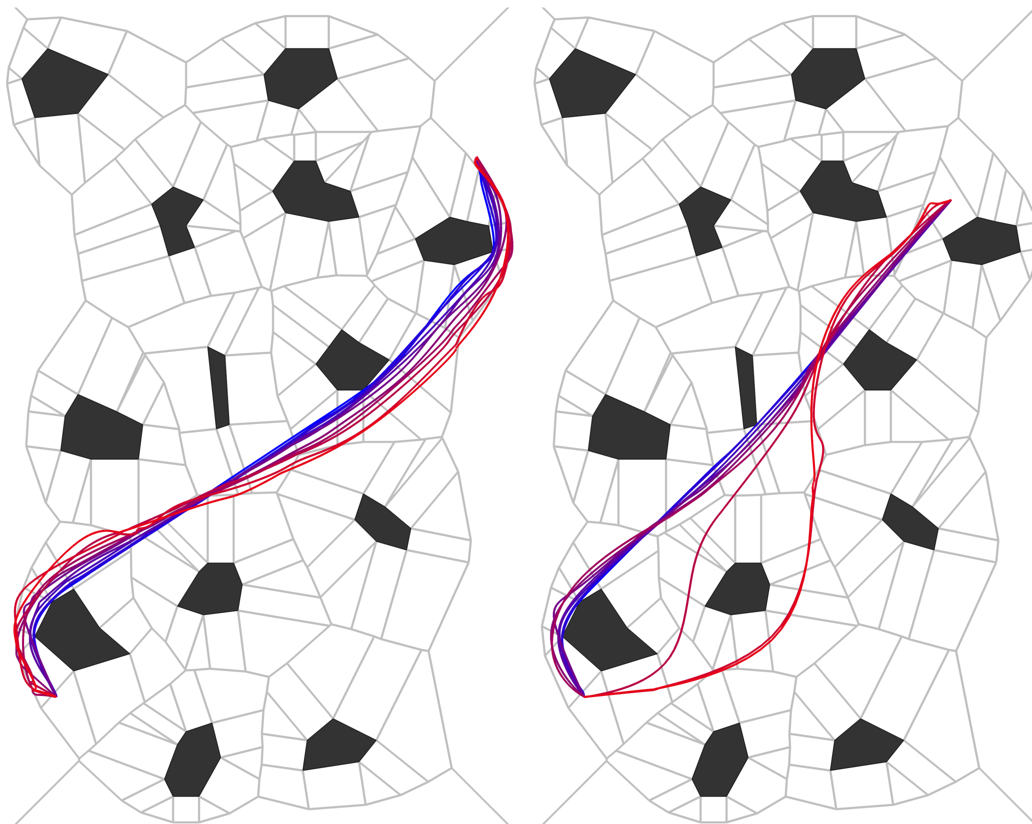

Figure 4.1: α scaling

4.2 Tolerance variation

Here we can observe the influence of simplification tolerance constant i.e. how much is simplified the polygonal polyline found by A* algorithm. As was already said the dimension of optimization problem relates with the number of control points and number of line segments in polygonal path. Tolerance was choosen at the sequence:

$$Tolerance = \{0, 3, 10, 15\}$$

In figure 4.2 we can observe the influence of simplification tolerance constant as the sequence passes also the pictures are sorted from the left top corner to the right bottom corner.

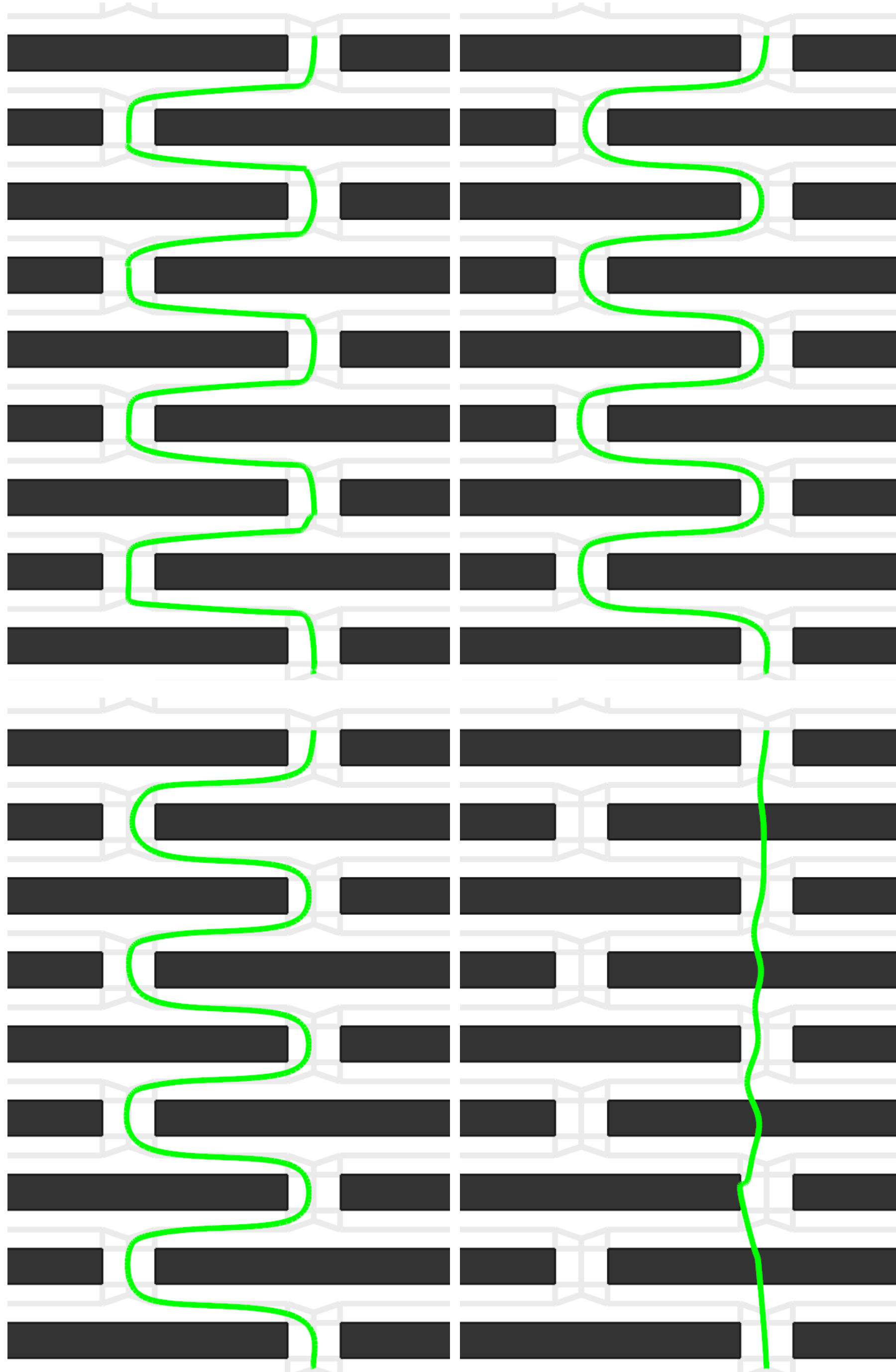

Figure 4.2: Tolerance impact

4.3 Final paths

This section shows the final paths in several different environments.

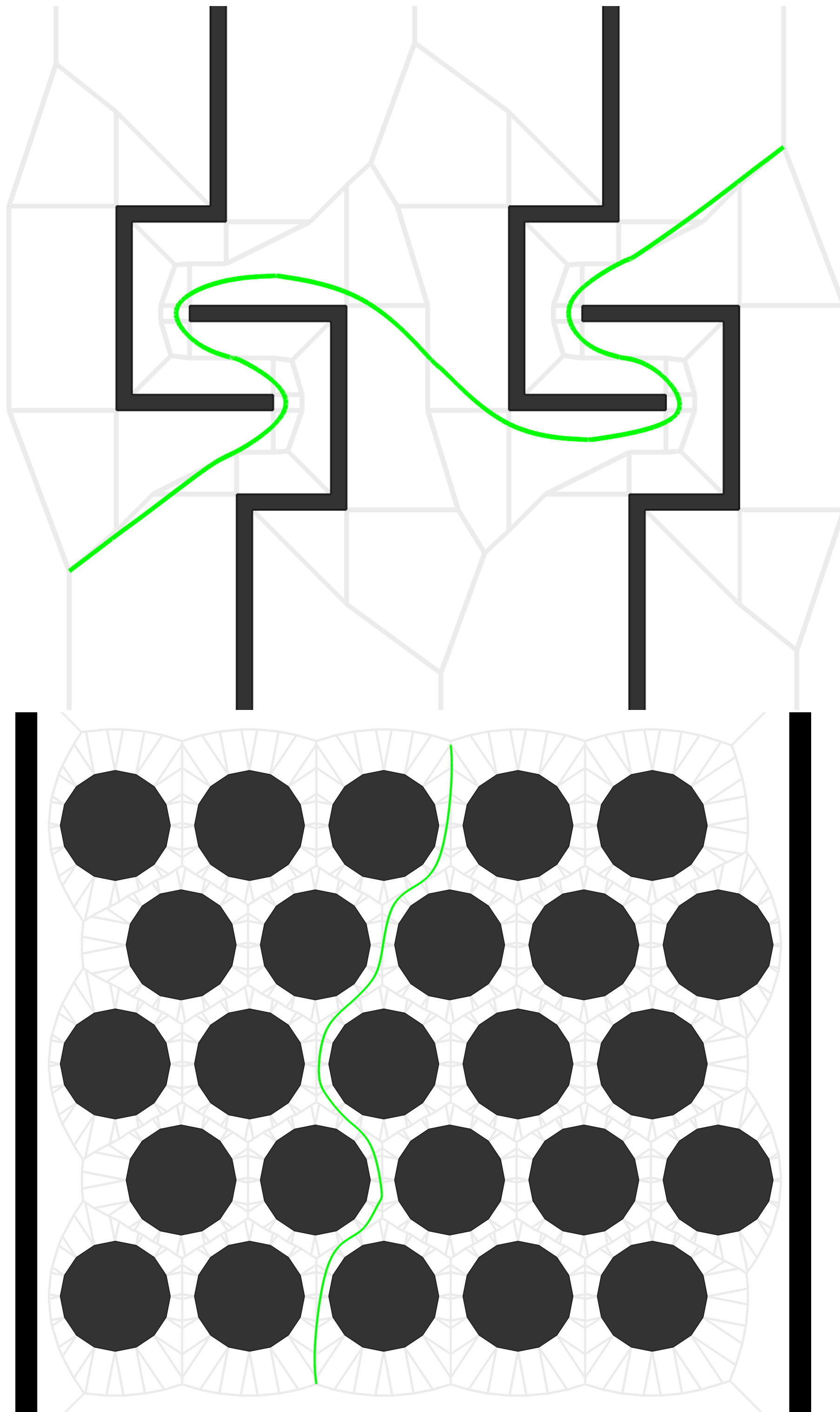

Figure 4.3: Final paths in different environment

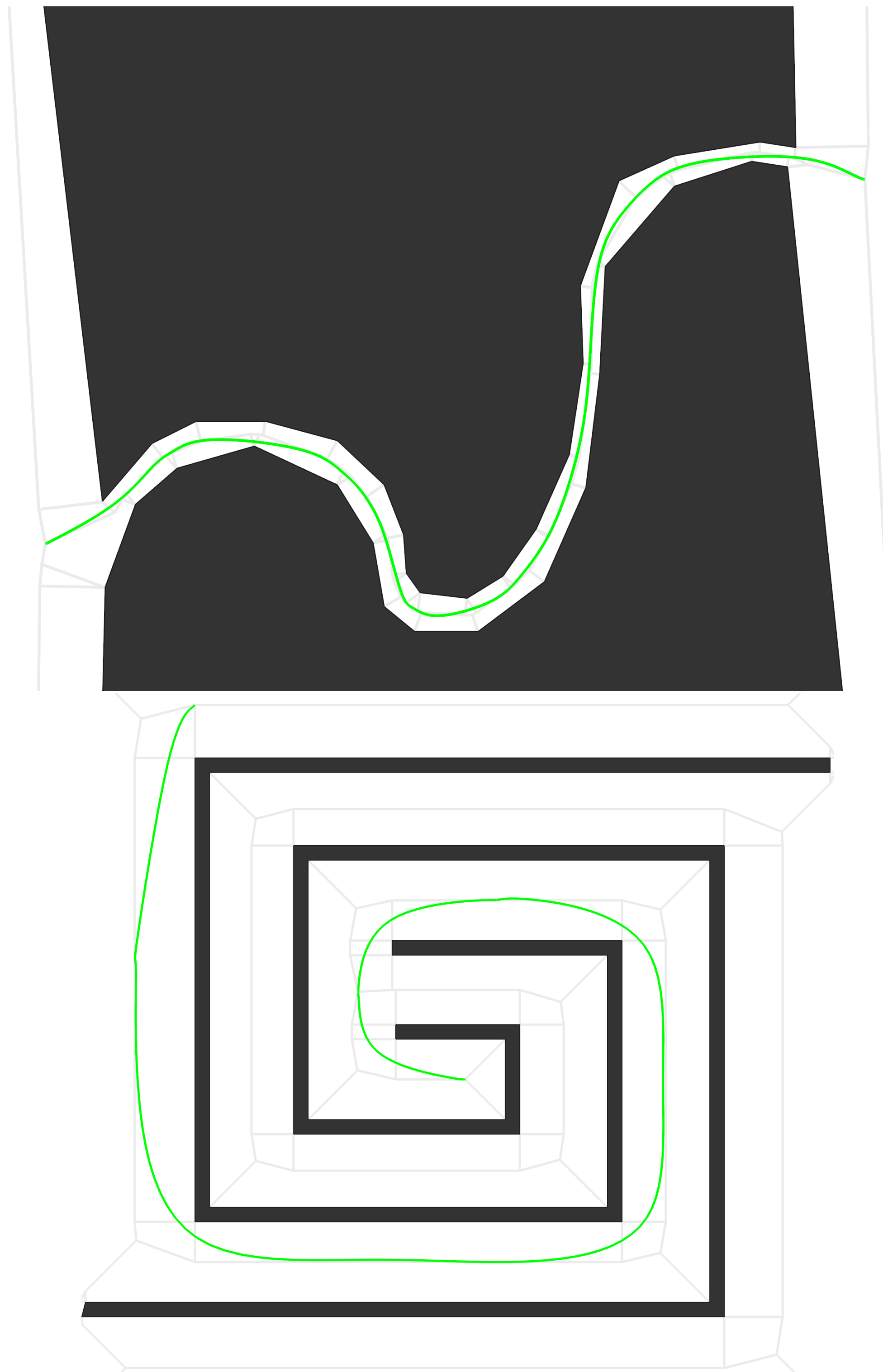

Figure 4.4: Final paths in different environment 2

Figure 4.5: Final paths in different environment 3

Figure 4.6: Final paths in different environment 4

Chapter 5

Conclusion

The thesis aims to design and implement an algorithm for generation of 2D smooth trajectories. The method implemented is based on construction of a spline consisting of a sequence of cubic Coons curves. The initial shape of the spline is determined from a shortest path on a generated Voronoi diagram of polygonal obstacles, while the final shape is generated as a result of an optimization process taking into account a length of the spline and its distance to obstacles.

The most algorithmically problematic part was to find intersections of the spline with a Voronoi diagram, where problems with numeric inaccuracies appeared. Finding intersection is crucial for counting the penalty function and it significantly affects the optimisation process, in which evaluation of the penalty function is called many times. Due to these facts intersections determination must be as effective as possible. Inaccuracies were treated with tests denoted at 2.4. The next occasional problem lied in misdetermination of a point to a particular cell, which was resolved by the correction mechanism denoted at 3.4.

As we can see in Chapter 5, the goal has been reached but the method has still several issues to be solved before a real deployment for path planning.

The final processed path always depends on configuration of parameters such as integration/measurement step, α parameter (which trade-offs between safety and a length of the path) and simplification tolerance affecting the dimension of the optimization problem. We discovered that these ideal combinations of parameters vary quite widely with the character of the environment e.g. if there are narrow stages or sharp corners. Of course, the next substantial affection is the size of the map, a magnitude specifically – it depends a lot if proportions are in degree of 1 or degree of 100. This settings could be ensured by a detailed analysis of the given map followed by precomputation of the parameters but it would require deeper research of its behaviour for design such an autoconfiguration mechanism.

For ensuring the relevant outcome from optimization the generation of the initial spline must be permissible (not cross any obstacle). The conceptual method of generating was used not giving certainty in all cases processing the permissible curve. Development of this method would

add to this generation task a solid foundamentals. For example, this method could be based on feature of B-spline laying always inside its convex hull constructed over set of control points.

We must always find a compromise between algorithm speed and spline not being crooked much. If the number of optimization iterations is restricted for higher speeds, knobs sometimes occur on the spline. This could be improved with keeping higher speed with optimization of spline curvature.

Bibliography

Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. ISBN 3-540-61270-X.

Elmar de Konig. Polyline simplification, 2011. URL http://www.codeproject.com/Articles/114797/Polyline-Simplification.

Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In Jürgen Lerner, Dorothea Wagner, and KatharinaA. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02093-3. doi: 10.1007/978-3-642-02094-0_7. URL http://dx.doi.org/10.1007/978-3-642-02094-0_7.

A. Escande, S. Miossec, M. Benallegue, and A. Kheddar. A strictly convex hull for computing proximity distances with continuous gradients. *Robotics, IEEE Transactions on*, 30(3):666–678, June 2014. ISSN 1552-3098. doi: 10.1109/TRO.2013.2296332.

S Fortune. A sweepline algorithm for voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 313–322, New York, NY, USA, 1986. ACM. ISBN 0-89791-194-6. doi: 10.1145/10515.10549. URL http://doi.acm.org/10.1145/10515.10549.

E.G. Gilbert and D.W. Johnson. Distance functions and their application to robot path planning in the presence of obstacles. *Robotics and Automation, IEEE Journal of*, 1(1):21–30, Mar 1985. ISSN 0882-4967. doi: 10.1109/JRA.1985.1087003.

M. Kulich. Vyhlazování cesty pro autonomní vozítko. diplomová práce, MFF UK Praha, 1996.

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

Max McGuire. The half-edge data structure, 2000. URL http://www.flipcode.com/archives/The_Half-Edge_Data_Structure.shtml.

Michael Ian Shamos and Dan Hoey. Closest-point problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 151–162, Oct 1975. doi: 10.1109/SFCS.1975.8.

E.V. Shikin and A.I. Plis. *Handbook on Splines for the User*. Number sv. 1. Taylor & Francis, 1995. ISBN 9780849394041. URL https://books.google.cz/books?id=DL88KouJCQkC.

Appendix

CD Content

In table 5.1 are listed names of all root directories on CD

Directory name	**Description**

Table 5.1: CD Content