RRT based Path Planning in Static and Dynamic Environment for Model Cars

Bachelor Thesis Computer Science

David Gödicke

Berlin
2018

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

March 29, 2018

David Gödicke

Contents

1 Introduction

The research on autonomous robots is a very active field. Since the first DARPA challenge 2004 both industries and academic institutions opened research groups on autonomous vehicles. Particularly autonomous cars as a comfort have gained in interest with well known projects like the Google car Waymo[4] or the commercialization of semi-autonomous cars[3]. It is believed that AI controlled vehicle could be safer than human driving. And the traffic flow could be optimized through better communication between cars.
The FU Berlin developed three cars with the AutoNOMOS research group. The latest car *MadeInGermany* is allowed to drive in Berlin. The car also drove in Mexico. The University also participated in the Carolo-Cup where 1:10 scaled model cars compete against each other. The goal of this work is to improve the autonomous skills of the model car in regards to path planning and collision avoidance.

1.1 AutoNOMOS Mini v3

The AutoNOMOS Mini v3 is a 1:10 scaled car developed at the FU Berlin for educational purpose.
The car is motorized with a Brushless DC-Servomotor FAULHABER 2232. A servo motor HS-645MG is used to change the steering angle. Both motors are controlled through an Arduino Nano board.
The main control unit of the car is the Odroid board (XU4 64GB). The Odroid board runs the Robot Operation System (ROS) over Ubuntu. ROS is widely used in the academic field and allows simplified communication between different nodes (programs) running on the robot. Every node can subscribe or publish to typed communication channels called topics. Programming in ROS is done in python or C++. Python is usually used for prototyping and C++ for finished projects as it gives better performance.
The car also embeds some sensors. The MPU6050 is a gyroscope and accelerometer used to provide information about the car position and orientation. The Intel RealSense SR300 Kinect-type camera gives 3D information about the space in form of a point cloud and image.
A fish-eye camera (ELP 1080p) looking at the ceiling is also mounted on the car to provide indoor GPS like localization.
The RPLIDAR A2 360 provides information about obstacles around the car. Each scan returns 360 distance to wall values.
The measured values of each sensor can be accessed through the Odroid board on the corresponding ROS topic.

1.2 Path Planning

A path planning algorithm is computing a path between a starting position and a defined goal. The path is expected to avoid obstacles in order to assure safety. Most

planning algorithm are searching for the optimal path. The path optimality is defined by a cost function where the minimum cost is considered as optimal. A cost function could for example represent the distance, time, or energy consumption.
In our case the car has to follow the computed path and it is therefore important that the path is kinematically feasible. Car like robots are constrained in their movement by the steering angle of the front wheels. The car can also not slide to the side nor rotate on place.
Robots with a constraint on one or more axis of freedom are called non-holonomic.
Planning for non-holonomic robots is more complex as not any free trajectory is kinematically feasible. The RRT structure presented by LaValle[11] is a random sampling based searching algorithm and is widely used for constrained trajectories. Since the first RRT algorithm, variations and optimizations have been published.

1.3 Control Architecture

The capabilities of an autonomous car can be split in three functional components: Perception, Decision/Control and Vehicle manipulation [16].

Perception of the surrounding world is done through sensing. While sensing is simple the difficulty is to interpret the available data to compute a trustworthy word model. For example lane detection, localization, obstacle detection, etc..

The Decision/Control component uses the world model to compute trajectories for the car. Trajectories are constantly recomputed in order to assure that they are still valid. The component is responsible to find valid and safe paths. We will refer to the plan computed from the control component as global plan.

The vehicle manipulation component executes the trajectory by controlling the propulsion, steering and breaking. Using a look-ahead point on the global plan the local goal is computed. The local controller should also implement safety procedure (vehicle stabilization, emergency obstacle avoidance).

1.4 Objective

The objective of this work is to develop a global planner for the AutoNOMOS Mini v3 car. The planner should be able to use the world model in order to find a safe path to the goal. Given the minimum turning radius and the dimension of the car the path is expected to be feasible. Execution of trajectories is not considered by this work as it is the role of the vehicle manipulation component.

2 Related Work

The pathfinding problem consist of computing a valid path between a start and a goal position in a given state space. A path is a sequence of states where the first state is the start position and the last the goal. The graph structure is used to define the relation between states. Each node of the graph is a state i.e. a possible position in the world. Edges between nodes represent the trajectories between the corresponding states. Two nodes connected by an edge are called neighbors.
The Dijkstra-Algorithm computes an optimal path in a given graph. Directed edges and motion cost between nodes can be defined to optimize the path against wanted properties.
The A* algorithm extends Dijkstra by using an heuristic to direct the search to the goal, highly reducing the computational time in most cases.
Both algorithm expect that the graph is already build and iterate over it to find the optimal sequence. A common solution are grid based maps where each cell is a possible state. The neighbors of a cell are the surrounding cells.
While grid based maps offer a simple solution to build a graph they are not suited to non-holonomic trajectories as the transition between two states are often to sharp. The possible states are also limited to grid resolution and cannot be between two cells.

2.1 Sampling Based Algorithm

Sampling based algorithm can be used to build a graph during the expansion.
The Probabilistic Roadmaps algorithm[13] (PRM) uniformly samples states over a bounded state space. Each state is tested for collision and removed if not free. The states are then connected with valid motions between them. Motions passing through obstacles are removed. By choosing trajectories that are kinematically feasible the planner assure that following a sequence of states in the graph is possible for the car. After the graph is build it can be used with Dijkstra or A* to compute a path between two states. PRM are not incremental as the number of nodes has to be defined before building the graph.
The Rapidly-exploring Random Tree algorithm[11] (RRT) incrementally builds a graph by sampling states one by one. Each iteration of the algorithm inserts one state at maximum. Given a new sampled state x the nearest neighbor $x_{nearest}$ in the tree is computed. If the motion connecting those states exceeds a predefined distance a new state is selected between them. The state x and the edge connecting x to the parent $x_{nearest}$ is then inserted in the RRT if it is collision free. The tree is initialized with the starting state and the path can be computed by inversely following the parent relations from the goal to start. The notion of cost does not exist therefore the RRT algorithm cannot be an optimal planner.
Both PRM and RRT are probabilistic complete. Given enough sampled point it is guaranteed that a solution is found if the problem is solvable.
Since the presentation of the RRT algorithm research has been focused on improving the path quality and reducing execution time of the algorithm. The RRG and RRT^*[9]

extend the *RRT* algorithm in order to guarantee asymptotically optimality. Each state stores its cost from start. After a new state has been sampled and appended to the Tree the surrounding neighbors are tested. If a better path going through the new sampled state is found the neighbors are rewired.
The $RRT - Connect$[7] algorithm expands two Trees. One rooted at the goal, the other rooted at the start state. After each insertion the algorithm tests if the two graphs can be connected. $RRT - connect$ finds the path faster than *RRT* but is not optimal as it uses the *RRT* expansion.
Other improvement of the *RRT* algorithm are trying to reduce the number of sampled point in order to save computational time. The $Informed - RRT^*$[8] reduces the sampled space accordingly to the actual best path. States that are two far away from the goal will not be sampled anymore. The $Theta - RRT^*$[12] algorithm computes a first path in a low resolution grid based map and then directs the RRT^* sampling along the found path. The resulting Tree expands along the already known optimal path and is only used to compute a kinematically feasible path.

2.2 Dynamic Planning

When planning for real world situation it cannot be assumed that the world is fully predictable. The computed path is therefore not guaranteed to stay valid. The need for constant re-planning implies that planning is either really fast or that the path can be rapidly corrected. In[17] the authors present a planning solution using RRT^* and a full size robotic forklift as example. The planner must run during an initial time of "a few seconds" before the robot can start moving. In case that the path is no longer valid the full RRT^* has to be recomputed.
The RRT^x algorithm[14] offers a repairing mechanism similar to D^*[18] and the same complexity in time as RRT^* in a static environment. Repairing the tree should cost less computational than a full growth as only the invalid portion have to be rewired and not the full tree. Because of the limited computational power of the AutoNOMOS car a solution with RRT^x will be implemented. The Open Motion Planning Library[1] implements the RRT^x algorithm but without the dynamic capabilities.
Table 1 shows the different capabilities of some *RRT* based algorithm.

Table 1: RRT comparaison

Algorithm	Probrabilistic Completeness	Asymptotic Optimaltiy	Replanning	Processing Complexity	Query Complexity
RRT	yes	no	no	O(n log n)	O(n)
RRT*	yes	yes	no	O(n log n)	O(n)
Informed RRT*	yes	yes	no	O(n log n)	O(n)
RRT connect	yes	no	no	O(n log n)	O(n)
RRTx	yes	yes	yes	O(n log n)	O(n)

3 Definition

Because of the kinematic constraints it is not trivial to compute a valid path for a car. In order to understand and plan with those constraints we define a mathematical model of the car.

3.1 Car Model

3.1.1 Configuration Space

The Ackermann steering geometry model describes the mechanical model of the car. The wheelbase L corresponds to distance separating the front and the back wheels. The position of the car is the middle point between the back wheels with coordinate (x, y).
The angle θ defines the car orientation in world space. The actual steering angle is ϕ. Assuming that the car is driving on a flat world we can define the Configuration space as $C = \mathbb{R}^2 \times \mathbb{S}$. A configuration $q \in C$ is $q = (x, y, \theta)$. We consider that the steering angle can change instantly and is not representative of the cars position in the world and can therefore be ignored in the configuration.

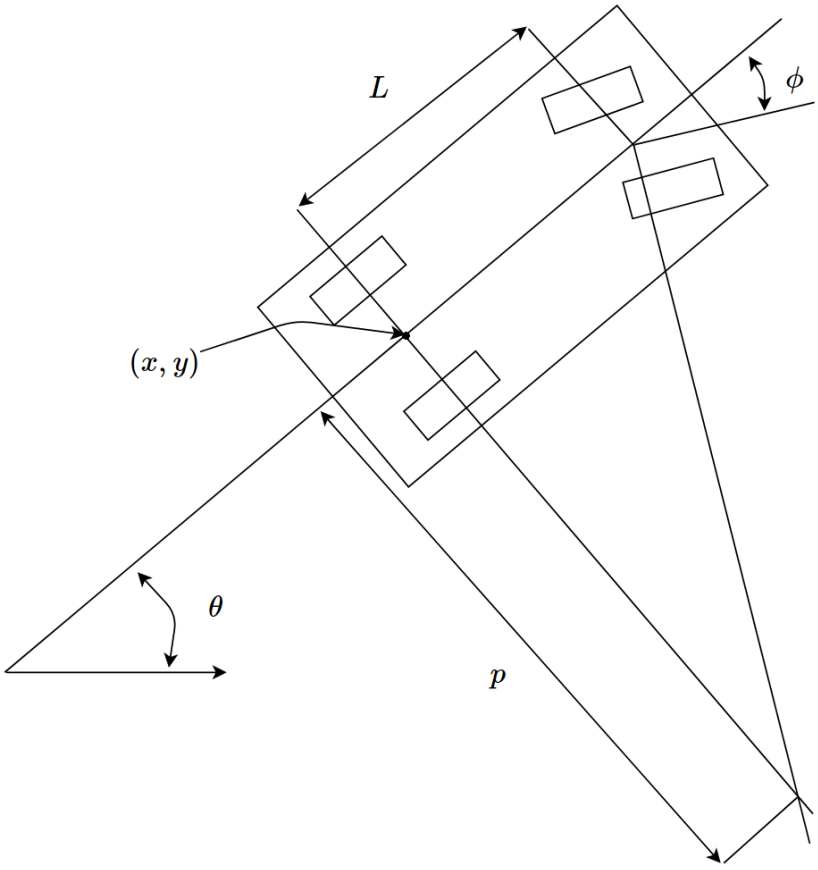

Figure 1: Steering geometry for a simple car[10]

3.1.2 Kinematic Constraints

A car has two degrees of freedom. It can move either forward or backward along its own x axis and change the steering angle ϕ of the front wheels. The mechanical limits of the car are bounding ϕ to a maximum steering angle ϕ_{max}. The steering angle modifies the orientation θ if the car is driving with speed s where $|s| > 0$.
With constant ϕ and $\phi \neq 0$ the car drives a circle with radius p. The steering angle can be calculated from the turning radius p:

$$\phi = \arctan\left(\frac{L}{p}\right)$$

The maximum steering angle is:

$$\phi_{max} = \arctan\left(\frac{L}{p_{min}}\right)$$

A system with differential constraints is said to be nonholonomic. Path planning for nonholonomic robots is more complex as it needs to take more parameters into account.
Given two input variables u_s and u_ϕ we can express the car motion as:

$$\dot{x} = u_s \cos\theta$$

$$\dot{y} = u_s \sin\theta$$

$$\dot{\theta} = \frac{u_s}{L} \tan u_\phi$$

In order to satisfy the kinematic constraints a path planning algorithm has to find a path that the car can drive where $|u_\phi| \leq \phi_{max}$ at any time.

3.2 Basic Motions

A first approach to the path planning problem is to find the optimal motion between two states in the given state space. As we know the optimal path between two point in $\mathbb{R}^2$ is the straight line passing through the points. In the case of a car it is not always possible to drive the straight line because of the kinematic constraints. Therefore we have to define an optimal motion between two states $v, u \in \mathbb{R}^2 \times \mathbb{S}$, respecting a minimum turning radius of p_{min}

3.2.1 Dubins Curve

The Dubins curve presented by Lester. E. Dubins in [5] are defined over the state space $\mathbb{R}^2 \times \mathbb{S}$. A Dubins curve starts at an initial position q_I and tries to reach the goal q_G. The minimal turning radius is bounded by p_{min}. Without constraints $p_{min} = 0$ the result is a straight line from q_I to q_G. The Dubins curve is considered as a bounded curvature shortest path problem.

In our previous model the car can be controlled by the input variable u_s and u_ϕ. A Dubins curve is expecting that the speed is constant and positive $u_s = 1$ i.e. the car can only move forward and the acceleration is null. The steering angle is limited to three states $u_\phi \in \{-\phi_{max}, 0, \phi_{max}\}$. The car is either moving straight or doing a sharp turn to the left or right. By maximizing the steering angle the turning radius is reduced, which reduces the total length of the curve.
Replacing u_ϕ by the possible value we get three possible value for $\dot{\theta}$: $\{\frac{1}{-p_{min}}, 0, \frac{1}{p_{min}}\}$
With the the constant speed we can simplify the system to

$$\begin{aligned} \dot{x} &= \cos\theta \\ \dot{y} &= \sin\theta \\ \dot{\theta} &= u \end{aligned} \tag{1}$$

Where the input variable

$$u \in \{\frac{1}{-p_{min}}, 0, \frac{1}{p_{min}}\}$$

Dubins proves that every optimal path can be expressed as three consecutive motion primitives. The primitives can be denoted as symbols $\{L, S, R\}$ corresponding to left: $u = \frac{1}{-p_{min}}$, straight $u = 0$ and right $u = \frac{1}{p_{min}}$. Because two equal symbols can be reduced it follows that for each sequence $\{S_1, S_2\}$: $S_1 \neq S_2$. A sequence of three symbols is called a word. There are six words that are possibly optimal:

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

In order to define an exact dubin path we denote the duration of each primitive by the subset α, β, γ, d. Where $\alpha, \gamma \in [0, \pi)$,$\beta \in (\pi, 2\pi)$ and d is a positive distance $d \geq 0$. Rewriting the six words

$$\{L_\alpha R_\beta L_\gamma, R_\alpha L_\beta R_\gamma, L_\alpha S_d L_\gamma, L_\alpha S_d R_\gamma, R_\alpha S_d L_\gamma, R_\alpha S_d R_\gamma\}$$

Let denote C a symbol defining a curve, corresponding to L or R. We can compress the optimal paths in two base words:

$$\{C_\alpha C_\beta C_\gamma, C_\alpha, S_d, C_\gamma\}$$

To find the optimal path between two states in $\mathbb{R}^2 \times \mathbb{S}$ one has to test every possible word and save the solution with minimal distance.
Computing the dubins path are not in the scope of this work. More information can be found in Dubins work[5]. An open source implementation of the Dubins state space can be found in the ompl library[1].

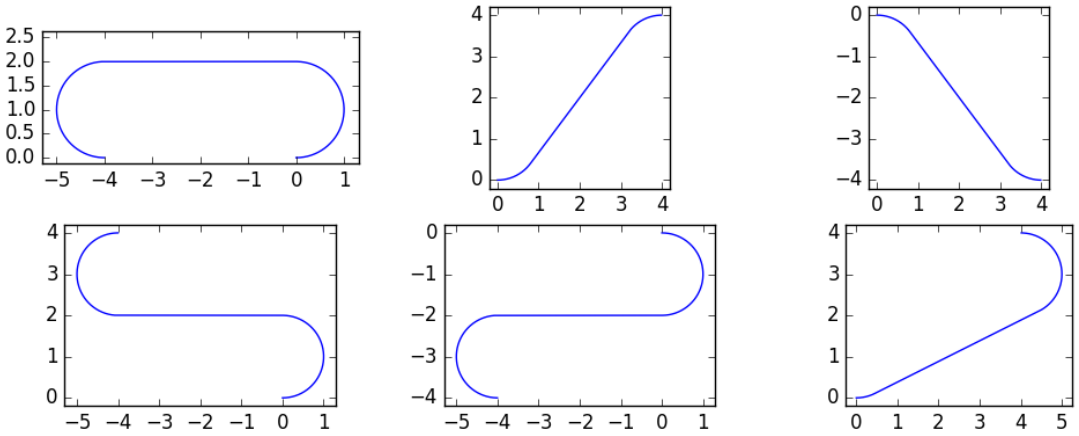

Figure 2: Dubins paths example

3.2.2 Reeds Shepp Curve

One limitation of the Dubins curve is that the car is only allowed to move forward. In some situation where the car has a limited space to turn, for example when parking, it is necessary to be able to switch between forward and backward driving as human drivers would do it. The Reeds Shepp curve is similar to the Dubins curve as it is using the same motion primitive L, R and S. The difference is that a negative speed is allowed, making backward paths possible.
Considering the Dubins control system 1 we can extend it to

$$\begin{aligned} \dot{x} &= u_1 \cos\theta \\ \dot{y} &= u_1 \sin\theta \\ \dot{\theta} &= u_1 u_2 \end{aligned} \tag{2}$$

Where u_1 is the speed $u_1 \in \{-1, 1\}$ and u_2 the steering $u_2 \in \{\frac{1}{-p_{min}}, 0, \frac{1}{p_{min}}\}$.

In [15] it is proved that the optimal Reeds-Shepp path between two states is one of 46 possible words. Let C be the curve symbol analog to the Dubins curve.
The symbol $|$ denote a speed change from 1 to -1 or inversely from -1 to 1. In compressed form those words are

$$\begin{aligned} &C|C|C, \quad CC|C, \quad CSC, \quad CC_u|C_uC, \quad C|C_uC_u|C, \\ &C|C_{\pi/2}SC, \quad C|C_{\pi/2}SC_{\pi/2}|C, \quad C|CC, \quad CSC_{\pi/2}|C \end{aligned} \tag{3}$$

The subset $\pi/2$ force a duration of $\pi/2$ on the corresponding primitive motions. The subset u on two following curves means that both curves have the same duration u. For more precision about the motions intervals and how to compute them we refer the reader to original publication [15] and to an open source implementation of the Reeds-Shepp state space[1].

Because of the higher number of words it is computationally more expensive to find the optimal Reeds Shepp path than the Dubins path between two states.
The Reeds Shepp curves are symmetric which means that if the trajectory between a and b is collision free, it is also between b and a.

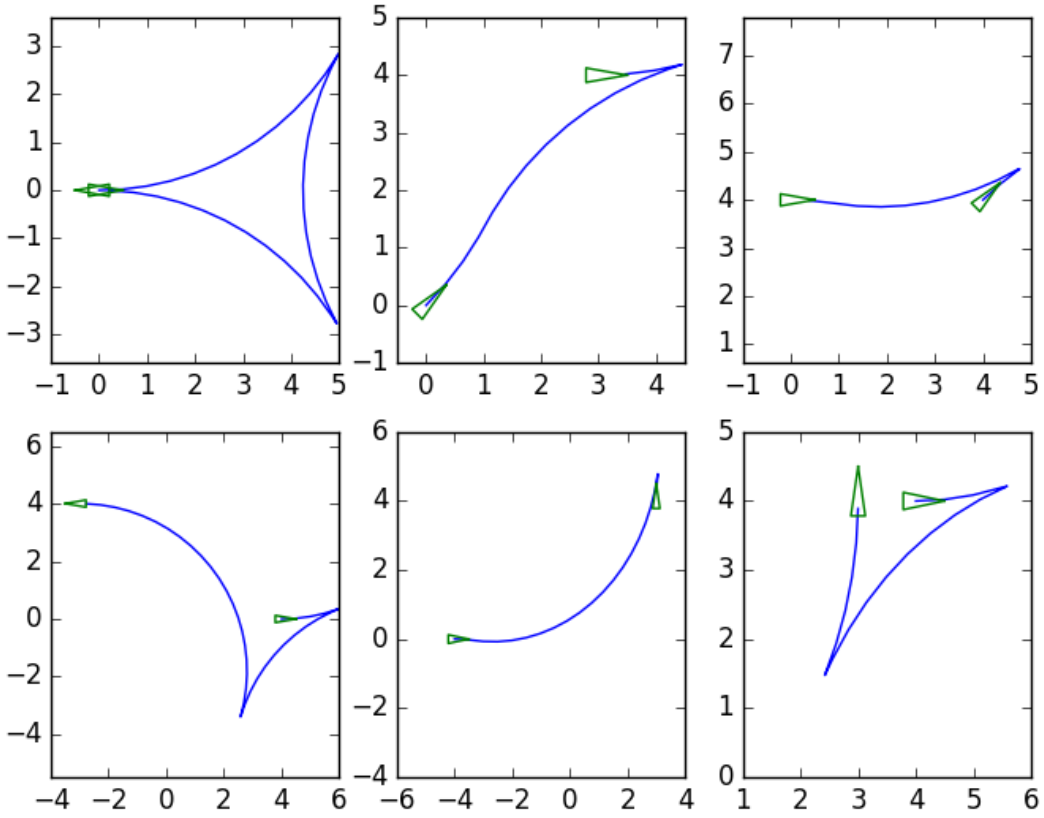

Figure 3: Reeds Sheep paths example

3.3 Problem Definition

Let $m^*(x_0, x_n) = x_0, x_1, .., x_n$ with $x_i \in X$ be the optimal path from x_0 to x_n. Each pair (x_i, x_{i+1}) corresponds to a basic motion $m(x_i, x_{i+1})$ respecting the steering constraints in X. Given two states x_{start}, x_{goal} the path finding problem P can be formaly defined as

$$P : (x_{start}, x_{goal}) \rightarrow m^*(x_{start}, x_{goal})$$

RRT based algorithms take the maximal distance η as argument where $\forall x_i, x_{i+1} \in m^*(x_0, x_n),\ d(x_i, x_{i+1}) \leq \eta$. RRT^x is ϵ-consistent therefore in a static environement it can formally be defined as

$$RRT^x_{static} : (x_{start}, x_{goal}, \eta, \epsilon) \rightarrow (m^*(x_{start}, x_{goal}), G)$$

The Graph G is rooted at x_{goal} and can be used to correct a path after a dynamic obstacle change ΔX_{obs} and ΔX_{free}. Given a new starting position v_{bot} between the

initial start and the static goal we can define the update function as

$$RRT^{x}_{update} : (G, v_{bot}, \Delta X_{obs}, \Delta X_{free}) \rightarrow (m^{*}(v_{bot}, x_{goal}), G)$$

4 Rapidly-exploring Random Tree x

The RRT^x algorithm is based over the RRT data structure and is using the same primitives functions. RRT^x is an ϵ-consistent algorithm meaning that a cost difference smaller or equal to ϵ is not triggering the rewiring cascade when comparing the cost of two paths. The ϵ-consistency is needed to remain asymptotically optimal.

4.1 Notation

4.1.1 State Space

Let X be a D-dimensional state space. The RRT^x algorithm requires that X has boundaries and is therefore a finite measurable metric space. The Lebesgue measure $L(.)$ is a standard way to describe a D-dimensional space as volume. It is expected that $L(X) < inf$.
Each state is either free or an obstacle. A state that can never be reached by the robot is considered as obstacle. The open subset of obstacles states is X_{obs}. And $X_{free} = X / X_{obs}$ is the closed subset of free states.
The optimal motion between two states is denoted $m(x_1, x_2) = \sigma$ where $x_1, x_2 \in X$ and σ is the resulting trajectory. Every motion is expected to be feasible i.e. the trajectory is always respecting the kinematic constraints of the robot. RRT^x expects a symmetric motion primitive $m(x_1, x_2) = m(x_2, x_1)$. The trajectory σ can be formally defined as a normalized mapping $\sigma(t) = x$ where $\sigma(0) = x_1$ and $\sigma(1) = x_2$ with $x \in X$ and $t \in [0, 1]$.
The distance between two states is denoted $d(x_1, x_2)$ and defines the length of the trajectory given by $m(x_1, x_2)$. The distance is expected to be always positive $d(x_1, x_2) >= 0$. Because a motion is the smallest trajectory between two states and that every distance is positive it follows that $d(x_1, x_3) \leq d(x_1, x_2) + d(x_2, x_3)$.
Because motions are symmetric the distance is also symmetric: $d(x_1, x_2) = d(x_2, x_1)$.
The travel cost between edges is defined by the cost function $c(v, u)$. The cost functions allows to optimize the path against another metric than the distance between states.

4.1.2 Vertices and Edges

$RRTx$ is building a graph $G = (V, E)$ during the expansion over the state space X. V denotes the set of vertices or nodes in the graph. Each vertex corresponds to a state.

We define $x_{start}, vbot, x_{goal}$ respectively as start vertex, actual robot position and goal vertex. E is the set of edges between two vertices in V. Each edges corresponds to a motion in X. Edges are directed $c(u,v) \neq c(v,u)$ with $u, v \in V$.
Similar to the D^* algorithm each vertex stores a cost-to-goal and a look-ahead value. The cost-to-goal value $g(.)$ saves the last optimal cost. The look-ahead value $lmc(.)$ saves the cost of a potential better path. The vertex v is called consistent when $g(v) = lmc(v)$. Consistent vertices are optimal as there is no potentially better path in the Tree. The $lmc(.)$ value allows cost propagation over the Tree after a better path is found.
In case that $g(v) > lmc(v)$ the vertex v is non consistent as it has a potential better path. Non consistent vertices are sequentially rewired to rebuild the optimal Tree. In order to reduce computational time RRT^x is ϵ-consistent. Therefore we consider that a state v is non consistent only if $g(v) - lmc(v) > \epsilon$.
Each vertex stores a set of neighbors. $N^-(v)$ denotes the set of incoming neighbors of v and $N^+(v)$ the outgoing neighbors. Each set of neighbors is the union between the initial neighbor's and the running neighbors:

$$N^- = N_0^- \cup N_r^-$$

$$N^+ = N_0^+ \cup N_r^+$$

where N_0 denotes the initial neighbors and N_r the running neighbors.
The initial neighbors are set at insertion time and stay constant during the expansion. The running neighbors are filled by new sampled neighbors. To reduce the number of edges the running neighbors are culled according to a shrinking ball radius later covered in section 4.3.
Each vertex $v \neq v_{goal}$ can store a parent. The parent vertex defines the next state in the optimal path $m^*(v, v_{goal})$. At any instant $g(v) = d(v, P(v)) + g(P(v))$.
A vertex without parent is said to be orphan and has no known path to v_{goal}. Let $V_{orphan} \subset V$ be the set of orphan vertices. For each $v_o \in V_{orphan}$ it follows that $g(v_o) = \infty$. Parent children relation are stored both direction. $C(v)$ is the set of children node with parent v. The result of the search is the sub-tree $G_T = (V_T, E_T)$. G_T is the shortest path tree. The set of optimal edges E_T contains only the edges (v, u) where $P(v) = u$. Orphan vertices are therefore not in the optimal tree.

4.1.3 Priority Queue

During the rewiring routine non consistent vertices are sequentially processed and made consistent by accepting the potential better path as new optimal path. In order to guarantee path optimality the vertices with the best lmc value is prioritized. The priority Q is used to order the vertices. The key used is defined as an ordered pair:

$$Key(v) = (min(g(v), lmc(v), g(v))$$

where $(a, b) < (c, d)$ if: $a < c \vee (a = c \wedge b < d)$

4.2 RRT Primitives

Rapidly exploring random trees can be build with a basic set of primitive functions:

$Sample(X)$: Given a state space X the sample function returns a randomly sampled state x where $x \in X_{free}$. The algorithm is only probabilistic complete if the $sample()$ function returns a normal distribution over X_{free} i.e. all the free space has to be discovered. The search is not deterministic because of the randomized sampling. In practice the sample function usually has a goal bias so that function will sample the goal state more often if it is not already in the tree. In case of the RRT^x algorithm the search is rooted at the v_{goal} and the sample function should sample the the start position v_{start}.

$Near(x, r)$: Given a state $x \in X$ and a radius $r \in R$. The function near returns every state u with $d(x, u) <= r$. The function is expected to have a complexity of maximum $O(\log n)$. This can be achieved through optimized data structures like R-Trees or Kd-trees. Achieving a complexity of $O(\log n)$ becomes more difficult in high dimensional space. The function is used to find potential new neighbours. During the expansion r will become smaller thus reducing the search range.

$Nearest(x)$: Given a state $x \in X$ the function nearest returns the nearest neighbour to x. Like the Near function it is expected to have a complexity of maximum $O(\log n)$.

$Steer(v, u, n)$: Given two states $v, u \in X$ and $n \in R$ a positive number. The function steer returns a feasible motion between v and u with maximal length n. If $d(v, u) > n$ the steer function returns a motion $m(v, u')$ where $d(v, u') \leq n$ and $u' \in m(v, u)$. The motion is not guaranteed to be collision free.

$CollisionFree(v, u)$: Given two states $v, u \in X$ the function test the motion $m(v, u)$ for collision. The motion is considered as collision free only if all states in $m(v, u)$ are valid.

4.3 Shrinking Ball Radius

The complexity of RRT^x is determined by the complexity of one iteration multiplied by the number of iterations. In order to achieve a complexity of $O(n \log n)$ for $n = |V|$ sampled states, one iteration has to take $O(\log n)$ time.
RRT^* introduces r the shrinking ball radius also used by RRT^x. The goal is to reduce the number of available neighbors during the vertex insertion (alg. 2) or the rewiring process (alg. 4) to a factor of $\log n/n$. Given the max distance η the radius is defined as

$$r = min\left(Y_{RRT^*}\left(\frac{\log |V|}{|V|}\right), \eta\right)$$

$$Y_{RRT^*} > \left(2\left(1+\frac{1}{d}\right)\right)^{1/d}\left(\frac{\mu(X_{free})}{C_d}\right)^{1/d}$$

Where C_d is the volume of the unit ball with dimension d and $\mu(X_{free})$ the free space in X. With a lower Y_{RRT^*} value RRT^* and therefore RRT^x is not probabilistic complete. With a higher value the ball radius and the complexity of the algorithm increase. Y_{RRT^*} Should be selected as small as possible.
RRT^x is storing the set of neighbors for each vertex. Those have to be culled at each iteration in order to respect the shrinking radius constraint. The $cullNeighbors()$ routine removes all running neighbors $u \in N_r(v)$ with $d(v,u) > r$.
We show that the original neighbors do not have to be culled over time:
Let v_n be a new vertex and r_n be the radius at insertion n. The maximum number of neighbors for v_n is $|Near(v_n, r_n)|$. $Near$ is limited by the shrinking ball radius (alg:2, line 1). $N_0(v_n)$ is set only once at insertion time and is therefore limited to a factor of $\log n$.

4.4 Algorithm

In this section we will present how the RRT^x algorithm is building the optimal tree. For simplicity we consider that states and vertices are equal. So that the state $x \in V$ corresponds to the vertices at position x in the graph $G = (V, E)$.

4.4.1 Initialization

The graph $G = (V, E)$ is initialized with:

$V = v_{goal}$
$E = \{\emptyset\}$

For each vertex $v \in V$ the default values are:

$g(v) = \infty$
$lmc(v) = \infty$
$P(v) = \emptyset$
$C(v) = \emptyset$

With goal initialization:

$g(v_{goal}) = 0$
$lmc(v_{goal}) = 0$

The states have no neighbors and the priority queue is empty $Q = \{\emptyset\}$.

4.4.2 Expansion

In optimal sampling based algorithms the expansion has two roles. One is to build a graph over the given state space, the other is to rewire the vertices in order to maintain

path optimality similar to the expansion in deterministic algorithms (Dijkstra, A*).
Static planners (A*, RRT, RRT*) initialize the expansion at the start state. G is rooted at x_{start} and for each state in the optimal tree a path from the start is known. For each $x \in V_T : m^*(x_{start}, x) \neq \emptyset$.
Dynamic planners (D*, RRTx) start the expansion at the goal and try to find the start state. It follows that for every state $x \in V$ where $x \cap V_{orphan} = \emptyset$ a path from state to goal is known $m^*(x, x_{goal}) \neq \emptyset$.
Because G is rooted at x_{goal} it is possible to update the robot position over time by choosing a new x_{start} without rebuilding the tree. The cost-to-goal value stored by each vertices is used to compare path optimality of the surrounding neighbors. Inversely when expanding from the start state it follows that x_{start} is static and x_{goal} can be updated without loosing the expansion.

The $BuildRRT^x$ function is shown in Algorithm 1. After the graph is initialized with the goal state(line 1-3) the main loop is executed until the termination condition ptc is met. At each iteration the shrinking ball radius is updated (line 5). The $Sample$ function samples a new state $x \in X$ (line 6) and the nearest state $x_{nearest} \in V$ is computed. In order to respect the steering constraint a state x_{steer} is selected between x and $x_{nearest}$ with maximal distance r. The new state x_{steer} is assigned to x and is guaranteed to have at least one feasible motion: $m(x_{steer}, x_{nearest})$ with distance smaller or equal to r. The Extend function tries to append x to the tree and is only executed if x is not an obstacle (line9-10). When x is successfully inserted in the tree the neighbors are rewired through the $RewireNeighbors$ and $ReduceInconsistency$ procedure. When the termination condition is met the algorithm returns G.

Algorithm 1 $BuildRRT^x(X, x_{start}, x_{goal}, ptc)$

```
1:  G ← (V, E)
2:  V ← {x_goal}
3:  vbot ← x_start
4:  while ptc ≠ True do
5:      r ← shrinkingBallRadius()
6:      x ← Sample(X)
7:      x_nearest ← Nearest(x)
8:      x ← x_steer ← Steer(x, x_nearest, r)
9:      if x ∉ X_obs then
10:         Extend(x, r)
11:     end if
12:     if x ∈ V then
13:         RewireNeighbors(x)
14:         ReduceInconsistency()
15:     end if
16: end while
17: return G
```

The function $Extend(x, r)$ (alg. 2) takes a new sampled state $x \in X$ and the shrinking ball radius as argument. The goal is to find a parent for x in V and to set the in-going and out-going neighbors. V_{near} contains all states $u \in V$ with distance $d(x, u) \leq r$ (line 1). The $FindParent$ function (Algorithm 3) sets the parent of x with $P(x) = u \in v_{near}$ and $m(x, u)$ obstacle free. The look-ahead value $lmc(x)$ is updated to the sum of the distance between x and u and the look-ahead value of u. If no valid motion could be found the parent stays null.
When a parent is found x is inserted to the graph and the parent-child information is saved (line 5-6). All neighbors $u \in V_{near}$ with valid motion $m(x, u)$ are appended to the original out-going neighbors of x. If the reverse motion $m(u, x)$ is valid u is stored as an original in-going neighbor of x.
Inversely u is storing x as a running in-going neighbor if $m(x, u)$ is valid and as running out-going neighbors if $m(u, x)$ is valid. Every edge is stored once as original neighbor and is therefore guaranteed be persistent during the culling process.

Algorithm 2 $Extend(x, r)$

1: $V_{near} \leftarrow Near(x, r)$
2: $FindParent(x, V_{near})$
3: **if** $P(x) = \emptyset$ **then**
4: **return**
5: **end if**
6: $C(P(x)) \leftarrow C(P(x)) \cup \{x\}$
7: $V \leftarrow V \cup \{x\}$
8: **for all** $u \in V_{near}$ **do**
9: **if** $CollisionFree(x, u)$ **then**
10: $N_0^+(x) \leftarrow N_0^+(x) \cup \{u\}$
11: $N_r^-(u) \leftarrow N_r^-(u) \cup \{x\}$
12: **end if**
13: **if** $CollisionFree(u, x)$ **then**
14: $N_r^+(u) \leftarrow N_r^+(u) \cup \{x\}$
15: $N_0^-(x) \leftarrow N_0^-(x) \cup \{u\}$
16: **end if**
17: **end for**

Algorithm 3 $FindParent(x, U)$

1: **for all** $u \in U$ **do**
2: **if** $lmc(x) > d(x, u) + lmc(u)$ & $CollisionFree(x, u)$ **then**
3: $P(x) = u$
4: $lmc(x) = d(x, u) + lmc(u)$
5: **end if**
6: **end for**

RewireNeighbors (alg. 4) takes a state $x \in V$ as argument. If x is not ϵ-consistent the rewiring process is started (line 1). A vertice x is considered inconsistent when $g(x) -$

$lmc(x) > \epsilon$ i.e the cost difference between the last optimal path and the potential new path is bigger than ϵ. The new inserted node is not consistent and therefore triggers the rewiring process. First the considered neighbors are reduced according to the shrinking ball radius with the *CullNeighbors* function in order to get the $O(\log n)$ complexity in one iteration. The function then iterates over the remaining in-going neighbors of x (line 3) ignoring the parent $P(x)$. If a neighbor u can find a new optimal path through x the $lmc(u)$ value is updated and the parent becomes x (line 4-5). If u became inconsistent it is appended to the priority Queue (line 7-8) to be later processed.

Algorithm 4 *RewireNeighbors*(x)

```
1:  if g(x) − lmc(x) > ε then
2:      CullNeighbors(x, r)
3:      for all u ∈ N⁻(x) \ P(x) do
4:          if lmc(u) > d(u, x) + lmc(x) then
5:              lmc(u) ← d(u, x) + lmc(x)
6:              MakeParentOf(x, u)
7:              if g(u) − lmc(u) > ε then
8:                  VerrifyQueue(u)
9:              end if
10:         end if
11:     end for
12: end if
```

Vertices are inserted in the priority queue Q when they are inconsistent.
ReduceInconsistency (alg. 5) is iterating over the priority queue as long as Q is not empty and that the path $m^*(vbot, x_{goal})$ is not finite or that the actual position $vbot$ is in the queue. The goal of the function is to remove inconsistencies in the Graph.
The vertex with smallest lmc value is processed first, let x be this vertex. If x is still not ϵ consistent a the new optimal parent is set through the $UpdateLMC$ function (algorithm 6). Potential cost changes are then propagated with the rewiring routine.
Because of the queue ordering it follows that every vertex $u \in V$ with smaller lmc value than x has to be consistent. Therefore any chosen parent $p(x)$ must be consistent and has a confirmed optimal path to goal.
The cost-to-goal value is updated to $g(x) = lmc(x)$ (line 7) making x consistent.

4.4.3 Replanning

The algorithms seen above are used to build the tree over a static state space. The following repairing functions are used to react to dynamic obstacle changes. Let O_{vanish} be the vanishing obstacles and O_{new} the new obstacles. The optimal tree is corrected by the $UpdateObstacles$ function (alg. 7). For each vanishing obstacle the function *RemoveObstacle* is called (line 2-4). The new obstacles are added through the function

Algorithm 5 $ReduceInconsistency()$

1: **while** $Q \neq \emptyset$ **and** $(Key(Top(Q) < Key(vbot)$ **or** $lmc(vbot) \neq g(vbot)$ **or** $g(vbot) = \infty$ **or** $vbot \in Q)$ **do**
2: $x \leftarrow Pop(Q)$
3: **if** $g(x) - lmc(x) > \epsilon$ **then**
4: $UpdateLmc(x)$
5: $RewireNeighbors(x)$
6: **end if**
7: $g(x) = lmc(x)$
8: **end while**

Algorithm 6 $UpdateLMC(x)$

1: $CullNeighbors(x, r)$
2: **for all** $u \in N^+(x) \setminus V_{orphan} : P(x) \neq u$ **do**
3: **if** $lmc(x) > d(x, u) + lmc(u)$ **then**
4: $p' = u$
5: **end if**
6: **end for**
7: $MakeParentOf(p', x)$

AddNewObstacle (line 8-10). The resulting orphan nodes are propagating the cost change to their children with the *PropageDescendent* function. In both cases the path optimality is restored with the *ReduceInconsistency* function (line 5 and 13).

Algorithm 7 $UpdateObstacles()$

1: **if** $O_{vanish} \neq \emptyset$ **then**
2: **for all** $o \in O_{vanish}$ **do**
3: $RemoveObstacle(o)$
4: **end for**
5: $reduceInconsistency()$
6: **end if**
7: **if** $O_{new} \neq \emptyset$ **then**
8: **for all** $o \in O_{new}$ **do**
9: $AddNewObstacle(o)$
10: **end for**
11: $PropagateDescendent()$
12: $verrifyQueue(v_{bot})$
13: $reduceInconsistency()$
14: **end if**

RemoveObstacle (alg. 8) searches for edges colliding exclusively with the vanishing obstacles (line 1-4). Edges that become obstacle free must be tested for insertion in the optimal tree. The motion costs are also actualized (line 7). The *lmc* value and parent are updated through the $UpdateLMC$ function (line 9). If a vertex can find a

better path it is inserted in the priority queue (line 10-11). The cost reduction is then propagated through the rewiring cascade.

Algorithm 8 *RemoveObstacle*(O_{vanish})

1: $E_o \leftarrow \{(v,u) \in E : m(v,u) \cup O_{vanish} \neq \emptyset\}$
2: $O \leftarrow O \setminus O_{vanish}$
3: $E_o \leftarrow E_o \setminus \{(v,u) \in E_o : m(v,u) \cup O \neq \emptyset\}$
4: $V_o \leftarrow \{v : (v,u) \in E_o\}$
5: **for all** $v \in V_o$ **do**
6: **for all** $u : (v,u) \in E_o$ **do**
7: $d(v,u) \leftarrow recalculate \quad d(v,u)$
8: **end for**
9: $UpdateLMC(v)$
10: **if** $lmc(v) \neq g(v)$ **then**
11: $verrifyQueue(v)$
12: **end if**
13: **end for**

Obstacles are inserted with the *AddNewObstacle* function (alg. 9). The motion cost of edges colliding with the obstacle are set to infinity (line 4). If an edge (v,u) is part of the optimal tree V_T the vertex v is added to the orphan set with the function $verrifyOrphan$ as the parent is not reachable anymore(line 6).

Algorithm 9 *AddNewObstacle*(O_{new})

1: $O \leftarrow O \cup O_{new}$
2: $E_o = \{(v,u) \in E : m(v,u) \cup O_{new} \neq \emptyset\}$
3: **for all** $(v,u) \in E_o$ **do**
4: $d(v,u) = \infty$
5: **if** $P(v) = u$ **then**
6: $verrifyOrphan(v)$
7: **end if**
8: **end for**

Both vertex insertion and obstacle removal propagate cost reductions over the tree. In case of obstacle insertion the new cost can be higher than the last optimal path. The function *PropagateDescendent* (alg. 10) propagates the cost change from orphan vertices to their children. All vertices having an orphan parent also become orphan (line 2). The non orphan neighbors are inserted in the queue (line 7). In order to force the rewiring routine for those vertices the cost-to-goal value g is set to infinity. The reason that the neighbors are added to the queue and not the orphan is that only vertices with an optimal parent can be used to find an optimal path as the *RewireNeigbhors* function rewires the in-coming neighbors.

Algorithm 10 *PropagateDescendent*()

```
1: for all v ∈ V_orphan do
2:     V_orphan ← V_orphan ∪ C(v)
3: end for
4: for all v ∈ V_orphan do
5:     for all u ∈ (N⁺(v) ∪ P(v)) \ V_orphan do
6:         g(u) ← ∞
7:         verrifyQueue(u)
8:     end for
9: end for
10: for all v ∈ V_orphan do
11:     g(v) ← ∞
12:     lmc(v) ← ∞
13:     if P(v) ≠ ∅ then
14:         removeParent(v)
15:     end if
16: end for
17: V_orphan = ∅
```

5 Implementation on AutoNOMOS

In this section we will present the implementation of a global planner based on the RRT^x algorithm for the AutoNOMOS car.

5.1 Kinematic Constraints

To avoid collision the cars dimension and constraints have to be taken in account. The cars length is 0.3m and width 0.1m. The distance between the front and back wheels is 0.26m. The minimal turning radius can be calculate by driving the car in a circle with the maximum steering angle. We measure a circle with radius $p_{min} = 0.74$m.
The ReedsSheppsCurves will be used as motion primitive to find drivable paths for the car with minimum turning radius p_{min}

5.2 Localization

Following the ROS convention[2] the car position is defined in the base_link frame. In our case the car is at position $(0, 0, 0)$ in base_link. The odom frame is used by the odometry to record the driven distance over time. The transformation between odom frame and base_link gives the actual position of the car as an offset to the starting position. The information is enough for the local controller as it allows to measure where the car is on the trajectory. When planning over a map the goal is not defined as an offset to the actual position but rather as a point on the map. The car

therefore needs to know its own position on the map in order to compute a trajectory. The position on the map can be computed with the visual GPS. The fish-eye camera is used to detect 4 lamps with different colors on the roof. With the resulting position the offset between the map frame and odom frame can be computed. The ROS transform library[6] allows transform chaining. The transform between frame a and c can be expressed as a multiplication between two transforms.

$$T_a^c = T_a^b * T_b^c$$

The inverse is also defined

$$T_a^b = (T_b^a)^{-1}$$

The GPS position gives us the transformation between the map frame and base link $T_{map}^{base\_link}$. Because the odometry runs at higher frequency it makes sense to use it to compute the positions more smoothly and not rely only on GPS updates. Therefore we will use the GPS data to correct the odometry position on the map by publishing the transform from map to odom.

$$T_{map}^{odom} = T_{map}^{base\_link} * (T_{odom}^{base\_link})^{-1}$$

In order to avoid position jumps we can use a Kalman filter on the GPS. Figure 4 shows the published transform tree. The additional laser frame defines the position of the LIDAR on the car. The map position can also be computed with other methods like SLAM mapping which randomly samples position on the known map and compares the potential sensor input with the measured input.

5.3 Obstacle Detection

The LIDAR sensor is used to detect surrounding obstacles. The sensor gives information on the distance between the car and obstacles for 360 angles around the car. Given the car position and orientation we can compute the position of an obstacle in the world. We will use the ROS costmap_2d package to build a map around the car.
The costmap package advertises a grid, each cell has a value between 0 and 255. Cells with value 0 are considered as free and cells with value $\geq$ 254 as obstacles. Cost between 0 and 254 can be interpreted as probably free. The costmap uses a system of layers to compute the cost of each cells. Usually the first layer is the static layer, representing the initially known map. In our case the next layer is the obstacle layer. The obstacle layer subscribes to sensor data and adds or removes obstacles on the map.
In previous experiments we considered the robot as a point when testing for collision. For the model car the whole footprint has to be tested. The inflation layer takes into account the robots dimension to inflate every obstacle present on the map. With a width of 0.1m and a length of 0.3m we can can consider that every obstacle with distance smaller than 0.1m to the center of the car is colliding with the car. This region is defined by the inscribed circle of the footprint. The circumscribed circle defines the states which are possibly in collision with the car depending on the actual orientation.

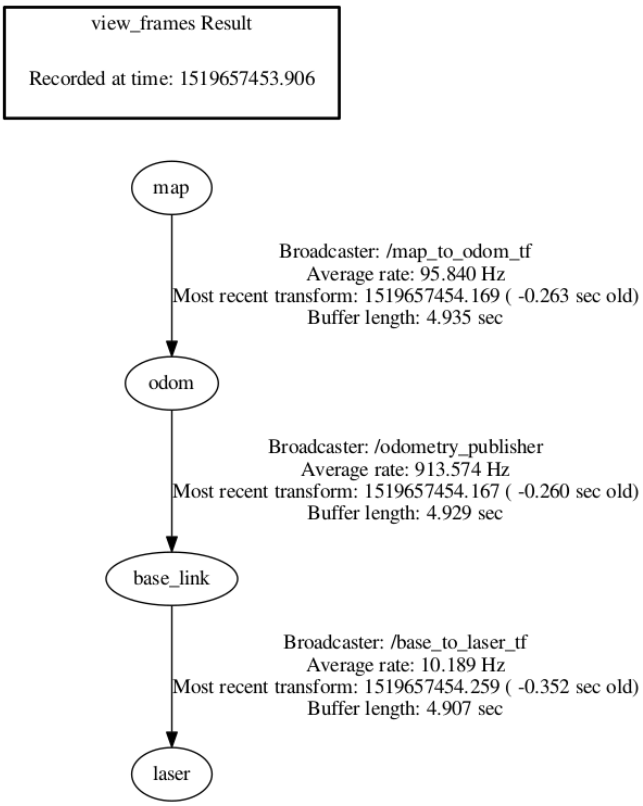

Figure 4: Transform tree

In our case the circumscribed circle has a radius of $0.3m$
Let x be the robots center, x_f, x_b the front and the back of the robot.
With the Inflation layer we can consider that x is free if:

$$Costmap(x) = 0 \textbf{ or } Costmap(x) < 254 \textbf{ and } Costmap(x_f) < 254 \textbf{ and } Costmap(x_b) < 254$$

Figure 5 shows the data flow from sensors to the build costmap.

5.4 Re-planing

The car should be able to react to new obstacles and find a new path. Because the car is running with less computational power we want to optimize the re-planning process to get the best reactivity.
To assure path validity the actual plan is permanently tested for collision. If a collision is detected the tree should be corrected and publish a new path. This methods guarantees that the optimal path is obstacle free. This lazy approach does not scan the whole costmap for appearing or disappearing obstacles in order to save computational time. Cost reduction are therefore not detected when the plan is still valid.
Figure 6 shows the process flow of the implemented planner.

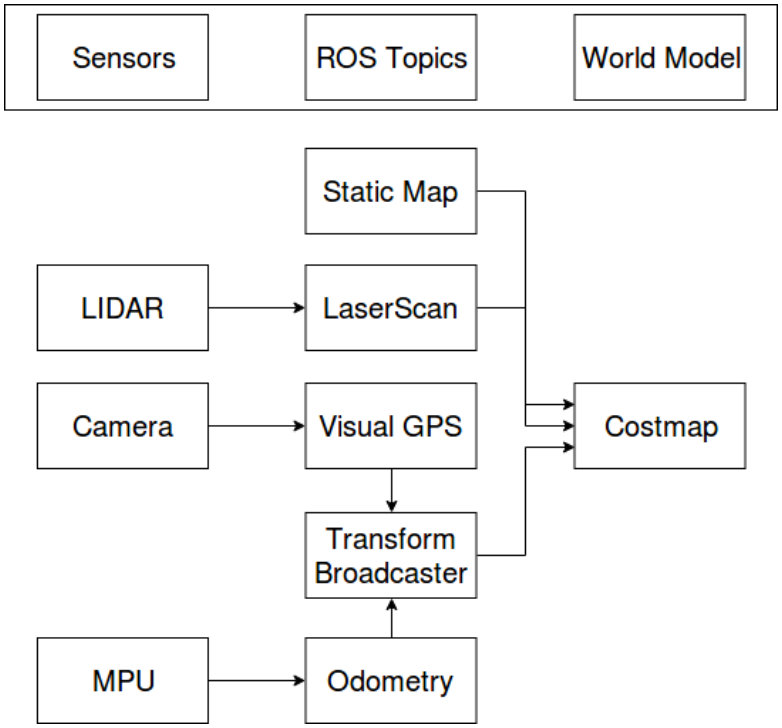

Figure 5: Data flow from sensors to world model

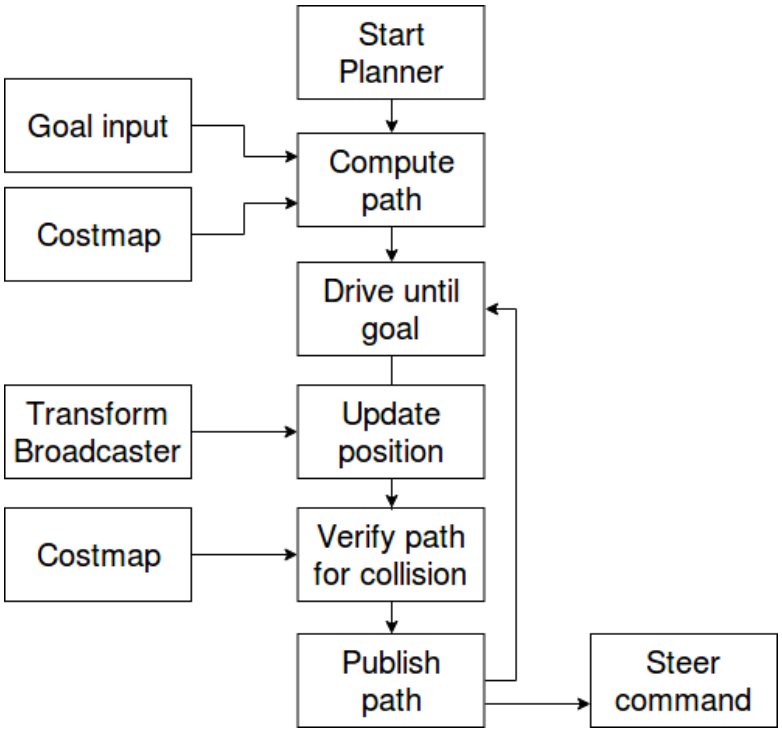

Figure 6: Planner process flow

6 Experiments and Analysis

The performance of the RRTx algorithm depends on user defined parameters. In this section we will discus the impact of each parameter on the optimal path. The robots position is represented by a point in $\mathbb{R}^2 \times \mathbb{S}$. Reeds Shepp curves are used as motion primitives. In the following figure the start and goal positions are represented respectively by the the yellow and red arrow. The blue dots are sampled states $v \in V$. The optimal Tree G_T is shown by the green edges. The purple trajectory represents the optimal path.
The following experiments will be executed. Experiment 1 to 5 are done in simulation, experiment 6 and 7 are done on the AutoNOMOS car.

1. In the first experiment the path quality is analyzed over time. The tree is build during 35 seconds. The cost function is defined as the curve length. Because the RRT^x algorithm is asymptotically optimal we consider that the optimal path is found after a long expansion ($t = 35s$).

2. In the second experiment we measure the computational time needed for a number of iterations. A complexity of $O(n \log n)$ is expected and would allow to update the position in $O(\log n)$ by sampling one new state.

3. In this experiment we simulate the same problem with different maximal distance η. The goal is to find the best settings for the planner. The path length is used to define the path optimality.

4. The efficiency of the RRT^x planner is tested for driving maneuvers in a constrained environment. The planner should be able to do a u-turn or park between two obstacles.

5. The advantage of the RRT^x algorithm over other static planners is the re-planning function. The gain of computational time between re planning and building a complete tree is analyzed.

6. The first experiment on the AutoNOMOS car is to drive along a path between two positions in free space.

7. Assuming that the previous experiment is a success we want to avoid obstacles with help of the laser scanner. First in a static and then in a dynamic environment.

6.1 Experiment 1-2: Time

For the first experiment the expansion is paused at intervals of $1s$ starting at $t = 0.5s$. Figures 7 to 9 visualize the optimal tree G_T at times $t = 0.5s$, $t = 1.5s$ and $t = 2.5$. Figure 10 shows the expansion after 35 seconds.
From the result we can see that a path is rapidly found at $t = 0.5$ with cost $C_p = 14.72$. At the next step the path cost are reduced by 2.2%. In figure 9 we can see more edges and sampled states but no improvement of the optimal path. At time $t = 35s$ the

optimal cost are $C_p = 14.00$. Between the first expansion and the tree at $t = 35s$ the path cost improved by 5% in $30s$.
Figure 11 shows the optimal cost over time during a similar expansion in free space. We can see the same pattern, a path is found in the first milliseconds and is improved over time. We can also see that the duration between two cost reductions grows over time.
The algorithm seems to be able to rapidly find an initial kinematically feasible path with near to optimal cost but takes a long time to minimally improve the path.

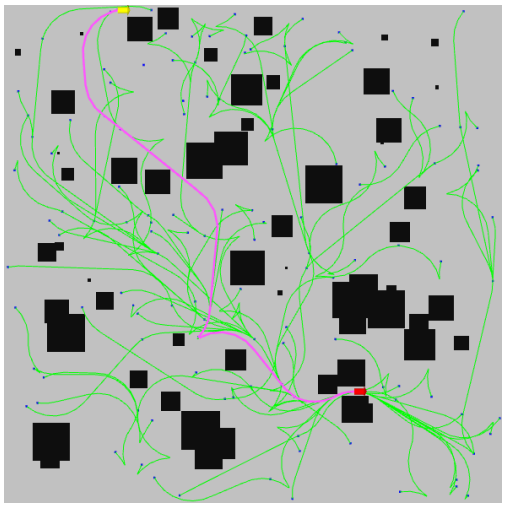

Figure 7: V_T at $t = 0.52$ with $C_p = 14.72$

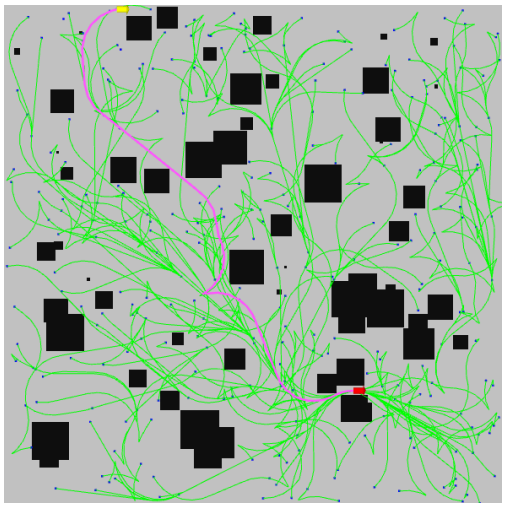

Figure 8: V_T at $t = 1.57$ with $C_p = 14.39$

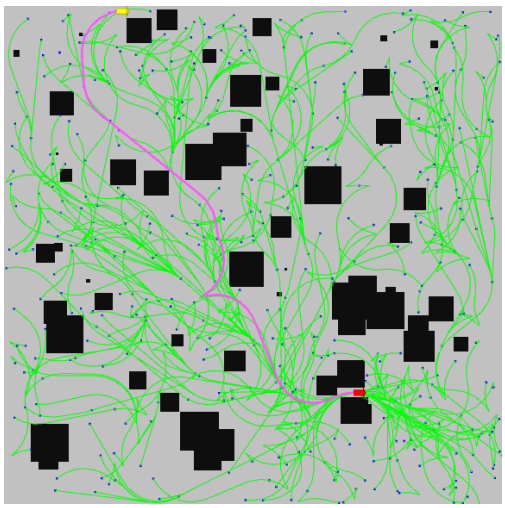

Figure 9: V_T at $t = 2.54$ with $C_p = 14.39$

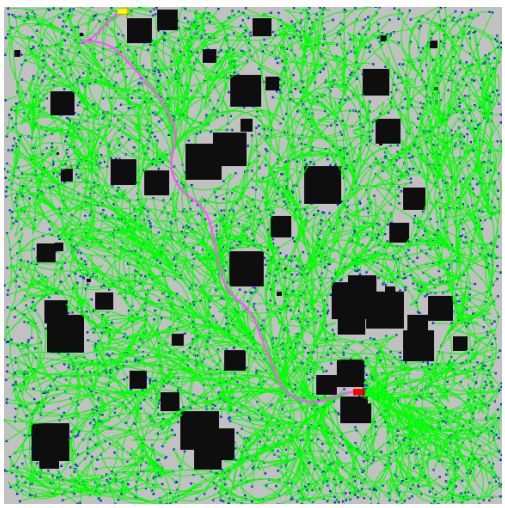

Figure 10: V_T at $t = 35.02$ with $C_p = 14.00$

In the second experiment we want to measure the time complexity of the algorithm. To do so we save the time and shrinking ball radius at each iteration during 10 minutes. Figure 12 shows the time at each iteration and figure 13 shows the radius length. The evolution of time per iteration seems to be near linear. We can therefore conclude that the implementation respects the $O(n \log n)$ time complexity. It follows that the robots position can be update in $O(\log n)$ time. This complexity is enabled by the shrinking ball radius. As we can see in figure 13 the radius is starting at the selected $\eta = 6m$ value and rapidly falls below $1m$. The set of considered neighbors during the rewiring function is therefore reduced over time.

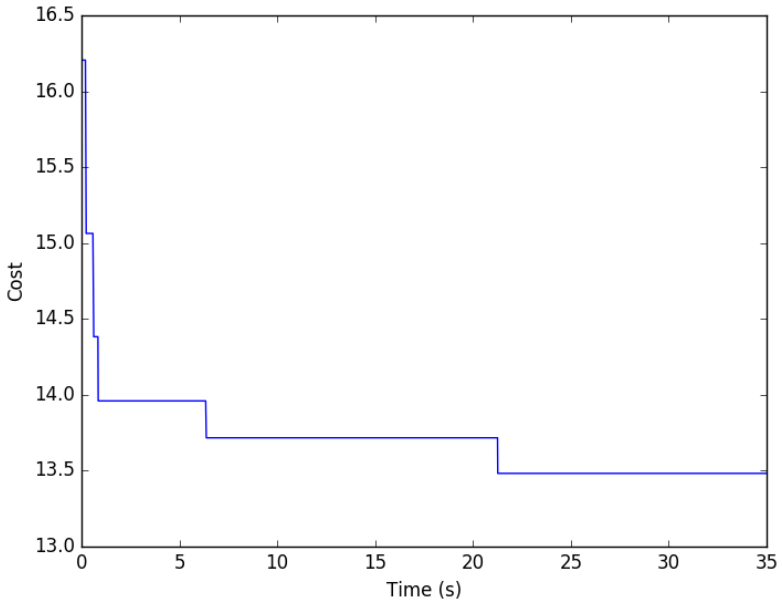

Figure 11: Path cost over time

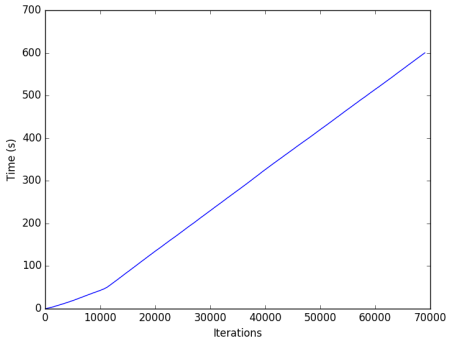

Figure 12: Time per iterations over 10 minutes

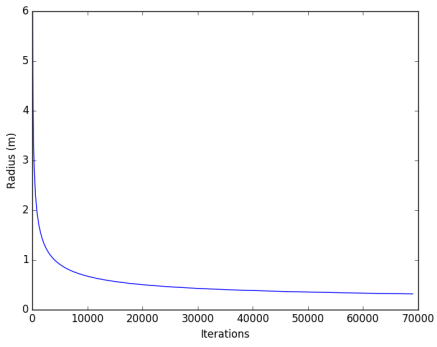

Figure 13: Shrinking ball radius r during expansion

6.2 Experiment 3: Sampling Distance

To measure the impact of the maximal sampling distance η we compare expansions with different distance settings. Each expansion is given the same computational time. Figure 14 shows an expansion at time $t = 1s$ with maximal distance of 0.5 meters. Figure 15 shows an expansion at time t with maximal distance of 5 meters. In figure 14 no path could be found. Because of the small distance most vertices have very few neighbors. A lot of space is not covered by the Tree. In figure 15 a path was found, there are more edges.
From the result we can see that a bigger distance between vertices allows the tree to expand faster over the space. The path is also usually smoother than with small η values.
The drawback of long trajectories are the collision test which become more heavy. In an environment with many obstacles big motions may constantly lead to collision, in this case a smaller η value should be selected.

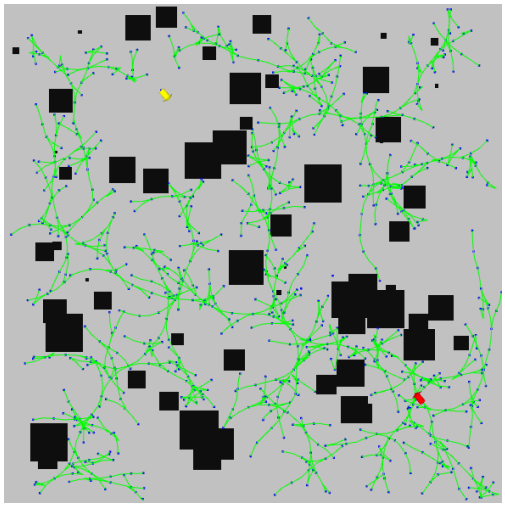

Figure 14: Expansion with max distance = 0.5m, turning radius = 1m

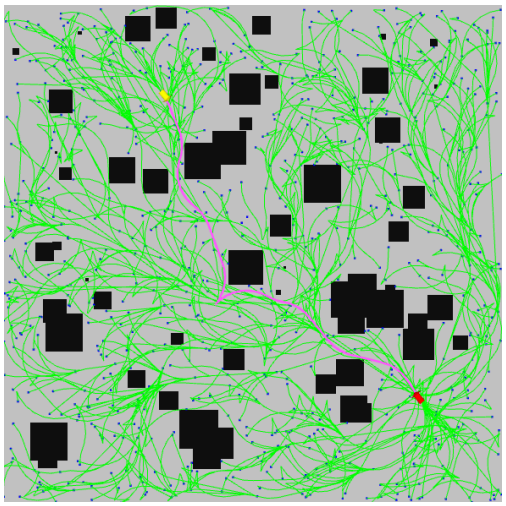

Figure 15: Expansion with max distance = 5m, turning radius = 1m

6.3 Experiment 4: Motion cost penalty

While the path in figure 10 is optimal in terms of distance it is not similar to human driving as the path has no preferences between forward and backward driving. In our case the algorithm is optimizing the path length toward the raw motion distance. In order to prioritize forward driving want to apply a penalty for backward motion segments.
Let $d_f(v, u)$ be the distance driven forward in the motion between v and u are two states. And $d_b(v, u) = d(v, u) - d_f(v, u)$ the distance driven backward. We define the

motion

$$m(v,u) = d_f(v,u) + penalty * d_b(v,u)$$

In Figure 16 the path is computed without penalty, most of the distance is driven backward. By multiplying the cost of backward motion by a factor of 2, forward paths are prioritized, as shown in figure 17.
The cost function can also be used to prioritize position over other in a structured space. For example the center of the lane on roads.

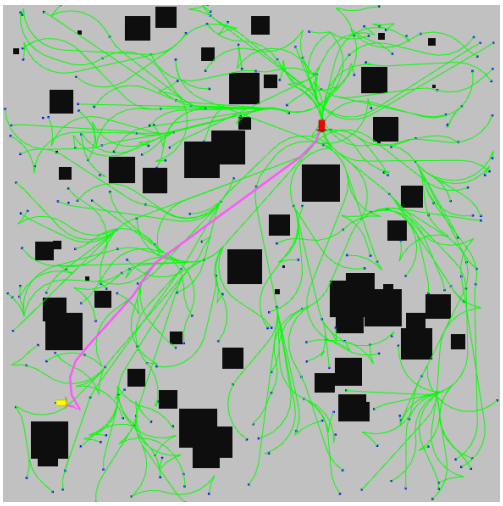

Figure 16: Path at time $t = 1$ without penalty

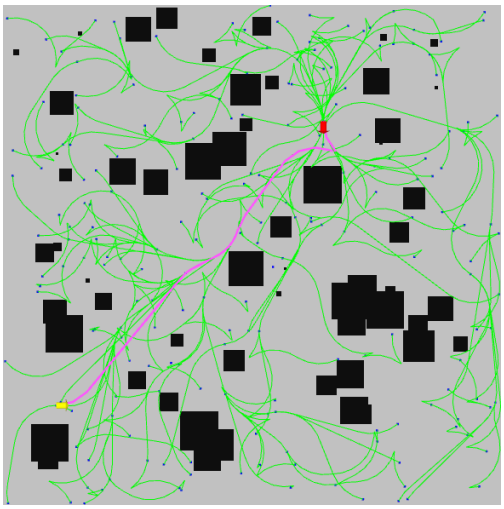

Figure 17: Path at time $t = 1$ with a cost penalty on backward motion ($\times 2$)

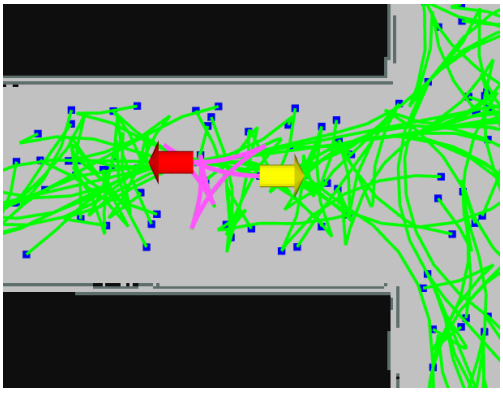

Figure 18: U-turn without penalty

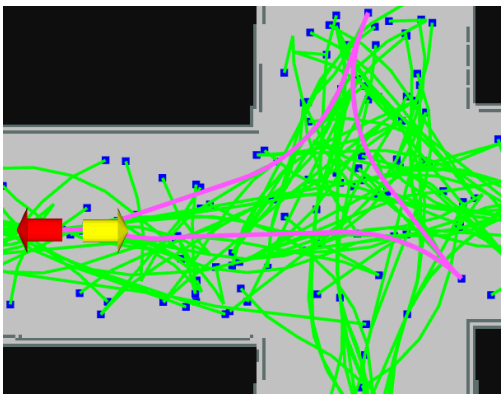

Figure 19: U-turn with direction change penalty

When driving in a constrained space the car may have to switch between forward and backward driving in order to reach the goal. The U-turn for example requires to switch 2 times in most cases. Figure 18 shows a U-turn without penalty on direction changes. Because of the steering constraint the car cannot reach the goal without

complex maneuver. The resulting path contains 5 direction changes. The optimization toward path length is prioritizing small paths and does not take comfort or time into account. In order to avoid direction changes it is possible to apply a penalty on those. Figure 19 shows a U-turn with a penalty of 3 meters on each direction change. The computed path is similar to human driving and contains only 2 direction changes.

6.4 Experiment 5: Re-planning

The RRT^x algorithm allows to correct the computed path if new obstacles appeared or disappeared. To do so the algorithm has to know which motions are no longer valid and which became valid. Finding those motions can be computationally heavy depending on the collision tests.
Figure 20 shows the tree after an expansion of t=2s. In Figure 21 the map contains a new obstacle and the path has been corrected. We can see that the sampled states are still at the same position and only the optimal Tree V_T has been rewired.
In this example the colliding edges are found by executing the collision test on every edge. The tree is then corrected as shown in algorithm 7. In our experiment we will measure the re planning time for different sampling time. We also want to reduce the space in which we execute the collision test. The hypothesis is that less collision test should also reduce the computational time.
RRT based algorithm do not offer the possibility to find edges around some state. We must therefore use the *Nearest* function to find states around the obstacle. Every edge between those states and their neighbors is then tested. In order to assure that all possibly colliding edges have been found we have to select the vertices in a radius of

$$r = r_o + (\eta/2)$$

around the obstacle, where r_o is the expected obstacle size.

Table 2: Re-planning time

Expansion (s)	1	2	4	8
Scan 100%	0.70	1.1	1.37	1.81
Scan 50%	0.24	0.35	0.63	0.73
Scan 25%	0.15	0.23	0.36	0.40

Table 2 shows the time needed to correct the path depending on the expansion duration and the percent of vertices selected to find new obstacles. The required time is always smaller than the initial expansion. The rewiring process seems specially efficient for bigger trees. When scanning all edges the performance gain at $t = 1s$ is 30% whereas at $t = 8s$ it is around 77%. Testing less edges further reduces the time, the performance gain seems linear.
The shrinking ball radius seems to also reduce the cost of the collision test since every new edges becomes smaller. Over time most of the computation comes from the

rewiring cascade. In bigger trees the proportion of orphans seems to dictate the re-planning time. In experiment 2 we confirmed that rewiring one state takes $O(\log n)$ time. The required time to rewire all orphans is therefore $O(n_o * \log n)$ time. Where n_o is the number of orphans after obstacle insertion. If every node becomes orphan $n_o = n$ the re-planning function takes as long as the expansion. In every other case re-planning is more efficient than re-building.

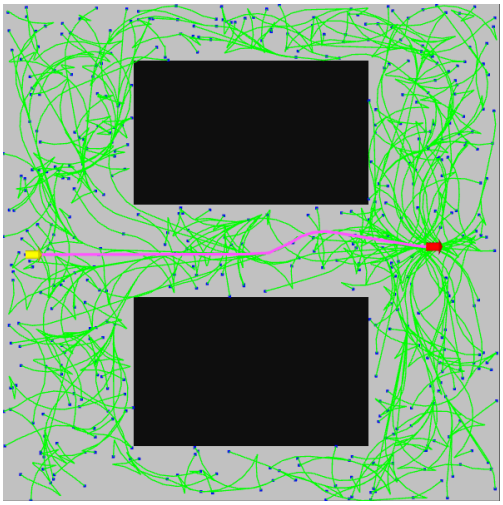

Figure 20: Tree expansion after 2 second

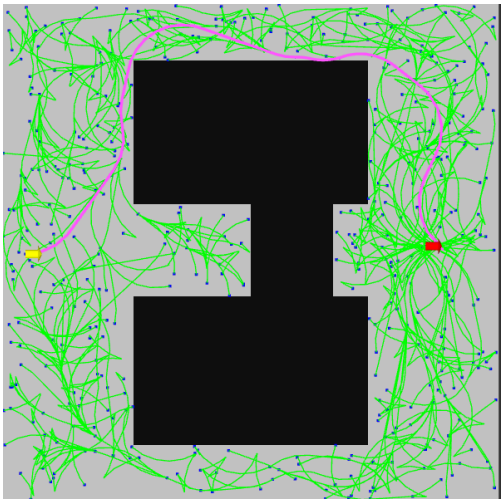

Figure 21: Tree after re-planning

During the path execution the position can to be updated. The new state can be rapidly inserted in one iteration of the $BuildRRT^x$ function by forcing the sampled state to be the updated position. In case that no parent could be found the tree has to grow until a new solution is found. Assuming that the robot is following the previous path the updated position should always find a parent in one iteration. Updating the position is therefore in the order of milliseconds.

6.5 Experiment 6-7: Driving

The first experiment on the car is to drive to a goal location from the actual position. Figure 22 shows the driven path, the red arrows represent the car positions over time. The map defines the free space in the robotic lab. The position is reached and the orientation is close to goal orientation.

The next experiment is to drive around an obstacle detected by the LIDAR scanner. Figure 23 shows the generated costmap with the detected obstacle. We chose a resolution of $5cm$ per cell in order to reduce the required time per map update. The computed path is shown in figure 24. The car receives a valid path and is able to avoid collision.

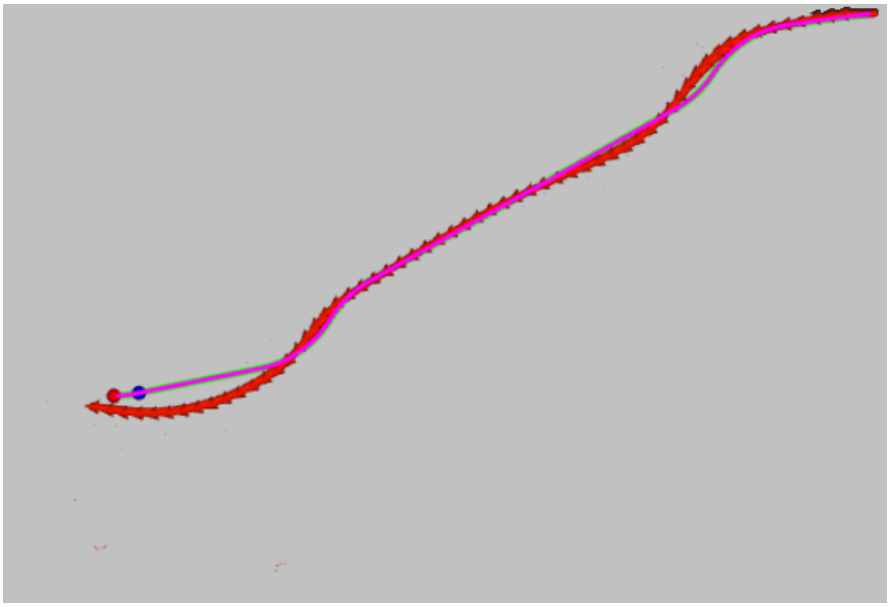

Figure 22: Path in free space

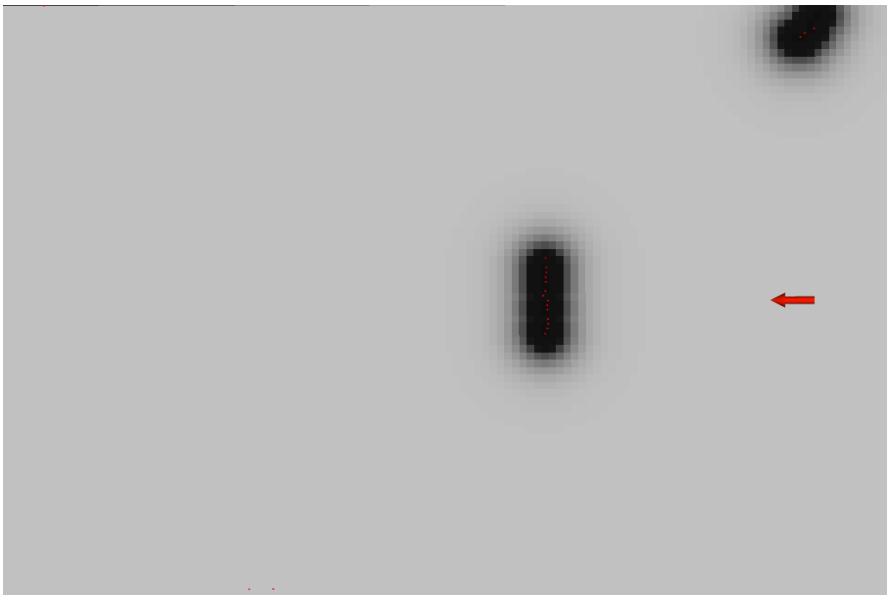

Figure 23: LIDAR detected obstacle

The robot position is not updated over time when the path is still valid. Experimental result showed that the new path is often worse than the initial path. The reason seems to be the delay between the real orientation and the measured orientation resulting in a wrong starting position. The effect is specially inconvenient in sharp turns.
We found that a sampling of 2 seconds is needed to compute a near to optimal path.

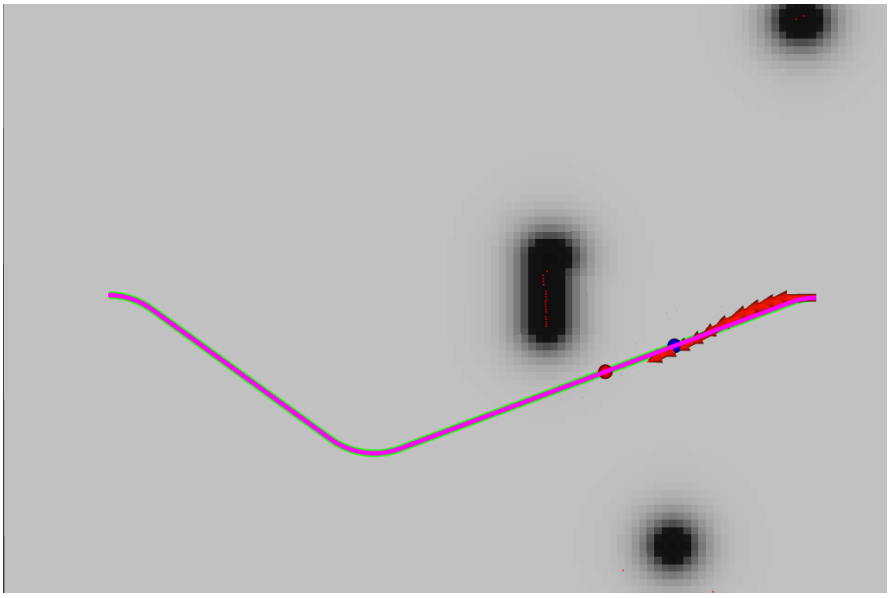

Figure 24: Planning around obstacles

We test the re-planning time by computing a path and putting an obstacle on the trajectory while the car is driving. For a sampling time of 2 second the re-planning time is around $0.9s$. Because of the selected η value and the dimension of the room more than 60% of the vertices are tested. Because of the long re-planing time the car should stop before finding a new path in order to avoid collision.
The performance of the RRT^x algorithm is therefore not good enough for real time planning ($\geq 10hz$) for our use case. While the RRT^x implementation has a time complexity of $O(n \log n)$ it is possible that the RRT^* may have less overhead per iteration than RRT^x. We will therefore test the same problem with the OMPL implementation of RRT^* to verify the hypothesis.

Using the same η value and steering constraint the RRT^* algorithm seems to find a good path in $0.5s$. Under $0.5s$ the result are inconsistent and the planner may not find any path. For our experiment we compute a new path only when the actual plan becomes invalid. Re-planning takes as long as the initial expansion ($0.5s$). The reduced planning times is small enough to avoid dynamic obstacles when driving at low speed. Driving with high speed is not possible with both algorithm as the reaction time is to long.

6.6 Interpretation and Limitations

From the experiment we can see that optimal RRT based planners are able to rapidly find a kinematically feasible path but need a long additional time for small path improvement. The $InformedRRT^*$ can be used to rapidly improve the solution by sampling only states that could potentially improve the best path. An informed sampling technique may be counterproductive on dynamic planner since the alternative paths

are no longer sampled and should therefore be implemented for the RRT^*. From experiment 5 we can also conclude that the RRT^x algorithm may be better suited for bigger spaces where small portion of the tree can be updated.

The performance difference between both algorithm may be induced by the memory access (culling neighbors, saving neighbors). It is also possible that the OMPL implementation is better optimized. After a quick look at the source code it seems that the algorithm is implementing an informed sampling method.

In the experiments above we did not plan a path over time, which would be useful in real world situation. It seems difficult to plan in time for dynamic planners because the tree is rooted at the goal. The goal must therefore be defined in time before the path is computed. Knowing the optimal duration between the actual position and the goal position is not possible before we know the optimal path, it follows that it is not possible to define the goal in time. Static planners like RRT^* are rooted at the starting position which is defined with the actual position and time. Any sampled goal in the goal coordinate can be accepted. The goal time can therefore be undefined before the expansion.
Because RRT^x cannot be used to plan in time, we think that a solution with RRT^* would be better suited to the driving problem. The RRT^* algorithm is also simpler and doesn't require to save the edges between neighbors, reducing the memory requirement.

Overall both algorithm seem to offer a good solution to find a kinematically feasible path in a unstructured environment but are too computationally heavy for real time planning. One solution could be to have a local planner that avoids obstacles by testing trajectories around the local goal at higher frequency. The RRT computed path is used to direct the local goal over the global plan.

7 Conclusion

A global planner for the AutoNOMOS model car has been presented. The planner is using a costmap as world model in order to avoid obstacles. The RRT^x and RRT^* algorithm have shown to be able to compute a kinematically feasible path for the car in an unstructured environment. With help of a motion cost function it is possible to optimize the path in order to reduce backward driving or direction changes.
The resulting path can be used as a global plan to follow but suffers from the limitation of Dubins and therefore Reeds Shepp curves. Both curves have instant steering changes that are easily reproducible when driving slowly but not when driving at high velocities. Splines can be used to smooth the path as they are continuous in the first and second derivative.
The computed tree can be corrected to react to dynamic obstacle changes in less time than the initial expansion. Rooting the tree at the goal position also allows to rapidly update the starting position without rebuilding the tree.
The drawback of the dynamic planners is the difficulty to plan in time. Since the optimal path is not known it is not possible to define the optimal goal time before the expansion. We therefore advise to look at RRT^* based solution. Improving the computational cost can be done by reducing the sampled space.
The expansion of RRT based algorithm are very efficient in unstructured environment but may not be the best choice for path planning in structured spaces like roads. When driving along a road the global plan should already be given by the GPS system and corrected through lane detection. The possible states are therefore highly reduced as the orientation has to follow the road and the optimal positions are in the middle of the lane. The difficulty in this case does not come from the path generation along the road but more from the anticipation of other moving vehicles and respecting the traffic rules. A deterministic sampling may therefore offer more consistent result.

Bibliography

[1] Open Motion Planning Library. http://ompl.kavrakilab.org.

[2] REP 105: Coordinate Frames for Mobile Platforms. http://www.ros.org/reps/rep-0105.html.

[3] Tesla. https://www.tesla.com.

[4] Waymo. https://waymo.com/.

[5] Lester E. Dubins. ON PLANE CURVES WITH CURVATURE. In *Pacific Journal of Mathematics*, 1961.

[6] Tully Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop, pages 1–6, April 2013.

[7] Steven M. LaValle James J. Kuffner. RRT-Connect: An Efficient Approach to Single-Query Path Planning.

[8] Siddhartha S. Srinivasa Jonathan D. Gammell and Timothy D. Barfoot. Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic.

[9] Sertac Karaman and Emilio Frazzoli. Sampling-based Algorithms for Optimal Motion Planning. In *International Journal of Robotics Research.*

[10] Steven M. LaValle. 13.1.2.1 a simple car. In *Planning Algorithms*.

[11] Steven M. LaValle. Rapidly Exploring Random Trees: A New Tool For Path Planning.

[12] Kai O. Arras Luigi Palmieri, Sven Koenig. RRT-Based Nonholonomic Motion Planning Using Any-Angle Path Biasing.

[13] Jean-Claude Latombe Lydia E. Kavraki, Milhail N. Kolounstzakis. Analysis of probabilistic roadmaps for path planning.

[14] JMichael Otte and Emilio Frazzoli. RRTx: Real-Time Motion Planning/Replanning for Environments with Unpredictable Obstacles. In *Massachusetts Institute of Technology, Cambridge MA 02139, USA*.

[15] J. A. REEDS and L. A. SHEPP. OPTIMAL PATHS FOR A CAR THAT GOES BOTH FORWARDS AND BACKWARDS. In *Pacific Journal of Mathematics*, 1990.

[16] Martin Törngren Sagar Behere. A functional architecture for autonomous driving.

[17] Alejandro Perez-Emilio Frazzoli Seth Teller Sertac Karaman, Matthew R. Walter. Anytime Motion Planning using the RRT$*$.

[18] Anthony Stentz. Optimal and Efficient Path Planning for Unknown and Dynamic Environments. In *The Robotics Instilute Carnegie Mellon University Pittsburgh, Pennsylvania 15213*, 1993.