Online Trajectory Planning in ROS Under Kinodynamic Constraints with Timed-Elastic-Bands

Christoph Rösmann, Frank Hoffmann and Torsten Bertram

Abstract This tutorial chapter provides a comprehensive and extensive step-by-step guide on the ROS setup of a differential-drive as well as a car-like mobile robot with the navigation stack in conjunction with the *teb_local_planner* package. It covers the theoretical foundations of the TEB local planner, package details, customization and its integration with the navigation stack and the simulation environment. This tutorial is designated for ROS Kinetic running on Ubuntu Xenial (16.04) but the examples and code also work with Indigo, Jade and is maintained in future ROS distributions.

1 Introduction

Service robotics and autonomous transportation systems require mobile robots to navigate safely and efficiently in highly dynamic environments to accomplish their tasks. This observation poses the fundamental challenge in mobile robotics to conceive universal motion planning strategies that are applicable to different robot kinematics, environments and objectives. Online planning is preferred over offline approaches due to its immediate response to changes in a dynamic environment or perturbations of the robot motion at runtime. In addition to generating a collision free path towards the goal online trajectory optimization considers secondary objectives such as control effort, control error, clearance from obstacles, trajectory length and travel time.

The authors developed a novel, efficient online trajectory optimization scheme termed Timed-Elastic-Band (TEB) in [1, 2]. The TEB efficiently optimizes the robot trajectory w.r.t. (kino-)dynamic constraints and non-holonomic kinematics while

C. Rösmann (✉) · F. Hoffmann · T. Bertram
Institute of Control Theory and Systems Engineering, TU Dortmund University, 44227 Dortmund, Germany
e-mail: christoph.roesmann@tu-dortmund.de

F. Hoffmann
e-mail: frank.hoffmann@tu-dortmund.de

T. Bertram
e-mail: torsten.bertram@tu-dortmund.de

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 707, DOI 10.1007/978-3-319-54927-9_7

explicitly incorporating temporal information in order to reach the goal pose in minimal time. The approach accounts for efficiency by exploiting the sparsity structure of the underlying optimization problem formulation. In practice, due to limited computational resources online optimization usually rests upon local optimization techniques for which convergence towards the global optimal trajectory is not guaranteed. In mobile robot navigation locally optimal trajectories emerge due to the presence of obstacles. The original TEB planner is extended in [3] to a fully integrated online trajectory planning approach that combines the exploration and simultaneous optimization of multiple admissible topologically distinctive trajectories during runtime.

The complete integrated approach is implemented as an open-source package *teb_local_planner*[1] within the Robot Operating System (ROS). The package constitutes a local planner plugin for the navigation stack.[2] Thus, it takes advantage of the features of the established mobile navigation framework in ROS, e.g. such as sharing common interfaces for robot hardware nodes, sensor data fusion and the definition of navigation tasks (by requesting navigation goals). Furthermore, it conforms to the global planning plugins available in ROS. A video that describes the package and its utilization is available online.[3] Recently, the package has been extended to accomplish navigation tasks for car-like robots (with Ackermann steering) beyond the originally considered differential-drive robots.[4] To our best knowledge, the *teb_local_planner* is currently the only local planning package for the navigation stack that explicitly supports car-like robots with limited turning radius. The main features and highlights of the planner are:

- seamless integration with the ROS navigation stack,
- general objectives for optimal trajectory planning, such as time optimality and path following,
- explicit consideration of kino-dynamic constraints,
- applicable to general non-holonomic kinematics, such as car like robots,
- explicit exploration of distinctive topologies in case of dynamic obstacles,
- computationally efficiency for online trajectory optimization.

This chapter covers the following topics:

1. the theoretical foundations of the underlying trajectory optimization method is presented (Sect. 2),
2. description of the ROS package and its integration with the navigation stack (Sect. 3),
3. package test and parameter exploration for optimization of an example trajectory (Sect. 4),
4. modeling differential-drive and car-like robots for simulation in *stage* (Sect. 5),
5. Finally, complete navigation setup of the differential-drive robot (Sect. 6) and the car-like robot (Sect. 7).

[1] *teb_local_planner*, URL: http://wiki.ros.org/teb_local_planner.

[2] ROS navigation, URL: http://wiki.ros.org/navigation.

[3] *teb_local_planner*, online-video, URL: https://youtu.be/e1Bw6JOgHME.

[4] *teb_local_planner* extensions, online-video, URL: https://youtu.be/o5wnRCzdUMo.

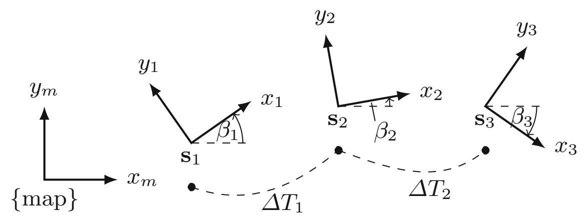


Fig. 1 Discretized trajectory with $n = 3$ poses

2 Theoretical Foundations of TEB

This section introduces and explains the fundamental concepts of the TEB optimal planner. It provides the theoretical foundations for its successful utilization and customization in own applications. For a detailed description of trajectory planning with TEB the interested reader is referred to [1, 2].

2.1 Trajectory Representation and Optimization

A discretized trajectory $\mathbf{b} = [\mathbf{s}_1, \Delta T_1, \mathbf{s}_2, \Delta T_2, \ldots, \Delta T_{N-1}, \mathbf{s}_N]^\intercal$ is represented by an ordered sequence of poses augmented with time stamps. $\mathbf{s}_k = [x_k, y_k, \beta_k]^\intercal \in \mathbb{R}^2 \times S^1$ with $k = 1, 2, \ldots, N$ denotes the pose of the robot and $\Delta T_k \in \mathbb{R}_{>0}$ with $k = 1, 2, \ldots, N-1$ represents the time interval associated with the transition between two consecutive poses $\mathbf{s}_k$ and $\mathbf{s}_{k+1}$, respectively. Figure 1 depicts an example trajectory with three poses. The reference frame of the trajectory representation and planning frame respectively is denoted as *map*-frame.[5]

Trajectory optimization seeks for a trajectory $\mathbf{b}^*$ that constitutes a minimizer of a predefined cost function. Common cost functions capture criteria such as the total transition time, energy consumption, path length and weighted combinations of those. Admissible solutions are restricted to a feasible set for which the trajectory does not intersect with obstacles or conforms to the (kino-)dynamic constraints of the mobile robot. Improving the efficiency of solving such nonlinear programs with hard constraints has become an important research topic over the past decade. The TEB approach includes constraints as soft penalty functions into the overall cost function. The introduction of soft rather than hard constraints enables the exploitation of efficient and well studied unconstrained optimization techniques for which mature open-source implementations exist.

[5]Conventions for names of common coordinate frames in ROS are listed here: http://www.ros.org/reps/rep-0105.html.

The TEB optimization problem is defined such that $\mathbf{b}^*$ minimizes a weighted and aggregated nonlinear least-squares cost function:

$$\mathbf{b}^* = \underset{\mathbf{b}\setminus\{\mathbf{s}_1,\mathbf{s}_N\}}{\arg\min} \sum_i \sigma_i f_i^2(\mathbf{b}), \quad i \in \{\mathcal{J}, \mathcal{P}\} \tag{1}$$

The terms $f_i : \mathcal{B} \to \mathbb{R}_{\geq 0}$ capture conflicting objectives and penalty functions. The set of indices associated with objectives is denoted by $\mathcal{J}$ and the set of indices that refer to penalty functions by $\mathcal{P}$. The trade-off among individual terms is determined by weights σ_i. The notation $\mathbf{b}\setminus\{\mathbf{s}_1, \mathbf{s}_N\}$ indicates that start pose $\mathbf{s}_1 = \mathbf{s}_s$ and goal pose $\mathbf{s}_N = \mathbf{s}_g$ are fixed and hence not subject to optimization. In the cost function, $\mathbf{s}_1$ and $\mathbf{s}_N$ are substituted by the current robot pose $\mathbf{s}_s$ and desired goal pose $\mathbf{s}_g$.

The TEB optimization problem (1) is represented as a hyper-graph in which poses $\mathbf{s}_k$ and time intervals ΔT_k denote the vertices of the graph and individual cost terms f_i define the (hyper-)edges. The term *hyper* indicates that an edge connects an arbitrary number of vertices, in particular temporal related poses and time intervals. The resulting hyper-graph is efficiently solved by utilizing the *g2o*-framework[6] [4]. The interested reader is referred to [2] for a detailed description on how to integrate the *g2o*-framework with the TEB approach. The formulation as hyper-graph benefits from both the direct capture of the sparsity structure for its exploitation within the optimization framework and its modularity which easily allows incorporation of additional secondary objectives f_k.

Before the individual cost terms f_k are described, the approximation of constraints by penalty functions is introduced. Let $\mathcal{B}$ denote the entire set of trajectory poses and time intervals such that $\mathbf{b} \in \mathcal{B}$. The inequality constraint $g_i(\mathbf{b}) \geq 0$ with $g_i : \mathcal{B} \to \mathbb{R}$ is approximated by a positive semi-definite penalty function $p_i : \mathcal{B} \to \mathbb{R}_{\geq 0}$ which captures the degree of violation:

$$p_i(\mathbf{b}) = \max\{\mathbf{0}, -g_i(\mathbf{b}) + \epsilon\} \tag{2}$$

The parameter ϵ adds a margin to the inequality constraint such that the cost only vanishes for $g_i(\mathbf{b}) \geq \epsilon$. Combining indices of inequality constraints g_i respectively penalty functions p_i into the set $\mathcal{P}$ results in the overall cost function (1) by assigning $f_i(\mathbf{b}) = p_i(\mathbf{b}),\ \forall i \in \mathcal{P}$. It is assumed that the choice of $g_i(\mathbf{b})$ preserves a continuous derivative (C^1-differentiability) of $p_i^2(\mathbf{b})$ and that $g_i(\mathbf{b})$ adheres to eligible monotonicity or convexity constraints.

In order to guarantee the true compliance of a solution with the constraint $g_i(\mathbf{b}) \geq 0$ by means of (2) the corresponding weights in the overall objective function (1) are required to tend towards infinity $\sigma_i \to \infty,\ \forall i \in \mathcal{P}$. For a comprehensive introduction to the theory of penalty methods the reader is referred to [5]. On the other hand, large weights prevent the underlying solver to converge properly as they cause the optimization problem to become numerically ill-conditioned. Hence, the

[6] *libg2o*, URL: http://wiki.ros.org/libg2o.

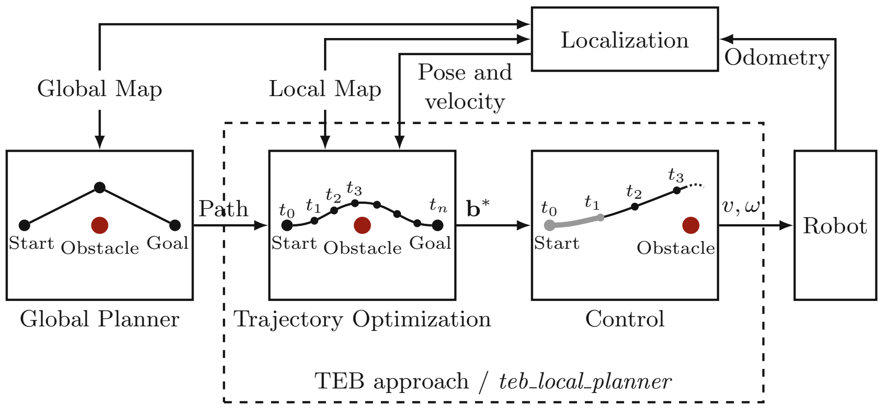


Fig. 2 System setup of a robot controlled by the TEB approach

TEB approach compensates the true minimizer with a suboptimal but computationally more efficiently obtained solution with user defined weights and the specification of an additional margin ϵ. The ROS implementation provides the parameter `penalty_epsilon` that specifies ϵ.

The TEB method utilizes multiple cost terms f_i for e.g. obstacle avoidance, compliance with (kino-)dynamic constraints of mobile robots and visiting of via-points. The list of currently implemented cost terms is described in the ROS package description in Sect. 3.3.

2.2 Closed-Loop Control

Figure 2 shows the general control architecture with the local TEB planner. The optimization scheme for (1) starts with an initial solution trajectory generated from the path provided by the global planner w.r.t. a static representation of the environment (global map). Instead of a tracking controller which regulates the motion along the planned optimal trajectory $\mathbf{b}^*$, a predictive control scheme is applied in order to account for dynamic environments encapsulated in the local map and to allow the refinement of the trajectory during runtime. Thus, optimization problem (1) is solved repeatedly w.r.t. the current robot pose and velocity. The current position of the robot is usually obtained from a localization scheme. Within each sampling interval[7] only the first control action of the TEB is commanded to the robot, which is the basic idea in model predictive control [6]. As most robots are velocity controlled by their base controllers, low-level hardware interfaces accept translational and angular velocity components w.r.t. the robot base frame. These components are easily extracted from

[7]The *move_base* node (navigation stack) provides a parameter `controller_frequency` to adjust the sampling interval.

the optimal trajectory $\mathbf{b}^*$ by investigating finite differences on the position and orientation part. Car-like robots often require the steering angle rather than angular velocity. The corresponding steering angle is calculated from the turn rate and the car-like kinematic model.

In order to improve computational efficiency the trajectory optimization pursues a warm start approach. Trajectories generated in previous iterations are reused as initial solutions in subsequent sampling intervals with updated start and goal poses. Since the time differences ΔT_k are subject to optimization the resolution of the trajectory is adjusted at each iteration according to an adaptation rule. If the resolution is too high, overly many poses increase the computational load of the optimization. On the other hand, if the resolution is too low, the finite difference approximations of quantities related to the (kino-)dynamic model of the robot are no longer accurate, causing a degradation of navigation capabilities. Therefore, the approach accounts for changing magnitudes of ΔT by regulating the resolution towards a desired temporal discretization ΔT_{ref} (ROS parameter `dt_ref`). In case of low resolution $\Delta T_k > \Delta T_{ref} + \Delta T_{hyst}$ an additional pose and time interval are filled in between $\mathbf{s}_k$ and $\mathbf{s}_{k+1}$. In case of inflated resolution $\Delta T_k < \Delta T_{ref} - \Delta T_{hyst}$ pose $\mathbf{s}_{k+1}$ is removed. The hysteresis specified by ΔT_{hyst} (ROS parameter `dt_hyst`) avoids oscillations in the number of TEB states. In case of a static goal pose this adaption implies a shrinking horizon since the overall transition time decreases as the robot advances towards to the goal.

Algorithm 1 Online TEB feedback control

```
1: procedure TEBALGORITHM($\mathbf{b}$, $\mathbf{x}_s$, $\mathbf{x}_g$, $\mathcal{O}$, $\mathcal{V}$)      ▷ Invoked each sampling interval
2:     Initialize or update trajectory
3:     for all Iterations 1 to $I_{teb}$ do
4:         Adjust length $n$ of the trajectory
5:         Build/update hyper-graph incl. association of obstacles $\mathcal{O}$ and via-points $\mathcal{V}$ with poses of
   the trajectory
6:         $\mathbf{b}^* \leftarrow$ CALLOPTIMIZER($\mathbf{b}$)      ▷ solve (1), e.g. with libg2o
7:         Check feasibility
       return First (sub-) optimal control inputs $(v_1, \omega_1)$
```

The major steps performed at each sampling interval are captured by Algorithm 1. The loop starting at line 3 is referred to as the outer optimization loop, which adjusts the length of the trajectory as described above and associates the current set of obstacles $\mathcal{O}$ and via-points $\mathcal{V}$ with their corresponding states $\mathbf{s}_k$ of the current trajectory. Further information on obstacles and via-points is provided in Sects. 3.3 and 3.5. The loop is repeated I_{teb} times (ROS parameter `no_outer_iterations`). The actual solver for optimization problem (1) is invoked in line 6 which itself performs multiple solver iterations. The corresponding ROS parameter for the number of iterations of the inner optimization loop is `no_inner_iterations`. The choice of these parameters significantly influences the required computation time as well as the convergence properties. After obtaining the optimized trajectory $\mathbf{b}^*$ a feasibility check is performed that verifies if the first M poses actually are collision free based on their original footprint model defined in the navigation stack (note, this is not the footprint model used for optimization as presented in Sect. 3.4). The verification horizon M is represented by the ROS parameter `feasibility_check_no_poses`.

2.3 Planning in Distinctive Topologies

The previously introduced TEB approach and its closed-loop application are subject to local optimization schemes which might cause the robot to get stuck in local minima. Local minima often emerge due to the presence of obstacles. Identifying those local minima coincides with analyzing distinctive topologies between start and goal poses. For instance the robot either chooses the left or right side in order to circumnavigate an obstacle. Our TEB ROS implementation investigates the discovery and optimization of multiple trajectories in distinctive topologies and selects the best candidate for control at each sampling interval. The equivalence relation presented in [7] determines whether two trajectories share the same topology. However, the configuration and theory of this extension is beyond the scope of this tutorial. The predefined default parameters are usually appropriate for applications as presented in the following sections. For further details the reader is referred to [3].

3 The teb_local_planner ROS Package

This section provides an overview about the *teb_local_planner* ROS package which implements the TEB approach for online trajectory optimization as described in Sect. 2.

3.1 Prerequisites and Installation

In order to install and configure the *teb_local_planner* package for a particular application, observe the following limitations and prerequisites:

- Although online trajectory optimization approaches pursue mature computational efficiency, their application still consumes substantial CPU resources. Depending on the desired trajectory length respectively resolution as well as the number of considered obstacles, common desktop computers or modern notebooks usually cope with the computational burden. However, older systems and embedded systems might not be capable to perform trajectory optimization at a reasonable rate.

- Results and discussions on stability and optimality properties for online trajectory optimization schemes are widespread in the literature, especially in the field of model predictive control. However, since these results are often theoretically and the planner is confronted with e.g. sensor noise and dynamic environments in real applications, finding a feasible and stable trajectory in every conceivable scenario cannot be guaranteed. However, the planner tries to detect and resolve failures to generate a feasible trajectory by post-introspection of the optimized trajectory. Its ongoing algorithmic improvement is subject to further investigations.
- The package currently supports differential-drive, car-like and omnidirectional robots. Since the planner is integrated with the navigation stack as plugin it provides a `geometry_msgs/Twist` message containing the velocity commands for controlling the robots motion. Since the original navigation stack is not intended for car-like robots yet, the additional recovery behaviors must be turned off and the global planner is expected to provide appropriate plans. However, the default global planners work well for small and medium sized car-like robots as long as the environment does not contain long and narrow passages unless the length of the vehicle exceeds their width. A conversion to steering angle has to be applied in case the car-like robot only accepts a steering angle rather than the angular velocity and interprets the `geometry_msgs/Twist` or `ackermann_msgs/AckermannDriveStamped` message different from the nominal convention. The former is directly enabled (see Sect. 6) and the latter requires a dedicated conversion ROS node.
- The oldest officially supported ROS distribution is Indigo. At the time of writing the planner is also available in Jade and Kinetic. Support of future distributions is expected. The package is released for both default and ARM architectures.
- Completion of the common ROS beginner tutorials, e.g. being aware of navigating the filesystem, creating and building packages as well as dealing with *rviz*,[8] launch files, topics, parameters and *yaml* files is essential. Experiences with the navigation stack are highly recommended. The user should be familiar with concepts and components of ROS navigation such as local and global costmaps and local and global planners (*move_base* node), coordinate transforms, odometry and localization. This tutorial outlines the configuration of a complete navigation setup. However, explanation of the underlying concepts in detail is beyond the scope of this chapter. Tutorials on ROS navigation are available at the wiki page[2] and [8].
- Table 1 provides an overview of currently available local planners for the ROS navigation stack and summarizes its main features.

The *teb_local_planner* is easily installed from the official ROS repositories by invoking in *terminal*:

```
$ sudo apt-get install ros-kinetic-teb-local-planner
```

[8] *rviz*, URL: http://wiki.ros.org/rviz.

Table 1 Comparison of available local planners in the ROS navigation stack

	EBand[a]	TEB	DWA[b]
Strategy	Force-based path deformation and path following controller	Continuous trajectory optimization resp. predictive controller	Sampling-based trajectory generation, predictive controller
Optimality	Shortest path without considering kinodynamic constraints (local solutions)	Time-optimal (or ref. path fidelity) with kinodynamic constraints (multiple local solutions, parallel optimization)	Time-sub-optimal with kinodynamic constraints, samples of trajectories with constant curvature for prediction (multiple local solutions)
Kinematics	Omnidirectional and differential-drive robots	Omnidirectional, differential-drive and car-like robots	Omnidirectional and differential-drive robots
Computational burden	Medium	High	Low/Medium

[a] *eband_local_planner*, URL: http://wiki.ros.org/eband_local_planner
[b] *TrajectoryPlannerROS*, URL: http://wiki.ros.org/base_local_planner

The distribution name `kinetic` might be adapted to match the currently installed one. In the following, terminal commands are usually indicated by a leading $-sign. As an alternative to the default package installation, recent versions (albeit experimental) can be obtained and compiled from source:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/rst-tu-dortmund/
    teb_local_planner.git  --branch kinetic-devel
$ cd ../
$ rosdep install --from-paths src --ignore-src --rosdistro
    kinetic -y
$ catkin_make
```

Hereby, it is assumed that `~/catkin_ws` points to the user-created *catkin* workspace.

3.2 *Integration with ROS Navigation*

The *teb_local_planner* package seamlessly integrates with the ROS navigation stack since it complies with the interface `nav_core::BaseLocalPlanner` specified in the *nav_core*[2] package. Figure 3 shows an overview of the main components that constitute the navigation stack and the *move_base* node respectively.[9] The *move_base* node takes care about the combination of the global and local planner as well as

[9]Adopted from the *move_base* wiki page, URL: http://wiki.ros.org/move_base.

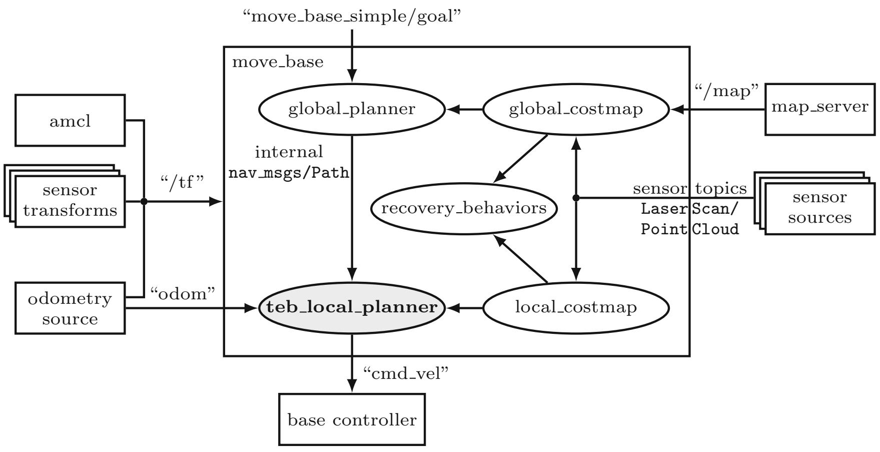


Fig. 3 ROS navigation component-view including *teb_local_planner*

handling costmaps for obstacle avoidance. The *amcl* node[10] provides an adaptive monte carlo localization algorithm which corrects the accumulated odometric error and localizes the robot w.r.t. the global map. For the following tutorials the reader is expected to be familiar with the navigation stack components and the corresponding topics.

The *teb_local_planner* package comes with its own parameters which are configurable by means of the parameter server. The full list of parameters is available on the package wiki page, but many of them are presented and described in this tutorial. Parameters are set according to the relative namespace of *move_base*, e.g. `/move_base/TebLocalPlannerROS/param_name`. Most of the parameters can also be configured during runtime with *rqt_reconfigure* which is instantiated as follows (assuming a running planner instance):

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Within each local planner invocation (respectively each sampling interval) the *teb_local_planner* chooses an intermediate virtual goal within a specified lookahead distance on the current global plan. Only the local stretch of the global plan between current pose and lookahead point is subject to trajectory optimization by means of Algorithm 1. Hence the lookahead distance implies a receding horizon control strategy which transits to a shrinking horizon once the virtual goal coincides with the final goal pose of the global plan. The lookahead distance to the virtual goal is set by parameter `max_global_plan_lookahead_dist` but the virtual goal is never located beyond the boundaries of the local costmap.

[10]*amcl*, URL: http://wiki.ros.org/amcl.

3.3 Included Cost Terms: Objectives and Penalties

The *teb_local_planner* determines the current control commands in terms of minimizing the future trajectory w.r.t. a specified cost function (1) which itself consists of aggregated objectives and penalty terms as described in Sect. 2. Currently implemented cost terms f_i of the optimization problem (1) are summarized in the following overview including their corresponding ROS parameters such as the optimization weights σ_i.

Limiting translational velocity (Penalty)
Description: Constrains the translational velocity v_k to the interval $[-v_{back}, v_{max}]$. v_k is computed with $\mathbf{s}_k$, $\mathbf{s}_{k+1}$ and ΔT_k using finite differences.
Weight parameter: `weight_max_vel_x`
Additional parameters: `max_vel_x` (v_{max}), `max_vel_x_backwards` (v_{back})

Limiting angular velocity (Penalty)
Description: Constrains the angular velocity to $|\omega_k| \leq \omega_{max}$ (finite differences).
Weight parameter: `weight_max_vel_theta`
Additional parameters: `max_vel_theta` (ω_{max})

Limiting translational acceleration (Penalty)
Description: Constrains the translational acceleration to $|a_k| \leq a_{max}$ (finite differences).
Weight parameter: `weight_acc_lim_x`
Additional parameters: `acc_lim_x` (a_{max})

Limiting angular acceleration (Penalty)
Description: Constrains the angular acceleration to $|\dot{\omega}_k| \leq \dot{\omega}_{max}$ (finite differences).
Weight parameter: `weight_acc_lim_theta`
Additional parameters: `acc_lim_theta` ($\dot{\omega}_{max}$)

Compliance with non-holonomic kinematics (Objective)
Description: Minimize deviations from the geometric constraint that requires two consecutive poses $\mathbf{s}_k$ and $\mathbf{s}_{k+1}$ to be located on a common arc of constant curvature. Actually, kinematic compliance is not merely an objective, but rather an equality constraint. However, since as the planner rests upon unconstrained optimization a sufficient compliance is ensured by a large weight.
Weight parameter: `weight_kinematics_nh`

Limiting the minimum turning radius (Penalty)
Description: Some mobile robots exhibit a non-zero turning radius (e.g. implicated by a limited steering angle). In particular car-like robots are unable to rotate in place. This penalty term enforces $r = \frac{v_k}{\omega_k} \geq r_{min}$. Differential drive and unicycle robots can turn in place $r_{min} = 0$.
Weight parameter: `weight_kinematics_turning_radius`
Additional parameters: `min_turning_radius` (r_{min})

Penalizing backwards motions (Penalty)
Description: This cost term expresses preference for forward motions independent of the actual maximum backward velocity v_{back} in terms of a bias weight. The penalty is deactivated if `min_turning_radius` is non-zero.
Weight parameter: `weight_kinematics_forward_drive`

Obstacle avoidance (Penalty)
Description: This cost term maintains a minimum separation d_{min} of the trajectory from obstacles. A dedicated robot footprint model is taken into account for distance calculation (see Sect. 3.3).
Weight parameter: `weight_obstacle`
Additional parameters: `min_obstacle_dist` (d_{min})

Via-points (Objective)
Description: This cost term minimizes the distance to via-points, e.g. located along the global plan. Each via-point defines an attractor for the planned trajectory.
Weight parameter: `weight_viapoint`
Additional parameters: `global_plan_viapoint_sep`

Arrival at the goal in minimum time (Objective)
Description: This term minimizes ΔT_k in order seek for a time-optimal trajectory.
Weight parameter: `weight_optimaltime`.

3.4 *Robot Footprint for Optimization*

The obstacle avoidance penalty function introduced in Sect. 3.3 depends on a dedicated robot footprint model. The reason behind not using the original footprint specified in the navigation stack resp. costmap configuration is to promote efficiency in the optimization formulation while keeping the original footprint for feasibility checks. Since the optimization scheme is subject to a large number of distance calculations between robot and obstacles, the original polygonal footprint would drastically increase the computational load, as each polygon edge has to be taken into account. However, the user might still duplicate the original footprint model for optimization, but in practice simpler approximations are often sufficient. The current package version provides four different models (see Fig. 4). Parameters for defining the footprint model (as listed below) are defined w.r.t. the robot base frame, e.g. `base_link`, such that $\mathbf{s}_k$ defines its origin. An example robot frame is depicted in Fig. 4d. In particular, the four models are:

- **Point model**: The most efficient representation in which the robot is modeled as a single point. The robot's radial extension is captured by inflating the minimum distance from obstacles d_{min} (`min_obstacle_dist`, see Fig. 4a) by the robot's radius.
- **Line model**: The line model is ideal for robots which dimensions differ in longitudinal and lateral directions. Start and end of the underlying line segment, $\mathbf{l}_s \in \mathbb{R}^2$

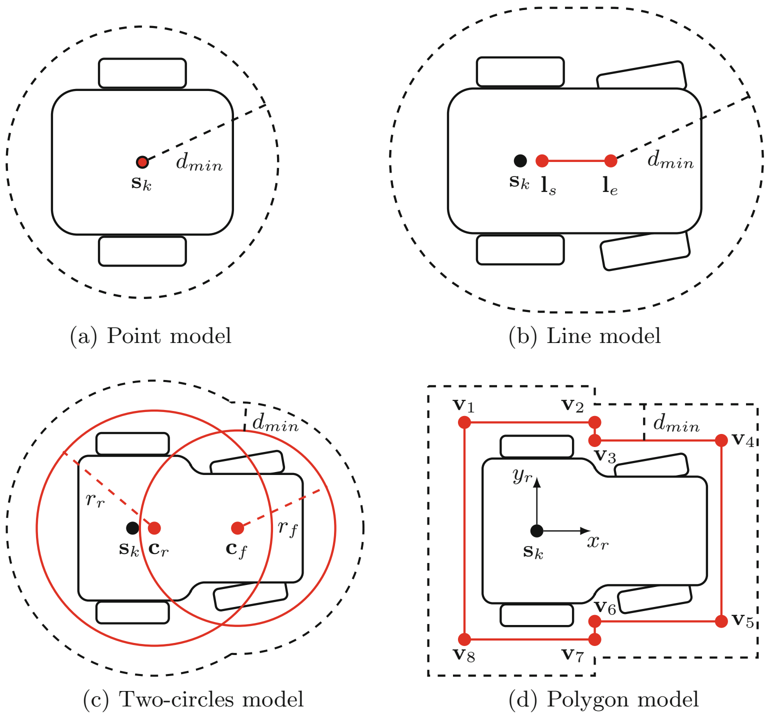

d_{min}
$\mathbf{s}_k$
(a) Point model
d_{min}
$\mathbf{s}_k$ $\mathbf{l}_s$ $\mathbf{l}_e$
(b) Line model
d_{min}
r_r
r_f
$\mathbf{s}_k$ $\mathbf{c}_r$ $\mathbf{c}_f$
(c) Two-circles model
$\mathbf{v}_1$ $\mathbf{v}_2$ $\mathbf{v}_3$ $\mathbf{v}_4$ $\mathbf{v}_5$ $\mathbf{v}_6$ $\mathbf{v}_7$ $\mathbf{v}_8$
d_{min}
y_r x_r
$\mathbf{s}_k$
(d) Polygon model

Fig. 4 Available footprint models for optimization

and $\mathbf{l}_e \in \mathbb{R}^2$ respectively, are arbitrary w.r.t. the robot's center of rotation $\mathbf{s}_k$ as origin $(0, 0)$. The robot's radial extension is controlled similar to the point model by inflation of d_{min} (refer to Fig. 4b).

- **Two-circle model**: The two-circle model is suited for robots that exhibit a more cone-shaped footprint rather than a rectangular one (see Fig. 4c). The centers of both circles, $\mathbf{c}_r$ and $\mathbf{c}_f$ respectively, are restricted to be located on the robot's x-axis. Their offsets w.r.t. the center of rotation $\mathbf{s}_k$ and their radii r_r and r_f are arbitrary.
- **Polygon model**: The polygon model is the most general one, since the number of edges is arbitrary. Figure 4d depicts a footprint defined by 8 vertices $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_8 \in \mathbb{R}^2$. The polygon is automatically completed by adding an edge between the first and last vertex.

The following *yaml* file contains example parameter values for customizing the footprint. All parameters are defined w.r.t. the local planner namespace `TebLocalPlannerROS`:

```
TebLocalPlannerROS:
 footprint_model:
   type: "point" # types: "point", "line", "two_circles",
     "polygon"
   line_start: [-0.3, 0.0] # for type "line"
   line_end: [0.3, 0.0] # for type "line"
   front_offset: 0.2 # for type "two_circles"
   front_radius: 0.2 # for type "two_circles"
   rear_offset: 0.2 # for type "two_circles"
   rear_radius: 0.2 # for type "two_circles"
   vertices: [ [-0.1,0.2], [0.2,0.2], [0.2,-0.2], [-0.1,-0.2] ]
     # for type "polygon"
```

3.5 Obstacle Representations

The *teb_local_planner* takes the local costmap into account as already stated in Sect. 3.2. The costmap mainly consists of a grid in which each cell stores an 8-bit cost value that determines whether the cell is free (0), unknown, undesired or occupied (255). Besides, the ability to implement multiple layers and to fuse data from different sensor sources, the costmap is perfectly suited for local planners in the navigation stack due to their sampling based nature. In contrast, the TEB optimization problem (1) cannot just test discrete cell states inside its own cost function for collisions, but rather requires continuous functions based on the distance to obstacles. Therefore, our implementation extracts relevant obstacles from the current costmap at the beginning of each sampling interval and considers each occupied cell as single dimensionless point-shaped obstacle. Hence, the computation time strongly depends on the local costmap size and resolution (see Sect. 6). Additionally, custom obstacles can be provided by a specific topic. The *teb_local_planner* supports obstacle representations in terms of points, lines and closed polygons. A costmap conversion[11] might be activated in order to convert costmap cells into primitive types such as lines and polygons in a separate thread. However, these extensions are beyond the scope of this introductory tutorial, but the interested reader is referred to the package wiki page.

Once obstacles are extracted and cost terms (hyper-edges) according to Sect. 3.3 are constructed, obstacles are associated with the discrete poses $\mathbf{s}_k$ of the trajectory in order to maintain a minimal separation.

In order to speed up the time spent by the solver for computing the cost function multiple times during optimization, each pose $\mathbf{s}_k$ is only associated with its nearest obstacles. The association is renewed at every outer iteration (refer to Algorithm 1) in order to correct vague associations during convergence. For advanced parameters of the association strategy the reader is referred to the *teb_local_planner* ROS wiki page. Figure 5 depicts an example planning scenario in which the three

[11] *costmap_converter*, URL: http://wiki.ros.org/costmap_converter.

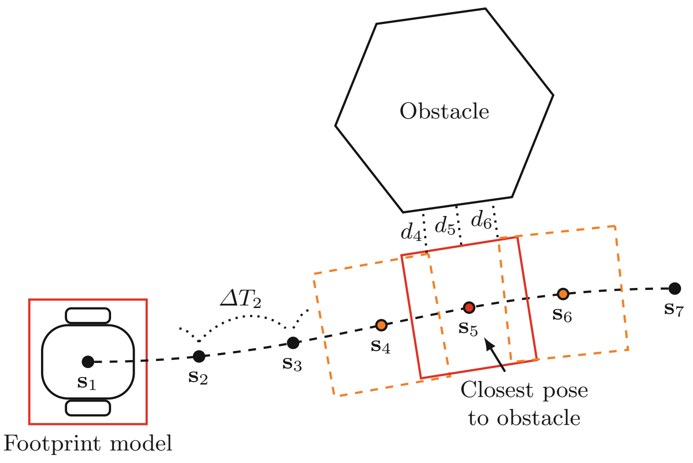


Fig. 5 Association between poses and obstacles

closest poses are associated with the polygonal obstacle. Notice, the minimum distances d_4, d_5 and d_6 to the robot footprints located at $\mathbf{s}_4$, $\mathbf{s}_5$ and $\mathbf{s}_6$ are constrained to `min_obstacle_dist`. The minimum distance should account for an additional safety margin around the robot, since the penalty functions cannot guarantee its fulfillment and small violations might cause a rejection of the trajectory by the feasibility check (refer to Sect. 2).

4 Testing Trajectory Optimization

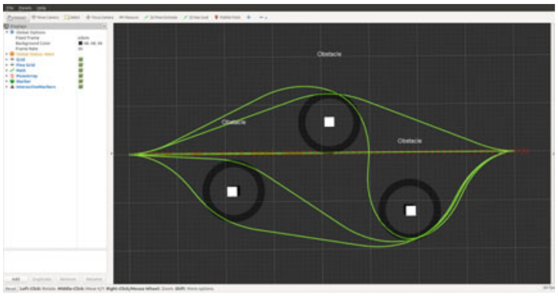
Before starting with the configuration process of the *teb_local_planner* for a particular robot and application, we recommend the reader to familiarize himself with the optimization process and furthermore to check the performance on the target hardware. The package includes a simple test node (*test_optim_node*) that optimizes a trajectory between a fixed start and goal pose. Some obstacles are included with interactive markers[12] which are conveniently moved via the GUI in *rviz*.

Launch the *test_optim_node* in combination with a preconfigured *rviz* node as follows:

```
$ roslaunch teb_local_planner test_optim_node.launch
```

An *rviz* window should open showing the trajectory and obstacles. Select the menu button *Interact* in order to move the obstacles around. An example setting is depicted in Fig. 6a. As briefly stated in Sect. 2, the package generates and optimizes trajectories in different topologies in parallel. The currently selected trajectory for navigation is

[12]*interactive_markers*, URL: http://wiki.ros.org/interactive_markers.



(a) *test_optim_node* and its visualization

(b) *rqt_reconfigure* window

Fig. 6 Testing trajectory optimization with *test_optim_node*

augmented with red pose arrows in visualization. In order to change parameters during runtime, invoke

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

in a new terminal window and select *test_optim_node* from the list of available nodes (refer to Fig. 6b). Try to customize the optimization with different parameter settings. Since some parameters significantly influence the optimization result, adjustments should be performed slightly and in a step-by-step manner. In case you encounter a poor performance on your target system even with the default settings, try to decrease parameters `no_inner_iterations`, `no_outer_iterations` or increase `dt_ref` slightly.

5 Creating a Mobile Robot in Stage Simulator

This section introduces a minimal *stage*[13] simulation setup with a differential-drive and a car-like robot. *stage* is chosen for this tutorial since it constitutes a package which is commonly used in ROS tutorials and is thus expected to be available in future ROS distributions as well. Furthermore, *stage* is fast and lightweight in terms of visualization which allows its execution even on slow CPUs and older graphic cards. It supports kinematic models for differential-drive, car-like and holonomic robots, but it is not intended to perform dynamic simulations such as *Gazebo*.[14] However, even if the following sections refer to a stage model, the procedures and configurations are directly applicable to other simulation environments or a real mobile robot without major modifications.

[13]*stage_ros*, URL: http://wiki.ros.org/stage_ros.

[14]*gazebo_ros_pkgs*, URL: http://wiki.ros.org/gazebo_ros_pkgs.

Note *stage* (resp. *stage_ros*) publishes the coordinate transformation between `odom` and `base_link` and the odometry information to the `odom` topic. It subscribes to velocity commands by the topic `cmd_vel`.

Make sure to install *stage* for your particular ROS distribution:

```
$ sudo apt-get install ros-kinetic-stage-ros
```

In order to gather all configuration and launch files that will be created during the tutorial, a new package is initiated as follows:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg teb_tutorial
```

It is a good practice to add *teb_local_planner* and *stage_ros* to the run dependencies of your newly created package (check the new *package.xml* file). Now, create a *stage* and a *maps* folder inside the *teb_tutorial* package and download a predefined map called *maze.png*[15] and its *yaml* configuration for the *map_server*:

```
$ roscd teb_tutorial
$ mkdir stage && mkdir maps
$ cd maps
# remove any whitespaces in the URLs below after copying
$ wget https://cdn.rawgit.com/rst-tu-dortmund/
    teb_local_planner_tutorials/rosbook/maps/maze.png
$ wget https://cdn.rawgit.com/rst-tu-dortmund/
    teb_local_planner_tutorials/rosbook/maps/maze.yaml
```

5.1 *Differential-Drive Robot*

Stage loads its environment from world files that define a static map and agents such like your robot (in plain text format). In the following, we add a world file to the *teb_tutorial* package which loads the map *maze.png* and spawns a differential-drive robot. The robot is assumed to be represented as a box ($0.25\,\text{m} \times 0.25\,\text{m} \times 0.4\,\text{m}$). Whenever text files are edited, the editor *gedit* is utilized (`sudo apt-get install gedit`), but you might employ the editor of your preferred choice.

```
$ roscd teb_tutorial/stage
$ gedit maze_diff_drive.world # or use the editor of your
    choice
```

The second command creates and opens a new file with *gedit*. Add the following code and save the contents to file *maze_diff_drive.world*:

[15]Borrowed from the *turtlebot_stage* package: http://wiki.ros.org/turtlebot_stage.

```
   ## Simulation settings
 2 resolution 0.02
   interval_sim 100  # simulation timestep in milliseconds
 4 ## Load a static map
   model(
 6   name "maze"
     bitmap "../maps/maze.png"
 8   size [ 10.0 10.0 2.0 ]
     pose [ 5.0  5.0 0.0 0.0 ]
10   color "gray30"
   )
12 ## Definition of a laser range finder
   define mylaser ranger(
14   sensor(
       range_max 6.5 # maximum range
16     fov 58.0 # field of view
       samples 640 # number of samples
18   )
     size [ 0.06 0.15 0.03 ]
20 )
   ## Spawn robot
22 position(
     name "robot"
24   size [ 0.25 0.25 0.40 ] # (x,y,z)
     drive "diff" # kinematic model of a differential-drive robot
26   mylaser(pose [ -0.1 0.0 -0.11 0.0 ]) # spawn laser sensor
     pose [ 2.0 2.0 0.0 0.0 ] # initial pose (x,y,z,beta[deg])
28 )
```

General simulation settings are defined in lines 1–3. Afterwards the static map is defined using a *stage model* object. The size property is important in order to define the transformation between pixels of *maze.png* and their actual sizes in the world. The bitmap is shifted by an offset defined in *pose* in order to adjust the bitmap position relative to the `map` frame. Simulated robots in this tutorial are equipped with a laser range finder set up in lines 12–20. Finally, the robot itself is setup in lines 22–28. The code specifies a differential drive robot (`drive "diff"`) with the previously defined laser scanner attached and the desired box size as well as the initial pose. The `base_link` frame is automatically located in the geometric center of the box, specified by the `size` parameter, which in this case coincides with the center of rotation. A case in which the origin must be corrected occurs for the car-like model (see Sect. 5.2).

In order to test the created robot model invoke the following commands in a terminal:

```
$ roscore
$ rosrun stage_ros stageros `rospack find teb_tutorial`/stage/
    maze_diff_drive.world
```

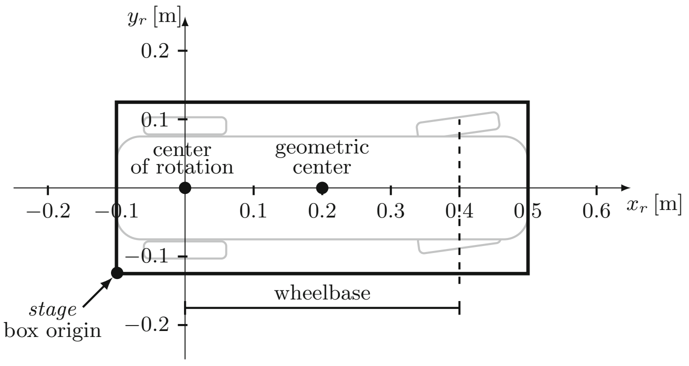


Fig. 7 Dimensions of the car-like robot for simulation

5.2 Car-Like Robot

In this section you generate a second world file that spawns a car-like robot. The 2D-contours of the car-like robot are depicted in Fig. 7 (gray-colored). For the purpose of this tutorial, only the boundary box of the robot is considered in the simple *stage* model. The length in y_r is increased slightly in order to account for the steerable front wheels. The top left and bottom right corners are located at $\mathbf{v}_{tl} = [-0.1,\ 0.125]^T$ m and $\mathbf{v}_{br} = [0.5,\ -0.125]^T$ m respectively w.r.t. the robot's base frame `base_link` (defined by the x_r- and y_r-axis). For car-like robots with front steering wheels the center of rotation coincides with the center of the rear axle. Since the TEB approach assumes a unicycle model for planning but with additional constraints for car-like robots such as minimum turning radius, **the robot's base frame must be placed at the center of rotation** in order to fulfill this relation.

The next step consist of duplicating the previous world file from Sect. 5.1 and modifying the robot model according to Fig. 7:

```
$ roscd teb_tutorial/stage
$ cp maze_diff_drive.world maze_carlike.world # duplicate
    diffdrive.world
$ gedit maze_carlike.world # or use the editor of your choice
```

Replace the robot model (line 21–28) by the following one and save the file:

```
## Spawn robot
position(
  name "robot"
  size [ 0.6 0.25 0.40 ] # (x,y,z) - bounding box of the robot
  origin [ 0.2 0.0 0.0 0.0] # correct center of rotation (x,y,z
    ,beta)
  drive "car" # kinematic model of a car-like robot
  wheelbase 0.4 # distance between rear and front axles
  mylaser(pose [ -0.1 0.0 -0.11 0.0 ]) # spawn laser sensor
  pose [ 2.0 2.0 0.0 0.0 ] # initial pose (x,y,z,beta[deg])
)
```

Notice, the kinematic model is changed to the car-like one (`drive "car"`). Parameter `wheelbase` denotes the distance between the rear and front axle (see Fig. 7). The size of the robot's bounding box is set in line 4 w.r.t. the box origin as depicted in Fig. 7. *Stage* automatically defines the center of rotation in the geometric center, which is located at $[0.3,\ 0.125]^T$ m w.r.t. the box origin. In order to move the center of rotation towards the correct location $[0.1,\ 0.125]^T$ m w.r.t. the box origin, the frame is shifted as specified by parameter `origin`. Load and inspect your robot model in stage for testing purposes:

```
$ roscore
$ rosrun stage_ros stageros `rospack find teb_tutorial`/stage/maze_carlike.world
```

The robot is controlled via a `geometry_msgs/Twist` message even though the actual kinematics refer to a car-like robot. But in contrast to the differentential-drive robot, the angular velocity (yaw-speed, around z-axis) is interpreted as steering angle rather than the true velocity component.

6 Planning for a Differential-Drive Robot

This section covers the complete navigation setup with the *teb_local_planner* for the differential-drive robot defined in Sect. 5.1. Start by creating configuration files for the global and local costmap (refer to Fig. 3). In the following, configuration files are stored in a separate *cfg* folder inside your *teb_tutorial* package which was created during the steps in Sect. 5.

Create a *costmap_common_params.yaml* file which contains parameters for both the global and local costmap:

```
$ roscd teb_tutorial
$ mkdir cfg && cd cfg
$ gedit costmap_common_params.yaml
```

Now insert the following lines and save the file afterwards:

```
# file: costmap_common_params.yaml
# Make sure to preserve indentation if copied (for all yaml files)
footprint: [  [-0.125,0.125], [0.125,0.125], [0.125,-0.125], [-0.125,-0.125]  ]

transform_tolerance: 0.5
map_type: costmap
global_frame: /map
robot_base_frame: base_link

obstacle_layer:
  enabled: true
  obstacle_range: 3.0
  raytrace_range: 4.0
```

```
14    track_unknown_space: true
      combination_method: 1
16    observation_sources: laser_scan_sensor
      laser_scan_sensor: {data_type: LaserScan, topic: scan,
        marking: true, clearing: true}
18
    inflation_layer:
20    enabled:              true
      inflation_radius:     0.5
22
    static_layer:
24    enabled:              true
```

The robot footprint is specified according to the projection of the dimensions of the robot (0.25 m × 0.25 m × 0.4 m) introduced in Sect. 6 onto the x-y-plane. The footprint must be defined in the `base_link` frame, which center coincides with the center of rotation. In this tutorial the selected `map_type` is `costmap` which creates an internal 2d grid. If the robot is equipped with 3D range sensors it is often desired to include the height of obstacles. This allows for ignoring obstacles beyond a specific height or tiny obstacles above the ground floor. For this purpose, the *costmap_2d*[16] package also supports voxel grids. Refer to the *costmap_2d* wiki page for further information.

The obstacle layer is defined in lines 9–16 which includes external sensors such like our laser range finder by implementing ray-tracing. In this tutorial the laser range finder is expected to publish its range data on topic `scan`.

An inflation layer adds exponentially decreasing cost to cells w.r.t. their distance from actual (lethal) obstacles. This allows the user to set a preference for maintaining larger separation from obstacles whenever possible. Although the *teb_local_planner* only extracts lethal obstacles from the costmap as described in Sect. 3.5 and ignores inflation, an activated inflation layer still influences the global planner and thus the location of virtual goals for local planning (refer to Sect. 3.2 for the description of virtual goals). Consequently, a non-zero `inflation_radius` moves virtual goals further away from (static) obstacles. Finally, the static layer includes obstacles from the static map which are retrieved from the `map` topic by default. The map *maze.png* is published later by the *map_server* node.

After saving and closing the file, specific configurations for the global and local costmap are created.

```
$ roscd teb_tutorial/cfg
$ gedit global_costmap_params.yaml # insert content, save and
    close
$ gedit local_costmap_params.yaml # insert content, save and
    close
```

[16] *costmap_2d*, URL: http://wiki.ros.org/costmap_2d.

The default content for `global_costmap_params.yaml` is listed below:

```
# file: global_costmap_params.yaml
global_costmap:
  update_frequency: 1.0
  publish_frequency: 0.5
  static_map: true
  plugins:
    - {name: static_layer,    type: "costmap_2d::StaticLayer"}
    - {name: inflation_layer, type: "costmap_2d::InflationLayer
    "}
```

The global costmap is intended to be a static one which means its size is inherited from the map provided by the `map_server` node (notice the loaded static layer plugin which was defined in `costmap_common_params.yaml`). The previously defined inflation layer is added as well.

The content for `local_costmap_params.yaml` is as follows:

```
# file: local_costmap_params.yaml
local_costmap:
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 5.5       # -> computation time: teb_local_planner
  height: 5.5      # -> computation time: teb_local_planner
  resolution: 0.1  # -> computation time: teb_local_planner
  plugins:
     - {name: obstacle_layer,  type: "costmap_2d::ObstacleLayer
    "}
```

It is highly recommended to define the local costmap as a rolling window in medium or large environments, since otherwise the implied huge number of obstacles might lead to intractable computational loads. The rolling window is specified by its width, height and resolution. These parameters have a significant impact on the computation time of the planner. The size should not exceed the local sensor range and it is often sufficient to set the width and height to values of approx. 5–6 m. The resolution determines the discretization granularity respectively how many grid cells are allocated in order to represent the rolling window. Since each occupied cell is treated as a single obstacle by default (see Sect. 3.5), a small value (resp. high resolution) indicates a huge number of obstacles and therefore long computation times. On the other hand, the resolution must be fine enough to cope with small obstacles, narrow hallways and passing doors. Finally, the previously defined obstacle layer is activated in order to incorporated dynamic obstacles obtained from the laser range finder.

Prior to generating the overall launch file, a configuration file for the local planner is created:

```
$ roscd teb_tutorial/cfg
$ gedit teb_local_planner_params.yaml
```

The content of the `teb_local_planner_params.yaml` is listed below:

```
# file: teb_local_planner_params.yaml
TebLocalPlannerROS:

 # Trajectory
 dt_ref: 0.3
 dt_hysteresis: 0.1
 global_plan_overwrite_orientation: True
 allow_init_with_backwards_motion: False
 max_global_plan_lookahead_dist: 3.0
 feasibility_check_no_poses: 3

 # Robot
 max_vel_x: 0.4
 max_vel_x_backwards: 0.2
 max_vel_theta: 0.3
 acc_lim_x: 0.5
 acc_lim_theta: 0.5
 min_turning_radius: 0.0 # diff-drive robot (can turn in place
    !)
 footprint_model:
   type: "point" # include robot radius in min_obstacle_dist

 # Goal Tolerance
 xy_goal_tolerance: 0.2
 yaw_goal_tolerance: 0.1

 # Obstacles
 min_obstacle_dist: 0.25
 costmap_obstacles_behind_robot_dist: 1.0
 obstacle_poses_affected: 10

 # Optimization
 no_inner_iterations: 5
 no_outer_iterations: 4
```

For the sake of readability, only a small subset of available parameters is defined here. Feel free to add other parameters, e.g. after determining suitable parameter sets with *rqt_reconfigure*. In this example configuration, the point footprint model is chosen for optimization (parameter `footprint_model`). The circumscribed radius R of the robot defined in Sect. 5.1 is derived by applying geometry calculus: $R = 0.5\sqrt{0.25^2 + 0.25^2}\,\text{m} \approx 0.18\,\text{m}$. In order to compensate possible small distance violations due to penalty terms the parameter `min_obstacle_dist` is set to 0.25 m.

Parameter `costmap_obstacles_behind_robot_dist` specifies how many meters of the local costmap portion beyond the robot are taken into account in order to extract obstacles from cells.

After all related configuration files are created, the next step consist of defining the launch file that starts all nodes required for navigation and includes configuration files. Launch files are usually added to a subfolder called *launch*:

```
$ cd ~/catkin_ws/src/teb_tutorial/
$ mkdir launch && cd launch
$ gedit robot_in_stage.launch # create a new launch file
```

Add the following content to your newly created launch file:

```
<!-- file: robot_in_stage.launch -->
<launch>
<!--  ************** Global Parameters ***************  -->
<param name="/use_sim_time" value="true"/>

<!--  ************** Stage Simulator ***************  -->
<node pkg="stage_ros" type="stageros" name="stageros" args="$(
    find teb_tutorial)/stage/maze_diff_drive.world">
    <remap from="base_scan" to="scan"/>
</node>

<!--  ************** Navigation ***************  -->
<node pkg="move_base" type="move_base" respawn="false" name="
    move_base" output="screen">
    <rosparam file="$(find teb_tutorial)/cfg/
    costmap_common_params.yaml" command="load" ns="
    global_costmap"/>
    <rosparam file="$(find teb_tutorial)/cfg/
    costmap_common_params.yaml" command="load" ns="
    local_costmap"/>
    <rosparam file="$(find teb_tutorial)/cfg/
    local_costmap_params.yaml" command="load" />
    <rosparam file="$(find teb_tutorial)/cfg/
    global_costmap_params.yaml" command="load" />
    <rosparam file="$(find teb_tutorial)/cfg/
    teb_local_planner_params.yaml" command="load" />

    <param name="base_global_planner" value="global_planner/
    GlobalPlanner"/>
    <param name="planner_frequency"  value="1.0" />
    <param name="planner_patience"   value="5.0" />

    <param name="base_local_planner" value="teb_local_planner/
    TebLocalPlannerROS"/>
    <param name="controller_frequency"  value="5.0" />
    <param name="controller_patience"   value="15.0" />
</node>


<!--  ****** Maps *****  -->
<node name="map_server" pkg="map_server" type="map_server" args
    ="$(find teb_tutorial)/maps/maze.yaml" output="screen">
    <param name="frame_id" value="/map"/>
</node>

<!--  ****** AMCL *****  -->
<node pkg="amcl" type="amcl" name="amcl" output="screen">
    <param name="initial_pose_x"            value="2"/>
    <param name="initial_pose_y"            value="2"/>
    <param name="initial_pose_a"            value="0"/>
    <param name="odom_model_type"           value="diff"/>
    <param name="use_map_topic"             value="true"/>
    <param name="transform_tolerance"       value="0.5"/>
    </node>
</launch>
```

After activating simulation time, the *stage_ros* node is loaded. The path to the world file previously created in Sect. 5.1 is forwarded as an additional argument. The command `$(find teb_tutorial)` automatically searches for the *teb_tutorial* package path in your workspace. Since *stage_ros* publishes the simulated laser range data on topic `base_scan`, but the costmap is configured for listening on `scan`, remapping is performed here.

Afterwards, the core navigation node *move_base* is loaded. All costmap and planner parameters are included relative to the *move_base* namespace. Some additional parameters are defined, such as which local and global planner plugins to be loaded. The selected global planner is commonly used in ROS. Parameters `planner_frequency` and `planner_patience` define the rate (in Hz) at which the global planner is invoked and how long the planner waits (seconds) without receiving any valid control before backup operations are performed, respectively. Similar settings are applied to the local planner with parameters `controller_frequency` and `controller_patience`. We also specify the `teb_local_planner/TebLocalPlannerROS` as the local planner plugin.

The two final nodes are the *map_server* node which provides the maze map and the *amcl* node for adaptive monte carlo localization. The latter corrects odometry errors of the robot by providing and adjusting the transformation between `map` and `odom`. *amcl* requires an initial pose which is set to the actual robot pose as defined in the stage world file. All other parameters are kept at their default settings in this tutorial.

Congratulations, the initial navigation setup is completed now. Further parameter adjustments are easily integrated into the configuration and launch files. Start your launch file in order to test the overall scheme:

```
$ roslaunch teb_tutorial robot_in_stage.launch
```

Ideally, a stage window occurs, no error messages appear and `move_base` prints “odom received!”. Afterwards start a new terminal and open *rviz* in order to start the visualization and send navigation goals:

```
$ rosrun rviz rviz
```

Make sure to set the fixed frame to `map`. Add relevant displays by clicking on the *Add* button. You can easily show all available displays by selecting the tab *by topic*. Select the following displays:

- from `/map/`: `Map`
- from `/move_base/TebLocalPlannerROS/`: `global_plan/Path`, `local_plan/Path`, `teb_markers/Marker` and `teb_poses/PoseArray`
- from `global_costmap/`: `costmap/Map`
- from `local_costmap/`: `costmap/Map` and `footprint/Polygon` (we do not have a robot model to display, so use the footprint).

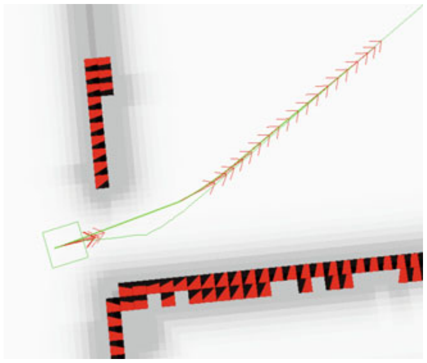
Now specify a desired goal pose using the *2D Nav Goal* button in order to start navigation.

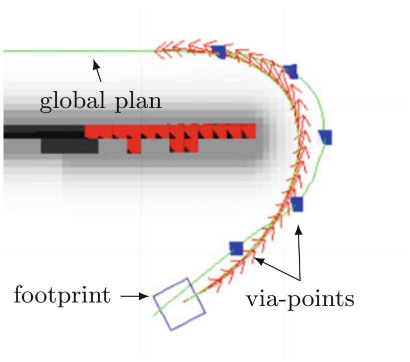
In the following, some parameters are modified during runtime: Keep everything running and open in a new terminal:

```
$ rosrun rqt_reconfiugre rqt_reconfigure
```

Select `move_base/TebLocalPlanner` in the menu in order to list all parameters which can be changed online. Now increase `min_obstacle_dist` to 0.5 m which is larger than the door widths in the map (the robot assumes a free space of 2 × 0.5 m then). Move the robot through a door and observe what is happening. The behavior should be similar to the one shown in Fig. 8a. The planner still tries to plan through the door according to the global plan and since the solution constitutes a local minimum. From the optimization point of view, the distance to each pose is minimized such that poses must be moved along the trajectory in both directions in order to avoid penalties (introducing the gap). However, `min_obstacle_dist` is chosen such that a door passing cannot be intended. As a consequence, the robot collides. After testing, reset the parameter back to 0.25 m.

The following task involves configuration of the trade-off between time optimality and global path following. Activate the via-points objective function by increasing the parameter `global_plan_viapoint_sep` to 0.5. Command a


(a) Improper parameter value for `min_obstacle_dist`



(b) Via-points added along the global plan

Fig. 8 Testing navigation with the differential-drive robot

new navigation goal and observe the new blue quadratic markers along the global plan (see Fig. 8b). Via-points are generated each 0.5 m (according to parameter value `global_plan_viapoint_sep`). Each via-point constitutes an attractor for the trajectory during optimization. Obviously, the trajectory still keeps a certain distance to some via-points as shown in Fig. 8b. The optimizer minimizes the weighted sum of both objectives: time optimality and reaching via-points. By increasing the weight `global_plan_viapoint_sep` (via *rqt_reconfigure*) and commanding new navigation goals you might recognize that the robot increasingly tends to prefer the original global plan over the fastest trajectory. Note, an excessive optimization weight for via-points might cause the obstacle cost to become negligible in comparison to the via-point cost. In that case avoiding dynamic obstacles does not work properly anymore. However, a suitable parameter setting for a particular application is determined in simulation.

7 Planning for a Car-Like Robot

This section describes how to configure the car-like robot defined in Sect. 5.2 for simulation with *stage*. The steps for setting up the differential drive robot as described in the previous section must be completed before in order to avoid redundant explanations.

Create a copy of the `teb_local_planner_params.yaml` file for the new car-like robot:

```
$ roscd teb_tutorial/cfg
$ cp teb_local_planner_params.yaml
    teb_local_planner_params_carlike.yaml
$ gedit teb_local_planner_params_carlike.yaml
```

Change the robot section according to the following snippet:

```
# file: teb_local_planner_params_carlike.yaml
 # Robot
 max_vel_x: 0.4
 max_vel_x_backwards: 0.2
 max_vel_theta: 0.3
 acc_lim_x: 0.5
 acc_lim_theta: 0.5
 min_turning_radius: 0.5        # we have a car-like robot!
 wheelbase: 0.4                 # wheelbase of our robot
 cmd_angle_instead_rotvel: True # angle instead of the rotvel
    for stage
 weight_kinematics_turning_radius: 1 # increase, if the penalty
     for min_turning_radius is not sufficient
 footprint_model:
   type:"line"
   line_start: [0.0, 0.0] # include robot expanse in
    min_obstacle_dist
```

```
  line_end: [0.4, 0.0] # include robot expanse in
   min_obstacle_dist
```

Parameter `min_turning_radius` is non-zero in comparison to the differential-drive robot configuration. The steering angle ϕ of the front wheels of the robot is limited to $\pm 40\,\text{deg}(\approx \pm 0.7\,\text{rad})$. From trigonometry the relation between the turning radius r and the steering angle ϕ is defined by $r = L/\tan\phi$ [9]. Hereby, L denotes the wheelbase. Evaluating the expression with $\phi = 0.7\,\text{rad}$ and $L = 0.4\,\text{m}$ reveals a minimum turning radius of 0.47 m. Due to the penalty terms, it is rounded up to 0.5 m for the parameter `min_turning_radius`. Since *move_base* provides a `geometry_msgs/Twist` message containing linear and angular velocity commands v and ω respectively, the signals are transformed to a robot base driver that only accepts the linear velocity v and a steering angle ϕ. Since the turning radius is expressed by $r = v/\omega$, the relation to the steering angle ϕ follows immediately: $\phi = \text{atan}(Lv/\omega)$. The case $v = 0$ is treated separately, e.g. by keeping the previous angle or by setting the steering wheels to their default position. For robots accepting an `ackermann_msgs/AckermannDriveStamped` message type, a simple converter node/script is added to communicate and map between *move_base* and the base driver. As described in Sect. 5.2 *stage* requires the default `geometry_msgs/Twist` type but with changed semantics: the angular velocity component is interpreted as steering angle. The *teb_local_planner* already provides the automatic conversion for this type of interface by activating parameter `cmd_angle_instead_rotvel`. The steering angle ϕ is set to zero in case of zero linear velocities ($v = 0\,\frac{\text{m}}{\text{s}}$).

The footprint model is redefined for the rectangular robot (according to Sect. 5.2). The line model is recommended for rectangular-shaped robots. Instead of defining the line over the complete width ($-0.1\,\text{m} \le x_r \le 0.5\,\text{m}$), 0.1 m are subtracted in order to account for the robot's expansion along the y_r-axis, since this value is added to parameter `min_obstacle_dist` similar to the differential-drive robot in Sect. 6. With some additional margin, `min_obstacle_dist` $= 0.25\,\text{m}$ should perform well, such that the parameter remains unchanged w.r.t. the previous configuration.

Create a new launch file in order to test the modified configuration:

```
$ roscd teb_tutorial/launch
$ cp robot_in_stage.launch carlike_robot_in_stage.launch
$ gedit carlike_robot_in_stage.launch
```

The `stage_ros` node must now load the *maze_carlike.world* file. Additionally, the local planner parameter configuration file must be replaced by the car-like version. An additional parameter `clearing_rotation_allows` is set to `false` in order to deactivate recovery behaviors which require the robot to turn in place. Relevant snippets are listed below:

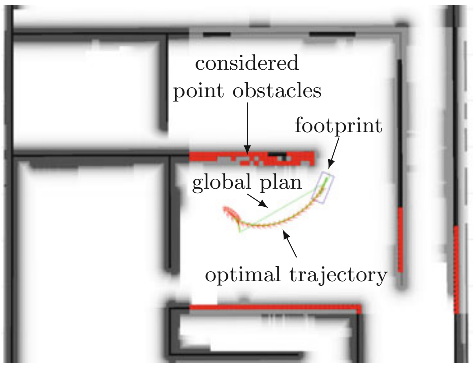
```
<!-- file: carlike_robot_in_stage.launch -->
<!-- ... -->
<!--  ************** Stage Simulator ***************  -->
<node pkg="stage_ros" type="stageros" name="stageros" args="$(
    find teb_tutorial)/stage/maze_carlike.world">
    <remap from="base_scan" to="scan"/>
</node>

<!--  ************** Navigation ***************  -->
<node pkg="move_base" type="move_base" respawn="false" name="
    move_base" output="screen">
    <!-- ... -->
    <rosparam file="$(find teb_tutorial)/cfg/
    teb_local_planner_params_carlike.yaml" command="load" />
    <!-- ... -->
    <param name="clearing_rotation_allowed" value="false" /> <!
    -- Our carlike robot is not able to rotate in place -->
</node>
<!-- ... -->
```

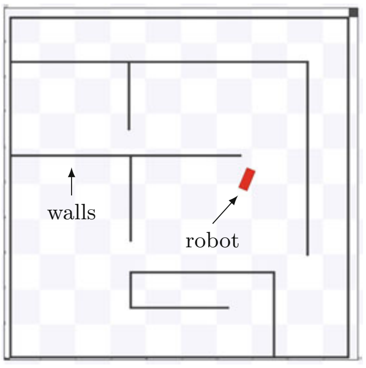
Close any previous ROS nodes and terminals and start the car-like robot simulation:

```
$ roslaunch teb_tutorial robot_in_stage.launch
```

If no errors occur, navigate your robot through the environment. Again run *rviz* for visualization with all displays configured in Sect. 5.1 and *rqt_reconfigure* for playing with different parameter settings. An example scenario is depicted in Fig. 9. The displayed markers in *rviz* indicate occupied cells of the local costmap which are taken into account as point obstacle during trajectory optimization.



(a) Visualization in *rviz*



(b) *Stage* simulator preview

Fig. 9 Navigating a car-like robot in simulation

8 Conclusion

This tutorial chapter presented a step-by-step guide on how to setup the package *teb_local_planner* in ROS for navigation with a differential-drive and a car-like robot. The package implements an online trajectory optimization scheme termed Timed-Elastic-Band approach and it seamlessly integrates with the navigation stack as local planner plugin. The fundamental theory and concepts of the underlying approach along with related ROS parameters was introduced. The package provides an effective alternative to the currently available local planners as it supports trajectories planning with cusps (backward motion) and car-like robots. To our knowledge, the latter is currently not provided by any other local planner. The package allows the user to quantify a spatial-temporal trade-off between a time optimal trajectory and compliance with the original global plan. Further work intends to address the automatic tuning of cost function weights for common cluttered environments and maneuvers. Furthermore, a benchmark suite for the performance evaluation of the different planners available in ROS could be of large interests for the community. Benchmark results facilitate the appropriate selection of planners for different kinds of applications. Additionally, future work aims to include dynamic obstacles, support of additional kinematic models as well as further improving algorithmic efficiency.

References

1. Rösmann, C., Feiten, W., Wösch, T., Hoffmann, F., and T. Bertram. 2012. Trajectory modification considering dynamic constraints of autonomous robots. In *7th German Conference on Robotics (ROBOTIK)*, 74–79.
2. Rösmann, C., Feiten, W., Wösch, T., Hoffmann, F., and T. Bertram. 2013. Efficient trajectory optimization using a sparse model. In *6th European Conference on Mobile Robots (ECMR)*, 138–143.
3. Rösmann, C., Hoffmann, F., and T. Bertram. 2015. Planning of multiple robot trajectories in distinctive topologies. In *IEEE European Conference on Mobile Robots*, 1–6.
4. Kümmerle, R., Grisetti, G., Strasdat, H., Konolige, K., and W. Burgard. 2011. G2o: A general framework for graph optimization. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 3607–3613.
5. Nocedal, J., and S.J. Wright. 1999. *Numerical Optimization.*, Springer series in operations research New York: Springer.
6. Morari, M., and J.H. Lee. 1999. Model predictive control: past, present and future. *Computers and Chemical Engineering* 23 (4–5): 667–682.
7. Bhattacharya, S., Kumar, V., and M. Likhachev. 2010. Search-based path planning with homotopy class constraints. In *Proceedings of National Conference on Artificial Intelligence*.
8. Guimarães, R.L., de Oliveira, A.S., Fabro, J.A., Becker, T., and V.A. Brenner. 2016. ROS Navigation: Concepts and Tutorial. In *Robot Operating System (ROS) - The Complete Reference* (A. Koubaa, ed.), vol. 625 of *Studies in Computational Intelligence*, pp. 121–160, Springer International Publishing.
9. LaValle, S.M. 2006. *Planning Algorithms*. New York, USA: Cambridge University Press.

Christoph Rösmann was born in Münster, Germany, on December 8, 1988. He received the B.Sc. and M.Sc. degree in electrical engineering and information technology from the Technische Unversität Dortmund, Germany, in 2011 and 2013 respectively. He is currently working towards the Dr.-Ing. degree at the Institute of Control Theory and Systems Engineering, Technische Universität Dortmund, Germany. His research interests include nonlinear model predictive control, mobile robot navigation and fast optimization techniques.

Frank Hoffmann received the Diploma and Dr. rer. nat. degrees in physics from Christian-Albrechts University of Kiel, Germany. He was a Postdoctoral Researcher at the University of California, Berkeley from 1996–1999. From 2000 to 2003, he was a lecturer in computer science at the Royal Institute of Technology, Stockholm, Sweden. He is currently a Professor at TU Dortmund and affiliated with the Institute of Control Theory and Systems Engineering. His research interests are in the areas of robotics, computer vision, computational intelligence, and control system design.

Torsten Bertram received the Dipl.-Ing. and Dr.-Ing. degrees in mechanical engineering from the Gerhard Mercator Universität Duisburg, Duisburg, Germany, in 1990 and 1995, respectively. In 1990, he joined the Gerhard Mercator Universität Duisburg, Duisburg, Germany, in the Department of Mechanical Engineering, as a Research Associate. During 1995–1998, he was a Subject Specialist with the Corporate Research Division, Bosch Group, Stuttgart, Germany, In 1998, he returned to Gerhard Mercator Universität Duisburg as an Assistant Professor. In 2002, he became a Professor with the Department of Mechanical Engineering, Technische Universität Ilmenau, Ilmenau, Germany, and, since 2005, he has been a member of the Department of Electrical Engineering and Information Technology, Technische Universität Dortmund, Dortmund, Germany, as a Professor of systems and control engineering and he is head of the Institute of Control Theory and Systems Engineering. His research fields are control theory and computational intelligence and their application to mechatronics, service robotics, and automotive systems.