Online Trajectory Optimization and Navigation in Dynamic Environments in ROS

Franz Albers, Christoph Rösmann, Frank Hoffmann and Torsten Bertram

Abstract This tutorial chapter provides a comprehensive step-by-step guide on the setup of the navigation stack and the *teb_local_planner* package for mobile robot navigation in dynamic environments. The *teb_local_planner* explicitly considers dynamic obstacles and their predicted motions to plan an optimal collision-free trajectory. The chapter introduces a novel plugin to the *costmap_converter* ROS package which supports the detection and motion estimation of moving objects from the local costmap. This tutorial covers the theoretical foundations of the obstacle detection and trajectory optimization in dynamic scenarios. The presentation is designated for ROS Kinetic and Lunar and both packages will be maintained in future ROS distributions.

Keywords Navigation · Local planning · Online trajectory optimization Dynamic obstacles · Dynamic environment · Obstacle tracking

1 Introduction

In the context of service robotics and autonomous transportation systems mobile robots are required to safely navigate environments populated with humans and other robots. On this occasion, universally applicable motion planning strategies are of utmost importance in mobile robot applications. Online planning is favored over offline approaches as it responds to dynamic environments, map inconsistencies or robot motion uncertainty. Furthermore, online trajectory optimization conciliates

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

C. Rösmann
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 778, https://doi.org/10.1007/978-3-319-91590-6_8

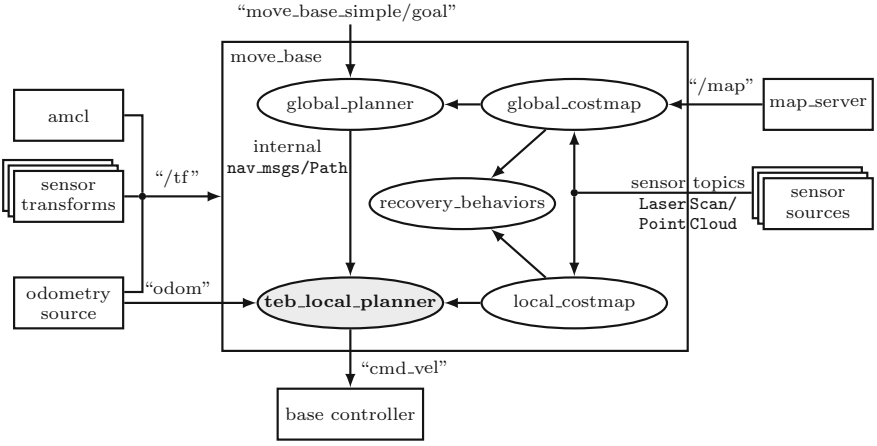


Fig. 1 Overview of the ROS navigation stack including the *teb_local_planner*

among partially conflicting objectives such as control effort, path fidelity, overall path length or transition time.

The navigation stack[1] [12] along with its plugins for environment representations (in terms of occupancy grids respectively costmaps) and local and global planners constitutes a widely established framework for mobile robot navigation in the Robot Operating System (ROS). Figure 1 shows an overview of the navigation stack setup[2] employed in the course of this chapter.

A *global_costmap* is generated based on a known map of the environment (published by the *map_server*). Additionally, sensor readings which are classified as observations of static obstacles are incorporated into the *global_costmap*. The *global_planner* computes an initial path to the goal based on the *global_costmap*. This path does neither consider any time information nor obstacles which are not represented in the *global_costmap* (for example because they are dynamic or simply were not present at the time of mapping). In order to consider these obstacles as well, the *local_costmap* is generated from fused readings of the robot's sensors. Taking into account the *local_costmap*, the *local_planner* optimizes the initial plan defined by the *global_planner* and publishes the appropriate velocities to the underlying *base controller*. In contrast to the standard architecture of the navigation stack, the default *local_planner* is replaced by the gray highlighted *teb_local_planner* plugin in this chapter. The robot localizes itself using a combination of odometry (motion estimation relative to the robot's starting position based on data from its drive system) and Adaptive Monte Carlo Localization (amcl) [22]. Transformations between multiple coordinate frames are provided by the *tf* package.[3]

[1] *ROS navigation*, http://wiki.ros.org/navigation.

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

[3] *tf*, http://wiki.ros.org/tf.

The current implementation of the navigation stack assumes a quasi-static environment and neither predicts nor considers the motion of dynamic obstacles (such as humans or other robots) explicitly. The costmap representation only provides a static view of the current environment and lacks the temporal evolution of grid occupancy. Consequently, planners are unable to benefit from the knowledge of a moving obstacle's estimated velocity and heading. In order to achieve robust navigation in dynamic environments, ROS navigation implements so-called inflation in which static occupied costmap cells are inflated by an exponentially decreasing cost decay rate. Thus, the robot plans a more pessimistic trajectory and maintains a larger separation from obstacles than actually required by collision avoidance.

The authors developed a package for dynamic obstacle detection and tracking based on the two-dimensional costmap of the ROS navigation stack. The approach rests upon an algorithm for foreground detection in the rolling window of the costmap that discriminates between occupied cells attributed to moving objects and the static *background*. The approach compensates the robot's ego-motion to obtain an unbiased estimate of obstacle velocities w.r.t. a global frame. The foreground cells are clustered into a set of obstacles for which individual model-based filters (Kalman-Filters) are applied for the ongoing state estimation of the obstacle motion. The node publishes the set of current dynamic obstacles in terms of their estimated location, footprint (shape), translational velocity vector and its uncertainty at every sampling interval. Local trajectory planners utilize the estimated motion of obstacles to plan the future collision-free robot trajectory ahead of time. For that purpose the interface for local planners should not only consider the current costmap but also its temporal evolution due to obstacle motion.

The previous volume of the book *Robot Operating System - The Complete Reference* includes a tutorial chapter on kinodynamic motion planning with Timed-Elastic-Bands (TEB) [18]. The package *teb_local_planner*[4] implements a local planner plugin for the ROS navigation stack. The underlying TEB approach efficiently optimizes the robot trajectory w.r.t. (kino-)dynamic constraints and non-holonomic kinematics of differential-drive, car-like or omnidirectional mobile robots while explicitly incorporating temporal information in order to reach the goal pose in minimal time [16, 17]. With its recent update, the *teb_local_planner* explicitly considers dynamic obstacles based on the estimates provided by the tracker for local planning. Multiple questions and inquiries in (*ROS Answers*) and personal feedback to the authors clearly indicate a broad interest of the community to support dynamic obstacles in ROS mobile robot navigation.

This chapter covers the following topics:

- The current state of the art for mobile robot path planning is briefly summarized in Sect. 2 with a focus on currently available *local_planner* plugins for the ROS navigation stack
- The algorithmic background for local costmap conversion is explained in Sect. 3
- Section 4 introduces the novel `CostmapToDynamicObstacles` plugin for the *costmap_converter* along with an intuitive example setup in Sect. 4.4

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

- Section 5 presents the theoretical foundations of the TEB trajectory optimization methods
- Section 6 discusses the *teb_local_planner* along with a basic test node for spatio-temporal trajectory optimization (Sect. 7)
- Finally, Sect. 8 explains the setup of a navigation task in environments with moving obstacles.

2 Related Work

Collision-free locomotion is a fundamental skill for mobile robots. Especially in dynamic environments, online planning is preferred over offline approaches due to its immediate response to alterations in the vicinity of the robot. The *Dynamic Window Approach* (DWA) constitutes a well-known online trajectory planning approach [4]. It rests upon a dynamic window in the control input space from which admissible velocities for the robot are sampled in each time step. The search space is restricted to collision-free velocities considering the dynamics of the robot. For each sample, a short-term prediction of the future motion is simulated and evaluated w.r.t. a cost function (including a distance measure to the goal and obstacle avoidance terms) by assuming constant control inputs. The DWA was extended by Seder and Petrović for navigation in dynamic environments [21]. The *Trajectory Rollout* approach operates in a similar manner as the original DWA, but rather samples a set of achievable velocities over the entire forward simulation period [6]. In the context of car-like robots, [15] restricts the search space of rotational velocities to the set of feasible solution. However, due to the assumption of constant velocities in the prediction, motion reversals which are required for car-like robots to navigate in confined spaces are not explicitly considered during planning.

Fiorini and Shiller [3] present another approach called *Velocity Obstacles* to velocity-based sampling in the state space. The search space is restricted to admissible velocities that do not collide with dynamic obstacles within the prediction horizon. Fulgenzi et al. extended this method in order to account for uncertainty in the obstacle's motion by means of a Bayesian Occupancy Filter [5].

The *Elastic-Band* approach deforms a path to the goal by applying an internal contraction force resulting in the shortest path and external repulsive forces radiating from the obstacles to receive a collision-free path [23]. However, this approach does not incorporate time information. Hence, the robot's kinodynamic constraints are not considered explicitly and a dedicated path following controller is required. The *2-Step-Trajectory-Deformer* approach incorporates time information in a subsequent planning stage [10]. Resulting trajectories are feasible for holonomic robots with kinodynamic constraints. An extension to non-holonomic robots is presented by Delsart and Fraichard [2]. Gu et al. [7] present a multi-state planning approach which utilizes an optimization-free *Elastic-Band* to generate paths followed by a speed planning stage for car-like robots.

Likewise, the TEB approach augments the *Elastic-Band* method with time information in order to generate time-optimal trajectories [16]. The approach was recently

extended to parallel trajectory planning in spatially distinctive topologies [17] and to car-like robot kinematics [19]. The TEB is efficiently integrated with state feedback to repeatedly refine the trajectory w.r.t. disturbances and changing environments. Desired velocities are directly extracted from the planned trajectory.

A variety of the previously presented approaches (DWA, *Trajectory Rollout*, *Elastic-Band* and the *Timed-Elastic-Band*) are provided as local planner plugin in the ROS navigation stack [12]. However, the TEB approach presented in this chapter is currently the only local planner plugin that explicitly incorporates the estimated future motions of dynamic obstacles into trajectory optimization.

The *costmap_converter* package presented in this chapter estimates the velocity of dynamic obstacles according to a constant velocity model. More sophisticated obstacle tracking approaches like the social forces model [11] take cooperative joint motions among a group of agents into account. These models are currently not included, but they might be implemented in potential future package versions.

3 Costmap-Based Obstacle Velocity Estimation

In the ROS navigation stack (see Fig. 1) the costmap represents an occupancy grid. The status of each cell is either free (0), occupied (254), or unknown (255). Costmaps are updated at a specific rate. Unfortunately, the costmap does not include historical data from which velocity and heading of moving obstacles could be inferred. Therefore, the authors implemented a tracker and velocity estimation based on the history of the costmap. The ego-motion is compensated by transforming the observed obstacle velocities from the sensor frame to the global map frame w.r.t. the robot's odometry.

Typical tracking algorithms operate with range measurements but make particular assumptions about the sensor characteristics. Our tracking scheme merely relies on the costmap with already aggregated sensor data, which facilitates its integration into the navigation stack, but arguably sacrifices tracking accuracy and resolution. The tracker is universally applicable as it requires no configuration or adaptation to different types of sensors since data fusion is already accomplished at the costmap stage. The tracker is implemented as a plugin to the existing *costmap_converter* package which provides plugins that convert the static costmap to geometric primitives like lines or polygons. The presented method converts only dynamic obstacles to geometric primitives augmented with estimated location and velocity. Static obstacles are not processed and remain point-shaped.

The foreground detection algorithm extracts dynamic obstacles from the local costmap by subtracting the outputs of two running average filters resulting in a bandpass filter. The binary map labels the cells of moving foreground objects as true, and static and obstacle-free regions as false (see Fig. 2). The centroids and contours of these obstacles are determined by a blob detector from computer vision. The second step is concerned with clustering and tracking connected and coherent cells to individual obstacles and to estimate their location and velocity.

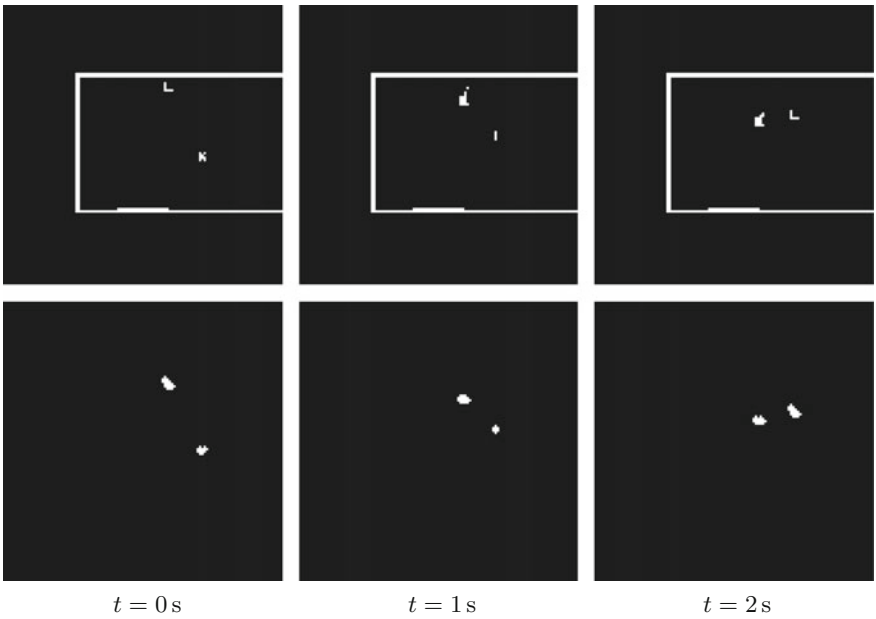


Fig. 2 Evolution of the costmap (upper row) and the corresponding detected dynamic obstacles (lower row) with two dynamic obstacles perceived by a static observer

3.1 Dynamic Obstacle Detection

Though the approach also works when the ego robot is moving, the discussion of the theoretical background assumes that the observing robot and therefore the position of the local costmap w.r.t. to the global frame remains static.

The foreground detection operates with a slow and a fast *running average filter*. These filters are applied to each cell in the costmap. For a single cell these filters are described by:

$$P_{\mathrm{f}}(t+1) = ((1-\alpha_{\mathrm{f}})\, P_{\mathrm{f}}(t) + \alpha_{\mathrm{f}}\, C(t)) \tag{1}$$

$$P_{\mathrm{s}}(t+1) = ((1-\alpha_{\mathrm{s}})\, P_{\mathrm{s}}(t) + \alpha_{\mathrm{s}}\, C(t)) \tag{2}$$

$P_{\mathrm{f}}(t)$ and $P_{\mathrm{s}}(t)$ represent the output of the fast and the slow running average filter at time t, respectively. The gains α_{f} and α_{s} define the effect of the current costmap $C(t)$ on P and comply with:

$$0 \leq \alpha_{\mathrm{s}} < \alpha_{\mathrm{f}} \leq 1 \tag{3}$$

For the detection of particular large objects, which form blocks of cells in the local costmap, the Eqs. 1 and 2 are extended by a term that captures the running

average filter of the cells nearest neighbors (NN). β denotes the ratio between the contribution of the central cell filter and the effect of the neighboring cells to $P_{\mathrm{f}}(t)$ and $P_{\mathrm{s}}(t)$.

$$P_{\mathrm{f}}(t+1) = \beta\left((1-\alpha_{\mathrm{f}})\, P_{\mathrm{f}}(t) + \alpha_{\mathrm{f}}\, C(t)\right) + \frac{(1-\beta)}{8} \sum_{i \in \mathrm{NN}} P_{\mathrm{f},i}(t) \tag{4}$$

$$P_{\mathrm{s}}(t+1) = \beta\left((1-\alpha_{\mathrm{s}})\, P_{\mathrm{s}}(t) + \alpha_{\mathrm{s}}\, C(t)\right) + \frac{(1-\beta)}{8} \sum_{i \in \mathrm{NN}} P_{\mathrm{s},i}(t) \tag{5}$$

These filters identify those cells that are occupied by moving obstacles if they comply with two criteria that filter out high and low frequency noise. The fast filter has to exhibit an activation that exceeds a threshold c_1:

$$P_{\mathrm{f}}(t) > c_1 \tag{6}$$

In addition, the difference between the fast and the slow filter has to exceed a threshold c_2 in order to eliminate quasi-static obstacles with low frequency noise.

$$P_{\mathrm{f}}(t) - P_{\mathrm{s}}(t) > c_2 \tag{7}$$

Figure 3 shows the the filtered signal in the local costmap C of the fast filter P_{f} and slow filter P_{s} in case of a dynamic obstacle that traverses the cell over a period of two seconds together with the activation thresholds c_1 and c_2.

The thresholding operations generate a binary map that labels dynamic obstacles as ones, whereas free space and static obstacles are labeled as zeros. The sequence of erosion and dilation on the binary map results in a closing operation which reduces noise in the foreground map.

A slightly modified version of OpenCV's `SimpleBlobDetector` extracts obstacle centroids and contours from the binary map. These obstacle features provide the input to the algorithm for simultaneous tracking and velocity estimation of multiple obstacles.

3.2 Dynamic Obstacle Tracking

The centroid of dynamic obstacles progresses with each costmap update and subsequent foreground detection. The assignment of blobs in the current map to obstacle tracks constitutes a data association problem. In order to disambiguate and track multiple objects over time, the current obstacles are matched with the corresponding tracks of previous obstacles. A new track is generated whenever a novel obstacle emerges that is not tracked yet. Tracks that are not assigned to current objects in the

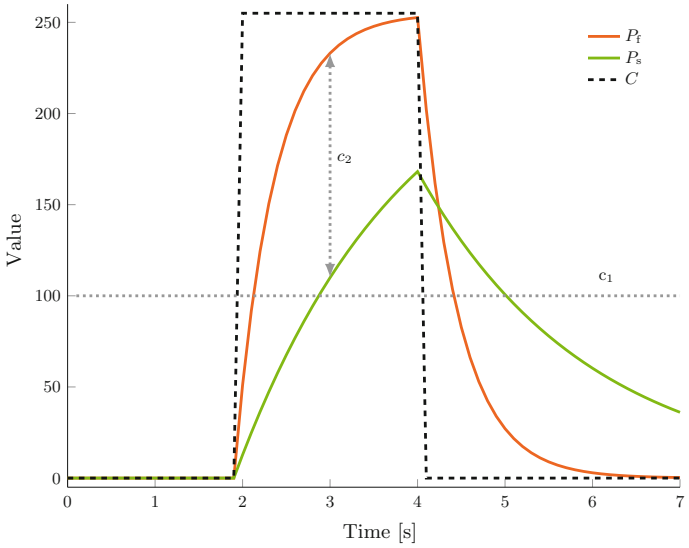


Fig. 3 Slow and fast filter responses

foreground frame are temporarily maintained. The track is removed if it is no longer confirmed by object detections over an extended period of time.

The assignment problem is solved by the so-called Hungarian algorithm, which was originally introduced by [8]. The algorithm efficiently solves weighted assignment problems by minimizing the total Euclidean distance between the tracks and the current set of obstacle centroids.

A Kalman filter estimates the current velocity of tracked obstacles assuming a first order constant velocity model. The constant velocity model sufficiently captures the prevalent motion patterns of humans and robots in indoor environments for the designated spatio-temporal horizon of motion planning.

4 Costmap_Converter ROS Package

This section introduces the technical aspects of the *costmap_converter* ROS package. It handles the conversion of dynamic obstacles extracted from the local costmap (in terms of a `nav_msgs/OccupancyGrid` message) into polygons augmented with velocity and heading information. In addition, the package provides various plugins to convert connected regions in the local costmap to geometric primitives without estimating their velocities. The presentation here focuses on the `CostmapToDynamicObstacles` plugin. The ROS wiki[5] documents the addi-

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

tional plugins. The utilized plugin can be selected prior to run-time via the parameter `costmap_converter_plugin`. The `CostmapToDynamicObstacles` plugin pursues a two-step approach with the initial conversion and velocity estimation of dynamic obstacles and the subsequent conversion of static obstacles by means of an additional static plugin. The parameter `static_costmap_converter_plugin` specifies the employed plugin for static costmap conversion.

The *costmap_converter* package is available for ROS Kinetic and Lunar. The package defines a pluginlib[6] interface and is primarily intended for direct embedding in the source code. Some applications might prefer a dedicated subscriber `standalone_converter` node to process `nav_msgs/OccupancyGrid` messages.

4.1 Prerequisites and Installation

It is assumed that the reader is accustomed to basic ROS concepts such as navigating the filesystem, creating and building packages, as well as dealing with *rviz*[7], launch files, topics, parameters and *yaml* files discussed in the common ROS beginner tutorials. Familiarity with the 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 is expected.

In the following, terminal commands are indicated by a leading $-sign. The *costmap_converter* package is installed from the official ROS repositories by invoking:

```
$ sudo apt-get install ros-kinetic-costmap-converter
```

More recent, albeit experimental, versions of the *costmap_converter* package can be obtained and compiled from source:

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

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

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

The user-created *catkin* workspace is assumed to be located at `~/catkin_ws`.

Currently, the *costmap_converter* does not handle incremental costmap updates published to the `grid_updates` topic. In order to use this package, the full costmap has to be published in every update cycle. Incremental updates can be deactivated by enabling the *costmap_2d*[8] parameter `always_send_full_costmap`. In the tutorial scenarios introduced in this chapter, this parameter is already defined in the *costmap_common_params.yaml* file.

4.2 Obstacle Messages

Converted obstacles are published as the novel message type `ObstacleMsg` specifically designed for the purpose of publishing obstacles in conjunction with their velocities. The compact definition of an `ObstacleMsg` is pictured below.

```
# Special types:
# Polygon with 1 vertex: Point obstacle
# Polygon with 2 vertices: Line obstacle
# Polygon with more than 2 vertices: First and last points are
    assumed to be connected

std_msgs/Header header

# Obstacle footprint (polygon descriptions)
geometry_msgs/Polygon polygon

# Obstacle ID
# Specify IDs in order to provide (temporal) relationships
# between obstacles among multiple messages.
int64 id

# Individual orientation (centroid)
geometry_msgs/Quaternion orientation

# Individual velocities (centroid)
geometry_msgs/TwistWithCovariance velocities
```

An obstacle is represented by a polygon, i.e. array of vertices. Polygons with a single vertex refer to point obstacles and polygons with two vertices denote line obstacles. In case of true polygons with more than two vertices, the first and last points are assumed to be connected in order to close the polygon perimeter. Additionally, the message provides the `velocities`, `orientation` and `id` of obstacles. The orientation and velocity of an obstacle are specified w. r. t. their respective centroid.

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

`ObstacleMsgs` are grouped to an `ObstacleArrayMsg` which can include not merely the converted dynamic obstacles, but also the static background of the costmap in terms of point-shaped obstacles.

4.3 Parameters

Most of the package parameters originate from OpenCV's *SimpleBlobDetector*, for which the reader is referred to the OpenCV documentation.[9] This section explains the additional parameters of the *costmap_converter* plugin for dynamic obstacle conversion.

`alpha_fast` (α_f)
Adaption rate of the fast running average filter in Eq. (4). A higher rate indicates a higher confidence in the most recent costmap. `alpha_fast` has to be larger than `alpha_slow` (see Eq. (3)).

`alpha_slow` (α_s)
Adaption rate of the slow running average filter in Eq. (5). A higher rate indicates a higher confidence in the most recent costmap. `alpha_slow` has to be smaller than `alpha_fast` (see Eq. (3)).

`beta` (β)
Ratio, of the contribution to the running filterscenter cell relative to the neighboring cells (see Eqs. (4) and (5)).

`min_occupancy_probability` (c_1)
Threshold of the fast filter for classification of a cell as foreground.

`min_sep_between_slow_and_fast_filter` (c_2)
Threshold of minimal difference between the slow and fast running average filters for classification of the cell as foreground.

`max_occupancy_neighbors`
Maximum mean value of the 8-neighborhood for classification of the cell as foreground.

`morph_size`
Size of the structuring element (circle) used for the closing operation applied to the binary map after foreground detection.

[9] OpenCV *SimpleBlobDetector*, http://docs.opencv.org/3.3.0/d0/d7a/classcv_1_1SimpleBlobDetector.html.

`dist_thresh`
Maximum Euclidean distance between obstacles and tracks to be considered for matching in the assignment problem.

`max_allowed_skipped_frames`
Maximum number of frames for which a dynamic obstacle is tracked without confirmation in the current foreground map.

`max_trace_length`
Maximum number of points representing in the object trace.

`publish_static_obstacles`
Include obstacles from the static background cells. By default, static costmap cells are subsequently converted to polygons by the `CostmapToPolygonsDBSMCCH` plugin for static costmap conversion.

4.4 Prototype Scenario for Obstacle Velocity Estimation

This section introduces a minimal *stage*[10] simulation setup with costmap conversion respectively velocity estimation for dynamic obstacles. The simulation setup consists of an observing robot and a single dynamic obstacle moving in a simple square environment. A costmap conversion scenario including navigation by means of the *teb_local_planner* is discussed in Sect. 8.

Stage is a fast and lightweight mobile robot simulator. Although this tutorial chapter refers to *stage*, the demo code is equally applicable to other simulation environments such as *gazebo*.[11] In case *Stage* is not yet installed along with the full ROS distribution package invoke:

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

Clone (or download and unzip) the *teb_local_planner_tutorials* package[12] for this tutorial:

```
$ cd ~/catkin_ws/src
$ git clone -b rosbook_volume3 https://github.com/rst-tu-
    dortmund/teb_local_planner_tutorials.git
```

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

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

[12] *teb_local_planner_tutorials*, https://github.com/rst-tu-dortmund/teb_local_planner_tutorials/tree/rosbook_volume3.

The simulated environment in *stage* consists of a static map and other agents such as robots or (dynamic) obstacles. The additional agents are included in a separate definition file for the observing robot. Observe the robot and sensor definitions for *stage* defined in *myRobot.inc* located in the subfolder *stage*. The following sensor and robot definitions are utilized in the course of this tutorial:

```
define mylaser ranger
(
  sensor
  (
    # just for demonstration purposes
    range [ 0.1 25 ] # minimum and maximum range
    fov 360.0 # field of view
    samples 1920 # number of samples
  )
  size [ 0.06 0.15 0.03 ]
)

define myrobot position
(
  size [ 0.25 0.25 0.4 ] # (x,y,z)
  localization "gps" # exact localization
  gui_nose 1 # draw nose on the model showing the heading
  drive "diff" # diff-drive
  color "red" # red model
  mylaser(pose [ -0.1 0.0 -0.11 0.0 ]) # add mylaser sensor
)
```

The first code block defines a range sensor named `mylaser` with a complete field of view (`fov 360.0`). The second block defines a robot which utilizes the previously defined ranging sensor. The dynamic obstacle model is defined in the *myObstacle.inc* file in a similar way:

```
define myobstacle position
(
  localization "gps" # exact localization
  size [ 0.25 0.25 0.4 ] # (x,y,z)
  gui_nose 1 # draw nose on the model showing the heading
  drive "omni" # omni-directional movement possible
  color "blue" # blue model
)
```

The *stage* environment along with robots and obstacles is defined in the file *emptyBox.world*:

```
## include our robot and obstacle definitions
include "robots/myRobot.inc"
include "robots/myObstacle.inc"

## Simulation settings
resolution 0.02
interval_sim 100 # simulation timestep in milliseconds

## Load a static map
model
(
  name "emptyBox"
  bitmap "../maps/emptyBox.png"
  size [ 6.0 6.0 2.0 ]
  pose [ 0.0 0.0. 0.0 0.0]
  laser_return 1
  color "gray30"
)

# throw in a robot and an obstacle
myrobot
(
  pose [ -2.0 0.0 0.0 -90.0 ] # initial pose (x,y,z,beta[deg])
  name "myRobot"
)

myobstacle
(
  pose [ 0.0 1.0 0.0 0.0 ] # initial pose (x,y,z,beta[deg])
  name "myObstacle"
)
```

The robot definition files are included in lines 2–3. General simulation settings are defined in lines 6–7. Line 10–18 define the map model as an empty square box with an edge length of 6 m. The observing robot and the dynamic obstacle are defined from line 21 onward. Next, we will inspect the *costmap_conversion.launch* file step by step which is located in the *launch* subfolder of the package.

```
<!--  ************** Stage Simulator *****************  -->
<node pkg="stage_ros" type="stageros" name="stageros" args="$(find
    teb_local_planner_tutorials)/stage/emptyBox.world">
  <remap from="/robot_0/base_scan" to="/robot_0/scan"/>
</node>

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

These commands start the *stageros* and *map_server* nodes and load the *emptyBox* map.

```
<!-- ******** Localization ********* -->
<!-- See stage world file for initial poses -->
<node pkg="tf" type="static_transform_publisher" name="perfect_loc_robot
    " args="-2 0 0 -1.570796 0 0 /map robot_0/odom 100" />
<node pkg="tf" type="static_transform_publisher" name="
    perfect_loc_obstacle" args="0 1 0 0 0 0 /map robot_1/odom 100" />
```

In this section `static_transform_publishers` are launched publishing the transformations to the perfect robot and obstacle locations for localization.

```
<!--  ************* Navigation Ego Robot ************  -->
<group ns="robot_0">
  <param name="tf_prefix" value="robot_0"/>

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

    <param name="base_local_planner" value="teb_local_planner/
    TebLocalPlannerROS" />
    <param name="controller_frequency" value="5.0" />
    <param name="controller_patience" value="15.0" />
    <remap from="map" to="/map"/>
  </node>
</group>
```

This code brings up the ROS navigation stack in the namespace of *robot_0*. Since the navigation stack contains the *costmap_2d* package, it is required for costmap conversion. Various parameter files for the local and global costmap are loaded. The *teb_local_planner* is utilized as local planner, though other local planners can be used for costmap conversion.

```
<!--  ************* Costmap conversion ************  -->
<node name="standalone_converter" pkg="costmap_converter" type="
    standalone_converter" output="screen">
  <param name="converter_plugin" value="
    costmap_converter::CostmapToDynamicObstacles" />
  <param name="costmap_topic" value="/robot_0/move_base/local_costmap/
    costmap" />
  <param name="odom_topic" value="/robot_0/odom" />
</node>
```

These commands launch the *costmap_converter* standalone node with the previously introduced `CostmapToDynamicObstacles` plugin. The `odom_topic` is required in order to compensate a robots ego motion while estimating obstacle velocities. The *costmap_converter* subscribes to the `costmap_2d` and publishes an `ObstacleArrayMsg` with an array of `ObstacleMsgs`, each containing estimated obstacle velocities and shapes.

```
<!-- ******************* Obstacles ******************* -->
<group ns="robot_1">
  <param name="tf_prefix" value="robot_1"/>
  <node name="Mover" pkg="teb_local_planner_tutorials" type="
    move_obstacle.py" output="screen"/>
  <node name="visualize_obstacle_velocity_profile" pkg="
    teb_local_planner_tutorials" type="
    visualize_obstacle_velocity_profile.py" output="screen" />
</group>
```

Two distinctive nodes are launched in the namespace of *robot_1*. The script *move_obstacle.py* actuates the obstacle (by publishing a `cmd_vel` message to the obstacles namespace) with a constant velocity by sampling a velocity in the opposite direction in case of an encounter with the walls. The collision with a wall is detected from the `base_pose_ground_truth` topic in the obstacles namespace. Additionally, a node which plots both the estimated and ground truth velocities of an obstacle is started by the *visualize_obstacle_velocity_profile.py* script.

```
<!--  **************** Visualisation ****************  -->
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find
    teb_local_planner_tutorials)/cfg/rviz_navigation_cc.rviz">
  <remap from="/move_base_simple/goal" to="/robot_0/move_base_simple/
    goal" />
</node>
```

This section starts up *rviz* and loads a predefined configuration file. Launch the simulation by invoking:

```
roslaunch teb_local_planner_tutorials costmap_conversion.launch
```

stage and *rviz* pop up and display the simulated environment. The estimated footprint of the dynamic obstacle is indicated by a green polygon in *rviz*. Estimated and ground truth velocities are visualized in a live plot. In case of very slow obstacle velocities the *costmap_converter* might consider the obstacle as static and therefore lose track of the object. Try to customize the obstacle detection by changing the parameters of the *costmap_converter* using the *rqt_reconfigure* tool:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

It is recommended to invoke costmap conversion in a separate thread since the conversion of dynamic obstacles is a time critical task for safe navigation. The computational effort for one cycle of obstacle detection in the local costmap, velocity estimation and the publishing of these informations is depicted in Fig. 4. Regardless of the number of tracked obstacles, the median for one conversion cycle is located at around 1 ms. These measurements were taken using an Intel Core i5-6500 processor along with 8GB RAM. As the default costmap conversion rate is 5 Hz, costmap conversion can also be performed on less powerful hardware.

In the following, the estimated velocities provide the basis for optimal spatio-temporal trajectory planning by the *teb_local_planner*. Costmap conversion and velocity estimation do not depend on the *teb_local_planner* package and might be incorporated standalone into custom applications.

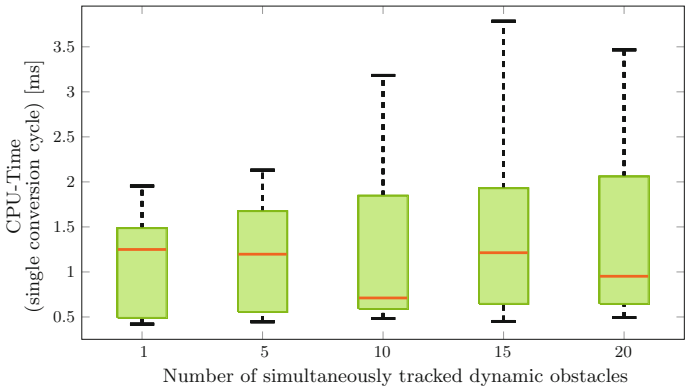


Fig. 4 Computational performance of the *costmap_converter*

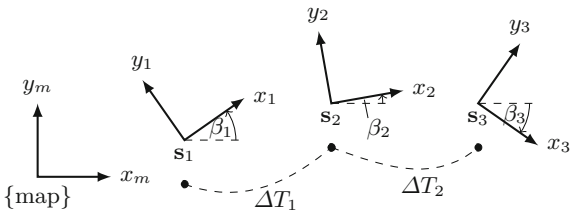


Fig. 5 Discretized trajectory with $n = 3$ poses

5 Theoretical Foundations of TEB

This section introduces the fundamental concepts of the TEB trajectory planning approach. It discusses the theoretical foundations for the successful integration and customization of the *teb_local_planner* in mobile robot applications. A more detailed description of online trajectory optimization with Timed-Elastic-Bands is given in [17].

The tutorial chapter on kinodynamic motion planning with Timed-Elastic-Bands published in the second volume of this book [18] discusses the theoretical foundations of the TEB approach. Even though the main features of TEB remain the same, for the sake of a self-contained presentation the basic operation of TEB optimization is explained in summary. The main modification is the extension from a quasi-static environment to a dynamic world that explicitly takes the future poses of moving obstacles into account for planning an optimal spatio-temporal trajectory.

5.1 Trajectory Representation and Optimization

A discretized trajectory $\mathbf{b}$ is defined by an ordered sequence of robot poses $\mathbf{s}_k = [x_k, y_k, \beta_k]^\mathsf{T} \in \mathbb{R}^2 \times S^1$ with $k = 1, 2, \ldots, N$ and time stamps $\Delta T_k \in \mathbb{R}_{>0}$ with $k = 1, 2, \ldots, N-1$.

$$\mathbf{b} = \left[\mathbf{s}_1, \Delta T_1, \mathbf{s}_2, \Delta T_2, \ldots, \Delta T_{N-1}, \mathbf{s}_N\right]^\mathsf{T} \tag{8}$$

ΔT_k denotes the transition time between two consecutive poses $\mathbf{s}_k$ and $\mathbf{s}_{k+1}$, respectively. Figure 5 depicts an example trajectory with three poses. The reference frame of the trajectory representation is denoted as *map*-frame.[13]

The optimal trajectory $\mathbf{b}^*$ is obtained by minimizing a cost function which captures partially conflicting objectives and constraints of motion planning. These objectives include energy consumption, path length, the total transition time, or a weighted

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

combination of the above. Admissible solutions are restricted to a feasible set by penalizing trajectories, which do not comply with the kinodynamic constraints of the mobile robot.

The TEB optimization problem is defined as an aggregated nonlinear least-squares cost function, which considers conflicting sets of objectives $\mathcal{J}$ and penalties $\mathcal{P}$, each weighted by a factor σ_i:

$$\mathbf{b}^* = \underset{\mathbf{b}\setminus\{\mathbf{s}_1,\mathbf{s}_N\}}{\operatorname{argmin}} \sum_i \sigma_i f_i^2(\mathbf{b}), \quad i \in \{\mathcal{J}, \mathcal{P}\} \tag{9}$$

The notation $\mathbf{b} \setminus \{\mathbf{s}_1, \mathbf{s}_N\}$ implies that neither the start pose $\mathbf{s}_1 = \mathbf{s}_s$ nor the goal pose $\mathbf{s}_N = \mathbf{s}_g$ are subject to optimization. During optimization the trajectory is clipped at the current robot pose $\mathbf{s}_s$ and the desired goal pose $\mathbf{s}_g$.

In order to account for the dynamic environment and to refine the trajectory during runtime, a model predictive control scheme is applied. Thus, the optimization problem (9) is solved repeatedly in each sampling interval[14] with respect to the current robot pose and velocity. The current robot pose and velocity are provided by a localization scheme. In compliance with the basic concept of model predictive control [13], during each time step only the first control action of the computed trajectory is commanded to the robot. The *teb_local_planner* pursues a warm-start approach, hence the optimal trajectory of the previous time interval serves as initial solution for the subsequent optimization problem.

In the navigation stack, the *base controller* interface typically subscribes to a *cmd_vel* message (see Fig. 1) composed of translational and angular velocities. These components can easily be extracted from the optimal trajectory $\mathbf{b}^*$ by investigating finite differences both on the position and orientation part. As car-like robots often require the steering angle rather than the angular velocity, the steering angle can be calculated from the turn rate and the car-like robot's kinematic model, e.g. refer to [19].

The TEB optimization problem is mapped onto a hyper-graph in which vertices correspond to the poses $\mathbf{s}_k$ and time intervals ΔT_k that form the solution vector and the (hyper)-edges denote the cost terms f_i that set up the nonlinear program. In addition, fixed vertices not subject to optimization include start and goal pose ($\mathbf{s}_1$ or $\mathbf{s}_N$), obstacle positions $\mathcal{O}_i$, or other static parameters. The prefix *hyper* indicates that an edge connects an arbitrary number of vertices. An edge connecting various parameters represents a cost term, that is dependent on these parameters.

The resulting hyper-graph is efficiently solved by the *g2o-framework*[15] [9]. The framework exploits the sparse structure of the system matrix by using the Levenberg–Marquardt Algorithm. The sparse structure emerges from a formulation that expresses relationships in the solution by soft rather than hard constraints. The computational efficiency of the algorithm benefits from the sparse structure in the

[14] The sampling interval can be adjusted by means of the parameter `controller_frequency` provided by the *move_base* node of the navigation stack.

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

Cholesky-decomposition step. The hyper-graph formulation comes with the additional advantage of modularity which allows the seamless integration of additional constraints and objectives.

Soft constraints tolerate a certain amount of violation at the price of an abrupt increase in the cost function. Let $\mathcal{B}$ denote the entire set of potential trajectories such that $\mathbf{b} \in \mathcal{B}$. An inequality constraint $g_i(\mathbf{b}) \geq a$ with $g_i : \mathcal{B} \rightarrow \mathbb{R}$ is approximated by a positive semi-definite penalty function which captures the amount of constraint violation:

$$f_i(\mathbf{b}) = \max\{0, -g_i(\mathbf{b}) + a + \epsilon\} \qquad \forall i \in \mathcal{P} \tag{10}$$

The parameter ϵ adds a margin to the lower bound a of the inequality constraint such that the cost merely vanishes for $g_i(\mathbf{b}) \geq a + \epsilon$. The theory of penalty optimization methods [14] postulates that the weights of the individual penalty terms should tend towards infinity in order to comply with the truly optimal solution. Unfortunately, large weights result in a numerically ill-conditioned optimization problem for which the underlying solver does not converge properly. For this reason, the TEB approach approximates the optimal trajectory with finite weights in order to achieve a computationally more efficient solution.

The TEB approach employs multiple cost terms f_i for trajectory optimization. For example, limited velocities and accelerations, compliance with non-holonomic kinematics or transition to the goal pose in minimal time are considered. Section 6.3 summarizes the currently implemented cost terms of the optimization problem (9). The TEB cost structure is extended by a novel distinctive penalty term for dynamic obstacles that complements the previous penalty term for static obstacles. The following section considers and analyzes different options for the configuration and parametrization of the cost term for dynamic obstacles.

5.2 Transition Time Estimation for Dynamic Obstacles

Collision avoidance demands a minimal separation of the robot and obstacle poses. The spatio-temporal distance between an obstacle and a pose $\mathbf{s}_k$ along the trajectory is calculated based on the predicted poses of dynamic obstacles at time Δt_k rather than their current location. The robot-obstacle distance calculations consider the polygonal footprints of the robot and the obstacles. Distances between the robot and obstacles are bounded from below by the minimal separation (a) and an additional tolerance value ϵ. Hence, these distances can be directly inserted into the penalty function for inequality constraints (10) along with parameters a and ϵ. The penalty term itself is used in the optimization problem (9) for each penalty and each objective term listed in Sect. 6.3.

Under the assumption that dynamic obstacles $\mathcal{O}_i$ maintain their current (estimated) speed $\hat{v}$ and orientation the constant velocity model predicts the future obstacle poses $\hat{P}(\mathcal{O}_i,)$.

$$\hat{P}(\mathcal{O}_i, t) = P(\mathcal{O}_i, t_0) + \Delta t_k \cdot \hat{v}(\mathcal{O}_i) \tag{11}$$

The current obstacle position $P(\mathcal{O}_i, t_0)$ and the estimated velocity $\hat{v}(\mathcal{O}_i)$ are provided by the *costmap_converter* package. The predicted obstacle pose depends on the robot's transition time Δt_k between its current and its kth future pose.

This transition time amounts to the accumulated time steps $\sum_{i=1}^{k} \Delta T_i$ up to pose $\mathbf{s}_k$. However, the dependency of the total transition time Δt_k on all previous time intervals ΔT_i with $i \leq k$ compromises the sparsity of the system matrix which causes a degradation of computationally efficiency.

In order to preserve computational efficiency an improved strategy exploits the iterative nature of the online trajectory optimization. Changes of the environment and the underlying nonlinear program between two consecutive control cycles are rather small. Thus it is valid to approximate the true total transition time Δt_k by summation of the time intervals $\Delta T_k'$ in the previous optimization step and consider the Δt_k as constant within the obstacle pose prediction step. This approximation maintains the sparse structure of the system matrix and causes a significant improvement in performance.

5.3 Planning in Distinctive Spatio-Temporal Topologies

The previously introduced TEB approach is subject to local optimization. Instead of finding the globally optimal solution, the optimized trajectory might get stuck in local minima due to the presence of obstacles. Identifying these local minima coincides with exploring and analyzing distinctive topologies between start and goal poses. For instance, the robot might circumnavigate an obstacle either on the left- or right-hand side. In case of moving obstacles the concept of a traversal to left and right are augmented by a traversal before or after passage of a dynamic obstacle. Therefore the new trajectory optimization not only considers the spatial topology of the trajectory but also its temporal dimension.

The approach rests upon two theorems of electromagnetism, the Biot–Savart and Ampere's law. It defines a new equivalence relation in order to distinguish among trajectories of distinctive topologies in the three-dimensional x-y-t-space. Notice, the additional temporal dimension that extends the mere spatial analysis of the previous TEB trajectory optimization [18]. The TEB ROS implementation explores and optimizes multiple trajectories in distinctive spatio-temporal topologies in parallel and selects the best candidate trajectory at each sampling interval. However, the theory of this method is beyond the scope of this tutorial. For a detailed description of this approach, the reader is referred to [1].

6 teb_local_planner ROS Package

This section provides an overview of the *teb_local_planner* ROS package which employs the previously described TEB approach for online trajectory optimization. Since the prerequisites and basics of the *teb_local_planner* did not change since the release of the last volume of this book, some fundamental parts from [18] are only briefly revisited.

6.1 Prerequisites and Installation

To install and configure the *teb_local_planner* package for a particular application, comply with the following limitations and prerequisites:

- Although current online trajectory optimization approaches feature mature computational efficiency, their application still requires 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 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 theoretical and the planner is confronted with e.g., sensor and actuator uncertainty and dynamic environments in real applications, finding a feasible and stable trajectory in every conceivable scenario is not guaranteed. Especially due to noisy velocity estimations for dynamic obstacles, planned trajectories may oscillate. In order to generate a feasible trajectory, the planner detects and resolves failures by post-introspection of the optimized trajectory.
- Even though the presentation focuses on differential-drive robots, the package currently supports car-like and omnidirectional robots as well. For a detailed tutorial on the setup and configuration of car-like robots, the reader is referred to [18].
- Officially supported ROS distributions are Kinetic and Lunar. Legacy versions of the planner without support for dynamic obstacles are also available in Indigo and Jade. Support of future distributions is expected. The package is released for both default and ARM architectures.

Similar to the installation of the *costmap_converter* package, the *teb_local_planner* is installed from the official ROS repositories by invoking:

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

As before, the distribution name `kinetic` should be adapted to match the currently installed distribution. As an alternative, compile the most recent versions of the *teb_local_planner* 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
```

6.2 Integration with ROS Navigation

As a plugin for the ROS navigation stack, the *teb_local_planner* package replaces the navigation stack's default local planner (refer to Fig. 1). The global planner of the *move_base* package plans a global path to the goal according to a global costmap which corresponds to a known map of the environment and serves as initial solution for the local planner. However, the global costmap does not support dynamic obstacles. Therefore, robocentric sensor readings are fused with the global costmap in order to calculate a local costmap. The *teb_local_planner* computes a feasible trajectory corresponding to the local costmap and publishes the associated velocity commands. The robot is localized with respect to the global map by means of the *amcl* node which employs an adaptive monte carlo localization algorithm which compensates for the accumulated odometric error.

The package wiki page[16] contains the complete list of *teb_local_planner* parameters which are nested in the relative namespace of the *move_base* node, e.g. `/move_base/TebLocalPlannerROS/param_name`. Assuming a running instance of the *teb_local_planner*, parameters are configured at runtime by launching the *rqt_reconfigure* GUI:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

6.3 Included Cost Terms: Objectives and Penalties

The *teb_local_planner* optimizes the planned trajectory respectively current control commands by minimizing a designated cost function (9) composed of objective and penalty terms which are approximated by a penalty function (10). For a detailed explanation of the mathematics behind these cost terms the reader is referred to [17]. Currently implemented cost terms f_i, their respective weights σ_i and related ROS

[16] teb_local_planner, http://wiki.ros.org/teb_local_planner.

parameters are summarized below according to [18]. A distinction is made between alternative cost functions for static and dynamic obstacles.

Minimal Total Transition Time (Objective)
Description: Minimizes the total transition time to the goal in order to find a time-optimal solution.
Weight parameter: `weight_optimaltime`
Related parameters: `selection_alternative_time_cost`

Via-points (Objective)
Description: Minimizes the distance to specific via-points which define attractors for the trajectory.
Weight parameter: `weight_viapoint`
Related parameters: `global_plan_viapoint_sep`

Compliance with Non-holonomic Kinematics (Objective)
Description: Enforces the geometric constraint for non-holonomic robots that requires two consecutive poses $\mathbf{s}_k$ and $\mathbf{s}_{k+1}$ to be located on a common arc of constant curvature. Actually the compliance is an equality constraint ensured by a large weight rather than an objective.
Weight parameter: `weight_kinematics_turning_radius`
Related parameters: `min_turning_radius`

Limiting Translational Velocity (Penalty)
Description: Constrains the translational velocity v_k to the interval $[-v_{back}, v_{max}]$. v_k is computed for each time interval ΔT_k with $\mathbf{s}_k, \mathbf{s}_{k+1}$ by means of finite differences.
Weight parameter: `weight_max_vel_x`
Related parameters: `max_vel_x` (v_{max}), `max_vel_x_backwards` (v_{back})

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

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

Limiting Angular Acceleration (Penalty)
Description: Constrains the angular acceleration to $|\dot{\omega}_k| \leq \dot{\omega}_{max}$ by means of finite differences.

Weight parameter: `weight_acc_lim_theta`
Related parameters: `acc_lim_theta` ($\dot{\omega}_{max}$)

Limiting the Minimum Turning Radius (Penalty)
Description: This penalty enforces a minimum turning radius designated for car-like robots with limited steering angles. Differential drive robots are able to turn in place with a turning radius $r_{min} = 0$.
Weight parameter: `weight_kinematics_turning_radius`
Related parameters: `min_turning_radius` (r_{min})

Penalizing Backward Motions (Penalty)
Description: This cost term reflects a bias for forward motions, even though small weights still allow backward motions.
Weight parameter: `weight_kinematics_forward_drive`

Limiting Distance to Static Obstacles (Penalty)
Description: Enforces a minimum separation d_{min} of poses along the planned trajectory to a static obstacle (incorporates the robot footprint). In addition, obstacle inflation considers a buffer zone (larger than d_{min} in order to take effect) around an obstacle.
Weight parameters: `weight_obstacle`, `weight_obstacle_inflation`
Related parameters: `min_obstacle_dist` (d_{min}), `inflation_dist`

Limiting Distance to Predicted Positions of Dynamic Obstacles (Penalty)
Description: Enforces a minimum separation d_{min} of poses along the planned trajectory to the dynamic obstacle pose predicted according to a constant velocity model. The distance refers to the obstacle pose at the corresponding time step according to closest separation of the robot footprint and the most recent estimated obstacle shape. Again, obstacle inflation accounts for a buffer zone. The obstacle motion prediction is disabled by the parameter `include_dynamic_obstacles` which causes all obstacles to be considered as static.
Weight parameters: `weight_dynamic_obstacle`, `weight_dynamic_obstacle_inflation`
Related parameters: `min_obstacle_distance` (d_{min}), `dynamic_obstacle_inflation_dist`, `include_dynamic_obstacles`

7 Testing Spatio-Temporal Trajectory Optimization

The *teb_local_planner* package includes a basic test node (*test_optim_node*) for testing and analysis of the trajectory optimization between a fixed start and goal pose. It supports parameter configurations and performance validation on the target

hardware. Obstacles are represented by interactive markers[17] and can be animated with the *rviz* GUI.

The *teb_local_planner* package provides a launch file in order to start up the *test_optim_node* along with a preconfigured *rviz* node:

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

Rviz shows the planned trajectories and obstacles. The *teb_local_planner* plans multiple trajectories in parallel and selects the optimal trajectory indicated by red arrows. Select the menu button *interact* to move the obstacles around and observe the optimization and reconfiguration of the planned trajectories. Trajectory optimization is customized at runtime with:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Select the *test_optim_node* from the list of available nodes and enable dynamic obstacles with the parameter `include_dynamic_obstacles`. Furthermore, set the parameter `visualize_with_time_as_z_axis_scale` to some positive number, e.g. 0.2 (refer to Fig. 6). The temporal evolution of the trajectory and obstacle configuration (indicated by red line markers) is displayed along the z-axis according to the scaling factor. Currently, the obstacles are static and maintain their pose.

Obstacles in the *test_optim_node* subscribe to individual topics, which specify their velocities in the *map*-frame. The velocity of an obstacle is defined by publishing a `geometry_msgs/Twist` message to the corresponding topic:

```
$ rostopic pub --once /test_optim_node/obstacle_0/cmd_vel
    geometry_msgs/Twist  '{linear:  {x: 0.2, y: 0.3, z: 0.0},
    angular: {x: 0.0, y: 0.0, z: 0.0}}'
```

In order to define the velocities of the remaining obstacles, replace the respective obstacle number. Admissible topics can be listed with:

```
$ rostopic list /test_optim_node/ | grep obstacle_
```

Only the translational x- and y-components of the message are considered. The planned trajectory avoids dynamic obstacles according to the predictions of their future positions. The current prediction operates with the ground-truth velocities provided by the message published previously. In case of estimated obstacle velocities, these predictions are obviously less accurate, especially for remote poses. This causes the planned trajectories to oscillate in particular as the uncertainties of pose estimates increase with the transition time.

[17] *Interactive markers*, http://wiki.ros.org/interactive_markers.

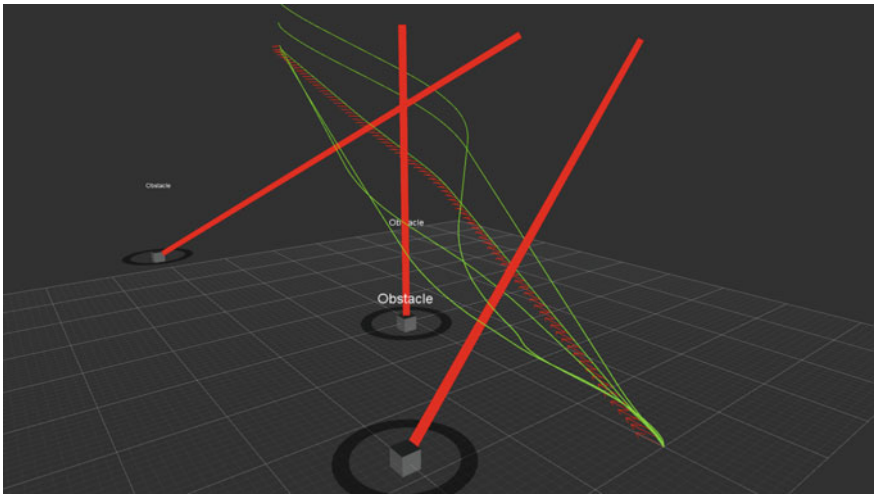


Fig. 6 Spatio-temporal obstacle avoidance in *rviz*

Switch back to the *rqt_reconfigure* GUI and customize the optimization with different parameter settings. Adjust parameters one at a time, as some parameters significantly influence the optimization performance. In case of insufficient performance on your target system, either decrease the parameters `no_inner_iterations` or `no_outer_iterations` to reduce the number of executed inner respectively outer iterations per TEB optimization step. As an alternative, increase the reference time step `dt_ref` slightly to obtain a coarser trajectory with fewer poses. These workarounds may deteriorate the optimality of the planned trajectory.

8 Obstacle Motion Predictive Planning

This section introduces a simple albeit challenging scenario for local path planning algorithms in a highly dynamic environment and illustrates the benefits of spatio-temporal trajectory planning. The scenario mimics an arcade game in which the robot traverses a corridor while evading intruders. Figure 7 illustrates the obstacles moving back and forth across the corridor along parallel paths with a constant velocity. Note, that the intruder motion proceeds blindly in a purely open loop manner. The obstacle velocities are estimated from the costmap conversion and do not reflect ground truth velocities. The analysis compares the static *teb_local_planner* with its dynamic extension.

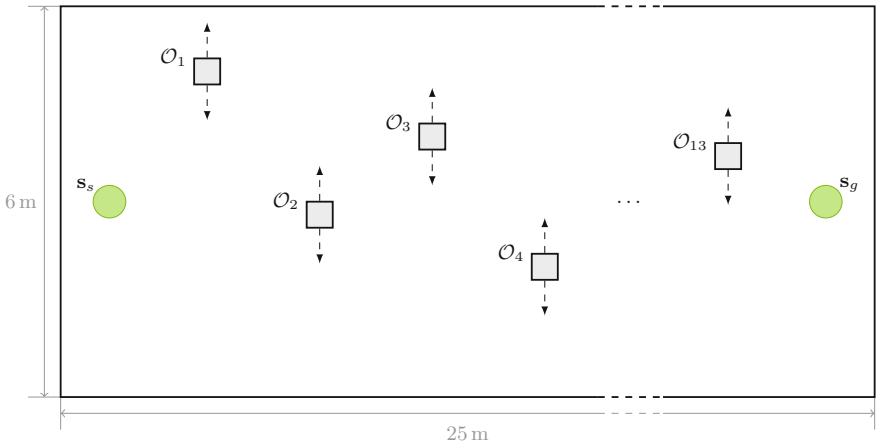


Fig. 7 Setup of the corridor scenario

In the following details of the launch file parameters and configuration are explained.

```
<!--  ************** Global Parameters ***************  -->
<param name="/use_sim_time" value="true"/>
```

The global parameter `use_sim_time` enables ROS to operate with a simulated clock rather than the CPU system time.

```
<!--  ************** Stage Simulator *****************  -->
<node pkg="stage_ros" type="stageros" name="stageros" args="$(find
    dynamic_obstacle_test)/stage/corridor.world">
  <remap from="/robot_0/base_scan" to="/robot_0/scan"/>
</node>
```

This command starts up the *stage* node and loads the environment configuration from *corridor.world* file.

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

This code segment starts the *map_server* node with the parameters defined in the *corridor.yaml* file.

```
<!--  ************** Navigation ROBOT 0 *************  -->
<group ns="robot_0">
  <param name="tf_prefix" value="robot_0"/>

  <node pkg="tf" type="static_transform_publisher" name="
    link1_broadcaster" args="2 3 0 0 0 0 1 /map /robot_0/odom 100" />

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

    <!-- Here we load our costmap conversion settings -->
    <!-- If you uncomment the following line, disable the
    ground_truth_obstacles node at the bottom of this script! -->
    <!-- rosparam file="$(find teb_local_planner_tutorials)/cfg/
    diff_drive/costmap_converter_params.yaml" command="load" /-->

    <param name="TebLocalPlannerROS/include_costmap_obstacles" value="
    False" />
    <param name="TebLocalPlannerROS/include_dynamic_obstacles" value="
    True" />

    <param name="base_global_planner" value="navfn/NavfnROS" />
    <!--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" />
    <remap from="map" to="/map"/>
  </node>
</group>
```

This code initializes the navigation stack and launches the *move_base* node in the namespace of `robot_0` (which corresponds to the observing robot). Several parameters regarding the local and global costmap are loaded. The utilized plugin for the *costmap_converter* is specified in the *costmap_converter_params.yaml* file. The *teb_local_planner* plugin replaces the default local planner.

```
<!-- ****************** Obstacles ******************* -->
<group ns="robot_1">
  <param name="tf_prefix" value="robot_1"/>
  <node name="Mover" pkg="dynamic_obstacle_test" type="move_obstacle.py"
      output="screen"/>
  <param name="pos_ub" value="5.0" />
  <param name="pos_lb" value="1.0" />
  <param name="vel_y" value="0.3" />
</group>
```

Intruder obstacles are set in motion by a `cmd_vel` message published by the *Mover* node in the obstacle's namespace. The code is replicated for each obstacle with its respective namespace. The *Mover* node in *move_obstacle.py* subscribes to the `base_pose_ground_truth` topic published by *stage*. Each obstacle moves back and forth between `pos_lb` and `pos_ub` with the velocity `vel_y`. In case no velocity `vel_y` is specified, a random velocity is sampled at each turnaround.

```
<!--  ***************** Visualisation ****************  -->
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find
    teb_local_planner_tutorials)/cfg/rviz_navigation_cc.rviz">
  <remap from="/move_base_simple/goal" to="/robot_0/move_base_simple/
    goal" />
</node>
```

This code launches the visualization tool *rviz* and loads a configuration file which contains some predefined *rviz* displays, e.g. the robots footprint.

```
<!--  ************ Ground Truth Obstacles ************  -->
<node name="ground_truth_obstacles" pkg="dynamic_obstacle_test" type="
    publish_ground_truth_obstacles.py" output="screen" />
```

This code segment launches the node that publishes the ground truth velocities of the obstacles. The node subscribes to the `base_pose_ground_truth` topic and publishes an `ObstacleArrayMsg` further processed by the *teb_local_planner*. Ground truth velocities of the obstacles are only utilized for comparison with the ideal optimal collision avoidance maneuvers. Launch the simulation:

```
$ roslaunch teb_local_planner_tutorials corridor_scenario.
    launch
```

rviz and *stage* show the simulation setup. The dynamic obstacles move across the corridor. By default, dynamic obstacles are not explicitly considered during the trajectory optimization. In order to monitor the reference trajectory planning without motion prediction for dynamic obstacles, publish a navigation goal at the end of the corridor using the '2D Nav Goal' button in *rviz*. Observe the robots behavior when obstacles cross its way. The robot responds only until the obstacle is located directly in front of the robot. The static perspective severely compromises the planning due to the mismatch of the static environment and its true status within the evolving scenario. As a consequence, the robot merely relies on reactive control and exhibits detours. In many cases, the planned trajectory lacks robustness frequently causing collisions with the intruders. Close the simulation (*Ctrl+C*).

Relaunch the simulation and enable obstacle motion predictive planning in the *rqt_reconfigure* tool with the parameter `use_dynamic_obstacles`. Additionally, the temporal dimension of the planned trajectory can be visualized as z-axis in *rviz* by means of the parameter `visualize_with_time_as_z_axis_scale`. Again, publish a navigation goal at the end of the corridor and observe the planned trajectories. The robot avoids the obstacles ahead of time which not only significantly reduces the total transition time to the goal but also results in less frequent collisions. Customize trajectory planning with different optimization weights and adapt the minimal distance to dynamic obstacles (`min_obstacle_distace`).

Activate costmap conversion by uncommenting the line loading the costmap conversion settings in the *corridor_scenario.launch* file:

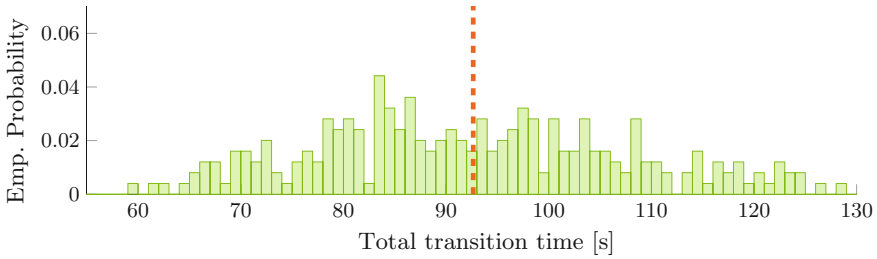
```
<rosparam file="$(find teb_local_planner_tutorials)/cfg/diff_drive/
    costmap_converter_params.yaml" command="load" />
```

Disable the node that publishes ground truth data about the obstacles from the simulation setup by removing the following lines from the launch file *corridor_scenario.launch*:

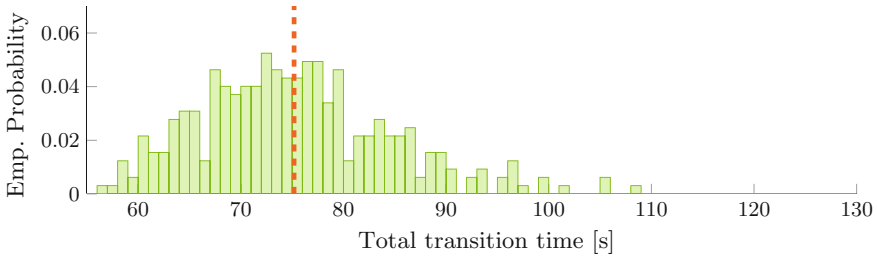
```
<!--  ************ Ground Truth Obstacles ************  -->
<node name="ground_truth_obstacles" pkg="dynamic_obstacle_test" type="
    publish_ground_truth_obstacles.py" output="screen" />
```

Relaunch the simulation, enable `use_dynamic_obstacles` in the *rqt_reconfigure* tool, and publish a navigation goal at the end of the corridor using the '2D Nav Goal' button. Footprints of the observed obstacles are visualized as red polygons. Notice, the slight oscillations of the planned trajectory attributed to the uncertainty of obstacle velocity estimations. Nevertheless, the robot avoids most obstacles merely based on their predicted movements.

The enhanced trajectory planning of the *teb_local_planner* in dynamic environments due to the incorporation of estimated obstacle velocities is confirmed in simulations. Figure 8 depicts a comparison of the total transition times between the legacy and the current version of the *teb_local_planner*. Compared to the default configuration, these results were recorded with a slightly increased minimum distance to obstacles in order to increase the influence of the rather small obstacles. In the introduced corridor scenario with random initial obstacle positions, the mean transition time can be reduced by more than 20 s. Additionally, the prediction of obstacle movement results in a more robust trajectory which is shown by the reduced collision probability (see Table 1). Note, that the robot does not collide actively, but rather passively due to too optimistic trajectory planning and the blind motion of obstacles. Collisions occurring with the current version of the *teb_local_planner* are mostly the result of abrupt changes in direction performed by dynamic obstacles at corridor walls. In these cases the planned trajectory avoids the current motion of an obstacle, but does not account for the sudden change in the obstacles movement direction.



(a) Legacy version of the *teb_local_planner*



(b) *teb_local_planner* with dynamic obstacle support

Fig. 8 Comparison of total transition times

Table 1 Comparison of total transition times and collision probabilities

	Legacy version	Current version
Number of simulations	368	470
Number of collisions	117	96
Emp. collision probability	0,318	0,205
Emp. mean transition time [s]	92,64	71,00
Standard deviation [s]	15,71	8,41

9 Conclusion

This tutorial chapter provides a comprehensive step-by-step guide for the setup of the *teb_local_planner* and *costmap_converter* ROS packages for mobile robot navigation with explicit consideration of dynamic obstacles. The literature contains several advanced approaches for optimal trajectory planning with motion prediction of dynamic obstacles. However, these algorithms are currently not compatible with the static local costmap representation of the ROS navigation stack. The proposed costmap conversion exploits the established architecture of the navigation stack and estimates velocities, orientations, and shapes of dynamic obstacles only based on information on the local costmap. These informations are incorporated into the Timed-Elastic-Band approach to calculate optimal spatio-temporal trajectories in the presence of dynamic obstacles. Besides its ability to include dynamic obstacles

explicitly into trajectory optimization, to our best knowledge, the *teb_local_planner* is currently the only local planner plugin for the ROS navigation stack, which is capable of trajectory planning for car-like robots without the use of an additional low-level controller.

Future work addresses the automatic tuning of cost function weights in order to improve the overall performance of the *teb_local_planner*. Furthermore, a benchmark suite for the comparative analysis of ROS local planners would be a valuable addition to ROS in order to support users in their decision for the local planner most suitable for their particular mobile robot applications.

References

1. S. Bhattacharya, M. Likhachev, V. Kumar, Identification and representation of homotopy classes of trajectories for search-based path planning in 3D. in *Proceedings of Robotics: Science and Systems* (2011)
2. V. Delsart, T. Fraichard, Reactive trajectory deformation to navigate dynamic environments. in *European Robotics Symposium* (2008), pp. 233–241
3. P. Fiorini, Z. Shiller, Motion planning in dynamic environments using velocity obstacles. Int. J. Robot. Res. **17**(7), 760–772 (1998)
4. D. Fox, W. Burgard, S. Thrun, The dynamic window approach to collision avoidance. IEEE Robot. Autom. Magaz. **4**(1), 23–33 (1997)
5. C. Fulgenzi, A. Spalanzani, C. Laugier, Dynamic obstacle avoidance in uncertain environment combining PVOs and occupancy grid. in *IEEE International Conference on Robotics and Automation (ICRA)* (2007)
6. B. Gerkey, K. Konolige, Planning and control in unstructured terrain. in *Proceedings of the ICRA Workshop on Path Planning on Costmaps* (2008)
7. T. Gu, J. Atwood, C. Dong, J. M. Dolan, J.-W. Lee, Tunable and stable real-time trajectory planning for urban autonomous driving. in *IEEE International Conference on Intelligent Robots and Systems (IROS)* (2015), pp. 250–256
8. H.W. Kuhn, The Hungarian method for the assignment problem. Nav. Res. Logist. Q. **2**, 83–97 (1955)
9. R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, W. Burgard, G2o: a general framework for graph optimization. in *IEEE International Conference on Robotics and Automation (ICRA)* (2011), pp. 3607–3613
10. H. Kurniawati, T. Fraichard, From path to trajectory deformation. in *IEEE/RSJ International Conference on Intelligent Robots and Systems (RISO)* (2007), pp. 159–164
11. M. Luber, A. Stork, G.D. Tipaldi, K.O. Arras, People tracking with human motion predictions from social forces. in *IEEE International Conference on Robotics and Automation (ICRA)* (2010), pp. 464–469
12. E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, K. Konolige, The office marathon: robust navigation in an indoor office environment. in *IEEE International Conference on Robotics and Automation (ICRA)* (2010)
13. M. Morari, J.H. Lee, Model predictive control: past, present and future. Comput. Chem. Eng. **23**(4–5), 667–682 (1999)
14. J. Nocedal, S.J. Wright, *Numerical Optimization*, Operations Research (Springer, New York, 1999)
15. K. Rebai, O. Azouaoui, M. Benmami, A. Larabi, Car-like robot navigation at high speed. in *IEEE International Conference on Robotics and Biomimetics (ROBIO)* (2007), pp. 2053–2057
16. C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, T. Bertram, Trajectory modification considering dynamic constraints of autonomous robots. in *7th German Conference on Robotics (ROBOTIK)* (2012), pp. 74–79

17. C. Rösmann, F. Hoffmann, T. Bertram, Integrated online trajectory planning and optimization in distinctive topologies. Robot. Auton. Syst. **88**, 142–153 (2017)
18. C. Rösmann, F. Hoffmann, T. Bertram, Online trajectory planning in ROS under kinodynamic constraints with timed-elastic-bands, *Robot Operating System (ROS) - The Complete Reference 2*, vol. 707, Studies in Computational Intelligence (Springer International Publishing, 2017)
19. C. Rösmann, F. Hoffmann, T. Bertram, Kinodynamic trajectory optimization and control for car-like robots. in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), pp. 5681–5686
20. C. Rösmann, M. Oeljeklaus, F. Hoffmann, T. Bertram, Online trajectory prediction and planning for social robot navigation. in *IEEE International Conference on Advanced Intelligent Mechatronics (AIM)* (2017), pp. 1255–1260
21. M. Seder, I. Petrović, Dynamic window based approach to mobile robot motion control in the presence of moving obstacles. in *IEEE International Conference on Robotics and Automation (ICRA)* (2007), pp. 1986–1991
22. S. Thrun, D. Fox, W. Burgard, F. Dellaert, Robust Monte Carlo localization for mobile robots. Artif. Intell. **128**, 99–141 (2001)
23. S. Quinlan, O. Khatib, Elastic bands: connecting path planning and control. in *IEEE International Conference on Robotics and Automation (ICRA)* (1993), pp. 802–807

Franz Albers received his B.Sc. and M.Sc. degree in electrical engineering and computer science from the Technische Universität Dortmund, Germany, in 2015 and 2017, respectively. He is currently working as a Dr.-Ing. candidate at the Institute of Control Theory and Systems Engineering, Technische Universität Dortmund, Germany. His research interests include trajectory planning and automotive systems.

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 the 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, Germany, in 1990 and 1995, respectively. In 1990, he joined the Gerhard Mercator Universität 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, Germany, and, since 2005, he has been a member of the Department of Electrical Engineering and Information Technology, Technische Universität 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.