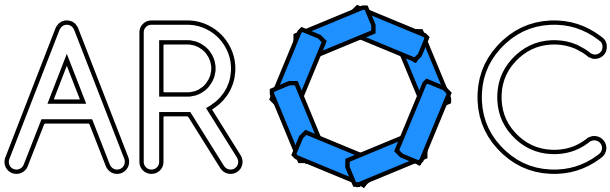
AROC 2022 Manual

Niklas Kochdumper, Felix Gruber, Bastian Schürmann, Victor Gaßmann,
Moritz Klischat, Lukas Schäfer, and Matthias Althoff

Technische Universität München, 85748 Garching, Germany



Abstract

In this manual we present the **A**utomated **R**eachset **O**ptimal **C**ontrol (AROC) toolbox. AROC is a MATLAB toolbox that automatically synthesizes verified controllers for solving reach-avoid problems using reachability analysis. Two different types of controllers are considered: For *model predictive control* verified controllers are constructed in real-time during online application; The *motion primitive based control* algorithms, on the other hand, first synthesize verified controllers for many different motion primitives offline, which are then used for online planning with a maneuver automaton. AROC contains one model predictive control algorithm for linear systems and one for nonlinear systems, and also implements several approaches for computing safe terminal regions for model predictive control. Moreover, the toolbox currently contains 6 different methods for motion primitive based control, and also provides an implementation of a maneuver automaton for convenient online-planning with motion primitives. Yet another feature of AROC is that it includes an implementation of conformant synthesis to automatically construct over-approximative models from data. AROC is released under the GPLv3 license.

Contents

1 Introduction **4**
- 1.1 Getting Started . . . 4
- 1.2 Installation . . . 5
- 1.3 New Features . . . 5
- 1.4 Architecture . . . 6
- 1.5 Code Documentation . . . 7
- 1.6 Unit Tests . . . 7
- 1.7 Connections to CommonRoad and CommonOcean . . . 7

2 Control Algorithms **9**
- 2.1 Motion Primitive Based Control . . . 10
  - 2.1.1 Optimization Based Control . . . 11
  - 2.1.2 Convex Interpolation Control . . . 12
  - 2.1.3 Generator Space Control . . . 14
  - 2.1.4 Polynomial Control . . . 15

2.1.5 Combined Control . . . 17
2.1.6 Safety Net Control . . . 19
2.2 Model Predictive Control . . . 21
2.2.1 Reachset Model Predictive Control . . . 22
2.2.2 Model Predictive Control for Linear Systems . . . 24

3 Maneuver Automata **26**
3.1 Class `maneuverAutomaton` . . . 26
3.2 Function `postprocessing` . . . 27
3.3 Function `shiftInitSet` . . . 27
3.4 Function `shiftOccupancySet` . . . 28
3.5 Motion Planner . . . 28

4 Terminal Region **30**
4.1 Subpaving Algorithm . . . 30
4.2 Zonotope Approach for Linear Systems . . . 32

5 Conformant Synthesis **34**

6 Benchmarks **36**
6.1 Double Integrator . . . 36
6.2 Cart . . . 36
6.3 Cartpole . . . 37
6.4 Stirred Tank Reactor . . . 37
6.5 Artificial System . . . 38
6.6 Car . . . 39
6.7 Truck . . . 39
6.8 Quadrotor 2D . . . 40
6.9 Ship . . . 41
6.10 Robot Arm . . . 42
6.11 Mobile Robot . . . 43
6.12 Platoon . . . 44

7 Additional Functionality **45**
7.1 Adding New Benchmark Systems . . . 45
7.2 Class `results` . . . 45
7.2.1 Function `plotReach` . . . 46
7.2.2 Function `plotReachTimePoint` . . . 46
7.2.3 Function `plotSimulation` . . . 47
7.2.4 Function `animate` . . . 47
7.3 Class `objController` . . . 47
7.3.1 Function `simulate` . . . 48
7.3.2 Function `simulateRandom` . . . 48
7.4 Class `terminalRegion` . . . 49
7.4.1 Function `simulate` . . . 49
7.4.2 Function `simulateRandom` . . . 49
7.5 Reference Trajectory . . . 49
7.6 Extended Optimization Horizon . . . 50
7.7 Reachability Settings . . . 52
7.8 Adding Custom Comfort Controllers . . . 53

8 Examples **55**
8.1 Example Motion Primitive Based Control . . . 55

8.2 Example Model Predictive Control 56
8.3 Example Maneuver Automaton 58
8.4 Example Terminal Region 60
8.5 Example Conformant Synthesis 62

1 Introduction

In this section we give a short introduction to the philosophy and architecture of the AROC toolbox, we describe how AROC can be installed, and we explain how to connect AROC with other tools.

1.1 Getting Started

The acronym AROC stands for **A**utomated **R**eachset **O**ptimal **C**ontrol. AROC is a toolbox for the automated construction of verified controllers for solving reach-avoid problems. A typical reach-avoid problem is shown in Fig. 1: Given a set of initial states $\mathcal{R}_0$ the goal is to construct a controller that drives all states inside the initial set as close as possible to a desired final state x_f while not colliding with the sets of unsafe sets depicted in red in Fig. 1. For the system dynamics, we consider the very general case of nonlinear systems with input constraints that are influenced by bounded uncertainties (see (1)). To verify that the system does not collide with any unsafe set and that the input constraints are satisfied for all times despite disturbances are acting on the system, we use reachability analysis. In particular, we use the CORA [1] toolbox to compute reachable sets.

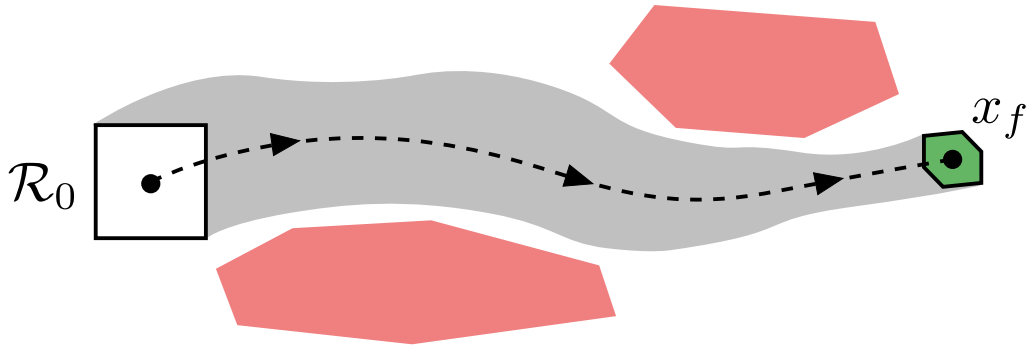


Figure 1: Illustration of a typical reach-avoid problem, where the unsafe sets are depicted in red, $\mathcal{R}_0$ is the initial set, x_f is the goal state that should be reached, and the reachable set of the controlled system is shown in gray.

AROC considers two different types of controllers for solving reach-avoid problems: For *model predictive control* (see Sec. 2.2) a verified controller that steers the system into a safe terminal region (see Sec. 4) is constructed in real-time during online application; The *motion primitive based control* algorithms (see Sec. 2.1), on the other hand, construct verified controllers for many different motion primitives offline, which are the used for online-planning with a maneuver automaton (see Sec. 3). To guarantee that the synthesized controllers are safe for the real system and not just the model, we require a conformant model that over-approximates all behaviours of the real system. Such a model can be constructed automatically from measurements of the real system using conformant synthesis (see Sec. 5). An overview of the workflow for controller synthesis using the AROC toolbox is shown in Fig. 2.

The AROC toolbox provides some predefined benchmark systems (see Sec. 6), and additional custom benchmarks can be easily added (see Sec. 7.1). To get started with AROC, we recommend to read the mathematical problem description at the beginning of Sec. 2, and to take a look at the code examples that are provided in Sec. 8, which can also be found in the directory */examples/...* in the AROC toolbox.

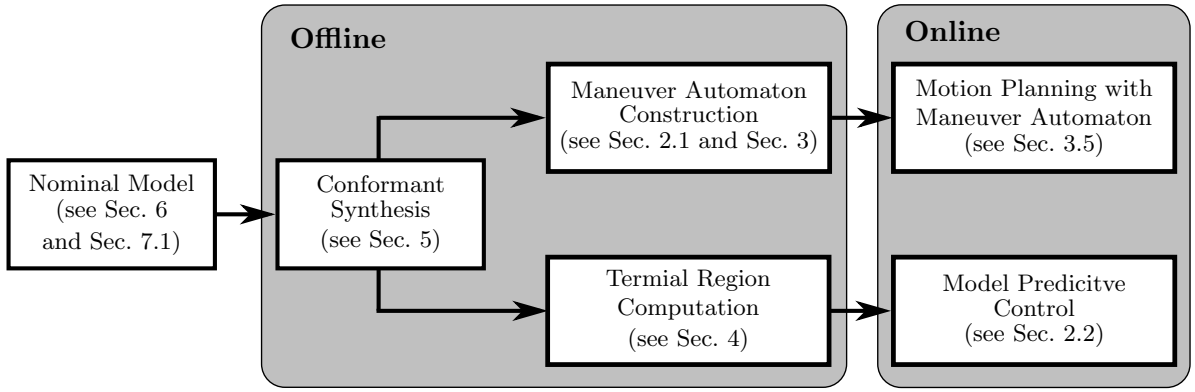


Figure 2: Workflow for controller synthesis using the AROC toolbox.

1.2 Installation

The AROC toolbox can be conveniently installed by simply adding the directory that contains the code to the MATLAB path. In addition, AROC requires the following third-party software:

- **CORA**: CORA is a MATLAB toolbox for reachability analysis. AROC is compatible with the 2022 release of CORA, which can be downloaded from the website `http://cora.in.tum.de` or the public repository `https://github.com/TUMcps/CORA`. After the download, add the folder containing the CORA toolbox to your MATLAB path.
- **ACADO**: ACADO is a C++ toolbox for solving optimal control problems. AROC requires the MATLAB interface of the ACADO toolbox, which can be found at `http://acado.github.io/matlab_overview.html`. AROC also works if ACADO is not installed, but the computations might be significantly slower.
- **MPT** and **YALMIP**: MPT is a toolbox for geometric computations that is used by the CORA toolbox, and YALMIP is a toolbox for solving optimization problems of various types. MPT and YALMIP can be conveniently installed together using the installation routine described in `https://www.mpt3.org/Main/Installation`.

After installation it is advisable to run the unit-tests (see Sec. 1.6) to check if everything is set-up correctly.

1.3 New Features

The 2022 release of AROC contains several new features compared to the 2020 release:

- **Terminal regions**: AROC now includes algorithms for calculating safe terminal regions for model predictive control (see Sec. 4).
- **Conformant synthesis**: Conformant models enclosing all possible behaviours of the real system can now be automatically constructed from measurements of the real system using the conformant synthesis algorithm provided by AROC (see Sec. 5).
- **New algorithms for motion primitive based control**: We added several new approaches for synthesising controllers for single motion primitives, including a polynomial controller (see Sec. 2.1.4), a controller combining feed-forward and feedback control (see Sec. 2.1.5), and a safety net controller that can be used to shield unsafe comfort controllers (see Sec. 2.1.6).
- **Linear model predictive control**: AROC now contains a specialized model predictive control algorithm for linear systems (see Sec. 2.2.2)

- **Measurement errors**: All control algorithms in AROC now support systems with measurement errors.
- **New benchmarks**: We added several new benchmarks, including a cartpole system, an autonomous truck, a 2D quadrotor, and a ship (see Sec. 6).
- **Animations**: AROC can now produce animations which visualize how the control action influences the system.

1.4 Architecture

A UML class diagram for the AROC toolbox is shown in Fig. 3: All motion primitive based control algorithms return an object that inherits certain properties and methods from the parent class `objController`. These objects store the parameter of the motion primitive, the constructed controller, and the occupancy set (see Sec. 7.3). Since the class `maneuverAutomaton` requires a list of motion primitives represented as objects of class `objController` as input argument (see Sec. 3), it is therefore possible to construct a maneuver automaton with any of the implemented motion primitive based controllers, or even mix motion primitives generated with different controllers. Similarly, all objects representing terminal regions inherit from the common parent class `terminalRegion` (see Sec. 7.4), which ensures that terminal regions constructed with different approaches can be used for model predictive control. The class `results` stores the reachable sets computed during controller synthesis and simulated trajectories from the online application of the control algorithm (see Sec. 7.2).

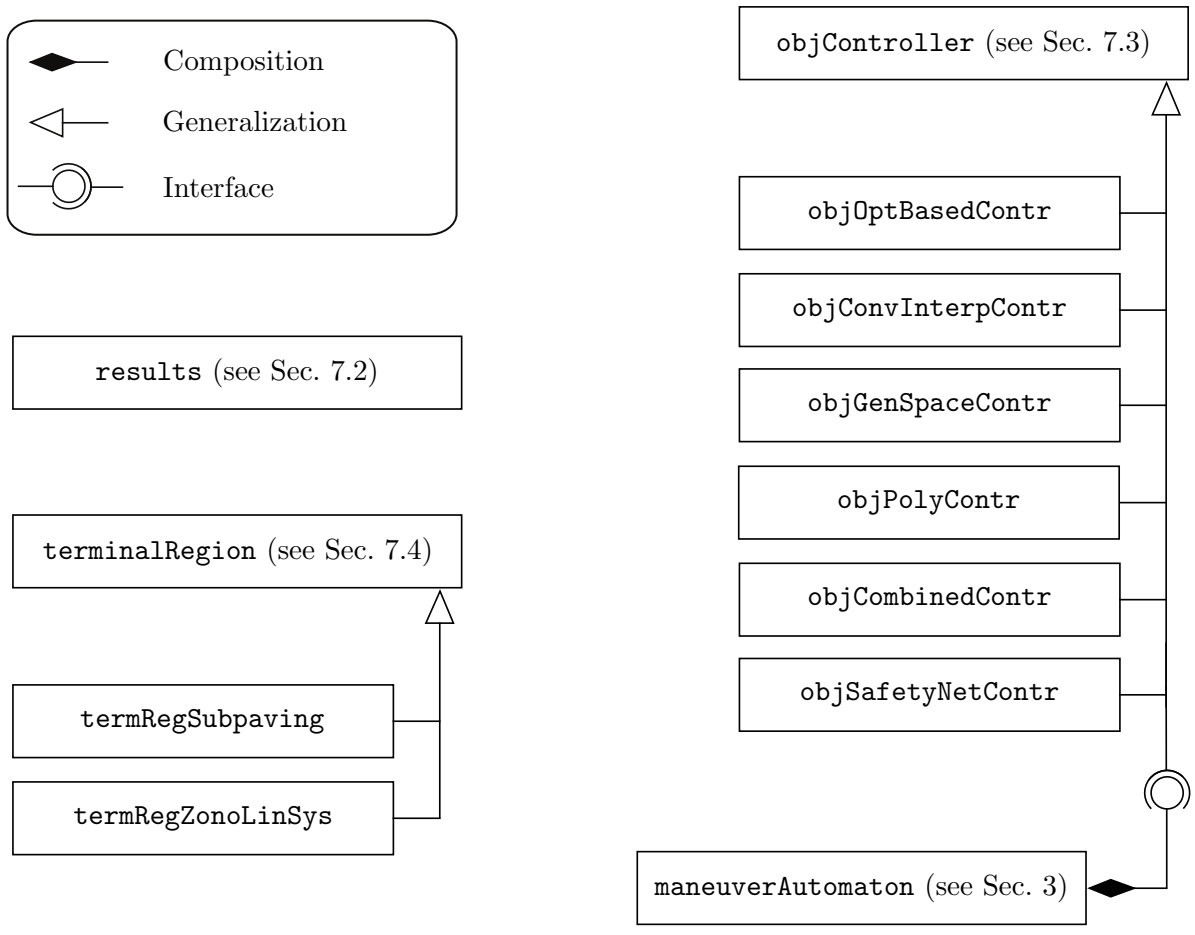


Figure 3: Unified Modeling Language (UML) class diagram for AROC.

1.5 Code Documentation

In addition to the documentation provided in this manual AROC has HTML code documentation that can be viewed and browsed directly in MATLAB. This code documentation contains a short description as well as a list of input and output arguments for each function contained in the AROC toolbox. To view the HTML code documentation type the command

```
>> doc
```

into the MATLAB command line, which will open a window containing the MATLAB documentation. The documentation for the AROC toolbox can be found under the menu item *Supplemental Software*.

For developers: The HTML code documentation is automatically generated from the function headers. To generate the documentation type the command

```
>> pubishHelp
```

into the MATLAB command line. To generate the HTML documentation for a single MATLAB function type

```
>> publishFunc('fileName')
```

which opens a new window showing the generated documentation for the file.

1.6 Unit Tests

In order to guarantee that AROC functions correctly and that there are no bugs in our implementation we integrated several unit-tests into the toolbox. These tests check for example if the input and state constraints are satisfied, or that the reachable set contains all trajectories of the controlled system. In order to execute all unit-tests type the command

```
>> runUnitTests
```

into the MATLAB command line. To execute a single unit test, simply type the name of the test file. All unit-test files are located in the directory */unitTests/...* in the AROC toolbox.

It is advisable to run the unit-tests after installation to check if everything is set-up correctly. Developers should run the unit test every time they changed something on the implementation of the algorithms.

1.7 Connections to CommonRoad and CommonOcean

The CommonRoad framework [2] provides multiple thousands of different traffic scenarios as benchmarks for testing motion planning algorithms for autonomous cars. Similarly, CommonOcean [3] provides marine traffic scenarios for autonomous ships. AROC provides interfaces to easily load CommonRoad and CommonOcean benchmarks for testing the control algorithms. In order to load a CommonRoad or CommonOcean benchmark into AROC, the following two steps are required:

1. Download the CommonRoad or CommonOcean file for the selected traffic scenario from the corresponding website `https://commonroad.in.tum.de` or `https://commonocean.cps.cit.tum.de`
2. Use the function `commonroad2cora` or `commonocean2cora` provided by the CORA toolbox [1] to load initial state, goal set, as well as static and dynamic obstacles for the planning problem.

The syntax for loading a CommonRoad or CommonOcean file with the function `commonroad2cora` or `commonocean2cora` is as follows:

```
[statObs, dynObs, x0, goalSet, lanelets] = commonroad2cora(filename)
[statObs, dynObs, x0, goalSet, waters, shallows] = commonocean2cora(filename),
```

where `filename` is a string with the file name of the CommonRoad or CommonOcean file that should be loaded, and the output arguments are defined as:

- `statObs` MATLAB cell-array storing the static obstacles for the planning problem as objects of class `polygon` (see [4]).
- `dynObs` MATLAB cell-array storing the dynamic obstacles for the planning problem as objects of class `polygon` (see [4]). In addition, the corresponding time interval for each obstacle is stored.
- `x0` struct with fields `.x`, `.y`, `.time`, `.velocity` and `.orientation` storing the initial state for the planning problem.
- `goalSet` struct with fields `.set`, `.time`, `.velocity` and `.orientation` storing the goal set for the planning problem.
- `lanelets` MATLAB cell-array storing the lanelets for the traffic scenario as objects of class `polygon` (see [4]).
- `waters` MATLAB cell-array storing the water ways for the marine traffic scenario as objects of class `polygon` (see [4]).
- `shallows` MATLAB cell-array storing the shallows for the marine traffic scenario as objects of class `polygon` (see [4]).

Initial state, goal set, static obstacles, and dynamic obstacles can then be used for online planning with a maneuver automaton as described in Sec. 3.5. In Fig. 4 an exemplary CommonRoad traffic scenario is visualized. A code example that demonstrates how a CommonRoad benchmark can be solved with AROC is provided in Sec. 8.3 and in the directory */example/maneuverAutomaton/...* in the AROC toolbox.

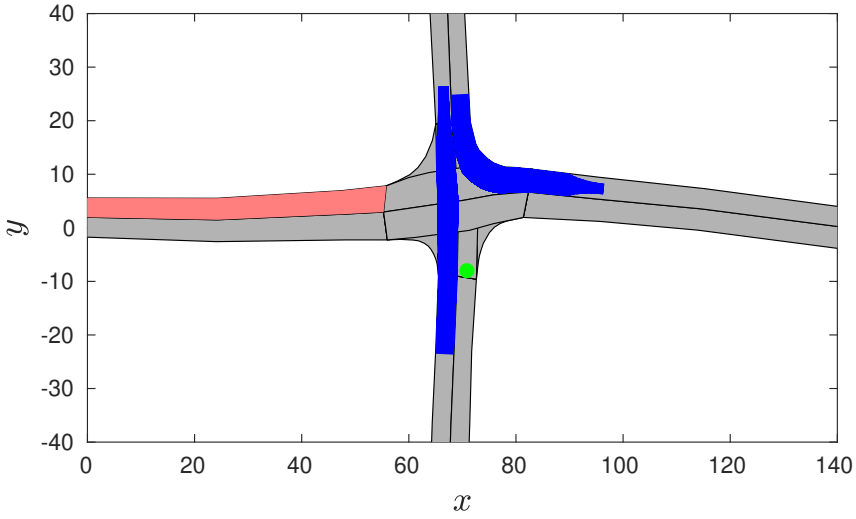


Figure 4: Visualization of the CommonRoad benchmark *DEU_Ffb-1_2_S-1*. The dynamic obstacles imposed by the other cars are shown in blue, the goal set is shown in red, and the initial state for the ego vehicle is shown in green.

2 Control Algorithms

AROC is a toolbox that automatically synthesizes verified controllers for solving reach-avoid problems. We consider general nonlinear disturbed systems with measurement uncertainty defined by the differential equation

$$\dot{x}(t) = f(x(t), u(t), w(t)), \quad x(0) \in \mathcal{R}_0,\ x(t) \in \mathcal{X},\ u(t) \in \mathcal{U},\ w(t) \in \mathcal{W}, \tag{1}$$

where $x(t) \in \mathbb{R}^n$ is the vector of system states, $u(t) \in \mathbb{R}^m$ is the vector of control inputs, $w(t) \in \mathbb{R}^q$ is the vector of disturbances and $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^q \to \mathbb{R}^n$ is a Lipschitz continuous function. Furthermore, we consider a set of initial states $\mathcal{R}_0 \subset \mathbb{R}^n$, a set of state constraints $\mathcal{X} \subset \mathbb{R}^n$, a set of input constraints $\mathcal{U} \subset \mathbb{R}^m$, and a set of disturbances $\mathcal{W} \subset \mathbb{R}^q$. The measured system state $\widehat{x}(t) \in \mathbb{R}^n$ is subject to an uncertain measurement error $v(t) \in \mathbb{R}^n$:

$$\widehat{x}(t) = x(t) + v(t), \quad v(t) \in \mathcal{V}, \tag{2}$$

where $\mathcal{V} \subset \mathbb{R}^n$ is the set of measurement errors. Given a control law $u_c(\widehat{x}(t), t)$, the dynamic of the controlled system is

$$\dot{x}(t) = f(x(t), u_c(\widehat{x}(t), t), w(t)). \tag{3}$$

Let us denote the solution to (3) at time t by $\xi(t, x(0), u_c(\cdot), w(\cdot), v(\cdot))$. The reachable set of the controlled system in (3) is defined as

$$\mathcal{R}_{u_c(\cdot)}(t) = \Big\{ \xi\big(t, x(0), u_c(\cdot), w(\cdot), v(\cdot)\big) \;\Big|\; x(0) \in \mathcal{R}_0,\ \forall \tau \in [0, t] :\ w(\tau) \in \mathcal{W} \wedge v(\tau) \in \mathcal{V} \Big\}. \tag{4}$$

AROC automatically synthesizes a suitable control law $u_c(\widehat{x}(t), t)$ such that input and state constraints are satisfied:

$$\begin{aligned} &\forall t \in [0, t_f],\ \forall \widehat{x}(t) \in \mathcal{R}_{u_c(\cdot)}(t) \oplus \mathcal{V} :\ u_c(\widehat{x}(t), t) \in \mathcal{U} \\ &\forall t \in [0, t_f],\ \forall x(0) \in \mathcal{R}_0,\ \forall w(\cdot) \in \mathcal{W},\ \forall v(\cdot) \in \mathcal{V} :\ \xi\big(t, x(0), u_c(\cdot), w(\cdot), v(\cdot)\big) \in \mathcal{X}, \end{aligned} \tag{5}$$

where t_f is the final time of the control action. The objective that the controller aims to fulfill depends on the controller type: The motion primitive based control algorithms in Sec. 2.1 aim to drive all states from the initial set at the final time t_f as close as possible to a desired final state $x_f \in \mathbb{R}^n$. The model predictive control algorithm in Sec. 2.2 on the other hand tries to stabilize the system around a desired equilibrium point x_f for an infinite time horizon $t_f = \infty$. To achieve this, the model predictive control algorithm considers a terminal region $\mathcal{T}$ around x_f, and the goal is to reach this terminal region in finite time.

Many of the control algorithms implemented in AROC require to solve optimal control problems. An optimal control problem finds the control input that minimizes a certain cost function [5]. In this toolbox we consider optimal control problems defined as

$$\begin{aligned} &\min_{u(t)} \quad \big(x(t_f) - x_f\big)^T \cdot Q \cdot \big(x(t_f) - x_f\big) + \int_{t=0}^{t_f} u(t)^T \cdot R \cdot u(t)\ dt \\ &s.t.\ \ \dot{x}(t) = f(x(t), u(t), \mathbf{0}), \end{aligned} \tag{6}$$

where the input $u(t)$ is piecewise constant, $Q \in \mathbb{R}^{n \times n}$ is the state weighting matrix, and $R \in \mathbb{R}^{m \times m}$ is the input weighting matrix.

Next, we describe the different control algorithms implemented in AROC in detail.

2.1 Motion Primitive Based Control

The motion primitive based control algorithms described in this chapter automatically synthesize feasible and close-to-optimal controllers for single motion primitives offline. These motion primitives can be used to construct a maneuver automaton (see Sec. 3), which is then applied for online control (see Sec. 3.5).

For each motion primitive the goal of the control action is to drive all states inside the initial set at the final time t_f as close as possible to the desired final state x_f:

$$\min_{u_c(x,t)} \quad \rho\big(\mathcal{R}_{u_c(\cdot)}(t_f), x_f\big), \tag{7}$$

where $\mathcal{R}_{u_c(\cdot)}(t_f)$ is the reachable set of the controlled system at the final time t_f (see (4)), and $\rho(\mathcal{R}_{u_c(\cdot)}(t_f), x_f) \to \mathbb{R}_0^+$ is a cost function measuring the distance between the states in $\mathcal{R}_{u_c(\cdot)}(t_f)$ and the desired final state x_f. There exist many different possibilities for suitable cost functions, like for example the maximum euclidean distance:

$$\rho\big(\mathcal{R}_{u_c(\cdot)}(t_f), x_f\big) = \max_{x \in \mathcal{R}_{u_c(\cdot)}(t_f)} \; ||x - x_f||_2.$$

The syntax for executing the control algorithm to synthesize a suitable controller is identical for all motion primitive based control algorithms:

$$[\texttt{obj}, \texttt{res}] = \texttt{controlAlgorithmName}(\texttt{benchmark}, \texttt{Param})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{controlAlgorithmName}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{controlAlgorithmName}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts}, \texttt{Post}),$$

where `controlAlgorithmName` $\in$ {`optimizationBasedControl`, `convexInterpolationControl`, `generatorSpaceControl`} is the name of the control algorithm, the input arguments are defined as

- `benchmark` name of the benchmark system that is considered (see Sec. 6).
- `Param` struct containing the parameter that define the control problem
  - `.R0` initial set $\mathcal{R}_0$ (see (1)) represented as an object of class `interval` (see [4, Sec. 2.2.1.2]).
  - `.U` set of input constraints $\mathcal{U}$ (see (1)) represented as an object of class `interval` (see [4, Sec. 2.2.1.2]).
  - `.W` set of disturbances $\mathcal{W}$ (see (1)) represented as an object of class `interval` or `zonotope` (see [4, Sec. 2.2.1]).
  - `.V` set of measurement errors $\mathcal{V}$ (see (2)) represented as an object of class `interval` or `zonotope` (see [4, Sec. 2.2.1]).
  - `.X` set of state constraints $\mathcal{X}$ (see (1)) represented as an object of class `mptPolytope` (see [4, Sec. 2.2.1.4]).
  - `.tFinal` final time t_f (see (7)).
  - `.xf` desired final state x_f (see (7)).
- `Opts` struct containing the settings for the control algorithm. Since the settings are different for each control algorithm they are documented in Sec. 2.1.1, 2.1.2, and 2.1.3.
- `Post` MATLAB function handle to the post-processing function that computes the occupancy set from the reachable set (see Sec. 3.2). This argument is only required if the motion primitive controller is used to construct a maneuver automaton (see Sec. 3).

and the output arguments are defined as

- `obj` object of class `contrObj` (see Sec. 7.3) that stores the synthesized control law.
- `res` object of class `result` (see Sec. 7.2) that stores the computed reachable set of the controlled system.

In the following sections we describe the motion primitive based control algorithms that are implemented in AROC in detail.

2.1.1 Optimization Based Control

Optimization-based control implements the control algorithm described in [6]. While the work in [6] specialized on linear systems, we extended the approach to also handle systems with nonlinear dynamics. However, since reachability analysis for linear systems is computationally much more efficient than reachability analysis for nonlinear systems, our implementation of the algorithm detects automatically if the system is linear or nonlinear and then executes the corresponding reachability algorithm.

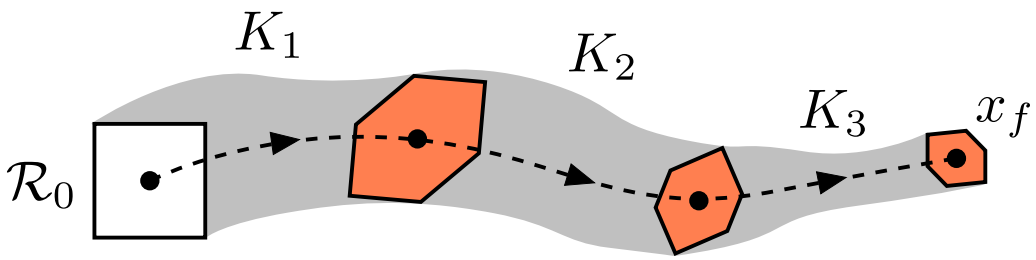


Figure 5: Illustration of the optimization based control algorithm with $N = 3$ constant segments.

The control algorithm uses the following control law:

$$u_c(\widehat{x}, t) = u_{ref}(t) + K(t)(\widehat{x}(t) - x_{ref}(t)),$$

where $u_{ref}(t) \in \mathbb{R}^m$ is the piecewise constant control input for the reference trajectory (see Sec. 7.5), $x_{ref}(t) \in \mathbb{R}^n$ is the state of the reference trajectory (see Sec. 7.5), and $K(t) \in \mathbb{R}^{m \times n}$ is a time-varying feedback matrix. The optimization based control algorithm determines a feasible and close-to-optimal value for the time-varying feedback matrix $K(t)$ by solving the following optimization problem:

$$\begin{aligned} &\min_{K(t)} \ \rho\big(\mathcal{R}_{u_c(\cdot)}(t_f), x_f\big) \\ &s.t. \ \ \forall t \in [0, t_f], \ \forall \widehat{x}(t) \in \mathcal{R}_{u_c(\cdot)}(t) \oplus \mathcal{V} : \ u_{ref}(t) + K(t)(\widehat{x}(t) - x_{ref}(t)) \in \mathcal{U} \\ &\quad \forall t \in [0, t_f], \ \forall x(0) \in \mathcal{R}_0, \ \forall w(\cdot) \in \mathcal{W}, \ \forall v(\cdot) \in \mathcal{V} : \ \xi\big(t, x(0), u_c(\cdot), w(\cdot), v(\cdot)\big) \in \mathcal{X}, \end{aligned} \tag{8}$$

where $\rho(\cdot)$ is the cost function (see (7)). In order to express the optimization problem with a finite number of optimization variables, a piecewise constant time-varying feedback matrix $K(t)$ is used: $\forall t \in [(i-1)\Delta t, i\Delta t] : \ K(t) = K_i, \ i \in \{1, \ldots, N\}$, where $\Delta t = t_f/N$ and $N \in \mathbb{N}_{\geq 1}$ is the number of piecewise constant segments (see Fig. 5). Furthermore, in order to reduce the number of variables for the optimization problem, we use a *Linear Quadratic Regulator* (LQR) approach [7, Chapter 3.3] to compute the feedback matrices K_i. Instead of directly optimizing the feedback matrices K_i we then optimize the weighting matrices $Q \in \mathbb{R}^{n \times n}$, $R \in \mathbb{R}^{m \times m}$ from

the cost function of the *Linear Quadratic Regulator*, where we choose the weighting matrices to be diagonal. For solving the optimization problem (8), we use MATLABs *fmincon* algorithm[1].

The syntax for executing the optimization based control algorithm is as follows:

$$[\texttt{obj}, \texttt{res}] = \texttt{optimizationBasedControl}(\texttt{benchmark}, \texttt{Param})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{optimizationBasedControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{optimizationBasedControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts}, \texttt{Post}),$$

where `benchmark`, `Param`, `Post`, `obj`, and `res` are defined as at the beginning of Sec. 2.1, and `Opts` is a struct that contains the following algorithm settings:

- `.N` number of piecewise constant segments N for the time-varying feedback matrix $K(t)$. The default value is 5.
- `.reachSteps` number of time steps for reachability analysis during one of the N piecewise constant segments. The default value is 10.
- `.reachStepsFin` number of time steps for reachability analysis during one of the N piecewise constant segments for the computation of the final reachable set after the optimization finished. To accelerate the optimization it is advisable to use less reachability time steps during optimization than for the computation of the final reachable set. The default value is 15.
- `.maxIter` maximum number of iterations for MATLABs *fmincon* algorithm that is used to solve the optimization problem (8) (see `https://de.mathworks.com/help/optim/ug/fmincon.html`). The default value is 100.
- `.bound` scaling factor δ between the upper and the lower bound for the entries of the LQR weighting matrices Q and R. It holds for all matrix entries $Q_{i,j}$ and $R_{i,j}$ that $Q_{i,j} \in [1/\delta, \delta]$ and $R_{i,j} \in [1/\delta, \delta]$. The default value is 1000.
- `.refTraj` struct containing the settings for the reference trajectory (see Sec. 7.5).
- `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for the optimization based control algorithm are provided in Sec. 8.1 and in the directory */example/optimizationBasedControl/...* in the AROC toolbox.

2.1.2 Convex Interpolation Control

The convex interpolation control algorithm implements the approach in [8]. For convex interpolation control the time horizon is divided into N time steps, where in each time step the following procedure is applied (see Fig. 6):

1. The reachable set at the beginning of the time step is enclosed by a parallelotope.
2. Optimal control problems (see (6)) are solved for all vertices of the parallelotope.
3. The control law is obtained by interpolation between the optimal control inputs for the parallelotope vertices (see [8, Sec. 4]).

[1] `https://de.mathworks.com/help/optim/ug/fmincon.html`

Since the interpolation control law in [8, Sec. 4] is quite complex, it is often advisable to use a linear or a quadratic approximation instead (see [8, Sec. 5]). While the optimization based controller in Sec. 2.1.1 considers continuous feedback, the convex interpolation control algorithm only measures the system state at the beginning of each time step, which results in discrete-time feedback. Each time step consists of N_{inter} intermediate time steps, which correspond to the piecewise constant segments of the control input for the optimal control problems.

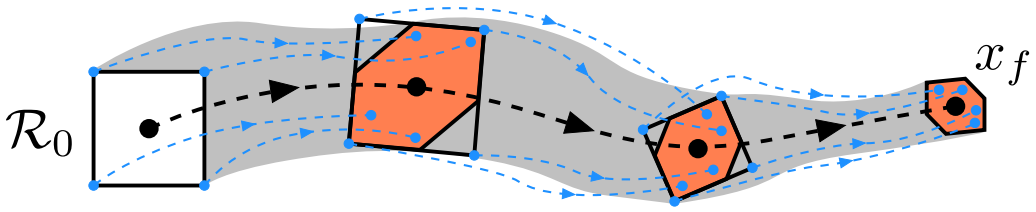


Figure 6: Illustration of the convex interpolation control algorithm with $N = 3$ time steps.

The syntax for executing the convex interpolation control algorithm is as follows:

$$[\texttt{obj}, \texttt{res}] = \texttt{convexInterpolationControl}(\texttt{benchmark}, \texttt{Param})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{convexInterpolationControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{convexInterpolationControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts}, \texttt{Post}),$$

where `benchmark`, `Param`, `Post`, `obj`, and `res` are defined as at the beginning of Sec. 2.1, and `Opts` is a struct that contains the following algorithm settings:

- `.controller` string specifying the control law that is used. The available control laws are `'exact'` (interpolation control law, see [8, Sec. 4]), `'quadratic'` (quadratic approximation), and `'linear'` (linear approximation, see [8, Sec. 5]). The default value is `'linear'`.
- `.N` number of time steps N. The default value is 10.
- `.Ninter` number of intermediate time steps N_{inter}. The default value is 4.
- `.reachSteps` number of time steps for reachability analysis during one of the N_{inter} intermediate time steps. The default value is 20.
- `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for the optimal control problems (see (6)). The default value is the identity matrix.
- `.R` input weighting matrix $R \in \mathbb{R}^{m \times m}$ for the optimal control problems (see (6)). The default value is an all-zero matrix.
- `.parallel` flag specifying if parallel computing is used (`Opts.parallel` $= 1$) or not (`Opts.parallel` $= 0$). The default value is 0.
- `.approx` struct containing the settings for the approximation of the interpolation control law (for `Opts.controller = 'linear'` and `Opts.controller = 'quadratic'` only).

  - `.method` string specifying the method that is used to obtain the approximated control law. The available methods are `'scaled'`, `'optimized'`, and `'center'`. The default value is `'scaled'`.
  - `.lambda` parameter $\lambda \in [0, 1]$ representing the tradeoff between matching the optimal control inputs at the vertices ($\lambda = 0$) and matching the interpolation control law ($\lambda = 1$). The default value is 0.5.
- `.polyZono` struct containing the settings for restructuring polynomial zonotopes (for `Opts.cora.alg = 'poly'` only).
  - `.N` number of time steps after which the polynomial zonotope representing the reachable set is restructured. The default value is ∞ (no restructuring).
  - `.orderDep` zonotope order of the dependent part of the polynomial zonotope after restructuring. The default value is 10.
  - `.order` overall zonotope order of the polynomial zonotope after restructuring. The default value is 20.
- `.refTraj` struct containing the settings for the reference trajectory (see Sec. 7.5).
- `.extHorizon` struct containing the settings for an extended optimization horizon (see Sec. 7.6).
- `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for the convex interpolation control algorithm are provided in the directory */example/convexInterpolationControl/...* in the AROC toolbox.

2.1.3 Generator Space Control

One of the main disadvantages of the convex interpolation controller in Sec. 2.1.2 is that the computational complexity grows exponentially with the number of system dimensions n. The reason for this is that for convex interpolation control an optimal control problem is solved for each vertex of a parallelotope enclosure of the reachable set, and a parallelotope has 2^n vertices. The generator space controller proposed in [9] circumvents this problem by solving one optimal control problem for each generator of the parallelotope, instead of for each vertex (see Fig. 7). Since a parallelotope has only n generators, this is computationally much more efficient.

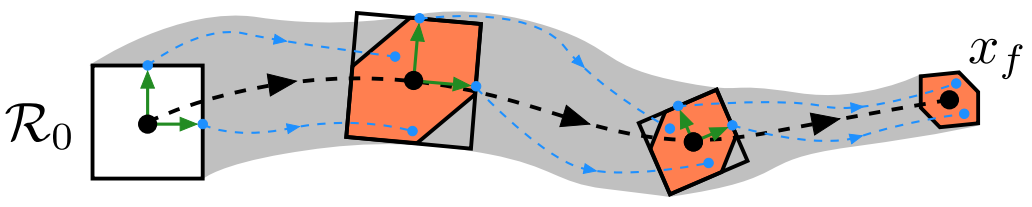


Figure 7: Illustration of the generator space control algorithm with $N = 3$ time steps.

As for convex interpolation control, the time horizon is divided into N time steps, and a feed-

forward controller is computed for each of these time steps. Furthermore, each time step consists of N_{inter} intermediate time steps, which correspond to the piecewise constant segments of the control input for the optimal control problems. To obtain the control law from the optimal control inputs for the generators, the generator space controller expresses each state inside the reachable set as a linear combination of the generators.

The syntax for executing the generator space control algorithm is as follows:

$$[\texttt{obj}, \texttt{res}] = \texttt{generatorSpaceControl}(\texttt{benchmark}, \texttt{Param})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{generatorSpaceControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{generatorSpaceControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts}, \texttt{Post}),$$

where `benchmark`, `Param`, `Post`, `obj`, and `res` are defined as at the beginning of Sec. 2.1, and `Opts` is a struct that contains the following algorithm settings:

- `.N` number of time steps N. The default value is 10.
- `.Ninter` number of intermediate time steps N_{inter}. The default value is 4.
- `.reachSteps` number of time steps for reachability analysis during one of the N_{inter} intermediate time steps. The default value is 10.
- `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for the optimal control problems (see (6)). The default value is the identity matrix.
- `.R` input weighting matrix $R \in \mathbb{R}^{m \times m}$ for the optimal control problems (see (6)). The default value is an all-zero matrix.
- `.refInput` flag specifying if the control input from the reference trajectory is used to control the center of the reachable set (`Opts.refInput` = 1) or not (`Opts.refInput` = 0). The default value is 0.
- `.refTraj` struct containing the settings for the reference trajectory (see Sec. 7.5).
- `.extHorizon` struct containing the settings for an extended optimization horizon (see Sec. 7.6).
- `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for the generator space control algorithm are provided in the directory */example/generatorSpaceControl/...* in the AROC toolbox.

2.1.4 Polynomial Control

The polynomial control approach proposed in [10] is similar to the generator space controller in Sec. 2.1.3, but applies a polynomial instead of a linear control law. Moreover, rather than solving optimal control problems the polynomial control approach utilizes dependency preservation [11] for polynomial zonotopes [12] to determine the optimal control law parameters with the following procedure (see Fig. 8):

1. Compute the reachable set for the set of all parameter values that satisfy the input constraints.
2. Utilize the analytical relation between parameter values and reachable states given by dependency preservation to determine the control law parameters that minimize the size of the final reachable set.

Equivalently to the generator space controller, the polynomial controller divides the time horizon into N time steps consisting of N_{inter} segments with constant control inputs, and encloses the reachable set by a parallelotope at the end of each time step. To refine the estimate for the set of control law parameters that satisfy the input constraints, the reachable set can be splitted multiple times.

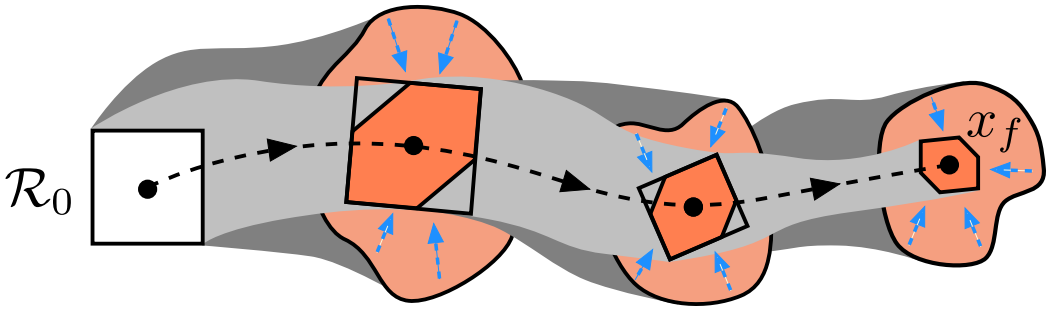


Figure 8: Illustration of the polynomial control algorithm with $N = 3$ time steps.

The syntax for executing the polynomial control algorithm is as follows:

$$[\texttt{obj}, \texttt{res}] = \texttt{polynomialControl}(\texttt{benchmark}, \texttt{Param})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{polynomialControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts})$$
$$[\texttt{obj}, \texttt{res}] = \texttt{polynomialControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts}, \texttt{Post}),$$

where `benchmark`, `Param`, `Post`, `obj`, and `res` are defined as at the beginning of Sec. 2.1, and `Opts` is a struct that contains the following algorithm settings:

- `.N` number of time steps N. The default value is 10.
- `.Ninter` number of intermediate time steps N_{inter}. The default value is 4.
- `.ctrlOrder` polynomial degree of the polynomial control law. The default value is 2, which corresponds to a quadratic controller.
- `.reachSteps` number of time steps for reachability analysis during one of the N_{inter} intermediate time steps used for computing the reachable set for the set of all parameter values. The default value is 10.
- `.reachStepsFin` number of time steps for reachability analysis during one of the N_{inter} intermediate time steps used for computing the final reachable set after control law parameter optimization. The default value is 20.
- `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for the cost function that is minimized in order to determine the optimal control law parameters. The default value is the identity matrix.
- `.splits` number of splits applied to refine the set of parameters that satisfies the input constraints. The default value is 0.
- `.refInput` flag specifying if the control input from the reference trajectory is used to control the center of the reachable set (`Opts.refInput` = 1) or not (`Opts.refInput` = 0). The default value is 0.
- `.refUpdate` flag specifying if the reference trajectory is updated after each time step (`Opts.refUpdate` = 1) or not (`Opts.refUpdate` = 0). The default value is 0.

- `.refTraj` struct containing the settings for the reference trajectory (see Sec. 7.5).
- `.extHorizon` struct containing the settings for an extended optimization horizon (see Sec. 7.6).
- `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for the polynomial control algorithm are provided in the directory */example/polynomialControl/...* in the AROC toolbox.

2.1.5 Combined Control

Combined control implements the control algorithm described in [13], which combines continuous feedback based on the optimization based control algorithm in Sec. 2.1.1 with feed-forward control based on the generator space control algorithm in Sec. 2.1.3. Due to the continuous feedback, the optimization based controller is very robust against disturbances. However, since the control law tracks the reference trajectory and the distance between reachable states and reference trajectory can be quite large, the feedback matrices that are automatically determined by the algorithm are usually quite conservative to guarantee that the input constraints are satisfied. In order to avoid the disadvantage and use more aggressive feedback matrices for faster disturbance rejection the combined control algorithm uses a different reference trajectory for each state inside the initial set. In summary, the control algorithm consists of the following two steps:

1. **Feed-forward control:** Compute an initial-state-dependent reference trajectory using the generator space control algorithm.
2. **Continuous feedback:** Use optimization to compute a suitable feedback matrix for tracking the initial-state-dependent reference trajectory.

An illustration of the combined control algorithm is shown in Fig. 9.

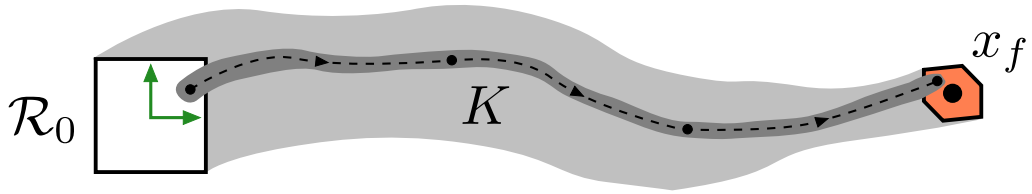


Figure 9: Illustration of the combined control algorithm.

The combined control algorithm uses the following control law:

$$u_c(\widehat{x}, x(0), t) = u_{ff}(x(0), t) + K(\widehat{x}(t) - x_{ff}(x(0), t)), \tag{9}$$

where $u_{ff}(x(0), t) \in \mathbb{R}^m$ and $x_{ff}(x(0), t) \in \mathbb{R}^n$ represent the control inputs and the states of the initial-state-dependent reference trajectory computed with the generator space control algorithm in Sec. 2.1.3. The control input $u_{ff}(x(0), t)$ is piecewise constant, where the number of pieces is $N \in \mathbb{N}_{\geq 0}$. To leave some control input for the feedback controller, the feed-forward control law $u_{ff}(x(0), t)$ is computed using a set of tightend input constraints $\overline{\mathcal{U}} \subset \mathcal{U}$. The feedback matrix $K \in \mathbb{R}^{m \times n}$ for the control law in (9) is determined by solving the following optimization

problem:

$$\min_{K} \underbrace{\max_{x \in \mathcal{R}_{u_c(\cdot)}(t_f)} ||Q \cdot (x - x_f)||_1 + \int_0^{t_f} \max_{x \in \mathcal{R}_{u_c(\cdot)}(t)} ||R \cdot u_c(x, x(0), t)||_1 \, dt}_{\rho\left(\mathcal{R}_{u_c(\cdot)}(t_f), x_f\right)}. \tag{10}$$

$$\begin{aligned} s.t.\ \ & \forall t \in [0, t_f],\ \forall \widehat{x}(t) \in \mathcal{R}_{u_c(\cdot)}(t) \oplus \mathcal{V}:\ u_{ff}(x(0), t) + K(\widehat{x}(t) - x_{ff}(x(0), t) \in \mathcal{U} \\ & \forall t \in [0, t_f],\ \forall x(0) \in \mathcal{R}_0,\ \forall w(\cdot) \in \mathcal{W},\ \forall v(\cdot) \in \mathcal{V}:\ \xi\big(t, x(0), u_c(\cdot), w(\cdot), v(\cdot)\big) \in \mathcal{X}. \end{aligned}$$

In order to reduce the number of variables for the optimization problem, we use a *Linear Quadratic Regulator* (LQR) approach [7, Chapter 3.3] to compute the feedback matrix K. Instead of directly optimizing the feedback matrix K we then optimize the weighting matrices $Q \in \mathbb{R}^{n \times n}$, $R \in \mathbb{R}^{m \times m}$ from the cost function of the *Linear Quadratic Regulator*, where we choose the weighting matrices to be diagonal. For solving the optimization problem (10), we use MATLABs *fmincon* algorithm[2]. Instead of the generator space controller, it is also possible to use the polynomial controller from Sec. 2.1.4 as the feed-forward controller.

The syntax for executing the combined control algorithm is as follows:

$$\begin{aligned} &[\texttt{obj}, \texttt{res}] = \texttt{combinedControl}(\texttt{benchmark}, \texttt{Param}) \\ &[\texttt{obj}, \texttt{res}] = \texttt{combinedControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts}) \\ &[\texttt{obj}, \texttt{res}] = \texttt{combinedControl}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts}, \texttt{Post}), \end{aligned}$$

where `benchmark`, `Param`, `Post`, `obj`, and `res` are defined as at the beginning of Sec. 2.1, and `Opts` is a struct that contains the following algorithm settings:

- `.N` — number of piecewise constant segments N for feed-forward control inputs $u_{ff}(x(0), t)$. The default value is 5.
- `.feedForward` — feed-forward controller used, which can either be the generator space controller from Sec. 2.1.3 (`Opts.feedForward = 'genSpace'`) or the polynomial controller from Sec. 2.1.4 (`Opts.feedForward = 'poly'`). The default is the generator space controller.
- `.reachSteps` — number of time steps for reachability analysis during one of the N piecewise constant segments. The default value is 10.
- `.reachStepsFin` — number of time steps for reachability analysis during one of the N piecewise constant segments for the computation of the final reachable set after the optimization finished. To accelerate the optimization it is advisable to use less reachability time steps during optimization than for the computation of the final reachable set. The default value is 50.
- `.scale:` — scaling factor used to determine set of tightened input constraints $\overline{\mathcal{U}} \subset \mathcal{U}$. The default value is 0.9.
- `.Q` — state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for the cost function $\rho\left(\mathcal{R}_{u_c(\cdot)}(t_f), x_f\right)$ in (10). The default value is the identity matrix.
- `.R` — input weighting matrix $R \in \mathbb{R}^{m \times m}$ for the cost function $\rho\left(\mathcal{R}_{u_c(\cdot)}(t_f), x_f\right)$ in (10). The default value is an all-zero matrix.

[2] `https://de.mathworks.com/help/optim/ug/fmincon.html`

- `.Qff` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for the feed-forward controller (see Sec. 2.1.3). The default value is the identity matrix.
- `.Rff` input weighting matrix $R \in \mathbb{R}^{m \times m}$ for the feed-forward controller (see Sec. 2.1.3). The default value is an all-zero matrix.
- `.finStateCon` flag specifying whether the additional constraint that the final reachable set has to be located inside the shifted initial set is applied (`Opts.finStateCon = true`) or not (`Opts.finStateCon = false`). The default value is `false`.
- `.maxIter` maximum number of iterations for MATLABs *fmincon* algorithm that is used to solve the optimization problem (8) (see `https://de.mathworks.com/help/optim/ug/fmincon.html`). The default value is 5.
- `.bound` scaling factor δ between the upper and the lower bound for the entries of the LQR weighting matrices Q and R used to calculate the feedback matrix K. It holds for all matrix entries $Q_{i,j}$ and $R_{i,j}$ that $Q_{i,j} \in [1/\delta, \delta]$ and $R_{i,j} \in [1/\delta, \delta]$. The default value is 1000.
- `.refTraj` struct containing the settings for the reference trajectory (see Sec. 7.5).
- `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for the combined control algorithm are provided in the directory */example/combinedControl/...* in the AROC toolbox.

2.1.6 Safety Net Control

While all previous control algorithms introduced in this section are able to guarantee safety, they are usually not optimal with regard to other criteria such as comfort, energy consumption, or low wear. To solve this issue, safety net control implements the control algorithm in [14], where a safety controller is used as a safety net for a comfort controller that shows good performance with regard to the afore mentioned criteria. To guarantee safety for the overall control approach, we perform the following procedure for each time step during online execution of the controller (see Fig. 10):

1. We compute the reachable set of the comfort controller to check if the comfort controller satisfies input and state constraints.
2. If the comfort controller is safe, we apply the comfort controller for the current time step.
3. If the comfort controller is not safe or the final reachable set is outside the reachable set of the safety controller, we apply the safety net controller for the current time step.

Since our goal is to minimize the number of cases where the safety controller has to take over, we want to design a safety net controller that maximizes the size of the reachable set to give the comfort controller some space. To achieve this, the safety net controller is computed based on backward reachable sets, where the shifted initial set $\mathcal{R}_0 - \texttt{center}(\mathcal{R}_0) + x_f$ is used as a goal set that should be reached at the end of the time horizon t_f. The control law is then computed by optimization with the goal to maximize the size of the set that can be controlled into the goal set. Moreover, the time horizon is divided into N time steps, where in each time step either the comfort controller or the safety controller is active. Each time step is again divided into N_{inter}

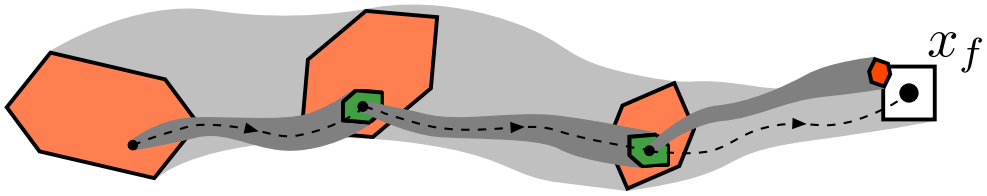


Figure 10: Illustration of the safety net control algorithm with $N = 3$ time steps, where the comfort controller is applied during the first two time steps, and the safety net controller takes over during the last time step.

intermediate time steps, which correspond to the number of input changes for the safety control law.

One problem that we are facing during online application of the controller is that the computation of the reachable set for the comfort controller requires a certain computation time t_{comp}. During this time, however, the system will evolve further so that the state of the system after the computation finished will be different from the state that we measured before the computation, which would make the computed reachable set invalid. To solve this issue, the safety net control algorithm first predicts with reachability analysis where the system will be after the computation finished, and then computes the reachable set starting from the set of predicted states.

Currently, a *Linear Quadratic Regulator* (LQR) and a *Model Predictive Controller* (MPC) are implemented as comfort controllers in AROC. Custom comfort controllers can be conveniently added as described in Sec. 7.8.

The syntax for executing the generator space control algorithm is as follows:

$$[\texttt{obj},\texttt{res}] = \texttt{safetyNetControl}(\texttt{benchmark},\texttt{Param})$$
$$[\texttt{obj},\texttt{res}] = \texttt{safetyNetControl}(\texttt{benchmark},\texttt{Param},\texttt{Opts})$$
$$[\texttt{obj},\texttt{res}] = \texttt{safetyNetControl}(\texttt{benchmark},\texttt{Param},\texttt{Opts},\texttt{Post}),$$

where `benchmark`, `Param`, `Post`, `obj`, and `res` are defined as at the beginning of Sec. 2.1, and `Opts` is a struct that contains the following algorithm settings:

- `.N` number of time steps N. The default value is 10.
- `.Ninter` number of intermediate time steps N_{inter}. The default value is 4.
- `.reachSteps` number of time steps for reachability analysis during one of the N_{inter} intermediate time steps. The default value is 10.
- `.order` maximum zonotope order for backward reachable sets. The default value is 3.
- `.iter` number of iterations for backward reachable set refinement. The default value is 1.
- `.realTime` flag specifying if the algorithm only applies the comfort controller if the computation time is less than the allocated time `Opts.tComp` (`Opts.realTime` = 1), or if this real-time constraint is not considered (`Opts.realTime` = 0). The default value is 0.

- `.tComp` allocated computation time t_{comp}.
- `.controller` string specifying the name of the comfort controller that is applied during online execution (see Sec. 7.8). Currently, the controllers `'LQR'` and `'MPC'` are available. If multiple comfort controller should be applied in parallel, `Opts.controller` is specified as a cell-array.
- `.contrOpts` struct containing the settings for the comfort controller (see Sec. 7.8). If multiple comfort controller should be applied in parallel, `Opts.contrOpts` is specfied as a cell-array.
- `.refTraj` struct containing the settings for the reference trajectory (see Sec. 7.5).
- `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for the safety net control algorithm are provided in the directory */example/safetyNetControl/...* in the AROC toolbox.

2.2 Model Predictive Control

Model predictive control (MPC) aims to stabilize the system for an infinite time horizon $t_f = \infty$. To achieve this, we consider a terminal region $\mathcal{T}$ (see Sec. 4) around the equilibrium point $x_f \in \mathcal{T}$ for which we have a controller that is guaranteed to stabilize the system around x_f for all states inside the terminal region. The goal is then to synthesize a suitable control law $u_c(\widehat{x}(t), t)$ that drives the system from its current state to the terminal region while minimizing the cost function

$$J(x,u) = \big(x(t_{opt}) - x_f\big)^T \cdot Q \cdot \big(x(t_{opt}) - x_f\big) + \int_{t=0}^{t_{opt}} u(t)^T \cdot R \cdot u(t)\ dt, \tag{11}$$

where t_{opt} is the lookahead time. For controller synthesis we then apply the following procedure (see Fig. 11) which is commonly used in MPC:

1. Based on the current measurement of the system state $\widehat{x}(t)$, we compute an optimal control law $u_c(\widehat{x}(t), t)$ that minimizes the cost function $J(x,u)$ and guarantees that the system reaches the terminal region $\mathcal{T}$ after time t_{opt}.
2. We apply the optimal control law $u_c(\widehat{x}(t), t)$ for the time period $t \in [0, t_{opt}/N]$, where N is the number of time steps.
3. We measure the system state and try to compute a control law $u_c(\widehat{x}(t), t)$ with lower costs than the old control law $u_c(\widehat{x}(t), t)$ based on the updated system state.

This procedure is repeated until the system reaches the terminal region. To guarantee that the terminal region is reached in finite time we consider an additional contraction constraint (see [15, Eq. (13)]) when optimizing the control law $u_c(\widehat{x}(t), t)$. One key difference of the MPC algorithms in AROC compared to other MPC approaches is that reachability analysis is used to verify that input and state constraints are satisfied despite disturbances acting on the system.

One problem that we are facing with the procedure described above is that the computation of the new optimal control law $u_c(\widehat{x}(t), t)$ as well as the verification of the control law using reachability analysis require a certain computation time t_{comp}. During this time, however, the system will evolve further so that the state of the system after the computation finished will be different from the state that we measured before the computation. Since we computed and verified the new optimal control law based on the state measured before the start of the computation, our new control law would therefore be invalid. To solve this issue the reachset

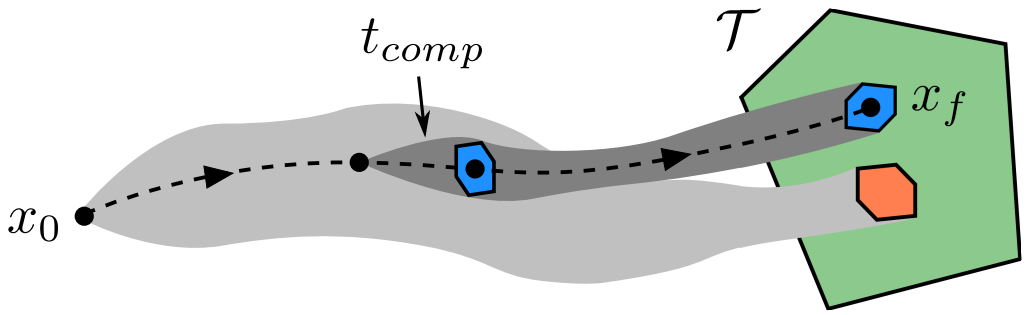


Figure 11: Illustration of model predictive control based on reachable sets.

MPC algorithm first predicts with reachability analysis where the system will be after the computation finished, and then computes a new optimal control law based on the set of predicted states (see Fig. 11).

The syntax for running the control algorithm is identical for all model predictive controllers implemented in AROC:

$$\texttt{res} = \texttt{controlAlgorithmName}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts}),$$

where $\texttt{controlAlgorithmName} \in \{\texttt{reachsetMPC}, \texttt{linSysMPC}\}$ is the name of the MPC control algorithm and the input arguments are defined as

- `benchmark` name of the benchmark system that is considered (see Sec. 6).
- `Param` struct containing the parameter that define the control problem
  - `.x0` initial state $x(0) \in \mathbb{R}^n$.
  - `.xf` desired final state x_f (see (7)).
  - `.U` set of input constraints $\mathcal{U}$ (see (1)) represented as an object of class `interval` (see [4, Sec. 2.2.1.2]).
  - `.W` set of disturbances $\mathcal{W}$ (see (1)) represented as an object of class `interval` or `zonotope` (see [4, Sec. 2.2.1]).
  - `.V` set of measurement errors $\mathcal{V}$ (see (2)) represented as an object of class `interval` or `zonotope` (see [4, Sec. 2.2.1]).
  - `.X` set of state constraints $\mathcal{X}$ (see (1)) represented as an object of class `mptPolytope` (see [4, Sec. 2.2.1.4]).
- `Opts` struct containing the settings for the control algorithm.

and the output arguments are defined as

- `res` object of class `result` (see Sec. 7.2) that stores the computed reachable set of the controlled system.

In the following sections we describe the model predictive control algorithms that are implemented in AROC in detail.

2.2.1 Reachset Model Predictive Control

For model predictive control of nonlinear systems AROC implements the reachset MPC algorithm in [15]. This algorithm applies the tracking controller

$$u_c(\widehat{x}(t), t) = u_{ref}(t) + K(t)(\widehat{x}(t) - x_{ref}(t)), \tag{12}$$

which is synthesized using the following three step procedure:

1. The piecewise constant reference input $u_{ref}(t)$ for the reference trajectory $x_{ref}(t)$ is determined by solving an optimal control problem that minimizes the cost function $J(x, u)$ in (11). The number of piecewise constant segments for $u_{ref}(t)$ is $N \cdot N_{inter}$ and a tightened set of input constraints $\overline{\mathcal{U}} \subseteq \mathcal{U}$ as well as a tightend set of state constraints $\overline{\mathcal{X}} \subseteq \mathcal{X}$ are used so that some control input and space is left for the feedback part of the tracking controller.
2. The feedback matrix $K(t)$ is determined by applying the *Linear Quadratic Regulator* (LQR) approach [7, Chapter 3.3] with weighting matrices Q_{lqr} and R_{lqr} to the linearized system.
3. The reachable set of the controlled system is computed to check if the synthesized controller $u_c(\widehat{x}(t), t)$ is robustly safe despite disturbances and measurement errors.

The syntax for running the reachset model predictive control algorithm is as follows:

$$\texttt{res} = \texttt{reachsetMPC}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts})$$

where `benchmark`, `Param`, and `res` are defined as at the beginning of Sec. 2.2, and `Opts` is a struct that contains the following algorithm settings:

- `.tOpt` lookahead time t_{opt} (see Sec. 2.2).
- `.N` number of time steps N. The default value is 10.
- `.Ninter` number of intermediate time steps N_{inter}. The default value is 1.
- `.reachSteps` number of time steps for reachability analysis during one of the N_{inter} time steps. The default value is 10.
- `.scale:` scaling factor used to determine the tightend input and state constraints $\overline{U}$ and $\overline{X}$ from the sets $\mathcal{U}$ and $\mathcal{X}$. The default value is 0.9.
- `.termReg:` terminal region $\mathcal{T}$ represented as an object of class `mptPolytope` (see [4, Sec. 2.2.1.4]) or class `terminalRegion` (see Sec. 4).
- `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for the cost function $J(x, u)$ in (11). The default value is the identity matrix.
- `.R` input weighting matrix $R \in \mathbb{R}^{m \times m}$ for the cost function $J(x, u)$ in (11). The default value is an all-zero matrix.
- `.Qlqr` state weighting matrix $Q_{lqr} \in \mathbb{R}^{n \times n}$ used to compute the feedback matrix K for the tracking controller (12) with an LQR approach. The default value is the identity matrix.
- `.Rlqr` input weighting matrix $R \in \mathbb{R}^{m \times m}$ used to compute the feedback matrix K for the tracking controller (12) with an LQR approach. The default value is an all-zero matrix.
- `.realTime` flag specifying if the algorithm only switches to a new solution if the computation time is less than the allocated time `Opts.tComp` (`Opts.realTime` = 1), or if this real-time constraint is not considered (`Opts.realTime` = 0). The default value is 1.
- `.tComp` allocated computation time t_{comp}.
- `.alpha` contraction rate α for the contraction constraint (see [15, Eq. (13)]). The default value is $\alpha = 0.1$.
- `.maxIter` maximum number of optimization iterations for the optimal control problem. The default value is 10.

– `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for reachset model predictive control algorithm are provided in Sec. 8.2 and in the directory */examples/reachsetMPC/...* in the AROC toolbox.

2.2.2 Model Predictive Control for Linear Systems

For model predictive control of linear systems AROC implements the MPC algorithm in [16]. This approach considers a sampled-data controller

$$u_c(\widehat{x}(t), t) = u_k + K\left(\widehat{x}(t_k) - x_f\right), \qquad t \in [t_k, t_{k+1}] \tag{13}$$

which updates the control input at discrete time points $t_0 = 0, t_1 = \Delta t, t_2 = 2\Delta t, \dots N\Delta t$ only, where the time step size $\Delta t = t_{opt}/N$ is given by the lookahead time t_{opt} divided by the number of time steps N. If the feedback matrix K in (13) is not provided by the user, it is determined by applying a *Linear Quadratic Regulator* (LQR) approach [7, Chapter 3.3] with weighting matrices Q_{lqr} and R_{lqr}. The correction inputs u_k in (13) are optimized during controller synthesis so that the cost function $J(x, u)$ in (11) is minimized. While for nonlinear systems we first synthesized the control law and then used reachability analysis to verify it (see Sec. 2.2.1), for linear system controller synthesis and verification with reachability analysis can be combined into a single linear optimization problem that can be solved very efficiently. AROC supports the solvers Gurobi[3] and Mosek[4] in addition to the Matlab build-in linear programming solvers, where the best available solver is determined automatically. Since we can only switch to an updated control law at discrete points in time, the allocated computation time t_{comp} (see Sec. 2.2 is given by the size of one time step $t_{comp} = t_{opt}/N$.

The syntax for running the model predictive control algorithm for linear systems is as follows:

$$\texttt{res} = \texttt{linSysMPC}(\texttt{benchmark}, \texttt{Param}, \texttt{Opts})$$

where `benchmark`, `Param`, and `res` are defined as at the beginning of Sec. 2.2, and `Opts` is a struct that contains the following algorithm settings:

– `.tOpt` lookahead time t_{opt} (see Sec. 2.2).

– `.N` number of time steps N. The default value is 10.

– `.termReg:` terminal region $\mathcal{T}$ represented as an object of class `mptPolytope` (see [4, Sec. 2.2.1.4]) or class `terminalRegion` (see Sec. 4).

– `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for the cost function $J(x, u)$ in (11). The default value is the identity matrix.

– `.R` input weighting matrix $R \in \mathbb{R}^{m \times m}$ for the cost function $J(x, u)$ in (11). The default value is an all-zero matrix.

– `.K` feedback matrix $K \in \mathbb{R}^{m \times n}$ for the control law (13).

– `.Qlqr` state weighting matrix $Q_{lqr} \in \mathbb{R}^{n \times n}$ used to compute the feedback matrix K for the control law (13) with an LQR approach (only if `Opts.K` is not specified). The default value is the identity matrix.

– `.Rlqr` input weighting matrix $R \in \mathbb{R}^{m \times m}$ used to compute the feedback matrix K for the control law (13) with an LQR approach (only if `Opts.K` is not specified). The default value is an all-zero matrix.

[3] `https://www.gurobi.com/`
[4] `https://www.mosek.com/`

- `.realTime` flag specifying if the algorithm only switches to a new solution if the computation time is less than the allocated time `Opts.tComp` (`Opts.realTime` = 1), or if this real-time constraint is not considered (`Opts.realTime` = 0). The default value is 1.
- `.alpha` contraction rate α for the contraction constraint (see [16, Eq. (12)]). The default value is $\alpha = 0.1$.
- `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for the model predictive control algorithm for linear systems are provided the directory */examples/linSysMPC/...* in the AROC toolbox.

3 Maneuver Automata

In Sec. 2.1 we described how motion primitive based control algorithms can be used to construct feasible controllers for single motion primitives offline. This section now explains how a maneuver automaton can be generated from these offline generated motion primitives (see Sec 3.1), and how online planning tasks can be solved with this maneuver automaton (see Sec. 3.5). An illustration of online motion planning with a maneuver automaton is shown in Fig. 12.

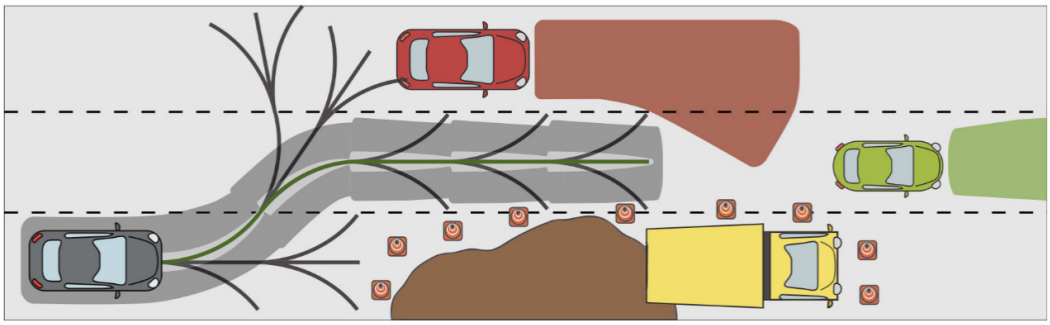


Figure 12: Illustration of online motion planning for an autonomous vehicle with a maneuver automaton. The reachable set of the vehicle center $\mathcal{R}_{uc(\cdot)}(t)$ is shown in light gray, and the occupancy set $\mathcal{O}(t)$ of the vehicle is depicted in dark gray.

3.1 Class `maneuverAutomaton`

Maneuver automata are in AROC represented by the class `maneuverAutomaton`. An object of class `maneuverAutomaton` can be constructed as follows:

$$\texttt{obj} = \texttt{maneuverAutomaton}(\texttt{primitives}, \texttt{shiftFun}, \texttt{shiftOccFun}),$$

where `obj` is an object of class `maneuverAutomaton`, and the input arguments are defined as:

- `primitives` MATLAB cell-array storing the motion primitives, where each motion primitive is represented as an object of class `objController` (see Sec. 7.3 and Sec. 2.1).
- `shiftFun` MATLAB function handle to a system specific function `shiftInitSet` that describes how to translate a set of system states under consideration of invariant states (see Sec. 3.3).
- `shiftOccFun` MATLAB function handle to a system specific function `shiftOccupancySet` that describes how to translate the occupancy set under consideration of invariant states (see Sec. 3.3).

When an object of class `maneuverAutomaton` is constructed, it is automatically determined which motion primitives can be connected to each other. The resulting connectivity matrix is then stored in the property `.conMat` of the class `maneuverAutomaton`. Two motion primitives can be connected to each other if the final reachable set of the first motion primitive is a subset of the initial set of the second motion primitive. Since of course a maneuver automaton with many connections is desirable, it is important that the motion primitive based control algorithms described in Sec. 2.1 are able to contract the reachable set to ensure high connectivity.

3.2 Function `postprocessing`

Usually, the reachable set of the controlled system $\mathcal{R}_{uc(\cdot)}(t)$ (see (4)) as computed by the motion primitive based control algorithms in Sec. 2.1 only describes the position of a certain reference point. For the autonomous vehicle benchmark in Sec. 6.6 the control algorithms for example compute the reachable set of the center of mass. However, for online motion planning one also has to consider the dimensions of the vehicle when testing for collisions with static and dynamic obstacles (see Fig. 12). We call the reachable set bloated by the vehicle dimensions the occupancy set $\mathcal{O}(t)$ of the system since this set describes the space that is occupied by the vehicle. As an example we consider the occupancy set for the autonomous vehicle benchmark in Sec. 6.6, which is

$$\mathcal{O}(t) = \left\{ \begin{bmatrix} x_3 + \cos(x_2)\delta_1 - \sin(x_2)\delta_2 \\ x_4 + \sin(x_2)\delta_1 + \cos(x_2)\delta_2 \end{bmatrix} \;\middle|\; x \in \mathcal{R}_{uc(\cdot)}(t),\ \delta_1 \in \left[-\frac{l}{2}, \frac{l}{2} \right],\ \delta_2 \in \left[-\frac{w}{2}, \frac{w}{2} \right] \right\},$$

where $l \in \mathbb{R}^+$ is the length and $w \in \mathbb{R}^+$ the width of the vehicle. Note that the occupancy set usually does not have the same dimension as the reachable set.

In order to construct a maneuver automaton from motion primitives one has to provide a system specific function `postprocessing` which computes the occupancy set $\mathcal{O}(t)$ from the reachable set $\mathcal{R}_{uc(\cdot)}(t)$ as an additional input argument `Post` for controller synthesis (see Sec. 2.1). This function is then used internally to automatically compute the occupancy set from the reachable set. The syntax for the function `postprocessing` is as follows:

$$\mathcal{O}(t) = \texttt{postprocessing}(\mathcal{R}_{uc(\cdot)}(t)),$$

where the occupancy set $\mathcal{O}(t)$ and the reachable set $\mathcal{R}_{uc(\cdot)}(t)$ are both represented as MATLAB cell-arrays with each entry being a struct with fields `.set` and `.time`, which store the set and the corresponding time interval, respectively. An example for the system specific implementation of the `postprocessing` function for the autonomous car benchmark in Sec. 6.6 can be found in the file */benchmarks/automaton/postprocessing_car.m* in the AROC toolbox.

3.3 Function `shiftInitSet`

Many systems have invariant states. The autonomous car benchmark in Sec. 6.6 for example is translation invariant as well as rotation invariant, so that the only state that is not invariant is the velocity of the car. Invariant states are very advantageous for the construction of maneuver automata since they allow to shift motion primitives to different positions (see Fig. 12), which significantly reduces the number of motion primitives that are required to solve motion planning problems.

In AROC, the invariance of the system is defined by a system specific function `shiftInitSet` which returns the set $\mathcal{R}_{shift}$ resulting from the translation of a set of initial states $\mathcal{R}_0 \subset \mathbb{R}^n$ to the final state $x_f \in \mathbb{R}^n$ while considering the invariant states:

$$\mathcal{R}_{shift} = \texttt{shiftInitSet}(\mathcal{R}_0, x_f).$$

A MATLAB function handle to the system specific implementation of the function `shiftInitSet` has to be provided for the construction of a maneuver automaton (see Sec. 3.1).

As an example we consider the implementation of the function `shiftInitSet` for the autonomous vehicle benchmark in Sec. 6.6:

$$\mathcal{R}_{shift} = \begin{bmatrix} 0 \\ x_{f,2} \\ x_{f,3} \\ x_{f,4} \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos(x_{f,2}) & -\sin(x_{f,2}) \\ 0 & 0 & \sin(x_{f,2}) & \cos(x_{f,2}) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos(c_2) & -\sin(c_2) \\ 0 & 0 & \sin(c_2) & \cos(c_2) \end{bmatrix}^{-1} \left(\mathcal{R}_0 - \begin{bmatrix} 0 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \right),$$

where $c = \texttt{center}(\mathcal{R}_0)$ is the center of the initial set. The velocity, which is state x_1 is not changed since it is not an invariant state. However, the orientation x_2 and the positions x_3 and x_4 are updated to the orientation and positions of the final states due to the translation and rotation invariance of the system. Furthermore, the positions x_3 and x_4 are rotated due to the change in orientation. The function `shiftInitSet` for the autonomous vehicle benchmark is implemented in the file */benchmarks/automaton/shiftInitSet_car.m* in the AROC toolbox.

3.4 Function `shiftOccupancySet`

As described in Sec. 3.3 invariant states of the system allow us to shift motion primitives to different positions. When doing so, we of course also have to update the occupancy set $\mathcal{O}(t)$ (see Sec. 3.2) for these motion primitives. In AROC, the rules for updating the occupancy set can be specified with a system specific function `shiftOccupancySet` which returns the new occupancy set after shifting the motion primitive to the new initial state $x_0 \in \mathbb{R}^n$ at time $t \in \mathbb{R}^+$:

$$\mathcal{O}_{shift}(t) = \texttt{shiftOccupancySet}(\mathcal{O}(t), x_0, t),$$

where $\mathcal{O}(t)$ is the original occupancy set of the motion primitive. A MATLAB function handle to the system specific implementation of the function `shiftOccupancySet` has to be provided for the construction of a maneuver automaton (see Sec. 3.1).

As an example we consider the implementation of the function `shiftOccupancySet` for the autonomous vehicle benchmark in Sec. 6.6:

$$\mathcal{O}_{shift}(t) = \begin{bmatrix} x_{0,3} \\ x_{0,4} \end{bmatrix} + \begin{bmatrix} \cos(x_{0,2}) & -\sin(x_{0,2}) \\ \sin(x_{0,2}) & \cos(x_{0,2}) \end{bmatrix} \mathcal{O}(t),$$

where we assume without loss of generality that the initial orientation and positions of the motion primitive are equal to 0. The function `shiftOccupancySet` for the autonomous vehicle benchmark is implemented in the file */benchmarks/automaton/shiftOccupancySet_car.m* in the AROC toolbox.

3.5 Motion Planner

Given an offline constructed maneuver automaton, motion planning is reduced to the task of solving a classical search problem (see Fig. 12), which can be implemented very efficiently and is therefore well suited for online control. In AROC, online motion planning with a maneuver automaton is implemented in the function `motionPlanner`:

$\texttt{ind} = \texttt{motionPlanner}(\texttt{obj}, x_0, \texttt{goalSet}, \texttt{statObs}, \texttt{dynObs}, \texttt{search})$

$\texttt{ind} = \texttt{motionPlanner}(\texttt{obj}, x_0, \texttt{goalSet}, \texttt{statObs}, \texttt{dynObs}, \texttt{search}, \texttt{costFun})$

$\texttt{ind} = \texttt{motionPlanner}(\texttt{obj}, x_0, \texttt{goalSet}, \texttt{statObs}, \texttt{dynObs}, \texttt{search}, \texttt{costFun}, \texttt{goalFun}),$

where `ind` is a vector that stores the indices of the motion primitives that correspond to the planned trajectory, and the input arguments are defined as

- `obj` object of class `maneuverAutomaton` that represents the maneuver automaton that is used for online planning.
- x_0 Inital state $x_0 \in \mathbb{R}^n$ for the motion planning problem.
- `goalSet` target set which should be reached by the system specified as a struct with fields `.set` and `.time` which specify the target set and the corresponding time interval, respectively.
- `statObs` MATLAB cell-array storing the static obstacles for the motion planning problem.

- `dynObs` MATLAB cell-array storing the dynamic obstacles for the motion planning problem, where each entry of the cell-array is a struct with fields `.set` and `.time` which store the set and the corresponding time interval of the dynamic obstacles.
- `search` string specifying the search algorithm that is used to solve the motion planning problem. The available algorithms are depth-first search (`'depth-first'`), breadth-first search (`'breadth-first'`), and A* search (`'Astar'`).
- `costFun` function handle to a custom cost function for A* search.
- `goalFun` function handle to a custom function for checking if the goal set for motion planning has been reached.

The sets for goal set, static obstacles, and dynamic obstacles can be represented by any of the set representations from the CORA toolbox [4, Sec. 2.2.1]. For the car and ship benchmarks in Sec. 6.6 and Sec. 6.9, the parameter x_0, `goalSet`, `statObs`, and `dynObs` which define the motion planning problem can be conveniently loaded from CommonRoad or CommonOcean files (see Sec. 1.7). The trajectory planned by the motion planner can be visualized and simulated using the functions `plotPlannedTrajectory`, `simulate`, and `simulateRandom` located in the directory */classes/maneuverAutomaton/...*. Code examples that demonstrate the construction of a maneuver automaton as well as online planning using the function `motionPlanner` are provided in Sec. 8.3 and in the directory */examples/maneuverAutomaton/...* in the AROC toolbox.

4 Terminal Region

For the model predictive control algorithms described in Sec. 2.2 one requires a terminal region $\mathcal{T}$. In this section we present algorithms to compute terminal regions. These algorithm first synthesize a terminal controller $u_c(x(t), t)$, and then compute a terminal region for the controlled system. The terminal region is defined as a set $\mathcal{T} \subset \mathbb{R}^n$ for which the terminal controller steers all points inside the set in finite time back into the set, while satisfying input and state constraints and considering disturbances acting on the system:

$$\begin{aligned}\mathcal{T} = \Big\{ x \;\Big|\; \exists t_f \in \mathbb{R}^+ :\; & \forall w(\cdot) \in \mathcal{W},\; \forall v(\cdot) \in \mathcal{V} :\; \xi(t_f, x, u_c(\cdot), w(\cdot), v(\cdot)) \in \mathcal{T}, \\ & \forall t \in [0, t_f],\; \forall \widehat{x}(t) \in \mathcal{R}_{u_c(\cdot)}(t) \oplus \mathcal{V} :\; u_c(\widehat{x}(t), t) \in \mathcal{U}, \\ & \forall t \in [0, t_f],\; \forall w(\cdot) \in \mathcal{W},\; \forall v(\cdot) \in \mathcal{V} :\; \xi\big(t, x, u_c(\cdot), w(\cdot), v(\cdot)\big) \in \mathcal{X} \Big\}\end{aligned} \tag{14}$$

In AROC, terminal regions can be computed with the function `computeTerminalRegion`, which is defined as follows:

$$\mathcal{T} = \texttt{computeTerminalRegion}(\texttt{bemchmark}, \texttt{alg}, \texttt{Param}, \texttt{Opts}),$$

where the output argument is the terminal region $\mathcal{T}$ represented as an object of class `terminalRegion` (see Sec. 7.4), and the input arguments are defined as follows:

- `benchmark` name of the benchmark system that is considered (see Sec. 6).
- `alg` string specifying the algorithm that is used to compute the terminal region. The available algorithms are `'subpaving'` (see Sec. 4.1) and `'zonoLinSys'` (see Sec. 4.2).
- `Param` struct containing the system parameter
  - `.U` set of input constraints $\mathcal{U}$ (see (1)) represented as an object of class `interval` (see [4, Sec. 2.2.1.2]).
  - `.W` set of disturbances $\mathcal{W}$ (see (1)) represented as an object of class `interval` or `zonotope` (see [4, Sec. 2.2.1]).
  - `.V` set of measurement errors $\mathcal{V}$ (see (2)) represented as an object of class `interval` or `zonotope` (see [4, Sec. 2.2.1]).
  - `.X` set of state constraints $\mathcal{X}$ (see (1)) represented as an object of class `mptPolytope` (see [4, Sec. 2.2.1.4]).
- `Opts` struct containing algorithm settings. Since the settings are different for each algorithm they are documented in Sec. 4.1.

Next, we describe the different algorithms AROC provides for computing terminal regions in detail.

4.1 Subpaving Algorithm

The approach considered first is the one in [17]. This approach represents the terminal region as a subpaving, which is defined as the union of non-overlapping boxes [18, Chapter 3]. A schematic visualization of the subpaving algorithm is shown in Fig. 13.

For the computation of the terminal region an equilibrium point $x_{eq} \in \mathbb{R}^n$, $u_{eq} \in \mathbb{R}^m$ of the system satisfying $\dot{x} = f(x_{eq}, u_{eq}, \mathbf{0}) = \mathbf{0}$ is considered. If the feedback matrix $K \in \mathbb{R}^{m \times n}$ for the terminal controller

$$u_c(\widehat{x}(t), t) = u_{eq} + K\big(\widehat{x}(t) - x_{eq}\big) \tag{15}$$

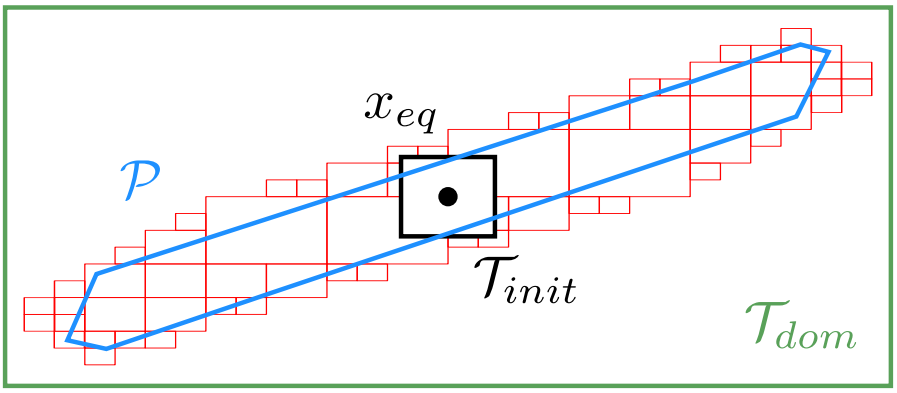


Figure 13: Illustration of the subpaving algorithm for the computation of a terminal region.

is not provided it is computed by applying a *Linear Quadratic Regualtor* (LQR) approach [7, Chapter 3.3] to the system linearized at the equilibrium point. Furthermore, the approach requires a set $\mathcal{T}_{init} \subset \mathbb{R}^n$ with $x_{eq} \in \mathcal{T}_{init}$, which is guaranteed to be part of the terminal region. Starting from a search domain $\mathcal{T}_{dom}$, the algorithm then recursively divides $\mathcal{T}_{dom}$ into smaller boxes, and checks for each box if the reachable set of the controlled system starting from the box reaches the terminal set $\mathcal{T}_{init}$ while satisfying input and state constraints. If this is the case, the box is added to the subpaving representing the terminal region. Otherwise, the box is divided into smaller boxes and the procedure is repeated until a certain recursion limit is reached. Subpavings have the disadvantage that containment checks as required for model predictive control are computationally demanding. Our implementation of the algorithm therefore automatically computes a polytope $\mathcal{P} \subseteq \mathcal{T}$ which is an inner-approximation of the subpaving, and therefore guaranteed to be contained in the terminal region. For polytopes, set containment checks are very fast.

The settings for the subpaving algorithm, which are specified as fields of the struct `Opts` (see Sec. 4), are as follows:

- `.Tdomain` search domain $\mathcal{T}_{dom} \subset \mathbb{R}^n$ represented as an object of class `interval` (see [4, Sec. 2.2.1.2]).
- `.Tinit` initial guess $\mathcal{T}_{init} \subset \mathcal{T}$ for the terminal region represented as an object of class `interval` (see [4, Sec. 2.2.1.2]).
- `.xEq` equilibrium point $x_{eq} \in \mathbb{R}^n$.
- `.uEq` control input for the equilibrium point $u_{eq} \in \mathbb{R}^m$.
- `.numRef` recursion limit `Opts.numRef` > 1 for the number of refinement of the box sizes. The default value is 4.
- `.enlargeFac` enlargement factor `Opts.enlargeFac` > 1 applied to enlarge the initial guess T_{init} in order to accelerate the computation. The default value is 1.5.
- `.tMax` final time for reachability analysis.
- `.reachSteps` number of reachability steps. The default value is 100.
- `.K` feedback matrix $K \in \mathbb{R}^{m \times n}$ for the terminal controller.
- `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ used to compute the feedback matrix K for the terminal controller with an LQR approach (only if `Opts.K` is not specified). The default value is the identity matrix.

- `.R` input weighting matrix $R \in \mathbb{R}^{m \times m}$ used to compute the feedback matrix K for the terminal controller with an LQR approach (only if `Opts.K` is not specified). The default value is the identity matrix..
- `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for terminal region construction using the subpaving algorithm are provided in the directory */example/terminalRegion/examples/...* in the AROC toolbox.

4.2 Zonotope Approach for Linear Systems

Another approach for computing terminal regions is the one from [19], which specializes on linear systems and represents the terminal region as a zonotope. As for the subpaving algorithm the terminal region around an equilibrium point $x_{eq} \in \mathbb{R}^n$, $u_{eq} \in \mathbb{R}^m$ satisfying $\dot{x} = f(x_{eq}, u_{eq}, \mathbf{0}) = \mathbf{0}$ is computed. The high-level concept of the algorithm consists of two phases:

1. First, a terminal set $\mathcal{T}_{fin}$ is constructed by computing the reachable set of the controlled system starting from a search domain $\mathcal{T}_{dom}$ and from the equilibrium point x_{eq} in parallel until both reachable sets converge to a common set with tolerance $\epsilon > 0$.
2. The aim of the second step is to determine a larger terminal set $\mathcal{T}$ by solving a convex optimization problem with the goal of steering all states in N time steps into the previously determined terminal set $\mathcal{T}_{fin}$.

A schematic visualization of this approach is shown in Fig. 14.

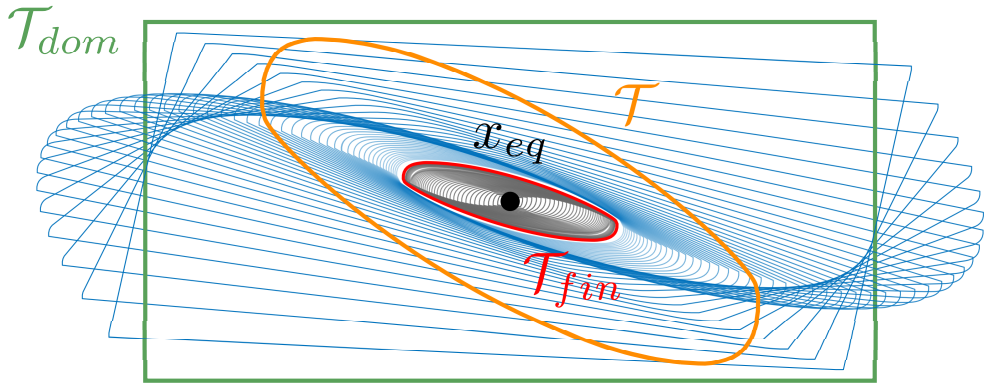


Figure 14: Illustration of terminal region computation using the zonotope approach for linear systems, where the reachable sets for phase one starting from the search domain $\mathcal{T}_{dom}$ and from the equilibrium point x_{eq} are depicted in blue and black, respectively.

The approach considers a sampled-data controller that updates the control input at discrete time points $t_0 = 0, t_1 = \Delta t, t_2 = 2\Delta t, \dots$ only. For the first phase the control law

$$u_c(\hat{x}(t), t) = u_{eq} + K\,(\hat{x}(t_k) - x_{eq}), \qquad t \in [t_k, t_{k+1}]$$

is used for the terminal controller, while for the second phase the control law

$$u_c(\hat{x}(t), t) = u_{eq} + K\,(\hat{x}(t_k) - x_{eq}) + c_{u,k} + G_{u,k}\,\alpha_{init}, \qquad t \in [t_k, t_{k+1}] \tag{16}$$

is applied. The value α_{init} in (16) representing the zonotope factors that correspond to the initial state $x(0)$ is determined from the equation

$$x(0) = c_{init} + G_{init}\,\alpha_{init},$$

where c_{init} and G_{init} are the center and generator matrix of the zonotope that represents the terminal region $\mathcal{T}$. The values $c_{u,k}$ and $G_{u,k}$ that define a correction control input are optimized together with proper scaling factors for the generators of the zonotope $\mathcal{T}$ by the optimization problem that is solved in the second phase. If the feedback matrix $K \in \mathbb{R}^{m \times n}$ is not specified a suitable feedback matrix is determined by applying a *Linear Quadratic Regualtor* (LQR) approach [7, Chapter 3.3]. For the convex optimization problem in phase two the AROC supports the solvers Gurobi[5] and Mosek[6] in addition to the Matlab build-in solvers, where the best available sover is determined automatically.

The settings for the zonotope approach for linear systems, which are specified as fields of the struct `Opts` (see Sec. 4), are as follows:

- `.Tdomain` search domain $\mathcal{T}_{dom} \subset \mathbb{R}^n$ represented as an object of class `interval` (see [4, Sec. 2.2.1.2]).
- `.xEq` equilibrium point $x_{eq} \in \mathbb{R}^n$. The default value is $\mathbf{0}$.
- `.uEq` control input for the equilibrium point $u_{eq} \in \mathbb{R}^m$. The default value is $\mathbf{0}$.
- `.timeStep` time step size Δt for the sampled-data controller.
- `.N` number of time steps. The default value is 10.
- `.K` feedback matrix $K \in \mathbb{R}^{m \times n}$ for the terminal controller.
- `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ used to compute the feedback matrix K for the terminal controller with a LQR approach (only if `Opts.K` is not specified). The default value is the identity matrix.
- `.R` input weighting matrix $R \in \mathbb{R}^{m \times m}$ used to compute the feedback matrix K for the terminal controller with a LQR approach (only if `Opts.K` is not specified). The default value is the identity matrix.
- `.maxDist` Convergence tolerance ϵ for the computation of $\mathcal{T}_{fin}$ during the first phase of the algorithm. The default value is 0.01.
- `.genMethod` String specifying the method that is used to select the generator matrix G_{init} for the terminal set $\mathcal{T}$. The available methods are `'termSet'` (same generator matrix as $\mathcal{T}_{fin}$), `'sampling2D'` (uniformly sampled directions, for $n = 2$ only), and `'provided'` (matrix provided by user in `Opts.G`). The default value is `'termSet'`.
- `.G` Generator matrix G_{init} for the terminal set $\mathcal{T}$ (only if `Opts.genMethod='provided'`).
- `.costFun` String specifying the cost function used for the optimization problem in the second phase. The available cost functions are `'sum'` (sum of scaling factors), `'geomean'` (geometric mean of scaling factors) and `'none'` (terminate algorithm after phase one). The default value is `'sum'`.
- `.cora` struct containing the settings for reachability analysis using the CORA toolbox (see Sec. 7.7).

Code examples for terminal region construction using the zonotope approach for linear systems are provided in Sec. 8.4 and in the directory */examples/terminalRegion/...* in the AROC toolbox.

[5] `https://www.gurobi.com/`
[6] `https://www.mosek.com/`

5 Conformant Synthesis

No matter how accurate a model is, it will never fully capture all behaviors of the real system. To formally verify the real system one therefore has to use an over-approximative model. In AROC we obtain over-approximative models by considering uncertain inputs $w \in \mathcal{W}$ and measurement errors $v \in \mathcal{V}$ (see (1)). Conformant synthesis is a method to determine the set of uncertain inputs $\mathcal{W} \subset \mathbb{R}^q$ and the set of measurement errors $\mathcal{V} \subset \mathbb{R}^n$ from measurements of the real system, where we chose the smallest set $\mathcal{W}$ and $\mathcal{V}$ such that all measurements are covered by the over-approximative model.

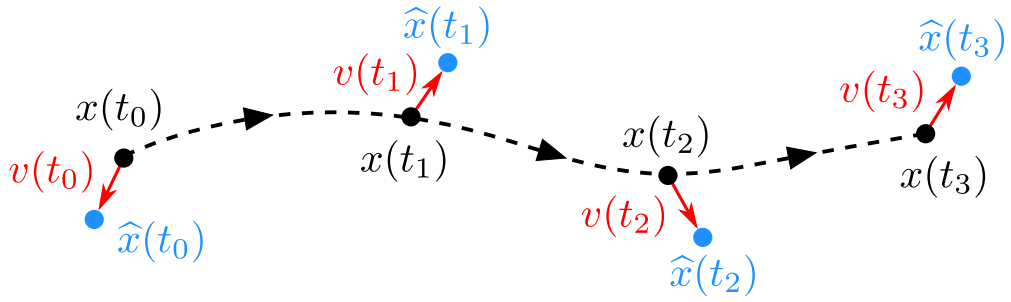


Figure 15: Illustration of conformant synthesis.

In detail, conformant synthesis is defined as follows: Given a measurement $(\widehat{X}, U, T)$ consisting of the measured trajectory $\widehat{X} = [\widehat{x}(t_0), \dots, \widehat{x}(t_K)] \in \mathbb{R}^{n \times K}$, the corresponding piecewise constant input signal $U = [u(t_0), \dots, u(t_{K-1})] \in \mathbb{R}^{m \times (K-1)}$, and the vector of measurement times $T = [t_0, \dots, t_K] \in \mathbb{R}^K$, the conformance checking algorithm determines a sequence of piecewise constant uncertain inputs $w(t_0), \dots, w(t_{K-1})$ and measurement errors $v(t_0), \dots, v(t_K)$ matching the measured trajectory $\widehat{X}$ by solving the following optimization problem:

$$\begin{aligned} \min_{\substack{w(t_0),\dots,w(t_{K-1}) \\ v(t_0),\dots,v(t_K)}} \quad & \mu \Big(\sum_{i=1}^{K-1} w(t_i)^T w(t_i) \Big) + (1-\mu) \Big(\sum_{i=1}^{K} v(t_i)^T v(t_i) \Big) \\ s.t. \quad & x(t_{i+1}) = x(t_i) + \int_0^{t_{i+1}-t_i} f\big(x(t_i), u(t_i), w(t_i)\big)\, dt, \quad \forall i = 0, \dots K-1, \\ & \widehat{x}(t_i) = x(t_i) + v(t_i), \quad \forall i = 0, \dots K, \end{aligned} \tag{17}$$

where the user-defined parameter $\mu \in [0, 1]$ defines how much uncertainty is captured by the set of uncertain inputs $\mathcal{W}$ and now much by the set of measurement errors $\mathcal{V}$. To obtain reliable values for the model uncertainty one has to consider many different measurements from the real system which correspond to all possible system behaviors. We therefore solve a separate optimization problem (17) for each measured trajectory of the system. The resulting uncertain inputs and measurement errors are then tightly enclosed by sets $\mathcal{W}$ and $\mathcal{V}$ such that $\forall i = 1, \dots, K-1 : \; w(t_i) \in \mathcal{W}$ and $\forall i = 1, \dots, K : \; v(t_i) \in \mathcal{V}$. A visualization of conformance checking is shown in Fig. 15.

The syntax for the conformance checking algorithm implemented in AROC is as follows:

$$[\mathcal{W}, \mathcal{V}] = \texttt{conformantSynthesis}(\texttt{benchmark}, \texttt{measurements})$$
$$[\mathcal{W}, \mathcal{V}] = \texttt{conformantSynthesis}(\texttt{benchmark}, \texttt{measurements}, \texttt{Opts}),$$

where the input arguments are defined as follows:

- `benchmark` name of the benchmark system that is considered (see Sec. 6).
- `measurements` cell-array storing the measurements of the system as a list, where each entry corresponds to one measured trajectory, which is represented as a struct with the following fields:
  - `.x` matrix $\widehat{X} = [\widehat{x}(t_0), \dots, \widehat{x}(t_K)] \in \mathbb{R}^{n \times K}$ storing the measured system states.
  - `.u` matrix $U = [u(t_0), \dots, u(t_{K-1})] \in \mathbb{R}^{m \times (K-1)}$ storing the piecewise constant control inputs corresponding to the measured trajectory.
  - `.t` vector $T = [t_0, \dots, t_K] \in \mathbb{R}^K$ storing the discrete time points at which the system state is measured.
- `Opts` struct containing algorithm settings:
  - `.set` string specifying the set representation that is used to represent the sets $\mathcal{W}$ and $\mathcal{V}$. The available set representations are `'interval'` (see [4, Sec. 2.2.1.2]) and `'zonotope'` (see [4, Sec. 2.2.1.1]). The default value is `'interval'`
  - `.group` Number of measurement points that are grouped together into a single optimization problem. The default value is 10.
  - `.measErr` flag specifying if a set of measurement errors $\mathcal{V}$ is used in addition to the set of uncertain inputs $\mathcal{W}$ to capture the uncertainty (`Opts.measErr = true`) or not (`Opts.measErr = false`). The default value is `false`.
  - `.mu` parameter $\mu \in [0, 1]$ in (17) that defines how much uncertainty is captured by the set of uncertain inputs $\mathcal{W}$ and how much by the set of measurement errors $\mathcal{V}$. The default value is 0.5.

Code examples demonstrating conformant synthesis are provided in Sec. 8.5 and in the directory */example/conformance/...* in the AROC toolbox.

6 Benchmarks

In this section we provide a short description for all benchmark systems contained in the AROC toolbox. New custom benchmarks can be easily added as described in Sec. 7.1. The code for all benchmarks is contained in the directory */benchmarks/..* in the AROC toolbox.

6.1 Double Integrator

The first benchmark system is a simple double integrator that describes a point-mass sliding frictionless on a plane [19, Sec. V.A] (see Fig. 16).

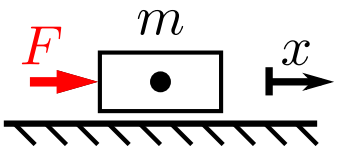


Figure 16: Visualization of the double integrator benchmark system.

The system dynamics for the double integrator is as follows [19, Sec. V.A]:

$$\dot{x}_1 = x_2 + w_1$$
$$\dot{x}_2 = \frac{1}{m}u + w_2,$$

where the system states are the position $x_1 = x$ and the velocity $x_2 = \dot{x}$ of the point-mass, the system input is the force $u = F$, and the weight of the point-mass is $m = 1kg$. The input constraint is $u \in [-1, 1]N$ and the set of disturbances is $w_1 \in [-0.1, 0.1]m/s$ and $w_2 \in [-0.1, 0.1]m/s^2$. Furthermore, we consider the initial set $x_1(0) \in [-0.2, 0.2]m$ and $x_2(0) \in [-0.2, 0.2]m/s$.

The differential equation describing the double integrator is implemented in the file */benchmarks/dynamics/doubleIntegrator.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_doubleIntegrator.m*. The name of the benchmark is `benchmark = 'doubleIntegrator'`.

6.2 Cart

The second benchmark describes a cart that is coupled to the environment with a damping element and a spring with nonlinear stiffness [20, Eq. (3)] (see Fig. 17).

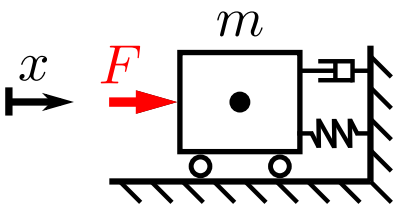


Figure 17: Visualization of the cart benchmark system.

The system dynamics for the cart benchmark is as follows [20, Eq. (3)]:

$$\dot{x}_1 = x_2 + w_1$$
$$\dot{x}_2 = \frac{1}{m}(-d \cdot x_2^2 - k \cdot x_1^3 + u) + w_2,$$

where the system states are the position $x_1 = x$ and the velocity $x_2 = \dot{x}$ of the cart, the system input is the force $u = F$, the weight of the cart is $m = 1kg$, the damping constant is $d = 1kg/m$, and the spring stiffness constant is $k = 1N/m^2$. The input constraint is $u \in [-14, 14]N$ and the set of disturbances is $w_1 \in [-0.1, 0.1]m/s$ and $w_2 \in [-0.1, 0.1]m/s^2$. Furthermore, we consider the initial set $x_1(0) \in [-0.2, 0.2]m$ and $x_2(0) \in [-0.2, 0.2]m/s$.

The differential equation describing the cart benchmark is implemented in the file */benchmarks/dynamics/cart.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_cart.m*. The name of the benchmark is `benchmark = 'cart'`.

6.3 Cartpole

A classical benchmark for nonlinear control is a cartpole, where the task is to balance the pole in its upward instable equilibrium point (see Fig. 18).

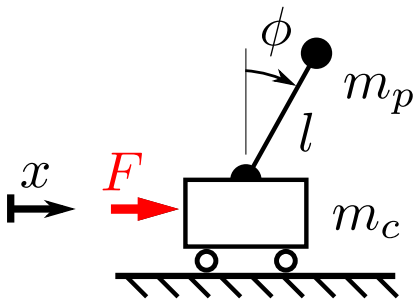


Figure 18: Visualization of the cartpole benchmark system.

The system dynamics for the cartpole benchmark is as follows [21, Eq. (1)(2)]:

$$\dot{x}_1 = x_3$$
$$\dot{x}_2 = x_4$$
$$\dot{x}_3 = \frac{m_p\, l\, x_4^2 \sin(x_2)\cos(x_2)^2 - (m_c+m_p) g \cos(x_2)\sin(x_2) + \cos(x_2)^2 u}{\frac{4}{3}(m_c+m_p)^2 l - m_p (m_c+m_p) l \cos(x_2)^2} + \frac{u + m_p\, l\, x_4^2 \sin(x_2)}{m_c+m_p} + w_1$$
$$\dot{x}_4 = \frac{(m_c+m_p) g \sin(x_2) - m_p\, l\, x_4^2 \sin(x_2)\cos(x_2) - \cos(x_2) u}{\frac{4}{3}(m_c+m_p) l - m_p\, l \cos(x_2)^2} + w_2$$

where the system states are the position $x_1 = x$ and velocity $x_3 = \dot{x}$ of the cart as well as the angle $x_2 = \phi$ and angular velocity $x_4 = \dot{\phi}$ of the pendulum, the system input is the force $u = F$, the weight of the cart is $m_c = 1kg$, the weight of the pole is $m_p = 0.1kg$, the length of the pole $= l = 1m$, and $g = 9.81m/s^2$ is the gravitational constant. The input constraint is $u \in [-10, 10]N$ and the set of disturbances is $w_1 \in [-0.5, 0.5]m/s^2$ and $w_2 \in [-0.02, 0.02]rad/s^2$. Moreover, we consider the initial set $x_1(0) \in [-0.2, 0.2]m$, $x_2(0) \in [-0.02, 0.02]rad$, $x_3(0) \in [-0.2, 0.2]m/s$, and $x_4(0) \in [-0.02, 0.02]rad/s$.

The differential equation describing the cartpole benchmark is implemented in the file */benchmarks/dynamics/cartpole.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_cartpole.m*. The name of the benchmark is `benchmark = 'cartpole'`.

6.4 Stirred Tank Reactor

The next benchmark is taken from [22, Sec. 5] and considers an exothermic, irreversible reaction $A \to B$ of the reactant A to the product B inside a stirred tank reactor (see Fig. 19).

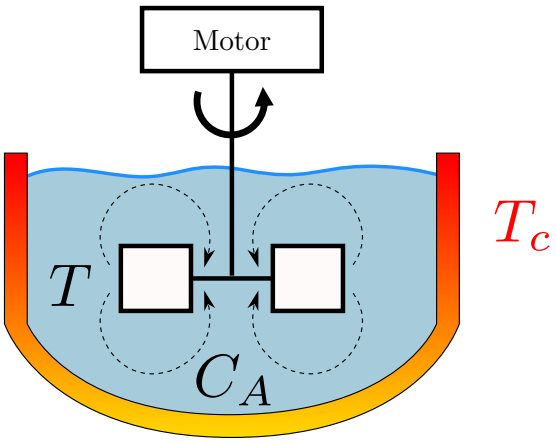


Figure 19: Visualization of the stirred tank reactor benchmark system.

The system dynamics for the stirred tank reactor is as follows [22, Eq. (15)]:

$$\begin{aligned}\dot{x}_1 &= \frac{q}{V}\big(C_{Af} - (x_1 + C_A^{eq})\big) - k_0 \cdot e^{-E/(R(x_2+T^{eq}))} \cdot (x_1 + C_A^{eq}) + w_1 \\ \dot{x}_2 &= \frac{q}{V}\big(T_f - (x_2 + T^{eq})\big) - \frac{\Delta H}{\rho \cdot C_p} k_0 \cdot e^{-E/(R(x_2+T^{eq}))} \cdot (x_1 + C_A^{eq}) \\ &\quad + \frac{UA}{V \cdot \rho \cdot C_p}\big(u + T_c^{eq} - (x_2 + T^{eq})\big) + w_2,\end{aligned}$$

where the system states are the difference of the concentration of reactant A from the equilibrium point $x_1 = C_A - C_A^{eq}$ and the difference of the reactor temperature from the equilibrium point $x_2 = T - T^{eq}$, and the system input is the difference of the cooling stream temperature from the equilibrium point $u = T_c - T_c^{eq}$. The parameter are defined as $C_A^{eq} = 0.5\ mol/l$, $T^{eq} = 350\ K$, $T_c^{eq} = 300\ K$, $q = 5/3\ l/s$, $T_f = 350\ K$, $V = 100\ l$, $\rho = 1000\ g/l$, $C_p = 0.239\ J/g\ K$, $\Delta H = -5 \cdot 10^4\ J/mol$, $E/R = 8750\ K$, $k_0 = 7.2/60 \cdot 10^{10}\ s^{-1}$, $UA = 1/12 \cdot 10^4\ J/s\ K$. The input constraint is $u \in [-20, 70]K$ and the set of disturbances is $w_1 \in [-0.1, 0.1]mol/l\ s^{-1}$ and $w_2 \in [-2, 2]K/s$. Furthermore, we consider the initial set $x_1(0) \in [-0.17, -0.13]mol/l$ and $x_2(0) \in [-48, -43]K$.

The differential equation describing the stirred tank reactor is implemented in the file */benchmarks/dynamics/stirredTankReactor.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_stirredTankReactor.m*. The name of the benchmark is `benchmark = 'stirredTankReactor'`.

6.5 Artificial System

Now, we consider the artificial nonlinear system in [23, Sec. 5]. The system dynamics for the artificial system is as follows [23, Eq. (19)]:

$$\begin{aligned}\dot{x}_1 &= -x_1 + 2x_2 + 0.5u \\ \dot{x}_2 &= -3x_1 + 4x_2 - 0.25x_2^3 - 2u + w.\end{aligned}$$

The input constraint is $u \in [-2, 2]1/min$, and the set of disturbances is $w \in [-0.1, 0.1]1/min$. Furthermore, we consider the initial set $x_1(0) \in [0.5, 0.7]$ and $x_2(0) \in [-0.65, -0.55]$.

The differential equation describing the artificial system is implemented in the file */benchmarks/dynamics/artificialSystem.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_artificialSystem.m*. The name of the benchmark is `benchmark = 'artificialSystem'`.

6.6 Car

One of the most often used benchmarks in AROC is the kinematic single-track model of an autonomous car taken from [8, Sec. 6] (see Fig. 20).

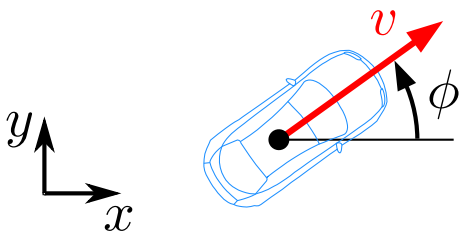


Figure 20: Visualization of the autonomous car benchmark system.

The system dynamics for the autonomous car benchmark is as follows [8, Eq. (19)]:

$$\begin{aligned}\dot{x}_1 &= u_1 + w_1\\ \dot{x}_2 &= u_2 + w_2\\ \dot{x}_3 &= x_1 \cdot \cos(x_2)\\ \dot{x}_4 &= x_1 \cdot \sin(x_2),\end{aligned}$$

where the system states are the velocity $x_1 = v$, the orientation $x_2 = \phi$, and the position $x_3 = x$, $x_4 = y$ of the car. The system inputs are the acceleration u_1 and the normalized steering angle u_2. The input constraints are $u_1 \in [-9.81, 9.81] m/s^2$ and $u_2 \in [-0.4, 0.4] rad/s$, and the set of disturbances is $w_1 \in [-0.5, 0.5] m/s^2$ and $w_2 \in [-0.02, 0.02] rad/s$. Furthermore, we consider the initial set $x_1(0) \in [19.8, 20.2] m/s$, $x_2(0) \in [-0.02, 0.02] rad$, $x_3(0) \in [-0.2, 0.2] m$ and $x_4(0) \in [-0.2, 0.2] m$.

The differential equation describing the autonomous car benchmark is implemented in the file */benchmarks/dynamics/car.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_car.m*. The name of the benchmark is `benchmark = 'car'`.

6.7 Truck

Also the semi-trailer truck benchmark taken from [24, Sec. 6] considers autonomous road traffic (see Fig. 21).

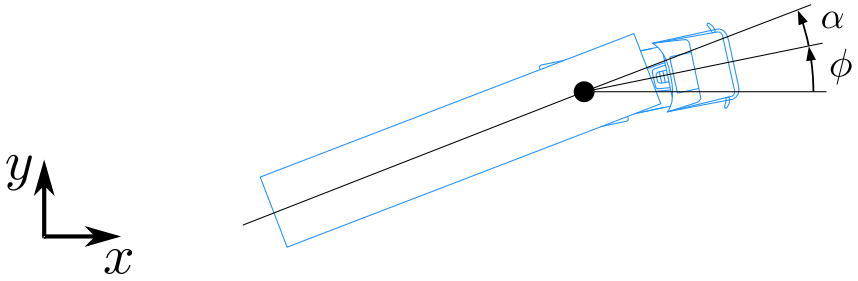


Figure 21: Visualization of the autonomous truck benchmark system.

The system dynamics for the autonomous truck benchmark is as follows [24, Eq. (9)]:

$$\begin{aligned}
\dot{x}_1 &= u_1 + w_1 \\
\dot{x}_2 &= \frac{1}{l_{wb}} \cdot x_4 \cdot \tan(x_1) \\
\dot{x}_3 &= -x_4 \cdot \Big(\frac{1}{l_{wbt}} \sin(x_3) + \frac{1}{l_{wb}} \tan(x_1) \Big) \\
\dot{x}_4 &= u_2 + w_2 \\
\dot{x}_5 &= x_4 \cdot \cos(x_2) \\
\dot{x}_6 &= x_4 \cdot \sin(x_2),
\end{aligned}$$

where the system states are the steering angle $x_1 = \delta$, the orientation of the truck $x_2 = \phi$, the orientation of the trailer $x_3 = \alpha$, the velocity $x_4 = v$, and the position $x_3 = x$, $x_4 = y$ of the trucks rear axis. The system inputs are the steering velocity u_1 and the acceleration u_2, and $l_{wb} = 3.6m$ is the length of the wheelbase of the truck and $l_{wbt} = 8.1m$ is the length of the wheelbase of the trailer. The input constraints are $u_1 \in [-1, 1]rad/s$ and $u_2 \in [-9.81, 9.81]m/s^2$, the set of disturbances is $w_1 \in [-0.02, 0.02]rad/s$ and $w_2 \in [-0.5, 0.5]m/s^2$, and the state constraints are $0.9rad \leq x_1 \leq 0.9rad$ and $-\pi/2 \leq x_3 \leq \pi/2$. Furthermore, we consider the initial set $x_1(0), x_2(0), x_3(0) \in [-0.02, 0.02]rad$, $x_4(0) \in [14.8, 15.2]m/s$, $x_5(0) \in [-0.2, 0.2]m$ and $x_6(0) \in [-0.2, 0.2]m$.

The differential equation describing the autonomous truck benchmark is implemented in the file */benchmarks/dynamics/truck.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_truck.m*. The name of the benchmark is `benchmark = 'truck'`.

6.8 Quadrotor 2D

Another benchmark taken from [25, Sec. IV.B] describes a quadrotor that flies in a 2-dimensional plane (see Fig. 22).

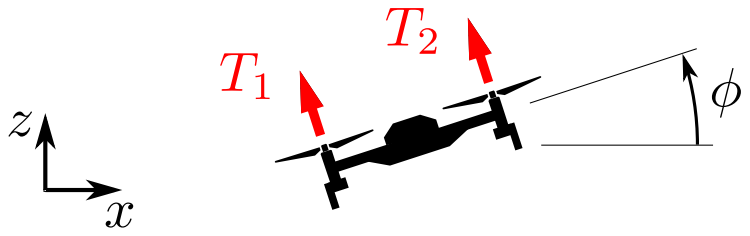


Figure 22: Visualization of the 2D quadrotor benchmark system.

The system dynamics for the 2D quadrotor benchmark is as follows [25, Eq. (3)]:

$$\begin{aligned}
\dot{x}_1 &= x_4 \\
\dot{x}_2 &= x_5 \\
\dot{x}_3 &= x_6 \\
\dot{x}_4 &= \frac{1}{m} \sin(x_3)(u_1 + u_2) + w_1 \\
\dot{x}_5 &= \frac{1}{m} \cos(x_3)(u_1 + u_2) - g + w_2 \\
\dot{x}_6 &= \frac{l}{\sqrt{2}\, I_{yy}} (u_2 - u_1) + w_3,
\end{aligned}$$

where the system states are the positions $x_1 = x$ and $x_2 = z$, the orientation $x_3 = \phi$, the velocities $x_4 = \dot{x}$ and $x_5 = \dot{z}$ in the spatial dimension, and the angular velocity $x_6 = \dot{\phi}$ of the quadrotor. The system inputs are the thrusts $u_1 = T_1$ and $u_2 = T_2$ generated by the two rotors. Moreover, the weight of the quadrotor is $m = 0.027kg$, the moment of inertia is $I_{yy} = 1.4 \cdot 10^{-5} kg\, m^2$, the arm length of the quadrotor is $l = 0.0397m$, and the acceleration due to gravity is $g = 9.81 m/s^2$. The input constraints are $u_1 \in [0, 0.2646]N$ and $u_2 = [0, 0.2646]N$, and the set of disturbances is $w_1 \in [-0.1, 0.1]m/s^2$, $w_2 \in [-0.1, 0.1]m/s^2$, and $w_3 \in [-0.1, 0.1]rad/s^2$. Furthermore, we consider the initial set $x_1(0) \in [-0.1, 0.1]m$, $x_2(0) \in [-0.1, 0.1]m$, $x_3(0) \in [-0.05, 0.05]rad$, $x_4(0) \in [-0.1, 0.1]m/s$, $x_5(0) \in [-0.1, 0.1]m/s$, and $x_6(0) \in [-0.05, 0.05]rad/s$.

The differential equation describing the 2D quadrotor benchmark is implemented in the file */benchmarks/dynamics/quadrotor2D.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_quadrotor2D.m*. The name of the benchmark is `benchmark = 'quadrotor2D'`.

6.9 Ship

The ship benchmark in AROC represents the model of a container vessel taken from [26, Sec. 6] (see Fig. 23).

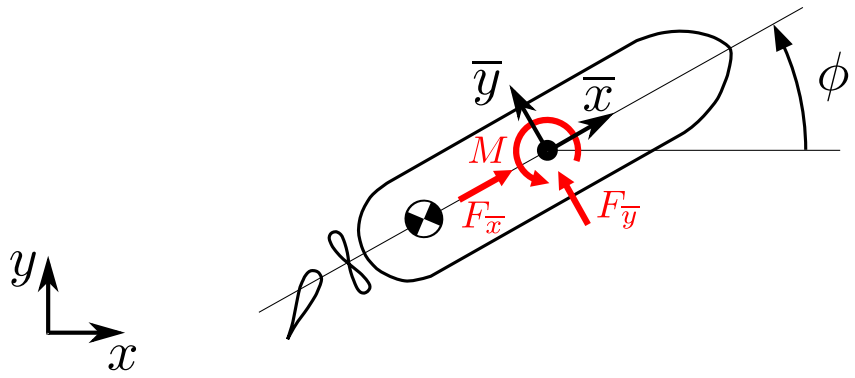


Figure 23: Visualization of the container ship benchmark system.

The system dynamics for the container ship is as follows [26, Eq. (15)]:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} R(x) \begin{bmatrix} x_4 \\ x_5 \\ x_6 \end{bmatrix} \\ M^{-1} \left(-C(x) \begin{bmatrix} x_4 \\ x_5 \\ x_6 \end{bmatrix} - D \begin{bmatrix} x_4 \\ x_5 \\ x_6 \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \right) \end{bmatrix}$$

where the matrices $R(x)$, M^{-1}, $C(x)$, and D are given as

$$R(x) = \begin{bmatrix} \cos(x_3) & \sin(x_3) & 0 \\ \sin(x_3) & \cos(x_3) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad M^{-1} = \begin{bmatrix} \frac{1}{m - X_{\dot{u}}} & 0 & 0 \\ 0 & \frac{I_z - N_{\dot{r}}}{C_m} & \frac{Y_{\dot{r}} - m x_g}{C_m} \\ 0 & \frac{Y_{\dot{r}} - m x_g}{C_m} & \frac{m - Y_{\dot{v}}}{C_m} \end{bmatrix}, \quad D = \begin{bmatrix} -X_u & 0 & 0 \\ 0 & -Y_v & -Y_r \\ 0 & -N_v & -N_r \end{bmatrix},$$

$$C(x) = \begin{bmatrix} 0 & 0 & -m(x_g x_6 + x_5) + Y_{\dot{v}} x_5 + Y_{\dot{r}} x_6 \\ 0 & 0 & m x_6 - X_{\dot{u}} x_4 \\ m(x_g x_6 + x_5) - Y_{\dot{v}} x_5 - Y_{\dot{r}} x_6 & -m x_6 + X_{\dot{u}} x_4 & 0 \end{bmatrix}$$

and $C_m = I_z m - N_{\dot{r}} m - Y_{\dot{v}} I_z + N_{\dot{r}} Y_{\dot{v}} - m^2 x_g^2 - 2 m x_g Y_{\dot{r}} - Y_{\dot{r}}^2$. According to [26, Tab. 2], the total vessel mass is $m = 24.56 \cdot 10^6 kg$, the distance to the center of gravity is $x_g = -2.55m$, the inertia is $I_z = 54.99 \cdot 10^9 kg\, m^2$, the surge aligned force coefficient is $X_u = -150.64 \cdot 10^3$, the sway lateral force coefficient is $Y_v = -218.6$, the yaw lateral force coefficient is $Y_r = 9.277 \cdot 10^3$, the sway moment coefficient is $N_v = -14.23 \cdot 10^3$, the yaw moment coefficient is $N_r = -1.113 \cdot 10^6$, the surge-rate aligned force coefficient is $X_{\dot{u}} = -1.8 \cdot 10^6$, the sway-rate lateral force coefficient is $Y_{\dot{v}} = -26.54 \cdot 10^6$, the yaw-rate lateral force coefficient is $Y_{\dot{r}} = -283.4 \cdot 10^6$, and the yaw-rate moment coefficient is $N_{\dot{r}} = -44.85 \cdot 10^9$. The system states are the position $x_1 = x$, $x_2 = y$ and orientation $x_3 = \phi$ of the vessel, the velocities $x_4 = v_{\overline{x}}$ and $x_5 = v_{\overline{y}}$ in a vessel-fixed coordinate frame, and the angular velocity $x_5 = \dot{\phi}$. The control inputs are the forces $u_1 = F_{\overline{x}}$, $u_2 = F_{\overline{y}}$ expressed in the vessel-fixed coordinate frame as well as the torque $u_3 = M$. According to [26, Tab. 1], the input constraints are $u_1 \in [-5894896.77, 5894896.77]N$, $u_2 \in [-5894896.77, 5894896.77]N$, and $u_3 \in [-1350409.48, 1350409.48]Nm$. Moreover, the disturbance is $w_1 \in [-1000, 1000]N$, $w_2 \in [-1000, 1000]N$, and $w_3 \in [-1000, 1000]Nm$, and we consider the initial set $x_1(0) \in [-1, 1]m$, $x_2(0) \in [-1, 1]m$, $x_3(0) \in [-0.01, 0.01]rad$, $x_4(0) \in [4.8, 5.2]m/s$, $x_5(0) \in [-0.2, 0.2]m/s$, and $x_6(0) \in [-0.001, 0.001]rad/s$.

The differential equation describing the ship benchmark is implemented in the file */benchmarks/dynamics/ship.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_ship.m*. The name of the benchmark is `benchmark = 'ship'`.

6.10 Robot Arm

This benchmark describes a planar robot arm with two rotational joints (see Fig. 24).

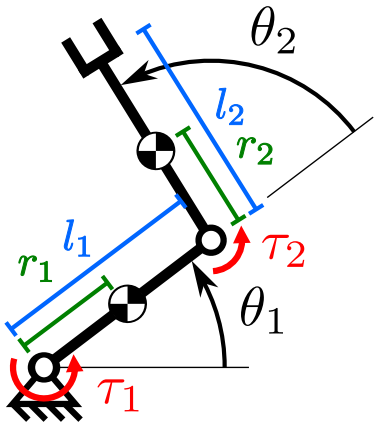


Figure 24: Visualization of the robot arm benchmark system.

The system dynamics for the robot arm benchmark is as follows:

$$
\begin{aligned}
\dot{x}_1 &= x_3 + w_1 \\
\dot{x}_2 &= x_4 + w_2 \\
\dot{x}_3 &= (\beta\delta s_2 + 2\beta^2 s_2 c_2) x_3^2 + 2\beta\delta s_2 x_3 x_4 + \delta\beta s_2 x_4^2 + \delta u_1 - (\delta + 2\beta c_2) u_2 + w_3 \\
\dot{x}_4 &= -(\alpha\beta s_2 + 2\beta^2 s_2 c_2) x_3^2 - (2\beta\delta s_2 + 4\beta^2 s_2 c_2) x_3 x_4 - (\delta\beta s_2 + 2\beta^2 s_2 c_2) x_4^2 \\
&\qquad - (\delta + 2\beta c_2) u_1 + (\alpha + 2\beta c_2) u_2 + w_4,
\end{aligned}
$$

where the system states are the angles $x_1 = \theta_1$ and $x_2 = \theta_2$ and the angular velocities $x_3 = \dot{\theta}_1$ and $x_4 = \dot{\theta}_2$ of the first and the second joint. The system inputs are the joint torques $u_1 = \tau_1$ and $u_2 = \tau_2$. Furthermore, we use the shorthands $c_1 = \cos(\theta_1)$, $s_1 = \sin(\theta_2)$, $c_2 = \cos(\theta_2)$,

$s_2 = \sin(\theta_2)$, $\alpha = m_1 r_1^2 + m_2 l_1^2 + m_2 r_2^2 + I_{z,1} + I_{z,2}$, $\beta = m_2 l_1 r_2$, and $\delta = m_2 r_2^2 + I_{z,2}$. The parameter values are $m_1 = 1kg$, $m_2 = 1kg$, $r_1 = 0.1m$, $r_2 = 0.1m$, $l_1 = 0.2m$, $l_2 = 0.2m$, $I_{z,1} = 1\ kg\ m^2$, and $I_{z,1} = 1\ kg\ m^2$, where $I_{z,1}$, $I_{z,2}$ is the inertia of the two links. The input constraints are $u_1 \in [-3, 3]Nm$ and $u_2 \in [-1, 1]Nm$, the set of disturbances is $w_1, w_2 \in [-0.01, 0.01]rad/s$ and $w_3, w_4 \in [-0.01, 0.01]rad/s^2$, and the state constraints are $0 \leq x_1 \leq \pi$ and $-\pi \leq x_2 \leq \pi$. Furthermore, we consider the initial set $x_1(0), x_2(0) \in [-0.05, 0.05]rad$ and $x_3(0), x_4(0) \in [-0.05, 0.05]rad/s$.

The differential equation describing the robot arm is implemented in the file */benchmarks/dynamics/robotArm.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_robotArm.m*. The name of the benchmark is `benchmark = 'robotArm'`.

6.11 Mobile Robot

Next, we consider the model of a Pioneer 3DX mobile robot (see Fig. 25), which is taken from [27].

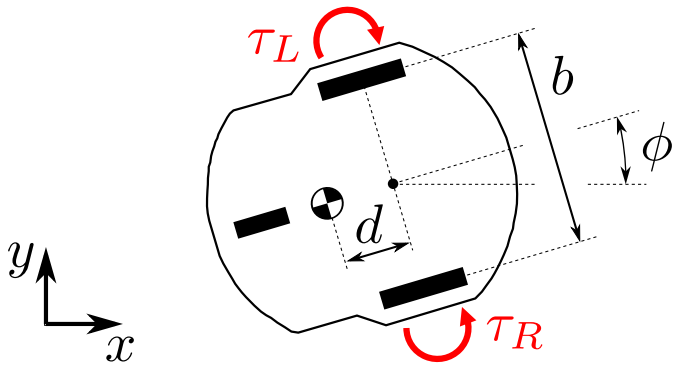


Figure 25: Visualization of the Pioneer 3DX mobile robot.

The system dynamics for the mobile robot benchmark is as follows [27, Sec. 2.1]:

$$
\begin{aligned}
\dot{x}_1 &= \frac{r}{2}(x_4 + x_5)\cos(x_3) \\
\dot{x}_2 &= \frac{r}{2}(x_4 + x_5)\sin(x_3) \\
\dot{x}_3 &= \frac{r}{b}(x_4 - x_5) \\
\dot{x}_4 &= \frac{1}{A^2 - B^2}(Au_1 - AKx_4 - Bu_2 + BKx_5) + w_1 \\
\dot{x}_5 &= \frac{1}{A^2 - B^2}(-Bu_1 + BKx_4 + Au_2 - AKx_5) + w_2,
\end{aligned}
$$

where

$$
A = \frac{mr^2}{4} + \frac{(I + md^2)r^2}{b^2} + I_0, \qquad B = \frac{mr^2}{4} - \frac{(I + md^2)r^2}{b^2}.
$$

The system states are the position $x_1 = x$, $x_2 = y$ and the orientation $x_3 = \phi$ of the mobile robot, as well as the angular velocities $x_4 = \dot{\theta}_R$, $x_5 = \dot{\theta}_L$ of the right and the left actuated wheel. The system inputs are the torques $u_1 = \tau_R$ and $u_2 = \tau_L$ acting on the two actuated wheels. According to [27, Tab. 1] and [27, Tab. 2], the mass of the mobile robot is $m = 28.05kg$, the radius of the wheels is $r = 0.095m$, and the additional parameter are defined as $b = 0.32m$, $d = 0.0578m$, $I = 17.5kgm^2$, $I_0 = 9.24 \cdot 10^{-6}kgm^2$, and $K = 35 \cdot 10^{-7}Nms/rad$. The input constraints are $u_1, u_2 \in [-0.5, 0.5]Nm$, and the set of disturbances is $w_1, w_2 \in [-0.001, 0.001]rad/s^2$.

The differential equation describing the mobile robot is implemented in the file */benchmarks/dynamics/mobileRobot.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_mobileRobot.m*. The name of the benchmark is `benchmark = 'mobileRobot'`.

6.12 Platoon

The last benchmark describes a vehicle platoon with $N = 4$ vehicles (see [6, Sec. IV]). This benchmark can easily be extended to higher dimensions by increasing the number of vehicles N.

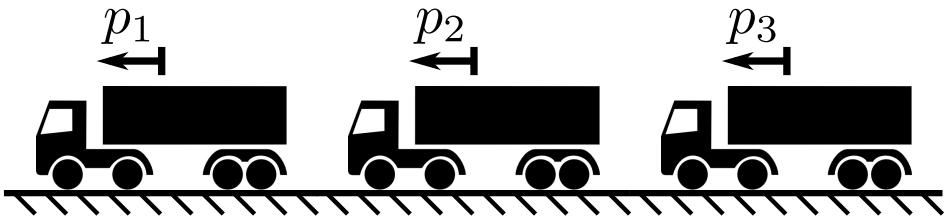


Figure 26: Visualization of a platoon with $N = 3$ vehicles.

The system dynamics for the platoon benchmark is as follows [6, Sec. IV]:

$$\begin{aligned}
\dot{x}_1 &= x_2 & \dot{x}_2 &= u_1 + w_1 \\
\dot{x}_3 &= x_4 & \dot{x}_4 &= u_1 - u_2 + w_1 - w_2 \\
\dot{x}_5 &= x_6 & \dot{x}_6 &= u_2 - u_3 + w_2 - w_3 \\
\dot{x}_7 &= x_8 & \dot{x}_4 &= u_3 - u_4 + w_3 - w_4,
\end{aligned}$$

where the system states are the position $x_1 = p_1$ and velocity $x_2 = v_1$ of the first vehicle, and the relative positions $x_3 = p_1 - p_2 - c_s$, $x_5 = p_2 - p_3 - c_s$, $x_3 = p_1 - p_2 - c_s$ and relative velocities $x_4 = v_1 - v_2$, $x_6 = v_2 - v_3$, $x_8 = v_3 - v_4$ between the remaining vehicles, where $c_s \in \mathbb{R}^+$ is the minimal safe distance. The system inputs are the accelerations u_1, u_2, u_3 and u_4 of the four vehicles. The input constraints are $u_1, u_2, u_3, u_4 \in [-10, 10] m/s^2$, the set of disturbances is $w_1, w_2, w_3, w_4 \in [-1, 1] m/s^2$, and the state constraints are $x_3, x_5, x_7 \geq 0$. Furthermore, we consider the initial set $x_1(0) \in [-0.2, 0.2] m$, $x_2(0) \in [19.8, 20.2] m/s$, $x_3(0), x_5(0), x_7(0) \in [0.8, 1.2] m$, and $x_4(0), x_6(0), x_8(0) \in [-0.2, 0.2] m/s$.

The differential equation describing the platoon benchmark is implemented in the file */benchmarks/dynamics/platoon.m*, and the parameters for the system are specified in the file */benchmarks/parameter/param_platoon.m*. The name of the benchmark is `benchmark = 'platoon'`.

7 Additional Functionality

In this section we document some additional functionality of AROC that was not yet explained in the previous sections.

7.1 Adding New Benchmark Systems

To add a new custom benchmark system to the AROC toolbox one has to create a MATLAB function

$$\texttt{f} = \texttt{benchmarkName}(x, u, w)$$

which implements the nonlinear function $f(x, u, w)$ from the differential equation $\dot{x} = f(x, u, w)$ (see (1)) that describes the system dynamics, where $x \in \mathbb{R}^n$ is the vector of system states, $u \in \mathbb{R}^m$ is the vector of system inputs, and $w \in \mathbb{R}^q$ is the vector of disturbances. The name `benchmarkName` of the function can then be used to select the desired benchmark for the control algorithms (see Sec. 2). In general, it is sufficient if the function that implements the differential equation is located somewhere on the MATLAB path. For the sake of clarity, however, we recommend to store all functions that implement differential equations for benchmark system in the directory */benchmarks/dynamics/...*. Furthermore, we recommend to also store additional benchmark parameters such as input constraints, initial set, etc., in a parameter file located in the directory */benchmarks/parameter/...*.

As an example, we consider the autonomous car benchmark described in Sec. 6.6. The MATLAB function that implements the differential equation for this system is:

```
function f = car(x,u,w)

    f(1,1) = u(1) + w(1);
    f(2,1) = u(2) + w(2);
    f(3,1) = cos(x(2))*x(1);
    f(4,1) = sin(x(2))*x(1);
end
```

7.2 Class `results`

The class `results` stores the reachable set of the controlled system, simulated trajectories of the controlled system, and the reference trajectory. All control algorithms return an object of class `result` (see Sec. 2) which can be used to conveniently visualize and post-process the results from controller synthesis and online application of the controller.

An object of class `results` can be constructed as follows:

$$\texttt{obj} = \texttt{results}(\texttt{reachSet}, \texttt{reachSetTimePoint}, \texttt{refTraj}),$$
$$\texttt{obj} = \texttt{results}(\texttt{reachSet}, \texttt{reachSetTimePoint}, \texttt{refTraj}, \texttt{simulation}),$$

where `obj` is an object of class `results` and the input arguments are defined as follows:

- `reachSet` — object of class `reachSet` (see [4, Sec. 6.1]) storing the reachable set of the controlled system.
- `reachSetTimePoint` — MATLAB cell-array storing the reachable set at the end of each of the `Opts.N` time steps.
- `refTraj` — matrix with n rows and `Opts.N` columns storing the reference trajectory.

- `simulation` MATLAB cell-array storing the different simulated trajectories, where each cell is a struct with fields `.t`, `.x`, and `.u` that store the time, the simulated trajectory, and the control inputs, respectively.

For visualization, the class `results` provides three function `plotReach`, `plotReachTimePoint`, `plotSimulation`, and `animate` which we now explain in detail.

7.2.1 Function `plotReach`

The function `plotReach` visualizes a two-dimensional projection of the reachable set for the controlled system:

$$\begin{aligned}
&\texttt{han} = \texttt{plotReach}(\texttt{obj}),\\
&\texttt{han} = \texttt{plotReach}(\texttt{obj}, \texttt{dim}),\\
&\texttt{han} = \texttt{plotReach}(\texttt{obj}, \texttt{dim}, \texttt{color}),\\
&\texttt{han} = \texttt{plotReach}(\texttt{obj}, \texttt{dim}, \texttt{color}, \texttt{options}),
\end{aligned}$$

where `obj` is an object of class `results` and `han` is a handle to the plotted MATLAB graphics object that can for example be used to add a legend to the plot. The additional input arguments are defined as follows:

- `dim` integer vector $\texttt{dim} \in \mathbb{N}^2_{\leq n}$ specifying the dimensions for which the projection is visualized. The default value is `dim = [1,2]`.
- `color` color of the plotted set. The color can either be specified as a string with line specifications supported by MATLAB[7], e.g., `'r'`, or as a 3-dimensional vector containing the normalized RGB values for the color, e.g., `[1,0,0]`.
- `options` additional MATLAB plot specifications[8] passed as name-value pairs, e.g., `'LineWidth'`. The CORA toolbox defines some additional properties (see [4, Sec. 6.1.3]), e.g., `'Order'`, which are also supported.

Code examples that demonstrate how to use the function `plotReach` are provided in Sec. 8 and in the directory */examples/...* in the AROC toolbox.

7.2.2 Function `plotReachTimePoint`

The function `plotReachTimePoint` visualizes a two-dimensional projection of the time point reachable set for the controlled system at the end of each of the `Opts.N` time steps:

$$\begin{aligned}
&\texttt{han} = \texttt{plotReachTimePoint}(\texttt{obj}),\\
&\texttt{han} = \texttt{plotReachTimePoint}(\texttt{obj}, \texttt{dim}),\\
&\texttt{han} = \texttt{plotReachTimePoint}(\texttt{obj}, \texttt{dim}, \texttt{color}),\\
&\texttt{han} = \texttt{plotReachTimePoint}(\texttt{obj}, \texttt{dim}, \texttt{color}, \texttt{options}),
\end{aligned}$$

where the input and output arguments are defined as in Sec. 7.2.1. Code examples that demonstrate how to use the function `plotReachTimePoint` are provided in Sec. 8 and in the directory */examples/...* in the AROC toolbox.

[7] `https://www.mathworks.com/help/matlab/ref/linespec.html`
[8] `https://www.mathworks.com/help/matlab/ref/matlab.graphics.primitive.patch-properties.html`

7.2.3 Function `plotSimulation`

The function `plotSimulation` visualizes a two-dimensional projection of all simulated trajectories:

```
han = plotSimulation(obj),
han = plotSimulation(obj, dim),
han = plotSimulation(obj, dim, color),
han = plotSimulation(obj, dim, color, options),
```

where the input and output arguments are defined as in Sec. 7.2.1. Code examples that demonstrate how to use the function `plotSimulation` are provided in Sec. 8 and in the directory */examples/...* in the AROC toolbox.

7.2.4 Function `animate`

The function `animate` shows an animation that visualizes the motion of the system for a simulated trajectory of the controlled system:

```
animate(obj, benchmark)
animate(obj, benchmark, statObs, dynObs, goalSet)
animate(obj, benchmark, statObs, dynObs, goalSet, speedUp)
animate(obj, benchmark, statObs, dynObs, goalSet, speedUp, addArg)
```

where `obj` is an object of class `results` and the input arguments are defined as follows:

- `benchmark` name of the benchmark system that is considered (see Sec. 6).
- `statObs` MATLAB cell-array storing the static obstacles for the motion planning problem.
- `dynObs` MATLAB cell-array storing the dynamic obstacles for the motion planning problem, where each entry of the cell-array is a struct with fields `.set` and `.time` which store the set and the corresponding time interval of the dynamic obstacles.
- `goalSet` target set which should be reached by the system specified as a stuct with fields `.set` and `.time` which specify the target set and the corresponding time interval, respectively.
- `speedUp` speed-up factor for the time to make the animation run faster (`speedUp` > 1) or slower (`speedUp` < 1).
- `addArg` additional benchmark specific arguments, like for example lanelets for the autonomous car benchmark.

The function `createVideo` can be used to create a video from an animation.

7.3 Class `objController`

As described in Sec. 1.4, the class `objController` is the parent class for all controller objects that belong to motion primitive based control algorithms (see Sec. 2.1) and store the constructed control law for one motion primitive. The class `objController` defines certain properties which store all information required to construct a maneuver automaton from a list of motion primitives with controllers represented as objects of class `objController` (see Sec. 3). Since the child classes inherit these properties, it is therefore possible to construct a maneuver automaton using any

of the motion primitive based controllers from Sec. 2.1. In addition, the class `objController` provides the two functions `simulate` (see Sec. 7.3.1) and `simulateRandom` (see Sec. 7.3.2), which simulate the online application of the constructed controller.

An object of class `objController` can be constructed as follows:

$$\texttt{obj} = \texttt{objController}(\texttt{dyn}, \texttt{Rfin}, \texttt{Param}),$$
$$\texttt{obj} = \texttt{objController}(\texttt{dyn}, \texttt{Rfin}, \texttt{Param}, \texttt{occSet}),$$

where `obj` is an object of class `objController` and the input arguments are defined as follows:

- `dyn` MATLAB function handle to the function $f(x, u, w)$ in (1) describing the dynamics of the open-loop system.
- `Rfin` final reachable set $\mathcal{R}_{uc(\cdot)}(t_f)$ at the end of the motion primitive represented by any of the set representations from the CORA toolbox (see [4, Sec. 2.2.1]).
- `Param` struct containing the parameter that define the control problem (see Sec. 2.1).
- `occSet` occupancy set (see Sec. 3) stored as a MATLAB cell-array, where each cell is a struct with fields `.set` and `.time`, which store the occupancy set and the corresponding time interval, respectively. Only required if the resulting object of class `objController` is used to construct a maneuver automaton.

7.3.1 Function `simulate`

The function `simulate` simulates the closed-loop system for an initial point $x_0 \in \mathbb{R}_0$, a specific disturbance signal $w(t) \in \mathcal{W}$, and a specific measurement error signal $v(t) \in \mathcal{V}$:

$$[\texttt{res}, \texttt{t}, \texttt{x}, \texttt{u}] = \texttt{simulate}(\texttt{obj}, x_0, w(t))$$
$$[\texttt{res}, \texttt{t}, \texttt{x}, \texttt{u}] = \texttt{simulate}(\texttt{obj}, x_0, w(t), v(t)),$$

where `obj` is an object of any class that is a child of class `objController`, `res` is an object of class `results` (see Sec. 7.2), and $\texttt{t} \in \mathbb{R}^M$, $\texttt{x} \in \mathbb{R}^{M \times n}$, and $\texttt{u} \in \mathbb{R}^{M \times m}$ store the time, the states, and the inputs of the simulated trajectory, respectively, with $M \in \mathbb{N}^+$ being the number of simulation time steps. For the disturbance signal $w(t)$ and the measurement errors $v(t)$ we consider piecewise constant signals with $D \in \mathbb{N}^+$ segments, so that $w(t)$ and $v(t)$ are specified as a matrix $w(t) \in \mathbb{R}^{q \times D}$ and $v(t) \in \mathbb{R}^{n \times D}$.

7.3.2 Function `simulateRandom`

The function `simulateRandom` simulates the closed-loop system for $E \in \mathbb{N}^+$ randomly selected initial points $x_0 \in \mathcal{R}_0$ and randomly selected input signals $w(t)$:

$$[\texttt{res}, \texttt{t}, \texttt{x}, \texttt{u}] = \texttt{simulateRandom}(\texttt{obj})$$
$$[\texttt{res}, \texttt{t}, \texttt{x}, \texttt{u}] = \texttt{simulateRandom}(\texttt{obj}, E, \texttt{fracVert}, \texttt{fracDistVert}, D),$$

where `obj` is an object of any class that is a child of class `objController`, `res` is an object of class `results` (see Sec. 7.2), `fracVert` $\in [0, 1]$ is the fraction of initial points drawn randomly from the vertices of the initial set $\mathcal{R}_0$, `fracDistVert` $\in [0, 1]$ is the fraction of disturbance values drawn randomly from the vertices of the disturbance set $\mathcal{W}$, and $D \in \mathbb{N}^+$ is the number of segments for the piecewise constant disturbance signals $w(t)$ (see Sec. 7.3.1). Code examples that demonstrate how to use the function `simulateRandom` are provided in Sec. 8 and in the directory */examples/...* in the AROC toolbox.

7.4 Class `terminalRegion`

The class `terminalRegion` represents terminal regions computed with one of the algorithms from Sec. 4. The class stores the set which represents the terminal region as well as the parameter for the terminal controller. An object of class `terminalRegion` can be constructed as follows:

$$\texttt{obj} = \texttt{terminalRegion}(\texttt{dyn}, \texttt{set}, \texttt{Param}),$$

where `obj` is an object of class `terminalRegion` and the input arguments are defined as follows:

- `dyn` MATLAB function handle to the function $f(x, u, w)$ in (1) describing the dynamics of the open-loop system.
- `set` terminal region $\mathcal{T} \subset \mathbb{R}^n$ represented by any of the set representations from the CORA toolbox (see [4, Sec. 2.2.1]).
- `Param` struct containing the system parameter (see Sec. 4).

7.4.1 Function `simulate`

The function `simulate` simulates the closed-loop system controlled by the terminal controller for an initial point $x_0 \in \mathbb{R}_0$, a specific disturbance signal $w(t) \in \mathcal{W}$, and a specific measurement error signal $v(t) \in \mathcal{V}$:

$$[\texttt{res}, \texttt{t}, \texttt{x}, \texttt{u}] = \texttt{simulate}(\texttt{obj}, x_0, t_f, w(t))$$
$$[\texttt{res}, \texttt{t}, \texttt{x}, \texttt{u}] = \texttt{simulate}(\texttt{obj}, x_0, t_f, w(t), v(t)),$$

where `obj` is an object of any class that is a child of class `terminalRegion`, `res` is an object of class `results` (see Sec. 7.2), $t_f \in \mathbb{R}^+$ is the final time of the simulation, and $\texttt{t} \in \mathbb{R}^M$, $\texttt{x} \in \mathbb{R}^{M \times n}$, and $\texttt{u} \in \mathbb{R}^{M \times m}$ store the time, the states, and the inputs of the simulated trajectory, respectively, with $M \in \mathbb{N}^+$ being the number of simulation time steps. For the disturbance signal $w(t)$ and the measurement errors $v(t)$ we consider piecewise constant signals with $D \in \mathbb{N}^+$ segments, so that $w(t)$ and $v(t)$ are specified as matrices $w(t) \in \mathbb{R}^{q \times D}$ and $v(t) \in \mathbb{R}^{n \times D}$.

7.4.2 Function `simulateRandom`

The function `simulateRandom` simulates the closed-loop system controlled by the terminal controller for $E \in \mathbb{N}^+$ randomly selected initial points $x_0 \in \mathcal{T}$ inside the terminal region and randomly selected input signals $w(t)$:

$$[\texttt{res}, \texttt{t}, \texttt{x}, \texttt{u}] = \texttt{simulateRandom}(\texttt{obj}, t_f)$$
$$[\texttt{res}, \texttt{t}, \texttt{x}, \texttt{u}] = \texttt{simulateRandom}(\texttt{obj}, t_f, E, \texttt{fracVert}, \texttt{fracDistVert}, D),$$

where `obj` is an object of any class that is a child of class `terminalRegion`, $t_f \in \mathbb{R}^+$ is the final time of the simulation, `res` is an object of class `results` (see Sec. 7.2), $\texttt{fracVert} \in [0, 1]$ is the fraction of initial points drawn randomly from the vertices of the initial set $\mathcal{R}_0$, $\texttt{fracDistVert} \in [0, 1]$ is the fraction of disturbance values drawn randomly from the vertices of the disturbance set $\mathcal{W}$, and $D \in \mathbb{N}^+$ is the number of segments for the piecewise constant disturbance signals $w(t)$ (see Sec. 7.4.1). Code examples that demonstrate how to use the function `simulateRandom` are provided in the directory */examples/terminalRegion/...* in the AROC toolbox.

7.5 Reference Trajectory

For motion primitive based controllers (see Sec. 2.1) the reference trajectory can either be provided by the user, or it is automatically computed by solving an optimal control problem

that aims to bring the system as close as possible to the desired goal state. In AROC, we consider reference trajectories that correspond to piecewise constant reference inputs, where the number of piecewise constant segments is identical to the number of time steps `Opts.N` for the controller (see Fig. 27).

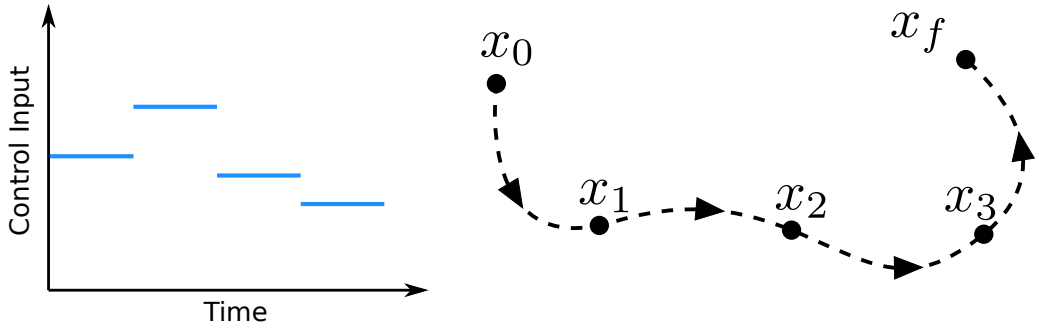


Figure 27: Illustration of a reference trajectory (right) that corresponds to a piecewise constant input signal (left) with `Opts.N` $= 4$ time steps.

A custom reference trajectory can be provided using the following settings for `Opts.refTraj` (see Sec. 2):

- `.x` matrix storing the states of the reference trajectory. The number of rows of the matrix has to be equal to the number of system states n, and the number of columns has to be equal to the number of time steps `Opts.N` plus one since the initial state has to be included. The final state has to be equal to `Params.xf`, and the initial state has to be equal to the center of `Params.R0` (see Sec. 2).
- `.u` matrix storing the inputs that correspond to the reference trajectory. The number of rows of the matrix has to be equal to the number of system inputs m, and the number of columns has to be equal to the number of time steps `Opts.N`. The input is constant during the period of one time step, and all inputs have to satisfy the input constraints.

If no custom reference trajectory is provided the reference trajectory is determined automatically by solving an optimal control problem (see (6)). To improve the result, one can provide custom weighting matrices Q and R by using the following settings for `Opts.refTraj` (see Sec. 2):

- `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for the cost function of the optimal control in (6). The default value is the identity matrix.
- `.R` input weighting matrix $R \in \mathbb{R}^{m \times m}$ for the cost function of the optimal control in (6). The default value is an all-zero matrix.

7.6 Extended Optimization Horizon

The convex interpolation control algorithm (see Sec. 2.1.2) and the generator space control algorithm (see Sec. 2.1.3) are based on optimal control problems (see (6)). In the classical set-up the objective for the optimal control problems is to drive the system states as close as possible to the next point of the reference trajectory (see Fig. 28 (top)). However, this can often be suboptimal since for many systems a certain deviation of some system states from the reference trajectory is required in order to reduce the deviations in other system states. For an autonomous car for example (see Sec. 6.6), a certain deviation in the orientation is required in order to reduce the deviation in the position. One way to solve this problem is to

use an extended optimization horizon, where the optimal control problem is solved for multiple reference trajectory time steps, but only the control inputs for the first time step are applied to the system (see Fig. 28 (bottom)).

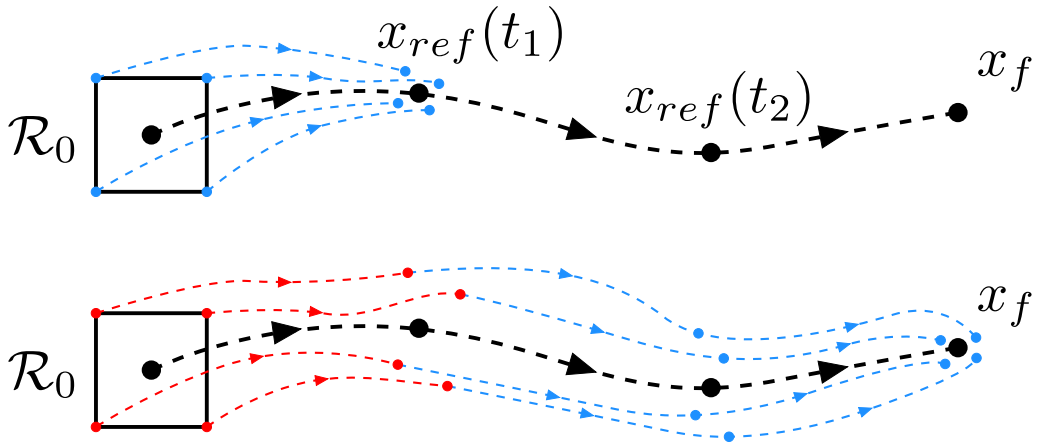


Figure 28: Illustration of the convex interpolation control algorithm with (bottom) and without (top) extended optimization horizon.

The optimal control problem with an extended optimization horizon is defined as follows:

$$\min_{u(t)} \left(\sum_{i=1}^{M} w(i) \cdot \big(x(t_i) - x_{ref}(t_i)\big)^T \cdot Q \cdot \big(x(t_i) - x_{ref}(t_i)\big) \right) + \int_{t=0}^{t_M} u(t)^T \cdot R \cdot u(t)\ dt \tag{18}$$

$$s.t.\ \ \dot{x}(t) = f(x(t), u(t), \mathbf{0}),$$

where $x_{ref}(t)$ is the reference trajectory, $w : \mathbb{N}^+ \to \mathbb{R}^+$ is a weighting function, $t_i = i\ t_f/N$, $M \in \mathbb{N}^+_{\leq N}$ is the length of the extended optimization horizon, and $N \in \mathbb{N}^+$ is the number of reference trajectory time steps.

The settings for an extended optimization horizon are provided with the struct `Opts.extHorizon`:

- `.active` flag specifying if an extended optimization horizon is used (`Opts.extHorizon.active` = 1) or not (`Opts.extHorizon.active` = 0). The default value is 0.
- `.horizon` length of the extended optimization horizon $M \in \mathbb{N}^+_{\leq N}$ in reference trajectory time steps (see (18)).
- `.decay` string specifying the type of weighting function $w(\cdot)$ (see (18)) that is used. The available types are `'uniform'`, `'end'`, `'fall'`, `'fall+End'`, `'fallLinear'`, `'fallLinear+End'`, `'fallEqDiff'`, `'fallEqDiff+End'`, `'rise'`, `'quad'`, `'riseLinear'`, and `'riseEqDiff'` (see (19) and Fig. 29).

The different types of weighting functions are defined as follows:

`'uniform'` : $w(i) = 1$

`'end'` : $w(i) = \begin{cases} 1, & i = M \\ 0, & \text{otherwise} \end{cases}$

`'fall'` : $w(i) = \frac{1}{i}$

$$
\begin{aligned}
&\texttt{'rise'}: && w(i) = \frac{1}{M+1-i} \\
&\texttt{'quad'}: && w(i) = \frac{\lfloor |i - \frac{M+1}{2}| \rfloor^2 + 1}{\max_{j=\{1,\dots,M\}} \lfloor |j - \frac{M+1}{2}| \rfloor^2 + 1} \\
&\texttt{'fallLinear'}: && w(i) = 1 - (i-1)\frac{1-\frac{1}{M}}{M-1} \\
&\texttt{'fallEqDiff'}: && w(i) = \begin{cases} \frac{1}{\sum_{j=2}^{M} w(j)}, & i = M \\ \frac{\sum_{j=i+1}^{M} w(j)}{\sum_{j=2}^{M} w(j)}, & \text{otherwise} \end{cases}
\end{aligned} \tag{19}
$$

For the weigthing functions `'fall+End'`, `'fallLinear+End'`, and `'fallEqDiff+End'` the last weight is equal to one ($w(M) = 1$). The weighting functions `'riseLinear'` and `'riseEqDiff'` are defined as the weighting functions `'fallLinear'` and `'fallEqDiff'`, but with increasing weights.

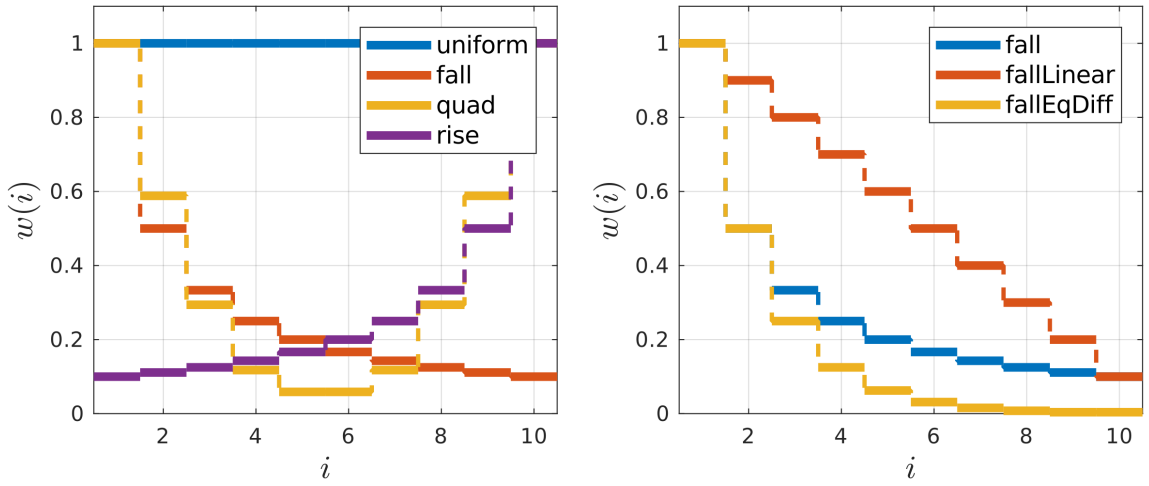


Figure 29: Visualization of the different types of weighting functions.

7.7 Reachability Settings

AROC uses the CORA toolbox [1] to compute reachable sets. The reachabiliy algorithms implemented in CORA require some user-defined settings like, e.g., maximum zonotope order, maximum tensor order, etc. [4]. In AROC, the settings for reachability analysis using CORA are provided with the struct `Opts.cora`, which has the following fields:

- `.alg` string specifying the reachability algorithm for nonlinear systems. The available algorithms are conservative linearization (`'lin'`) and conservative polynomialization (`'poly'`) (see [4, Sec. 4.2.5.1]).
- `.linAlg` string specifying the reachability algorithm for nonlinear systems. The available algorithms are `'standard'`, `'fromStart'`, `'wrapping-free'`, and `'adap'`. For optimization based control in combination with linear systems only (see [4, Sec. 4.2.1.1]).

- `.tensorOrder` order $\kappa \in \{2, 3\}$ of the Taylor series expansion that is used to obtain an abstraction of the nonlinear system dynamics. (see [4, Sec. 4.2.5.1])
- `.taylorTerms` number of Taylor series terms used to obtain an enclosure of the exponential matrix e^{At} (see [4, Sec. 4.2.1.1] and [4, Sec. 4.2.5.1]).
- `.zonotopeOrder` upper bound for the zonotope order of the zonotopes that represent the reachable set (see [4, Sec. 4.2.1.1] and [4, Sec. 4.2.5.1]).
- `.intermediateOrder` upper bound for the zonotope order during internal computations. For `Opts.cora.tensorOrder = 3` only (see [4, Sec. 4.2.5.1]).
- `.errorOrder` upper bound for the zonotope order before the abstraction error is computed. For `Opts.cora.tensorOrder = 3` only (see [4, Sec. 4.2.5.1]).
- `.error` upper bound for the Hausdorff-distance between the exact reachable set and the computed over-approximation. For `Opts.cora.linAlg = 'adap'` only (see [4, Sec. 4.2.1.1]).

If no reachability settings are specified, the default values listed in Tab. 22 are used.

Table 22: Default reachability settings for all control algorithms implemented in AROC.

	alg	linAlg	tensorOrder	taylorTerms	zonotopeOrder	intermediateOrder	errorOrder	error
opt. based control (lin. sys.)	-	'standard'	-	10	50	30	5	see [4]
opt. based control (nonlin. sys.)	'lin'	-	2	10	50	30	5	-
conv. int. (lin. contr.)	'lin'	-	2	20	100	50	5	-
conv. int. (exact + quad. contr.)	'poly'	-	3	20	100	50	30	-
generator space control	'lin'	-	2	20	30	20	5	-
polynomial control	'poly'	-	3	20	30	20	10	-
combined control	'lin'	'standard'	2	10	50	20	5	
safety net control	'poly'	-	3	20	30	20	5	-
reachset MPC	'lin'	-	2	10	5	3	3	-
linear system MPC	-	'standard'	-	10	50	-	-	-
terminal region subpaving	'lin'	-	2	10	50	50	5	-
terminal region zonotope approach	-	'standard'	-	4	150	-	-	-

7.8 Adding Custom Comfort Controllers

This section describes how to add custom comfort controllers for the safety net controller described in Sec. 2.1.6. The comfort controller for safety net control are implemented in the directory */algorithms/safetyNetControl/comfortController*. To add a new custom comfort controller one has to add the comfort controller as a new class named `objContrName` to this directory, where `Name` is the name of the controller (see setting `Opts.controller` in Sec. 2.1.6. For compatibility with the safety net control framework, the comfort controller class has to implement the following functions:

- **Class constructor:** Constructor of the custom controller class.

$$\texttt{obj} = \texttt{objContrName}(\texttt{benchmark}, \texttt{Opts}, \texttt{ContrOpts}).$$

- **Initialization:** Initialization of the comfort controller. This function is executed once prior to online application.

$$\texttt{init}(\texttt{obj}).$$

- **Reachability Analysis:** Computation of the reachable set of the comfort controller for one time step of the safety net controller:

$$[\texttt{res}, \mathcal{R}, \texttt{Param}] = \texttt{reachSet}(\texttt{obj}, \mathcal{R}_0, \texttt{iter}).$$

- **Prediction:** Computation of the reachable set at the end of the allocated computation time:

$$\mathcal{R} = \texttt{reachSetPred}(\texttt{obj}, x_0, \texttt{iter}, \texttt{Param}),$$

- **Simulation:** Simulation of the comfort controller for one time step of the safety net controller.

$$[\texttt{t}, \texttt{x}, \texttt{u}] = \texttt{simulate}(\texttt{obj}, x_0, w(t), v(t), \texttt{Param}, \texttt{iter}).$$

where `obj` is an object of the comfort controller class, `benchmark` is the name of the benchmark system (see Sec. 6), `Param` is a struct storing comfort controller parameters that are later required for simulation or prediction, `iter` is the current time step of the safety net controller, and `res` is a flag specifying if the comfort controller is safe (`res = 1`) or unsafe (`res = 0`). Furthermore, `Opts` is a struct containing the required data from the safety controller, and `ContrOpts` is a struct with comfort controller settings, which can for the safety net controller be specified with `Opts.contrOpts` (see Sec. 2.1.6). Moreover, $x_0 \in \mathbb{R}^n$ is the initial state, $\mathcal{R}_0 \subset \mathbb{R}^n$ is the initial set, $\mathcal{R} \in \mathbb{R}^n$ is the final reachable set, and $w(t) \in \mathbb{R}^{q \times D}$, $v(t) \in \mathbb{R}^{n \times D}$ are vectors of disturbances and measurement errors, where $D \in \mathbb{N}^+$ is the number of disturbance changes. Finally, $\texttt{t} \in \mathbb{R}^M$, $\texttt{x} \in \mathbb{R}^{M \times n}$, and $\texttt{u} \in \mathbb{R}^{M \times m}$ store the time, the states and the inputs for the simulated trajectory, with $M \in \mathbb{N}^+$ being the number of simulation time steps.

Currently, a *Linear Quadratic Regulator* (`Opts.controller = 'LQR'`) and a *Model Predictive Controller* (`Opts.controller = 'MPC'`) are implemented as comfort controllers in AROC. The settings for the LQR controller specified in the struct `Opts.contrOpts` (see Sec. 2.1.6) are as follows:

- `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for LQR control.
- `.R` input weighting matrix $R \in \mathbb{R}^{m \times m}$ for LQR control

The settings for the MPC controller specified in the struct `Opts.contrOpts` (see Sec. 2.1.6) are as follows:

- `.Q` state weighting matrix $Q \in \mathbb{R}^{n \times n}$ for LQR control.
- `.R` input weighting matrix $R \in \mathbb{R}^{m \times m}$ for LQR control.
- `.horizon` optimization horizon for MPC in safety net controller time steps.
- `.Ninter` number of piecewise constant control input segments during one time step of the safety net controller.

8 Examples

In this section we provide some code examples that demonstrate how to apply the control algorithms implemented in AROC. All code examples presented in this section as well as many additional examples can found in the directory */examples/...* in the AROC toolbox.

8.1 Example Motion Primitive Based Control

In this section we present a code example that demonstrates how to construct a feasible controller for the turn-right maneuver of the autonomous car benchmark (see Sec. 6.6) described in [8, Sec. 6] with the optimization based control algorithm (see Sec. 2.1.1). The generated plot is shown in Fig. 30, and the code for the example is implemented in the file */examples/optimizationBasedControl/example_optBasedContr_car.m* in the AROC toolbox.

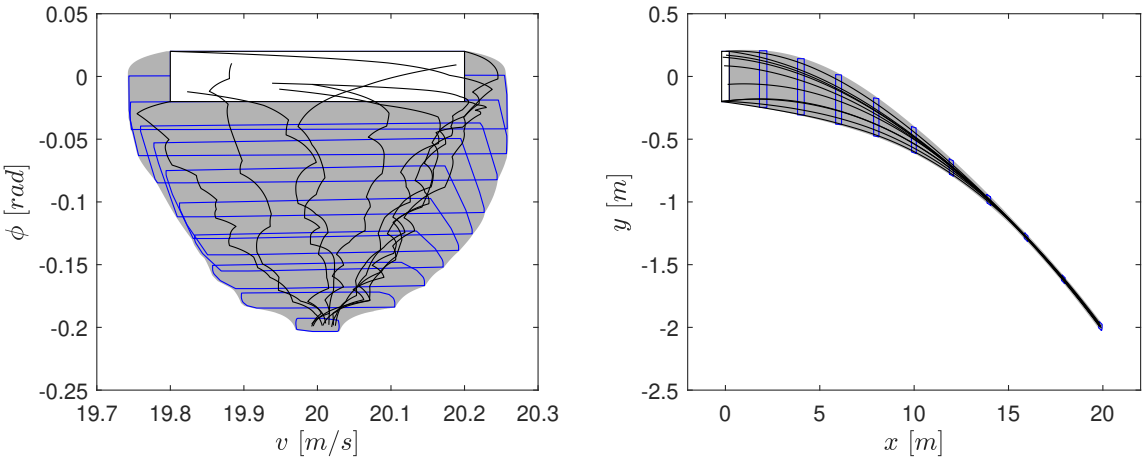


Figure 30: Plot generated by the optimization based control code example in Sec. 8.1, where the reachable set (gray) as well as simulated trajectories (black) of the controlled system are shown for different dimensions.

```
% Benchmark Parameter -------------------------------------------------------

% initial set
x0 = [20;0;0;0];
width = [0.2; 0.02; 0.2; 0.2];
Param.R0 = interval(x0-width,x0+width);

% goal state and final time
Param.xf = [20; -0.2; 19.87; -1.99];
Param.tFinal = 1;

% set of admissible control inputs
width = [9.81;0.4];
Param.U = interval(-width,width);

% set of uncertain disturbances
width = [0.5;0.02];
Param.W = interval(-width,width);


% Algorithm Settings ---------------------------------------------------------

% number of time steps
Opts.N = 10;
```

```
% number of reachability analysis time steps
Opts.reachSteps = 12;
Opts.reachStepsFin = 100;

% parameters for optimization
Opts.maxIter = 10;
Opts.bound = 10000;

% weighting matrices for reference trajectory
Opts.refTraj.Q = 10*eye(4);
Opts.refTraj.R = 1/10*eye(2);


% Control Algorithm ------------------------------------------------------

% construct controller for motion primitive
[objContr,res] = optimizationBasedControl('car',Param,Opts);

% simulation
res = simulateRandom(objContr);


% Visualization ----------------------------------------------------------

% visualization (velocity and orientation)
figure; hold on; box on;
plotReach(res,[1,2],[.7 .7 .7]);
plotReachTimePoint(res,[1,2],'b');
plot(Param.R0,[1,2],'FaceColor','w','EdgeColor','k');
plotSimulation(res,[1,2],'k');
xlabel('v [m/s]'); ylabel('\phi [rad]');

% visualization (position)
figure; hold on; box on;
plotReach(res,[3,4],[.7 .7 .7]);
plotReachTimePoint(res,[3,4],'b');
plot(Param.R0,[3,4],'FaceColor','w','EdgeColor','k');
plotSimulation(res,[3,4],'k');
xlabel('x [m]'); ylabel('y [m]');
```

8.2 Example Model Predictive Control

In this section we present a code example that demonstrates the reachset model predictive control algorithm (see Sec. 2.2) on the stirred tank reactor benchmark in Sec. 6.4 for the same initial set as considered in [15, Sec. IV]. The generated plot is shown in Fig. 31, and the code for the example is implemented in the file */examples/reachsetMPC/example_reachsetMPC_stirredTankReactor2.m* in the AROC toolbox.

```
% Benchmark Parameter ----------------------------------------------------

% initial state
Param.x0 = [-0.3;-30];

% goal state
Param.xf = [0;0];

% set of admissible control inputs
Param.U = interval(-20,70);

% set of uncertain disturbances
```

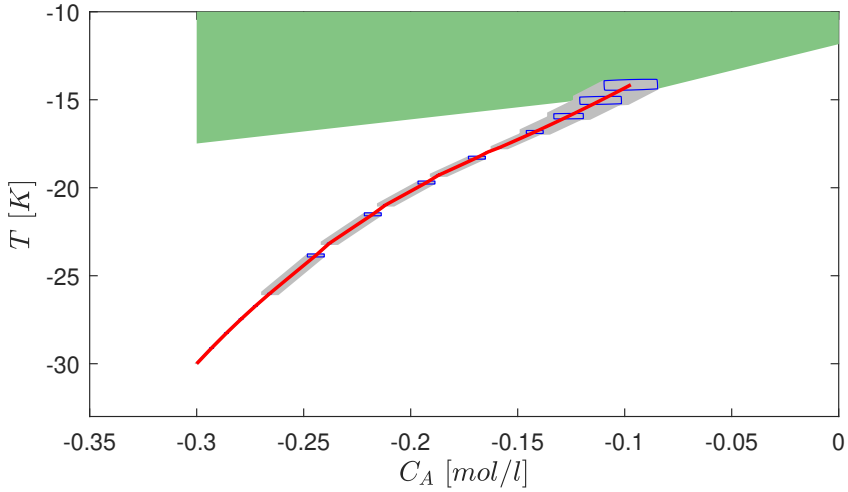


Figure 31: Plot generated by the reachset model predictive control code example in Sec. 8.2. The terminal region is visualized in green, the reachable set in gray, and the resulting trajectory of the controlled system in red.

```
width = [0.1;2];
Param.W = interval(-width,width);


% Algorithm Settings ------------------------------------------------------

% scaling factor for the tightend set of admissible control inputs
Opts.scale = 0.9556;

% number of time steps and optimization horizon
Opts.N = 5;
Opts.tOpt = 9;

% weighting matrices for the optmial control problem
Opts.Q = diag([100,1]);
Opts.R = 0.9;

% weigthing matrices for the tracking controller
Opts.Qlqr = diag([1;1]);
Opts.Rlqr = 100;

% terminal region
A = [-1.0000 0;1.0000 0;30.0000 -1.0000;66.6526 -4.8603;-66.6526 4.8603];
b = [0.3000;0.0620;11.8400;65.0000;15.0000];
Opts.termReg = mptPolytope(A,b);

% additional settings
Opts.tComp = 0.54;
Opts.alpha = 0.1;
Opts.maxIter = 50;
Opts.reachSteps = 1;


% Control Algorithm -------------------------------------------------------

% execute control algorithm
res = reachsetMPC('stirredTankReactor',Param,Opts);
```

```
% Visualization ------------------------------------------------------------

figure; hold on; box on
plot(Opts.termReg,[1,2],'FaceColor',[100 182 100]./255,...
     'EdgeColor','none','FaceAlpha',0.8);
plotReach(res,[1,2],[.75 .75 .75]);
plotReachTimePoint(res,[1,2],'b');
plotSimulation(res,[1,2],'r','LineWidth',1.5);
xlabel('C_A [mol/l]'); ylabel('T [K]');
xlim([-0.35,0]); ylim([-33,-10]);
```

8.3 Example Maneuver Automaton

In this section we present a code example that demonstrates how a maneuver automaton for the autonomous car benchmark in Sec. 6.6 can be constructed and applied online to solve a CommonRoad scenario (see Sec. 1.7). The generated plot is shown in Fig. 32, and the code for the example is implemented in the file */examples/maneuverAutomaton/example_maneuverAutomaton_car2.m* in the AROC toolbox.

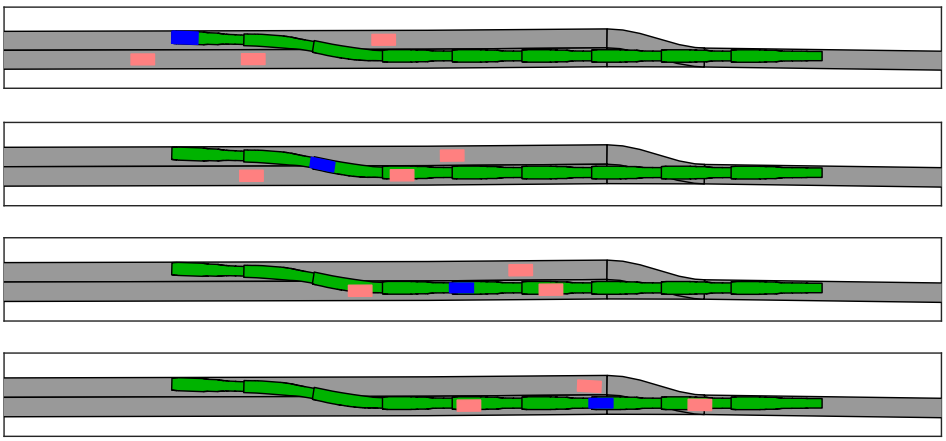


Figure 32: Plot generated by the maneuver automaton code example in Sec. 8.3, where the occupancy sets of the other vehicles (red) as well as the planned trajectory (blue) are visualized for at times $t = 0s$ (top), $t = 2s$ (upper middle), $t = 4s$ (lower middle), and $t = 6s$ (bottom).

```
% Generate Motion Primitives -----------------------------------------------

% load postprocessing function
Post = @postprocessing_car;

% load system parameter
Params = param_car();

% define algorithm options
Opts = [];

Opts.N = 5;                          % number of time steps
Opts.Ninter = 5;                     % number of intermediate time steps
Opts.extHorizon.active = 1;          % use extended optimization horizon
Opts.extHorizon.horizon = 5;         % time steps for ext. horizon
Opts.extHorizon.decay = 'fall';      % weight function for ext. horizon

% define control inputs and initial states for motion primitives
```

```
list_x0 = {[15.8773;0;0;0];[14.8773;0;0;0];[14.8773;0;0;0]};
list_u1 = {-1;0;0};
list_u2 = {0;-0.15:0.15:0;0.18};

% loop over all motion primitives
primitives = {};
counter = 1;

for i = 1:length(list_x0)

    % define ranges for inputs and get initial state
    [U1,U2] = meshgrid(list_u1{i},list_u2{i});

    % loop over the different control input combinations
    for j = 1:size(U1,1)
        for k = 1:size(U1,2)

            % get reference trajectory by simulating the system
            x0 = list_x0{i};
            u = [U1(j,k); U2(j,k)];
            tspan = 0:Params.tFinal/(Opts.N*Opts.Ninter):Params.tFinal;
            fun = @(t,x) car(x,u,zeros(4,1));

            % get reference trajectory by simulating the system
            [t,x] = ode45(fun,tspan,x0);

            % provide reference trajectory as an additional input argument
            Opts.refTraj.x = x';
            Opts.refTraj.u = u*ones(1,size(x,1)-1);

            % update parameter
            Params.xf = x(end,:)';
            Params.R0 = Params.R0 + (-center(Params.R0)) + x0;

            % compute controller for the current motion primitive
            objContr = generatorSpaceControl('car',Params,Opts,Post);

            primitives{counter} = objContr;
            counter = counter + 1;
        end
    end
end


% Construct Maneuver Automaton ------------------------------------------

% assemble input arguments
shiftFun = @shiftInitSet_car;
shiftOccFun = @shiftOccupancySet_car;

% construct maneuver automaton
MA = maneuverAutomaton(primitives,shiftFun,shiftOccFun);


% Online Control ---------------------------------------------------------

% load a CommonRoad traffic scenario
scenario = 'ZAM_Zip-1_19_T-1';
[statObs,dynObs,x0,goalSet,lanelets] = commonroad2cora(scenario);
x0 = [x0.velocity; x0.orientation; x0.x; x0.y];

% plan a verified trajectory with the maneuver automaton
ind = motionPlanner(MA,x0,goalSet{1},statObs,dynObs,'Astar',@costFun);
```

```
% Visualization ----------------------------------------------------------

% visualize the planned trajectory for different times
figure
times = {0,2,4,6};

for i = 1:length(times)
   subplot(length(times),1,i); hold on; box on;
   for j = 1:length(lanelets)
        plot(lanelets{j},[1,2],'FaceColor',[.6 .6 .6],'EdgeColor','k');
   end
   plotPlannedTrajectory(MA,ind,x0,[],[0 0.7 0],'EdgeColor','k');
   plotPlannedTrajectory(MA,ind,x0,interval(times{i}),'b');
   plotObstacles([],dynObs,interval(times{i}-0.05,times{i}+0.05));
   xlim([-150,50]); ylim([0,15]); xticks([]); yticks([]);
end


% Auxiliary Functions -----------------------------------------------------

function cost = costFun(obj,node,goalSet)
% compute the costs for A* star search for the current node

    % compute final state and final time at the end of the motion primitive
    index = node.ind(end);
    occSet = updateOccupancy(obj,node.parent.xf,index,node.parent.time);
    xCurr = center(occSet{end}.set);
    time = node.parent.time + obj.primitives{index}.tFinal;

    % compute estimated remaining time for reaching the goal set
    c = center(goalSet.set);
    dist = sqrt(sum(c-xCurr).^2);
    v = node.parent.xf(1);
    h = dist/v;
    g = time;

    % compute costs for the node
    cost = h + g;
end
```

8.4 Example Terminal Region

In this section we present a code example that demonstrate how to construct a safe terminal region for the double integrator benchmark (see Sec. 6.1) using the zonotope approach for linear systems in Sec. 4.2. The generated plot is shown in Fig. 33, and the code for the example is implemented in the file */examples/terminalRegion/example_termReg_zonoLinSys_doubleIntegrator.m* in the AROC toolbox.

```
% Benchmark Parameter -----------------------------------------------------

% set of admissible control inputs
Param.U = interval(-1,1);

% set of uncertain disturbances
width = [0.1; 0.1];
Param.W = interval(-width,width);


% Algorithm Settings ------------------------------------------------------
```

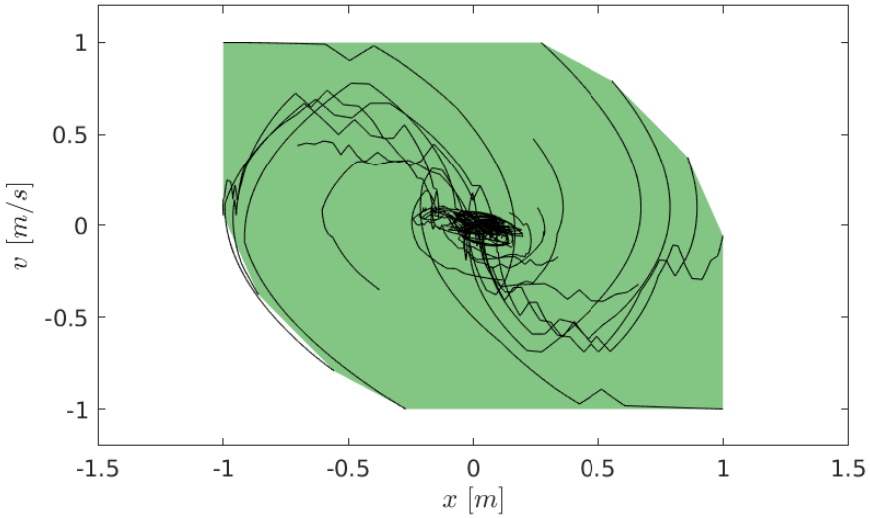


Figure 33: Terminal region (green) for the double integrator benchmark together with simulations (black) of the terminal controller.

```
% search domain
Opts.Tdomain = interval(-ones(2,1),ones(2,1));

% number of time steps and time step size
Opts.N = 30;
Opts.timeStep = 0.1;

% weighting matrices for the LQR controller
Opts.Q = eye(2);
Opts.R = eye(1);

% parameters for optimization
Opts.genMethod = 'sampling2D';
Opts.costFun = 'sum';


% Terminal Region ---------------------------------------------------------

% construct terminal region
T = computeTerminalRegion('doubleIntegrator','zonoLinSys',Param,Opts);

% simulate the terminal region controller
tFinal = 10;
res = simulateRandom(T,tFinal);


% Visualization -----------------------------------------------------------

figure; hold on; box on;
plot(T.set,[1,2],'FaceColor',[100 182 100]./255,...
     'EdgeColor','none','FaceAlpha',0.8);
plotSimulation(res,[1,2],'k');
xlim([-1.5;1.5]); ylim([-1.2,1.2]);
xlabel('x [m]'); ylabel('v [m/s]');
```

8.5 Example Conformant Synthesis

Here we present a code example that demonstrates how conformant synthesis as described in Sec. 5 can be used to construct an over-approximative model from measurements of the real system:

```
% load measurements
load('measurements_car');

% algorithm settings
Opts.group = 6;              % number of measurements for each opt. problem
Opts.measErr = true;         % represent uncertainty using measurement error
Opts.set = 'interval';       % set representation for uncertainty sets

% conformant synthesis
[W,V] = conformantSynthesis('car',M,Opts)
```

This code produces the following command line output:

```
W =

  [-0.0055, 0.0039]
  [-0.0013, 0.0004]

V =

  [-0.0919, 0.0686]
  [-0.0031, 0.0049]
  [-0.0397, 0.0274]
  [-0.0052, 0.0368]
```

Acknowledgments

This toolbox is partially supported by the European Commission under the project UnCoVerCPS (grant number 643921) and by the German Research Foundation (DFG) project faveAC (grant number AL 1185/5-1). Furthermore, we would like to thank the students Jinyue Guan, Hana Mekic, Jan Wagener, and Ivan Hernandez who contributed to this toolbox as part of their practical course projects.

References

[1] M. Althoff, "An introduction to CORA 2015," in *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015, pp. 120–151.

[2] M. Althoff, M. Koschi, and S. Manzinger, "CommonRoad: Composable benchmarks for motion planning on roads," in *Proc. of the Intelligent Vehicles Symposium*, 2017, pp. 719–726.

[3] H. Krasowski and M. Althoff, "CommonOcean: Composable benchmarks for motion planning on oceans," in *Proc. of the International Conference on Intelligent Transportation Systems*, 2022, pp. 1676–1682.

[4] M. Althoff, N. Kochdumper, and M. Wetzlinger. (2022) CORA 2022 manual. [Online]. Available: https://tumcps.github.io/CORA/data/Cora2022Manual.pdf

[5] R. Sargent, "Optimal control," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1, pp. 361 – 371, 2000.

[6] B. Schürmann and M. Althoff, "Optimal control of sets of solutions to formally guarantee constraints of disturbed linear systems," in *Proc. of the American Control Conference*, 2017, pp. 2522–2529.

[7] H. Kwakernaak and R. Sivan, *Linear optimal control systems*. Wiley, 1972.

[8] B. Schürmann and M. Althoff, "Convex interpolation control with formal guarantees for disturbed and constrained nonlinear systems," in *Proc. of the International Conference on Hybrid Systems: Computation and Control*, 2017, pp. 121–130.

[9] ——, "Guaranteeing constraints of disturbed nonlinear systems using set-based optimal control in generator space," in *Proc. of the World Congress of the International Federation of Automatic Control*, 2017, pp. 11 515–11 522.

[10] V. Gaßmann and M. Althoff, "Verified polynomial controller synthesis for disturbed nonlinear systems," in *Proc. of the International Conference on Analysis and Design of Hybrid Systems*, 2021, pp. 85–90.

[11] N. Kochdumper, B. Schürmann, and M. Althoff, "Utilizing dependencies to obtain subsets of reachable sets," in *Proc. of the International Conference on Hybrid Systems: Computation and Control*, 2020, article 1.

[12] N. Kochdumper and M. Althoff, "Sparse polynomial zonotopes: A novel set representation for reachability analysis," *Transactions on Automatic Control*, vol. 66, no. 9, pp. 4043–4058, 2020.

[13] B. Schürmann and M. Althoff, "Optimizing sets of solutions for controlling constrained nonlinear systems," *Transactions on Automatic Control*, vol. 66, no. 3, pp. 981–994, 2020.

[14] B. Schürmann, M. Klischat, N. Kochdumper, and M. Althoff, "Formal safety net control using backward reachability analysis," *Transactions on Automatic Control*, vol. 67, no. 11, pp. 5698–5713, 2021.

[15] B. Schürmann, N. Kochdumper, and M. Althoff, "Reachset model predictive control for disturbed nonlinear systems," in *Proc. of the International Conference on Decision and Control*, 2018, pp. 3463–3470.

[16] F. Gruber and M. Althoff, "Scalable robust model predictive control for linear sampled-data systems," in *Proc. of the International Conference on Decision and Control*, 2019.

[17] A. El-Guindy, D. Han, and M. Althoff, "Estimating the region of attraction via forward reachable sets," in *Proc. of the American Control Conference*, 2017, pp. 1263–1270.

[18] L. Jaulin, M. Kieffer, and O. Didrit, *Applied Interval Analysis*. Springer, 2006.

[19] F. Gruber and M. Althoff, "Computing safe sets of linear sampled-data systems," *IEEE Control Systems Letters*, vol. 5, no. 2, pp. 385–390, 2020.

[20] N. Kochdumper, F. Gruber, B. Schürmann, V. Gaßmann, M. Klischat, and M. Althoff, "AROC: A toolbox for automated reachset optimal controller synthesis," in *Proc. of the International Conference on Hybrid Systems: Computation and Control*, 2021.

[21] S. Geva and J. Sitte, "A cartpole experiment benchmark for trainable controllers," *IEEE Control Systems Magazine*, vol. 13, no. 5, pp. 40–51, 1993.

[22] L. Magni, G. D. Nicolao, L. Magnani, and R. Scattolini, "A stabilizing model-based predictive control algorithm for nonlinear systems," *Automatica*, vol. 37, no. 9, pp. 1351–1362, 2001.

[23] S. Yu and et. al, "Tube mpc scheme based on robust control invariant set with application to lipschitz nonlinear systems," *Systems & Control Letters*, vol. 62, no. 2, pp. 194–200, 2013.

[24] M. Althoff and G. Würsching. (2020) CommonRoad: Vehicle Models (Version 2020a). [Online]. Available: https://gitlab.lrz.de/tum-cps/commonroad-vehicle-models/-/blob/master/vehicleModels_commonRoad.pdf

[25] Z. Yuan and et al., "safe-control-gym: A unified benchmark suite for safe learning-based control and reinforcement learning in robotics," *Robotics and Automation Letters*, vol. 7, no. 4, pp. 11 142–11 149, 2022.

[26] H. Krasowski and M. Althoff. (2022) CommonOcean: Vessel Models (Version 2022a). [Online]. Available: https://gitlab.lrz.de/tum-cps/commonocean-vessel-models/-/blob/main/documentation/vesselModels_commonOcean.pdf

[27] E. Ivanjko, T. Petrinic, and I. Petrovic, "Modelling of mobile robot dynamics," in *Proc. of the EUROSIM Congress on Modelling and Simulation*, 2010.