Chair of
Automatic Control

FAU

GRAMPC-S documentation

Daniel Landgraf, Andreas Völz, Knut Graichen

Chair of Automatic Control
Faculty of Engineering
Friedrich-Alexander-Universität Erlangen-Nürnberg

August 5, 2025

Contents

1 Introduction

This manual describes the structure and the application of GRAMPC-S, a framework for stochastic model predictive control of nonlinear continuous-time systems subject to nonlinear probabilistic constraints. It is based on the toolbox GRAMPC [1], which solves an optimization problem using the augmented Lagrangian approach and a tailored gradient method. GRAMPC-S is implemented as C++ code which allows implementation on embedded hardware and can be used in Simulink and dSPACE.

The documentation is outlined as follows. Section 2 describes the installation of GRAMPC-S for Windows and Linux. Section 3 describes the class of optimal control problems (OCP) that are considered. Section 4 shows the implementation of an OCP as C++ code. Several methods are implemented in GRAMPC-S in order to solve the OCP and propagate uncertainties, whose application is described in Section 5. For a more detailed explanation of these methods, please refer to [2] and [3]. Section 6 describes the realization of closed-loop simulations and the plotting of results in Matlab. Section 7 concludes with an example of the application and parameterization of GRAMPC-S.

2 Installation

This chapter describes the installation of GRAMPC-S for Windows and Linux. GRAMPC-S depends on the Eigen library in version 3.4.90 or higher (tested in version 3.4.90) [4]. If Eigen is not already installed on your system, clone the repository, create the folder `build`, and install Eigen. For **Linux**, execute the following commands:

```
git clone https://gitlab.com/libeigen/eigen.git
mkdir eigen/build
cd eigen/build
cmake -DCMAKE_BUILD_TYPE=Release ..
sudo make install
```

For Windows with MinGW compiler, execute the following commands:

```
git clone https://gitlab.com/libeigen/eigen.git
mkdir eigen\build
cd eigen\build
cmake -DCMAKE_BUILD_TYPE=Release ..  -G "MinGW Makefiles"
cmake --install .
```

After Eigen has been installed, GRAMPC-S can be built. The source code of GRAMPC-S is available at `https://github.com/grampc/grampc-s`. Clone the code including all submodules:

```
git clone --recurse-submodules https://github.com/grampc/grampc-s.git
```

After cloning the repository, a new directory with the following subfolders is created:

- `documentation`: The PDF-file of the GRAMPC-S documentation can be found here.
- `examples`: Contains MPC-examples and a template for implementing your own code.
- `include`: The header files of GRAMPC-S are located in this folder.
- `libs`: Includes external libraries that are linked as git submodules.
- `matlab`: This folder contains Matlab files.
- `src`: The source code of GRAMPC-S can be found here.

As soon as the code has been downloaded, it can be built. To build GRAMPC-S using Linux, run the CMakeLists.txt and call make afterwards:

```
cmake -DCMAKE_BUILD_TYPE=Release CMakeLists.txt
make
```

To build GRAMPC-S using Windows, the editor Visual Studio Code with the C++ extension is recommended. Open the folder `grampc-s` in Visual Studio Code, click the button `Build`, and select a compiler from the list. After building the code, the folders `bin` and `build` are created in addition to the folders above. The folder `bin` contains the executable files for the examples.

3 System class

GRAMPC-S considers optimal control problems of the form

$$\min_{\boldsymbol{u},T} \quad J(\boldsymbol{u};\, \boldsymbol{p},\, \boldsymbol{x}_0) = \mathbb{E}\left[V(\boldsymbol{x}(T),\, \boldsymbol{p},\, T) + \int_0^T l(\boldsymbol{x},\, \boldsymbol{u},\, \boldsymbol{p},\, t)\,\mathrm{d}t\right] \tag{3.1a}$$

$$\text{s.t.} \quad \mathrm{d}\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x},\, \boldsymbol{u},\, \boldsymbol{p},\, t)\,\mathrm{d}t + \boldsymbol{g}(\boldsymbol{x},\, \boldsymbol{u})\,\mathrm{d}t + \boldsymbol{\Sigma}\,\mathrm{d}\boldsymbol{w}\,, \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \tag{3.1b}$$

$$\mathbb{P}\left[h_i(\boldsymbol{x},\, \boldsymbol{u},\, \boldsymbol{p},\, t) \leq \boldsymbol{0}\right] \geq \alpha_i\,, \qquad i \in [1,\, n_h] \tag{3.1c}$$

$$\mathbb{P}\left[h_{T,j}(\boldsymbol{x}(T),\, \boldsymbol{p},\, T) \leq \boldsymbol{0}\right] \geq \alpha_{T,j}\,, \qquad j \in [1,\, n_{h_T}] \tag{3.1d}$$

$$\boldsymbol{u} \in [\boldsymbol{u}_{min},\, \boldsymbol{u}_{max}] \tag{3.1e}$$

$$T \in [T_{min},\, T_{max}] \tag{3.1f}$$

in the context of model predictive control. The functional of the expected cost J in (3.1a) consists of the expected value of terminal cost V and integral cost l, which depend on states $\boldsymbol{x} \in \mathbb{R}^{n_x}$, control inputs $\boldsymbol{u} \in \mathbb{R}^{n_u}$, parameters $\boldsymbol{p} \in \mathbb{R}^{n_p}$, and the prediction horizon T. States and parameters are assumed to be random variables.

The cost functional (3.1a) is minimized subject to the system dynamics (3.1b), probabilistic constraints (3.1c), probabilistic terminal constraints (3.1d), and box constraints (3.1e) and (3.1f) for $\boldsymbol{u}$ and T, respectively. The system dynamics are given in the form of a stochastic differential equation in which $\boldsymbol{f}$ describes the part that is known, $\boldsymbol{g}$ is a Gaussian process approximation of unknown parts of the system dynamics, and $\boldsymbol{w}$ is a multidimensional Wiener process with constant diffusion term $\boldsymbol{\Sigma}$. The Gaussian process approximation and the Wiener noise are optional components. If these are not specified, they are not taken into account in order to reduce the computation time. Based on the system dynamics, the states are predicted starting from the initial state $\boldsymbol{x}_0$. The vectors $\boldsymbol{h} \in \mathbb{R}^{n_h}$ in (3.1c) and $\boldsymbol{h}_T \in \mathbb{R}^{n_{h_T}}$ in (3.1d) denote inequality constraints and terminal inequality constraints that must be satisfied with probabilities $\boldsymbol{\alpha} \in \mathbb{R}^{n_h}$ and $\boldsymbol{\alpha}_T \in \mathbb{R}^{n_{h_T}}$, respectively.

4 Implementation of the optimal control problem

The optimization problem (3.1) must be implemented as C++ code in order to be used in GRAMPC-S. All functions that appear in (3.1) must be defined in a problem description class, which is described in Section 4.1. In addition, random variables such as initial states and parameters must be defined in each time step and, optionally, the Gaussian process $\boldsymbol{g}$ must be specified, which is described in Section 4.2.

4.1 Implementation of the problem description

The structure of the OCP (3.1) consisting of the system dynamics, costs and constraints must be implemented as methods of a class that inherits from the abstract class `ProblemDescription`. The `examples` folder contains implementations of several examples as well as the template `problem_description_TEMPLATE.hpp` for the header file and the template `problem_description_TEMPLATE.cpp` for the source code. The following functions must be implemented:

- `ocp_dim`: Definition of the dimensions of the OCP (3.1), such as the number of states, the number of control inputs, the number of constraints, etc.
- `ffct`: The system dynamics $\boldsymbol{f}$ in (3.1b).
- `dfdx_vec`: The matrix vector product $\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)^{\mathsf{T}} \boldsymbol{v}$ of the Jacobian $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$ with an arbitrary vector $\boldsymbol{v}$.

Note that by using the matrix product of Jacobian matrix and vector, excessive multiplications with zero should be avoided. Depending on the OCP, further optional parts of (3.1) can be implemented:

- `dfdu_vec`: The matrix product $\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right)^{\mathsf{T}} \boldsymbol{v}$ of the Jacobian $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}$ with an arbitrary vector $\boldsymbol{v}$.
- `lfct`, `Vfct`: The integral cost l and the terminal cost V in (3.1a).
- `dldx`, `dldu`: The gradient of the integral cost with respect to $\boldsymbol{x}$ and $\boldsymbol{u}$.
- `dVdx`, `dVdT`: The gradient of the terminal cost with respect to $\boldsymbol{x}$ and T.
- `hfct`, `hTfct`: The inequality constraints $\boldsymbol{h}$ and $\boldsymbol{h}_T$ in (3.1c) and (3.1d).

- `dhdx_vec`, `dhdu_vec`, `dhdp_vec`: The matrix vector products $\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\right)^\mathsf{T} \boldsymbol{v}$, $\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{u}}\right)^\mathsf{T} \boldsymbol{v}$, and $\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{p}}\right)^\mathsf{T} \boldsymbol{v}$ of the Jacobians with an arbitrary vector $\boldsymbol{v}$.
- `dhTdx_vec`, `dhTdp_vec`, `dhTdT_vec`: The matrix vector products $\left(\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{x}}\right)^\mathsf{T} \boldsymbol{v}$, $\left(\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{T}}\right)^\mathsf{T} \boldsymbol{v}$, and $\left(\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{p}}\right)^\mathsf{T} \boldsymbol{v}$ of the Jacobians with an arbitrary vector $\boldsymbol{v}$.

The above functions are sufficient for most of the propagation methods in GRAMPC-S. However, if you want to use the Taylor series expansion around the current state in order to propagate uncertainties (see Section 5 for details), the following functions must also be implemented:

- `dfdx`, `dfdp`: Jacobians $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$ and $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}}$.
- `dfdxdx`, `dfdxdp`, `dfdxdu`, `dfdpdu`: Hessian matrices $\frac{\partial^2 \boldsymbol{f}}{\partial^2 \boldsymbol{x}}$, $\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x} \partial \boldsymbol{p}}$, $\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x} \partial \boldsymbol{u}}$, and $\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{p} \partial \boldsymbol{u}}$.
- `dhdx`, `dhdu`: Jacobians $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ and $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{u}}$.
- `dhdxdx`, `dhdxdu`: Hessian matrices $\frac{\partial^2 \boldsymbol{h}}{\partial^2 \boldsymbol{x}}$ and $\frac{\partial^2 \boldsymbol{h}}{\partial \boldsymbol{x} \partial \boldsymbol{u}}$.
- `dhTdx`: Jacobian matrix $\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{x}}$.
- `dhTdxdx`, `dhTdxdT`: Hessian matrices $\frac{\partial^2 \boldsymbol{h}_T}{\partial^2 \boldsymbol{x}}$ and $\frac{\partial^2 \boldsymbol{h}_T}{\partial \boldsymbol{x} \partial T}$.

The problem description is usually implemented in a .cpp file. To ensure that this is taken into account in the compilation process, it must be listed in the `CMakeList.txt` file. The folder `examples` contains several examples that can be used as references for problem descriptions.

4.2 Implementation of random variables and Gaussian processes

The initial states $\boldsymbol{x}_0$ are generally random variables, with a given probability density function. Appendix A.1 lists all probability density functions that are implemented in GRAMPC-S. However, it is possible to write your own probability density function as a class, which must be declared as derived from the `Distribution` class. The following example presents the definition of a Gaussian distribution with mean $[1, 0]^\mathsf{T}$ and covariance matrix $10^{-4} \cdot \boldsymbol{I}$, where $\boldsymbol{I}$ is the identity matrix:

```
Vector x0(2);
x0 << 1.0, 0.0;
Matrix P0 = 1e-4 * Matrix::Identity(2, 2);
DistributionPtr state = Gaussian(x0, P0);
```

The resulting object is a shared pointer to the Gaussian distribution and can be used subsequently. In the closed-loop setting, the distribution must be updated in each time step. The `Distribution` class provides the member function `setMeanAndCovariance` for this purpose:

```
Vector x1(2);
x1 << 1.5, -1.0;
Matrix P1 = 1e-2 * Matrix::Identity(2, 2);
state->setMeanAndCovariance(x1, P1);
```

GRAMPC-S is also able to consider a Gaussian process that takes non-modeled parts of the system dynamics into account. A Gaussian process is a non-parametric data-based model, see [5] for details. First, the kernel and the hyperparameters of the Gaussian process must be defined, selecting a kernel from the list in Appendix A.2. It is possible to write your own kernel by writing a class that inherits from the `StationaryKernel` class. For example, a squared exponential kernel for a two-dimensional input variable with output variance $\sigma^2 = 1$ and length scaling $\boldsymbol{l} = [2,\ 3]^\mathrm{T}$ can be defined as follows:

```
StationaryKernelPtr kernel = SquaredExponential(2, 1, {2, 3});
```

Furthermore, the data points must be passed to the Gaussian process. There are two options for this: Using the structure `GaussianProcessData`, which is passed to the Gaussian process, or by reading in a text document in which the input data is stored. The `matlab` folder contains the Matlab function `writeGaussianProcessData.m`, which can be used to easily create such a text file. The folder `examples/double_integrator_GP` contains an example of a double integrator whose system dynamics are completely represented by two Gaussian processes. The Matlab script `double_integrator_data.m` in this folder shows an example how data can be generated and how the function `writeGaussianProcessData.m` should be called. Once the text document has been generated, a Gaussian process can be created for example with

```
GaussianProcessPtr gp = GP("GP.txt", kernel, {false, true}, {true});
```

Here, `GP.txt` is the name of the text document, `kernel` is the pointer to the previously created kernel and the following vectors of type boolean describe the dependency of the Gaussian process on the states and the control input. In this example, the Gaussian process depends on the second state and the control input.

5 Approximation of the stochastic OCP

The OCP (3.1) cannot be solved with common solvers, because the calculation of the probabilistic constraints requires the exact knowledge of the probability density functions of the predicted states. However, the exact propagation of the probability distributions for nonlinear functions can only be calculated by solving a partial differential equation. Therefore, several approximate methods of uncertainty propagation are implemented in GRAMPC-S. Their application is explained in Section 5.2. Since these only allow an evaluation of a finite number of stochastic moments, the probabilistic constraints (3.1c) and (3.1d) must be approximated, which is described in Section 5.1.

5.1 Approximation of probabilistic constraints

For the computationally efficient solution of the OCP (3.1), the probabilistic constraints must be converted into deterministic form. One possibility for this is the Chebyshev inequality, which specifies an upper limit for the probability of a deviation of a random variable from its mean based on the variance of the random variable. The class `ChebyshevConstraintApproximation` implements the approximation of a constraint with the Chebyshev inequality. An object of this class can be created using the constructor

```
ChebyshevConstraintApproximation(const Vector& probabilities)
```

where `probabilities` describes the vector of probabilities $\boldsymbol{\alpha}$ in (3.1c). If, in addition to the constraints (3.1c), terminal constraints (3.1d) are to be taken into account, the concatenation of the vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}_T$ is passed to the constructor. The Chebyshev inequality is valid for any probability density distribution with finite variance, so it is correspondingly conservative. A less conservative approximation of the probabilistic constraints can be realized if additional information about the distributions of the constraints is available. The approximation `Symmetric-ConstraintApproximation` assumes that the probability density distribution of the constraints is symmetric. The approximation can be even less conservative if it is assumed that the constraint distribution is Gaussian. This type of approximation is implemented in GRAMPC-S in the class `GaussianConstraintApproximation`. Objects of these classes can be created with

```
SymmetricConstraintApproximation(const Vector& probabilities)
GaussianConstraintApproximation(const Vector& probabilities)
```

5.2 Stochastic problem descriptions and uncertainty propagation

The system dynamics (3.1b) are given in the form of a stochastic nonlinear differential equation. Common MPC solvers such as GRAMPC are unable to predict the probability distributions of the states for this class of differential equations. GRAMPC-S therefore implements several approximate methods of uncertainty propagation that transform the stochastic differential equation into an ordinary differential equation. These can be divided into two categories: Approximations of the nonlinear function $\boldsymbol{f}$ in (3.1b) and approximations of the stochastic moments by point-based representations. The linearization of the system function is explained in Section 5.2.1, the propagation of uncertainties based on sampling points is explained in Section 5.2.2.

5.2.1 First-order Taylor series approximation

The consideration of nonlinear functions $\boldsymbol{f}$ in (3.1b) complicates the calculation of future state distributions. This can be easily avoided by linearizing the system dynamics $\boldsymbol{f}$ around the current state and calculating the uncertainty propagation with the linearized dynamics. GRAMPC-S provides the class `TaylorProblemDescription` for this purpose. The constructor

```
TaylorProblemDescription(ProblemDescriptionPtr problemDescription,
    ChanceConstraintApproximationConstPtr constraintApproximation,
    MatrixConstRef diffMatrixWienerProcess)
```

receives a pointer `problemDescription` to the problem description that implements the system dynamics, the cost function and the constraints. Since the linearization of the system dynamics is based on the gradients of the system dynamics $\boldsymbol{f}$ and GRAMPC in turn solves the optimization problem based on gradients, second-order derivatives are required in addition to the gradients of the functions, as described in Section 4.1. Optionally, the type of constraint approximation and the diffusion matrix of the Wiener process $\boldsymbol{\Sigma}$ are passed by the arguments `constraintApproximation` and `diffMatrixWienerProcess`, respectively. Gaussian processes are not yet implemented in the class `TaylorProblemDescription`.

5.2.2 Point-based uncertainty propagation

A further way to propagate uncertainties is to approximate probability density distributions by sampling. The nonlinear function $\boldsymbol{f}$ can be evaluated for a set of sampling points and the stochastic moments of the propagated state distribution can be calculated from the results. This requires two things in GRAMPC-S: A transformation between stochastic moments and sampling points and a stochastic problem description that utilizes this transformation to approximately solve the OCP (3.1).

The following transformations between stochastic moments and points are implemented in GRAMPC-S, see [2] for details:

- Unscented transformation: This is implemented in the class `UnscentedTransformation` and requires the scaling parameters α, β, and κ.
- First-order Stirling's interpolation: The class `StirlingInterpolationFirstOrder` implements the first-order Stirling's interpolation for an adjustable step size.
- Second-order Stirling's interpolation: The Stirling's interpolation of second order can be realized with the class `StirlingInterpolationSecondOrder`, where the step size can be specified as well.
- Gaussian quadrature: The class `ComposedQuadrature` generates quadrature points and associated weights for the multivariate Gaussian quadrature by composing it from the quadrature points of one-dimensional Gaussian quadratures. A family of orthogonal polynomials must be known for the respective one-dimensional distribution and the corresponding one-dimensional quadrature rule must be implemented. GRAMPC-S has implementations of the Gauss-Hermite quadrature for Gaussian distributions and the Gauss-Legendre quadrature for uniform distributions. However, it is possible to extend GRAMPC-S with your own quadrature rules by writing a class that inherits from the interface `QuadratureRule`. In addition to the vector of families of orthogonal polynomials, the constructor receives a vector with the orders of the quadrature rules. If a scalar is passed instead of a vector, this order is used for all quadrature rules.
- Polynomial chaos expansion (PCE): The class `PCE_Transformation` contains an implementation of polynomial chaos expansion, which solves the corresponding integrals by Gaussian quadrature and is therefore considered to be a point transformation as well. Its constructor takes the same arguments as the class `ComposedQuadrature` and also requires the specification of the maximum polynomial order.
- Monte Carlo sampling: A stochastic sampling of a probability density distribution is implemented in the class `MonteCarloTransformation`. The number of sampling points must be defined and a random number generator must be passed to the constructor.

It is possible to implement your own point transformations and integrate them into GRAMPC-S. These must be derived classes of the abstract class `PointTransformation`. It should be noted that in addition to the calculation of the stochastic moments and points, the associated gradients must also be implemented, because the OCP is solved with a gradient-based approach. Some of the above point transformations have a second constructor that receives an argument `considerUncertain`. This is a vector of boolean values that sets for each variable whether it should be assumed to be uncertain or not. If the respective value is set to `false`, it is considered as a deterministic variable with the value of the mean of the distribution. Uncertainties whose impact is negligible can be ignored in this way, which reduces the number of points and thus the required computation time of the controller. The same can be achieved for Gaussian quadrature and PCE by setting the corresponding entry in vector `quadratureOrder` to 1.

The above transformations can be used to represent probability density distributions by points and to calculate stochastic moments from transformed points. These transformations are used in several problem descriptions that convert the stochastic OCP into a deterministic OCP. The problem descriptions differ in particular in the internal representation of the states and in whether a Wiener process or a Gaussian process can be taken into account. An available implementation is the `SigmaPointProblemDescription`, which initially represents the random variables by points and predicts these points up to the prediction horizon. The implementation of the system dynamics, the cost function and the constraints are passed to the constructor

```
SigmaPointProblemDescription(ProblemDescriptionPtr problemDescription,
    ChanceConstraintApproximationConstPtr constraintApproximation,
    PointTransformationPtr pointTransformation)
```

as a pointer to a problem description. In addition, the type of constraint approximation and the transformation between points and stochastic moments are passed. The `SigmaPointProblemDescription` is computationally efficient, as the computation of points to represent the random variables is only performed once per time step. However, it is not possible to consider Gaussian processes or an additive Wiener process. To take these into account, it is not sufficient to initially convert the random variables into sampling points. Instead, differential equations for the mean and the covariance matrix of the states must be derived, which is implemented in the classes `ResamplingProblemDescription` and `ResamplingGPProblemDescription`. The integration of these differential equations requires a transformation of stochastic moments into sampling points at each discretization point of the numerical integration. The system dynamics $\boldsymbol{f}$ is evaluated at these points and the time derivative of the mean and covariance matrix of the states is calculated from the results. An additive Wiener process can be taken into account by the class `ResamplingProblemDescription` by passing the diffusion matrix of the Wiener process to the constructor

```
ResamplingProblemDescription(..., MatrixConstRef diffMatrixWienerProcess)
```

A disadvantage of `ResamplingProblemDescription` compared to `SigmaPointProblemDescription` is the increased computation time, because the sampling points must be calculated more frequently. Furthermore, the class `ResamplingGPProblemDescription` can be used to consider a Gaussian process approximation of an unknown part of the system dynamics. For this purpose, the constructor

```
ResamplingGPProblemDescription(..., const std::vector<GaussianProcessPtr>&
    gaussianProcessVec, const std::vector<typeInt>& dynamicsIndicesWithGP)
```

receives a vector of Gaussian processes `gaussianProcessVec`, which are created as described in Section 4.2. The output values of the Gaussian processes are each one-dimensional and are combined to obtain the vector $\boldsymbol{g}$ in (3.1b). However, as a Gaussian process should not always be considered for all state differential equations, the argument `dynamicsIndicesWithGP` can be used to specify which state differential equations the Gaussian processes relate to. The size of vector `gaussianProcessVec` must be the same as the size of vector `dynamicsIndicesWithGP`.

A further approximation of the stochastic OCP (3.1) is implemented in the class `MonteCarloProblemDescription`, which provides the constructor

```
MonteCarloProblemDescription(ProblemDescriptionPtr problemDescription,
    PointTransformationPtr pointTransformation)
```

As in `SigmaPointProblemDescription`, the random variables are initially converted into sampling points, but the probabilistic constraints (3.1c) and (3.1d) are not approximated. Instead, all sampling points must fulfill the deterministic constraints. In combination with Monte Carlo sampling, the confidence of the fulfillment of the probabilistic constraints (3.1c) and (3.1d) can be stated. The confidence increases with an increasing number of sampling points, however, the corresponding computation time increases as well.

6 Simulation and Visualization

This Section describes how open-loop and closed-loop simulations can be realized and how the results can be plotted. Section 6.1 describes the creation and parameterization of a solver as well as the simulation process including the saving of simulation data. Subsequently, Section 6.2 shows how the results can be plotted in Matlab and how GRAMPC-S can be integrated into Simulink using an S-function.

6.1 Solver and Simulator

Once the approximation of the stochastic OCP has been selected as described in Section 5, a GRAMPC object can be created as a solver using the function

```
GrampcPtr Solver(ProblemDescriptionPtr problemDescription)
```

where problemDescription is a shared pointer to the stochastic problem description. Before this solver can be used, the initial states and parameters must be passed to it. Since the representation of states and parameters depends on the used propagation method, these must be calculated by the problem description. All problem descriptions provide the function

```
void compute_x0_and_p0(DistributionPtr state, DistributionPtr param)
```

for this purpose. The initial states and parameters can then be accessed by the functions

```
ctypeRNum* x0()
ctypeRNum* p0()
```

These can be passed directly to the solver with

```
solver->setparam_real_vector("x0", problem->x0());
solver->setparam_real_vector("p0", problem->p0());
```

Next, options and parameters of the solver should be selected. For this purpose, the GRAMPC class provides the member functions

```
void setopt_real(const char* optName, ctypeRNum optValue)
void setopt_int(const char* optName, ctypeInt optValue)
void setopt_string(const char* optName, const char* optValue)
```

```
void setopt_real_vector(const char* optName, ctypeRNum* optValue)
void setopt_int_vector(const char* optName, ctypeInt* optValue)
```

which call the corresponding GRAMPC functions. For details, please refer to the documentation of GRAMPC. Once the solver has been parameterized, it can be called as follows:

```
solver->run();
```

The solution found for the OCP can then be read from the solver object. However, if a closed-loop simulation should be implemented, the actual system behavior must also be simulated, which may deviate from the model of the controller. For this purpose, GRAMPC-S provides the `Simulator` class, which receives the actual system dynamics, the actual initial states and system parameters. The system can either be simulated without noise, where you can choose between the Euler and Heun method for integration, or simulated with an additive Wiener process, where the Euler-Maruyama method is applied. If the `writeDataToFile` flag is set, the simulator writes the results of the simulation to text files, which can be plotted in Matlab.

6.2 Using GRAMPC-S with Matlab/Simulink

GRAMPC-S is programmed in C++ in order to realize low computation times and to enable implementation on embedded hardware. For reasons of simplicity, however, Matlab is used to plot the results. For this purpose, it is necessary to write the results to text files so they can be read into Matlab afterwards. If a closed-loop simulation is executed, this can be ensured by setting the argument `writeDataToFile` in the definition of the `Simulator` object. If no simulator is used because only the OCP is to be solved and the resulting trajectories should be plotted, this can be done with the function

```
void writeTrajectoriesToFile(GrampcPtr solver, typeInt numberOfStates)
```

A Matlab script can then be executed to plot the data. The `examples/TEMPLATES` folder contains three templates:

- `plot_simulation`: This script plots the results of a closed-loop simulation.
- `plot_predicted_trajectories.m`: This script plots the obtained solution of the OCP (3.1) in the current time step for stochastic problem descriptions that represent the states internally in the form of sigma points. These are `SigmaPointProblemDescription` and `MonteCarloProblemDescription`.
- `plot_predicted_mean_covariance.m`: This script plots the found solution of the OCP (3.1) in the current time step for stochastic problem descriptions that represent the states internally in the form of mean and covariance matrix: These are `TaylorProblemDescription`, `ResamplingProblemDescription`, and `ResamplingGPProblemDescription`.

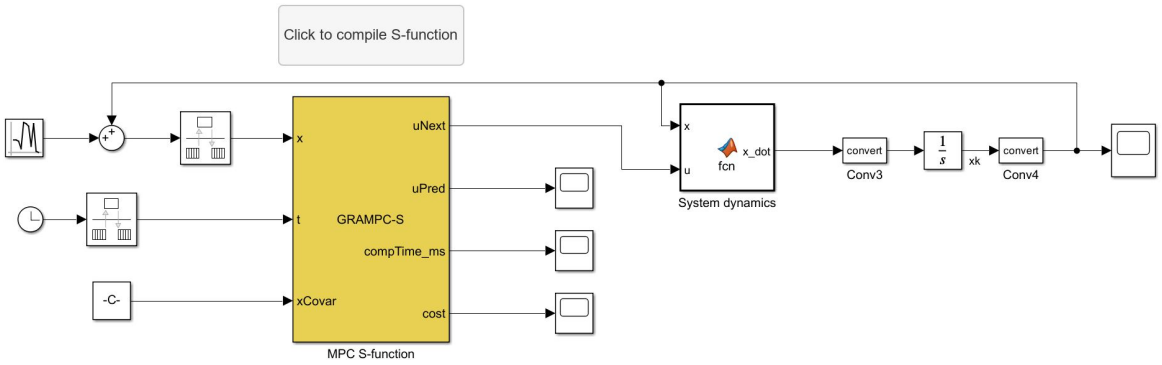

Figure 6.1: Simulink model of GRAMPC-S.

GRAMPC-S can also be used in combination with Simulink. Figure 6.1 shows the Simulink model of GRAMPC-S, implemented using an S-function. By double-clicking on the button `Click to compile S-function`, the S-function is compiled and can be used for the simulation in Simulink. Before compiling, make sure that the folder containing the Simulink model is open in Matlab. The Simulink model shown in Figure 6.1 is located in the folder `examples/inverted_pendulum` and is designed for the example of an inverted pendulum. The S-function is implemented in the file `grampc_s_Sfct.cpp` in the same folder. The S-function must be adapted accordingly for use with other systems. The model parameters, such as sampling time, initial states, etc., are set in the `initFct` callback of the Simulink model, which can be found in `ModelingModelPropertiesCallbacks`.

7 Example: Continuous stirred tank reactor

This section shows the implementation of a control of a continuous stirred tank reactor with GRAMPC-S, which can be found in the folder `examples/reactor`. A simplified and normalized version of the model from [6, 7] is used, which is [8]

$$\dot{c}_A = -k_1 c_A - k_3 c_A^2 + (1 - c_A)u \tag{7.1a}$$

$$\dot{c}_B = k_1 c_A - k_2 c_B - c_B u\,, \tag{7.1b}$$

where c_A is the normalized concentration of the educt A, c_B is the normalized concentration of product B, u is the volume flow rate, and

$$k_1 = 50\,\mathrm{h}^{-1}\,, \qquad k_2 = 100\,\mathrm{h}^{-1}\,, \qquad k_3 = 100\,\mathrm{h}^{-1} \tag{7.2}$$

are parameters of the process. The optimization problem

$$\min_{u} \quad J(u;\, \boldsymbol{x}_0) = \mathbb{E}\left[\int_0^T \Delta\boldsymbol{x}^{\mathsf{T}} \begin{bmatrix} 10^2 & 0 \\ 0 & 10^2 \end{bmatrix} \Delta\boldsymbol{x} + 0.1\Delta u^2 \,\mathrm{d}t\right] \tag{7.3a}$$

$$\text{s.t.} \quad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x},\, u)\,, \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \tag{7.3b}$$

$$\mathbb{P}\left[c_B \leq 0.14\right] \geq 0.95 \tag{7.3c}$$

$$u \in [10,\, 100] \tag{7.3d}$$

should be solved, where the system dynamics (7.3b) are given by (7.1) and $\Delta\boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_{\mathrm{des}}$ and $\Delta u = u - u_{\mathrm{des}}$ denote the deviation of the states and the control input from the corresponding desired values. It is assumed that the current state is not exactly known due to measurement noise, which makes it necessary to predict the uncertainties in (7.3b). These are required to evaluate the probabilistic constraint (7.3c) for the concentration c_B. The problem description will now be implemented for this OCP. For this purpose, a class `ReactorProblemDescription` is created, which provides all the necessary functions from Section 4.1. First, the constructor is implemented:

```
ReactorProblemDescription::ReactorProblemDescription(
    const std::vector<typeRNum>& pSys, const std::vector<typeRNum>& pCost,
    const std::vector<typeRNum>& pCon)
 : pSys_(pSys), pCost_(pCost), pCon_(pCon)
{ }
```

The constructor receives vectors containing the system parameters, the coefficients for the cost function and for the constraint and saves them in member variables. Next, the number of states, the number of control inputs, the number of parameters, the number of constraints and the number of terminal constraints of the OCP must be specified. This is realized in the function `ocp_dim`:

```
void ReactorProblemDescription::ocp_dim(typeInt *Nx, typeInt *Nu,
    typeInt *Np, typeInt *Ng, typeInt *Nh, typeInt *NgT, typeInt *NhT)
{
    *Nx = 2;      *Nu = 1;      *Np = 0;      *Ng = 0;
    *Nh = 1;      *NgT = 0;      *NhT = 0;
}
```

Please note that the generic data type `typeInt` for integer values is used here which is defined by GRAMPC in `grampc_macro.h`. The generic data types `typeRNum` and `ctypeRNum` for floating point numbers are also defined there. The system dynamics (7.1) are implemented in the function `ffct`:

```
void ReactorProblemDescription::ffct(VectorRef out, ctypeRNum t,
    VectorConstRef x, VectorConstRef u, VectorConstRef p)
{
    out[0] = -pSys_[0] * x[0] - pSys_[2] * x[0] * x[0] + (1 - x[0]) * u[0];
    out[1] = pSys_[0] * x[0] - pSys_[1] * x[1] - x[1] * u[0];
}
```

Here, the member variable pSys_ is used, which was set in the constructor. Next, the cost function (7.3a) must be implemented in the function `lfct`:

```
void ReactorProblemDescription::lfct(VectorRef out, ctypeRNum t, VectorConstRef x,
    VectorConstRef u, VectorConstRef p, VectorConstRef xdes, VectorConstRef udes)
{
    out[0] = pCost_[2] * (x[0] - xdes[0]) * (x[0] - xdes[0]) +
    pCost_[3] * (x[1] - xdes[1]) * (x[1] - xdes[1]) +
    pCost_[4] * (u[0] - udes[0]) * (u[0] - udes[0]);
}
```

Again, a member variable is accessed, which was set in the constructor. If terminal constraints need to be taken into account in addition to the iterative costs, these must be implemented in the function `Vfct`. The inequality constraint (7.3c) is implemented in the function `hfct`:

```
void ReactorProblemDescription::hfct(VectorRef out, ctypeRNum t, VectorConstRef x,
    VectorConstRef u, VectorConstRef p)
{
    out[0] = x[1] - pCon_[0];
}
```

The underlying solver GRAMPC is gradient-based, so some derivatives with respect to $\boldsymbol{x}$ and u must be implemented for the above functions. As described in Section 4.1, the matrix product of the Jacobian matrix and an arbitrary vector, which is passed as a pointer, is returned for the derivatives of the system dynamics and the constraint:

```
void ReactorProblemDescription::dfdx_vec(VectorRef out, ctypeRNum t,
    VectorConstRef x, VectorConstRef adj, VectorConstRef u, VectorConstRef p)
{
  out[0] = (-pSys_[0] - pSys_[2] * 2 * x[0] - u[0]) * adj[0] + pSys_[0] * adj[1];
  out[1] = (-pSys_[1] - u[0]) * adj[1];
}

void ReactorProblemDescription::dfdu_vec(VectorRef out, ctypeRNum t,
  VectorConstRef x, VectorConstRef adj, VectorConstRef u, VectorConstRef p)
{
  out[0] = (1 - x[0]) * adj[0] + (-x[1]) * adj[1];
}

void ReactorProblemDescription::dldx(VectorRef out, ctypeRNum t, VectorConstRef x,
  VectorConstRef u, VectorConstRef p, VectorConstRef xdes, VectorConstRef udes)
{
  out[0] = 2 * pCost_[2] * (x[0] - xdes[0]);
  out[1] = 2 * pCost_[3] * (x[1] - xdes[1]);
}

void ReactorProblemDescription::dldu(VectorRef out, ctypeRNum t, VectorConstRef x,
  VectorConstRef u, VectorConstRef p, VectorConstRef xdes, VectorConstRef udes)
{
  out[0] = 2 * pCost_[4] * (u[0] - udes[0]);
}

void ReactorProblemDescription::dhdx_vec(VectorRef out, ctypeRNum t,
  VectorConstRef x, VectorConstRef u, VectorConstRef p, VectorConstRef vec)
{
  out[0] = 0.0;
  out[1] = vec[0];
}

void ReactorProblemDescription::dhdu_vec(VectorRef out, ctypeRNum t,
  VectorConstRef x, VectorConstRef u, VectorConstRef p, VectorConstRef vec)
{
  out[0] = 0.0;
}
```

All functions of the OCP (7.3) are now defined. The next step is to define the distribution of states, create and parameterize the solver and implement the MPC loop. This is realized in the

function `main`, which is located in the file `reactor_simulation_main.cpp` for this example. First, the initial state $\boldsymbol{x}_0$ is created as a Gaussian distribution with mean $\mathbb{E}[\boldsymbol{x}_0] = [0.7,\ 0.01]^\mathsf{T}$ and covariance matrix $P = 10^{-5}\ \boldsymbol{I}$:

```
Vector x0(2);
Matrix P0(2,2);
x0 << 0.7, 0.01;
P0 << 1e-5, 0, 0, 1e-5;
DistributionPtr state = Gaussian(x0, P0);
```

In principle, all methods from Section 5 can be used to propagate the uncertainties. In this example we use polynomial chaos expansion with a maximum polynomial order of 2 and solve the occurring integrals using Gaussian quadrature with a grid of 3 points per dimension:

```
PointTransformationPtr transform = PCE(2, 2, state->polynomialFamily(), 2, 3);
```

Next, the approximation of the probabilistic constraint (7.3b) is determined by the Chebyshev inequality:

```
Vector vec_chance_constraint(1);
vec_chance_constraint << 0.95;
ChanceConstraintApproximationPtr constraintApprox =
    Chebyshev(vec_chance_constraint);
```

The class `ReactorProblemDescription` has already been implemented in the previous steps. An object of this class is now created and passed to a `SigmaPointProblemDescription`:

```
ProblemDescriptionPtr reactorProblem = ProblemDescriptionPtr(
    new ReactorProblemDescription(pSys, pCost, pCon));
SigmaPointProblemDescriptionPtr problem = SigmaPointProblem(
    reactorProblem, constraintApprox, transform);
```

The SigmaPointProblemDescription represents the approximation of the stochastic OCP (7.3), which can be solved by GRAMPC. To this end, a solver must be created and the pointer to the problem description must be passed to it:

```
GrampcPtr solver = Solver(problem);
```

Next, the solver must be parameterized, which is skipped here for reasons of space, but can be found in the file `reactor_simulation_main.cpp`. Finally, the MPC loop is implemented. In each time step, the solver is called, the system is simulated and the current state and time are passed to the solver:

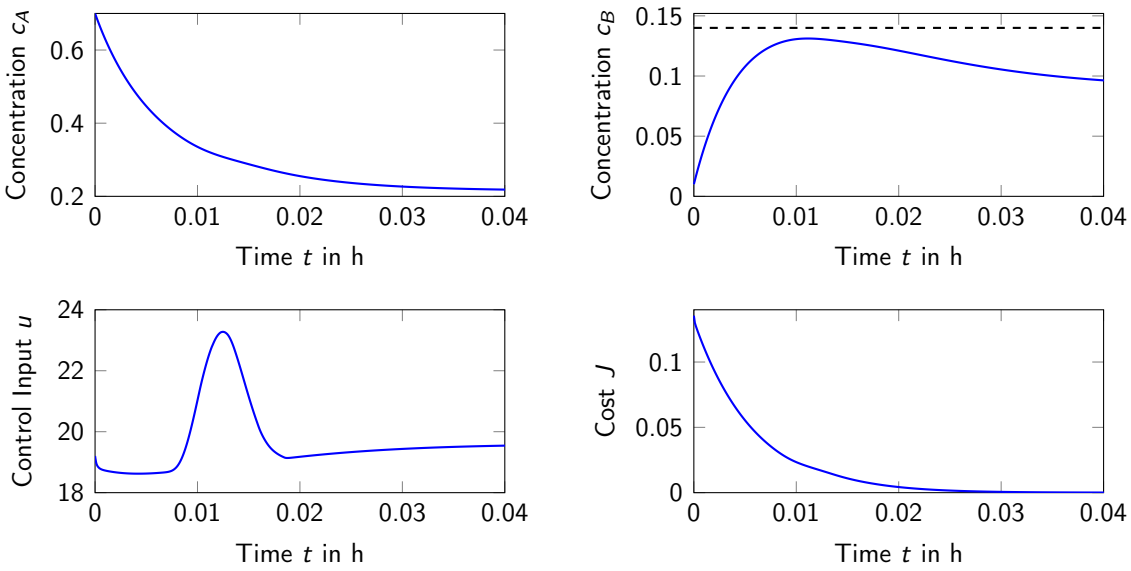

Figure 7.1: Results of the stochastic MPC simulation for the chemical reactor.

```
for(typeRNum t = 0; t < Tsim; t += dt_MPC)
{
    solver->run();

    // simulate system
    Eigen::Map<const Vector> uMap(solver->getSolution()->unext, u0.size());
    x0 = sim.simulate(uMap);
    state->setMeanAndCovariance(x0, P0);

    // set current state and time
    problem->compute_x0_and_p0(state);
    solver->setparam_real_vector("x0", problem->x0());
    solver->setparam_real("t0", t);
}
```

Figure 7.1 shows the results of the simulation for desired states $\boldsymbol{x}_{\text{des}} = [0.215, 0.09]^{\mathsf{T}}$ and a desired control input $u_{\text{des}} = 19.6$. First, the concentration c_A decreases and the concentration c_B increases until the constraint becomes active. The deviation between the trajectory of c_B and the value of 0.14 results from the constraint tightening with the Chebyshev inequality due to the state uncertainty.

A Appendix

This appendix contains lists of all probability density distributions and all kernels that GRAMPC-S supports.

A.1 List of probability density functions

The following list contains all probability density functions that are implemented in GRAMPC-S. Note that it is possible to implement your own probability density functions. These must be declared as derived from the `Distribution` class.

Table A.1: Overview of the available probability distributions and associated parameters.

Name	Probability density function	Parameters
Gaussian distribution	$p(\boldsymbol{x}) = (2\pi)^{-\frac{d}{2}} \det(\boldsymbol{\Sigma})^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)$	$\boldsymbol{\mu} \in \mathbb{R}^d$ $\boldsymbol{\Sigma} \in \mathbb{R}^{d\times d}$
Beta distribution	$p(x) = \frac{1}{B(p,q)}\, x^{p-1}\,(1-x)^{q-1}$	$p>0\,,\ q>0$
Chi-squared distribution	$p(x) = \frac{x^{\frac{n}{2}-1}\exp\left(-\frac{x}{2}\right)}{\Gamma\left(\frac{n}{2}\right) 2^{\frac{n}{2}}}$	$n>0$
Exponential distribution	$p(x) = \lambda \exp(-\lambda\, x)$	$\lambda > 0$
Extreme value distribution	$p(x) = \frac{1}{b}\exp\left(\frac{a-x}{b} - \exp\left(\frac{a-x}{b}\right)\right)$	$a\in\mathbb{R}\,,\ b>0$
F-distribution	$p(x) = \frac{\Gamma\left(\frac{m+n}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\Gamma\left(\frac{n}{2}\right)} \left(\frac{m}{n}\right)^{\frac{m}{2}} x^{\frac{m}{2}-1}(1+\frac{m}{n}x)^{-\frac{m+n}{2}}$	$m>0\,,\ n>4$
Gamma distribution	$p(x) = \frac{\exp\left(-\frac{x}{\beta}\right)}{\beta^{\alpha}\,\Gamma(\alpha)}\, x^{\alpha-1}$	$\alpha>0\,,\ \beta>0$
Log-normal distribution	$p(x) = \frac{1}{\sigma x\sqrt{2\pi}} \exp\left(-\frac{(\ln(x)-\mu)^2}{2\sigma^2}\right)$	$\mu\in\mathbb{R}\,,\ \sigma>0$
Piecewise constant distribution	Histogram of an arbitrary distribution	
Student's t-distribution	$p(x) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sigma\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left(1+\left(\frac{x-\mu}{\sigma}\right)^2 \frac{1}{\nu}\right)^{-\frac{\nu+1}{2}}$	$\mu\in\mathbb{R}\,,\ \sigma>0$ $\nu>2$
Uniform distribution	$p(x) = \frac{1}{b-a}$	$a<b$
Weibull distribution	$p(x) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} \exp(-\left(\frac{x}{b}\right)^a)$	$a>0\,,\ b>0$
Product of uncorrelated distributions	$p(\boldsymbol{x}) = \prod_{i=1}^{d} p_i(x_i)$	

A.2 List of kernels for Gaussian processes

The following list contains all kernels that are implemented in GRAMPC-S. Note that it is possible to implement your own kernels. These must be declared as derived from the `StationaryKernel` class.

Table A.2: Overview of the available kernel functions and associated parameters.

Name	Kernel	Parameters
Squared exponential kernel	$k(\tau) = \sigma^2 \exp\left(-\frac{1}{2}\sum_i \left(\frac{\tau_i^2}{l_i^2}\right)\right)$	σ, $\boldsymbol{l}$
Periodic kernel	$k(\tau) = \sigma^2 \exp\left(-2\sum_i \left(\frac{\sin^2\left(\pi\frac{\tau_i}{p_i}\right)}{l_i^2}\right)\right)$	σ, $\boldsymbol{l}$, $\boldsymbol{p}$
Locally periodic kernel	$k(\tau) = \sigma^2 \exp\left(-2\sum_i \left(\frac{\sin^2(\pi\frac{\tau_i}{p_i})}{l_i^2}\right)\right) \exp\left(-\frac{1}{2}\sum_i \left(\frac{\tau_i^2}{l_i^2}\right)\right)$	σ, $\boldsymbol{l}$, $\boldsymbol{p}$
Matern 3/2 kernel	$k(\tau) = \sigma^2 \left(1 + \sqrt{3\sum_i \left(\frac{\tau_i^2}{l_i^2}\right)}\right) \exp\left(-\sqrt{3\sum_i \left(\frac{\tau_i^2}{l_i^2}\right)}\right)$	σ, $\boldsymbol{l}$
Matern 5/2 kernel	$k(\tau) = \sigma^2 \left(1 + \sqrt{5\sum_i \left(\frac{\tau_i^2}{l_i^2}\right)} + \frac{5}{3}\sum_i \left(\frac{\tau_i^2}{l_i^2}\right)\right) \exp\left(-\sqrt{5\sum_i \left(\frac{\tau_i^2}{l_i^2}\right)}\right)$	σ, $\boldsymbol{l}$
Sum of kernels	$k(\tau) = k_1(\tau) + k_2(\tau)$	
Product of kernels	$k(\tau) = k_1(\tau) k_2(\tau)$	

Bibliography

[1] T. Englert, A. Völz, F. Mesmer, S. Rhein, and K. Graichen, "A software framework for embedded nonlinear model predictive control using a gradient-based augmented Lagrangian approach (GRAMPC)," *Optimization and Engineering*, vol. 20, no. 3, pp. 769–809, 2019.

[2] D. Landgraf, A. Völz, F. Berkel, K. Schmidt, T. Specker, and K. Graichen, "Probabilistic prediction methods for nonlinear systems with application to stochastic model predictive control," *Annual Reviews in Control*, vol. 56, p. 100905, 2023.

[3] D. Landgraf, A. Völz, and K. Graichen, "A software framework for stochastic model predictive control of nonlinear systems (GRAMPC-S)."

[4] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[5] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. The MIT Press, Cambridge, MA, 2005.

[6] K.-U. Klatt, S. Engell, A. Kremling, and F. Allgöwer, "Testbeispiel: Rührkesselreaktor mit Parallel– und Folgereaktion," in *Entwurf Nichtlineare Regelungen*, S. Engell, Ed. Oldenbourg, 1995, pp. 425–432.

[7] R. Rothfuss, J. Rudolph, and M. Zeitz, "Flatness based control of a nonlinear chemical reactor model," *Automatica*, vol. 32, no. 10, pp. 1433–1439, 1996.

[8] K. Graichen, V. Hagenmeyer, and M. Zeitz, "Van de vusse CSTR as a benchmark problem for nonlinear feedforward control design techniques," *IFAC Proceedings Volumes*, vol. 37, no. 13, pp. 1123–1128, 2004.