High-Level Interfaces for the Multiple Shooting Code for Optimal Control MUSCOD

M. Kudruss[a]* and F. Lenders[a] and C. Kirches[a]

[a]*Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University, Germany.*

The demand for model-based simulation and optimization solutions requires the availability of software frameworks that not only provide computational capabilities, but also help to ease the formulation and implementation of the respective optimal control problems. In this article, we present and discuss recent development efforts and applicable work flows using the example of MUSCOD, the Multiple Shooting Code for optimal control, a contemporary and frequently used software package based on a direct and simultaneous approach to optimal control. We show how to facilitate its usage by providing convenient high-level language interfaces that open up the possibility of using well-established modeling languages as back-ends for problem formulation, implementation and evaluation. This is in accordance to well-known design principles that ask for implementing everything in the highest suitable programming language available (here Python), while crunching numbers in the lowest and most efficient one that allows for best exploitation of structure, memory, and CPU usage (here Fortran and C). Using the example of an introductory optimal control problem, we present the most common use patterns in both C++ and Python, for optimal control, nonlinear model predictive control (NMPC), and moving horizon estimation (MHE) applications.

Keywords: Optimal Control, Nonlinear Model Predictive Control, Moving Horizon Estimation, Direct Multiple Shooting, Numerical Software

AMS Subject Classification: 34H05, 49M37, 65K10, 93C15

1. Introduction

Optimal control has found widespread use in science and engineering over the past fifty years. Optimal control problems are generally stated as optimization problems in function space, with the aim to find controls and states such that the state corresponds to the time evolution of a dynamic process as governed by the control, and such that further restrictions are satisfied and a running cost, depending on states and controls, is minimized or maximized.

Numerous techniques aiming at the solution of optimal control problems have been developed during this time. In particular, direct and simultaneous methods based on a first-discretize-then-optimize approach have gained popularity for tackling practical optimal control problems, as can be seen by the large number of software frameworks that follow this route [2, 7, 14, 15, 26, 30, 36, 39, 40, 47]. In the so-called *direct* approach to optimal control, the optimal control problem, posed as an optimization problem in function spaces, is replaced by an finite-dimensional counterpart problem obtained by discretizing the infinite dimensional control function. This typically leads to a high-dimensional, struc-

*Corresponding author. Email: manuel.kudruss@iwr.uni-heidelberg.de

tured nonlinear program (NLP) that is subsequently solved by nonlinear programming methods, see, e.g., [37]. Depending on the type of discretization, different direct methods are distinguished in literature. Single shooting codes, e.g. `VPLAN` [30], aim at solving the reduced problem by using the control to state mapping defined by the dynamic process to eliminate state variables. Multiple shooting methods [9], e.g. `MUSCOD` [14], `ACADO` [26], or `Omuses/HQP` [21], work by subdividing the time horizon and using the control to state mapping separately on these subhorizons to eliminate the states on the subhorizon. In addition, continuity across the subhorizons must be enforced by additional constraints. This leads to nonlinear programs that have a linearly separability structure and high sparsity, and ask for structure exploitation during the solution of the resulting nonlinear programs. Collocation codes, e.g. $\mathbb{SOCS}$ [7], `DIRCOL` [47], `gPROMS` [15], `PROPT` [39], `GPOPS-II` [40], `BOCOP` [36], use a finite dimensional approximation to the state space and enforce satisfaction of the dynamic process only on a discrete set of time points, the collocation points. This leads to a highly sparse nonlinear program that may be solved by sparsity exploiting optimization codes. `CasADi` [2] does not immediately fit into this classification. It is rather a framework for automatic differentiation that interfaces differential equation solvers and nonlinear programming software. It thus eases the realization of custom implementations of single shooting, multiple shooting, and collocation methods. Finally, `TACO` [29] is an interface between the modeling language `AMPL` [19, 20] and the solver MUSCOD that attempts to extend the applicability of `AMPL` also to dynamic optimization problems.

In the sequel of this article, we focus on the software MUSCOD, an implementation of a direct multiple shooting method whose current implementational state has its roots in 1995. A lot of design choices in MUSCOD are influenced by technology available back then, and make the software rather challenging to use for practitioners used to convenient and interactive environments as Python or MATLAB of contemporary times. To set up an optimal control problem and solve it, it is necessary to set up a C library that provides the function describing the dynamic process, objective, and constraints, and provide a textual `.dat` file that specifies several numerical values. Solutions can then be generated by executing the MUSCOD executable, which loads the library, reads the data file, and provides solution information on screen and in another textual output file. We found that for certain applications, and for users trained to work with interactive environments, this approach constitutes a significant barrier and makes MUSCOD unnecessarily hard to use. It has also proven inconvenient in use cases where, for example, a wide range of numerical parameters had to be sampled and assessed. This typically required to set up a new `.dat` file for every parameter set, to execute MUSCOD with this data set and to read back the resulting values after successful execution; a workflow often realized through shell scripts since 1995.

We have thus seen the need to develop easily accessible high-level interfaces to MUSCOD that help to overcome these limitations, making MUSCOD easier to use and more accessable to a wider range of practitioners. We will describe the necessary steps to build such an interface in both Python and C++, and illustrate some real world examples of its successful usage. In particular, we show how to use MUSCOD with a model provided as Functional Mock-Up Unit (FMU) by `Modelica` [3], which has found widespread use in engineering and nowadays is a de-facto standard for model exchange.

1.1 *Contributions*

We introduce `pymuscod` and `pynmpc`, which are two python packages that provide interfaces to the optimal control software MUSCOD and the extensions of MUSCOD for Nonlinear

Model Predictive Control (NMPC) and Moving Horizon Estimation (MHE). We report on our rationale and findings during development and use of these interface, and demonstrate their use for solving two academic optimal control examples. In addition, we show how to combine MUSCOD with models provided as Functional Mock-Up Unit (FMU) as created by the industry standard `Modelica`. We also report on several examples of real- world use cases of MUSCOD enabled by the new interfaces.

1.2 *Structure of the Article*

The remainder of this article is structured as follows. In §2, we introduce the problem formulations that can be treated by MUSCOD. The direct multiple shooting method for optimal control is briefly reviewed in §3, and the real-time iteration scheme required in Nonlinear Model Predictive Control (NMPC) is briefly explained. §4 introduces the Python interface to MUSCOD and the NMPC and MHE facilities, and shows examples to demonstrate how to make use of them. Further examples of real-world use cases of these interfaces are given in §5. Here we also demonstrate how MUSCOD can be used with models built in environments that adhere to the industry standard `Modelica`. Finally, we offer a summary and conclusions in §6, and provide an outlook on future work.

2. Problem Formulations and Examples

The software package MUSCOD is a robust and efficient optimization tool that allows to implement and solve a wide class of optimal control problems constrained by differential-algebraic equations (DAE). The software package is capable of solving multi-stage optimal control problems, including discontinuities in stage boundary points. Further specialization of the optimal control problem enables the solution of nonlinear least-squares problems, which may also be used to formulate state and parameter estimation problems for dynamic systems required for model validation. A dedicated implementation of the required derivative-based optimization algorithms, here sequential quadratic programming, allow the application of MUSCOD for nonlinear model predictive control and moving horizon estimation on top. We give a brief overview of the above mentioned optimal control problems.

2.1 *Optimal control problem formulation*

We consider the following class of optimal control problems (OCP) on a finite time horizon $\mathcal{T} = \bigcup_{0\le j<n_\mathrm{m}} \mathcal{T}_j \subset \mathbb{R}$ partitioned into $n_\mathrm{m} \in \mathbb{N}$ model stages $\mathcal{T}_j = [t_j, t_{j+1}]$:

$$\min_{\substack{y(\cdot),z(\cdot),u(\cdot),\\ p,\{t_{i,j}\}}} \sum_{0\le j<n_\mathrm{m}} \int_{t_j}^{t_{j+1}} L_j(y(t),z(t),u(t),p)\,\mathrm{d}t + E_j(y(t_{j+1}),z(t_{j+1}),p) \tag{1a}$$

$$\text{s.t.} \quad 0 = f_j(y(t),\dot{y}(t),z(t),u(t),p), \qquad t\in\mathcal{T}_j, \quad 0\le j<n_\mathrm{m}, \tag{1b}$$

$$0 = g_j(y(t),z(t),u(t),p), \qquad t\in\mathcal{T}_j, \quad 0\le j<n_\mathrm{m}, \tag{1c}$$

$$y(t_j^+) = \Delta_{j-1}(y(t_j^-),z(t_j^-),p), \qquad 1\le j<n_\mathrm{m}, \tag{1d}$$

$$0 \le c_j(y(t),z(t),u(t),p), \qquad t\in\mathcal{T}_j, \quad 0\le j<n_\mathrm{m}, \tag{1e}$$

$$0 \leqq \sum_{t_{i,j}} r_{i,j}(y(t_{i,j}),z(t_{i,j}),p), \qquad \{t_{i,j}\}\subset\mathcal{T}_j, \quad 0\le j<n_\mathrm{m}. \tag{1f}$$

In this problem class, we strive to find a control trajectory $u : \mathcal{T} \to \mathbb{R}^{n_\mathrm{u}}$ such that an objective function composed of stage-wise Lagrange terms $L_j : \mathbb{R}^{n_\mathrm{y}} \times \mathbb{R}^{n_\mathrm{z}} \times \mathbb{R}^{n_\mathrm{u}} \times \mathbb{R}^{n_\mathrm{p}} \to \mathbb{R}$, $0 \leq j < n_\mathrm{m}$ and Mayer terms $E_j : \mathbb{R}^{n_\mathrm{y}} \times \mathbb{R}^{n_\mathrm{z}} \times \mathbb{R}^{n_\mathrm{p}} \to \mathbb{R}$. The state trajectory $x : \mathcal{T} \to \mathbb{R}^{n_\mathrm{x}}$ of the dynamic system is defined by stage-wise systems of differential algebraic equations (DAEs) with implicit differential right hand sides $f_j : \mathbb{R}^{n_\mathrm{y}} \times \mathbb{R}^{n_\mathrm{z}} \times \mathbb{R}^{n_\mathrm{u}} \times \mathbb{R}^{n_\mathrm{p}} \to \mathbb{R}^{n_\mathrm{y}}$ and algebraic equations $g_j : \mathbb{R}^{n_\mathrm{y}} \times \mathbb{R}^{n_\mathrm{z}} \times \mathbb{R}^{n_\mathrm{u}} \times \mathbb{R}^{n_\mathrm{p}} \to \mathbb{R}^{n_\mathrm{z}}$, $0 \leq j < n_\mathrm{m}$ that may both depend on the global model parameters $p \in \mathbb{R}^{n_\mathrm{p}}$ of the system. Discontinuities at the model stage borders $t_j \in \mathbb{R}$, $1 \leq j < n_\mathrm{m}$ are defined by transition functions $\Delta_j : \mathbb{R}^{n_\mathrm{y}} \times \mathbb{R}^{n_\mathrm{z}} \times \mathbb{R}^{n_\mathrm{u}} \times \mathbb{R}^{n_\mathrm{p}} \to \mathbb{R}^{n_\mathrm{x}}$, $0 \leq j < n_\mathrm{m}$. In (1d), superscripts $+$ and $-$ denote one-sided limits into t_j. In addition, mixed state-control inequality path constraints $c_j : \mathbb{R}^{n_\mathrm{y}} \times \mathbb{R}^{n_\mathrm{z}} \times \mathbb{R}^{n_\mathrm{u}} \times \mathbb{R}^{n_\mathrm{p}} \to \mathbb{R}^{n_\mathrm{c}}$, $0 \leq j < n_\mathrm{m}$ on the horizon and contributions $r_{i,j} : \mathbb{R}^{n_\mathrm{y}} \times \mathbb{R}^{n_\mathrm{z}} \times \mathbb{R}^{n_\mathrm{p}} \to \mathbb{R}^{n_{\mathrm{r},i,j}}$, $0 \leq j < n_\mathrm{m}$ to coupled (in-)equality point constraints on a finite number of N_j+1 time points $\{t_{i,j}\} \subset \mathcal{T}_j$, $0 \leq i \leq N_j$, are imposed on model stages $0 \leq j < n_\mathrm{m}$ of the problem. The latter also comprise, for example, periodicity constraints, and decoupled constraints such as initial values and terminal values.

2.2 *An Introductory Optimal Control Problem*

As a prototypical example of an optimal control problem, we consider the classical re-entry of a space shuttle of Apollo type [43]. Using the following notation and constants

Symbol	Description	Value
v	tangential velocity	
γ	flight path angle	
R	earth radius	209
ξ	normalized altitude above earth's surface	h/R
ϱ	atmospheric density	$\varrho = \varrho_0 e^{-\beta R \xi}$, $\varrho_0 = 2.704 \cdot 10^{-3}$
u	control influencing drag and lift	
C_W	aerodynamical drag coefficient	$1.174 - 0.9 \cos u$
C_A	aerodynamical lift coefficient	$0.6 \sin u$
S/m	frontal area over mass of vehicle	53200
g	gravitational acceleration	$3.2172 \cdot 10^{-4}$

the optimal control problem is stated as follows:

$$\min_{y(\cdot),u(\cdot),T} \int_0^T 10\ v(t)^3 \sqrt{\varrho(t)}\, \mathrm{d}t \tag{2a}$$

$$\text{s.t.} \quad \dot{v}(t) = -\tfrac{S\varrho(t)\ v(t)^2}{2m} C_W(u(t)) - \tfrac{g\ \sin\gamma(t)}{(1+\xi(t))^2}, \tag{2b}$$

$$\dot{\gamma}(t) = \tfrac{S\varrho(t)\ v(t)}{2m} C_A(u(t)) + \tfrac{v(t)\ \cos\gamma(t)}{R(1+\xi(t))} - \tfrac{g\ \sin\gamma(t)}{v(t)(1+\xi(t))^2}, \tag{2c}$$

$$\dot{\xi}(t) = \tfrac{v(t)\ \sin\gamma(t)}{R}, \tag{2d}$$

$$v(0) = 0.36, \qquad v(T) = 0.27, \tag{2e}$$

$$\gamma(0) = -\tfrac{8.1^\circ \pi}{180^\circ}, \quad \gamma(T) = 0, \tag{2f}$$

$$\xi(0) = \tfrac{4}{R}, \qquad \xi(T) = \tfrac{2.5}{R}. \tag{2g}$$

2.3 *Nonlinear model predictive control*

Nonlinear model predictive control (NMPC) is a closed-loop control strategy in which the feedback control is computed from the current system state by solving an open-loop

optimal control problem over a single finite prediction horizon on-line, therefore also denoted as receding horizon control. In contrast to the OCP problem (1) and for brevity of exposition, we focus on a single stage time horizon $\mathcal{T} = [0, T]$ for the NMPC problem and also limit the problem formulation to ordinary differential equations (ODE) instead of DAEs. Furthermore, we only present a tracking NMPC problem. However note that this is not a limitation of the software package MUSCOD but allows an easier derivation from a limited problem class. The NMPC problem under consideration is of the form of

$$\begin{aligned}
\min_{y(\cdot),u(\cdot),p} \quad & \int_0^T \|\ell(y(t),u(t),p) - \bar{\ell}(t,p)\|_W^2 \,\mathrm{d}t && \text{(3a)}\\
& + \|e(y(T),p) - \bar{e}(p)\|_{\bar{W}}^2 && \text{(3b)}\\
\text{s.t.} \quad & 0 = f(y(t),\dot{y}(t),u(t),p), \qquad t \in \mathcal{T}, && \text{(3c)}\\
& 0 = y(0) - \hat{y}_0, && \text{(3d)}\\
& 0 = p - \hat{p}, && \text{(3e)}\\
& 0 \leq c(y(t),u(t),p), \qquad t \in \mathcal{T}, && \text{(3f)}\\
& 0 \leqq \sum_{t_i} r_i(y(t_i),p), \qquad \{t_i\} \subset \mathcal{T}. && \text{(3g)}
\end{aligned}$$

In this problem, the deviation of a non-linear objective $\ell :\to \mathbb{R}_1^n$ from a given set-point $\bar{\ell}$, as well as a penalty for an end-point $e :\to \mathbb{R}_1^n$ deviation from a given set-point $\bar{e}$ is to be minimized with respect to a weighted L^2-norm with positive definite weighting matrices W and $\bar{W}$. For the NMPC problem, the initial value and parameter embedding constraints (3d, 3e) are of special interest as they enter the problem linearly and can be exploited to solve NMPC problems in real-time.

2.4 *Moving horizon state and parameter estimation*

Model validations, both online and off-line, are possible through state and parameter estimation techniques based on time discrete nonlinear least-squares problems. In the NMPC context, moving horizon estimation (MHE) methods, cf. [33], are employed to provide state and parameter estimates for $\hat{y}_0 \in \mathbb{R}^{n_\mathrm{y}}$, $\hat{p} \in \mathbb{R}^{n_\mathrm{p}}$ in (3).

We apply the same specialization of the problem formulation as for the NMPC problem (3) for better readability, i.e. we define it for a single stage time horizon $\mathcal{T} = [-T, 0]$ and focus on ODE only. The MHE problem minimizes the error between the model response $h : \mathbb{R}^{n_\mathrm{y}} \times \mathbb{R}^{n_\mathrm{u}} \times \mathbb{R}^{n_\mathrm{p}} \to \mathbb{R}^{n_\mathrm{h}}$ and N measurements $\eta = h(y(t),u(t),p^*) + \epsilon(t)$, from the real system defined by the true but unknown parameters p^*, subject to an additive measurement error $\epsilon(t) \sim \mathcal{N}(0,\Xi)$, with zero-mean and covariance matrix $\Xi = \mathrm{diag}(\xi_0,\ldots,\xi_{n_\mathrm{h}})$. The MHE problem reads as follows:

$$\begin{aligned}
\min_{y(\cdot),p} \quad & \sum_{k=-N}^{0} \|h(y(t_k),u(t_k),p) - \eta_k\|_{\Xi_k}^2 + \left\| \begin{pmatrix} y(-T) - \bar{y}_{-T} \\ p - \bar{p} \end{pmatrix} \right\|_P^2 && \text{(4a)}\\
\text{s.t.} \quad & 0 = f(y(t),\dot{y}(t),u(t),p), \quad t \in \mathcal{T}, && \text{(4b)}\\
& 0 \leq c(t,y(t),u(t),p), \qquad t \in \mathcal{T}, && \text{(4c)}
\end{aligned}$$

where the controls u are fixed to the (in general finitely many) values that have already been applied to the system in the past on $[-T, 0]$. The initial values $\bar{y}_{-T}$, $\bar{p}$ in (4a) weighted by the covariance matrix P are used to incorporate *a-priori* information into the problem.

The quantities of interest here are the state and parameter estimates for $\hat{y}_0$ and $\hat{p}$ in (3), which are required for (3d) and (3e).

2.5 *An Introductory NMPC and MHE Example*

We discuss the NMPC and MHE capabilities using an academic benchmark problem, e.g. [28], as prototypical NMPC example:

$$\min_{y(\cdot),u(\cdot)} \int_0^T 10 \, \|y(t)\|_2^2 + 1 \, \|u(t)\|_2^2 \, \mathrm{d}t \tag{5a}$$

$$\text{s.t.} \quad \dot{y}(t) = (1 + y(t)) \, y(t) + u(t), \qquad t \in [0, T] \tag{5b}$$

$$-1 \leq y(t) \leq 1, \; -1 \leq u(t) \leq 1. \tag{5c}$$

The problem has a stable equilibrium point at $y \equiv -1$ and an instable equilibrium point at $y \equiv 0$. The latter is of particular interest as a benchmark set-point because the state y will blow up in finite time if the instable set-point is not controlled.

3. The Direct Multiple-Shooting Method for Optimal Control

In this section, we briefly present a direct and simultaneous approach to optimal control, the direct multiple shooting method, used by the software package MUSCOD to discretize, parameterize, and solve the optimal control problem (1) or its NMPC or MHE counterparts. The main purpose of this section is to serve as a reference, to which we may later refer when describing functions and data structures exposed by the high-level interfaces.

3.1 *The Direct Approach to Optimal Control*

The direct approach to optimal control commences by discretizing the control $u(\cdot)$ on a prescribed time grid $0 = \tau_0 < \tau_1 < \ldots < \tau_N = n_\mathrm{m}$ comprised of N intervals. The end point τ_N is chosen such that each model stage is mapped onto a unit interval, and the grid points are assumed to comprise the integers, which correspond to model stage boundary points. This grid also partitions the physical time horizon $\mathcal{T} = [t_0, t_\mathrm{f}] = [t_0, t_0 + \sum_{k=0}^{n_\mathrm{m}-1} h_k]$ into N intervals, where the $h_i := t_{i+1} - t_i$ denote the physical durations of model stages. Then, we have

$$t(\tau) := t_0 + \sum_{k=0}^{\ell-1} h_k + h_\ell(\tau - \ell), \quad \tau \in [0, n_\mathrm{m}] \subset \mathbb{R},$$

wherein ℓ is the index of the stage containing τ, i.e. $\ell = \lfloor \tau \rfloor$.

The discretization of a control $u_j(\cdot)$, is carried out by means of base functions b_j parametrized by $n_{\mathrm{q},j}$ control parameters $q_{i,j,\ell} \in \mathbb{R}$,

$$u_j(\tau)\Big|_{[\tau_i, \tau_{i+1})} := b_j(\tau, q_{i,j,1}, \ldots, q_{i,j,n_{\mathrm{q},j}}) \in \mathbb{R} \tag{6}$$

on time intervals $[\tau_i, \tau_{i+1})$ for $0 \leq i \leq N$ and for controls $1 \leq j \leq n_\mathrm{u}$. Typical choices are piecewise constant controls, $b_j(\tau, q_{i,j}) = q_{i,j}$ and $n_{\mathrm{q},j} = 1$, piecewise linear controls,

$(\tau_{i+1}-\tau_i)b_j(\tau,q_{i,j}) = (\tau-\tau_i)q_{i,j,2} + (\tau_{i+1}-\tau)q_{i,j,1}$ and $n_{\mathrm{q},j}=2$, or piecewise cubic spline controls ($n_{\mathrm{q},j}=4$). Moreover, continuity conditions

$$u_j\Big|_{[\tau_i,\tau_{i+1})}(\tau_{i+1}) = b_j(\tau_{i+1},q_{i,j}) \stackrel{!}{=} b_j(\tau_{i+1},q_{i+1,j}) = u_j\Big|_{[\tau_{i+1},\tau_{i+2})}(\tau_{i+1}), \quad 0\le i<N-1$$

in the grid points τ_i may be imposed if $n_{\mathrm{q},j}\ge 2$.

3.2 *The Direct Multiple Shooting Method*

The direct multiple shooting method for optimal control [9] then further parametrizes the state trajectory $y(\cdot)$ by introducing $N+1$ state variables $s_i\in\mathbb{R}^{n_{\mathrm{y}}}$, $z_i\in\mathbb{R}^{n_{\mathrm{z}}}$ on the time grid points, and by solving relaxed DAE initial value problems (IVP) separately on the time intervals:

$$\begin{aligned}
&0 = h_\ell f(y(t),\tfrac{\mathrm{d}y(t)}{\mathrm{d}\tau},z(t),b(\tau,q_i),p), && t=t(\tau),\ \tau\in[\tau_i,\tau_{i+1}], && \text{(7a)}\\
&0 = g(y(t),z(t),b(\tau,q_i),p) - \theta_i(\tau)g(s_i,z_i,b(\tau,q_i),p), && t=t(\tau),\ \tau\in[\tau_i,\tau_{i+1}], && \text{(7b)}\\
&0 = y(t(\tau_i))-s_i, \quad 0 = z(t(\tau_i))-z_i. && && \text{(7c)}
\end{aligned}$$

Herein, h_ℓ again denotes the duration of the model stage to which the time interval $[\tau_i,\tau_{i+1}]$ belongs. By way of (7b), the task of finding consistent algebraic initial values z_i is relaxed by choosing suitable nonnegative strictly decreasing real-valued functions $\theta_i(\tau)$ with $\theta_i(\tau_i)=1$, e.g. $\theta_i(\tau)=\exp(-\alpha(\tau-\tau_i))$ for some $\alpha>0$. The solutions of these IVPs are denoted by $y(\tau_{i+1};\tau_i,s_i,q_i,p,h)$ and $z(\tau_{i+1};\tau_i,s_i,q_i,p,h)$.

In the solution of the problem thusly obtained, continuity of the differential state trajectory is enforced by imposing additional matching conditions

$$0 = s_{i+1} - y(\tau_{i+1};\tau_i,s_i,q_i,p,h), \qquad \tau_{i+1}\notin\mathbb{N} \qquad \text{(8a)}$$

that also cover the model stage transitions (1d) in grid points τ_{i+1} coinciding with model stage boundaries,

$$0 = s_{i+1} - \Delta_\ell(y(\tau_{i+1};\tau_i,s_i,q_i,p,h),z(\tau_{i+1};\tau_i,s_i,q_i,p,h),p), \quad \tau_{i+1}\in\mathbb{N} \qquad \text{(8b)}$$

Here, index ℓ denotes the index of the model stage ending in τ_{i+1}. Algebraic consistency is enforced by imposing

$$0 = g(s_i,z_i,b(\tau_i,q_i),p), \quad 0\le i\le N. \qquad (9)$$

3.3 *Structured Nonlinear Programming*

From this discretization and parametrization, a large but structured nonlinear programming problem (NLP) is derived, and reads

$$\begin{aligned}
\min_{s,z,q,p,h}\ & \sum_{i=0}^{N-1}\tilde{L}_i(s_i,z_i,q_i,p,h) + \tilde{E}(s_N,z_N,p) && && \text{(10a)}\\
\text{s.t.}\ & 0 = y(\tau_{i+1};\tau_i,s_i,z_i,q_i,p,h) - s_{i+1}, && 0\le i<N, && \text{(10b)}\\
& 0 = \tilde{g}(s_i,z_i,q_i,p), && 0\le i\le N, && \text{(10c)}
\end{aligned}$$

$$0 \leq \tilde{c}(\tau_i, s_i, z_i, q_i, p), \qquad 0 \leq i \leq N, \tag{10d}$$
$$0 \leqq \tilde{r}_j(\tau_i, s_i, z_i, q_i, p), \qquad 0 \leq i \leq N, \tag{10e}$$

Herein, the functions $\tilde{L}_i$, $\tilde{E}$, $\tilde{g}$, $\tilde{c}$, and $\tilde{r}_j$ are discretized counterparts of the respective functions in (1) that also account for the control discretization (6) and possibly control continuity conditions. Evaluations of the constraints (10b, 10c) require the solution of the IVPs (7). To do so, shooting approaches adaptively discretize the IVPs in time by making use of state-of-the-art DAE solvers, for example [1].

Problem (10) may be solved efficiently by tailored structure-exploiting sequential quadratic programming (SQP) methods. In particular, the equality constrained (10b) and (10c) may be eliminated by a partial null-space approach, referred to as condensing and partial reduction. We refer the reader to [9, 34] for details. First and possibly second order derivatives, required by the SQP method, then involve the computation of sensitivities of the solution of the dynamics with respect to all dependencies according to the principle of internal numerical differentiation (IND), cf. [1, 8].

After elimination of algebraic states z^k by partial reduction and differential states $s_1^k, \ldots, s_N^k$ by condensing, an iterate w^k in iteration k of an SQP method comprises subvectors $w^k = (p^k, h^k, s_0^k, q^k)$. A second-order model of the NLP that takes the form of a quadratic programming problem (QP) is computed in this iterate, and reads

$$\min_{\Delta w} \quad \tfrac{1}{2}\Delta w^T B(w^k)\Delta w + \Delta w^T b(w^k) \tag{11a}$$
$$\text{s.t.} \quad 0 = c(w^k) + \nabla c(w^k)^T \Delta w, \tag{11b}$$
$$0 \leq d(w^k) + \nabla d(w^k)^T \Delta w, \tag{11c}$$

Matrix $B(w^k)$ denotes the Hessian of the Lagrangian of (10) in w^k, or a symmetric and typically positive definite approximation thereof, and $b(w^k)$ denotes the gradient of the objective (10a). Constraints (11b) and (11c) denote local linearizations of the equality- and inequality constraints (10b, 10c, 10e) and (10d, 10e), respectively. This QP will be small and dense in general, and may be solved by an active-set QP solver such as QPSOL [22], qpOASES [16, 17], or QORE [42]. Its solution $\Delta w^k = (\Delta p^k, \Delta h^k, \Delta s_0^k, \Delta q^k)$ serves as a search direction to find the next SQP iterate w^{k+1}. Globalization of the convergence of the SQP method, i.e. the guaranteed finding of a local minimum from any initial guess, may (here) be achieved by either a trust-region or a line search method, cf. [37].

3.4 *Real-Time Iterations for NMPC and MHE*

State-of-the-art NLP-based NMPC methods rely on the so-called real-time iteration scheme due to [12, 13]. The idea is to use the local contractivity properties of Newton's method and only apply one SQP iteration, which requires the solution of a single QP, to provide feedback in real-time. This is achieved by careful initialization of the SQP method. The SQP iteration can be separated into three distinct phases, i.e. preparation, feedback and transition.

(1) *Preparation:* In the preparation phase the evaluation of states and derivatives as well as the structure exploitation takes place. This way a QP is set up except for the values for the current initial system state $\hat{y}_0 \in \mathbb{R}^{n_\mathrm{y}}$ and of the model parameters $\hat{p} \in \mathbb{R}^{n_\mathrm{p}}$, which enter the NMPC problem (3) linearly.

In particular, the data $B(w^k)$, $b(w^k)$, $\nabla c(w^k)$, $d(w^k)$ and $\nabla d(w^k)$ of the QP (11) is evaluated as well as the parts of $c(w^k)$ that do not correspond to discretizations of $\hat{y}_0$

and $\hat{p}$.

(2) *Feedback:* As soon as the missing estimates for $\hat{y}_0$ and $\hat{p}$ are available the feedback control is computed in the feedback phase by solving the QP.

(3) *Transition:* After the control is fed back to the system the structure exploitation is rolled back, the horizon is shifted and the Lagrange multipliers are updated.

Every feedback phase is then again followed by a preparation phase of the next time step.

In NMPC, there are two characterical times that have to be taken into account: sampling time and feedback delay. The feedback delay is the time between the availability of the measurements of system state $\hat{y}_0$ and parameters $\hat{p}$ until the subsequent computation of the feedback control while the sampling time determines the cycle length between subsequent measurements. It is important to ensure that feedback times are short so that the system state and parameters do not deviate too much from the measurements that are used to determine to feedback control. The feedback time in this real-time iteration scheme is essentially comprised by cost to solve one quadratic program, while the computational heavy parts constituting of evaluation of QP data already have been performed in the preparation phase before the measurements are known. The sampling time in this scheme is the sum of the times of the preparation, feedback and transition phase.

This way computational expensive parts can be separated from time-critical ones and the computational delay of the feedback is reduced to only the time required to solve a single QP.

Approaches similar to those used to achieve real-time feedback control for NMPC can be used to solve the MHE problem.

4. Interfaces and Features

In the following section, we give a brief overview as well as some example use- cases of the current in-use expansions as they are shown in the overview of Figure 1.

4.1 *Providing Modeling Language Interfaces*

In this section, we present developments towards easier interfacing of MUSCOD to contemporary modeling languages for dynamic systems. The issue here is to facilitate the user's task of implementing differential right hand side function f and algebraic right hand side function g in problem (1), typically with the help of graphical user interfaces such as Modelica or Matlab/Simulink. The FMU format has found widespread adoption as a universal exchange format for dynamic models created in this way, and is supported, e.g., by Modelica tools. We hence discuss the interfacing workflow using an FMU adhering to the FMI 2.0 standard as an example.

4.1.1 FMU Initialization

For MUSCOD, we implemented a generic dynamic model that contains an interface to a loadable FMU library that may be freely specified. In Listing 2, the variable `instantiateFMU` of proxy class type `InstantiateFMU` is automatically created when MUSCOD loads the generic model. In its constructor, it instanciated the FMU model, given the Globally Unique Identified (GUID) and resource location (i.e. file path and name) that may be specified by the user. The FMU model is initialized subsequently, and is ready for serving function evaluation requests afterwards.

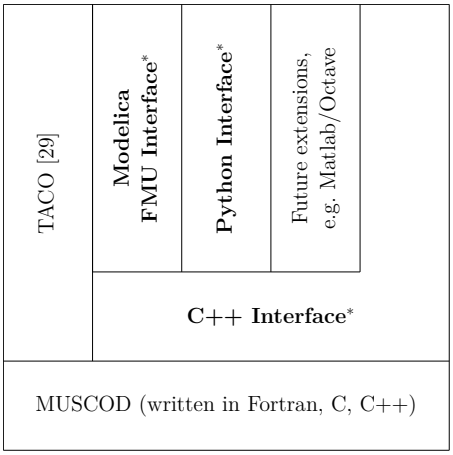

Figure 1. This scheme shows how different modeling languages and high-level interfaces connect to the lower-level core of MUSCOD, either through the C/C++ interface or, in case of TACO that predates developments reported in this article, directly to the respective data structures. * indicates extensions proposed in this article.

4.1.2 FMU Mapping of Variables

FMU variables, including differential states, controls, and model parameters are identified by reference integer values. that must be communicated to the generic MUSCOD FMU interface. These can either be hard-coded or read from an XML generated by FMU-creating tools such as Dymola. Listing 3 shows a list of reference integer values for a control $u(\cdot) \in \mathbb{R}^2$.

4.1.3 FMU Function Evaluation

A generic MUSCOD model library for interfacing dynamic models stored in FMU format was designed. Listing 4 shows the implementation of a generic differential right hand side function for ODE constrained optimal control. A similar implementation is possible for the Lagrange-type part of the objective functions, and for constraint residuals provided as FMU output variables.

4.1.4 Current FMU Requirement and Shortcomings

The current implementation of the FMU programming interface has proven sufficient to enable working with MUSCOD when the dynamic model is limited to continuous ODEs. DAE consistency is currently handled internally by FMU, and DAE systems are exposed as ODEs in a reduced space to the caller. This involves iterative solution of nonlinear DAE consistency constraints, and undermines the simultaneous approach followed by MUSCOD and most other efficient DAE optimal control packages. To extend the FMI standard to complement state-of-the-art optimization software, the paradigm of external control over all adaptive components needs to be adhered to. This currently is partially the case for switched systems, not discussed in this article, but needs to be extended to, e.g., iterative solvers for DAE consistency, to direct linear algebra involving pivoting decisions, and to iterative linear algebra involving termination criteria. Whenever it is desirable to call such procedures from MUSCOD through the FMU programming interface, all information about control about adaptive components, including pivoting sequences, iteration counts, matrix factors or outcome of conditional evaluations, should be conveyed to the FMU by the caller, here MUSCOD. This would grant the caller control over potential sources of non- differentiability inside the FMU. Indeed, the FMI 2.0 standard already made some

progress into this direction by admitting the caller to maintain an FMU state object that documents the state and outcome of some non-differentiable actions, and to pass this FMU state object to the FMU, to be used for subsequent function evaluation.

4.2 *Providing High-Level Language Interfaces*

MUSCOD was first written as monolithic application to be executed by the user by providing a model library and a problem specification in form of a DAT file (ini) format. In the scope of this article, we refactored the software package MUSCOD to be less monolithically and therefore better fit into the current use-cases. In the following section, we present the current state of this process and explain the new structure by following the implementation of two use-cases .

4.2.1 The MUSCOD C/C++ Facade

The MUSCOD core written in C/C++ and Fortran is a complex piece of software decomposed into several submodules, classes and routines with several thousands of lines of code. Therefore, we first added a C/C++ facade to only forward the important functions to offer a clear interface to the user. The functions calls forwarded as methods of a MUSCOD instance interact with different parts of the software package or the data and are briefly described in the following. Please note that the elementary linear algebra types (`LVec`, `LVecList`, `DVec`, `DVecList`, `DPartVec`, `DMat`) are from *LIBLAC* [35].

The required function calls to load a model in the form of a shared library as well as the specifications, options and initial guesses for the NLP (10) are

- `long setModelPathAndName (const char *model_path, const char *model_name);`

  allows to specify the relative path to and the name of the problem specific MUSCOD DAT file
- `long loadFromDatFile (const char *relative_dat_path, const char *dat_name);`

  is used to load all problem definitions from a MUSCOD DAT file as well as the there specified shared model library
- `long initializeModelFromLibrary (const char *libname);`

  is used to load the shared model library directly
- `long initializeModelFromFunction (void (*def_model)(), void (*def_model_userdata)(void *userdata));`

  MUSCOD relies on the definition of a *def_model* function pointer. The function call has to initialize all required dimensions, model stages, objectives and constraints. *def_model_userdata* is an additional, user-defined context pointer, which can be provided to MUSCOD.
- ```
  long loadFromData (LVec &nshoot, LVec &rob_sd0, LVec &rob_p, LVec &stage_fix,
          LVec &par_fix, LVecList &ctrl_par_mode, LVecList &ctrl_match, LVec &num_clsq);
  ```

  can be used to specify all initially required fields of MUSCOD without specifying them in a DAT file. *nshoot* is a discrete array specifying the number of shooting nodes in each model stage, *rob_sd0* and *rob_p* are used for optional robust optimal control, *stage_fix* and *par_fix* are discrete arrays specifying which of the global model stage lengths or parameters are free for the optimizer, *ctrl_par_mode* and *ctrl_match* are discrete arrays that control the stage-wise order of approximation and if additional constraints are imposed for the continuity, and *num_clsq* denotes the number of continuous least-square functions. Please note that one must have called *initializeModel* before using this function and one has to provide initial data after this function has been called via *setInitialGrid*, *setInitialStates*, *setInitialControls*, *setInitialParameters* and *setInitialStage*.
- `long setInitialGrid(DPartVec *ms_grid);`
  ```

initializes the multiple shooting time grid by providing each node explicitly where here the partitioned vector *ms_grid* contains the stage-wise time nodes indexed by $getNMSI(imos)+1, imos = 0, \ldots, getNMOS()-1$. Please note that passing a *NULL* pointer all model stages are initialized with an equidistant grid.

- ```
  long setInitialStates(
          long guess_type,
          DVecList &dstart, DVecList &dscale, DVecList &dlbound, DVecList &dubound,
          DVecList &astart, DVecList &ascale, DVecList &albound, DVecList &aubound
  );
  ```

  assigns initial input data on the shooting grid for both the differential and algebraic states depending on the *guess_type* (0: provide all entries, 1: interpolate linearly from first to last, 2: forward simulation from first entry) as well as their respective scaling factors **scale* and box constraints given by lower and upper bounds **lbound*, **ubound*. Please note that all entries are indexed by $imsn = 0, \ldots, getTotalMSN() - 1$, i.e. are not separated into different stages.
- ```
  long setInitialControls(
          DVecList &ustart, DVecList &uscale, DVecList &ulbound, DVecList &uubound,
          DVecList *uestart, DVecList *uescale, DVecList *uelbound, DVecList *ueubound,
          DVecList *udotstart, DVecList *udotscale, DVecList *udotlbound, DVecList *udotubound,
          DVecList *uedotstart, DVecList *uedotscale, DVecList *uedotlbound, DoVecList *uedotubound
  );
  ```

  assigns initial input data on the shooting grid for the controls *ustart*, their slopes *udotstart* and the required end point values *ue** depending on the chosen *u_type* or *ctrl_par_mode* and *ctrl_match*. The respective scaling factors **scale* and box constraints given by lower and upper bounds **lbound, *ubound* default values (1.0, −100, 100) if a *NULL* pointer is passed. Please note that all entries are indexed by $imsn = 0, \ldots, getTotalMSN() - 1$, but the vector at the last index is always 0 dimensional thus it efficiently has only $getTotalMSN() - 1$ elements.
- ```
  long setInitialParameters(
          DVec &pstart, DVec &pscale, DVec &plbound, DVec &pubound,
          DVecList &prstart, DVecList &prscale, DVecList &prlbound, DVecList &prubound
  );
  ```

  assigns initial input data for the global *pstart* and shooting node local model parameters *prstart*. The respective scaling factors **scale* and box constraints given by lower and upper bounds **lbound, *ubound* default values (1.0, −100, 100) if a *NULL* pointer is passed. Please note that all VecLists entries are indexed by $imsn = 0, \ldots, getTotalMSN() - 1$, i.e. are not separated into different stages.
- ```
  long setInitialStage(DVec &hstart, DVec &hscale, DVec &hlbound, DVec &hubound);
  ```

  assigns initial input data for the global model stage durations *hstart* as well as the respective scaling factors **scale* and box constraints given by lower and upper bounds **lbound, *ubound*. Default values are assigned for scaling, lower and upper bounds (1.0, −100, 100) if a *NULL* pointer is passed.
- ```
  long fixInitialStates(LVecList &sdfix);
  ```

  is used to fix certain differential states and remove the shooting nodes as free variables from optimization.
- ```
  long fixInitialControls(LVecList &ufix);
  ```

  is used to fix certain controls and remove the variables of the control parametrization as free variables from optimization.
- ```
  long fixInitialLocalParameters(long imos, long imsi, LVec &prfix);
  ```

  fixes the shooting node local parameters and removes them as free variables from optimization.
- ```
  long setInitialOfScale(double ofscale);
  ```

  assigns a scaling factor for value of the objective function of the OCP (1a).

- `long setInitialRHSScales(DVecList &arhsscales, DVecList &dipcscales, DVec &cipcscales);`

  assigns scaling factors *arhsscales* for the algebraic consistency conditions (9) as well as *dipcscales*, *cipcscales* for the decoupled and the coupled point constraints (10e). Please note that the VecList are indexed by $imsn = 0, \ldots, getTotalMSN() - 1$.

The forwarded calls interacting with the SQP solver of MUSCOD can be separated into methods related to the offline solution of an OCP, e.g.

- `long sqpInitialize(long guess_type, const char* restart_rel_path, const char* restart_name);`

  which initializes the NLP (10) data structure and where the discrete variable *guess_type* define if either all problem variables of the NLP have to specified or only some while the others are computed based on integration or interpolation, these variables can also be loaded from a binary file specified by *restart_rel_path* and *restart_name* of a previous run of the SQP solver,
- `long sqpSolve();`

  which starts the solution procedure of the SQP methods that runs until a solution is found (returns 0), i.e. a certain threshold of KKT tolerance is achieved, or a maximum number of iterations was reached as well as the respective error code < 0.

or related to the online solution of an OCP in the context of NMPC or MHE, e.g.

- `long nmpcInitialize(long guess_type, const char *restartpath, const char *restartname);`

  which fulfills the same functionality as *sqpInitialize*, but also prepares the QP (11) to be able to perform a call to *nmpcFeedback* directly afterwards,
- `long nmpcPrepare ();`

  evaluates all quantities required for the QP (11), which concludes a full integration of the DAE as well as the evaluation of all derivatives,
- `long nmpcFeedback (const double *initial_sd, const double *initial_pf, double *first_qc);`

  embeds the initial value and parameter estimates $\hat{y}_0$, $\hat{p}$ as in (3d),(3e) provided by the vector quantities *initial_sd*, *initial_pf* into the QP (11), which enter linearly, solves the QP and returns the piece-wise constant control of the first shooting interval *first_qc*
- `long nmpcShift (long shiftflag = 4);`

  implements different shifting strategies controlled by the discrete variable *shiftflag*, i.e. 0: shift controls, 1: also shift differential and algebraic states, 2: also shift evaluated function values, 3: also shift evaluated function derivatives and 4: also shift evaluated Hessian blocks. Please note that shifting is not required to solve NMPC problems, but can help in cases where shifting achieves a better initialization of the next subproblem.
- `long nmpcTransition ();`

  performs the NLP step for the nominal and dual variables
- ```
  long nmpcSimulate (
          long job, double timestep,
          const double *initial_sd, const double *initial_sa,
          const double *control, const double *param,
          double *final_sd, double *final_sa,
          double *final_sd_sd0, double *final_sd_q, double *final_sd_pf,
          double *final_sa_sd0, double *final_sa_q, double *final_sa_pf
  );
  ```

  can be used to run a forward simulation of the system dynamics for the current differential and algebraic initial values *initial_sd*, *initial_sa* on a time interval of length *timestep* using the provided controls *control* and parameters *param*. The following modes are supported via the discrete variable *job*: 0: Only carry out algebraic consistency iterations, 1: Also integrate and 2: Also compute sensitivity matrices, where the respective nominal results are provided via *final_sd*, *final_sa* and the sensitivity matrices have the subscripts *sd0*, *q*, *pf* to distinguish the computed derivatives.
  ```

All functions forwarded in the C/C++ facade are documented in-code using Doxygen [45], as shown in the example definition in Listing 1.

4.2.2 NMPC and MHE using the C/C++ Interface

Revisiting the benchmark example from Section 2.5, one way to implement this example in MUSCOD is to implement all required model functions in C in a shared library. Example code for the benchmark example is shown in Listing 5. The next step is to provide an initial guess for the NLP (10) data structure in the form of a `.dat` file, see Listing 6.

In Listing 9, example code is shown to demonstrate how to setup a NMPC problem using the C/C++ interface. Instead of having a monolithic executable, the C/C++ interface encapsulates everything in a MUSCOD object, to be instantiated first. The next step is to load the `.dat` file using that object via a call tos *loadFromDatFile*, which uses the values assigned by *setModelPathAndName* in the case a *NULL* pointer is provided. In the implementation of the benchmark example, an NMPC loop based on the real-time iteration as presented in § 3.4 is implemented. Here, MUSCOD is also used as a simulation tool, using the dynamics of the NMPC problem to provide a forward integration of the system over the sampling time of $0.05s$.

The benchmark example can serve as well to implement the combination of MHE and NMPC. Example code for the shared model library for the MHE problem is shown in Listing 8. The implementation shows, that a model parameter was added ($NP = 1$), which enters the right-hand side of the ODE (4b) and is to be estimated. The discrete least-squares objective *lsqfcn_mhe* , cf. (4a), is implemented node-wise and the current shooting node is retrieved using the *InfoPtr* instance. The function *inject_measurement* allows to interact with the measurements *meas_hs* and standard deviations *meas_ss* of the shared library when loaded. Please note that special care has to be taken when initializing of the MHE model to be consistent.

If a use-case demands more complicated linear algebra operations outside of the NMPC loop, we recommend to use a linear algebra library providing a C/C++ API, e.g. LIBLAC [35] or the more recent Eigen3 [24].

4.2.3 Python Interface using Cython

The high-level language of our choice is Python [46]. According to the TIOBE Index [38], it is the fifth amongst the most popular programming languages, trailing only Java, C, C++ and C#, and has found widespread adoption among scientists and engineeers. In particular, with the NumPy [44] and SciPy [27] packages, the Python ecosystem provides a rich functionality that is well suited for scientific computing and allows for convenient and interactive exploration and programming.

There are several ways of adding Python bindings to a C/C++ library. We opted in favor of using Cython [6], a universal programming language that is mainly compatible to Python. Cython code differs from Python code mainly in that variables have to be typed statically. It transforms into C or C++ code that may then be compiled into a library and imported into a CPython environment. This can be done in an automated and seamless way, such that the user's experience feels like writing Python code, with delayed execution due to compilation.

Functionality provided by C/C++ libraries can be accessed from Cython once function signatures are made accessible. This usually amounts to duplicating a C/C++ library header file in Cython syntax, in a so-called `.pxd` file. This mechanism allows to use functionality provided by any C, C++ or Fortran library from Python and is the way we

interfaced MUSCOD and Python. Other possibilites for interfacing are SWIG [5], as used e.g. by `CasADi` [2], using the foreign function interface `ffi` provided by the Python package `ctypes`, or the similar `cffi` package.

The functionality around MUSCOD and optimal control problems has been structured into three Python modules, namely `pymuscod`, `pymodel` and `pynmpc`.

- `pymodel` aims at replacing the C/C++ library that had to be used within MUSCOD to provide the functions f_j and g_j defining the right-hand side of the DAE constraint as well as the functions Δ_j, c_j and r_{ij} representing constraints and L_j, E_j constituting the objective of the optimal control problem. An example how this module can be used can be seen in the python version of the re-entry problem, however as most users decide to continue to use C/C++ model formulations to provide the functions to be evaluated, we will not describe this module in further detail.
- `pymuscod` is used to interface the functionality of MUSCOD and provides a class `MUSCOD` that can hold a MUSCOD instance with a particular optimal control problem.
- `pynmpc` provides the functionality of the NMPC module of MUSCOD and provides a class `NMPC` that is inherited from the `MUSCOD` class, allowing to access the additional functionality needed in `NMPC` applications.

The methods provided in the `MUSCOD` class of the `pymuscod` module closely follow the function signatures and nomenclature of the MUSCOD wrapper library described in §4.2.1, with some minor differences. They consist mainly in exchanging the LIBLAC data types by `numpy` arrays and by making use of the possibility in Python to declare standard values for arguments in Python and thus declaring some arguments to be optional. For these reasons we dispense of listing these class methods.

4.3 *Modeling in Python: from Convenience to Performance*

Formulating the re-entry problem with `pymodel` in Python, and using the `pymuscod` interface to set up an optimal control instanceleads to Python code as shown in Listings 6 and 6. Listing 6 shows how to formulate the model used in the optimal control problem with `pymuscod`. Listing 6 shows how to specify the numerical data initially required, and how to initiate the solution procedure. Finally, the listing also requests the optimal solution as computed by MUSCOD.

Formulating the model in pure Python as in listing 6 allows for easy interaction and exploration but comes with a huge speed penalty upon the integration. As a reference, 14.5 seconds CPU time are required, with only 0.2% of the CPU time being spent in computations *not* attributable to evaluation of model functions. We thus made it possible to also use a model provided via a plain C/C++ library, or one provided through a Cython implementation. For th example at hand, both require about 0.1 sec CPU time, with 10% CPU time being spent in computations *not* attributable to evaluation of model functions. Cython is an extension to Python that allows typed Python code to be compiled into C code, and to be loaded as a C library afterwards. It can be integrated in the Python toolchain, such that the compilation becomes possible without any further user interaction, and helps to ensures full performance in execution by getting rid of the overhead caused by the Python interpreter. This allows for a seamless workflow preserving the interactivity provided in the Python environment combining with the speed delivered by a compiled language.

Using Python to formulate the model and initial numerical values has several distinct advantages over the previously used approach that involved external libraries and `.dat`

files. For one, numerical data is now being exchanged using `numpy` arrays, a de-facto standard in scientific computing with Python that allows to analyze and modify these values in a convenient fashion. In particular, the intermediate step of generating a `.dat`-file can be disposed of. It is also possible to access computational results, including solution trajectories and solution data of the quadratic program directly in Python. Here, `numpy` arrays make post-processing, analysis, and plotting of these data sets significantly more convenient.

4.4 *NMPC using the Python interface*

Here, we briefly revisit the NMPC benchmark example from § 4.2.2 and present its implementation using the Python interface of MUSCOD. Listing 6 shows an example implementation of this NMPC problem. The implementation follows the same structure as presented in § 4.2.2. We briefly mention three differences to in contrast to Listing 9. First, note that the python interface provides a special NMPC interface encapsulated in class *pynmpc.NMPC*. The python interface provides named arguments to function calls that improves code readability, cf. *nmpcSimulate* in the code listing. The Python interface provides multiple return values, which is used *nmpcFeedback* in the listing to return both the error code and the feedback control.

The MHE example from § 4.2.2 can be implemented analogously.

4.5 *Postprocessing of Computational Results*

Both the C++ facade and the Python interface provide getter and setter routines for all numerical quantities of NLP (10) and QP (11) as obtained from the most recent SQP iterations carried out.

- `long getNMOS (), long getNMSN (long imos), long getNMSI (long imos), long getTotalNMSN (), long getNP (), long getNH (), ...`

  These functions are dimension getters, and are provided for all quantities as they arise in problem (1).
- `void getHF (double *hf), void getPF (double *pf), void getNodeSD (long imsn, double *sd), void getNodeQC (long imsn, double *sd), ...`

  These functions are value getters, and are provided to access the parts of the vector w^k of unknowns. These include the subset of free stage durations (hf), the subset of free model parameters (pf), the vector of differential states s_i at node i, the control parameters vector q_i on interval i, etc.
- Value setters to modify the SQP iterate w^k post iteration are provided in a similar fashion.

For trajectories discretized in time, the following interface functions are available:

- ```
  long getDifferentialTrajectory (long index, long msi, unsigned long *length,
                                  double *tdata, double *xddata);
  ```

  This function returns the discretized differential state trajectory $x_j(t)$ on $[t_i, t_{i+1}]$ in *xddata*, the discretization points in *tdata*, and the number of points in *length*. The trajectory index j is passed in *index*, the shooting interval index i in *msi*, and the maximum storable trajectory length in *length*.
- ```
  long getAlgebraicTrajectory (long index, long msi, unsigned long *length,
                               double *tdata, double *xddata);
  ```

  This function returns the discretized algebraic state trajectory $z_j(t)$ on $[t_i, t_{i+1}]$. The calling convention is identical to the one previously described.

Besides for postprocessing, conversion to user-defined data formats, and storage, these interface functions may be used for visualization of state and control trajectories. The Python interface provides identical functionality and users additionally benefit from the availability of convenient Python visualization packages, such as `matplotlib`.

- ```
  long getQPData (
      unsigned long *dimStep, unsigned long *dimConstraints, unsigned long *dimEquality,
      double *hessian, double *constraints,
      double *gradient, double *residual, double *lbound, double *ubound,
      long *idxEquality,
      double *step, double *multiplier, double *multiplierBounds
  );
  ```

  This function provides access to all quadratic programming (QP) data of the most recent SQP iteration. In detail, the QP dimensions are returned in the first three arguments. Then, access to dense column-wise matrix data for the Hessian $B(w^k)$ and the joint constraints matrix $(C(w^k)^T, D(w^k)^T)^T$ is possible through *hessian* and *constraints*. Vector data is accessible through *gradient* $(b(w^k))$, *residual* $((c(w^k)^T, d(w^k)^T)^T)$, and to lower and upper bounds on Δw^k through *lbound* and *ubound*. The index list *idxEquality* holds indices of equality constraints in *constraints* (rows) and *residual* (entries). Finally, the primal QP solution Δw^k is found in *step*, and the associated Lagrange multipliers for the constraints in *multiplier*, those for the bounds in *multiplierBounds*.

The QP data access function for example allows for analysis of local sensitivity, condition number, and well-posedness.

5. Current Use Cases

One key performance indicator are academic and industrial collaborations enabled or facilitated by the provided software framework. We found that the effort of providing a high-level language user-interface for both C++ and Python enabled also non-mathematicians, e.g. engineers with modeling and simulation knowledge, but little optimization background, to quickly use the package to solve optimal control problems on-line and off-line. The following section gives a summatory report of success stories eased by the the model-based optimization framework presented in this article.

5.1 *Exhaust Heat Recovery in Automotive Control*

In [25], the Python interface to MUSCOD is used to obtain energy optimal set-point change operations for an exhaust heat energy recovery system used to equip heavy duty trucks with an auxiliary green energy source. A C model is automatically generated from a Matlab Simulink description of the process, and is interfaced using the plain C model interface to MUSCOD.

5.2 *Hybrid Vehicle Temperature and Battery Management Control*

In [18], the Python and FMU interfaces to MUSCOD are used in an automotive control context. Optimizing controls for temperature and battery management of a hybrid electric vehicle are obtained by interfacing a Modelical model to MUSCOD via the FMU interface.
  ```

5.3 *Model-based Predictive Control of Adsorption Chillers*

The Python and FMU interfaces to MUSCOD are also used for thermodynamical engineering by [4]. A Modelica model of an adsorption chilling process in interfaced to MUSCOD using the FMU model interface. The Python interface is then used to implement a nonlinear model predictive control (NMPC) loop. An optimizing control task given the temperature profile of typical solar daylight cycle is solved in real-time.

5.4 *Cruise Control and Driving Strategies for Hybrid Vehicles*

Energy-optimal driving strategies for a hybrid electric vehicle are computed in [41]. Computations were performed off-line for the so-called New European Driving Cycle using the Python interface, and the FMU interface to MUSCOD was used for a Modelica model of the vehicle's thermal and mechanical components.

5.5 *NMPC and Reinforcement Learning in Humanoid Robotics*

Following the assumption that human locomotion is optimal, optimal control and NMPC are therefore well-suited methods to analyze human motions and/or generate motions for both humanoid robots. Optimal control with MUSCOD was applied in two different approaches [32] and [11] to successfully enable whole-body multi-contact for the humanoid robot HRP-2. Especially the combination with the model-free approach of Reinforcement Learning is of special interest because of their complementary approach to NMPC. While the combination of both approaches is under current research, the refactoring of MUSCOD documented by this article allowed its implementation in a joint software package [10] and therefore an in-depth comparison of them. The resulted comprehensive comparison study of NMPC and Reinforcement Learning [31].

5.6 *Convenient and Market-Ready Graphical User Interfaces*

Development efforts spent on the C++ and Python interfaces as well as on the FMU capabilities of MUSCOD also opened up opportunities for joint collaborations with the Aachen based company TLK Energy [23]. There, convenient and accessible graphical user interfaces for marked-ready optimal control solutions to specific segments and domains of the process engineering industry could be developed successfully, based on the Python interface.

6. Conclusion and Outlook

A monolithically designed software package for direct optimal control, MUSCOD, that was originally designed to solve a fixed class of problem has been refactored and equipped with high-level language interfaces in order to serve the demands witnessed during academic and industry cooperations. Low-level access to MUSCOD always provided flexibility and ease of developing new mathematical and numerical algorithms through modification and extension of existing ones, a necessity in numerical mathematical research. Complementing this, the newly created interfaces offer advantages for non-mathematics researchers that come thanks to clearer interfaces and template-based approaches to modeling offered by visual tools such as Matlab/Simulink and Modelica. We have demonstrated this ability for

MUSCOD at the example of FMU, Python, and C++, and have reported on a number of real-world use cases that were made possible by, or benefitted from, the development efforts reported in this article. In the future, lessons learned from the refactoring and interfacing development efforts should be taken into account when approaching the initial design and implementation phase of mathematical, numerical, and optimization software packages to come.

Acknowledgments

The authors were supported by DFG Graduate School 220 (Heidelberg Graduate School of Mathematical and Computational Methods for the Sciences) funded by the German Excellence Initiative. F. Lenders acknowledges funding by the German National Academic Foundation. C. Kirches was supported by the German Federal Ministry of Education and Research program "Mathematics for Innovations in Industry and Service 2013–2016" grant n^{o} 05M2013-GOSSIP. M. Kudruss and C. Kirches were supported by the European Union within the 7th Framework Programme under Grant Agreement n^{o} 611909 in the scope of the *KoroiBot* project.

References

[1] J. Albersmeyer. *Adjoint based algorithms and numerical methods for sensitivity generation and optimization of large scale dynamic systems.* PhD thesis, Heidelberg University, 2010.

[2] J. Andersson. *A General-Purpose Software Framework for Dynamic Optimization.* PhD thesis, Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium, October 2013.

[3] Modelica Association. Modelica. `http://www.modelica.org` (valid 2016-11-15).

[4] N. Baumgärtner. *Implementation and experimental validation of a model predictive control for adsorption heat pumps.* Master's thesis, RWTH Aachen University, 2016.

[5] D.M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996-Volume 4*, page 15. USENIX Association, 1996.

[6] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, March-April 2011.

[7] J.T. Betts, S.K. Eldersveld, and W.P. Huffman. Sparse nonlinear programming test problems (Release 1.0). Tech. rep. BCSTECH-93-074, Boeing Computer Services, 1993.

[8] H.G. Bock. Numerical treatment of inverse problems in chemical reaction kinetics. In K.H. Ebert, P. Deuflhard, and W. Jäger, editors, *Modelling of Chemical Reaction Systems*, volume 18 of *Springer Series in Chemical Physics*, pages 102–125. Springer, Heidelberg, 1981.

[9] H.G. Bock and K.J. Plitt. A Multiple Shooting algorithm for direct solution of optimal control problems. In *Proceedings of the 9th IFAC World Congress*, pages 242–247, Budapest, 1984. Pergamon Press.

[10] W. Caarls. Generic Reinforcement Learning Library, 2015. `https://github.com/wcaarls/grl` (valid 2016-11-15).

[11] J. Carpentier, S. Tonneau, M. Naveau, O. Stasse, and N. Mansard. A versatile and efficient pattern generator for generalized legged locomotion. In *2016 IEEE International Conference on Robotics and Automation*, 2016.

[12] M. Diehl. *Real-Time Optimization for Large Scale Nonlinear Processes.* PhD thesis, Heidelberg University, 2001.

[13] M. Diehl, H.G. Bock, and J.P. Schlöder. A real-time iteration scheme for nonlinear optimization in optimal feedback control. *SIAM Journal on Control and Optimization*, 43(5):1714–1736, January 2005.

[14] M. Diehl, D. Leineweber, A.A.S. Schäfer, C. Hoffmann, C. Kirches, A. Potschka, S. Sager, L. Wirsching, M. Kudruss, C. Leidereiter, and F. Lenders. MUSCOD-II Users' Manual. IWR

preprint, Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University, Im Neuenheimer Feld 205, 69120 Heidelberg, Germany, 2016.

[15] Process Systems Enterprise. gPROMS, 1997–2015. `http://www.psenterprise.com/gproms` (valid 2016-11-15).

[16] H.J. Ferreau. *Model Predictive Control Algorithms for Applications with Millisecond Timescales*. PhD thesis, K.U. Leuven, 2011.

[17] H.J. Ferreau. qpOASES user's manual. *Optimization in Engineering Center (OPTEC) and Department of Electrical Engineering, KU Leuven*, 2011.

[18] T. Fischer, F. Götz, L.F. Berg, H.-P. Kollmeier, and F. Gauterin. Model-based development of a holistic thermal management system for an electric car with a high temperature fuel cell range extender. In *Proceedings of the 11th International Modelica Conference, Versailles, France*, pages 127–133, 2015.

[19] R. Fourer, D.M. Gay, and B.W. Kernighan. A Modeling Language for Mathematical Programming. *Management Science*, 36:519–554, 1990.

[20] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002.

[21] R. Franke. Omuses — a tool for the optimization of multistage systems, and HQP — a solver for sparse nonlinear optimization. Technical report, TU Ilmenau, 1998.

[22] P.E. Gill, W. Murray, M.A. Saunders, and M.H. Wright. User's Guide for SOL/QPSOL: A Fortran Package for Quadratic Programming. Technical report, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1982.

[23] TLK Energy GmbH, 2016. Website: `http://www.tlk-energy.com` (valid 2016-11-14).

[24] G. Guennebaud and B. Jacob et al. Eigen v3, 2010. `http://eigen.tuxfamily.org` (valid 2016-11-15).

[25] E. Guerrero, C. Kirches, and J.P. Schlöder. Nonlinear optimal control of a heavy duty truck exhaust heat recovery system. In *Modeling, Simulation and Optimization of Complex Processes. Proceedings of the 6th International Conference on High Performance Scientific Computing (HPSC), Hanoi, Vietnam*. Springer Verlag, 2015. (accepted).

[26] B. Houska, H.J. Ferreau, and M. Diehl. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

[27] E. Jones, T. Oliphant, and P. Peterson et al. SciPy: Open source scientific tools for Python, 2001–. `http://www.scipy.org/` (valid 2016-11-15).

[28] C. Kirches. Fast Numerical Methods for Mixed-Integer Nonlinear Model-Predictive Control. In H.G. Bock, W. Hackbusch, M. Luskin, and R. Rannacher, editors, *Advances in Numerical Mathematics*. Springer Vieweg, Wiesbaden, July 2011. ISBN 978-3-8348-1572-9.

[29] C. Kirches and S. Leyffer. TACO – A Toolkit for AMPL Control Optimization. *Mathematical Programming Computation*, 5(2):227–265, 2013.

[30] S. Körkel. *Numerische Methoden für Optimale Versuchsplanungsprobleme bei nichtlinearen DAE-Modellen*. PhD thesis, Heidelberg University, 2002.

[31] I. Koryakovskiy, M. Kudruss, R. Babuška, W. Caarls, C. Kirches, K. Mombaur, J.P. Schlöder, and H. Vallery. Benchmarking model-free and model-based optimal control for bipedal locomotion. 2016. (submitted to Robotics and Autonomous Systems).

[32] M. Kudruss, M. Naveau, O. Stasse, N. Mansard, C. Kirches, P. Souerès, and K. Mombaur. Optimal control for multi-contact, whole-body motion generation using center-of-mass dynamics for multi-contact situations. In *Proceedings of the 2015 IEEE-RAS International Conference on Humanoid Robots (Humanoids 2015)*, pages 684–689, Seoul, Korea, November 2015.

[33] P. Kühl, M. Diehl, T. Kraus, J.P. Schlöder, and H.G. Bock. A real-time algorithm for moving horizon state and parameter estimation. *Computers & Chemical Engineering*, 35(1):71–83, January 2011.

[34] D.B. Leineweber, I. Bauer, H.G. Bock, and J.P. Schlöder. An efficient multiple shooting based reduced SQP strategy for large-scale dynamic process optimization. Part 1: theoretical aspects. *Computers & Chemical Engineering*, 27(2):157–166, 2003.

[35] D.B. Leineweber and J.A. Jost. LIBLAC — structured data types and basic operations for numerical linear algebra in an ANSI C/Fortran 77 environment. Technical Report 96-56, Heidelberg, 1996.

[36] P. Martinon. BOCOP. `http://www.bocop.org` (valid 2016-11-15).

[37] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Verlag, Berlin Heidelberg New York, second edition, 2006. ISBN 0-387-30303-0 (hardcover).

[38] TIOBE Index November, 2016. `http://www.tiobe.com/tiobe-index/` (valid 2016-11-15).

[39] Tomlab Optimization. PROPT. `http://www.tomdyn.com` (valid 2016-11-15).
[40] M.A. Patterson and A.V. Rao. GPOPS-II: A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp-Adaptive Gaussian Quadrature Collocation Methods and Sparse Nonlinear Programming. *ACM Trans. Math. Softw.*, 41(1):1–37, October 2014.
[41] P. Petr, C. Schröder, J. Köhler, and M. Gräber. Optimal Control of Waste Heat Recovery Systems Applying Nonlinear Model Predictive Control (NMPC). In *Proceedings of the 3rd International Seminar on ORC Power Systems (ASME ORC 2015)*, October 2015.
[42] L. Schork. A parametric active set method for general quadratic programming. Master's thesis, Heidelberg University, 2015.
[43] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer New York, New York, NY, 2002.
[44] S. van der Walt, S.C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 2011.
[45] D. van Heesch. Doxygen, 1997–2016. `http://www.doxygen.org` (valid 2016-11-15).
[46] G. van Rossum. Python reference manual. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1995.
[47] O. von Stryk. User's guide for DIRCOL (Version 2.1): A direct collocation method for the numerical solution of optimal control problems. Technical report, Technische Universität München, Germany, 1999.

Appendix

```
/** @brief Initialize the solver for a new run. Call this after the model
 *         has been initialized, and whenever dimensions have changed
 *  @param guess_type  How to obtain the intial guess of the state trajectory
 *                     0: all values specified in sd, sa
 *                     1: interpolate sd(0), sa(0) and sd(N), sa(N)
 *                     2: integrate starting with sd(0), sa(0)
 *                     3: cold start from restart file
 *                     4: warm start from restart file
 *  @param restart_rel_path  New relative path to the restart file, or 0 to
 *                           reuse the path specified in @a setModelPathAndName.
 *  @param restart_name  Name of the restart file, without extension, or 0 to
 *                       reuse the name specified in @a setModelPathAndName.
 *  @returns 0 if successful
 *  @note  Finalization is done automatically; there is no sqpFinalize.
 */
long sqpInitialize (
  long guess_type, const char *restart_rel_path = 0, const char *restart_name = 0
);
```

Listing 1: In-code documentation of the C/C++ facade using Doxygen.

```
// global FMU component instance
fmi2Component fmu;

// proxy class to trigger automatic instatiation of the FMU
class InstantiateFMU {
public:
  InstantiateFMU ();
  ~InstantiateFMU ();
};

// constructor to instantiate and initialize the FMU
InstantiateFMU::InstantiateFMU () {
  fmu = fmi2Instantiate (instanceName, fmi2ModelExchange, fmuGUID, fmuResourceLocation,
                         &callbacks, fmi2False, fmi2False);
  fmi2SetupExperiment (fmu, fmi2False, 0.0, 0.0, fmi2False, 100.0);
  fmi2EnterInitializationMode (fmu);
  fmi2ExitInitializationMode (fmu);
  fmi2EnterContinuousTimeMode (fmu);
}

// destructor to unload the FMU
InstantiateFMU::~InstantiateFMU () {
    fmi2FreeInstance (fmu);
}

// instance of the proxy class. Upon library loading, this triggers the instatiation of the FMU
InstantiateFMU instantiateFMU;
```

Listing 2: Automatic instantiation and initialization of an FMU as part of a generic MUSCOD dynamic model library for interfacing with FMUs.

```
// reference integer values for a control in R^2
const fmiValueReference uRef[NU] = {352321536, 352321537};
```

Listing 3: FMU variables, control components in this listing, are identified by reference integer values.

```
void ffcn (
  double *t, double *xd, double *xa, double *u, double *p, double *rhs, double *rwh, long *iwh,
  InfoPtr *info
){
  fmi2SetTime (fmu , *t); // set time
  fmi2SetReal (fmu, uRef, NU, u); // set controls
  fmi2SetReal (fmu, pRef, NP, p); // set parameters
  fmi2SetContinuousStates (fmu, xd, NXD); // set differential states
  fmi2GetDerivatives (fmu, rhs, NXD); // get time derivatives
}
```

Listing 4: Generic MUSCOD evaluation function for a differential right hand side provided by an FMU.

```cpp
#include <cmath> // for computing sqrt of weights
#include "def_usrmod.hpp" // MUSCOD definitions

// model dimensions
static const long NMOS = 1;
static const long NP   = 0;
static const long NRC  = 0;
static const long NRCE = 0;

static const long NXD  = 1;
static const long NXA  = 0;
static const long NU   = 1;
static const long NPR  = 0;

// weights for least-squares objective
const double w0 = 10.0;
const double w1 =  1.0;
const double sw0 = sqrt(w0);
const double sw1 = sqrt(w1);

// ordinary differential equation
static void ffcn(double *t, double *xd, double *xa, double *u,
  double *p, double *rhs, double *rwh, long *iwh, InfoPtr *info)
{
    rhs[0] = (1 + xd[0]) * xd[0] + u[0];
}

// least-squares function
static void lsqfcn(double *ts, double *sd, double *sa, double *u,
  double *p, double *pr, double *res, long *dpnd, InfoPtr *info)
{
    if (*dpnd) { *dpnd = RFCN_DPND(0, *sd, 0, *u, 0, 0); return; }
    res[0] = sw0 * sd[0];
    res[1] = sw1 *  u[0];
}

extern "C" void def_model(void)
{
    // define model dimensions
    def_mdims(NMOS, NP, NRC, NRCE);

    // define model stage
    def_mstage(
        0, // stage number
        NXD, NXA, NU, // variable dimensions
        NULL, NULL, // objective functions
        0, 0, 0, NULL, ffcn, NULL, // right-hand side
        NULL, NULL
    );

    // define least-squares objective
    def_lsq(0, "c", 0, NXD+NU, lsqfcn);

    // define end-point constraint
    def_mpc(0, "End Point", NPR, NXD, NXD, NULL, NULL);
}
```

Listing 5: Example legacy C code for the NMPC benchmark example, to be compiled as a shared library.

```
; # of multiple shooting intervals on each model stage
nshoot
0: 30

; model stage duration start values, scale factors, and bounds
h
0: 3

h_sca
0: 3

h_min
0: 3.0

h_max
0: 3.0

h_fix
0: 1

; specification mode for differential state variable start values
s_spec
2

; differential state start values, scale factors, and bounds
sd(0,0)
0: 0.0

sd_sca(*,*)
0: 1.0

sd_min(*,*)
0: -1.0

sd_max(*,*)
0: 1.0

; control parameterization types
u_type(*)
0: 0

; control start values, scale factors, and bounds
u(*,*)
0: 0

u_sca(*,*)
0: 1.0

u_min(0,*)
0: -1

u_max(0,*)
0: 1
```

Listing 6: Example legacy MUSCOD `.dat` file for the NMPC benchmark example.

```cpp
#include "wrapper.hpp" // MUSCOD C/C++ facade

int main(int argc, char const *argv[]) {
    // instantiate MUSCOD
    MUSCOD muscod;

    // load model library and DAT file
    const char* modelpath = ".";
    const char* modelname = "nmpc";

    muscod.setModelPathAndName(modelpath, modelname);
    muscod.loadFromDatFile(NULL, NULL);

    // initialize the SQP method in NMPC mode
    muscod.nmpcInitialize(0, NULL, NULL);

    // define initial value and placeholder for feedback control
    double initial_sd[] = {0.01}; // initial value
    double first_qc[] = {0.0}; // feedback control
    double final_sd[] = {0.0}; // simulation result

    const int n_iter = 10;
    for (int i_iter = 0; i_iter < n_iter; ++i_iter) {
        muscod.nmpcFeedback(&initial_sd[0], &first_qc[0]);
        muscod.nmpcShift(1);
        muscod.nmpcTransition();
        muscod.nmpcPrepare();
        muscod.nmpcSimulate(
            1,                  // simulate, no derivatives
            0.05,               // time interval in [s]
            &initial_sd[0],     // take current initial value
            NULL,               // no algebraic states
            &first_qc[0],       // use feedback control
            NULL,               // no parameters involved
            &final_sd[0], NULL, // final_sd, final_sa
            NULL, NULL, NULL,   // final_sd_sd0, final_sd_q, final_sd_pf
            NULL, NULL, NULL    // final_sa_sd0, final_sa_q, final_sa_pf
        );

        // overwrite initial value provided to NMPC
        initial_sd[0] = final_sd[0];
    }
    return 0;
}
```

Listing 7: Example code showing the implementation of a MUSCOD NMPC loop using the C/C++ interface.

```cpp
#include "def_usrmod.hpp" // MUSCOD definitions

const long NMNS = 4;
const long NMOS = 1;
const long NP   = 1;
const long NRC  = 0;
const long NRCE = 0;

const long NXD  = 1;
const long NXA  = 0;
const long NU   = 1;
const long NPR  = 0;

// measurement horizon
// measurements
double meas_hs[] = {5.0000000000000000E-01, 4.8870304458018765E-01,
    4.7438241357188599E-01, 4.5629163373509302E-01, 4.3353797161859531E-01};
// measurement errors
double meas_ss[] = {0.01, 0.01, 0.01, 0.01, 0.01};

// declare extern to prevent name mangling
extern "C" {
    void inject_measurement (double* hs, double* ss){
        for (int i = 0; i < NMNS; ++i) {
            meas_hs[i] = meas_hs[i+1];
            meas_ss[i] = meas_ss[i+1];
        }
        meas_hs[NMNS] = hs[0];
        meas_ss[NMNS] = ss[0];
    }
}

const unsigned int LSQFCN_MHE_NE = NXD;
void lsqfcn_mhe (double *ts, double *sd, double *sa, double *u,
    double *p, double *pr, double *res, long *dpnd, InfoPtr *info
) {
    if (*dpnd) {*dpnd = RFCN_DPND(0, *sd, 0, 0, 0, 0); return;}
    double hs = meas_hs[info->cnode];
    double ss = meas_ss[info->cnode];
    res[0] = (sd[0] - hs) / ss;
}

static void ffcn(double *t, double *xd, double *xa, double *u,
  double *p, double *rhs, double *rwh, long *iwh, InfoPtr *info
) {
    rhs[0] = p[0]*(1 + xd[0]) * xd[0] + u[0];
}

extern "C" void def_model(void) {
    def_mdims(NMOS, NP, NRC, NRCE);
    def_mstage(
        0, // stage number
        NXD, NXA, NU, // variable dimensions
        NULL, NULL, // objective functions
        //mfcn_e, lfcn, // objective functions
        0, 0, 0, NULL, ffcn, NULL, // rhs
        NULL, NULL
    );

    def_lsq(0, "*", NPR, LSQFCN_MHE_NE, lsqfcn_mhe);
}
```

Listing 8: Example C code for the NMPC benchmark example implementing a Moving Horizon Estimator, to be compiled as a shared library.

```cpp
#include "wrapper.hpp" // MUSCOD C/C++ facade
#include <dlfcn.h> // to handle dynamic libraries

// handles for the problem library and its symbols
void* model_so_handle; // handle to problem library
void (*so_inject_measurement) (double* hs, double* ss);

int main(int argc, char const *argv[]) {
    MUSCOD nmpc_muscod, mhe_muscod; // instantiate MUSCOD

    // load model library and DAT file
    const char* modelpath = "."; // relative model path
    const char* nmpc_modelname = "nmpc";
    nmpc_muscod.setModelPathAndName (modelpath, nmpc_modelname);
    nmpc_muscod.loadFromDatFile(NULL, NULL);
    nmpc_muscod.nmpcInitialize(0, NULL, NULL);

    const char* mhe_modelname = "mhe";
    mhe_muscod.setModelPathAndName (modelpath, mhe_modelname);
    mhe_muscod.loadFromDatFile(NULL, NULL);
    mhe_muscod.nmpcInitialize(0, NULL, NULL);

    // load model library and sympbol from it for MHE only
    const char* so_path = "./libmhe.so";
    model_so_handle = dlopen (so_path, RTLD_NOW|RTLD_GLOBAL);
    const char* symbol_name = "inject_measurement";
    so_inject_measurement = (void (*)(double*, double*))
        dlsym (model_so_handle, symbol_name);

    // define initial value and placeholder for feedback control
    double initial_sd[] = {0.43}; // initial value estimate
    double model_pf[] = {1.00}; // global parameter estimate
    double first_qc[] = {0.0}; // feedback control
    double real_pf[] = {1.20}; // real parameter for simulation
    double final_sd[] = {0.0}; // simulation result

    // retrieve initial value and model parameter estimate
    mhe_muscod.getPF(&model_pf[0]);
    mhe_muscod.getNodeSD(4, &initial_sd[0]);

    for (int i_iter = 0; i_iter < 10; ++i_iter) {
        // NMPC
        nmpc_muscod.nmpcFeedback(&initial_sd[0], &model_pf[0], &first_qc[0]);
        nmpc_muscod.nmpcShift(1);
        nmpc_muscod.nmpcTransition();
        nmpc_muscod.nmpcPrepare();

        nmpc_muscod.nmpcSimulate(1, 0.1, &initial_sd[0], NULL,
            &first_qc[0], &real_pf[0], &final_sd[0], NULL,
            NULL, NULL, NULL, NULL, NULL, NULL);

        // MHE, implemented *not* using the real-time iteration scheme
        double ss[] = {0.01};
        so_inject_measurement(&final_sd[0], &ss[0]);
        mhe_muscod.nmpcPrepare();
        mhe_muscod.nmpcFeedback(NULL, NULL, NULL);
        mhe_muscod.nmpcTransition();
        mhe_muscod.getPF(&model_pf[0]);
        mhe_muscod.getNodeSD(4, &initial_sd[0]);
    }
    return 0;
}
```

Listing 9: Example code implementing NMPC and MHE using the C/C++ interface.

```
import pymuscod, pymodel
import numpy as np

##########################
# Build up reentry model #
##########################

# constants
beta = 4.26
g    = 3.2172E-4
R    = 209.0
Sm   = 53200.0
rho0 = 2.704E-3

# lagrange objective term 10v^3*sqrt(rho)
def lfcn(t, xd, xa, u, p, lval, info):
    lval[0] = 10.0*xd[0]*xd[0]*xd[0]*np.sqrt(rho0*np.exp(-beta*R*xd[2]))

# rhs f of ODE y(t), = f(t,y(t),u(t))
def ffcn(t, xd, xa, u, p, rhs, info):
    rho    = rho0*np.exp(-beta*R*xd[2])
    cw     = 1.174 - 0.9*np.cos(u[0])
    ca     = 0.6*np.sin(u[0])
    rhs[0] = - 0.5*Sm*rho*xd[0]*xd[0]*cw - g*np.sin(xd[1])/(1.0+xd[2])/(1.0+xd[2])
    rhs[1] = 0.5*SM*rho*xd[0]*ca + xd[0]*np.cos(xd[1])/R/(1.0+xd[2]) \
             - g*np.cos(xd[1])/xd[0]/(1.0+xd[2])/(1.0+xd[2])
    rhs[2] = xd[0]*np.sin(xd[1])/R

# initial value constraint y(0) = y_0
def rdfcn_s(ts, sd, sa, u, p, pr, res, info):
    res[0] = sd[0] - 0.36
    res[1] = sd[1] + 8.1*np.pi/180.0
    res[2] = sd[2] - 4.0/R

# terminal constraint y(T) = y_T
def rdfcn_e(ts, sd, sa, u, p, pr, res, info):
    res[0] = sd[0] - 0.27
    res[1] = sd[1]
    res[2] = sd[2] - 2.5/R

# dynamic requires one model stage with
# two point constraints for initial and terminal point
mstage      = pymodel.Modelstage(nxd=3, nu=1, lfcn=lfcn, ffcn=ffcn)
mstage.ipcs = [
    pymodel.InteriorPointConstraint(scope='s', nrd=3, nrde=3, rdfcn=rdfcn_s),
    pymodel.InteriorPointConstraint(scope='e', nrd=3, nrde=3, rdfcn=rdfcn_e) ]
# mark that model stage constraints only depend on differential state
mstage.ipcs[0].rdfcndpnd['sd'] = True; mstage.ipcs[1].rdfcndpnd['sd'] = True

# setup model consisting of this modelstage and initialize muscod with it
model = pymodel.Model(modelstages=[mstage])
```

Listing 10: Part I of python script to run re-entry example, here the model with the functions defining dynamics and constraints are defined and set up.

```
m = pymuscod.MUSCOD(modelpath='.', model=model)

# Choosing libraries
m.libhessian = 'hess_finitediff'  # finite difference hessian approximation
m.libsolve   = 'solve_tbox'       # trust region globalization
m.libcond    = 'cond_std'         # standard condensing
m.libtchk    = 'tchk'             # standard termination criterion
m.libmssqp   = 'mssqp_standard'   # standard sqp solver
m.libeval    = 'eval_ind'         # standard evaluation module
m.libqps     = 'qps_qpopt'        # QP solver QPOPT
m.libplot    = 'plot_noplot'      # no muscod internal plotting routine
m.libind     = ['ind_rkf45']      # Runge-Kutta 4/5 integrator for nonstiff problem

# Setting Up Problem Specifications
# # of differential and algebraic states, controls and global and local parameters
nxd = 3; nxa = 0; nu = 1; npp = 0; npr = 0

# # of multiple shooting intervals and nodes on each model stage
nmsi  = np.array([6], dtype=np.int); nmsi  = np.array([7], dtype=np.int)

#  control parametrization type
u_type = [np.array([1], dtype=int)]

#  initialize the model and all data structures
robSd0    = np.zeros([nxd],       dtype=np.int)
robP      = np.zeros([npp],       dtype=np.int)
stageFix  = np.zeros([nmsi.size], dtype=np.int)
parFix    = np.zeros([npp],       dtype=np.int)
ctrlMatch = [ np.zeros([nu],      dtype=np.int) ] for imsi in range(nmsi.size) ]
numClsq   = np.zeros([nmsi[0]],   dtype=np.int)
m.initializeModel(nmsi, robSd0, robP, stageFix, parFix, u_type, ctrlMatch, numClsq)

#  multiple shooting grids on model stages
m.setInitialGrid([np.array([0.0, 0.25, 0.375, 0.5, 0.675, 0.75, 1.0], dtype=float)])

#  specification mode for differential state variable start values
#  initial states
dstart  = [ [np.zeros([nxd], dtype=float) for imsi in range(nmsn[0])] ]
dstart[0][0]  = np.array([0.36, -0.1414, 0.01914], dtype=float)
dstart[0][-1] = np.array([0.27,  0.0,    0.01196], dtype=float)
astart  = [ [np.empty([nxa], dtype=float) for imsi in range(nmsn[0])] ]

m.setInitialStates(1, dstart, dscale, dlbound, dubound, astart, ascale, albound, aubound)

#  initial controls, parameters, model stage duration
ustart  = [ [np.array([  .5], dtype=float) for imsi in range(nmsi[0])] ]
m.setInitialControls(ustart, uscale, ulbound, uubound)
pstart = np.empty([npp]); prstart = [ [np.empty([npr]) for imsi in range(nmsn[0])] ]
m.setInitialParameters(pstart, pscale, plbound, pubound, prstart, prscale, prlbound, prubound)
m.setInitialStage( hstart = np.array([230.0], dtype=float) )

# post process model initilization
m.postProcess()

m.solve()
```

Listing 11: Part II of python script to run re-entry example, here a muscod instance to hold the optimal control problem is created and numerical data is specified. Finally the optimal control problem is controlled. For brevity we omitted from the code the specification scale factors and lower and upper bounds which is similar to that of start values.

```python
import os
import numpy as np
# MUSCOD NMPC interface
import pynmpc

# load lib<muscod_problem>.so
path = os.path.abspath(os.path.dirname(__file__))

# instantiate MUSCOD
muscod = pynmpc.NMPC(path, 'nmpc')

# load model library and DAT file
muscod.loadFromDatFile()

# initialize the SQP method in NMPC mode
muscod.nmpcInitialize(guess_type=0)

# define initial value and placeholder for feedback control
initial_sd = np.array([0.5], dtype=np.float)  # initial value
first_qc   = np.array([0.0], dtype=np.float)  # feedback control

for ii in range(10):
    __, first_qc = muscod.nmpcFeedback(initial_sd)
    muscod.nmpcShift(1)
    muscod.nmpcTransition()
    muscod.nmpcPrepare()

    final_sd = muscod.nmpcSimulate(
        job=1, timestep=0.05, initial_sd=initial_sd, initial_sa=np.zeros(0),
        control=first_qc, param=np.zeros(0)
    )

    # overwrite initial value provided to NMPC
    initial_sd[...] = final_sd
```