8
Numerical Optimal Control

8.1 Introduction

Numerical optimal control methods are at the core of every model predictive control implementation, and algorithmic choices strongly affect the reliability and performance of the resulting MPC controller. The aim of this chapter is to explain some of the most widely used algorithms for the numerical solution of optimal control problems. Before we start, recall that the ultimate aim of the computations in MPC is to find a numerical approximation of the optimal feedback control $u^0(x_0)$ for a given current state x_0. This state x_0 serves as initial condition for an optimal control problem, and $u^0(x_0)$ is obtained as the first control of the trajectory that results from the numerical solution of the optimal control problem. Due to a multitude of approximations, the feedback law usually is not exact. Some of the reasons are the following.

- The system model is only an approximation of the real plant.
- The horizon length is finite instead of infinite.
- The system's differential equation is discretized.
- The optimization problem is not solved exactly.

While the first two of the above are discussed in Chapters 2 and 3 of this book, the last two are due to the numerical solution of the optimal control problems arising in model predictive control and are the focus of this chapter. We argue throughout the chapter that it is not a good idea to insist that the finite horizon MPC problem shall be solved exactly. First, it usually is impossible to solve a simulation or optimization problem without any numerical errors, due to finite precision arithmetic and finite computation time. Second, it might not even be desirable to solve the problem as exactly as possible, because the necessary computations might lead to large feedback delays or an excessive use of CPU resources. Third, in view of the other errors that are necessarily introduced in the modeling process and in the MPC problem

formulation, errors due to an inexact numerical solution do not significantly change the closed-loop performance, at least as long as they are smaller than the other error sources. Thus, the optimal choice of a numerical method for MPC should be based on a trade-off between accuracy and computation time. There are, however, tremendous differences between different numerical choices, and it turns out that some methods, compared to others, can have significantly lower computational cost for achieving the same accuracy. Also, reliability is an issue, as some methods might more often fail to find an approximate solution than other methods. Thus, the aim of this chapter is to give an overview of the necessary steps toward the numerical solution of the MPC problem, and to discuss the properties of the different choices that can be made in each step.

8.1.1 Discrete Time Optimal Control Problem

When working in a discrete time setting, the MPC optimization problem that needs to be solved numerically in each time step, for a given system state x_0, can be stated as follows. For ease of notation, we introduce the sequence of future control inputs on the prediction horizon, $\mathbf{u} := (u(0), u(1), \ldots, u(N-1))$, as well as the predicted state trajectories $\mathbf{x} := (x(0), x(1), \ldots, x(N))$.

$$\begin{aligned}
\underset{\mathbf{x},\,\mathbf{u}}{\text{minimize}} \quad & \sum_{k=0}^{N-1} \ell(x(k), u(k)) + V_f(x(N)) && && \text{(8.1a)}\\
\text{subject to} \quad & x(0) = x_0 && && \text{(8.1b)}\\
& x(k+1) = f(x(k), u(k)), && k = 0, 1, \ldots, N-1 && \text{(8.1c)}\\
& (x(k), u(k)) \in \mathbb{Z}, && k = 0, 1, \ldots, N-1 && \text{(8.1d)}\\
& x(N) \in \mathbb{X}_f && && \text{(8.1e)}
\end{aligned}$$

We call the above optimization problem $\mathbb{P}_N(x_0)$ to indicate its dependence on the parameter x_0, and denote the resulting optimal value function by $V_N(x_0)$. The value function $V_N(x_0)$ is mostly of theoretical interest, and is in practice computed only for those values of x_0 that actually arise in the MPC context. In this chapter, we are mostly interested in fast and efficient ways to find an optimal solution, which we denote by $(\mathbf{x}^0(x_0), \mathbf{u}^0(x_0))$. The solution need not be unique for a given problem $\mathbb{P}_N(x_0)$, and in a mathematically correct notation one could only define the set $S^0(x_0)$ of all solutions to $\mathbb{P}_N(x_0)$. Usually one tries to ensure by a proper formulation that the MPC optimization

problems have unique solutions, however, so that the set of solutions is a singleton, $S^0(x_0) = \{(\mathbf{x}^0(x_0), \mathbf{u}^0(x_0))\}$.

A few remarks are in order regarding the statement of the optimization problem (8.1a)-(8.1e). First, as usual in the field of optimization, we list the optimization variables of problem $\mathbb{P}_N(x_0)$ below the word "minimize." Here, they are given by the sequences $\mathbf{x}$ and $\mathbf{u}$. The constraints of the problem appear after the keywords "subject to" and restrict the search for the optimal solution. Let us discuss each of them briefly: constraint (8.1b) ensures that the trajectory $\mathbf{x} = (x(0), \dots)$ starts at x_0, and uniquely determines $x(0)$. Constraints (8.1c) ensure that the state and control trajectories obey the system dynamics for all time steps $k = 0, \dots, N-1$. If in addition to $x(0)$ one would also fix the controls $\mathbf{u}$, the whole state trajectory $\mathbf{x}$ would be uniquely determined by these constraints. Constraints (8.1d) shall ensure that the state control pairs $(x(k), u(k))$ are contained in the set $\mathbb{Z}$ at each time step k. Finally, the terminal state constraint (8.1e) requires the final state to be in a given terminal set $\mathbb{X}_f$. The set of all variables $(\mathbf{x}, \mathbf{u})$ that satisfy all constraints (8.1b)-(8.1e) is called the *feasible set*. Note that the feasible set is the intersection of all constraint sets defined by the individual constraints.

8.1.2 Convex Versus Nonconvex Optimization

The most important dividing line in the field of optimization is between convex and nonconvex optimization problems. If an optimization problem is convex, every local minimum is also a global one. One can reliably solve most convex optimization problems of interest, finding the globally optimal solution in polynomial time. On the other hand, if a problem is not convex, one can usually not find the global minimum. Even if one has accidentally found the global minimum, one usually cannot certify that it is the global minimum. Thus, in nonconvex optimization, one has usually to accept that one is only able to find feasible or locally optimal points. Fortunately, if one has found such a point, one usually is also able to certify that it is a feasible or locally optimal point. But in the worst case, one might not be able to find even a feasible point, without knowing if this is due to the problem being infeasible, or the optimization algorithm being just unable to find points in the feasible set. Thus, the difference between convex and nonconvex has significant implications in practice. To say it in the words of the famous mathematical optimizer R. Tyrrell Rockafellar, "The great watershed in optimization is not between linearity and nonlinearity, but

convexity and nonconvexity."

When is a given optimization problem a *convex optimization problem*? By definition, an optimization problem is convex if its feasible set is a convex set and if its objective function is a convex function. In MPC, we usually have freedom in choosing the objective function, and in most cases one chooses a convex objective function. For example, the sum of quadratic functions of the form $\ell(x,u) = x'Qx + u'Ru$ with positive semidefinite matrices Q and R is a convex function. Usually, one also chooses the terminal cost V_f to be a convex function, so that the objective function is a convex function.

Likewise, one usually chooses the terminal set $\mathbb{X}_f$ to be a convex set. For example, one might choose an ellipsoid $\mathbb{X}_f = \{x \mid x'Px \leq 1\}$ with a positive definite matrix P, which is a convex set. Very often, one is lucky and also has convex constraint sets $\mathbb{Z}$, for example box constraints on $x(k)$ and $u(k)$. The initial-value constraint (8.1b) restricts the variable $x(0)$ to be in the point set $\{x_0\}$, which is convex. Thus, most of the constraints in the MPC optimization problem usually can be chosen to be convex. On the other hand, the constraints (8.1c) reflect the system dynamics $x(k+1) = f(x(k),u(k))$ for all k, and these might or might not describe a convex set. Interestingly, it turns out that they describe a convex set if the system model is linear or affine, i.e., if $f(x(k),u(k)) = Ax(k) + Bu(k) + c$ with matrices A, B and vector c of appropriate dimensions. This follows because the solution set of linear equalities is an affine set, which is convex. Conversely, if the system model is nonlinear, the solution set of the dynamic constraints (8.1c) is most likely not a convex set. Thus, we can formulate a modification of Rockafellar's statement above: in MPC practice, the great watershed between convex and nonconvex optimization problems usually coincides with the division line between linear and nonlinear system models.

One speaks of *linear MPC* if a linear or affine simulation model is used, and of *nonlinear MPC* otherwise. When speaking of linear MPC, one implicitly assumes that all other constraints and the objective function are chosen to be convex, but not necessarily linear. In particular, in linear MPC, the objective function usually is chosen to be convex quadratic. Thus, in the MPC literature, the term *linear MPC* is used as if it coincides with "convex linear MPC." Theoretically possible "nonconvex linear MPC" methods, where the system model is linear but where the cost or constraint sets are not convex, are not of great practical interest. On the other hand, for nonlinear MPC, i.e., when a nonlinear model is used, convexity usually is lost anyway, and there are no

implicit convexity assumptions on the objective and constraints, such that the term *nonlinear MPC* nearly always coincides with "nonconvex nonlinear MPC."

Example 8.1: Nonlinear MPC

We regard a simple MPC optimization problem of the form (8.1) with one dimensional state x and control u, system dynamics $f(x,u) = x + u - 2u^2$, initial value $x_0 = 1$, and horizon length $N = 1$, as follows

$$\begin{aligned} \underset{x(0),x(1),u(0)}{\text{minimize}} \quad & x(0)^2 + u(0)^2 + 10x(1)^2 && \text{(8.2a)} \\ \text{subject to} \quad & x(0) = x_0 && \text{(8.2b)} \\ & x(1) = x(0) + u(0) - 2u(0)^2 && \text{(8.2c)} \\ & -1 \le u(0) \le 1 && \text{(8.2d)} \end{aligned}$$

First, we observe that the optimization problem has a three-dimensional space of optimization variables. To check convexity of the problem, we first regard the objective, which is a sum of positive quadratic functions, thus a convex function. On the other hand, we need to check convexity of the feasible set. The initial-value constraint (8.2b) fixes one of the three variables, thus selects a two-dimensional affine subset in the three-dimensional space. This subset is described by $x(0) = 1$ while $u(0)$ and $x(1)$ remain free. Likewise, the control bounds in (8.2d) cut away all values for $u(0)$ that are less than -1 or more than $+1$, thus, there remains only a straight stripe of width 2 in the affine subset, still extending to infinity in the $x(1)$ direction. This straight two-dimensional stripe still is a convex set. The system equation (8.2c) is a nonlinear constraint that selects a curve out of the stripe, which is visualized on the left of Figure 8.1. This curve is not a convex set, because the connecting lines between two points on the curve are not always contained in the curve. In a formula, the feasible set is given by $\{(x(0), x(1), u(0)) \mid x(0) = 1, u(0) \in [-1,1], x(1) = 1 + u(0) - 2u(0)^2\}$.

Even though the objective function is convex, the fact that the optimization problem has a nonconvex feasible set can lead to different local minima. This is indeed the case in our example. To see this, let us evaluate the objective function on all feasible points and plot it as a function of $u(0)$. This *reduced objective function* $\psi(u)$ can be obtained by inserting $x(0) = 1$ and $x(1) = x(0)+u(0)-2u(0)^2$ into the objective $x(0)^2+u(0)^2+10x(1)^2$, which yields $\psi(u) = 1+u^2+10(1+u-2u^2)^2 = 11 + 20u - 29u^2 - 40u^3 + 40u^4$. This reduced objective is visualized

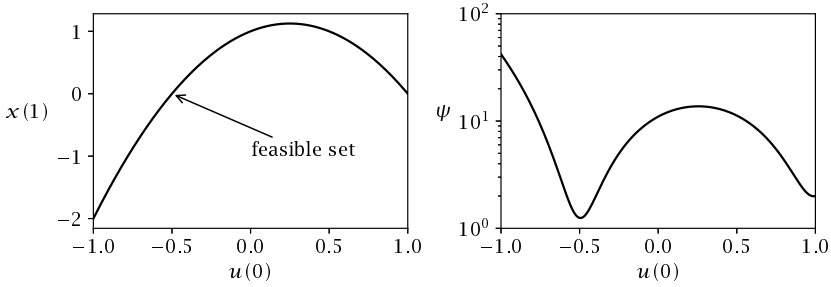


Figure 8.1: Feasible set and reduced objective $\psi(u(0))$ of the nonlinear MPC Example 8.1.

on the right of Figure 8.1 and it can clearly be seen that two different locally optimal solutions exist, only one of which is the globally optimal choice. □

8.1.3 Simultaneous Versus Sequential Optimal Control

The optimal control problem (OCP) (8.1) can be passed to an appropriate optimization routine without any modification. In this case, the optimization variables are given by both, the state trajectory $\mathbf{x}$ as well as the control trajectory $\mathbf{u}$. The pair $(\mathbf{x}, \mathbf{u})$ is consistent with the initial value x_0 and the simulation model if and only if the constraints (8.1b) and (8.1c) are satisfied, which is the case for any feasible solution of the problem. During the optimization calculations, however, these constraints might be violated, and the state trajectory $\mathbf{x}$ might not be a valid simulation corresponding to the controls $\mathbf{u}$. Since the optimization routine has to simultaneously solve the simulation and the optimization problem, one calls this approach the *simultaneous approach to optimal control*.

On the other hand, one could use the constraints (8.1b)-(8.1c) to find the unique feasible state trajectory $\mathbf{x}$ for any given control trajectory $\mathbf{u}$. We denote, as before in Chapter 2, the state $x(k)$ that results from a given initial condition x_0 and a given control trajectory $\mathbf{u} = (u(0), u(1), \ldots, u(N-1))$ by $\phi(k; x_0, \mathbf{u})$. Using this expression, that can be computed by a simple forward simulation routine, we can replace the equalities (8.1b)-(8.1c) by the trivial equalities $x(k) = \phi(k; x_0, \mathbf{u})$ for

$k = 0, 1, \ldots, N$. And these constraints can be used to eliminate the complete state trajectory $\mathbf{x} = (x(0), x(1), \ldots, x(N))$ from the optimization problem. The optimization problem in this reduced variable space is given by

$$\begin{aligned}
\underset{\mathbf{u}}{\text{minimize}} \quad & \sum_{k=0}^{N-1} \ell(\phi(k; x_0, \mathbf{u}), u(k)) + V_f(\phi(N; x_0, \mathbf{u})) && \text{(8.3a)}\\
\text{subject to} \quad & (\phi(k; x_0, \mathbf{u}), u(k)) \in \mathbb{Z}, \quad k = 0, 1, \ldots, N-1 && \text{(8.3b)}\\
& \phi(N; x_0, \mathbf{u}) \in \mathbb{X}_f && \text{(8.3c)}
\end{aligned}$$

If this reduced optimization problem is solved by an iterative optimization routine, in each iteration, one performs a sequence of two steps. First, for given $\mathbf{u}$, the simulation routine computes the state trajectory $\mathbf{x}$, and second, the optimization routine updates the control variables $\mathbf{u}$ to iterate toward an optimal solution. Due to this sequential evaluation of simulation and optimization routines, one calls this approach the *sequential approach to optimal control*. Though the simultaneous and the sequential approach solve equivalent optimization problems, their approach toward finding the solutions is different.

For linear MPC problems, where the system model is linear, the difference between the two approaches regards mostly the sparsity structure of the optimization problem, as discussed in Chapter 6 and in Section 8.8.4. In this case, one usually calls the reduced optimization problem (8.3) the *condensed problem*, and the computational process to generate the data for the condensed problem (8.3) from the data of the original problem (8.1) is called *condensing*. Though the condensed problem has fewer variables, the matrices defining it may have more nonzero entries than the original problem. Which of the two formulations leads to shorter computation times for a given problem depends on the number of states, controls and constraints, the specific sparsity structures, and on the horizon length N. For small N, condensing is typically preferable, while for large N, it is advisable to apply a sparse convex solver to the original problem in the full variable space. Despite the different sparsity structure, and different cost per iteration, many widely used convex optimization algorithms perform identical iterates on both problems, because the eliminated constraints are linear and are exactly respected in each iteration in both the condensed as well as the original problem formulation.

For nonlinear MPC problems, the sequential and simultaneous approach can lead to significantly different optimization iterations. Even

if both problems are addressed with the same optimization algorithm and are initialized with the same initial guess, i.e., the same $\mathbf{u}$ for both, together with the corresponding simulation result $\mathbf{x}$, the optimization iterations typically differ after the first iteration, such that the two formulations can need a significantly different number of iterations to converge; they might even converge to different local solutions or one formulation might converge while the other does not. As a rule of thumb, the sequential approach is preferable if the optimization solver cannot exploit sparsity and the system is stable, while the simultaneous approach is preferable for unstable nonlinear systems, for problems with state constraints, and for systems which need implicit simulation routines.

Example 8.2: Sequential approach

We regard again the simple MPC optimization problem (8.2a), but eliminate the states as a function of $\mathbf{u} = (u(0))$ by $x(0) = \phi(0; x_0, \mathbf{u}) = x_0$ and $x(1) = \phi(1; x_0, \mathbf{u}) = x_0 + u(0) - 2u(0)^2$. The reduced optimization problem in the sequential approach is then given by

$$\underset{u(0)}{\text{minimize}} \quad x_0^2 + u(0)^2 + 10\left(x_0 + u(0) - 2u(0)^2\right)^2 \tag{8.4a}$$

$$\text{subject to} \quad -1 \le u(0) \le 1 \tag{8.4b}$$

□

8.1.4 Continuous Time Optimal Control Problem

In most nonlinear MPC applications and many linear MPC applications, the system dynamics are not given in discrete time but in continuous time, in form of differential equations

$$\frac{dx}{dt} = f_c(x, u)$$

For notational convenience, we usually denote differentiation with respect to time by a dot above the quantity, i.e., we can abbreviate the above equations by $\dot{x} = f_c(x, u)$. Both the state and control trajectories are functions of continuous time, and we denote them by $x(t)$ and $u(t)$. The trajectories need only to be defined on the time horizon of interest, i.e., for all $t \in [0, T]$, where T is the horizon length. If we do not assume any discretization, and if we use the shorthand symbols

$x(\cdot)$ and $u(\cdot)$ to denote the state and control trajectories, the continuous time optimal control problem (OCP) can be formulated as follows

$$\begin{aligned}
\underset{x(\cdot),\,u(\cdot)}{\text{minimize}} \quad & \int_0^T \ell_c(x(t), u(t))\, dt + V_f(x(T)) && \text{(8.5a)}\\
\text{subject to} \quad & x(0) = x_0 && \text{(8.5b)}\\
& \dot{x}(t) = f_c(x(t), u(t)), \quad t \in [0, T] && \text{(8.5c)}\\
& (x(t), u(t)) \in \mathbb{Z}, \quad t \in [0, T] && \text{(8.5d)}\\
& x(T) \in \mathbb{X}_f && \text{(8.5e)}
\end{aligned}$$

It is important to note that the continuous time optimal control problem is an infinite-dimensional optimization problem with infinite-dimensional decision variables and an infinite number of constraints, because the time index t runs through infinitely many values $t \in [0, T]$. This is in contrast to discrete time, where the finite number of time indices $k \in \mathbb{I}_{0:N}$ leads to finitely many decision variables and constraints.

There exists a variety of methods to numerically solve continuous time OCPs. What all approaches have in common is that at one point, the infinite-dimensional problem needs to be discretized. One family of methods first formulates what is known as the Hamilton-Jacobi-Bellman (HJB) equation, a partial differential equation for the value function, which depends on both state space and time, and then discretizes and solves it. Unfortunately, due to the "curse of dimensionality," this approach is only practically applicable to systems with small state dimensions, say less than five, or to the special case of unconstrained linear systems with quadratic costs.

A second family of methods, the *indirect methods*, first derive optimality conditions in continuous time by algebraic manipulations that use similar expressions as the HJB equation; they typically result in the formulation of a boundary-value problem (BVP), and only discretize the resulting continuous time BVP at the very end of the procedure. One characterizes the indirect methods often as "first optimize, then discretize." A third class of methods, the *direct methods*, first discretizes the continuous time OCP, to convert it into a finite-dimensional optimization problem. The finite-dimensional optimization problem can then be solved by tailored algorithms from the field of numerical optimization. The direct methods are often characterized as "first discretize, then optimize." These methods are most widely used in MPC applications and are therefore the focus of this chapter.

To sketch the discretization methods, we look at the continuous time optimal control problem (8.5). In a direct method, we replace the continuous index $t \in [0, T]$ by a discrete integer index. For this aim, we can divide the time horizon T into N intervals, each of length $h = \frac{T}{N}$, and evaluate the quantities of interest only for the discrete time points $t = hk$ with $k \in \mathbb{I}_{0:N}$. We use the notation $h\mathbb{I}_{0:N} = \{0, h, 2h, \ldots, Nh\}$, such that we can use the expression "$t \in h\mathbb{I}_{0:N}$" to indicate that t is only considered at these discrete time points. To discretize the OCP, the objective integral is replaced by a Riemann sum, and the time derivative by a finite difference approximation: $\dot{x}(t) \approx \frac{x(t+h)-x(t)}{h}$. As before in discrete time, we denote the sequence of discrete states by $\mathbf{x} = (x(0), x(h), x(2h), \ldots, x(Nh))$ and the sequence of controls by $\mathbf{u} = (u(0), u(h), \ldots, u(Nh-h))$.

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{u}}{\text{minimize}} \quad & \sum_{t \in h\mathbb{I}_{0:N-1}} h\ell_{\mathrm{c}}(x(t), u(t)) + V_f(x(Nh)) && \text{(8.6a)}\\
\text{subject to} \quad & x(0) = x_0 && \text{(8.6b)}\\
& \frac{x(t+h)-x(t)}{h} = f_{\mathrm{c}}(x(t), u(t)), \quad t \in h\mathbb{I}_{0:N-1} && \text{(8.6c)}\\
& (x(t), u(t)) \in \mathbb{Z}, \qquad\qquad t \in h\mathbb{I}_{0:N-1} && \text{(8.6d)}\\
& x(Nh) \in \mathbb{X}_f && \text{(8.6e)}
\end{aligned}$$

It is easily checked that the constraints (8.6b)-(8.6c) uniquely determine all states $\mathbf{x}$ if the control sequence $\mathbf{u}$ is given. The above problem is exactly in the form of the discrete time optimization problem (8.1), if one uses the definitions $\ell(x, u) := h\ell_{\mathrm{c}}(x, u)$ and $f(x, u) := x + hf_{\mathrm{c}}(x, u)$. This simple way to go from continuous to discrete time, in particular the idea to solve a differential equation $\dot{x} = f_{\mathrm{c}}(x, u)$ by the simple difference method $x^+ = x + hf_{\mathrm{c}}(x, u)$, is originally due to Leonhard Euler (1707–1783), and is therefore called the *Euler integration method*. The Euler method is not the only possible integration method, and in fact, not the most efficient one. Numerical analysts have investigated the simulation of differential equations for more than two centuries, and discovered powerful discretization methods that have much lower computational cost and higher accuracy than the Euler method and are therefore more widely used in practice. These are the topic of the next section.

8.2 Numerical Simulation

The classical task of numerical simulation is the solution of *initial-value problems*. An initial-value problem is characterized by an initial state value x_0 at time 0, and a differential equation $\dot{x} = f(t, x)$ that the solution $x(t)$ should satisfy on the time interval of interest, i.e., for all $t \in [0, T]$ with $T > 0$. In particular, we are interested in computing an approximation of the final state $x(T)$. In this section, we allow an explicit dependence of the *right-hand-side* function $f(t, x)$ on time. To be consistent with the literature in the field of numerical simulation—and deviating from the notation in other chapters of this book—we use t here as the first input argument of $f(t, x)$. The time dependence might in particular be due to a fixed control trajectory $u(t)$, and if a given system is described by the continuous time ODE $\dot{x} = f_c(x, u)$, the time dependent right-hand-side function is defined by $f(t, x) := f_c(x, u(t))$. The choice of the control trajectory $u(t)$ is not the focus in this section, but becomes important later when we treat the solution of optimal control problems. Instead, in this section, we just review results from the field of numerical simulation of ordinary differential equations—which is sometimes also called *numerical integration*—that are most relevant to continuous time optimal control computations.

Throughout this section we consider the following initial-value problem

$$x(0) = x_0, \quad \dot{x}(t) = f(t, x(t)) \quad \text{for} \quad t \in [0, T] \tag{8.7}$$

with a given right-hand-side function $f : [0, T] \times \mathbb{R}^n \to \mathbb{R}^n$. We denote the exact solution, if it exists, by $x(t)$. Existence of a unique solution of the initial-value problem is guaranteed by a classical theorem by Émile Picard (1856–1941) and Ernst Lindelöf (1870–1946), which requires the function f to be continuous with respect to time t and Lipschitz continuous with respect to the state x. Lipschitz continuity is stronger than continuity and requires the existence of a constant $L > 0$ such that the following inequality

$$|f(t, x) - f(t, y)| \leq L\,|x - y| \tag{8.8}$$

holds for all $t \in [0, T]$ and all $x, y \in \mathbb{R}^n$. In many cases of interest, the function f is not defined on the whole state space, or there might exist no global Lipschitz constant L for all states x and y. Fortunately, a local version of the Picard-Lindelöf Theorem exists that only needs Lipschitz continuity in a neighborhood of the point $(0, x_0)$ and still

ensures the existence of a unique solution $x(t)$ for sufficiently small T. Local Lipschitz continuity is implied by continuous differentiability, which is easy to verify and holds for most functions f arising in practice. In fact, the function f usually is many times differentiable in both its arguments, and often even infinitely many times—for example, in the case of polynomials or other analytic functions. The higher differentiability of f also leads to higher differentiability of the solution trajectory $x(t)$ with respect to t, and is at the basis of the higher-order integration methods that are widely used in practice.

Because all numerical integration methods produce only approximations to the true solution $x(t)$, we use a different symbol for these approximations, namely $\tilde{x}(t)$. The numerical approximation is usually only exact for the initial value, where we simply set $\tilde{x}(0) := x_0$. For the final state at time T, we aim to have a small error $E(T) := |\tilde{x}(T) - x(T)|$, at low computational cost. All integration methods divide the time horizon of interest into smaller intervals, and proceed by making a sequence of integration steps, one per interval. For simplicity, assume that the steps are equidistant, and that in total N steps of size $h = T/N$ are taken. In each step, the integration method makes a *local error*, and the combined effect of the accumulated local errors at time t, i.e., the distance $E(t) = |\tilde{x}(t) - x(t)|$, is called the *global error*. After the first integrator step, local and global error coincide because the integration starts on the exact trajectory, but in subsequent steps, the global error typically grows while the local errors remain of similar size.

8.2.1 Explicit Runge-Kutta Methods

Let us first investigate the Euler integrator, that iterates according to the update rule

$$\tilde{x}(t+h) = \tilde{x}(t) + hf(t, \tilde{x}(t))$$

starting with $\tilde{x}(0) = x_0$. Which local error do we make in each step? For local error analysis, we assume that the starting point $\tilde{x}(t)$ was on an exact trajectory, i.e., equal to $x(t)$, while the result of the integrator step $\tilde{x}(t+h)$ is different from $x(t+h)$. For the analysis, we assume that the true trajectory $x(t)$ is twice continuously differentiable with bounded second derivatives, which implies that its first-order Taylor series satisfies $x(t+h) = x(t) + h\dot{x}(t) + O(h^2)$, where $O(h^2)$ denotes an arbitrary function whose size shrinks faster than h^2 for $h \to 0$. Since the first derivative is known exactly, $\dot{x}(t) = f(t, x(t))$, and was used in the Euler

integrator, we immediately obtain that $|\tilde{x}(t+h) - x(t+h)| = O(h^2)$. Because the global error is the accumulated and propagated effect of the local errors, and because the total number of integrator steps grows linearly with $1/h$, one can show that the global error at the end of the interval of interest is of size $1/h\, O(h^2) = O(h)$, i.e., of first order. For this reason one says that the Euler method is a first-order integration method. The Euler integrator is easy to remember and easy to implement, but the number of time steps that are needed to obtain even a moderate accuracy can be reduced significantly if higher-order methods are used.

Like the Euler integrator, all *one-step integration methods* create a discrete time system of the form

$$\tilde{x}(t+h) = \tilde{x}(t) + \Phi(t, \tilde{x}(t), h)$$

Here, the map Φ approximates the integral $\int_t^{t+h} f(\tau, x(\tau))\ d\tau$. If Φ would be equal to this integral, the integration method would be exact, due to the identity

$$x(t+h) - x(t) = \int_t^{t+h} \dot{x}(\tau)\ d\tau = \int_t^{t+h} f(\tau, x(\tau))\, d\tau$$

While the Euler integrator approximates the integral by the expression $\Phi(t, x, h) = hf(t, x(t))$ that has an error of $O(h^2)$ and needs only one evaluation of the function f per step, one can find more accurate approximations by allowing more than one function evaluation per integration step. This idea leads directly to the Runge-Kutta (RK) integration methods, that are named after Carl Runge (1856–1927) and Martin Wilhelm Kutta (1867–1944).

The classical Runge-Kutta method (RK4). One of the most widely used methods invented by Runge and Kutta performs four function evaluations, as follows.

$$\begin{aligned} k_1 &= f(t, x) \\ k_2 &= f(t + h/2, x + (h/2)k_1) \\ k_3 &= f(t + h/2, x + (h/2)k_2) \\ k_4 &= f(t + h, x + hk_3) \\ \Phi &= (h/6)k_1 + (h/3)k_2 + (h/3)k_3 + (h/6)k_4 \end{aligned}$$

It is a fourth-order method, and therefore often abbreviated RK4. Since it is one of the most competitive methods for the accuracies that are

typically needed in applications, the RK4 integrator is one of the most widely used integration methods for simulation of ordinary differential equations. A comparison of the RK4 method with Euler's first-order method and a second-order method named after Karl Heun (1859–1929) is shown in Figure 8.2.

Example 8.3: Integration methods of different order

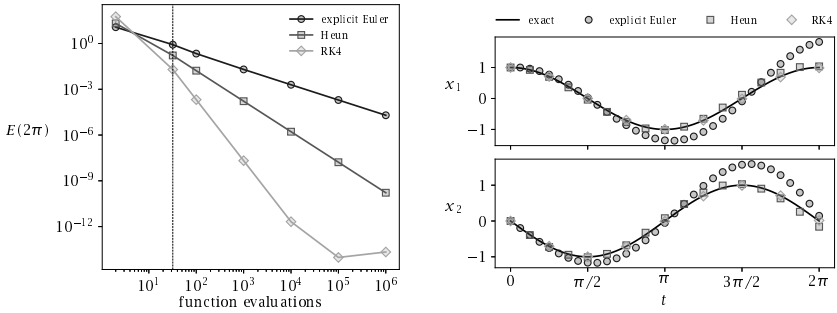
We regard the simulation of the linear ordinary differential equation (ODE)

$$\dot{x} = Ax \qquad \text{with} \qquad A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

over the interval $T = 2\pi$, starting at $x_0 = [1, 0]'$. The analytic solution of this system is known to be $x(t) = \exp(At)x_0 = [\cos(t), -\sin(t)]'$, such that the final state is given by $x(2\pi) = [1, 0]'$. To investigate the performance of different methods, we divide the time horizon into N equal integration intervals of length $h = 2\pi/N$. Note that a Runge-Kutta method with s stages needs in total $M := Ns$ function evaluations. We compare the Euler ($s = 1$), Heun ($s = 2$), and RK4 method ($s = 4$). For each integration method we evaluate the global error at the end of the integration interval, $E(2\pi) = |\tilde{x}(2\pi) - x(2\pi)|$, and plot it as a function of the number of function evaluations, M, in Figure 8.2. We use a doubly logarithmic scale, i.e., plot $\log(\epsilon)$ versus $\log(M)$, to show the effect of the order. Note that the slope of the higher-order methods is an integer multiple of the slope of the Euler method. Also note that the accuracy for each investigated method cannot exceed a certain base value due to the finite precision arithmetic, and that this limit is reached for the RK4 integrator at approximately $M = 10^5$. After this point, increasing the number of integration steps does not further improve the accuracy. □

The Butcher tableau. A general explicit Runge-Kutta method with s stages performs the following computations in each integration step

$$\begin{aligned}
k_1 &= f(t + c_1 h,\ x \qquad\qquad\qquad\qquad\qquad\qquad\quad) \\
k_2 &= f(t + c_2 h,\ x + h\,(a_{21} k_1) \qquad\qquad\qquad\quad) \\
k_3 &= f(t + c_3 h,\ x + h\,(a_{31} k_1 + a_{32} k_2) \qquad\quad) \\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \ddots \\
k_s &= f(t + c_s h,\ x + h\,(a_{s1} k_1 + \ldots + a_{s,s-1} k_{s-1}) \quad) \\
\\
\Phi &= \qquad\qquad\quad h\,(b_1 k_1 + \ldots + b_{s-1} k_{s-1} + b_s k_s)
\end{aligned}$$



(a) Accuracy vs. function evaluations.

(b) Simulation results for $M = 32$.

Figure 8.2: Performance of different integration methods.

It is important to note that on the right-hand side of each row, only those k_i values are used that are already computed. This property holds for every *explicit* integration method, and makes it possible to explicitly evaluate the first s equations one after the other to obtain all values $k_1, \ldots, k_s$ for the summation in the last line. One usually summarizes the coefficients of a Runge-Kutta method in what is known as a Butcher tableau (after John C. Butcher, born 1933) given by

$$\begin{array}{c|cccc} c_1 & & & & \\ c_2 & a_{21} & & & \\ c_3 & a_{31} & a_{32} & & \\ \vdots & \ddots & \ddots & & \\ c_s & a_{s1} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

The Butcher tableau of three popular RK methods is stated below

Euler

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}$$

Heun

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array}$$

RK4

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 2/6 & 2/6 & 1/6 \end{array}$$

Note that the b_i coefficients on the bottom always add to one. An interesting fact is that an s-stage explicit Runge-Kutta method can never

have a higher order than s. And only for orders equal or less than four exist explicit Runge-Kutta methods for which the order and the number of stages coincide.

8.2.2 Stiff Equations and Implicit Integrators

Unfortunately, some differential equations cannot reliably be solved by explicit integration methods; it can occur that even if the underlying ODE is stable, the integration method is not. Let us regard the scalar linear ODE

$$\dot{x} = \lambda x$$

with initial condition x_0 as a test case. The exact solution is known to be $x(t) = e^{\lambda t} x_0$. When this ODE is solved by an explicit Euler method, it iterates like $x^+ = x + h\lambda x$ and it is easy to see that the explicit solution is given by $\widetilde{x}(kh) = (1 + h\lambda)^k x_0$. For positive λ, this leads to exponential growth, which is not surprising given that the exact ODE solution grows exponentially. If λ is a large negative number, however, the exact solution $x(t)$ would decay very fast to zero, while the Euler integrator is unstable and oscillates with exponentially growing amplitude if h is larger than $2/(-\lambda)$. A similar observation holds for all explicit integration methods.

The most perturbing fact is that the explicit integration methods are extremely unstable exactly because of the fact that the system is extremely stable. Extremely stable ODEs are called *stiff* equations. For stiff ODE $\dot{x} = f(t, x)$, some of the eigenvalues of the Jacobian f_x have extremely large negative real parts, which lead to extremely stable subdynamics. Exactly these extremely stable subdynamics let the explicit integrators fail; even for relatively short stepsizes h, they overshoot the true solution and exhibit unstable oscillations. These oscillations do not just lead to inaccurate solutions, but in fact they quickly exceed the range of computer representable numbers (10^{308} for double precision), such that the explicit integrator just outputs "NaN" ("not a number") most of the time.

Fortunately, there exist integration methods that remain stable even for stiff ODE. Their only drawback is that they are implicit, i.e., they require the solution of an equation system to compute the next step. The simplest of these implicit methods is called the implicit Euler method and it iterates according to

$$x^+ = x + hf(t + h, x^+)$$

Note that the desired output value x^+ appears also on the right side of the equation. For the scalar linear ODE $\dot{x} = \lambda x$, the implicit Euler step is determined by $x^+ = x + h\lambda x^+$, which can explicitly be solved to give $x^+ = x/(1 - h\lambda)$. For any negative λ, the denominator is larger than one, and the numerical approximation $\tilde{x}(kh) = x_0/(1 - h\lambda)^k$ therefore decays exponentially, similar to the exact solution. An integration method which has the desirable property that it remains stable for the test ODE $\dot{x} = \lambda x$ whenever $\mathrm{Re}(\lambda) < 0$ is called *A-stable*. While none of the explicit Runge-Kutta methods is A-stable, the implicit Euler method is A-stable. But it has a low order. Can we devise A-stable methods that have a higher order?

8.2.3 Implicit Runge-Kutta and Collocation Methods

Once we accept that we need to solve a nonlinear equation system in order to compute an integration step, we can extend the family of Runge-Kutta methods by allowing diagonal and upper-triangular entries in the Butcher tableau. Our hope is to find integration methods that are both A-stable and have a high order. A general *implicit Runge-Kutta method* with s stages solves the following nonlinear system in each integration step

$$
\begin{array}{lllllll}
k_1 & = f(t + c_1 h\,, & x + & h\,(a_{11}k_1 & + a_{12}k_2 & + \ldots & + a_{1,s}k_s)\,) \\
k_2 & = f(t + c_2 h\,, & x + & h\,(a_{21}k_1 & + a_{22}k_2 & + \ldots & + a_{2,s}k_s)\,) \\
 & \vdots & & & \vdots & & \vdots \\
k_s & = f(t + c_s h\,, & x + & h\,(a_{s1}k_1 & + a_{s,2}k_2 & + \ldots & + a_{s,s}k_s)\,) \\
 & & & & & & \\
\Phi & = & & h\,(b_1\,k_1 & + b_2\,k_2 & + \ldots & + b_s\,k_s\,)
\end{array}
$$

Note that the upper s equations are implicit and form a root-finding problem with sn nonlinear equations in sn unknowns, where s is the number of RK stages and n is the state dimension of the differential equation $\dot{x} = f(t, x)$. Nonlinear root-finding problems are usually solved by Newton's method, which is treated in the next section. For Newton's method to work, one has to assume that the Jacobian of the residual function is invertible. For the RK equations above, this can be shown to always hold if the time step h is sufficiently small, depending on the right-hand-side function f. After the values $k_1, \ldots, k_s$ have been computed, the last line can be executed and yields the resulting map $\Phi(t, x, h)$. The integrator then uses the map Φ to proceed to the next integration step exactly as the other one-step methods, according to

the update equation

$$\widetilde{x}(t+h) = \widetilde{x}(t) + \Phi(t, \widetilde{x}(t), h)$$

For implicit integrators, contrary to the explicit ones, the map Φ cannot easily be written down as a series of function evaluations. Evaluation of $\Phi(t, x, h)$ includes the root-finding procedure and typically needs several evaluations of the root-finding equations and of their derivatives. Thus, an s-stage implicit Runge-Kutta method is significantly more expensive per step compared to an s-stage explicit Runge-Kutta method. Implicit integrators are usually preferable for stiff ordinary differential equations, however, due to their better stability properties.

Many different implicit Runge-Kutta methods exist, and each of them can be defined by its Butcher tableau. For an implicit RK method, at least one of the diagonal and upper-triangular entries (a_{ij} with $j \geq i$) is nonzero. Some methods try to limit the implicit part for easier computations. For example, the diagonally implicit Runge-Kutta methods have only the diagonal entries nonzero while the upper-triangular part remains zero.

Collocation methods. One particularly popular subclass of implicit Runge-Kutta methods is formed by the *collocation methods*. An s-stage collocation method first fixes the values c_i of the Butcher tableau, and chooses them so that they are all different and in the unit interval, i.e., $0 \leq c_1 < c_2 < \ldots < c_s \leq 1$. The resulting time points $(t + hc_i)$ are called the *collocation points*, and their choice uniquely determines all other entries in the Butcher tableau. The idea of collocation is to approximate the trajectory on the collocation interval by a polynomial $\widetilde{x}(\tau)$ for $\tau \in [t, t+h]$, and to require satisfaction of the ODE $\dot{x} = f(t, x)$ only on the collocation points, i.e., impose the conditions $\dot{\widetilde{x}}(t + hc_i) = f(t+hc_i, \widetilde{x}(t+hc_i))$ for $i = 1, \ldots, s$. Together with the requirement that the approximating polynomial $\widetilde{x}(\tau)$ should start at the initial value, i.e., $\widetilde{x}(t) = x$, we have $(s+1)$ conditions such that the polynomial needs to have $(s+1)$ coefficients, i.e., have the degree s, to yield a well-posed root-finding problem.

The polynomial $\widetilde{x}(\tau)$ can be represented in different ways, which are related via linear basis changes and therefore lead to numerically equivalent root-finding problems. One popular way is to parameterize $\widetilde{x}(\tau)$ as the interpolating polynomial through the initial value x and the state values at the collocation points. This only gives a unique parameterization if $c_1 \neq 0$. To have a more generally applicable derivation

of collocation, we use instead the value x together with the s derivative values $k_1, \ldots, k_s$ at the collocation time points to parameterize $\tilde{x}(\tau)$. More precisely, we use the identity $\tilde{x}(\tau) = x + \int_t^\tau \dot{\tilde{x}}(\tau_1; k_1, k_2, \ldots, k_s)\, d\tau_1$, where $\dot{\tilde{x}}(\cdot)$ is the time derivative of $\tilde{x}(\tau)$, and therefore a polynomial of degree $(s-1)$ that can be represented by s coefficients. Fortunately, due to the fact that all collocation points are different, the interpolating polynomial through the s vectors $k_1, \ldots, k_s$ is well defined and can easily be represented in a Lagrange basis, with basis functions $L_i\left(\frac{\tau - t}{h}\right)$ that are one on the i-th collocation point and zero on all others.[1] Collocation thus approximates $\dot{x}(\tau)$ by the polynomial

$$\dot{\tilde{x}}(\tau; k_1, k_2, \ldots, k_s) := k_1 L_1\left(\frac{\tau - t}{h}\right) + k_2 L_2\left(\frac{\tau - t}{h}\right) + \ldots + k_s L_s\left(\frac{\tau - t}{h}\right)$$

and $x(\tau)$ by its integral

$$\tilde{x}(\tau; x, k_1, k_2, \ldots, k_s) := x + \int_t^\tau \dot{\tilde{x}}(\tau_1; k_1, k_2, \ldots, k_s)\, d\tau_1$$

To obtain the state at the collocation point $(t + c_i h)$, we just need to evaluate $\tilde{x}(t + c_i h; x, k_1, k_2, \ldots, k_s)$, which is given by the following integral

$$x + \int_t^{t+c_i h} \dot{\tilde{x}}(\tau_1; k_1, k_2, \ldots, k_s)\, d\tau_1 = x + \sum_{j=1}^{s} k_j h \underbrace{\int_0^{c_i} L_j(\sigma)\, d\sigma}_{=: a_{ij}}$$

Note that the integrals over the Lagrange basis polynomials depend only on the relative positions of the collocation time points, and directly yield the coefficients a_{ij}. Likewise, to obtain the coefficients b_i, we evaluate $\tilde{x}(t + h; x, k_1, k_2, \ldots, k_s)$, which is given by

$$x + \int_t^{t+h} \dot{\tilde{x}}(\tau; k_1, k_2, \ldots, k_s)\, d\tau = x + \sum_{i=1}^{s} k_i h \underbrace{\int_0^1 L_i(\sigma)\, d\sigma}_{=: b_i}$$

In Figure 8.3, the difference between the exact solution $x(\tau)$ and the collocation polynomial $\tilde{x}(\tau)$ as well as the difference between their

[1] The Lagrange basis polynomials are defined by

$$L_i(\sigma) := \prod_{1 \le j \le s,\ j \ne i} \frac{(\sigma - c_j)}{(c_i - c_j)}$$

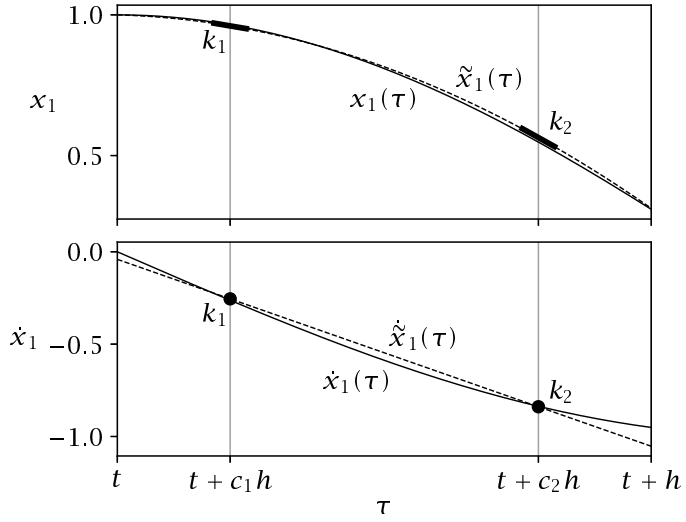


Figure 8.3: Polynomial approximation $\widetilde{x}_1(t)$ and true trajectory $x_1(t)$ of the first state and its derivative, computed at the first integration step of the GL4 collocation method applied to the stiff ODE from Example 8.4. Note that the accuracy of the polynomial at the end of the interval is significantly higher than in the interior. The result of this first GL4 step can also be seen on the right side of Figure 8.4.

time derivatives is visualized, for a collocation method with $s = 2$ collocation points (GL4) applied to the ODE from Example 8.4. Note that in this example, $\dot{\widetilde{x}}(\tau; k_1, k_2, \ldots, k_s)$ is a polynomial of order one, i.e., an affine function, and its integral, $\widetilde{x}(\tau; x, k_1, k_2, \ldots, k_s)$, is a polynomial of order two.

The Butcher tableau of three popular collocation methods is

Implicit Euler	Midpoint rule (GL2)	Gauss-Legendre of order 4 (GL4)
$\begin{array}{c\|c} 1 & 1 \\ \hline & 1 \end{array}$	$\begin{array}{c\|c} 1/2 & 1/2 \\ \hline & 1 \end{array}$	$\begin{array}{c\|cc} 1/2-\sqrt{3}/6 & 1/4 & 1/4-\sqrt{3}/6 \\ 1/2+\sqrt{3}/6 & 1/4+\sqrt{3}/6 & 1/4 \\ \hline & 1/2 & 1/2 \end{array}$

An interesting remark is that the highest order that an s-stage implicit Runge-Kutta method can achieve is given by $2s$, and that the Gauss-Legendre collocation methods achieve this order, due to a particularly smart choice of collocation points (namely as roots of the orthogonal Legendre polynomials, following the idea of Gaussian quadrature). The midpoint rule is a Gauss-Legendre method of second order (GL2). The Gauss-Legendre methods, like many other popular collocation methods, are A-stable. Some methods, such as the Radau IIA collocation methods, have even stronger stability properties (they are also L-stable), and are often preferable for stiff problems. All collocation methods need to solve a nonlinear system of equations in ns dimensions in each step, which can become costly for large state dimensions and many stages.

Example 8.4: Implicit integrators for a stiff ODE system

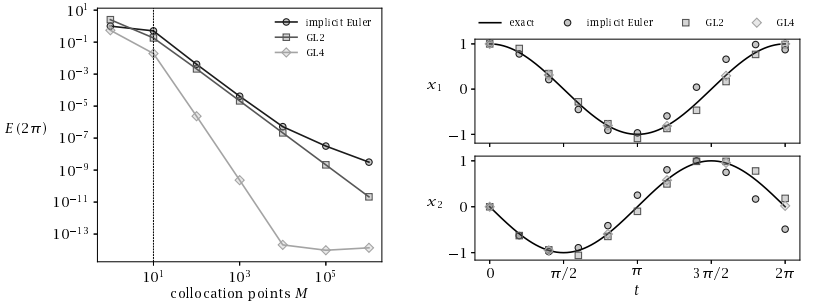
We consider the following ODE

$$\dot{x} = Ax - 500\,x\,(|x|^2 - 1)$$

with A and initial conditions as before in Example 8.3. In contrast to the previous example, this ODE is nonlinear and stiff, due to the additive nonlinear term $-500\,x\,(|x|^2 - 1)$. This term is zero only if the norm of x is one, i.e., if the state lies on the unit circle. If not, the state is strongly pushed toward the unit circle. This makes the system a stiff ODE. As we start at $[1, 0]'$, the exact solution lies again on the unit circle, and also ends at $[1, 0]'$. For comparison, we solve the initial value problem with three implicit integration methods, all of collocation type (implicit Euler, GL2, GL4). To have an approximate measure of the computational costs of the different methods, we denote by M the total number of collocation points on the time horizon. The results are shown in Figure 8.4. On the left-hand side, the different order behavior is observed. On the right-hand side, the trajectories resulting from a total of $M = 10$ collocation points are shown for the three different methods. In Figure 8.3, the first step of the GL4 method is visualized in detail, showing both the trajectory of the first state as well as its time derivative, together with their polynomial approximations. □

8.2.4 Differential Algebraic Equations

Some system models do not only contain differential, but also algebraic equations, and therefore belong to the class of *differential algebraic*



(a) Accuracy vs. collocation points. (b) Simulation result for $M = 10$ points.

Figure 8.4: Performance of implicit integration methods on a stiff ODE.

equations (DAEs). The algebraic equations might, for example, reflect conservation laws in chemical reaction models or kinematic constraints in robot models. DAE models come in many different forms, some of which are easier to treat numerically than others. One particularly favorable class of DAE are the *semiexplicit DAE of index one*, which can be written as

$$\dot{x} = f(t, x, z) \tag{8.9a}$$

$$0 = g(t, x, z) \tag{8.9b}$$

Here, the *differential states* $x \in \mathbb{R}^n$ are accompanied by *algebraic states* $z \in \mathbb{R}^{n_z}$, and the algebraic states are implicitly determined by the *algebraic equations* (8.9b). Here, the number of algebraic equations is equal to the number of algebraic states, i.e., $g : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^{n_z} \to \mathbb{R}^{n_z}$, such that for fixed t and x, the algebraic equation (8.9b) forms a nonlinear system of n_z equations for n_z unknowns.

The assumption of *index one* requires the Jacobian matrix of g with respect to z to be invertible at all points of interest. The fact that $\dot{x}$ appears alone on the left side of the differential equation (8.9a) makes the DAE *semiexplicit*. An interesting observation is that it is possible to reduce a semiexplicit DAE of index one to an ODE if one finds an explicit symbolic expression $z^*(t, x)$ for the implicit function defined by $g(t, x, z^*(t, x)) = 0$. The resulting ODE that is equivalent to the original DAE is given by $\dot{x} = f(t, x, z^*(t, x))$. Usually, this reduction from an index-one DAE to an ordinary differential equation is not possible analytically. A numerical computation of $z^*(t, x)$ is always possible in principle, but

requires the use of an underlying root-finding method. This way it is possible to solve a DAE with explicit integration methods. For implicit integration methods, however, one can simply augment the nonlinear equation system by the algebraic equations g at all evaluation points of the right-hand-side of the differential function f, and then rely on the root-finding method of the integrator. For this reason, and because they are often stiff, DAE are usually addressed with implicit integrators.

8.2.5 Integrator Adaptivity

Many practical integration methods use an *adaptive stepsize selection* to attain a good trade-off between numerical accuracy and computational effort. Instead of performing steps of equal length h, adaptive methods vary h in each step. Usually, they try to keep an estimate of the local error constant. The details are beyond our interest here, but we note that integrator adaptivity can be a crucial feature for the efficiency of nonlinear MPC implementations, in particular for the long simulation intervals which appear when one appends a prediction horizon at the end of the control horizon. On the other hand, integrator adaptivity needs to be treated with care when numerical derivatives of the simulation result are computed, as discussed in Section 8.4.6.

8.3 Solving Nonlinear Equation Systems

We have seen that an important subtask within numerical simulation—as well as in numerical optimization—is the solution of nonlinear equation systems. In this section, we therefore discuss the basic technologies that make it possible to solve implicit equation systems with thousands of unknowns within a few milliseconds. We start with linear equations, and then proceed to nonlinear equations and their solution with *Newton-type methods*.

8.3.1 Linear Systems

Solving a linear system of equations $Az = b$ with a square invertible matrix $A \in \mathbb{R}^{n_z \times n_z}$ is an easy task in the age of digital computers. The direct solution of the system requires only two computational steps: first, a factorization of the matrix A, for example, a lower-upper-factorization (LU-factorization) that yields a lower-triangular matrix L and an upper-triangular matrix U such that $LU = A$. Second, one

needs to perform a forward and a back substitution, yielding the solution as $z = U^{-1}(L^{-1}b)$. The computation of the LU-factorization, or LU-decomposition, requires $(2/3)n_z^3$ floating-point operations (FLOPs), while the forward and back substitution require together n_z^2 operations. Additional row or column permutations—in a process called *pivoting*—usually need to be employed and improve numerical stability, but only add little extra computational cost. The LU-decomposition algorithm was introduced by Alan Turing (1912–1954), and can be traced back to *Gaussian elimination*, after Carl Friedrich Gauss (1777–1855). Solving a dense linear system with $n_z = 3000$ variables needs about $18 \cdot 10^9$ FLOPs, which on a current quadcore processor (2.9 GHz Intel Core i5) need only 600 ms.

The runtime of the LU-decomposition and the substitutions can significantly be reduced if the matrix A is sparse, i.e., if it has many more zero than nonzero entries. Sparsity is particularly simple to exploit in case of banded matrices, which have their nonzero entries only in a band around the diagonal. Tailored direct methods also can exploit other structures, like block sparsity, or symmetry of the matrix A. For symmetric A, one usually performs a lower-diagonal-lower-transpose-factorization (LDLT-factorization) of the form $LDL' = A$ (with lower-triangular L and diagonal D), which reduces the computational cost by a factor of two compared to an LU-factorization. For symmetric and positive definite matrices A, one can even apply a *Cholesky decomposition* of the form $LL' = A$, with similar costs as the LDLT-factorization.

For huge linear systems that cannot be addressed by direct factorization approaches, there exist a variety of *indirect* or *iterative* solvers. Linear system solving is one of the most widely used numerical techniques in science and engineering, and the field of computational linear algebra is investigated by a vibrant and active research community. Contrary to only a century ago, when linear system solving was a tedious and error-prone task, today we rarely notice when we solve a linear equation, e.g., by using the backslash operator in MATLAB in the expression `A\b`, because computational linear algebra is such a reliable and mature technology.

8.3.2 Nonlinear Root-Finding Problems

A more difficult situation occurs when a nonlinear equation system $R(z) = 0$ needs to be solved, for example, in each step of an implicit Runge-Kutta method, or in nonlinear optimization. Depending

on the problem, one can usually not even be sure that a solution z^0 with $R(z^0) = 0$ exists. And if one has found a solution, one usually cannot be sure that it is the only one. Despite these theoretical difficulties with nonlinear root-finding problems, they are nearly as widely formulated and solved in science and engineering as linear equation systems.

In this section we therefore consider a continuously differentiable function $R : \mathbb{R}^{n_z} \to \mathbb{R}^{n_z}$, $z \mapsto R(z)$, where our aim is to solve the nonlinear equation

$$R(z) = 0$$

Nearly all algorithms to solve this system derive from an algorithm called *Newton's method* or *Newton-Raphson method* that is accredited to Isaac Newton (1643–1727) and Joseph Raphson (about 1648–1715), but which was first described in its current form by Thomas Simpson (1710–1761). The idea is to start with an initial guess z_0, and to generate a sequence of iterates $(z_k)_{k=0}^{\infty}$ by linearizing the nonlinear equation at the current iterate

$$R(z_k) + \frac{\partial R}{\partial z}(z_k)(z - z_k) = 0$$

This equation is a linear system in the variable z, and if the Jacobian $J(z_k) := \frac{\partial R}{\partial z}(z_k)$ is invertible, we can explicitly compute the next iterate as

$$z_{k+1} = z_k - J(z_k)^{-1} R(z_k)$$

Here, we use the notation $J(z_k)^{-1}R(z_k)$ as a shorthand for the algorithm that solves the linear system $J(z_k)\Delta z = R(z_k)$. In the actual computation of a Newton step, the inverse $J(z_k)^{-1}$ is never computed, but only a LU-decomposition of $J(z_k)$, and a forward and a back substitution, as described in the previous subsection.

More generally, we can use an invertible approximation M_k of the Jacobian $J(z_k)$, leading to the *Newton-type methods*. The general Newton-type method iterates according to

$$z_{k+1} = z_k - M_k^{-1} R(z_k)$$

Depending on how closely M_k approximates $J(z_k)$, the local convergence can be fast or slow, or the sequence may even not converge. The advantages of using an M_k that is different from $J(z_k)$ could be that it can be chosen to be invertible even if $J(z_k)$ is not, or that computation of M_k, or of its factorization, can be cheaper. For example, one could

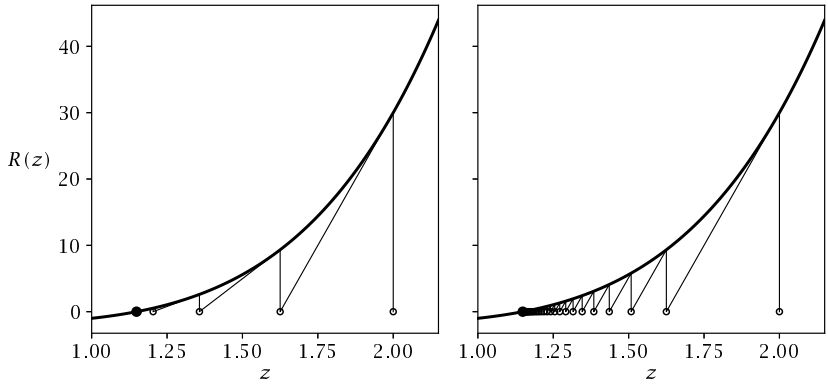


Figure 8.5: Newton-type iterations for solution of $R(z) = 0$ from Example 8.5. Left: exact Newton method. Right: constant Jacobian approximation.

reuse one matrix and its factorization throughout several Newton-type iterations.

Example 8.5: Finding a fifth root with Newton-type iterations

We find the zero of $R(z) = z^5 - 2$ for $z \in \mathbb{R}$. Here, the derivative is $\frac{\partial R}{\partial z}(z) = 5z^4$, such that the Newton method iterates

$$z_{k+1} = z_k - (5z_k^4)^{-1}(z_k^5 - 2)$$

When starting at $z_0 = 2$, the first step is given by $z_1 = 2 - (80)^{-1}(32 - 2) = 13/8$, and the following iterates quickly converge to the solution z^* with $R(z^*) = 0$, as visualized in Figure 8.5 on the left side.

Alternatively, we could use a Jacobian approximation $M_k \approx J(z_k)$, e.g., the constant value $M_k = 80$ corresponding to the true Jacobian at $z = 2$. The resulting iteration would be

$$z_{k+1} = z_k - (80)^{-1}(z_k^5 - 2)$$

When started at $z_0 = 2$ the first iteration would be the same as for Newton's method, but then the Newton-type method with constant Jacobian produces a different sequence, as can be seen on the right side of Figure 8.5. Here, the approximate method also converges; but in general,

when does a Newton-type method converge, and when it converges, how quickly? □

8.3.3 Local Convergence of Newton-Type Methods

Next we investigate the conditions on $R(z)$, z_0 and on M_k required to ensure local convergence of Newton-type iterations. In particular we discuss the speed of convergence. In fact, even if we assume that a sequence of iterates $z_k \in \mathbb{R}^n$ converges to a solution point z^*, i.e., if $z_k \to z^*$, the rate of convergence can be painstakingly slow or lightning fast. The speed of convergence can make the difference between a method being useful or useless for practical computations. Mathematically speaking, a sequence (z_k) is said to converge *q-linearly* if there exists a positive integer k_0 and a positive real number $c_{\max} < 1$, and sequence $(c_k)_{k_0}^{\infty}$ such that for all $k \geq k_0$ holds that $c_k \leq c_{\max}$ and that

$$|z_{k+1} - z^*| \leq c_k |z_k - z^*| \tag{8.10}$$

If in addition, $c_k \to 0$, the sequence is said to converge *q-superlinearly*. If in addition, $c_k = O(|z_k - z^*|)$, the sequence is said to converge *q-quadratically*.[2]

Example 8.6: Convergence rates

We discuss and visualize four examples with $z_k \in (0, \infty)$ and $z_k \to 0$, see Figure 8.6.

- $z_k = \frac{1}{2^k}$ converges q-linearly: $\frac{z_{k+1}}{z_k} = \frac{1}{2}$
- $z_k = 0.99^k$ also converges q-linearly: $\frac{z_{k+1}}{z_k} = 0.99$. This example converges very slowly. In practice we desire $c_{\max}$ to be smaller than, say, $\frac{1}{2}$
- $z_k = \frac{1}{k!}$ converges q-superlinearly, as $\frac{z_{k+1}}{z_k} = \frac{1}{k+1}$
- $z_k = \frac{1}{2^{2^k}}$ converges q-quadratically, because $\frac{z_{k+1}}{(z_k)^2} = \frac{(2^{2^k})^2}{2^{2^{k+1}}} = 1 < \infty$. For $k = 6$, $z_k = \frac{1}{2^{64}} \approx 0$. This is a typical feature of q-quadratic convergence: often, convergence up to machine precision is obtained in about six iterations. □

[2] The historical prefix "q" stands for "quotient," to distinguish it from a weaker form of convergence that is called "r-convergence," where "r" stands for "root."

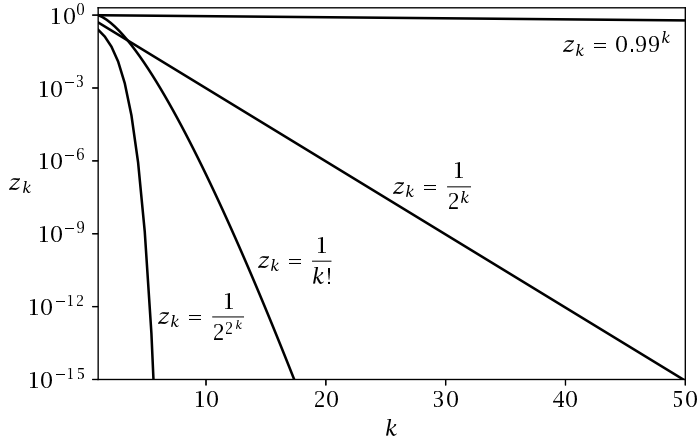


Figure 8.6: Convergence of different sequences as a function of k.

Local convergence of a Newton-type method can be guaranteed by the following classical result (see, e.g., Bock (1983) or Deuflhard (2011)), which also specifies the rate of convergence.

Theorem 8.7 (Local contraction for Newton-type methods)**.** *Regard a nonlinear continuously differentiable function $R : D \to \mathbb{R}^{n_z}$ defined on an open domain $D \subset \mathbb{R}^{n_z}$ and a solution point $z^* \in D$ with $R(z^*) = 0$. We start the Newton-type iteration with the initial guess $z_0 \in D$ and iterate according to $z_{k+1} = z_k - M_k^{-1} R(z_k)$. The sequence (z_k) converges at least q-linearly to z^* and obeys the contraction inequality*

$$|z_{k+1} - z^*| \leq \left(\kappa_k + \frac{\omega}{2} |z_k - z^*|\right) |z_k - z^*| \tag{8.11}$$

if there exist constants $\omega \in [0, \infty)$, $\kappa_{\max} \in [0, 1)$, and a sequence $(\kappa_k)_{k=0}^{\infty}$ with $\kappa_k \in [0, \kappa_{\max}]$, that satisfy for all z_k and all $z \in D$ the following two inequalities

$$\left|M_k^{-1}(J(z_k) - J(z))\right| \leq \omega |z_k - z| \qquad \textit{(Lipschitz condition)}$$

$$\left|M_k^{-1}(J(z_k) - M_k)\right| \leq \kappa_k \qquad \textit{(compatibility condition)}$$

and if the ball $B := \left\{z \in \mathbb{R}^{n_z} \mid |z - z^*| < \frac{2(1-\kappa_{\max})}{\omega}\right\}$ is completely contained in D and if $z_0 \in B$. If in addition $\kappa_k \to 0$, the sequence converges q-superlinearly. If in addition $\kappa_k = O(|z_k - z^*|)$ or even $\kappa_{\max} = 0$, the sequence converges q-quadratically.

Corollary 8.8 (Convergence of exact Newton's method)**.** *For an exact Newton's method, the convergence rate is q-quadratic, because we have* $M_k = J(z_k)$*, i.e.,* $\kappa_{\max} = 0$*.*

8.3.4 Affine Invariance

An iterative method to solve a root-finding problem $R(z) = 0$ is called *affine invariant* if affine basis transformations of the equations or variables do not change the resulting iterations. This is an important property in practice. It is not unreasonable to ask that a good numerical method should behave the same if it is applied to problems formulated in different units or coordinate systems.

The exact Newton method is affine invariant, and also some popular Newton-type optimization methods like the Gauss-Newton method for nonlinear least squares problems share this property. Their affine invariance makes them insensitive to the chosen problem scaling, and this is one reason why they are successful in practice. On the other hand, a method that is not affine invariant usually needs careful scaling of the model equations and decision variables to work well.

8.3.5 Globalization for Newton-Type Methods

The iterations of a Newton-type method can be regarded the trajectory of a nonlinear discrete time system, and the solution z^0 a fixed point. This system is autonomous if M_k is constant or a function of z, i.e., if $M_k = M(z_k)$. In this case, the discrete time system is given by $z^+ = f(z)$ with $f(z) := z - M(z)^{-1}R(z)$. When designing the Newton-type method, one usually wants the solution z^0 to be a stable fixed point with a large area of attraction. Local convergence to this fixed point usually can be guaranteed under conditions stated in Theorem 8.7, in particular if the exact Jacobian is available. On the other hand, the area of attraction for the full-step Newton-type methods described so far is unfortunately not very large in practice, and Newton-type methods usually need extra *globalization* features to make them globally convergent from arbitrary initial guesses. Some globalization techniques are based on a *merit function* that plays the role of a Lyapunov function to be reduced in each iteration; others are based on a *filter* as a measure of merit of a new iterate. To ensure progress from one iteration to the next, some form of *damping* is applied that either reduces the unmodified Newton-type step by doing a *line-search* along the proposed direction, or changes the step computation by adding a *trust-region*

constraint. For a detailed description of globalization techniques, we refer to textbooks on optimization such as Nocedal and Wright (2006).

8.4 Computing Derivatives

Whenever a Newton-type method is used for numerical simulation or optimization, we need to provide derivatives of nonlinear functions that exist as computer code. Throughout this section, we consider a differentiable function $F(u)$ with m inputs and p outputs $y = F(u)$, i.e., a function $F : \mathbb{R}^m \to \mathbb{R}^p$. The main object of interest is the Jacobian $J(u) \in \mathbb{R}^{m \times p}$ of F at the point u, or some of its elements.

Among the many ways to compute the derivatives of $F(u)$, the most obvious would be to apply the known differentiation rules on paper for each of its components, and then to write another computer code by hand that delivers the desired derivatives. This process can become tedious and error prone, but can be automated by using symbolic computer algebra systems such as Maple or Mathematica. This *symbolic differentiation* often works well, but typically suffers from two disadvantages. First, it requires the code to exist in the specific symbolic language. Second, the resulting derivative expressions can become much longer than the original function, such that the CPU time needed to evaluate the Jacobian $J(u)$ by symbolic differentiation can become significantly larger than the CPU time to evaluate $F(u)$.

In contrast, we next present three ways to evaluate the Jacobian $J(u)$ of any computer-represented function $F(u)$ by algorithms that have bounded costs: numerical differentiation, as well as the algorithmic differentiation (AD) in forward mode and in reverse mode. All three ways are based on the evaluation of directional derivatives of the form $J(u)\dot{u}$ with a vector $\dot{u} \in \mathbb{R}^m$ (forward directional derivatives used in numerical differentiation and forward AD) or of the form $\bar{y}'J(u)$ with $\bar{y} \in \mathbb{R}^p$ (reverse directional derivatives used in reverse AD). When unit vectors are used for $\dot{u}$ or $\bar{y}$, the directional derivatives correspond to columns or rows of $J(u)$, respectively. Evaluation of the full Jacobian thus needs either m forward derivatives or p reverse derivatives. Note that in this section, the use of a dot or a bar above a vector as in $\dot{u}$ and $\bar{y}$ just denotes another arbitrary vector with the same dimensions as the original one.

8.4.1 Numerical Differentiation

Numerical differentiation is based on multiple calls of the function $F(u)$ at different input values. In its simplest and cheapest form, it computes a *forward difference* approximation of $J(u)\dot{u}$ for given u and $\dot{u} \in \mathbb{R}^m$ by using a small but finite perturbation size $t_* > 0$ as follows

$$\frac{F(u + t_* \dot{u}) - F(u)}{t_*}$$

The optimal size of t_* for the forward difference approximation depends on the numerical accuracy of the evaluations of F, which we denote by $\epsilon > 0$, and on the relative size of the second derivatives of F compared to F, which we denote by $L > 0$. A detailed derivation leads to the optimal choice

$$t_* \approx \sqrt{\frac{\epsilon}{L}}$$

While ϵ is typically known and given by the machine precision, i.e., $\epsilon = 10^{-16}$ for double-precision floating-point computations, the relative size of the second derivative L is typically not known, but can be estimated. Often, L is just assumed to be of size one, resulting in the choice $t_* = \sqrt{\epsilon}$, i.e., $t_* = 10^{-8}$ for double precision. One can show that the accuracy of the forward derivative approximation is then also given by $\sqrt{\epsilon}$, i.e., one loses half of the valid digits compared to the function evaluation. To compute the full Jacobian $J(u)$, one needs to evaluate m forward differences, for the m seed vectors $\dot{u} = (1, 0, 0, \ldots)'$, $\dot{u} = (0, 1, 0 \ldots)'$, etc. Because the center point can be recovered, one needs in total $(m + 1)$ evaluations of the function F. Thus, we can summarize the cost for computation of the full Jacobian J (as well as the function F) by the statement

$$\text{cost}(F, J) = (1 + m)\,\text{cost}(F)$$

There exists a variety of more accurate, but also more expensive, forms of numerical differentiation, which can be derived from polynomial interpolation of multiple function evaluations of F. The easiest of these are central differences, which are based on a positive and a negative perturbation. Using such higher-order formulas with adaptive perturbation size selection, one can obtain high-accuracy derivatives with numerical differentiation, but at significant cost. One interesting way to actually *reduce* the cost of the numerical Jacobian calculation arises if the Jacobian is known to be sparse, and if many of its columns are structurally orthogonal, i.e., have their nonzero entries at different locations.

To efficiently generate a full Jacobian, one can, for example, use the algorithm by Curtis, Powell, and Reid (1974) that is implemented in the FORTRAN routine TD12 from the HSL Mathematical Software Library (formerly Harwell Subroutine Library). For details of sparse Jacobian evaluations, we refer to the review article by Gebremedhin, Manne, and Pothen (2005).

In summary, and despite the tricks to improve accuracy or efficiency, one has to conclude that numerical differentiation often results in quite inaccurate derivatives, and its only—but practically important—advantage is that it works for any black-box function that can be evaluated on a given computer. Fortunately, there exists a different technology, called AD, that also has tight bounds on the computational cost of the Jacobian evaluation, but avoids the numerical inaccuracies of numerical differentiation. It is often even faster than numerical differentiation, and in the case of reverse derivatives $\bar{y}'J$, it can be tremendously faster. It does so, however, by opening the black box.

8.4.2 Algorithmic Differentiation

We next consider a function $F : \mathbb{R}^m \to \mathbb{R}^p$ that is composed of a sequence of N elementary operations, where an elementary operation acts on only one or two variables. We also introduce a vector $x \in \mathbb{R}^n$ with $n = m + N$ that contains all intermediate variables including the inputs, $x_1 = u_1$, $x_2 = u_2$, $\ldots$ $x_m = u_m$. While the inputs are given before the function is called, each elementary operation generates a new intermediate variable, x_{m+i}, for $i = 1, \ldots, N$. Some of these intermediate variables are used as output $y \in \mathbb{R}^p$ of the code. This decomposition into elementary operations is automatically performed in each executable computer code, and best illustrated with an example.

Example 8.9: Function evaluation via elementary operations

We consider the function

$$F(u_1, u_2, u_3) = \begin{bmatrix} u_1 u_2 u_3 \\ \sin(u_1 u_2) + \exp(u_1 u_2 u_3) \end{bmatrix}$$

with $m = 3$ and $p = 2$. We can decompose this function into $N = 5$ elementary operations that are preceded by m and followed by p

renaming operations, as follows

$$\begin{array}{ll} x_1 & = u_1 \\ x_2 & = u_2 \\ x_3 & = u_3 \\ \hline x_4 & = x_1 x_2 \\ x_5 & = \sin(x_4) \\ x_6 & = x_4 x_3 \\ x_7 & = \exp(x_6) \\ x_8 & = x_5 + x_7 \\ \hline y_1 & = x_6 \\ y_2 & = x_8 \end{array} \tag{8.12}$$

Thus, if the $m = 3$ inputs u_1, u_2, u_3 are given, the $N = 5$ nontrivial elementary operations compute the intermediate quantities $x_4, \ldots, x_8$, and the sixth and eighth of the intermediate quantities are then used as the output $y = F(u)$ of our function. □

The idea of AD is to use the chain rule and differentiate each of the elementary operations separately. There exist two modes of AD, the forward mode and the reverse mode. Both can be derived in a mathematically rigorous way by interpreting the computer function $y = F(u)$ as the output of an implicit function, as explained next.

8.4.3 Implicit Function Interpretation

Let us regard all equations that recursively define the intermediate quantities $x \in \mathbb{R}^n$ for a given $u \in \mathbb{R}^m$ as one large nonlinear equation system

$$G(x, u) = 0 \tag{8.13}$$

with $G : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$. Here, the partial derivative $\frac{\partial G}{\partial x} \in \mathbb{R}^{n \times n}$ is a lower-triangular invertible matrix and $\frac{\partial G}{\partial u} \in \mathbb{R}^{n \times m}$ turns out to be an m-dimensional unit matrix augmented by zeros, which we will denote by B. The function G defines an implicit function $x^* : \mathbb{R}^m \to \mathbb{R}^n$, $u \mapsto x^*(u)$ that satisfies $G(x^*(u), u) = 0$. The output $y = F(u)$ is given by the selection of some entries of $x^*(u)$ via a selection matrix $C \in \mathbb{R}^{p \times n}$, i.e., the computer function is represented by the expression $F(u) = Cx^*(u)$. The derivative $\frac{dx^*}{du}$ of the implicit function satisfies $\frac{\partial G}{\partial x}\frac{dx^*}{du} + \frac{\partial G}{\partial u} = 0$ and is therefore given by

$$\frac{dx^*}{du} = \left(-\frac{\partial G}{\partial x}\right)^{-1} \underbrace{\frac{\partial G}{\partial u}}_{=:B} = \left(-\frac{\partial G}{\partial x}\right)^{-1} B$$

and the Jacobian of F is simply given by $J(u) = C\frac{dx^*}{du}(u)$. The forward directional derivative is given by

$$J(u)\dot{u} = C\underbrace{\left(-\frac{\partial G}{\partial x}\right)^{-1} B\dot{u}}_{=:\dot{x}} = C\dot{x}$$

Here, we have introduced the *dot quantities* $\dot{x}$ that denote the directional derivative of $x^*(u)$ into the direction $\dot{u}$, i.e., $\dot{x} = \frac{dx^*}{du}\dot{u}$. An efficient algorithm to compute $\dot{x}$ corresponds to the solution of a lower-triangular linear equation system that is given by

$$\left(-\frac{\partial G}{\partial x}\right)\dot{x} = B\dot{u} \tag{8.14}$$

Since the matrix $\frac{\partial G}{\partial x}$ is lower triangular, the linear system can be solved by a forward sweep that computes the components of $\dot{x}$ in the same order as the elementary operations, i.e., it first computes $\dot{x}_1$, then $\dot{x}_2$, etc. This leads to the forward mode of AD.

The reverse directional derivative, on the other hand, is given by

$$\bar{y}'J(u) = \underbrace{\bar{y}'\,C\left(-\frac{\partial G}{\partial x}\right)^{-1}}_{=:\bar{x}'} B = \bar{x}'B$$

where we define the *bar quantities* $\bar{x}$ that have a different meaning than the dot quantities. For computing $\bar{x}$, we need to also solve a linear system, but with the transposed system matrix

$$\left(-\frac{\partial G}{\partial x}\right)' \bar{x} = C'\bar{y} \tag{8.15}$$

Due to the transpose, this system involves an upper-triangular matrix and can thus be solved by a reverse sweep, i.e., one first computes $\bar{x}_n$, then $\bar{x}_{n-1}$, etc. This procedure leads to the reverse mode of AD.

Example 8.10: Implicit function representation

Let us regard Example 8.9 and find the corresponding function $G(x, u)$ as well as the involved matrices. The function G corresponds to the

first $n = 8$ rows of (8.12) and is given by

$$G(x,u) = \begin{bmatrix} u_1 - x_1 \\ u_2 - x_2 \\ u_3 - x_3 \\ \hline x_1 x_2 - x_4 \\ \sin(x_4) - x_5 \\ x_4 x_3 - x_6 \\ \exp(x_6) - x_7 \\ x_5 + x_7 - x_8 \end{bmatrix}$$

It is obvious that the nonlinear equation $G(x,u) = 0$ can be solved for any given u by a simple forward elimination of the variables x_1, x_2, ..., yielding the map $x^*(u)$. This fact implies also the lower-triangular structure of the Jacobian $\frac{\partial G}{\partial x}$ which is given by

$$\frac{\partial G}{\partial x} = \left[\begin{array}{cccccccc} -1 & & & & & & & \\ 0 & -1 & & & & & & \\ 0 & 0 & -1 & & & & & \\ \hline x_2 & x_1 & 0 & -1 & & & & \\ 0 & 0 & 0 & \cos(x_4) & -1 & & & \\ 0 & 0 & x_4 & x_3 & 0 & -1 & & \\ 0 & 0 & 0 & 0 & 0 & \exp(x_6) & -1 & \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -1 \end{array}\right]$$

The derivative of G with respect to u is given by a unit matrix to which zero rows are appended, and given by

$$B := \frac{\partial G}{\partial u} = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The identity $y = Cx$ corresponds to the last $p = 2$ rows of (8.12), and the matrix $C \in \mathbb{R}^{p \times n}$ is therefore given by

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The right-hand-side vectors in the equations (8.14) and (8.15) are given by

$$B\dot{u} = \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad C'\bar{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \bar{y}_1 \\ 0 \\ \bar{y}_2 \end{bmatrix}$$

□

8.4.4 Algorithmic Differentiation in Forward Mode

The forward mode of AD computes $\dot{x}$ by solving the lower-triangular linear system (8.14) with a forward sweep. After the trivial definition of the first m components of $\dot{x}$, it goes through all elementary operations in the same order as in the original function to compute the components of $\dot{x}$ one by one. If an original line of code reads $x_k = \phi_k(x_i, x_j)$, the corresponding line to compute $\dot{x}_k$ by forward AD is simply given by

$$\dot{x}_k = \frac{\partial \phi_k}{\partial x_i}(x_i, x_j)\,\dot{x}_i + \frac{\partial \phi_k}{\partial x_j}(x_i, x_j)\,\dot{x}_j$$

In forward AD, the function evaluation and the derivative evaluation can be performed simultaneously, if desired, eliminating the need to store any internal information. The algorithm is best explained by looking again at the example.

Example 8.11: Forward algorithmic differentiation

We differentiate the algorithm from Example 8.9. To highlight the relation to the original code, we list the original command again on the left side, and show the algorithm to compute $\dot{x}$ on the right side. For given $u = [u_1\ u_2\ u_3]'$ and $\dot{u} = [\dot{u}_1\ \dot{u}_2\ \dot{u}_3]'$, the two algorithms proceed as

follows

$$
\begin{array}{ll}
x_1 = u_1 & \dot{x}_1 = \dot{u}_1 \\
x_2 = u_2 & \dot{x}_2 = \dot{u}_2 \\
x_3 = u_3 & \dot{x}_3 = \dot{u}_3 \\
\hline
x_4 = x_1 x_2 & \dot{x}_4 = x_2 \dot{x}_1 + x_1 \dot{x}_2 \\
x_5 = \sin(x_4) & \dot{x}_5 = \cos(x_4)\dot{x}_4 \\
x_6 = x_4 x_3 & \dot{x}_6 = x_3 \dot{x}_4 + x_4 \dot{x}_3 \\
x_7 = \exp(x_6) & \dot{x}_7 = \exp(x_6)\dot{x}_6 \\
x_8 = x_5 + x_7 & \dot{x}_8 = \dot{x}_5 + \dot{x}_7 \\
\hline
y_1 = x_6 & \dot{y}_1 = \dot{x}_6 \\
y_2 = x_8 & \dot{y}_2 = \dot{x}_8
\end{array}
$$

The result of the original algorithm is $y = [y_1\ y_2]'$ and the result of the forward AD sweep is $\dot{y} = [\dot{y}_1\ \dot{y}_2]'$. If desired, one could perform both algorithms in parallel, i.e., evaluate first the left side, then the right side of each row consecutively. This procedure would allow one to delete each intermediate variable and the corresponding dot quantity after its last usage, making the memory demands of the joint evaluation just twice as big as those of the original function evaluation. □

One can see that the dot-quantity evaluations on the right-hand side—which we call a forward sweep—are never longer than about twice the original line of code. This is because each elementary operation depends on at maximum two intermediate variables. More generally, it can be proven that the computational cost of one forward sweep in AD is smaller than a small constant times the cost of a plain function evaluation. This constant depends on the chosen set of elementary operations, but is usually much less than two, so that we conclude

$$\text{cost}(J\dot{u}) \leq 2\,\text{cost}(F)$$

To obtain the full Jacobian J, we need to perform the forward sweep several times, each time with the seed vector corresponding to one of the m unit vectors in $\mathbb{R}^m$. The m forward sweeps all could be performed simultaneously with the evaluation of the function itself, so that one needs in total one function evaluation plus n forward sweeps, i.e., we have

$$\text{cost}(F, J) \leq (1 + 2m)\,\text{cost}(F)$$

This is a conservative bound, and depending on the AD tool used the cost of several combined forward sweeps can be significantly reduced,

and often become much cheaper than a finite difference approximation. Most important, the result of forward AD is exact up to machine precision.

8.4.5 Algorithmic Differentiation in Reverse Mode

The reverse mode of AD computes $\bar{x}$ by solving the upper-triangular linear system (8.15) with a reverse sweep. It does so by first computing the right-hand-side $C'\bar{y}$ vector and initializing all bar quantities with the respective values, i.e., it initially sets $\bar{x} = C'\bar{y}$. Then, the reverse AD algorithm modifies the bar quantities by going through all elementary operations in reverse order. The value of $\bar{x}_i$ is modified for each elementary operation in which x_i is involved. If two quantities x_i and x_j are used in the elementary operation $x_k = \phi_k(x_i, x_j)$, then the corresponding two update equations are given by

$$\bar{x}_i = \bar{x}_i + \bar{x}_k \frac{\partial \phi_k}{\partial x_i}(x_i, x_j) \quad \text{and}$$
$$\bar{x}_j = \bar{x}_j + \bar{x}_k \frac{\partial \phi_k}{\partial x_j}(x_i, x_j)$$

Again, the algorithm is best illustrated with the example.

Example 8.12: Algorithmic differentiation in reverse mode

We consider again the code from Example 8.9. In contrast to before in Example 8.11, now we compute the reverse directional derivative $\bar{y}'J(u)$ for given $[u_1 \ u_2 \ u_3]'$ and $\bar{y} = [\bar{y}_1 \ \bar{y}_2]'$. After the forward evaluation of the function, which is needed to define all intermediate quantities, we need to solve the linear system (8.15) to obtain $\bar{x}$. In the example, this system is explicitly given by

$$\begin{bmatrix} 1 & & -x_2 & & & & & \\ & 1 & -x_1 & & & & & \\ & & 1 & 0 & & -x_4 & & \\ & & & 1 & -\cos(x_4) & -x_3 & & \\ & & & & 1 & 0 & & -1 \\ & & & & & 1 & -\exp(x_6) & 0 \\ & & & & & & 1 & -1 \\ & & & & & & & 1 \end{bmatrix} \begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \bar{x}_3 \\ \bar{x}_4 \\ \bar{x}_5 \\ \bar{x}_6 \\ \bar{x}_7 \\ \bar{x}_8 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \bar{y}_1 \\ 0 \\ \bar{y}_2 \end{bmatrix}$$

To solve this equation without forming the matrix explicitly, we process the elementary operations in reverse order, i.e., one column after the

other, noting that the final result for each $\bar{x}_i$ will be a sum of the right-hand-side vector component $C'\bar{y}$ and a weighted sum of the values $\bar{x}_j$ for those $j > i$ which correspond to elementary operations that have x_i as an input. We therefore initialize all variables by $\bar{x} = C'\bar{y}$, which results for the example in the initialization

$$\begin{aligned} \bar{x}_1 &= 0 & \bar{x}_5 &= 0 \\ \bar{x}_2 &= 0 & \bar{x}_6 &= \bar{y}_1 \\ \bar{x}_3 &= 0 & \bar{x}_7 &= 0 \\ \bar{x}_4 &= 0 & \bar{x}_8 &= \bar{y}_2 \end{aligned}$$

In the reverse sweep, the algorithm updates the bar quantities in reverse order compared to the original algorithm, processing one column after the other.

$$\begin{aligned} &// \text{ differentiation of } x_8 = x_5 + x_7 \\ &\bar{x}_5 = \bar{x}_5 + \bar{x}_8 \\ &\bar{x}_7 = \bar{x}_7 + \bar{x}_8 \\ &// \text{ differentiation of } x_7 = \exp(x_6) \\ &\bar{x}_6 = \bar{x}_6 + \bar{x}_7 \exp(x_6) \\ &// \text{ differentiation of } x_6 = x_4 x_3 \\ &\bar{x}_4 = \bar{x}_4 + \bar{x}_6 x_3 \\ &\bar{x}_3 = \bar{x}_3 + \bar{x}_6 x_4 \\ &// \text{ differentiation of } x_5 = \sin(x_4) \\ &\bar{x}_4 = \bar{x}_4 + \bar{x}_5 \cos(x_4) \\ &// \text{ differentiation of } x_4 = x_1 x_2 \\ &\bar{x}_1 = \bar{x}_1 + \bar{x}_4 x_2 \\ &\bar{x}_2 = \bar{x}_2 + \bar{x}_4 x_1 \end{aligned}$$

At the very end, the algorithm sets

$$\begin{aligned} \bar{u}_1 &= \bar{x}_1 \\ \bar{u}_2 &= \bar{x}_2 \\ \bar{u}_3 &= \bar{x}_3 \end{aligned}$$

to read out the desired result $\bar{y}'J(x) = [\bar{u}_1\ \bar{u}_2\ \bar{u}_3]$. Note that all three of the components are returned by *only one* reverse sweep. □

It can be shown that the cost of one reverse sweep of AD is less than a small constant (which is certainly less than three) times the cost of a

function evaluation, i.e.,

$$\operatorname{cost}(\bar{y}' J) \leq 3 \operatorname{cost}(F)$$

To obtain the full Jacobian of F, we need to call the reverse sweep p times, with the seed vectors corresponding to the unit vectors in $\mathbb{R}^p$, i.e., together with one forward evaluation, we have

$$\operatorname{cost}(F, J) \leq (1 + 3p) \operatorname{cost}(F)$$

Remarkably, reverse AD can compute the full Jacobian at a cost that is independent of the input dimension m. This is particularly advantageous if $p \ll m$, e.g., if we compute the gradient of a scalar function like the objective in optimization. The reverse mode can be much faster than what we can obtain by forward finite differences, where we always need $(m+1)$ function evaluations. For example, to compute the gradient of a scalar function $f : \mathbb{R}^m \to \mathbb{R}$ when $m = 1,000,000$ and each call of the function requires one second of CPU time, the finite difference approximation of the gradient would take $1,000,001$ seconds, while the computation of the same quantity with the backward mode of AD requires only four seconds (one call of the function plus one backward sweep). Thus, besides being more accurate, reverse AD can also be much faster than numerical finite differences. This astonishing fact is also known as the "cheap gradient result" in the AD community, and in the field of neural networks it is exploited in the *back propagation* algorithm. The only disadvantage of the reverse mode of AD is that we have to store all intermediate variables and partial derivatives, in contrast to finite differences or forward AD.

Backward sweep for discrete time optimal control. In numerical optimal control we often have to differentiate a function that is the result of a dynamic system simulation. If the system simulation is in discrete time, one can directly apply the principles of AD to compute the desired derivatives by the forward or the reverse mode. For evaluating the gradient of the objective, the reverse mode is most efficient. If the controls are given by $\mathbf{u} = [u(0)' \ \cdots \ u(N-1)']'$ and the states $x(k)$ are obtained by a discrete time forward simulation of the form $x(k+1) = f(x(k), u(k))$ for $k = 0, \ldots, N-1$ started at $x(0) = x_0$, and if the objective function is given by $J(\mathbf{u}) := \sum_{k=0}^{N-1} \ell(x(k), u(k)) + V(x_N)$, then the backward sweep to compute $\nabla_{\mathbf{u}} J(\mathbf{u})$ performs the following

steps

$$\begin{aligned}
&\bar{x}(N)' = V_x(x(N)) \\
&\text{for} \quad k = N-1, N-2, \ldots, 0 \\
&\qquad \bar{x}(k)' = \ell_x(x(k), u(k)) + \bar{x}(k+1)' f_x(x(k), u(k)) \\
&\qquad \bar{u}(k)' = \ell_u(x(k), u(k)) + \bar{x}(k+1)' f_u(x(k), u(k)) \\
&\text{end}
\end{aligned} \tag{8.16}$$

The output of this algorithm is the vector $\bar{\mathbf{u}} = [\bar{u}(0)' \ \cdots \ \bar{u}(N-1)']'$ which equals the gradient $\nabla_{\mathbf{u}} J(\mathbf{u})$. This method to compute the objective gradient in the sequential approach was well known in the field of optimal control even before the field of algorithmic differentiation developed. From a modern perspective, however, it is simply an application of reverse AD to the algorithm that computes the objective function.

8.4.6 Differentiation of Simulation Routines

When a continuous time system is simulated by numerical integration methods and one wants to compute the derivatives of the state trajectory with respect to initial values or controls, as needed in shooting methods, there are many different approaches and many possible pitfalls. While a complete textbook could be written on the differentiation of just numerical integrators, we present and discuss only three popular approaches here.

External numerical differentiation (END). Probably the simplest approach to differentiate an integrator is to regard the integrator call as a black box, and to compute the desired derivatives by numerical finite differences. Here one computes one nominal trajectory, and one or more perturbed trajectories, depending on the desired number of forward derivatives. This approach, called external numerical differentiation (END), is easy to implement; it is generally not recommended because it suffers from some disadvantages.

- It is typically inaccurate because integrator accuracies ϵ_{int} are well above machine precision, e.g., $\epsilon_{\text{int}} \approx 10^{-6}$, such that the perturbation size needs to be chosen rather large, in particular for adaptive integrators.
- It usually is expensive because each call of the integrator for a perturbed trajectory creates some overhead, such as error control or matrix factorizations, which can be avoided in other approaches.

- It can only compute forward derivatives.

The first disadvantage can be mitigated for explicit integrators with fixed stepsize, where one is allowed to choose smaller perturbation sizes, in the order of the square root of the machine precision. For this special case, END becomes equivalent to the approach described next.

Internal numerical differentiation (IND). The idea behind internal numerical differentiation (IND) (Bock, 1981) is to regard the numerical integrator as a differentiable computer code in the spirit of algorithmic differentiation (AD). Similar to END, it works with perturbed trajectories. What is different from END is that all perturbed trajectories are treated in one single forward sweep, and that all adaptive integrator components are switched off for the perturbed trajectories. Thus, for an adaptive explicit integrator, the stepsize selection works only on the nominal trajectory; once the stepsize is chosen, the same size also is used for all perturbed trajectories.

For implicit integrators, where one performs Newton-type iterations in each step, the philosophy of IND is to choose the sequence of iteration matrices and numbers of Newton-type iterations for only the nominal trajectory, and to regard the iteration matrices as constant for all perturbed trajectories. Because all adaptive components are switched off during the numerical differentiation process, one can regard the integrator code as a function that evaluates its output with machine precision. For this reason, the perturbation size can be chosen significantly smaller than in END. Thus, IND is both more accurate and cheaper than END.

Algorithmic differentiation of integrators. Another approach that is related to IND is to directly apply the principles of AD to the integration algorithm. In an extreme case, one could just take the integrator code and process it with an AD tool—this approach can work well for explicit integrators with fixed stepsize, as we show in Example 8.13, but otherwise needs to be applied with care to avoid the many possible pitfalls of a blind application of AD. In particular, for adaptive integrators, one needs to avoid the differentiation of the stepsize selection procedure. If this simple rule is respected, AD in both forward and reverse modes can be easily applied to adaptive explicit integrators, and is both efficient and yields highly accurate results.

For implicit integrators, one should also regard the number and type of Newton-type iterations in each step as constant. Otherwise, the AD tool also tries to differentiate the Jacobian evaluations and factoriza-

tions, which would create unnecessary overhead. When AD is implemented in this way, i.e., if it respects the same guidelines as the IND approach, its forward mode has similar costs, but yields more accurate derivatives than IND. Depending on input and output dimensions, the reverse mode can accelerate computations further.

8.4.7 Algorithmic and Symbolic Differentiation Software

A crucial property of many AD tools is that they are able to process generic code from a standard programming language like C, C++, MATLAB, or FORTRAN, with no or only minor modifications to the source code. For example, the AD tools ADOL-C and CppAD can process generic user-supplied C or C++ code. This is in contrast to computer algebra systems such as Maple, Mathematica, or MATLAB's Symbolic Math Toolbox, which require the user to define the function to be differentiated using symbolic expressions in a domain-specific language. A further advantage of AD over symbolic differentiation is that it is able to provide tight bounds on the length of the resulting derivative code, as well as its runtime and memory requirements. On the other hand, some symbolic tools—such as AMPL or CasADi—make use of AD internally, so the performance differences between algorithmic and symbolic differentiation can become blurry.

An overview of nearly all available AD tools is given at `www.autodiff.org`. Most AD tools implement both the forward and reverse mode of AD, and allow recursive application of AD to generate higher-order derivatives. Some AD tools automatically perform graph-coloring strategies to reduce the cost of Jacobian evaluations, similar to the sparse numerical differentiation algorithm by Curtis et al. (1974) mentioned before in the context of numerical differentiation. We refer to the textbook on algorithmic differentiation by Griewank and Walther (2008) for an in-depth analysis of the different concepts of AD.

8.4.8 CasADi for Optimization

Many of the computational exercises in this text use the open-source tool CasADi, which implements AD on user-defined symbolic expressions. CasADi also provides standardized interfaces to a variety of numerical routines: simulation and optimization, and solution of linear and nonlinear equations. A key feature of these interfaces is that every user-defined CasADi function passed to a numerical solver automatically provides the necessary derivatives to this solver, without

any additional user input. Often, the result of the numerical solver itself can be interpreted as a differentiable CasADi function, such that derivatives up to any order can be generated without actually differentiating the source code of the solver. Thus, concatenated and recursive calls to numerical solvers are possible and still result in differentiable CasADi functions.

CasADi is written in C++, but allows user input to be provided from either C++, Python, Octave, or MATLAB. When CasADi is used from the interpreter languages Python, Octave, or MATLAB, the user does not have any direct contact with C++; but because the internal handling of all symbolic expressions as well as the numerical computations are performed in a compiled environment, the speed of simulation or optimization computations is similar to the performance of compiled C-code. One particularly powerful optimization solver interfaced to CasADi is IPOPT, an open-source C++ code developed and described by Wächter and Biegler (2006). IPOPT is automatically provided in the standard CasADi installation. For more information on CasADi and how to install it, we refer the reader to `casadi.org`. Here, we illustrate the use of CasADi for optimal control in a simple example.

Example 8.13: Sequential optimal control using CasADi from Octave

In the following example we formulate and solve a simple nonlinear MPC problem. The problem is formulated and solved by the sequential approach in discrete time, but the discrete time dynamics are the result of one step of an integrator applied to a continuous time ordinary differential equation (ODE). We go through the example problem and the corresponding solution using CasADi from Octave, which works without changes from MATLAB. The code is available from the book website as the file `casadi-example-mpc-book-1.m` along with a Python version of the same code, `casadi-example-mpc-book-1.py`.

As a first step, we define the ODE describing the system, which is given by a nonlinear oscillator described by the following ODE with $x \in \mathbb{R}^2$ and $u \in \mathbb{R}$

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \underbrace{\begin{bmatrix} x_2 \\ -x_1 - x_1^3 + u \end{bmatrix}}_{=:f_c(x,u)}$$

with the initial condition $x(0) = [0, 1]'$. We can encode this in Octave as follows

```
% Continuous time dynamics
f_c = @(x, u) [x(2); -x(1) - x(1)^3 + u];
```

To define the discrete time dynamics $x^+ = f(x,u)$, we perform one step of the classical Runge-Kutta method of fourth order. We choose a stepsize of 0.2 seconds. Given $x^+ = f(x,u)$, we can state an MPC optimization problem with zero terminal constraint that we solve, as follows

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{u}}{\text{minimize}} \quad & \sum_{k=0}^{N-1} x(k)' \begin{bmatrix} 10 & 0 \\ 0 & 5 \end{bmatrix} x(k) + u(k)^2 && && \text{(8.17a)}\\
\text{subject to} \quad & x(0) = [1,0]' && && \text{(8.17b)}\\
& x(k+1) = f(x(k),u(k)), && k = 0,1,\ldots,N-1 && \text{(8.17c)}\\
& u(k) \in [-1,1], && k = 0,1,\ldots,N-1 && \text{(8.17d)}\\
& x(N) = [0,0]' && && \text{(8.17e)}
\end{aligned}$$

For its numerical solution, we formulate this problem using the sequential approach, i.e., we regard only $\mathbf{u}$ as optimization variables and eliminate $\mathbf{x}$ by a system simulation. This elimination allows us to generate a cost function $c(\mathbf{u})$ and a constraint function $G(\mathbf{u})$ such that the above problem is equivalent to

$$\begin{aligned}
\underset{\mathbf{u}}{\text{minimize}} \quad & c(\mathbf{u}) && \text{(8.18a)}\\
\text{subject to} \quad & \mathbf{u} \in [-1,1]^N && \text{(8.18b)}\\
& G(\mathbf{u}) = 0 && \text{(8.18c)}
\end{aligned}$$

Here, $c : \mathbb{R}^N \to \mathbb{R}$ and $G : \mathbb{R}^N \to \mathbb{R}^2$, with $N = 50$.

To code this into CasADi/Octave, we begin by declaring a symbolic variable corresponding to $\mathbf{u}$ as follows

```
% Decision variable
N = 50;
U = casadi.SX.sym('U', N);
```

This symbolic variable can be used to construct expressions for c and G

```
% System simulation
xk = [1; 0];
c = 0;
for k=1:N
  % RK4 method
```

```
  dt = 0.2;
  k1 = f_c(xk, U(k));
  k2 = f_c(xk+0.5*dt*k1, U(k));
  k3 = f_c(xk+0.5*dt*k2, U(k));
  k4 = f_c(xk+dt*k3, U(k));
  xk = xk + dt/6.0*(k1 + 2*k2 + 2*k3 + k4);
  % Add contribution to objective function
  c = c + 10*xk(1)^2 + 5*xk(2)^2 + U(k)^2;
end
% Terminal constraint
G = xk - [0; 0];
```

The last remaining step is to pass the expressions for c and G to an optimization solver, more specifically, to the nonlinear programming solver IPOPT. The solver expects an optimization problem with lower and upper bounds for all variables and constraints of the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x_{\text{lb}} \leq x \leq x_{\text{ub}} \\ & g_{\text{lb}} \leq g(x) \leq g_{\text{ub}} \end{aligned} \tag{8.19}$$

To formulate equality constraints in the CasADi syntax for NLPs, one just sets the upper and lower bounds to equal values. The solver also expects an initial guess x_0 for the optimization variables (the initial guess x_0 for the NLP solver is not to be confused with the initial value x_0 for the state trajectory). The interface to the NLP solver uses the keywords `f` and `g` for the functions f and g, `x` for the variables x, `lbx` for x_{lb} etc. The corresponding CasADi code to pass all data to the NLP solver, call it, and retrieve the solution looks as follows.

```
% Create an NLP solver object
nlp = struct('x', U, 'f', c, 'g', G);
solver = casadi.nlpsol('solver', 'ipopt', nlp);
% Solve the NLP
solution = solver('x0', 0, 'lbx', -1, 'ubx', 1,
                  'lbg', 0, 'ubg', 0);
U_opt = solution.x;
```

□

8.5 Direct Optimal Control Parameterizations

Direct optimal control methods transform a continuous time optimal control problem of the form (8.5) into a finite-dimensional optimization

problem. For convenience, we restate the OCP (8.5) in a form that replaces the constraint sets $\mathbb{Z}$ and $\mathbb{X}_f$ by equivalent inequality constraints, as follows

$$\begin{aligned}
\underset{x(\cdot),\,u(\cdot)}{\text{minimize}} \quad & \int_0^T \ell_c(x(t), u(t))\, dt + V_f(x(T)) && \text{(8.20a)}\\
\text{subject to} \quad & x(0) = x_0 && \text{(8.20b)}\\
& \dot{x}(t) = f_c(x(t), u(t)), \quad t \in [0, T] && \text{(8.20c)}\\
& h(x(t), u(t)) \le 0, \qquad t \in [0, T] && \text{(8.20d)}\\
& h_f(x(T)) \le 0 && \text{(8.20e)}
\end{aligned}$$

While the above problem has infinitely many variables and constraints, the idea of direct optimal control methods is to solve instead a related finite-dimensional problem of the general form

$$\begin{aligned}
\underset{w \in \mathbb{R}^{n_w}}{\text{minimize}} \quad & F(w)\\
\text{subject to} \quad & G(x_0, w) = 0\\
& H(w) \le 0
\end{aligned} \qquad \text{(8.21)}$$

This finite-dimensional optimization problem is solved for given initial value x_0 with any of the Newton-type optimization methods described in the following section, Section 8.6. In this section, we are concerned only with the transformation of the continuous problem (8.20) into a finite-dimensional problem of form (8.21).

First, one chooses a finite representation of the continuous functions, which is often called *discretization*. This encompasses three parts of the OCP, namely the control trajectory (which is often represented by a piecewise constant function), the state trajectory (which is often discretized using a numerical integration rule), and the path constraints (which are often only imposed on some grid points). Second, one selects the variables w that are finally passed to the optimization solver. These can be all of the discretization variables (in the fully simultaneous or direct transcription approach), but are often only a subset of the parameters that represent the control and state trajectories. The remaining discretization parameters are hidden to the optimization solver, but are implicitly computed during the optimization computations—such as the state trajectories in the sequential approach, or the intermediate quantities in a Runge-Kutta step. Next we present some of the most widely used direct optimal control parameterizations.

8.5.1 Direct Single Shooting

Like most direct methods, the single-shooting approach first parameterizes the control trajectory with a finite-dimensional vector $\mathbf{q} \in \mathbb{R}^{n_q}$ and sets $u(t) = \tilde{u}(t;\mathbf{q})$ for $t \in [0,T]$. One sometimes calls this step "control vector parameterization." One example for such a function $\tilde{u} : [0,T] \times \mathbb{R}^{n_q} \to \mathbb{R}^m$ is a polynomial of degree p, which requires $(p+1)$ coefficients for each component of $u(t) \in \mathbb{R}^m$. With this choice, the resulting control parameter $\mathbf{q}$ would have the dimension $n_q = (p+1)m$. A disadvantage of the polynomials—as of any other "global" parameterization—is that the inherent problem sparsity due to the dynamic system structure is inevitably lost. For this reason, and also because it better corresponds to the discrete time implementation of MPC, most often one chooses basis functions with local support, for example, a piecewise constant control parameterization. In this case, one divides the time horizon $[0,T]$ into N subintervals $[t_i, t_{i+1}]$ with $0 = t_0 < t_1 < \ldots < t_N = T$, and sets

$$\tilde{u}(t;\mathbf{q}) := q_i \quad \text{for} \quad t \in [t_i, t_{i+1})$$

For each interval, one needs one vector $q_i \in \mathbb{R}^m$, such that the total dimension of $\mathbf{q} = (q_0, q_1, \ldots, q_{N-1})$ is given by $n_q = Nm$. In the following, we assume this form of piecewise constant control parameterization.

Regarding the state discretization, the direct single-shooting method relies on any of the numerical simulation methods described in Section 8.2 to find an approximation $\tilde{x}(t;x_0,\mathbf{q})$ of the state trajectory, given the initial value x_0 at $t = 0$ and the control trajectory $\tilde{u}(t;\mathbf{q})$. Often, adaptive integrators are chosen. In case of piecewise constant controls, the integration needs to stop and restart briefly at the time points t_i to avoid integrating a nonsmooth right-hand-side function. Due to state continuity, the state $\tilde{x}(t_i;x_0,\mathbf{q})$ is both the initial state of the interval $[t_i, t_{i+1}]$ as well as the last state of the previous interval $[t_{i-1}, t_i]$. The control values used in the numerical integrators on both sides differ, due to the jump at t_i, and are given by q_{i-1} and q_i, respectively.

Evaluating the integral in the objective (8.20a) requires an integration rule. One option is to just augment the ODE system with a *quadrature state* $x_{\text{quad}}(t)$ starting at $x_{\text{quad}}(0) = 0$, and obeying the trivial differential equation $\dot{x}_{\text{quad}}(t) = \ell_c(x(t),u(t))$ that can be solved with the same numerical solver as the standard ODE. Another option is to

evaluate $\ell_c(\widetilde{x}(t;x_0,\mathbf{q}),\widetilde{u}(t;\mathbf{q}))$ on some grid and to apply another integration rule that is external with respect to the integrator. For example, one can use a refinement of the grid that was used for the control discretization, where each interval $[t_i,t_{i+1}]$ is divided into M equally sized subintervals $[\tau_{i,j},\tau_{i,j+1}]$ with $\tau_{i,j} := t_i + j/M(t_{i+1}-t_i)$ for $j=0,\ldots,M$ and $i=0,\ldots,N-1$, and just apply a Riemann sum on each interval to yield the objective function

$$F(x_0,\mathbf{q}) := \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} \ell_c(\widetilde{x}(\tau_{i,j};x_0,\mathbf{q}),\widetilde{u}(\tau_{i,j};\mathbf{q}))\ (\tau_{i,j+1}-\tau_{i,j}) + \quad V_f(\widetilde{x}(T;x_0,\mathbf{q}))$$

In the context of the Gauss-Newton method for least squares integrals, this second option is preferable because it allows one to easily obtain a Gauss-Newton Hessian approximation from the sensitivities which are provided by the integrator. Note that the fine grid evaluation as described here requires an integrator able to output the states at arbitrary locations; collocation methods, for example, have this ability. If not, one must select points $\tau_{i,j}$ that coincide with the intermediate steps or stages of the integrator.

The last discretization choice considers the path constraints (8.20d). These often are evaluated on the same grid as the control discretization, or, more generally, on a finer grid, e.g., the time points $\tau_{i,j}$ defined above for the objective integral. Then, only finitely many constraints $h(\widetilde{x}(\tau_{i,j};x_0,\mathbf{q}),\widetilde{u}(\tau_{i,j};\mathbf{q})) \le 0$ are imposed for $j=0,\ldots,M$ and $i=0,1,\ldots,N-1$. Together with the terminal constraint, one defines the inequality constraint function

$$H(x_0,\mathbf{q}) := \begin{bmatrix} h(\widetilde{x}(\tau_{0,0};x_0,\mathbf{q}),\widetilde{u}(\tau_{0,0};\mathbf{q})) \\ h(\widetilde{x}(\tau_{0,1};x_0,\mathbf{q}),\widetilde{u}(\tau_{0,1};\mathbf{q})) \\ \vdots \\ h(\widetilde{x}(\tau_{1,0};x_0,\mathbf{q}),\widetilde{u}(\tau_{1,0};\mathbf{q})) \\ h(\widetilde{x}(\tau_{1,1};x_0,\mathbf{q}),\widetilde{u}(\tau_{1,1};\mathbf{q})) \\ \vdots \\ h(\widetilde{x}(\tau_{N-1,M-1};x_0,\mathbf{q}),\widetilde{u}(\tau_{N-1,M-1};\mathbf{q})) \\ h_f(\widetilde{x}(T;x_0,\mathbf{q})) \end{bmatrix}$$

If the function h maps to $\mathbb{R}^{n_h}$ and h_f to $\mathbb{R}^{n_{h_f}}$, the function H maps to $\mathbb{R}^{(NMn_h+n_{h_f})}$. The resulting finite-dimensional optimization problem in

single shooting is thus given by

$$\begin{aligned} \underset{s_0,\,\mathbf{q}}{\text{minimize}} \quad & F(s_0, \mathbf{q}) \\ \text{subject to} \quad & s_0 - x_0 = 0 \\ & H(s_0, \mathbf{q}) \leq 0 \end{aligned} \tag{8.22}$$

Of course, the trivial equality constraint $s_0 - x_0 = 0$ could easily be eliminated, and this is often done in single-shooting implementations. In the real-time optimization context, however, it is beneficial to include also the parameter x_0 as a trivially constrained variable s_0 of the single-shooting optimization problem, as we do here. This simple trick is called *initial-value embedding*, and allows one to initialize the optimization procedure with the past initial value s_0, for which an approximately optimal solution already exists; it also allows one to easily obtain a linearized feedback control for new values of x_0, as we discuss in the next section. Also, for moving horizon estimation (MHE) problems, one has to keep the (unconstrained) initial value s_0 as an optimization variable in the single-shooting optimization problem formulation.

In summary, the single-shooting method is a fully sequential approach that treats all intermediate state values computed in the numerical integration routine as hidden variables, and solves the optimization problem in the space of control parameters $\mathbf{q} \in \mathbb{R}^{n_q}$ and initial values $s_0 \in \mathbb{R}^n$ only.

There are many different ways to numerically solve the optimization problem (8.22) in the single-shooting approach using standard methods from the field of nonlinear programming. At first sight, the optimization problem in the single-shooting method is dense, and usually problem (8.22) is solved by a dense NLP solver. However, some single-shooting approaches use a piecewise control parameterization and are able to exploit the intrinsic sparsity structure of the OCP in the NLP solution, as discussed in Section 8.8.5.

8.5.2 Direct Multiple Shooting

The direct multiple-shooting method makes exactly the same discretization choices as the single-shooting method with piecewise control discretization, but it keeps the states $s_i \approx x(t_i)$ at the interval boundary time points as decision variables in the finite-dimensional optimization problem. This allows one to completely decouple the numerical integrations on the separate intervals. For simplicity, we regard

again a piecewise constant control parameterization that uses the constant control value $q_i \in \mathbb{R}^m$ on the interval $[t_i, t_{i+1}]$. On the same interval, we then define the N trajectory pieces $\tilde{x}_i(t; s_i, q_i)$ that are the numerical solutions of the initial-value problems

$$\tilde{x}_i(t_i; s_i, q_i) = s_i, \quad \frac{d\tilde{x}_i}{dt}(t; s_i, q_i) = f_c(\tilde{x}_i(t; s_i, q_i), q_i), \quad t \in [t_i, t_{i+1}]$$

for $i = 0, 1, \ldots, N-1$. Note that each trajectory piece only depends on the artificial initial value $s_i \in \mathbb{R}^n$ and the local control parameter $q_i \in \mathbb{R}^m$.

Using again a possibly refined grid on each interval, with time points $\tau_{i,j} \in [t_i, t_{i+1}]$ for $j = 0, \ldots, M$, we can formulate numerical approximations of the objective integrals $\int_{t_i}^{t_{i+1}} \ell_c(\tilde{x}_i(t; s_i, q_i), q_i)\, dt$ on each interval by

$$\ell_i(s_i, q_i) := \sum_{j=0}^{M-1} \ell_c(\tilde{x}_i(\tau_{i,j}; s_i, q_i), q_i)\ (\tau_{i,j+1} - \tau_{i,j})$$

The overall objective is thus given by $\sum_{i=0}^{N-1} \ell_i(s_i, q_i) + V_f(s_N)$. Note that the objective terms $\ell_i(s_i, q_i)$ each depend again only on the local initial values s_i and local controls q_i, and can thus be evaluated independently from each other. Likewise, we discretize the path constraints, for simplicity on the same refined grid, by defining the local inequality constraint functions

$$H_i(s_i, q_i) := \begin{bmatrix} h(\tilde{x}_i(\tau_{0,0}; s_i, q_i), q_i) \\ h(\tilde{x}_i(\tau_{0,1}; s_i, q_i), q_i) \\ \vdots \\ h(\tilde{x}_i(\tau_{0,M-1}; s_i, q_i), q_i) \end{bmatrix}$$

for $i = 0, 1, \ldots, N-1$. These are again independent functions, with $H_i : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^{(Mn_h)}$. Using these definitions, and the concatenations $\mathbf{s} := (s_0, s_1, \ldots, s_N)$ and $\mathbf{q} := (q_0, \ldots, q_{N-1})$, one can state the finite-dimensional optimization problem that is formulated and solved in

the direct multiple-shooting method

$$\begin{aligned}
\underset{\mathbf{s},\mathbf{q}}{\text{minimize}} \quad & \sum_{i=0}^{N-1} \ell_i(s_i, q_i) \; + \; V_f(s_N) && && (8.23a)\\
\text{subject to} \quad & s_0 = x_0 && && (8.23b)\\
& s_{i+1} = \tilde{x}_i(t_{i+1}; s_i, q_i), && \text{for} \quad i = 0, \ldots, N-1 && (8.23c)\\
& H_i(s_i, q_i) \le 0, && \text{for} \quad i = 0, \ldots, N-1 && (8.23d)\\
& h_f(s_N) \le 0 && && (8.23e)
\end{aligned}$$

By a straightforward definition of problem functions F, G, and H, and optimization variables $w = [s_0' \; q_0' \; s_1' \; q_1' \; \cdots \; s_{N-1}' \; q_{N-1}' \; s_N']'$, the above problem can be brought into the form (8.21).

Note that, due to the presence of $\mathbf{s}$ as optimization variables, the problem dimension is higher than in the single-shooting method, namely $n_w = (N+1)n + Nm$ variables compared with only $(n + Nm)$ in the single-shooting method. On the other hand, the additional Nn equality constraints (8.23c) eliminate the additional Nn degrees of freedom, and the problems (8.23) and (8.22) are fully equivalent if the same integration routines are used. Also note that the multiple-shooting NLP (8.23) has exactly the same form as the discrete time optimal control problem (8.1). From this perspective, the single-shooting problem (8.22) is thus identical to the sequential formulation, compare (8.3), and the multiple-shooting problem is identical to the simultaneous formulation, compare (8.1), of the same discrete time OCP.

When comparing the continuous time problem (8.20) with the nonlinear program (NLP) (8.23) in direct multiple shooting, it is interesting to note that the terminal cost and terminal constraint function are identical, while the cost integrals, the system dynamics, and the path constraints are all numerically approximated in the multiple-shooting NLP.

Multiple versus single shooting. The advantages of multiple compared to single shooting are the facts that the evaluation of the integrator calls can be performed in parallel on the different subintervals, that the state values $\mathbf{s}$ can also be used for initialization of the optimization solver, and that the contraction rate of Newton-type optimization iterations is often observed to be faster, in particular for nonlinear and unstable systems. Its disadvantage for problems without state constraints is that globalization strategies cannot simply rely on the objective function as merit function, but have to also monitor

the residuals of the dynamic constraints (8.23c), which can become cumbersome. Some people also prefer the single-shooting method for the simple reason, that, as a sequential approach, it shows "feasible," or more exactly, "physical" state trajectories in each optimization iteration, i.e., trajectories that satisfy, up to numerical integration errors, the system's differential equation.

We argue here, however, that this reason is not valid, because if one wants to see "physical" trajectories during an optimization run, one could numerically simulate and plot the system evolution for the currently best available guess of the control trajectory $\mathbf{q}$ in any simultaneous method at comparably low additional cost. On the other hand, in the presence of state constraints, the iterates of both sequential and simultaneous methods always lead to slightly infeasible state trajectories, while simultaneous methods often converge even faster in this case. Thus, "feasibility" is not really a reason to prefer one approach over the other.

A theoretical comparison of sequential and simultaneous ("lifted") formulations in the context of Newton-type optimization (Albersmeyer and Diehl, 2010) shows that both methods can be implemented with nearly identical computational cost per iteration. Also, it can be shown—and observed in practice—that simultaneous formulations lead to faster contraction rates if the nonlinearities of the concatenated system dynamics reinforce each other, e.g., if an exponential $x_1 = \exp(x_0)$ is concatenated with an exponential $x_2 = \exp(x_1)$, leading to $x_2 = \exp(\exp(x_0))$. On the other hand, the sequential approach would lead to faster contraction if the concatenated nonlinearities mitigate each other, e.g., if a logarithm $x_2 = \log(x_1)$ follows the exponential $x_1 = \exp(x_0)$ and renders the concatenation $x_2 = \log(\exp(x_0)) = x_0$ the identity (a linear map). In optimal control, one often observes that the concatenation reinforces the nonlinearities, which renders the simultaneous approach favorable.

Exact expressions for linear systems with quadratic costs. In the special case of linear systems $f_c(x, u) = A_c x + B_c u$ with quadratic costs $\ell_c(x, u) = x' Q_c x + u' R_c u$, the exact multiple-shooting functions $\tilde{x}_i(t_{i+1}; s_i, q_i)$ and $\ell_i(s_i, q_i)$ also turn out to be linear and quadratic, and it is possible to compute them explicitly. Specifically

$$\tilde{x}_i(t_{i+1}; s_i, q_i) = A s_i + B q_i$$

with

$$A = \exp\left(A_{\mathrm{c}}\left(t_{i+1} - t_i\right)\right) \quad \text{and} \quad B = \int_0^{(t_{i+1}-t_i)} \exp\left(A_{\mathrm{c}}\tau\right) B_{\mathrm{c}}\, d\tau$$

and

$$\ell_i(s_i, q_i) = \begin{bmatrix} s_i \\ q_i \end{bmatrix}' \begin{bmatrix} Q & S \\ S' & R \end{bmatrix} \begin{bmatrix} s_i \\ q_i \end{bmatrix}$$

with more complicated formulas for Q, R, and S that can be found in Van Loan (1978) or Pannocchia, Rawlings, Mayne, and Mancuso (2015). Note that approximations of the above matrices also can be obtained from the differentiation of numerical integration routines that are applied to the linear ODE system, augmented by the quadratic cost integral. The first-order derivatives of the final states yield A and B, and the second-order derivative of the cost gives Q, R, and S. Because these numerical computations can be done before an actual MPC implementation, they can be performed offline and with high accuracy.

8.5.3 Direct Transcription and Collocation Methods

The idea of simultaneous optimal control can be extended even further by keeping all ODE discretization variables as optimization variables. This fully simultaneous approach is taken in the family of *direct transcription methods*, which directly transcribe all data of the continuous time OCP (8.20) into an NLP without making use of numerical integration routines. Instead, they directly formulate the numerical simulation equations as equalities of the optimization problem. One example of a direct transcription method was already given in the introduction of this chapter, in (8.6), where an explicit Euler integration rule was employed. Because the state equations are equality constraints of the optimization problem, direct transcription methods often use implicit integration rules; they offer higher orders for the same number of state discretization variables, and come with better stability properties for stiff systems. Probably the most popular class of direct transcription methods are the direct collocation methods.

Direct transcription by collocation. In direct collocation, the time horizon $[0, T]$ is first divided into a typically large number N of collocation intervals $[t_i, t_{i+1}]$, with $0 = t_0 < t_1 < \ldots < t_N = T$. On each of these intervals, an implicit Runge-Kutta integration rule of collocation type is applied to transcribe the ODE $\dot{x} = f_{\mathrm{c}}(x, u)$ to a finite set of nonlinear equations. For this aim, we first introduce the states $s_i \approx x(t_i)$ at

the time points t_i, and then regard the implicit Runge-Kutta equations with M stages on the interval with length $h_i := (t_{i+1} - t_i)$, which create an implicit relation between s_i and s_{i+1}. We introduce additional variables $K_i := [k'_{i,1} \ \cdots \ k'_{i,M}]' \in \mathbb{R}^{nM}$, where $k_{i,j} \in \mathbb{R}^n$ corresponds to the state derivative at the collocation time point $t_i + c_j h_i$ for $j = 1, \ldots, M$. These variables K_i are uniquely defined by the collocation equations if s_i and the control value $q_i \in \mathbb{R}^m$ are given. We summarize the collocation equations as $G_i^{\mathrm{RK}}(s_i, K_i, q_i) = 0$ with

$$G_i^{\mathrm{RK}}(s_i, K_i, q_i) := \begin{bmatrix} k_{i,1} - f_c(s_i + h_i(a_{11}k_{i,1} + \ldots + a_{1,M}k_{i,M}), q_i) \\ k_{i,2} - f_c(s_i + h_i(a_{21}k_{i,1} + \ldots + a_{2,M}k_{i,M}), q_i) \\ \vdots \\ k_{i,M} - f_c(s_i + h_i(a_{M1}k_{i,1} + \ldots + a_{M,M}k_{i,M}), q_i) \end{bmatrix} \tag{8.24}$$

The transition to the next state is described by $s_{i+1} = F_i^{\mathrm{RK}}(s_i, K_i, q_i)$ with

$$F_i^{\mathrm{RK}}(s_i, K_i, q_i) := s_i + h_i(b_1 k_{i,1} + \ldots + b_M k_{i,M})$$

In contrast to shooting methods, where the controls are often held constant across several integration steps, in direct collocation one usually allows one new control value q_i per collocation interval, as we do here. Even a separate control parameter for every collocation time point within the interval is possible. This would introduce the maximum number of control degrees of freedom that is compatible with direct collocation methods and could be interpreted as a piecewise polynomial control parameterization of order $(M - 1)$.

Derivative versus state representation. In most direct collocation implementations, one uses a slightly different formulation, where the intermediate stage derivative variables $K_i = [k'_{i,1} \ \cdots \ k'_{i,M}]' \in \mathbb{R}^{nM}$ are replaced by the stage state variables $S_i = [s'_{i,1} \ \cdots \ s'_{i,M}]' \in \mathbb{R}^{nM}$ that are related to s_i and K_i via the linear map

$$s_{i,j} = s_i + h_i(a_{j1}k_{i,1} \ldots + a_{j,M}k_{i,M}) \quad \text{for} \quad j = 1, \ldots, M \tag{8.25}$$

If $c_1 > 0$, then the relative time points $(0, c_1, \ldots, c_M)$ are all different, such that the interpolation polynomial through the $(M + 1)$ states $(s_i, s_{i,1}, \ldots, s_{i,M})$ is uniquely defined, which renders the linear map (8.25) from (s_i, K_i) to (s_i, S_i) invertible. Concretely, the values $k_{i,j}$ can be obtained as the time derivatives of the interpolation polynomial at the collocation time points. The inverse map, for $j = 1, \ldots, M$, is given by

$$k_{i,j} = \frac{1}{h_i}\left(D_{j,1}(s_{i,1} - s_i) + \ldots + D_{j,M}(s_{i,M} - s_i)\right) \tag{8.26}$$

Interestingly, the matrix (D_{jl}) is the inverse of the matrix (a_{mj}) from the Butcher tableau, such that $\sum_{j=1}^{M} a_{mj} D_{jl} = \delta_{ml}$. Inserting this inverse map into $G_i^{\mathrm{RK}}(s_i, K_i, q_i)$ from Eq. (8.24) leads to the equivalent root-finding problem $G_i(s_i, S_i, q_i) = 0$ with

$$G_i(s_i, S_i, q_i) :=$$
$$\begin{bmatrix} \frac{1}{h_i}\left(D_{1,1}(s_{i,1} - s_i) + \ldots + D_{1,M}(s_{i,M} - s_i)\right) & - & f_{\mathrm{c}}(s_{i,1}, q_i) \\ \frac{1}{h_i}\left(D_{2,1}(s_{i,1} - s_i) + \ldots + D_{2,M}(s_{i,M} - s_i)\right) & - & f_{\mathrm{c}}(s_{i,2}, q_i) \\ & \vdots & \\ \frac{1}{h_i}\left(D_{M,1}(s_{i,1} - s_i) + \ldots + D_{M,M}(s_{i,M} - s_i)\right) & - & f_{\mathrm{c}}(s_{i,M}, q_i) \end{bmatrix} \tag{8.27}$$

Likewise, inserting the inverse map into $F_i^{\mathrm{RK}}(s_i, K_i, q_i)$ leads to the linear expression

$$F_i(s_i, S_i, q_i) := s_i + \widetilde{b}_1(s_{i,1} - s_i) + \ldots + \widetilde{b}_M(s_{i,M} - s_i)$$

where the coefficient vector $\widetilde{b} \in \mathbb{R}^M$ is obtained from the RK weight vector b by the relation $\widetilde{b} = D'b$. In the special case that $c_M = 1$, for example in Radau IIA collocation methods, the vector $\widetilde{b}$ becomes a unit vector and the simple relation $F_i(s_i, S_i, q_i) = s_{i,M}$ holds. Because the transition from (s_i, K_i) to (s_i, S_i) just amounts to a basis change, affine invariant Newton-type methods lead to identical iterates independent of the chosen parameterization. However, using either the derivative variables K_i or the state variables S_i leads to different sparsity patterns in the Jacobians and higher-order derivatives of the problem functions. In particular, the Hessian of the Lagrangian is typically sparser if the node state variables S_i are used. For this reason, the state representation is more often used than the derivative representation in direct collocation codes.

Direct collocation optimization problem. The objective integrals $\int_{t_i}^{t_{i+1}} \ell_{\mathrm{c}}(\widetilde{x}(t), q_i)\, dt$ on each interval are canonically approximated by a weighted sum of evaluations of ℓ_{c} on the collocation time points, as follows

$$\ell_i(s_i, S_i, q_i) := h_i \sum_{j=1}^{M} b_j \ell_{\mathrm{c}}(s_{i,j}, q_i)$$

Similarly, one might choose to impose the path constraints on all collocation time points, leading to the stage inequality function

$$H_i(s_i, S_i, q_i) := \begin{bmatrix} h(s_{i,1}, q_i) \\ h(s_{i,2}, q_i) \\ \vdots \\ h(s_{i,M}, q_i) \end{bmatrix}$$

The finite-dimensional optimization problem to be solved in direct collocation has as optimization variables the sequence of external states $\mathbf{s} := (s_0, s_1, \ldots, s_N)$, the sequence of the internal states $\mathbf{S} := (S_0, S_1, \ldots, S_{N-1})$ as well as the sequence of local control parameters, $\mathbf{q} := (q_0, q_1, \ldots, q_{N-1})$, and is formulated as follows

$$\begin{aligned}
\underset{\mathbf{s},\mathbf{S},\mathbf{q}}{\text{minimize}} \quad & \sum_{i=0}^{N-1} \ell_i(s_i, S_i, q_i) + V_f(s_N) && && \text{(8.28a)} \\
\text{subject to} \quad & s_0 = x_0 && && \text{(8.28b)} \\
& s_{i+1} = F_i(s_i, S_i, q_i), && \text{for} \quad i = 0, \ldots, N-1 && \text{(8.28c)} \\
& 0 = G_i(s_i, S_i, q_i), && \text{for} \quad i = 0, \ldots, N-1 && \text{(8.28d)} \\
& H_i(s_i, S_i, q_i) \leq 0, && \text{for} \quad i = 0, \ldots, N-1 && \text{(8.28e)} \\
& h_f(s_N) \leq 0 && && \text{(8.28f)}
\end{aligned}$$

One sees that the above nonlinear programming problem in direct collocation is similar to the NLP (8.23) arising in the direct multiple-shooting method, but is augmented by the intermediate state variables $\mathbf{S}$ and the corresponding algebraic constraints (8.28d). Typically, it is sparser, but has more variables than the multiple-shooting NLP, not only because of the presence of $\mathbf{S}$, but also because N is larger since it equals the total number of collocation intervals, each of which corresponds to one integration step in a shooting method. Typically, one chooses rather small stage orders M, e.g., two or three, and large numbers for N, e.g., 100 or 1000. The NLPs arising in the direct collocation method are large but sparse. If the sparsity is exploited in the optimization solver, direct collocation can be an extremely efficient optimal control method. For this reason, it is widely used.

Pseudospectral methods. The *pseudospectral optimal control method* can be regarded a special case of the direct collocation method, where only one collocation interval ($N = 1$) is chosen, but with a high-order M. By increasing the order M, one can obtain arbitrarily

high solution accuracies in case of smooth trajectories. The state trajectory is represented by one global polynomial of order M that is uniquely determined by the initial value s_0 and the M collocation node values $s_{0,1}, \ldots, s_{0,M}$. In this approach, the controls are typically parameterized by one parameter per collocation node, i.e., by M distinct values $q_{0,1}, \ldots, q_{0,M}$, such that the control trajectories can be regarded to be represented by global polynomials of order $(M-1)$. One gains a high approximation order, but at the cost that the typical sparsity of the direct collocation problem is lost.

8.6 Nonlinear Optimization

After the finite-dimensional optimization problem is formulated, it needs to be solved. From now on, we assume that a nonlinear program (NLP) of the form (8.21) is formulated, with variable $w \in \mathbb{R}^{n_w}$ and parameter $x_0 \in \mathbb{R}^n$, which we restate here for convenience.

$$\begin{aligned} \underset{w \in \mathbb{R}^{n_w}}{\text{minimize}} \quad & F(w) \\ \text{subject to} \quad & G(x_0, w) = 0 \\ & H(w) \leq 0 \end{aligned} \tag{8.29}$$

As before, we call the above optimization problem $\mathbb{P}_N(x_0)$ to indicate its dependence on the parameter x_0 and on the horizon length N. The aim of the optimization procedure is to reliably and efficiently find an approximation of the solution $w^0(x_0)$ of $\mathbb{P}_N(x_0)$ for a given value of x_0. Inside the MPC loop, the optimization solver is confronted with a sequence of related values of the parameter x_0, a fact that can be exploited in online optimization algorithms to improve speed and reliability compared to standard offline optimization algorithms.

Assumptions and definitions. In this chapter, we make only two assumptions on $\mathbb{P}_N(x_0)$: first, that all problem functions are at least twice continuously differentiable, and second, that the parameter x_0 enters the equalities G linearly, such that the Jacobian matrices G_x and G_w are independent of x_0. This second assumption is satisfied for all problem formulations from the previous sections, because the initial value enters only via the initial-value constraint $s_0 - x_0 = 0$. If one would encounter a problem where the parametric dependence is nonlinear, one could always use the same trick that we used in the single-shooting method and introduce a copy of the parameter as an additional optimization variable s_0—which becomes part of w—and constrain it by

the additional constraint $s_0 - x_0 = 0$. Throughout the section, we often make use of the linearization $H_L(\cdot; \bar{w})$ of a function $H(\cdot)$ at a point $\bar{w}$, i.e., its first-order Taylor series, as follows

$$H_L(w; \bar{w}) := H(\bar{w}) + H_w(\bar{w})(w - \bar{w})$$

Due to the linear parameter dependence of G, its Jacobian does not depend on x_0, such that we can write

$$G_L(x_0, w; \bar{w}) = G(x_0, \bar{w}) + G_w(\bar{w})(w - \bar{w})$$

We also heavily use the Lagrangian function defined by

$$\mathcal{L}(x_0, w, \lambda, \mu) := F(w) + \lambda' G(x_0, w) + \mu' H(w) \tag{8.30}$$

whose gradient and Hessian matrix with respect to w are often used. Again, they do not depend on x_0, and can thus be written as $\nabla_w \mathcal{L}(w, \lambda, \mu)$ and $\nabla_w^2 \mathcal{L}(w, \lambda, \mu)$. Note that the dimensions of the *multipliers*, or *dual variables* λ and μ, equal the output dimensions of the functions G and H, which we denote by n_G and n_H. We sometimes call $w \in \mathbb{R}^{n_w}$ the *primal* variable. At a feasible point w, we say that an inequality with index $i \in \{1, \ldots, n_H\}$ is *active* if and only if $H_i(w) = 0$. The linear independence constraint qualification (LICQ) is satisfied if and only if the gradients of all active inequalities, $\nabla_w H_i(w) \in \mathbb{R}^{n_w}$, and the gradients of the equality constraints, $\nabla_w G_j(w) \in \mathbb{R}^{n_w}$ for $j \in \{1, \ldots, n_G\}$, form a linearly independent set of vectors.

8.6.1 Optimality Conditions and Perturbation Analysis

The first-order necessary conditions for optimality of the above optimization problem are known as the Karush-Kuhn-Tucker (KKT) conditions, which are formulated as follows.

Theorem 8.14 (KKT conditions)**.** *If w^0 is a local minimizer of the optimization problem $\mathbb{P}_N(x_0)$ defined in* (8.29) *and if LICQ holds at w^0, then there exist multiplier vectors λ^0 and μ^0 such that*

$$\nabla_w \mathcal{L}(w^0, \lambda^0, \mu^0) = 0 \tag{8.31a}$$

$$G(x_0, w^0) = 0 \tag{8.31b}$$

$$0 \geq H(w^0) \perp \mu^0 \geq 0 \tag{8.31c}$$

Here, the last condition, known as the *complementarity condition*, states not only that all components of $H(w^0)$ are negative and all components of μ^0 are positive, but also that the two vectors are orthogonal,

which implies that the products $\mu_i^0 H_i(w^0)$ are zero for each $i \in \{1, \ldots, n_H\}$. Thus, each pair $(H_i(w^0), \mu_i^0) \in \mathbb{R}^2$ must be an element of a nonsmooth, L-shaped subset of $\mathbb{R}^2$ that comprises only the negative x-axis, the positive y-axis, and the origin.

Any triple (w^0, λ^0, μ^0) that satisfies the KKT conditions (8.31) and LICQ is called a KKT point, independent of local optimality.

In general, the existence of multipliers such that the KKT conditions (8.31) hold is just a necessary condition for local optimality of a point w^0 at which LICQ holds. Only in the special case that the optimization problem is convex, the KKT conditions can be shown to be both a necessary and a sufficient condition for global optimality. For the general case, we need to formulate additional conditions on the second-order derivatives of the problem functions to arrive at sufficient conditions for local optimality. This is only possible after making a few definitions.

Strictly active constraints and null space basis. At a KKT point (w, λ, μ), an active constraint with index $i \in \{1, \ldots, n_H\}$ is called *weakly active* if and only if $\mu_i = 0$ and *strictly active* if $\mu_i > 0$. Note that for weakly active constraints, the pair $(H_i(w), \mu_i)$ is located at the origin, i.e., at the nonsmooth point of the L-shaped set. For KKT points without weakly active constraints, i.e., when the inequalities are either strictly active or inactive, we say that the *strict complementarity* condition is satisfied.

Based on the division into weakly and strictly active constraints, one can construct the linear space $\mathcal{Z}$ of directions in which the strictly active constraints and the equality constraints remain constant up to first order. This space $\mathcal{Z}$ plays an important role in the second-order sufficient conditions for optimality that we state below, and can be defined as the null space of the matrix that is formed by putting the transposed gradient vectors of all equality constraints and all strictly active inequality constraints on top of each other. To define this properly at a KKT point (w, λ, μ), we reorder the inequality constraints such that

$$H(w) = \begin{bmatrix} H^+(w) \\ H^0(w) \\ H^-(w) \end{bmatrix}$$

In this reordered view on the function $H(w)$, the strictly active inequality constraints $H^+(w)$ come first, then the weakly active constraints $H^0(w)$, and finally the inactive constraints $H^-(w)$. Note that the output dimensions of the three functions add to n_H. The set $\mathcal{Z} \subset \mathbb{R}^{n_w}$ is

now defined as null space of the matrix

$$A := \begin{bmatrix} G_w(w) \\ H_w^+(w) \end{bmatrix} \in \mathbb{R}^{n_A \times n_w}$$

One can regard an orthogonal basis matrix $Z \in \mathbb{R}^{n_w \times (n_w - n_A)}$ of $\mathcal{Z}$ that satisfies $AZ = 0$ and $Z'Z = I$ and whose columns span $\mathcal{Z}$. This allows us to compactly formulate the following sufficient conditions for optimality.

Theorem 8.15 (Strong second-order sufficient conditions for optimality)**.** *If (w^0, λ^0, μ^0) is a KKT point and if the Hessian of its Lagrangian is positive definite on the corresponding space $\mathcal{Z}$, i.e., if*

$$Z' \nabla_w^2 \mathcal{L}(w^0, \lambda^0, \mu^0) Z > 0 \tag{8.32}$$

then the point w^0 is a local minimizer of problem $\mathbb{P}_N(x_0)$.

We call a KKT point that satisfies the conditions of Theorem 8.15 a *strongly regular* KKT point. We should mention that there exists also a weaker form of second-order sufficient conditions. We prefer to work with the stronger variant because it does not only imply optimality but also existence of neighboring solutions $w^0(x_0)$ as a function of the parameter x_0. Moreover, the solution map $w^0(x_0)$ is directionally differentiable, and the directional derivative can be obtained by the solution of a quadratic program, as stated in the following theorem that summarizes standard results from parametric optimization (Robinson, 1980; Guddat, Vasquez, and Jongen, 1990) and is proven in the specific form below in Diehl (2001).

Theorem 8.16 (Tangential predictor by quadratic program)**.** *If $(\bar{w}, \bar{\lambda}, \bar{\mu})$ is a strongly regular KKT point for problem $\mathbb{P}_N(\bar{x}_0)$ (i.e., it satisfies the conditions of Theorem 8.15) then there is a neighborhood $\mathcal{N} \subset \mathbb{R}^n$ around $\bar{x}_0$ such that for each $x_0 \in \mathcal{N}$ the problem $\mathbb{P}_N(x_0)$ has a local minimizer and corresponding strongly regular KKT point $(w^0(x_0), \lambda^0(x_0), \mu^0(x_0))$. Moreover, the map from $x_0 \in \mathcal{N}$ to $(w^0(x_0), \lambda^0(x_0), \mu^0(x_0))$ is directionally differentiable at $\bar{x}_0$, and the directional derivative can be obtained by the solution of the following quadratic program*

$$\begin{aligned} \underset{w \in \mathbb{R}^{n_w}}{\text{minimize}} \quad & F_{\mathrm{L}}(w; \bar{w}) + \frac{1}{2}(w - \bar{w})' \nabla_w^2 \mathcal{L}(\bar{w}, \bar{\lambda}, \bar{\mu})(w - \bar{w}) \\ \text{subject to} \quad & G_{\mathrm{L}}(x_0, w; \bar{w}) = 0 \\ & H_{\mathrm{L}}(w; \bar{w}) \le 0 \end{aligned} \tag{8.33}$$

More specifically, the solution $(w^{\mathrm{QP}}(x_0), \lambda^{\mathrm{QP}}(x_0), \mu^{\mathrm{QP}}(x_0))$ *of the above QP satisfies*

$$\left\| \begin{bmatrix} w^{\mathrm{QP}}(x_0) - w^0(x_0) \\ \lambda^{\mathrm{QP}}(x_0) - \lambda^0(x_0) \\ \mu^{\mathrm{QP}}(x_0) - \mu^0(x_0) \end{bmatrix} \right\| = O(|x_0 - \bar{x}_0|^2)$$

8.6.2 Nonlinear Optimization with Equalities

When we solve an optimization problem without inequalities, the KKT conditions simplify to

$$\begin{aligned} \nabla_w \mathcal{L}(w^0, \lambda^0) &= 0 \\ G(x_0, w^0) &= 0 \end{aligned}$$

This is a smooth root-finding problem that can be summarized as $R(x_0, z) = 0$ with $z = [w' \ \lambda']'$. Interestingly, if one regards the Lagrangian $\mathcal{L}$ as a function of x_0 and z, we have $R(x_0, z) = \nabla_z \mathcal{L}(x_0, z)$. The classical *Newton-Lagrange method* addresses the above root-finding problem by a Newton iteration of the form

$$z_{k+1} = z_k + \Delta z_k \quad \text{with} \quad R_z(z_k)\Delta z_k = -R(x_0, z_k) \tag{8.35}$$

To simplify notation and avoid that the iteration index k interferes with the indices of the optimization variables, we usually use the following notation for the Newton step

$$z^+ = \bar{z} + \Delta z \quad \text{with} \quad R_z(\bar{z})\Delta z = -R(x_0, \bar{z}) \tag{8.36}$$

Here, the old iterate and linearization point is called $\bar{z}$ and the new iterate z^+. The square Jacobian matrix $R_z(z)$ that needs to be factorized in each iteration to compute Δz has a particular structure and is given by

$$R_z(z) = \begin{bmatrix} \nabla_w^2 \mathcal{L}(w, \lambda) & G_w(w)' \\ G_w(w) & 0 \end{bmatrix}$$

This matrix is called the *KKT matrix* and plays an important role in all constrained optimization algorithms. The KKT matrix is invertible at a point z if the LICQ condition holds, i.e., $G_w(w)$ has rank n_G, and if the Hessian of the Lagrangian is positive definite on the null space of $G_w(w)$, i.e., if $Z' \nabla_w^2 \mathcal{L}(w, \lambda, \mu) Z > 0$, for Z being a null space basis. The matrix $Z' \nabla_w^2 \mathcal{L}(w, \lambda, \mu) Z$ is also called the *reduced Hessian*. Note that the KKT matrix is invertible at a strongly regular point, as well

as in a neighborhood of it, such that Newton's method is locally well defined. The KKT matrix is the second derivative of the Lagrangian $\mathcal{L}$ with respect to the primal-dual variables z, and is therefore symmetric. For this reason, it has only real eigenvalues, but it is typically indefinite. At strongly regular KKT points, it has n_w positive and n_G negative eigenvalues.

Quadratic program interpretation and tangential predictors. A particularly simple optimization problem arises if the objective function is linear quadratic, $F(w) = b'w + (1/2)w'Bw$, and the constraint linear, $G(w) = a + Aw$. In this case, we speak of a quadratic program (QP), and the KKT conditions of the QP directly form a linear system in the variables $z = [w' \ \lambda']'$, namely

$$\begin{bmatrix} B & A' \\ A & 0 \end{bmatrix} \begin{bmatrix} w \\ \lambda \end{bmatrix} = - \begin{bmatrix} b \\ a \end{bmatrix}$$

Due to the equivalence of the KKT conditions of the QP with a linear system one can show that the new point $z^+ = \bar{z} + \Delta z$ in the Newton iteration for the nonlinear problem (8.34) also can be obtained as the solution of a QP

$$\begin{aligned} \underset{w \in \mathbb{R}^{n_w}}{\text{minimize}} \quad & F_{\mathrm{L}}(w; \bar{w}) + \frac{1}{2}(w - \bar{w})' B_{\mathrm{ex}}(\bar{z})(w - \bar{w}) \\ \text{subject to} \quad & G_{\mathrm{L}}(x_0, w; \bar{w}) = 0 \end{aligned} \tag{8.37}$$

with $B_{\mathrm{ex}}(\bar{z}) := \nabla^2_w \mathcal{L}(\bar{w}, \bar{\lambda}, \bar{\mu})$. If the primal-dual solution of the above QP is denoted by w^{QP} and λ^{QP}, one can easily show that setting $w^+ := w^{\mathrm{QP}}$ and $\lambda^+ := \lambda^{\mathrm{QP}}$ yields the same step as the Newton iteration. The interpretation of the Newton step as a QP is not particularly relevant for equality constrained problems, but becomes a powerful tool in the context of inequality constrained optimization. It directly leads to the family of sequential quadratic programming (SQP) methods, which are treated in Section 8.7.1. One interesting observation is that the QP (8.37) is identical to the QP (8.33) from Theorem 8.16, and thus its solution cannot only be used as a Newton step for a fixed value of x_0, but it can also deliver a tangential predictor for changing values of x_0. This property is used extensively in continuation methods for nonlinear MPC, such as the real-time iteration presented in Section 8.9.2.

8.6.3 Hessian Approximations

Even though the reduced exact Hessian is guaranteed to be positive definite at regular points, it can become indefinite at nonoptimal points.

In that case the Newton's method would fail because the KKT matrix would become singular in one iteration. Also, the evaluation of the exact Hessian can be costly. For this reason, Newton-type optimization methods approximate the exact Hessian matrix $B_{\mathrm{ex}}(\bar{z})$ by an approximation $\bar{B}$ that is typically positive definite or at least positive semidefinite, and solve the QP

$$\begin{array}{ll} \underset{w \in \mathbb{R}^{n_w}}{\text{minimize}} & F_{\mathrm{L}}(w;\bar{w}) + \frac{1}{2}(w-\bar{w})'\bar{B}(w-\bar{w}) \\ \text{subject to} & G_{\mathrm{L}}(x_0, w;\bar{w}) = 0 \end{array} \tag{8.38}$$

in each iteration. These methods can be generalized to the case of inequality constrained optimization problems and then fall into the class of sequential quadratic programming (SQP) methods.

The local convergence rate of Newton-type optimization methods can be analyzed directly with the tools from Section 8.3.3. Since the difference between the exact KKT matrix $J(z_k)$ and the Newton-type iteration matrix M_k is due only to the difference in the Hessian approximation, Theorem 8.7 states that convergence can occur only if the difference $B_{\mathrm{ex}}(z_k) - \bar{B}_k$ is sufficiently small, and that the linear contraction factor $\kappa_{\max}$ directly depends on this difference and becomes zero if the exact Hessian is used. Thus, the convergence rate for an exact Hessian SQP method is quadratic, and superlinear convergence occurs if the difference between exact and approximate Hessian shrinks to zero in the relevant directions. Note that the algorithms described in this and the following sections only approximate the Hessian matrix, but evaluate the exact constraint Jacobian $G_w(\bar{w})$ in each iteration.

The generalized Gauss-Newton method. One particularly useful Hessian approximation is possible if the objective function $F(w)$ is a sum of squared residuals, i.e., if

$$F(w) = (1/2)\,|M(w)|^2$$

for a differentiable function $M : \mathbb{R}^{n_w} \to \mathbb{R}^{n_M}$. In this case, the exact Hessian $B_{\mathrm{ex}}(\bar{z})$ is given by

$$\underbrace{M_w(\bar{w})' M_w(\bar{w})}_{=:B_{\mathrm{GN}}(\bar{w})} + \sum_{j=1}^{n_M} M_j(\bar{w})\nabla^2 M_j(\bar{w}) + \sum_{i=1}^{n_G} \bar{\lambda}_i \nabla^2 G_i(\bar{w})$$

By taking only the first part of this expression, one obtains the *Gauss-Newton Hessian approximation* $B_{\mathrm{GN}}(\bar{w})$, which is by definition always

a positive semidefinite matrix. In the case that $M_w(\bar{w}) \in \mathbb{R}^{n_M \times n_w}$ has rank n_w, i.e., if $n_M \geq n_w$ and the n_w columns are linearly independent, the Gauss-Newton Hessian $B_{\mathrm{GN}}(\bar{w})$ is even positive definite. Note that $B_{\mathrm{GN}}(\bar{w})$ does not depend on the multipliers λ, but the error with respect to the exact Hessian does. This error would be zero if both the residuals $M_j(\bar{w})$ and the multipliers λ_i are zero. Because both can be shown to be small at a strongly regular solution with small objective function $(1/2)\,|M(w)|^2$, the Gauss-Newton Hessian $B_{\mathrm{GN}}(\bar{w})$ is a good approximation for problems with small residuals $|M(w)|$.

When the Gauss-Newton Hessian $B_{\mathrm{GN}}(\bar{w})$ is used within a constrained optimization algorithm, as we do here, the resulting algorithm is often called the *generalized Gauss-Newton method* (Bock, 1983). Newton-type optimization algorithms with Gauss-Newton Hessian converge only linearly, but their contraction rate can be surprisingly fast in practice, in particular for problems with small residuals. The QP subproblem that is solved in each iteration of the generalized Gauss-Newton method can be shown to be equivalent to

$$\begin{aligned} \underset{w \in \mathbb{R}^{n_w}}{\text{minimize}} \quad & (1/2)\,|M_{\mathrm{L}}(w;\bar{w})|^2 \\ \text{subject to} \quad & G_{\mathrm{L}}(x_0, w;\bar{w}) = 0 \end{aligned} \tag{8.39}$$

A particularly simple instance of the generalized Gauss-Newton method arises if the objective function is itself already a positive definite quadratic function, i.e., if $F(w) = (1/2)(w - w_{\mathrm{ref}})'B(w - w_{\mathrm{ref}})$. In this case, one could define $M(w) := B^{\frac{1}{2}}(w - w_{\mathrm{ref}})$ to see that the QP subproblem has the same objective as the NLP. Generalizing this approach to nonquadratic, but convex, objectives and convex constraint sets, leads to the class of sequential convex programming methods as discussed and analyzed in Tran-Dinh, Savorgnan, and Diehl (2012).

Hessian update methods. Another way to obtain a cheap and positive definite Hessian approximation $\bar{B}$ for Newton-type optimization is provided by Hessian update methods. In order to describe them, we recall the iteration index k to the primal-dual variables $z_k = [w_k' \; \lambda_k']'$ and the Hessian matrix B_k at the k-th iteration, such that the QP to be solved in each iteration is described by

$$\begin{aligned} \underset{w \in \mathbb{R}^{n_w}}{\text{minimize}} \quad & F_{\mathrm{L}}(w; w_k) + \frac{1}{2}(w - w_k)'B_k(w - w_k) \\ \text{subject to} \quad & G_{\mathrm{L}}(x_0, w; w_k) = 0 \end{aligned} \tag{8.40}$$

In a full step method, the primal-dual solution w_k^{QP} and λ_k^{QP} of the above QP is used as next iterate, i.e., $w_{k+1} := w_k^{\mathrm{QP}}$ and $\lambda_{k+1} := \lambda_k^{\mathrm{QP}}$. A Hessian update formula uses the previous Hessian approximation B_k and the Lagrange gradient evaluations at w_k and w_{k+1} to compute the next Hessian approximation B_{k+1}. Inspired from a directional derivative of the function $\nabla_w \mathcal{L}(\cdot, \lambda_{k+1})$ in the direction $s_k := (w_{k+1} - w_k)$, which, up-to-first order, should be equal to the finite difference approximation $y_k := \nabla_w \mathcal{L}(w_{k+1}, \lambda_{k+1}) - \nabla_w \mathcal{L}(w_k, \lambda_{k+1})$, all Hessian update formulas require the *secant condition*

$$B_{k+1} s_k = y_k$$

One particularly popular way of the many ways to obtain a matrix B_{k+1} that satisfies the secant condition is given by the Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula, which sets

$$B_{k+1} := B_k - \frac{B_k s_k s_k' B_k}{s_k' B_k s_k} + \frac{y_k y_k'}{y_k' s_k}$$

One often starts the update procedure with a scaled unit matrix, i.e., sets $B_0 := \alpha I$ with some $\alpha > 0$. It can be shown that for a positive definite B_k and for $y_k' s_k > 0$, the matrix B_{k+1} resulting from the BFGS formula is also positive definite. In a practical implementation, to ensure positive definiteness of B_{k+1}, the unmodified update formula is only applied if $y_k' s_k$ is sufficiently large, say if the inequality $y_k' s_k \geq \beta s_k' B_k s_k$ is satisfied with some $\beta \in (0, 1)$, e.g., $\beta = 0.2$. If it is not satisfied, the update can either be skipped, i.e., one sets $B_{k+1} := B_k$, or the vector y_k is first modified and then the BFGS update is performed with this modified vector. An important observation is that the gradient difference y_k can be computed with knowledge of the first-order derivatives of F and G at w_k and w_{k+1}, which are needed to define the linearizations F_{L} and G_{L} in the QP (8.40) at the current and next iteration point. Thus, a Hessian update formula does not create any additional costs in terms of derivative computations compared to a fixed Hessian method (like, for example, steepest descent); but it typically improves the convergence speed significantly. One can show that Hessian update methods lead to superlinear convergence under mild conditions.

8.7 Newton-Type Optimization with Inequalities

The necessary optimality conditions for an equality constrained optimization problem form a smooth system of nonlinear equations in the

primal-dual variables, and can therefore directly be addressed by Newton's method or its variants. In contrast to this, the KKT conditions for inequality constrained problems contain the complementarity conditions (8.31c), which define an inherently nonsmooth set in the primal-dual variable space, such that Newton-type methods can be applied only after some important modifications. In this section, we present two widely used classes of methods, namely sequential quadratic programming (SQP) and nonlinear interior point (IP) methods.

8.7.1 Sequential Quadratic Programming

Sequential quadratic programming (SQP) methods solve in each iteration an inequality constrained quadratic program (QP) that is obtained by linearizing all problem functions

$$\begin{aligned} \underset{w \in \mathbb{R}^{n_w}}{\text{minimize}} \quad & F_{\mathrm{L}}(w; w_k) + \frac{1}{2}(w - w_k)' B_k (w - w_k) \\ \text{subject to} \quad & G_{\mathrm{L}}(x_0, w; w_k) = 0 \\ & H_{\mathrm{L}}(w; w_k) \leq 0 \end{aligned} \tag{8.41}$$

The above QP is a quadratic approximation of the nonlinear problem $\mathbb{P}_N(x_0)$, and is denoted by $\mathbb{P}_N^{\mathrm{QP}}(x_0; w_k, B_k)$ to express its dependence on the linearization point w_k and the choice of Hessian approximation B_k. In the full-step SQP method, the primal-dual solution $z_k^{\mathrm{QP}} = (w_k^{\mathrm{QP}}, \lambda_k^{\mathrm{QP}}, \mu_k^{\mathrm{QP}})$ of the QP $\mathbb{P}_N^{\mathrm{QP}}(x_0; w_k, B_k)$ is directly taken as the next iterate, $z_{k+1} = (w_{k+1}, \lambda_{k+1}, \mu_{k+1})$, i.e., one sets $z_{k+1} := z_k^{\mathrm{QP}}$. Note that the multipliers $(\lambda_{k+1}, \mu_{k+1})$ only have an influence on the next QP via the Hessian approximation B_{k+1}, and can be completely discarded in case a multiplier-free Hessian approximation such as a Gauss-Newton Hessian is used.

The solution of an inequality constrained QP is a nontrivial task, but for convex QP problems there exist efficient and reliable algorithms that are just treated here as a black box. To render the QP subproblem convex, one often chooses positive semidefinite Hessian approximations B_k.

Active set detection and local convergence. A crucial property of SQP methods is that the set of active inequalities (the *active set*, in short) is discovered inside the QP solver, and that the active set can change significantly from one SQP iteration to the next. However, one can show that the QP solution discovers the correct active set when

the linearization point w_k is close to a strongly regular solution of the NLP (8.29) at which strict complementarity holds. Thus, in the vicinity of the solution, the active set remains stable, and, therefore, the SQP iterates become identical to the iterates of a Newton-type method for equality constrained optimization applied to a problem where all active constraints are treated as equalities, and where all other inequalities are discarded. Therefore, the local convergence results for general Newton-type methods can be applied; and the SQP method shows quadratic convergence in case of an exact Hessian, superlinear convergence in case of Hessian updates, and linear convergence in case of a Gauss-Newton Hessian.

Generalized tangential predictors in SQP methods. An appealing property of SQP methods for problems that depend on a parameter x_0 is that they deliver a generalized tangential predictor, even at points where the active set changes, i.e., where strict complementarity does not hold. More precisely, it is easily seen that the QP $\mathbb{P}_N^{\mathrm{QP}}(x_0; \bar{w}, \bar{B})$ formulated in an SQP method, with exact Hessian $\bar{B} = \nabla^2 \mathcal{L}(\bar{z})$ at a strongly regular solution $\bar{z} = (\bar{w}, \bar{\lambda}, \bar{\mu})$ of problem $\mathbb{P}_N(\bar{x}_0)$, delivers the tangential predictor of Theorem 8.16 for neighboring problems $\mathbb{P}_N(x_0)$ with $x_0 \neq \bar{x}_0$ (Diehl, 2001). A disadvantage of SQP methods is that they require in each iteration the solution of an inequality constrained QP, which is more expensive than solution of a linear system.

8.7.2 Nonlinear Interior Point Methods

Nonlinear interior point (IP) methods remove the nonsmoothness of the KKT conditions by formulating an approximate, but smooth root-finding problem. This smooth problem corresponds to the necessary optimality conditions of an equality constrained optimization problem that is an approximation of the original problem. In a first and trivial step, the nonlinear inequalities $H(w) \leq 0$ are reformulated into equality constraints $H(w) + s = 0$ by introduction of a slack variable $s \in \mathbb{R}^{n_H}$ that is required to be positive, such that the equivalent new problem has bounds of the form $s \geq 0$ as its only inequality constraints. In the second and crucial step, these bounds are replaced by a barrier term of the form $-\tau \sum_{i=1}^{n_H} \log s_i$ with $\tau > 0$ that is added to the objective. This leads to a different and purely equality constrained optimization

problem given by

$$\begin{aligned} \underset{w,\,s}{\text{minimize}} \quad & F(w) - \tau \sum_{i=1}^{n_H} \log s_i \\ \text{subject to} \quad & G(x_0, w) = 0 \\ & H(w) + s = 0 \end{aligned} \tag{8.42}$$

For $\tau \to 0$, the barrier term $-\tau \log s_i$ becomes zero for any strictly positive $s_i > 0$ while it always grows to infinity for $s_i \to 0$, i.e., on the boundary of the feasible set. Thus, for $\tau \to 0$, the barrier function would be a perfect indicator function of the true feasible set and one can show that the solution of the modified problem (8.42) tends to the solution of the original problem (8.29) for $\tau \to 0$. For any positive $\tau > 0$, the necessary optimality conditions of problem (8.42) are a smooth set of equations, and can, if we denote the multipliers for the equalities $H(w) + s = 0$ by $\mu \in \mathbb{R}^{n_H}$ and keep the original definition of the Lagrangian from (8.30), be equivalently formulated as

$$\nabla_w \mathcal{L}(w, \lambda, \mu) = 0 \tag{8.43a}$$

$$G(x_0, w) = 0 \tag{8.43b}$$

$$H(w) + s = 0 \tag{8.43c}$$

$$\mu_i s_i = \tau \quad \text{for} \quad i = 1, \ldots, n_H \tag{8.43d}$$

Note that for $\tau > 0$, the last condition (8.43d) is a smooth version of the complementarity condition $0 \le s \perp \mu \ge 0$ that would correspond to the KKT conditions of the original problem after introduction of the slack variable s.

A nonlinear IP method proceeds as follows: it first sets τ to a rather large value, and solves the corresponding root-finding problem (8.43) with a Newton-type method for equality constrained optimization. During these iterations, the implicit constraints $s_i > 0$ and $\mu_i > 0$ are strictly enforced by shortening the steps, if necessary, to avoid being attracted by spurious solutions of $\mu_i s_i = \tau$. Then, it slowly reduces the barrier parameter τ; for each new value of τ, the Newton-type iterations are initialized with the solution of the previous problem.

Of course, with finitely many Newton-type iterations, the root-finding problems for decreasing values of τ can only be solved approximately. In practice, one often performs only one Newton-type iteration per problem, i.e., one iterates while one changes the problem. Here, we have sketched the primal-dual IP method as it is for example

implemented in the NLP solver IPOPT (Wächter and Biegler, 2006); but there exist many other variants of nonlinear interior point methods. IP methods also exist in variants that are tailored to linear or quadratic programs and IP methods also can be applied to other convex optimization problems such as second-order cone programs or semidefinite programs (SDP). For these convex IP algorithms, one can establish polynomial runtime bounds, which unfortunately cannot be established for the more general case of nonlinear IP methods described here.

Nonlinear IP methods with fixed barrier parameter. Some variants of nonlinear IP methods popular in the field of nonlinear MPC use a fixed positive barrier parameter τ throughout all iterations, and therefore solve a modified MPC problem. The advantage of this approach is that a simple and straightforward Newton-type framework for equality constrained optimization can be used out of the box. The disadvantage is that for a large value of τ, the modified MPC problem is a conservative approximation of the original MPC problem; for a small value of τ, the nonlinearity due to the condition (8.43d) is severe and slows down the convergence of the Newton-type procedure. Interestingly, these nonlinear IP variants are sometimes based on different barrier functions than the logarithmic barrier described above; they use slack formulations that make violation of the implicit constraint $s_i \geq 0$ impossible by setting, for example, $s_i = (t_i)^2$ with new slacks t_i. This last variant is successfully used for nonlinear MPC by Ohtsuka (2004), and modifies the original problem to a related problem of the form

$$\begin{aligned} \underset{w,t}{\text{minimize}} \quad & F(w) - \tau \sum_{i=1}^{n_H} t_i \\ \text{subject to} \quad & G(x_0, w) = 0 \\ & H_i(w) + (t_i)^2 = 0, \quad i = 1, \ldots, n_H \end{aligned} \tag{8.44}$$

which is then solved by a tailored Newton-type method for equality constrained optimization.

8.7.3 Comparison of SQP and Nonlinear IP Methods

While SQP methods need to solve a QP in each iteration, nonlinear IP methods only solve a linear system of similar size in each iteration, which is cheaper. Some SQP methods even solve the QP by an interior point method, and then perform about 10-30 inner iterations—each of

which is as expensive as the linear system solution in a nonlinear IP method.

On the other hand, the cost per iteration for both SQP and nonlinear IP methods also comprises the evaluation of the problem functions and their derivatives. The number of high-level iterations required to reach a desired level of accuracy is often smaller for SQP methods than for nonlinear IP methods. Also, SQP methods are better at warmstarting, which is particularly important in the context of nonlinear MPC. Roughly speaking, for an NLP with cheap function and derivative evaluations, as in direct collocation, and if no good initial guess is provided, a nonlinear IP method is preferable. An SQP method would be favorable in case of expensive function evaluations, as in direct single or multiple shooting, and when good initial guesses can be provided, for example, if a sequence of neighboring problems is solved.

8.8 Structure in Discrete Time Optimal Control

When a Newton-type optimization method is applied to an optimal control problem, the dynamic system constraints lead to a specific sparsity structure in the KKT matrix. And the quadratic program (QP) in the Newton-type iteration corresponds to a linear quadratic (LQ) optimal control problem with time-varying matrices. To discuss this structure in detail, consider an unconstrained discrete time OCP as it arises in the direct multiple-shooting method

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & \sum_{i=0}^{N-1} \ell_i(x_i, u_i) \; + \; V_f(x_N) \\ \text{subject to} \quad & \bar{\bar{x}}_0 - x_0 = 0 \\ & f_i(x_i, u_i) - x_{i+1} = 0 \quad \text{for} \quad i = 0, \ldots, N-1 \end{aligned} \tag{8.45}$$

Here, the vector $w \in \mathbb{R}^{(N+1)n+Nm}$ of optimization variables is given by $w = [x_0' \; u_0' \; \cdots \; x_{N-1}' \; u_{N-1}' \; x_N']'$. The fixed vector $\bar{\bar{x}}_0$ is marked by two bars to distinguish it from the optimization variable x_0, as well as from a specific value $\bar{x}_0$ of x_0 that is used as linearization point in a Newton-type algorithm. We introduce also a partitioned vector of Lagrange multipliers, $\lambda = [\lambda_0' \; \lambda_1' \; \ldots \; \lambda_N']'$, with $\lambda \in \mathbb{R}^{(N+1)n}$, such that

the Lagrangian of the problem is given by

$$\mathcal{L}(\bar{\bar{x}}_0, w, \lambda) = \lambda_0'(\bar{\bar{x}}_0 - x_0) + \sum_{i=0}^{N-1} \ell_i(x_i, u_i) + \lambda_{i+1}'(f_i(x_i, u_i) - x_{i+1}) + V_f(x_N)$$

As before, we can combine w and λ to a vector $z \in \mathbb{R}^{2(N+1)n+Nm}$ of all primal-dual variables. Interestingly, the exact Hessian matrix $B_{\mathrm{ex}}(z) = \nabla_w^2 \mathcal{L}(z)$ is block diagonal (Bock and Plitt, 1984), because the Lagrangian function $\mathcal{L}$ is a sum of independent terms that each depend only on a small subset of the variables—a property called *partial separability*. The exact Hessian is easily computed to be a matrix with the structure

$$B_{\mathrm{ex}}(\bar{z}) = \begin{bmatrix} Q_0 & S_0' & & & & \\ S_0 & R_0 & & & & \\ & & \ddots & & & \\ & & & Q_{N-1} & S_{N-1}' & \\ & & & S_{N-1} & R_{N-1} & \\ & & & & & P_N \end{bmatrix} \tag{8.46}$$

where the blocks with index i, only depend on the primal variables with index i and the dual variables with index $(i+1)$. More specifically, for $i = 0, \ldots, N-1$ the blocks are readily shown to be given by

$$\begin{bmatrix} Q_i & S_i' \\ S_i & R_i \end{bmatrix} = \nabla^2_{(x_i,u_i)}[\ell_i(x_i, u_i) + \lambda_{i+1}' f_i(x_i, u_i)]$$

8.8.1 Simultaneous Approach

Most simultaneous Newton-type methods for optimal control preserve the block diagonal structure of the exact Hessian $B_{\mathrm{ex}}(\bar{z})$ and also of the Hessian approximation $\bar{B}$. Thus, the linear quadratic optimization problem (8.38) that is solved in one iteration of a Newton-type optimization method for a given linearization point $\bar{w} = [\bar{x}_0' \ \bar{u}_0' \ \cdots \ \bar{x}_{N-1}' \ \bar{u}_{N-1}' \ \bar{x}_N']'$ and a given Hessian approximation $\bar{B}$ is identical to the following time-varying LQ optimal control problem

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & \sum_{i=0}^{N-1} \ell_{\mathrm{QP},i}(x_i, u_i; \bar{w}, \bar{B}) + V_{\mathrm{QP},f}(x_N; \bar{w}, \bar{B}) \\ \text{subject to} \quad & \bar{\bar{x}}_0 - x_0 = 0 \\ & f_{\mathrm{L},i}(x_i, u_i; \bar{x}_i, \bar{u}_i) - x_{i+1} = 0 \quad \text{for} \quad i = 0, \ldots, N-1 \end{aligned} \tag{8.47}$$

Here, the quadratic objective contributions $\ell_{\mathrm{QP},i}(x_i, u_i; \bar{w}, \bar{B})$ are given by

$$\ell_i(\bar{x}_i, \bar{u}_i) + \nabla_{(s,q)}\ell_i(\bar{x}_i, \bar{u}_i)' \begin{bmatrix} x_i - \bar{x}_i \\ u_i - \bar{u}_i \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_i - \bar{x}_i \\ u_i - \bar{u}_i \end{bmatrix}' \begin{bmatrix} \bar{Q}_i & \bar{S}_i' \\ \bar{S}_i & \bar{R}_i \end{bmatrix} \begin{bmatrix} x_i - \bar{x}_i \\ u_i - \bar{u}_i \end{bmatrix}$$

the terminal cost $V_{\mathrm{QP},f}(x_N; \bar{w}, \bar{B})$ is given by

$$V_f(\bar{x}_N) + \nabla V_f(\bar{x}_N)'[x_N - \bar{x}_N] + (1/2)[x_N - \bar{x}_N]'\bar{P}_N[x_N - \bar{x}_N]$$

and the linearized constraint functions $f_{\mathrm{L},i}(x_i, u_i; \bar{x}_i, \bar{u}_i)$ are simply given by

$$f_i(\bar{x}_i, \bar{u}_i) + \underbrace{\frac{\partial f_i}{\partial s}(\bar{x}_i, \bar{u}_i)}_{=:\bar{A}_i}[x_i - \bar{x}_i] + \underbrace{\frac{\partial f_i}{\partial q}(\bar{x}_i, \bar{u}_i)}_{=:\bar{B}_i}[u_i - \bar{u}_i]$$

To create a banded structure, it is advantageous to order the primal-dual variable vector as $z = [\lambda_0' \; x_0' \; u_0' \; \cdots \; \lambda_{N-1}' \; x_{N-1}' \; u_{N-1}' \; \lambda_N' \; x_N']'$; then the solution of the above LQ optimal control problem at iterate $\bar{z}$ corresponds to the solution of a block-banded linear system $\bar{M}_{\mathrm{KKT}} \cdot (z - \bar{z}) = -\nabla_z \mathcal{L}(\bar{x}_0, \bar{z})$, which we can write equivalently as

$$\bar{M}_{\mathrm{KKT}} \cdot z = -\bar{r}_{\mathrm{KKT}} \tag{8.48}$$

where the residual vector is given by $\bar{r}_{\mathrm{KKT}} := \nabla_z \mathcal{L}(\bar{x}_0, \bar{z}) - \bar{M}_{\mathrm{KKT}}\bar{z}$. The matrix $\bar{M}_{\mathrm{KKT}}$ is an approximation of the block-banded KKT matrix $\nabla_z^2 \mathcal{L}(\bar{z})$ and given by

$$\bar{M}_{\mathrm{KKT}} = \begin{bmatrix} 0 & -I & & & & & & & & \\ -I & \bar{Q}_0 & \bar{S}_0' & \bar{A}_0' & & & & & & \\ & \bar{S}_0 & \bar{R}_0 & \bar{B}_0' & & & & & & \\ & \bar{A}_0 & \bar{B}_0 & 0 & -I & & & & & \\ & & & -I & \ddots & & & & & \\ & & & & & \bar{Q}_{N-1} & \bar{S}_{N-1}' & \bar{A}_{N-1}' & & \\ & & & & & \bar{S}_{N-1} & \bar{R}_{N-1} & \bar{B}_{N-1}' & & \\ & & & & & \bar{A}_{N-1} & \bar{B}_{N-1} & 0 & -I & \\ & & & & & & & -I & & \\ & & & & & & & & \bar{P}_N & \end{bmatrix} \tag{8.49}$$

Ignoring the specific block structure, this is a banded symmetric matrix with bandwidth $(2n + m)$ and total size $N(2n+m) + 2n$, and the

linear system can thus in principle be solved using a banded LDLT-factorization routine at a cost that is linear in the horizon length N and cubic in $(2n + m)$. There exists a variety of even more efficient solvers for this form of KKT systems with smaller runtime and smaller memory footprint. Many of these solvers exploit the specific block-banded structure of the LQ optimal control problem. Some of these solvers are based on the backward Riccati recursion, as introduced in Section 1.3.3 and Section 6.1.1, and described in Section 8.8.3 for the time-varying case.

8.8.2 Linear Quadratic Problems (LQP)

Consider a time-varying LQ optimal control problem of the form

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{u}}{\text{minimize}} \quad & \sum_{i=0}^{N-1} \begin{bmatrix} \bar{q}_i \\ \bar{r}_i \end{bmatrix}' \begin{bmatrix} x_i \\ u_i \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_i \\ u_i \end{bmatrix}' \begin{bmatrix} \bar{Q}_i & \bar{S}_i' \\ \bar{S}_i & \bar{R}_i \end{bmatrix} \begin{bmatrix} x_i \\ u_i \end{bmatrix} + \bar{p}_N' x_N + \frac{1}{2} x_N' \bar{P}_N x_N \\
\text{subject to} \quad & \bar{x}_0 - x_0 = 0 \\
& \bar{b}_i + \bar{A}_i x_i + \bar{B}_i u_i - x_{i+1} = 0 \quad \text{for} \quad i = 0, \ldots, N-1
\end{aligned} \tag{8.50}$$

Here, we use the bar above fixed quantities such as $\bar{A}_i, \bar{Q}_i$ to distinguish them from the optimization variables x_i, u_i, and the quantities that are computed during the solution of the optimization problem. This distinction makes it possible to directly interpret problem (8.50) as the LQ approximation (8.47) of a nonlinear problem (8.45) at a given linearization point $\bar{z} = [\bar{\lambda}_0' \; \bar{x}_0' \; \bar{u}_0' \; \cdots \; \bar{\lambda}_{N-1}' \; \bar{x}_{N-1}' \; \bar{u}_{N-1}' \; \bar{\lambda}_N' \; \bar{x}_N']'$ within a Newton-type optimization method. We call the above problem the linear quadratic problem (LQP), and present different solution approaches for the LQP in the following three subsections.

8.8.3 LQP Solution by Riccati Recursion

One band-structure-exploiting solution method for the above linear quadratic optimization problem is called the Riccati recursion. It can easily be derived by dynamic programming arguments. It is given by three recursions—one expensive matrix and two cheaper vector recursions.

First, and most important, we perform a backward matrix recursion which is started at $P_N := \bar{P}_N$, and goes backward through the indices

$i = N-1, \ldots, 0$ to compute $P_{N-1}, \ldots, P_0$ with the following formula

$$P_i := \bar{Q}_i + \bar{A}_i' P_{i+1} \bar{A}_i - (\bar{S}_i' + \bar{A}_i' P_{i+1} \bar{B}_i)(\bar{R}_i + \bar{B}_i' P_{i+1} \bar{B}_i)^{-1} (\bar{S}_i + \bar{B}_i' P_{i+1} \bar{A}_i) \quad (8.51)$$

The only condition for the above matrix recursion formula to be well defined is that the matrix $(\bar{R}_i + \bar{B}_i' P_{i+1} \bar{B}_i)$ is positive definite, which turns out to be equivalent to the optimization problem being well posed (otherwise, problem (8.50) would be unbounded from below). Note that the Riccati matrix recursion propagates symmetric matrices P_i, whose symmetry can and should be exploited for efficient computations.

The second recursion is a vector recursion that also goes backward in time and is based on the matrices $P_0, \ldots, P_N$ resulting from the first recursion, and can be performed concurrently. It starts with $p_N := \bar{p}_N$ and then runs through the indices $i = N-1, \ldots, 0$ to compute

$$p_i := \bar{q}_i + \bar{A}_i' (P_{i+1} \bar{b}_i + p_{i+1}) - (\bar{S}_i' + \bar{A}_i' P_{i+1} \bar{B}_i)(\bar{R}_i + \bar{B}_i' P_{i+1} \bar{B}_i)^{-1} (\bar{s}_i + \bar{B}_i' (P_{i+1} \bar{b}_i + p_{i+1})) \quad (8.52)$$

Interestingly, the result of the first and the second recursion together yield the optimal cost-to-go functions V_i^0 for the states x_i that are given by

$$V_i^0(x_i) = c_i + p_i' x_i + \frac{1}{2} x_i' P_i x_i$$

where the constants c_i are not of interest here. Also, one directly obtains the optimal feedback control laws u_i^0 that are given by

$$u_i^0(x_i) = k_i + K_i x_i$$

with

$$K_i := -(\bar{R}_i + \bar{B}_i' P_{i+1} \bar{B}_i)^{-1} (\bar{S}_i + \bar{B}_i' P_{i+1} \bar{A}_i) \quad \text{and} \quad (8.53a)$$

$$k_i := -(\bar{R}_i + \bar{B}_i' P_{i+1} \bar{B}_i)^{-1} (\bar{s}_i + \bar{B}_i' (P_{i+1} \bar{b}_i + p_{i+1})) \quad (8.53b)$$

Based on these data, the optimal solution to the optimal control problem is obtained by a forward vector recursion that is nothing other than a forward simulation of the linear dynamics using the optimal feedback control law. Thus, the third recursion starts with $x_0 := \bar{x}_0$ and goes through $i = 0, \ldots, N-1$ computing

$$u_i := k_i + K_i x_i \quad (8.54a)$$

$$x_{i+1} := \bar{b}_i + \bar{A}_i x_i + \bar{B}_i u_i \quad (8.54b)$$

For completeness, one would simultaneously also compute the Lagrange multipliers λ_i, which are for $i = 0, \ldots, N$ given by the gradient of the optimal cost-to-go function at the solution

$$\lambda_i := p_i + P_i x_i \tag{8.54c}$$

The result of the three recursions of the Riccati algorithm is a vector $z = [\lambda_0' \; x_0' \; u_0' \; \cdots \; \lambda_{N-1}' \; x_{N-1}' \; u_{N-1}' \; \lambda_N' \; x_N']'$ that solves the linear system $\bar{M}_{\mathrm{KKT}} \cdot z = -\bar{r}_{\mathrm{KKT}}$ with a right-hand side that is given by $\bar{r}_{\mathrm{KKT}} = [\bar{\bar{x}}_0' \; \bar{q}_0' \; \bar{r}_0' \; \bar{b}_0' \; \cdots \; \bar{q}_{N-1}' \; \bar{r}_{N-1}' \; \bar{b}_{N-1}' \; \bar{p}_N']'$.

The matrix recursion (8.51) can be interpreted as a factorization of the KKT matrix $\bar{M}_{\mathrm{KKT}}$, and in an efficient implementation it needs about $N(7/3n^3 + 4n^2m + 2nm^2 + 1/3m^3)$ FLOPs, which is about one-third the cost of a plain banded LDLT-factorization.

On the other hand, the two vector recursions (8.52) and (8.54a)-(8.54c) can be interpreted as a linear system solve with the already factorized matrix $\bar{M}_{\mathrm{KKT}}$. In an efficient implementation, this linear system solve needs about $N(8n^2 + 8nm + 2n^2)$ FLOPs.

If care is taken to reduce the number of memory movements and to optimize the linear algebra operations for full CPU usage, one can obtain significant speedups in the range of one order of magnitude compared to a standard implementation of the Riccati recursion—even for small- and medium-scale dynamic systems (Frison, 2015). With only minor modifications, the Riccati recursion can be used inside an interior point method for inequality constrained optimal control problems.

8.8.4 LQP Solution by Condensing

A different way to exploit the block-sparse structure of the LQ optimal control problem (8.50) is to first eliminate the state trajectory $\mathbf{x} = [x_0' \; x_1' \; \cdots \; x_N']'$ as a function of the initial value $\bar{\bar{x}}_0$ and the control $\mathbf{u} = [u_0' \; u_1' \; \cdots \; u_{N-1}']'$. After subdivision of the variables into states and controls, the equality constraints of the QP (8.50) can be expressed in the following form, where we omit the bar above the system matrices and vectors for better readability

$$\underbrace{\begin{bmatrix} I & & & & \\ -A_0 & I & & & \\ & -A_1 & I & & \\ & & \ddots & \ddots & \\ & & & -A_{N-1} & I \end{bmatrix}}_{=:\mathcal{A}} \mathbf{x} = \underbrace{\begin{bmatrix} 0 \\ b_0 \\ b_1 \\ \vdots \\ b_{N-1} \end{bmatrix}}_{=:b} + \underbrace{\begin{bmatrix} I \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{=:\mathcal{I}} \bar{\bar{x}}_0 + \underbrace{\begin{bmatrix} 0 & & & \\ B_0 & & & \\ & B_1 & & \\ & & \ddots & \\ & & & B_{N-1} \end{bmatrix}}_{=:\mathcal{B}} \mathbf{u}$$

It can easily be shown that the inverse of $\mathcal{A}$ is given by

$$\mathcal{A}^{-1} = \begin{bmatrix} I & & & & \\ A_0 & I & & & \\ A_1 A_0 & A_1 & I & & \\ \vdots & \vdots & \vdots & \ddots & \\ (A_{N-1} \cdots A_0) & (A_{N-1} \cdots A_1) & (A_{N-1} \cdots A_2) & & I \end{bmatrix}$$

and state elimination results in the affine map

$$\mathbf{x} = \mathcal{A}^{-1} b \; + \; \mathcal{A}^{-1} \mathcal{I} \, \bar{x}_0 \; + \; \mathcal{A}^{-1} \mathcal{B} \, \mathbf{u}$$

Using this explicit expression to eliminate all states in the objective results in a condensed, unconstrained quadratic optimization problem of the form

$$\underset{\mathbf{u}}{\text{minimize}} \quad c + \begin{bmatrix} q \\ r \end{bmatrix}' \begin{bmatrix} \bar{x}_0 \\ \mathbf{u} \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \bar{x}_0 \\ \mathbf{u} \end{bmatrix}' \begin{bmatrix} Q & S' \\ S & \mathcal{R} \end{bmatrix} \begin{bmatrix} \bar{x}_0 \\ \mathbf{u} \end{bmatrix} \tag{8.55}$$

that is equivalent to the original optimal control problem (8.50). Condensing algorithms process the vectors and matrices of the sparse problem (8.50) to yield the data of the condensed QP (8.55)—in particular the Hessian $\mathcal{R}$—and come in different variants. One classical condensing algorithm has a cost of about $(1/3)N^3nm^2$ FLOPS; a second variant, that can be derived by applying reverse AD to the quadratic cost function, has a different complexity and costs about $N^2(2n^2m + nm^2)$ FLOPs. See Frison (2015) for a detailed overview of these and other condensing approaches.

After condensing, the condensed QP still needs to be solved, and the solution of the above unconstrained QP (8.50) is given by $\mathbf{u}^0 = -\mathcal{R}^{-1}(r + S\bar{x}_0)$. Because the Hessian $\mathcal{R}$ is a dense symmetric and usually positive definite matrix of size (Nm), it can be factorized using a Cholesky decomposition, which costs about $(1/3)N^3m^3$ FLOPs. Interestingly, the Cholesky factorization also could be computed simultaneously with the second condensing procedure mentioned above, which results in an additional cost of only about Nm^3 FLOPs (Frison, 2015), resulting in a condensing based Cholesky factorization of quadratic complexity in N, as discovered by Axehill and Morari (2012). The condensing approach can easily be extended to the case of additional constraints, and results in a condensed QP with Nm variables and some additional equality and inequality constraints that can be addressed by a dense QP solver.

Is condensing a sequential approach? Condensing is similar in spirit to a sequential approach that is applied to the LQ subproblem. To distinguish the different algorithmic ingredients, we reserve the term "sequential" for the nonlinear OCP only, while we speak of "condensing" when we refer to an LQ optimal control problem. This distinction is useful because all four combinations of sequential or simultaneous approaches with either the Riccati recursion or the condensing algorithm are possible, and lead to different algorithms. For example, when the simultaneous approach is combined with the condensing algorithm, it leads to different Newton-type iterates than the plain sequential approach, even though the linear algebra operations in the quadratic subproblems are similar.

Comparing Riccati recursion and condensing. The Riccati recursion, or, more generally, the banded-LDLT-factorization approaches, have a runtime that is linear in the horizon length N; they are therefore always preferable to condensing for long horizons. They can easily be combined with interior point methods and result in highly competitive QP solution algorithms. On the other hand, condensing-based QP solutions become more competitive than the Riccati approach for short to moderate horizon lengths N—in particular if the state dimension n is larger than the control dimension m, and if an efficient dense active set QP solver is used for the condensed QPs. Interestingly, one can combine the advantages of condensing and band structured linear algebra to yield a *partial condensing* method (Axehill, 2015), which is even more efficient than the plain Riccati approach on long horizons.

8.8.5 Sequential Approaches and Sparsity Exploitation

So far, we have only presented the solution of the unconstrained OCP by Newton-type methods in the *simultaneous approach*, to highlight the specific sparsity structure that is inherent in the resulting LQ problem. Many Newton-type algorithms also exist which are based on the *sequential approach*, however, where the Newton-type iterations are performed in the space of control sequences $\mathbf{u} = [u_0' \ \cdots \ u_{N-1}']'$ only. We recall that one eliminates the state trajectory by a nonlinear forward simulation in the sequential approach to maintain physically feasible trajectories. The plain sequential approach does not exploit sparsity and is not applicable to strongly unstable systems. Interestingly, some sequential approaches exist that do exploit the sparsity structure of the OCP and some—notably differential dynamic programming—even

incorporate feedback into the forward simulation to better deal with unstable dynamic systems.

Plain dense sequential approach. We start by describing how the plain sequential approach—the direct single-shooting method introduced in Section 8.5.1—solves the unconstrained OCP (8.45) with a Newton-type method. Here, all states are directly eliminated as a function of the controls by a forward simulation that starts at $x_0 := \bar{x}_0$ and recursively defines $x_{i+1} := f_i(x_i, u_i)$ for $i = 0, \ldots, N-1$. The result is that the objective function $F(\bar{x}_0, \mathbf{u}) := \sum_{i=0}^{N-1} \ell_i(x_i, u_i) + V_f(x_N)$ directly depends on all optimization variables $\mathbf{u} = [u_0' \cdots u_{N-1}']'$. The task of optimization now is to find a root of the nonlinear equation system $\nabla_{\mathbf{u}} F(\bar{x}_0, \mathbf{u}) = 0$. At some iterate $\bar{\mathbf{u}}$, after choosing a Hessian approximation $\bar{B} \approx \nabla^2_{\mathbf{u}} F(\bar{x}_0, \bar{\mathbf{u}})$, one has to solve linear systems of the form

$$\bar{B}(\mathbf{u} - \bar{\mathbf{u}}) = -\nabla_{\mathbf{u}} F(\bar{x}_0, \bar{\mathbf{u}}) \tag{8.56}$$

It is important to note that the exact Hessian $\nabla^2_{\mathbf{u}} F(\bar{x}_0, \bar{\mathbf{u}})$ is a dense matrix of size Nm (where m is the control dimension), and that one usually also chooses a dense Hessian approximation $\bar{B}$ that is ideally positive definite.

A Cholesky decomposition of a symmetric positive definite linear system of size Nm has a computational cost of $(1/3)(Nm)^3$ FLOPs, i.e., the iteration cost of the plain sequential approach grows cubically with the horizon length N. In addition to the cost of the linear system solve, one has to consider the cost of computing the gradient $\nabla_{\mathbf{u}} F(\bar{x}_0, \bar{\mathbf{u}})$. This is ideally done by a backward sweep equivalent to the reverse mode of algorithmic differentiation (AD) as stated in (8.16), at a cost that grows linearly in N. The cost of forming the Hessian approximation depends on the chosen approximation, but is typically quadratic in N. For example, an exact Hessian could be computed by performing Nm forward derivatives of the gradient function $\nabla_{\mathbf{u}} F(\bar{x}_0, \mathbf{u})$.

The plain dense sequential approach results in a medium-sized optimization problem without much sparsity structure but with expensive function and derivative evaluations, and can thus be addressed by a standard nonlinear programming method that does not exploit sparsity, but converges with a limited number of function evaluations. Typically, an SQP method in combination with a dense active set QP solver is used.

Sparsity-exploiting sequential approaches. Interestingly, one can form and solve the same linear system as in (8.56) by using the sparse

linear algebra techniques described in the previous section for the simultaneous approach. To implement this, it would be easiest to start with an algorithm for the simultaneous approach that computes the full iterate in the vector z that contains as subsequences the controls $\mathbf{u} = [u_0' \ \cdots \ u_{N-1}']'$, the states $\mathbf{x} = [x_0' \ \cdots \ x_N]'$, and the multipliers $\lambda = [\lambda_0' \ \cdots \ \lambda_N']'$. After the linear system solve, one would simply overwrite the states $\mathbf{x}$ by the result of a nonlinear forward simulation for the given controls $\mathbf{u}$.

The sparse sequential approach is particularly easy to implement if a Gauss-Newton Hessian approximation is used (Sideris and Bobrow, 2005). To compute the exact Hessian blocks, one performs a second reverse sweep identical to (8.16) to overwrite the values of the multipliers λ. As in the simultaneous approach, the cost for each Newton-type iteration would be linear in N with this approach, while one can show that the resulting iterates would be identical to those of the dense sequential approach for both the exact and the Gauss-Newton Hessian approximations.

8.8.6 Differential Dynamic Programming

The sequential approaches presented so far first compute the complete control trajectory $\mathbf{u}$ in each iteration, and then simulate the nonlinear system open loop with this trajectory $\mathbf{u}$ to obtain the states $\mathbf{x}$ for the next linearization point. In contrast, differential dynamic programming (DDP) (Mayne, 1966; Jacobson and Mayne, 1970) uses the time-varying affine feedback law $u_i^0(x_i) = k_i + K_i x_i$ from the Riccati recursion to simulate the nonlinear system forward in time. Like other sequential approaches, the DDP algorithm starts with an initial guess for the control trajectory—or the assumption of some feedback law—and the corresponding state trajectory. But then in each DDP iteration, starting at $x_0 := \bar{x}_0$, one recursively defines for $i = 0, 1, \ldots, N-1$

$$u_i := k_i + K_i x_i \tag{8.57a}$$

$$x_{i+1} := f_i(x_i, u_i) \tag{8.57b}$$

with K_i and k_i from (8.53a) and (8.53b), to define the next control and state trajectory. Interestingly, DDP only performs the backward recursions (8.51) and (8.52) from the Riccati algorithm. The forward simulation of the linear system (8.54b) is replaced by the forward simulation of the *nonlinear* system (8.57b). Note that both the states and the controls in DDP are different from the standard sequential approach.

DDP with Gauss-Newton Hessian. Depending on the type of Hessian approximation, different variants of DDP can be derived. Conceptually the easiest is DDP with a Gauss-Newton Hessian approximation, because it has no need of the multipliers λ_i. In case of a quadratic objective with positive semidefinite cost matrices, these matrices coincide with the Gauss-Newton Hessian blocks, and the method becomes particularly simple; one needs only to compute the system linearization matrices $\bar{A}_i, \bar{B}_i$ for $i = 0, \ldots, N-1$ at the trajectory $(\mathbf{x}, \mathbf{u})$ from the previous iteration to obtain all data for the LQ optimal control problem, and then perform the backward recursions (8.51) and (8.52) to define K_i and k_i in (8.53a) and (8.53b). This DDP variant is sometimes called iterative linear quadratic regulator (LQR) (Li and Todorov, 2004) and is popular in the field of robotics. Like any method based on the Gauss-Newton Hessian, the iterative LQR algorithm has the advantage that the Hessian approximation is always positive semidefinite, but the disadvantage that its convergence rate is only linear.

DDP with exact Hessian. In contrast to the iterative LQR algorithm, the DDP algorithm from Mayne (1966) uses an exact Hessian approximation and thus offers a quadratic rate of convergence. Like all exact Hessian methods, it can encounter indefiniteness of the Hessian, which can be addressed by algorithmic modifications that are beyond our interest here. To compute the exact Hessian blocks

$$\begin{bmatrix} \bar{Q}_i & \bar{S}_i' \\ \bar{S}_i & \bar{R}_i \end{bmatrix} := \nabla^2_{(x_i,u_i)} [\ell_i(\bar{x}_i, \bar{u}_i) + \bar{\lambda}_{i+1}' f_i(\bar{x}_i, \bar{u}_i)]$$

the DDP algorithm needs not only the controls $\bar{u}_i$, but also the states $\bar{x}_i$ and the Lagrange multipliers $\bar{\lambda}_{i+1}$, which are not part of the memory of the algorithm. While the states $\bar{x}_i$ are readily obtained by the nonlinear forward simulation (8.57b), the Lagrange multipliers $\bar{\lambda}_{i+1}$ are obtained simultaneously with the combined backward recursions (8.51) and (8.52). They are chosen as the gradient of the quadratic cost-to-go function $V_i^0(x_i) = p_i' x_i + \frac{1}{2} x_i' P_i x_i$ at the corresponding state values, i.e., as

$$\bar{\lambda}_i := p_i + P_i \bar{x}_i \tag{8.58}$$

for $i = N-1, \ldots, 0$. The last Hessian block (which is needed first in the backward recursion) is independent of the multipliers and just given by the second derivative of the terminal cost and defined by $\bar{P}_N := \nabla^2 V_f(\bar{x}_N)$. Because $\bar{p}_N := \nabla V_f(\bar{x}_N) - \bar{P}_N \bar{x}_N$, the last multiplier is given by $\bar{\lambda}_N := \nabla V_f(\bar{x}_N)$. Starting with these values for $\bar{P}_N, \bar{p}_N$, and $\bar{\lambda}_N$, the

backward Riccati recursions (8.51) and (8.52) can be started and the Lagrange multipliers be computed simultaneously using (8.58).

The DDP algorithm in its original form is only applicable to unconstrained problems, but can easily be adapted to deal with control constraints. In order to deal with state constraints, a variety of heuristics can be employed that include, for example, barrier methods; a similar idea was presented in the more general context of constrained OCPs under the name *feasibility perturbed sequential quadratic programming* by Tenny, Wright, and Rawlings (2004).

8.8.7 Additional Constraints in Optimal Control

Most Newton-type methods for optimal control can be generalized to problems with additional equality or inequality constraints. In nonlinear MPC, these additional constraints could be terminal equality constraints of the form $r(x_N) = 0$, as in the case of a zero terminal constraint; or terminal inequality constraints of the form $r(x_N) \leq 0$, as in the case of a terminal region. They could also be path constraints of the form $r_i(x_i, u_i) = 0$ or $r_i(x_i, u_i) \leq 0$ for $i = 0, \ldots, N-1$. The Lagrangian function then comprises additional contributions, but the block diagonal structure of the exact Hessian in (8.46) and the general sparsity of the problem is preserved.

Simultaneous approaches. If the multipliers for the extra constraints are denoted by μ_i for $i = 0, \ldots, N$, the Lagrangian in the simultaneous approaches is given by

$$\begin{aligned}\mathcal{L}(\bar{x}_0, w, \lambda, \mu) = \lambda_0'(\bar{x}_0 - x_0) + \mu_N' r_N(x_N) \; + \; V_f(x_N) \\ + \sum_{i=0}^{N-1} \ell_i(x_i, u_i) + \lambda_{i+1}'(f_i(x_i, u_i) - x_{i+1}) + \mu_i' r_i(x_i, u_i)\end{aligned}$$

We can summarize all primal-dual variables in a vector $z := [w' \; \lambda' \; \mu']'$ and write the Lagrangian as $\mathcal{L}(\bar{x}_0, z)$. In the purely equality-constrained case, Newton-type optimization algorithms again just try to find a root of the nonlinear equation system $\nabla_z \mathcal{L}(z) = 0$ by solving at a given iterate $\bar{z}$ the linear system $\bar{M}(z - \bar{z}) = -\nabla_z \mathcal{L}(\bar{z})$ where $\bar{M}$ is an approximation of the exact KKT matrix $\nabla_z^2 \mathcal{L}(\bar{z})$. In the presence of inequalities, one can resort to SQP or nonlinear IP methods. In all cases, the Lagrangian remains partially separable and the KKT matrix has a similar sparsity structure as for the unconstrained OCP. Therefore, the

linear algebra operations again can be performed by band-structure-exploiting algorithms that have a linear complexity in the horizon length N, if desired, or by condensing based approaches.

One major difference with unconstrained optimal control is that the overall feasibility of the optimization problem and the satisfaction of the linear independence constraint qualification (LICQ) condition is no longer guaranteed a priori, and thus, care needs to be taken in formulating well-posed constrained OCPs. For example, one immediately runs into LICQ violation problems if one adds a zero terminal constraint $x_N = 0$ to a problem with a large state dimension n, but a small control dimension m, and such a short time horizon N that the total number of control degrees of freedom, Nm, is smaller than n. In these unfortunate circumstances, the total number of equality constraints, $(N+1)n+n$, would exceed the total number of optimization variables, $(N+1)n+Nm$, making satisfaction of LICQ impossible.

Sequential approaches. Like the simultaneous approaches, most sequential approaches to optimal control—with the exception of DDP—can easily be generalized to the case of extra equality constraints, with some adaptations to the linear algebra computations in each iteration. For the treatment of inequality constraints on states and controls, one can again resort to SQP or nonlinear IP-based solution approaches. In the presence of state constraints, however, the iterates violate in general these state constraints; thus the iterates are infeasible points of the optimization problem, and the main appeal of the sequential approach is lost. On the other hand, the disadvantages of the sequential approach, i.e., the smaller region of convergence and slower contraction rate, especially for nonlinear and unstable systems, remain or become even more pronounced. For this reason, state constrained optimal control problems are most often addressed with simultaneous approaches.

8.9 Online Optimization Algorithms

Optimization algorithms for model predictive control need to solve not only one OCP, but a sequence of problems $\mathbb{P}_N(x_0)$ for a sequence of different values of x_0, and the time to work on each problem is limited by the sampling time Δt. Many different ideas can be used alone or in combination to ensure that the numerical approximation errors do not become too large and that the computation times remain bounded. In this section, we first discuss some general algorithmic considerations,

then present the important class of *continuation methods* and discuss in some detail the *real-time iteration*.

8.9.1 General Algorithmic Considerations

We next discuss one by one some general algorithmic ideas to adapt standard numerical optimal control methods to the context of online optimization for MPC.

Coarse discretization of control and state trajectories. The CPU time per Newton-type iteration strongly depends on the number of optimization variables in the nonlinear program (NLP), which itself depends on the horizon length N, the number of free control parameters, and on the state discretization method. To keep the size of the NLP small, one would classically choose a relatively small horizon length N, and employ a suitable terminal cost and constraint set that ensures recursive feasibility and nominal stability in case of exact NLP solutions. The total number of control parameters would then be Nm, and the state discretization would be equally accurate on all N control intervals.

In the presence of modeling errors and unavoidably inexact NLP solutions, however, one could also accept additional discretization errors by choosing a coarser control or state discretization, in particular in the end of the MPC horizon. Often, practitioners use *move blocking* where only the first $M \ll N$ control moves in the MPC horizon have the feedback sampling time Δt. The remaining $(N - M)$ control moves are combined into blocks of size two or larger, such that the overall number of control parameters is less than Nm. In particular if a plain dense single-shooting algorithm is employed, move blocking can significantly reduce the CPU cost per iteration. Likewise, one could argue that the state evolution need only be simulated accurately on the immediate future, while a coarser state discretization could be used toward the end of the horizon.

From the viewpoint of dynamic programming, one could argue that only the first control interval of duration Δt needs to be simulated accurately using the exact discrete time model $x_1 = f(x_0, u_0)$, while the remaining $(N - 1)$ intervals of the MPC horizon only serve the purpose of providing an approximation of the gradient of the cost-to-go function, i.e., of the gradient of $V_{N-1}(f(x_0, u_0))$. Since the discrete time dynamics usually originate from the approximation of a continuous time system, one could even decide to use a different state and control parameterization on the remaining time horizon. For example, after

the first interval of length Δt, one could use one single long collocation interval of length $(N-1)\Delta t$ with one global polynomial approximation of states and controls, as in pseudospectral collocation, in the hope of obtaining a cheaper approximation of $V_{N-1}(f(x_0, u_0))$.

Code generation and fixed matrix formats. Since MPC optimization problems differ only in the value x_0, many problem functions, and even some complete matrices in the Newton-type iterations, remain identical across different optimization problems and iterations. This allows for the code generation of optimization solvers that are tailored to the specific system model and MPC formulation. While the user interface can be in a convenient high-level language, the automatically generated code is typically in a standardized lower-level programming language such as plain C, which is supported by many embedded computing systems. The generated code has fixed matrix and vector dimensions, needs no online memory allocations, and contains no or very few switches. As an alternative to code generation, one could also just fix the matrix and vector dimensions in the most computationally intensive algorithmic components, and use a fixed specific matrix storage format that is optimized for the given computing hardware.

Delay compensations by prediction. Often, at a sampling instant t_0, one has a current state estimate x_0, but knows in advance that the MPC optimization calculations take some time, e.g., a full sampling time Δt. In the meantime, i.e., on the time interval $[t_0, t_0 + \Delta t]$, one usually has to apply some previously computed control action u_0. As all this is known before the optimization calculations start, one could first predict the expected state $x_1 := f(x_0, u_0)$ at the time $(t_0 + \Delta t)$ when the MPC computations are finished, and directly let the optimization algorithm address the problem $\mathbb{P}_N(x_1)$. Though this prediction approach cannot eliminate the feedback delay of one sampling time Δt in case of unforeseen disturbances, it can alleviate its effect in the case that model predictions and reality are close to each other.

Division into preparation and feedback phases. An additional idea is to divide the computations during each sampling interval into a long preparation phase, and a much shorter feedback phase that could, for example, consist of only a matrix vector multiplication in case of linear state feedback. We assume that the computations in the feedback phase take a computational time Δt_{fb} with $\Delta t_{\text{fb}} \ll \Delta t$, while the preparation time takes the remaining duration of one sampling interval. Thus, during the time interval $[t_0, t_0 + \Delta t - \Delta t_{\text{fb}}]$ one would perform

a preparation phase that presolves as much as possible the optimization problem that one expects at time $(t_0 + \Delta t)$, corresponding to a predicted state $\bar{x}_1$.

At time $(t_0 + \Delta t - \Delta t_{\text{fb}})$, when the preparation phase is finished, one uses the most current state estimate to predict the state at time $(t_0 + \Delta t)$ more accurately than before. Denote this new prediction x_1. During the short time interval $[t_0 + \Delta t - \Delta t_{\text{fb}}, t_0 + \Delta t]$, one performs the computations of the feedback phase to obtain an approximate feedback u_1 that is based on x_1. In case of linear state feedback, one would, for example, precompute a vector v and a matrix K in the preparation phase that are solely based on $\bar{x}_1$, and then evaluate $u_1 := v + K(x_1 - \bar{x}_1)$ in the feedback phase. Alternatively, more complex computations—such as the solution of a condensed QP—can be performed in the feedback phase. The introduction of the feedback phase reduces the delay to unforeseen disturbances from Δt to Δt_{fb}. One has to accept, however, that the feedback is not the exact MPC feedback, but only an approximation. Some online algorithms, such as the real-time iteration discussed in Section 8.9.2, achieve the division into preparation and feedback phase by reordering the computational steps of a standard optimization algorithm, without creating any additional overhead per iteration.

Tangential predictors. A particularly powerful way to obtain a cheap approximation of the exact MPC feedback is based on the tangential predictors from Theorem 8.16. In case of strict complementarity at the solution $\bar{w}$ of an expected problem $\mathbb{P}_N(\bar{x}_1)$, one can show that for sufficiently small distance $|x_1 - \bar{x}_1|$, the solution of the parametric QP(8.33) corresponds to a linear map, i.e., $w^{\text{QP}}(x_1) = \bar{w} + A(x_1 - \bar{x}_1)$. The matrix A can be precomputed based on knowledge of the exact KKT matrix at the solution $\bar{w}$, but before the state x_1 is known.

Generalized tangential predictors are based on the (approximate) solution of the full QP (8.33), which is more expensive than a matrix vector multiplication, but is also applicable to the cases where strict complementarity does not hold or where the active set changes. The aim of all tangential predictors is to achieve a second-order approximation that satisfies $\left|w^{\text{QP}}(x_1) - w^*(x_1)\right| = O(|x_1 - \bar{x}_1|^2)$, which is only possible if the exact KKT matrix is known. If the exact KKT matrix is not used in the underlying optimization algorithm, e.g., in case of a Gauss-Newton Hessian approximation, one can alternatively compute an *approximate generalized tangential predictor* $\tilde{w}^{\text{QP}}(x_1) \approx w^{\text{QP}}(x_1)$, which only approximates the exact tangential predictor, but can be obtained without creating additional overhead compared to a standard

optimization iteration.

Warmstarting and shift. Another easy way to transfer solution information from one MPC problem to the next is to use an existing solution approximation as initial guess for the next MPC optimization problem, in a procedure called *warmstarting*. In its simplest variant, one can just use the existing solution guess without any modification. In the *shift initialization*, one first shifts the current solution guess to account for the advancement of time. The shift initialization can most easily be performed if an equidistant grid is used for control and state discretization, and is particularly advantageous for systems with time-varying dynamics or objectives, e.g., if a sequence of future disturbances is known, or one is tracking a time-varying trajectory.

Iterating while the problem changes. Extending the idea of warmstarting, some MPC algorithms do not separate between one optimization problem and the next, but always iterate while the problem changes. They only perform one iteration per sampling time, and they never try to iterate the optimization procedure to convergence for any fixed problem. Instead, they continue to iterate while the optimization problem changes. When implemented with care, this approach ensures that the algorithm always works with the most current information, and never loses precious time by working on outdated information.

8.9.2 Continuation Methods and Real-Time Iterations

Several of the ideas mentioned above are related to the idea of *continuation methods*, which we now discuss in more algorithmic detail. For this aim, we first regard a parameter-dependent root-finding problem of the form

$$R(x, z) = 0$$

with variable $z \in \mathbb{R}^{n_z}$, parameter $x \in \mathbb{R}^n$, and a smooth function $R : \mathbb{R}^n \times \mathbb{R}^{n_z} \to \mathbb{R}^{n_z}$. This root-finding problem could originate from an equality constrained MPC optimization problem with fixed barrier as it arises in a nonlinear IP method. The parameter dependence on x is due to the initial state value, which varies from one MPC optimization problem to the next. In case of infinite computational resources, on could just employ one of the Newton-type methods from Section 8.3.2 to converge to an accurate approximation of the exact solution $z^*(x)$ that satisfies $R(x, z^*(x)) = 0$. In practice, however, we only have limited computing power and finite time, and need to be satisfied with an approximation of $z^*(x)$.

Fortunately, it is a realistic assumption that we have an approximate solution of a related problem available, for the previous value of x. To clarify notation, we introduce a problem index k, such that the aim of the continuation method is to solve root-finding problems $R(x_k, z) = 0$ for a sequence $(x_k)_{k \in \mathbb{I}}$. For algorithmic simplicity, we assume that the parameter x enters the function R linearly. This assumption means that the Jacobian of R with respect to z does not depend on x but only on z, and can thus be written as $R_z(z)$. As a consequence, also the linearization of R depends only on the linearization point $\bar{z}$, i.e., it can be written as $R_L(x, z; \bar{z}) := R(x, \bar{z}) + R_z(\bar{z})(z - \bar{z})$.

A simple full-step Newton iteration for a fixed parameter x would iterate $z^+ = \bar{z} - R_z(\bar{z})^{-1} R(x, \bar{z})$. If we have a sequence of values x_k, we could decide to perform only one Newton iteration for each value x_k and then proceed to the next one. Given a solution guess z_k for the parameter value x_k, a continuation method would then generate the next solution guess by the iteration formula

$$z_{k+1} := z_k - R_z(z_k)^{-1} R(x_{k+1}, z_k)$$

Another viewpoint on this iteration is that z_{k+1} solves the linear equation system $R_L(x_{k+1}, z_{k+1}; z_k) = 0$. Interestingly, assuming only regularity of R_z, one can show that if z_k equals the exact solution $z^*(x_k)$ for the previous parameter x_k, the next iterate z_{k+1} is a first-order approximation, or tangential predictor, for the exact solution $z^*(x_{k+1})$. More generally, one can show that

$$\left| z_{k+1} - z^*(x_{k+1}) \right| = O\left(\left| \begin{bmatrix} z_k - z^*(x_k) \\ x_{k+1} - x_k \end{bmatrix} \right|^2 \right) \tag{8.59}$$

From this equation it follows that one can remain in the area of convergence of the Newton method if one starts close enough to an exact solution, $z_k \approx z^*(x_k)$, and if the parameter changes $(x_{k+1} - x_k)$ are small enough. Interestingly, it also implies quadratic convergence toward the solution in case the parameter values of x_k remain constant. Roughly speaking, the continuation method delivers tangential predictors in case the parameters x_k change a lot, and nearly quadratic convergence in case they change little.

The continuation method idea can be extended to Newton-type iterations of the form

$$z_{k+1} := z_k - M_k^{-1} R(x_{k+1}, z_k)$$

with approximations $M_k \approx R_z(z_k)$. In this case, only approximate tangential predictors are obtained.

Real-time iterations. To generalize the continuation idea to a sequence of inequality constrained optimization problems $\mathbb{P}_N(x_k)$ of the general form (8.29) with primal-dual solutions $z^*(x_k)$, one performs SQP type iterations of the form (8.41), but use in each iteration a new parameter value x_{k+1}. This idea directly leads to the *real-time iteration* (Diehl, Bock, Schlöder, Findeisen, Nagy, and Allgöwer, 2002) that determines the approximate solution $z_{k+1} = (w_{k+1}, \lambda_{k+1}, \mu_{k+1})$ of problem $\mathbb{P}_N(x_{k+1})$ from the primal-dual solution of the following QP

$$\begin{array}{ll} \underset{w \in \mathbb{R}^{n_w}}{\text{minimize}} & F_{\mathrm{L}}(w; w_k) + \frac{1}{2}(w - w_k)' B_k (w - w_k) \\ \text{subject to} & G_{\mathrm{L}}(x_{k+1}, w; w_k) = 0 \\ & H_{\mathrm{L}}(w; w_k) \leq 0 \end{array} \tag{8.60}$$

which we denote by $\mathbb{P}_N^{\mathrm{QP}}(x_{k+1}; w_k, B_k)$. If one uses the exact Hessian, $B_k = \nabla_w^2 \mathcal{L}(z_k)$, Theorem 8.16 ensures that the QP solution is a generalized tangential predictor of the exact solution if z_k was equal to an exact and strongly regular solution $z^*(x_k)$. Conversely, if the values of x_k would remain constant, the exact Hessian SQP method would have quadratic convergence.

More generally, the exact Hessian real-time iteration satisfies the quadratic approximation formula (8.59), despite the fact that active set changes lead to nondifferentiability in the solution map $z^*(\cdot)$. Loosely speaking, the SQP based real-time iteration is able to easily "jump" across this nondifferentiability, and its prediction and convergence properties are not directly affected by active set changes. If the Hessian is not the exact one, the real-time iteration method delivers only approximate tangential predictors, and shows linear instead of quadratic convergence. In practice, one often uses the Gauss-Newton Hessian in conjunction with a simultaneous approach to optimal control, but also sequential approaches were suggested in a similar framework (Li and Biegler, 1989). One can generalize the SQP based real-time iteration idea further by allowing the subproblems to be more general convex optimization problems, and by approximating also the constraint Jacobians, as proposed and investigated by Tran-Dinh et al. (2012).

Shift initialization and shrinking horizon problems. If the parametric optimization problems originate from an MPC optimal control problem with time-varying dynamics or objectives, it can be beneficial to

employ a shift strategy that shifts every approximate solution by one time step backward in time before the next QP problem is solved. For notational correctness, we need to denote the MPC problem by $\mathbb{P}_N(k, x_k)$ in this case, to reflect the direct dependence on the time index k. While most of the variable shift is canonical, the addition of an extra control, state, and multiplier at the end of the prediction horizon is not trivial, and different variants exist. Some are based on an auxiliary control law and a forward simulation, but also a plain repetition of the second-to-last interval, which needs no additional computations, is a possibility.

The guiding idea of the shift initialization is that a shifted optimal solution should ideally correspond to an optimal solution of the new MPC problem, if the new initial value x_{k+1} originates from the nominal system dynamics $x_{k+1} = f(x_k, u_k)$. But while recursive feasibility can be obtained easily by a shift, recursive optimality can usually not be obtained for receding horizon problems. Thus, a shift strategy perturbs the contraction of the real-time iterations and needs to be applied with care. In the special case of time-invariant MPC problems $\mathbb{P}_N(x_k)$ with a short horizon and tight terminal constraint or cost, a shift strategy is not beneficial.

On the other hand, in the case of finite-time (batch) processes that are addressed by MPC on shrinking horizons, recursive optimality can easily be achieved by shrinking a previously optimal solution. More concretely, if the initial horizon length was N, and at time k one would have the solution to the problem $\mathbb{P}_{N-k}(k, x_k)$ on the remaining time horizon, the optimal solution to the problem $\mathbb{P}_{N-k-1}(k+1, x_{k+1})$ is easily obtained by dropping the first component of the controls, states, and multipliers. Thus, the shrinking operation is canonical and should be used if real-time iterations—or other continuation methods—are applied to shrinking horizon MPC problems.

8.10 Notes

The description of numerical optimal control methods in this chapter is far from complete, and we have left out many details as well as many methods that are important in practice. We mention some related literature and software links that could complement this chapter.

General numerical optimal control methods are described in the textbooks by Bryson and Ho (1975); Betts (2001); Gerdts (2011); and in particular by Biegler (2010). The latter reference focuses on di-

rect methods and also provides an in-depth treatment of nonlinear programming. The overview articles by Binder, Blank, Bock, Bulirsch, Dahmen, Diehl, Kronseder, Marquardt, Schlöder, and Stryk (2001); and Diehl, Ferreau, and Haverbeke (2009); as well as a forthcoming textbook on numerical optimal control (Gros and Diehl, 2018); have a similar focus on online optimization for MPC as the current chapter.

General textbooks on numerical optimization are Bertsekas (1999); Nocedal and Wright (2006). Convex optimization is covered by Ben-Tal and Nemirovski (2001); Nesterov (2004); Boyd and Vandenberghe (2004). The last book is particularly accessible for an engineering audience, and its PDF is freely available on the home page of its first author. Newton's method for nonlinear equations and its many variants are described and analyzed in a textbook by Deuflhard (2011). An up-to-date overview of optimization tools can be found at `plato.asu.edu/guide.html`, many optimization solvers are available as source code at `www.coin-or.org`, and many optimization solvers can be accessed online via `neos-server.org`.

While the direct single-shooting method often is implemented by coupling an efficient numerical integration solver with a general nonlinear program (NLP) solver such as SNOPT (Gill, Murray, and Saunders, 2005), the direct multiple-shooting and direct collocation methods need to be implemented by using NLP solvers that fully exploit the sparsity structure, such as IPOPT[3] (Wächter and Biegler, 2006) There exist many custom implementations of the direct multiple-shooting method with their own structure-exploiting NLP solvers, such as, for example, HQP[3] (Franke, 1998); MUSCOD-II (Leineweber, Bauer, Schäfer, Bock, and Schlöder, 2003); ACADO[3](Houska, Ferreau, and Diehl, 2011); and FORCES-NLP (Zanelli, Domahidi, Jerez, and Morari, 2017).

Structure-exploiting QP solvers that can be used standalone for linear MPC or as subproblem solvers within SQP methods are, for example, the dense code qpOASES[3] (Ferreau, Kirches, Potschka, Bock, and Diehl, 2014), which is usually combined with condensing, or the sparse codes FORCES (Domahidi, 2013); qpDUNES[3] (Frasch, Sager, and Diehl, 2015); and HPMPC[3] (Frison, 2015). The latter is based on a CPU specific matrix storage format that by itself leads to speedups in the range of one order of magnitude, and which was made available to the public in the BLASFEO[3] library at `github.com/giaf/blasfeo`.

In Section 8.2 on numerical simulation methods, we have exclusively treated Runge-Kutta methods because they play an important

[3]This code is available to the public under a permissive open-source license.

role within a large variety of numerical optimal control algorithms, such as shooting, collocation, or pseudospectral methods. Another popular and important family of integration methods, however, are the *linear multistep methods*; in particular, the implicit backward differentiation formula (BDF) methods are widely used for simulation and optimization of large stiff differential algebraic equations (DAEs). For an in-depth treatment of general numerical simulation methods for ordinary differential equations (ODEs) and DAEs, we recommend the textbooks by Hairer, Nørsett, and Wanner (1993, 1996); as well as Brenan, Campbell, and Petzold (1996); Ascher and Petzold (1998).

For derivative generation of numerical simulation methods, we refer to the research articles Bauer, Bock, Körkel, and Schlöder (2000); Petzold, Li, Cao, and Serban (2006); Kristensen, Jørgensen, Thomsen, and Jørgensen (2004); Quirynen, Gros, Houska, and Diehl (2017a); Quirynen, Houska, and Diehl (2017b); and the Ph.D. theses by Albersmeyer (2010); Quirynen (2017). A collection of numerical ODE and DAE solvers with efficient derivative computations are implemented in the SUNDIALS[3] suite (Hindmarsh, Brown, Grant, Lee, Serban, Shumaker, and Woodward, 2005).

Regarding Section 8.4 on derivatives, we refer to a textbook on algorithmic differentiation (AD) by Griewank and Walther (2008), and an overview of AD tools at `www.autodiff.org`. The AD framework CasADi[3] can in its latest form be found at `casadi.org`, and is described in the article Andersson, Akesson, and Diehl (2012); and the Ph.D. theses by Andersson (2013); Gillis (2015).

8.11 Exercises

Some of the exercises in this chapter were developed for courses on numerical optimal control at the University of Freiburg, Germany. The authors gratefully acknowledge Joel Andersson, Joris Gillis, Sébastien Gros, Dimitris Kouzoupis, Jesus Lago Garcia, Rien Quirynen, Andrea Zanelli, and Mario Zanon for contributions to the formulation of these exercises; as well as Michael Risbeck, Nishith Patel, Douglas Allan, and Travis Arnold for testing and writing solution scripts.

Exercise 8.1: Newton's method for root finding

In this exercise, we experiment with a full-step Newton method and explore the dependence of the iterates on the problem formulation and the initial guess.

(a) Write a computer program that performs Newton iterations in $\mathbb{R}^n$ that takes as inputs a function $F(z)$, its Jacobian $J(z)$, and a starting point $z_{[0]} \in \mathbb{R}^n$. It shall output the first 20 full-step Newton iterations. Test your program with $R(z) = z^{32} - 2$ starting first at $z_{[0]} = 1$ and then at different positive initial guesses. How many iterations do you typically need in order to obtain a solution that is exact up to machine precision?

(b) An equivalent problem to $z^{32} - 2 = 0$ can be obtained by *lifting* it to a higher dimensional space (Albersmeyer and Diehl, 2010), as follows

$$R(z) = \begin{bmatrix} z_2 - z_1^2 \\ z_3 - z_2^2 \\ z_4 - z_3^2 \\ z_5 - z_4^2 \\ 2 - z_5^2 \end{bmatrix}$$

Use your algorithm to implement Newton's method for this lifted problem and start it at $z_{[0]} = [1\ 1\ 1\ 1\ 1]'$ (note that we use square brackets in the index to denote the Newton iteration). Compare the convergence of the iterates for the lifted problem with those of the equivalent unlifted problem from the previous task, initialized at one.

(c) Consider now the root-finding problem $R(z) = 0$ with $R : \mathbb{R} \to \mathbb{R}, R(z) := \tanh(z) - \frac{1}{2}$. Convergence of Newton's method is sensitive to the chosen initial value z_0. Plot $R(z)$ and observe the nonlinearity. Implement Newton's method with full steps for it, and test if it converges or not for different initial values $z_{[0]}$.

(d) Regard the problem of finding a solution to the nonlinear equation system $2x = e^{y/4}$ and $16x^4 + 81y^4 = 4$ in the two variables $x, y \in \mathbb{R}$. Solve it with your implementation of Newton's method using different initial guesses. Does it always converge, and, if it converges, does it always converge to the same solution?

Exercise 8.2: Newton-type methods for a boundary-value problem

Regard the scalar discrete time system

$$x(k+1) = \frac{1}{10}\left(11x(k) + x(k)^2 + u\right), \quad k = 0, \ldots, N-1$$

with boundary conditions

$$x(0) = x_0 \qquad x(N) = 0$$

We fix the initial condition to $x_0 = 0.1$ and the horizon length to $N = 30$. The aim is to find the control value $u \in \mathbb{R}$—which is kept constant over the whole horizon—in order to steer the system to the origin at the final time, i.e., to satisfy the constraint $x(N) = 0$. This is a two-point boundary-value problem (BVP). In this exercise, we formulate this BVP as a root-finding problem in two different ways: first, with the sequential approach, i.e., with only the single control as decision variable; and second, with the simultaneous approach, i.e., with all 31 states plus the control as decision variables.

(a) Formulate and solve the problem with the sequential approach, and solve it with an exact Newton's method initialized at $u = 0$. Plot the state trajectories in each iteration. Also plot the residual values $x(N)$ and the variable u as a function of the Newton iteration index.

(b) Now formulate and solve the problem with the simultaneous approach, and solve it with an exact Newton's method initialized at $u = 0$ and the corresponding state sequence that is obtained by forward simulation started at x_0. Plot the state trajectories in each iteration.

Plot again the residual values $x(N)$ and the variable u as a function of the Newton iteration index, and compare with the results that you have obtained with the sequential approach. Do you observe differences in the convergence speed?

(c) One feature of the simultaneous approach is that its states can be initialized with any trajectory, even an infeasible one. Initialize the simultaneous approach with the all-zero trajectory, and again observe the trajectories and the convergence speed.

(d) Now solve both formulations with a Newton-type method that uses a constant Jacobian. For both approaches, the constant Jacobian corresponds to the exact Jacobian at the solution of the same problem for $x_0 = 0$, where all states and the control are zero. Start with implementing the sequential approach, and initialize the iterates at $u = 0$. Again, plot the residual values $x(N)$ and the variable u as a function of iteration index.

(e) Now implement the simultaneous approach with a fixed Jacobian approximation. Again, the Jacobian approximation corresponds to the exact Jacobian at the solution of the neighboring problem with $x_0 = 0$, i.e., the all zero trajectory. Start the Newton-type iterations with all states and the control set to zero, and plot the residual values $x(N)$ and the variable u as a function of iteration index. Discuss the differences of convergence speed with the sequential approach and with the exact Newton methods from before.

(f) The performance of the sequential approach can be improved if one introduces the initial state $x(0)$ as a second decision variable. This allows more freedom for the initialization, and one can automatically profit from tangential solution

predictors. Adapt your exact Newton method, initialize the problem in the all-zero solution and again observe the results.

(g) If u^* is the exact solution that is found at the end of the iterations, plot the logarithm of $|u - u^*|$ versus the iteration number for all six numerical experiments (a)–(f), and compare.

(h) The linear system that needs to be solved in each iteration of the simultaneous approach is large and sparse. We can use condensing in order to reduce the linear system to size one. Implement a condensing-based linear system solver that only uses multiplications and additions, and one division. Compare the iterations with the full-space linear algebra approach, and discuss the differences in the iterations, if any.

Exercise 8.3: Convex functions

Determine and explain whether the following functions are convex or not on their respective domains.

(a) $f(x) = c'x + x'A'Ax$ on $\mathbb{R}^n$

(b) $f(x) = -c'x - x'A'Ax$ on $\mathbb{R}^n$

(c) $f(x) = \log(c'x) + \exp(b'x)$ on $\{x \in \mathbb{R}^n \mid c'x > 0\}$

(d) $f(x) = -\log(c'x) - \exp(b'x)$ on $\{x \in \mathbb{R}^n \mid c'x > 0\}$

(e) $f(x) = 1/(x_1x_2)$ on $\mathbb{R}^2_{++}$

(f) $f(x) = x_1/x_2$ on $\mathbb{R}^2_{++}$

Exercise 8.4: Convex sets

Determine and explain whether the following sets are convex or not.

(a) A ball, i.e., a set of the form

$$\Omega = \{x \mid |x - x_c| \leq r\}$$

(b) A sublevel set of a convex function $f : \mathbb{R}^n \to \mathbb{R}$ for a constant $c \in \mathbb{R}$

$$\Omega = \{x \in \mathbb{R}^n \mid f(x) \leq c\}$$

(c) A superlevel set of a convex function $f : \mathbb{R}^n \to \mathbb{R}$ for a constant $c \in \mathbb{R}$

$$\Omega = \{x \in \mathbb{R}^n \mid f(x) \geq c\}$$

(d) The set

$$\Omega = \{x \in \mathbb{R}^n \mid x'B'Bx \leq b'x\}$$

(e) The set

$$\Omega = \{x \in \mathbb{R}^n \mid x'B'Bx \geq b'x\}$$

(f) A cone, i.e., a set of the form

$$\Omega = \{(x, \alpha) \in \mathbb{R}^n \times \mathbb{R} \mid |x| \leq \alpha\}$$

(g) A wedge, i.e., a set of the form

$$\{x \in \mathbb{R}^n \mid a_1'x \le b_1, a_2'x \le b_2\}$$

(h) A polyhedron

$$\{x \in \mathbb{R}^n \mid Ax \le b\}$$

(i) The set of points closer to one set than another

$$\Omega = \{x \in \mathbb{R}^n \mid \operatorname{dist}(x, S) \le \operatorname{dist}(x, \mathcal{T})\}$$

where $\operatorname{dist}(x, S) := \inf\{|x - z|_2 \mid z \in S\}$.

Exercise 8.5: Finite differences: theory of optimal perturbation size

Assume we have a twice continuously differentiable function $f : \mathbb{R} \to \mathbb{R}$ and we want to evaluate its derivative $f'(x_0)$ at x_0 with finite differences. Further assume that for all $x \in [x_0 - \delta, x_0 + \delta]$ holds that

$$|f(x)| \le f_{\max} \qquad |f''(x)| \le f''_{\max} \qquad |f'''(x)| \le f'''_{\max}$$

We assume $\delta > t$ for any perturbation size t in the following finite difference approximations. Due to finite machine precision ϵ_{mach} that leads to truncation errors, the computed function $\tilde{f}(x) = f(x)(1 + \epsilon(x))$ is perturbed by noise $\epsilon(x)$ that satisfies the bound

$$|\epsilon(x)| \le \epsilon_{\text{mach}}$$

(a) Compute a bound on the error of the forward difference approximation

$$\tilde{f}'_{\text{fd},t}(x_0) := \frac{\tilde{f}(x_0 + t) - \tilde{f}(x_0)}{t}$$

namely, a function $\psi(t; f_{\max}, f''_{\max}, \epsilon_{\text{mach}})$ that satisfies

$$\left|\tilde{f}'_{\text{fd},t}(x_0) - f'(x_0)\right| \le \psi(t; f_{\max}, f''_{\max}, \epsilon_{\text{mach}})$$

(b) Which value t^* minimizes this bound and which value ψ^* has the bound at t^*?

(c) Perform a similar error analysis for the central difference quotient

$$\tilde{f}'_{\text{cd},t}(x_0) := \frac{\tilde{f}(x_0 + t) - \tilde{f}(x_0 - t)}{2t}$$

that is, compute a bound

$$\left|\tilde{f}'_{\text{fd},t}(x_0) - f'(x_0)\right| \le \psi_{\text{cd}}(t; f_{\max}, f''_{\max}, f'''_{\max}, \epsilon_{\text{mach}})$$

(d) For central differences, what is the optimal perturbation size t^*_{cd} and what is the size ψ^*_{cd} of the resulting bound on the error?

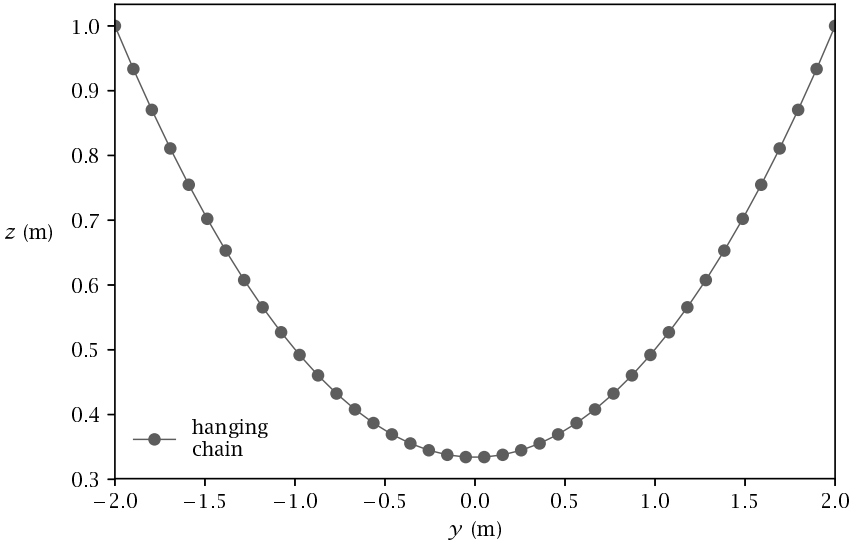


Figure 8.7: A hanging chain at rest. See Exercise 8.6(b).

Exercise 8.6: Finding the equilibrium point of a hanging chain using CasADi

Consider an elastic chain attached to two supports and hanging in-between. Let us discretize it with N mass points connected by $N-1$ springs. Each mass i has position (y_i, z_i), $i = 1, \ldots, N$.

Our task is to minimize the total potential energy, which is made up by potential energy in each string and potential energy of each mass according to

$$J(y_1, z_1, \ldots, y_n, z_n) = \underbrace{\frac{1}{2}\sum_{i=1}^{N-1} D_i\left((y_i - y_{i+1})^2 + (z_i - z_{i+1})^2\right)}_{\text{spring potential energy}} + \underbrace{g_0 \sum_{i=1}^{N} m_i z_i}_{\text{gravitational potential energy}} \tag{8.61}$$

subject to constraints modeling the ground.

(a) CasADi is an open-source software tool for solving optimization problems in general and optimal control problems (OCPs) in particular. In its most typical usage, it is used to formulate and solve constrained optimization problems of the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & \underline{x} \le x \le \overline{x} \\ & \underline{g} \le g(x) \le \overline{g} \end{aligned} \tag{8.62}$$

where $x \in \mathbb{R}^{n_x}$ is the decision variable, $f : \mathbb{R}^{n_x} \to \mathbb{R}$ is the objective function, and $g : \mathbb{R}^{n_x} \to \mathbb{R}^{n_g}$ is the constraint function. For equality constraints, the

upper and lower bounds are equal.

If you have not already done so, go to `casadi.org` and locate the installation instructions. On most platforms, installing CasADi amounts to downloading a binary installation and placing it somewhere in your path. Version 3.3 of CasADi on Octave/MATLAB was used in this edition, so make sure that you are not using a version older than this and keep an eye on the text website for incompatibilities with future versions of CasADi. Locate the CasADi user guide and, with an Octave or MATLAB interpreter in front of you, read Chapters 1 through 4. These chapters give you an overview of the scope and syntax of CasADi.

(b) We assume that f is a convex quadratic function and g is a linear function. In this case we refer to (8.62) as a convex quadratic program (QP). To solve a QP with CasADi, we construct symbolic expressions for x, f, and g, and use this to construct a solver object that can be called one or more times with different values for $\overline{x}$, $\underline{x}$, $\overline{g}$, and $\underline{g}$. An initial guess for x can also be provided, but this is less important for convex QPs, where the solution is unique.

Figure 8.7 shows the solution of the unconstrained problem using the open-source QP solver qpOASES with $N = 40$, $m_i = 40/N$ kg, $D_i = 70N$ N/m, and $g_0 = 9.81$ m/s^2. The first and last mass points are fixed to $(-2, 1)$ and $(2, 1)$, respectively. Go through the code for the figure and make sure you understand the steps.

(c) Now introduce ground constraints: $z_i \geq 0.5$ and $z_i \geq 0.5 + 0.1\,y_i$, for $i = 2, \cdots, N-2$. Resolve the QP and compare with the unconstrained solution.

(d) We now want to formulate and solve a nonlinear program (NLP). Since an NLP is a generalization of a QP, we can solve the above problem with an NLP solver. This can be done by simply changing `casadi.qpsol` in the script to `casadi.nlpsol` and the solver plugin `'qpoases'` with `'ipopt'`, corresponding to the open-source NLP solver IPOPT. Are the solutions of the NLP and QP solver the same?

(e) Now, replace the linear equalities by nonlinear ones that are given by $z_i \geq 0.5 + 0.1\,y_i^2$ for $i = 2, \cdots, N-2$. Modify the expressions from before to formulate and solve the NLP, and visualize the solution. Is the NLP convex?

(f) Now, by modifications of the expressions from before, formulate and solve an NLP where the inequality constraints are replaced by $z_i \geq 0.8 + 0.05\,y_i - 0.1\,y_i^2$ for $i = 2, \cdots, N-2$. Is this NLP convex?

Exercise 8.7: Direct single shooting versus direct multiple shooting

Consider the following OCP, corresponding to driving a Van der Pol oscillator to the origin, on a time horizon with length $T = 10$

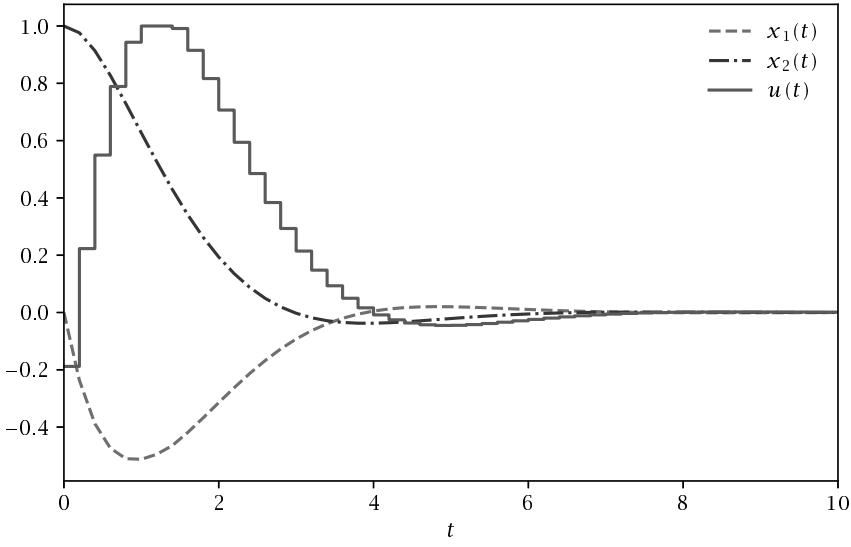


Figure 8.8: Direct single shooting solution for (8.63) without path constraints.

$$
\begin{aligned}
\underset{x(\cdot),u(\cdot)}{\text{minimize}} \quad & \int_0^T (x_1(t)^2 + x_2(t)^2 + u(t)^2)\, dt \\
\text{subject to} \quad & \dot{x}_1(t) = (1 - x_2(t)^2)\, x_1(t) - x_2(t) + u(t) \\
& \dot{x}_2(t) = x_1(t) \\
& -1 \le u(t) \le 1, \quad t \in [0, T] \\
& x_1(0) = 0, \quad x_1(T) = 0 \\
& x_2(0) = 1, \quad x_2(T) = 0 \\
& -0.25 \le x_1(t), \quad t \in [0, T]
\end{aligned}
\tag{8.63}
$$

We will solve this problem using direct single shooting and direct multiple shooting using IPOPT as the NLP solver.

(a) Figure 8.8 shows the solution to the above problem using a direct single shooting approach, without enforcing the constraint $-0.25 \le x_1(t)$. Go through the code for the figure step by step. The code begins with a modeling step, where symbolic expressions for the continuous-time model are constructed. Thereafter, the problem is transformed into discrete time by formulating an object that integrates the system forward in time using a single step of the RK4 method. This function also calculates the contribution to the objective function for the same interval using the same integrator method. In the next part of the code, a

symbolic representation of the NLP is constructed, starting with empty lists of variables and constraints. This symbolic representation of the NLP is used to define an NLP solver object using IPOPT as the underlying solver. Finally, the solver object is evaluated to obtain the optimal solution.

(b) Modify the code so that the path constraint on $x_1(t)$ is being respected. You only need to enforce this constraint at the end of each control interval. This should result in additional components to the NLP constraint function $G(w)$, which will now have upper and lower bounds similar to the decision variable w. Resolve the modified problem and compare the solution.

(c) Modify the code to implement the direct multiple-shooting method instead of direct single shooting. This means introducing decision variables corresponding to not only the control trajectory, but also the state trajectory. The added decision variables will be matched with an equal number of new equality constraints, enforcing that the NLP solution corresponds to a continuous state trajectory. The initial and terminal conditions on the state can be formulated as upper and lower bounds on the corresponding elements of w. Use $x(t) = 0$ as the initial guess for the state trajectory.

(d) Compare the IPOPT output for both transcriptions. How did the change from direct single shooting to direct multiple shooting influence

- The number of iterations?
- The number of nonzeros in the Jacobian of the constraints?
- The number of nonzeros in the Hessian of the Lagrangian?

(e) Generalize the RK4 method so that it takes $M = 4$ steps instead of just one. This corresponds to a higher-accuracy integration of the model dynamics. Approximately how much smaller discretization error can we expect from this change?

(f) Replace the RK4 integrator with the variable-order, variable-step size code CVODES from the SUNDIALS suite, available as the `'cvodes'` plugin for `casadi.integrator`. Use 10^{-8} for the relative and absolute tolerances. Consult CasADi's user guide for syntax. What are the advantages and disadvantages of using this integrator over the fixed-step RK4 method used until now?

Exercise 8.8: Direct collocation

Collocation, in its most basic sense, refers to a way of solving initial-value problems by approximating the state trajectory with piecewise polynomials. For each step of the integrator, corresponding to an interval of time, we choose the coefficients of these polynomials to ensure that the ODE becomes exactly satisfied at a given set of time points. In the following, we choose the *Gauss-Legendre* collocation integrator of sixth order, which has $d = 3$ collocation points. Together with the point 0 at the start of the interval $[0, 1]$, we have four time points to define the collocation polynomial

$$\tau_0 = 0 \qquad \tau_1 = 1/2 - \sqrt{15}/10 \qquad \tau_2 = 1/2 \qquad \tau_3 = 1/2 + \sqrt{15}/10$$

Using these time points, we define the corresponding Lagrange polynomials

$$L_j(\tau) = \prod_{r=0, r\neq j}^{d} \frac{\tau - \tau_r}{\tau_j - \tau_r}$$

Introducing a uniform time grid $t_k = k\,h$, $k = 0, \ldots, N$, with the corresponding state values $x_k := x(t_k)$, we can approximate the state trajectory inside each interval $[t_k, t_{k+1}]$ as a linear combination of these basis functions

$$\tilde{x}_k(t) = \sum_{r=0}^{d} L_r\left(\frac{t - t_k}{h}\right) x_{k,r}$$

By differentiation, we get an approximation of the time derivative at each collocation point for $j = 1, \ldots, 3$

$$\dot{\tilde{x}}_k(t_{k,j}) = \frac{1}{h} \sum_{r=0}^{d} \dot{L}_r(\tau_j)\, x_{k,r} := \frac{1}{h} \sum_{r=0}^{d} C_{r,j}\, x_{k,r}$$

We also can get an expression for the state at the end of the interval

$$\tilde{x}_{k+1,0} = \sum_{r=0}^{d} L_r(1)\, x_{k,r} := \sum_{r=0}^{d} D_r\, x_{k,r}$$

Finally, we also can integrate our approximation over the interval, giving a formula for *quadratures*

$$\int_{t_k}^{t_{k+1}} \tilde{x}_k(t)\, dt = h \sum_{r=0}^{d} \int_0^1 L_r(\tau)\, d\tau\, x_{k,r} := h \sum_{r=1}^{d} b_r\, x_{k,r}$$

(a) Figure 8.9 shows an open-loop simulation for the ODE in (8.63) using Gauss-Legendre collocation of order 2, 4, and 6. A constant control $u(t) = 0.5$ was applied and the initial conditions were given by $x(0) = [0, 1]'$. The figure on the left shows the first state $x_1(t)$ for the three methods as well as a high-accuracy solution obtained from CVODES, which uses a backward differentiation formula (BDF) method. In the figure on the right we see the discretization error, as compared with CVODES. Go through the code for the figure and make sure you understand it. Using this script as a template, replace the integrator in the direct multiple-shooting method from Exercise 8.7 with this collocation integrator. Make sure that you obtain the same solution. The structure of the NLP should remain unchanged—you are still implementing the direct multiple-shooting approach, only with a different integrator method.

(b) In the NLP transcription step, replace the embedded function call with additional degrees of freedom corresponding to the state at all the collocation points. Enforce the collocation equations at the NLP level instead of the integrator level.

Enforce upper and lower bounds on the state at all collocation points. Compare the solution time and number of nonzeros in the Jacobian and Hessian matrices with the direct multiple-shooting method.

Exercise 8.9: Gauss-Newton SQP iterations for optimal control

Consider a nonlinear pendulum defined by

$$\dot{x}(t) = f(x(t), u(t)) = \begin{bmatrix} v(t) \\ -C \sin(p(t)/C) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t)$$

with state $x = [p, v]'$ and $C := 180/\pi/10$, to solve an OCP using a direct multiple-shooting method and a self-written sequential quadratic programming (SQP) solver with a Gauss-Newton Hessian.

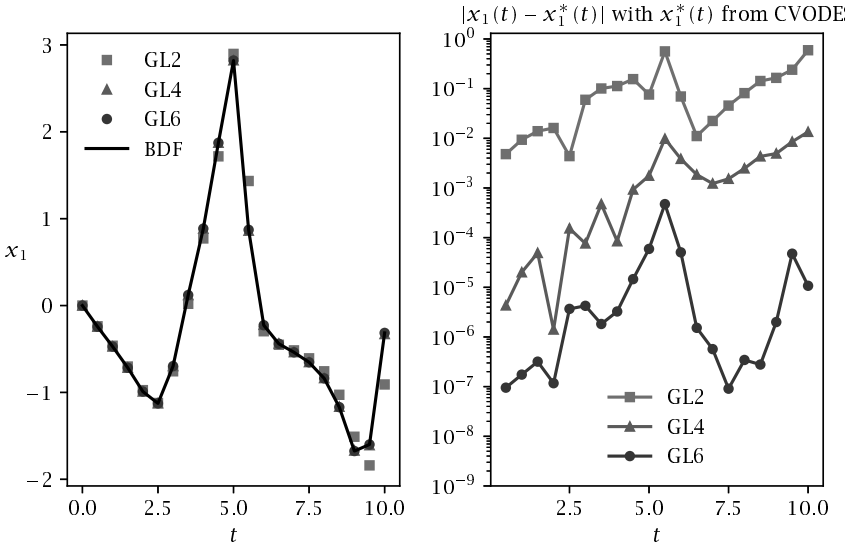


Figure 8.9: Open-loop simulation for (8.63) using collocation.

(a) Starting with the pendulum at $\bar{x}_0 = [10\ 0]'$, we aim to minimize the required controls to bring the pendulum to $x_N = [0\ 0]'$ in a time horizon $T = 10\,\text{s}$. Regarding bounds on p, v, and u, namely $p_{\max} = 10$, $v_{\max} = 10$, and $u_{\max} = 3$, the required controls can be obtained as the solution of the following OCP

$$
\begin{aligned}
\underset{\substack{x_0,u_0,x_1,\ldots,\\ u_{N-1},x_N}}{\text{minimize}} \quad & \frac{1}{2}\sum_{k=0}^{N-1} |u_k|_2^2 \\
\text{subject to} \quad & \bar{x}_0 - x_0 = 0 \\
& \Phi(x_k, u_k) - x_{k+1} = 0, \quad k = 0,\ldots,N-1 \\
& x_N = 0 \\
& -x_{\max} \le x_k \le x_{\max}, \quad k = 0,\ldots,N-1 \\
& -u_{\max} \le u_k \le u_{\max}, \quad k = 0,\ldots,N-1
\end{aligned}
$$

Formulate the discrete dynamics $x_{k+1} = \Phi(x_k, u_k)$ using a RK4 integrator with a time step $\Delta t = 0.2\,\text{s}$. Encapsulate the code in a single CasADi function of the form of a CasADi function object as in Exercise 8.7. Simulate the system forward in time and plot the result.

(b) Using $w = (x_0, u_0, \ldots, u_{N-1}, x_N)$ as the NLP decision variable, we can formulate the equality constraint function $G(w)$, the least squares function $M(w)$, and the

bounds vector $w_{\max}$ so that the above OCP can be written

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & \frac{1}{2}|M(w)|_2^2 \\ \text{subject to} \quad & G(w) = 0 \\ & -w_{\max} \le w \le w_{\max} \end{aligned}$$

The SQP method with Gauss-Newton Hessian solves a linearized version of this problem in each iteration. More specifically, if the current iterate is $\bar{w}$, the next iterate is given by $\bar{w} + \Delta w$, where Δw is the solution of the following QP

$$\begin{aligned} \underset{\Delta w}{\text{minimize}} \quad & \frac{1}{2}\Delta w' J_M(\bar{w})' J_M(\bar{w})\Delta w + M(\bar{w})' J_M(\bar{w})\Delta w \\ \text{subject to} \quad & G(\bar{w}) + J_G(\bar{w})\Delta w = 0 \\ & -w_{\max} - \bar{w} \le \Delta w \le w_{\max} - \bar{w} \end{aligned} \tag{8.64}$$

In order to implement the Gauss-Newton method, we need the Jacobians $J_G(w) = \frac{\partial G}{\partial w}(w)$ and $J_M(w) = \frac{\partial M}{\partial w}(w)$, both of which can be efficiently obtained using CasADi's `jacobian` command. In this case the Gauss-Newton Hessian $H = J_M(\bar{w})' J_M(\bar{w})$ can readily be obtained by pen and paper. Define what H_x and H_u need to be in the Hessian

$$H = \begin{bmatrix} H_x & & & \\ & H_u & & \\ & & \ddots & \\ & & & H_x \end{bmatrix} \qquad H_x = \begin{bmatrix} & \end{bmatrix} \qquad H_u = \begin{bmatrix} & \end{bmatrix}$$

(c) Figure 8.10 shows the control trajectory after 0, 1, 2, and 6 iterations of the Gauss-Newton method applied to a direct multiple-shooting transcription of (8.63). Go through the code for the figure step by step. You should recognize much of the code from the solution to Exercise 8.7. The code represents a simplified, yet efficient way of using CasADi to solve OCPs.

Modify the code to solve the pendulum problem. Note that the sparsity patterns of the linear and quadratic terms of the QP are printed out at the beginning of the execution. $J_G(w)$ is a block sparse matrix with blocks being either identity matrices I or partial derivatives $A_k = \frac{\partial \Phi}{\partial x}(x_k, u_k)$ and $B_k = \frac{\partial \Phi}{\partial u}(x_k, u_k)$.

Initialize the Gauss-Newton procedure at $w = 0$, and stop the iterations when $|w_{k+1} - w_k|$ gets smaller than 10^{-4}. Plot the iterates as well as the vector G during the iterations. How many iterations do you need?

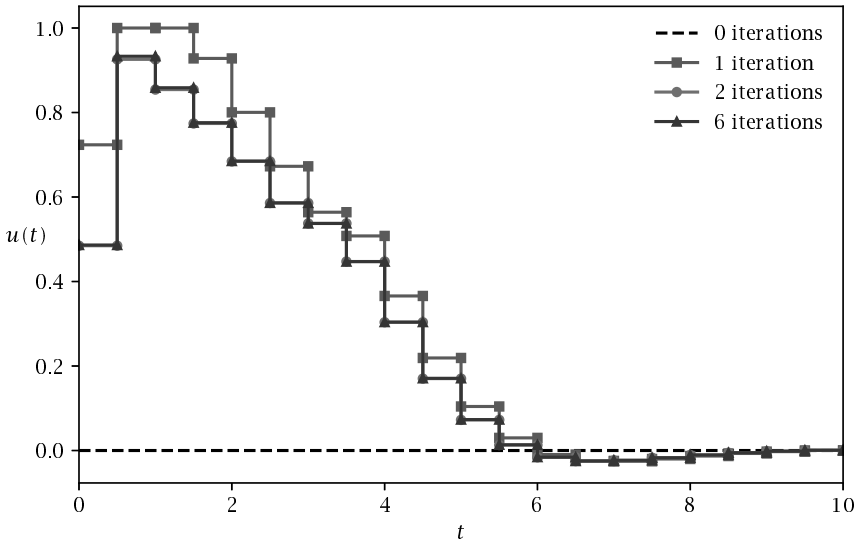


Figure 8.10: Gauss-Newton iterations for a direct multiple-shooting transcription of (8.63); $u(t)$ after 0, 1, 2, and 6 Gauss-Newton iterations.

Exercise 8.10: Real-time iterations and nonlinear MPC

We return to the OCP from Exercise 8.9

$$
\begin{aligned}
\underset{\substack{x_0,u_0,x_1,\ldots,\\ u_{N-1},x_N}}{\text{minimize}} \quad & \frac{1}{2}\sum_{k=0}^{N-1} |\, u_k \,|_2^2 \\
\text{subject to} \quad & \bar{x}_0 - x_0 = 0 \\
& \Phi(x_k, u_k) - x_{k+1} = 0, \qquad k = 0, \ldots, N-1 \\
& x_N = 0 \\
& -x_{\max} \leq x_k \leq x_{\max}, \quad k = 0, \ldots, N-1 \\
& -u_{\max} \leq u_k \leq u_{\max}, \quad k = 0, \ldots, N-1
\end{aligned}
$$

In this problem, we regard $\bar{x}_0$ as a parameter and modify the simultaneous Gauss-Newton algorithm from Exercise 8.9. In particular, we modify this algorithm to perform real-time iterations for different values of $\bar{x}_0$, so that we can use the algorithm to perform closed-loop nonlinear MPC simulations for stabilization of the nonlinear pendulum.

(a) Modify the function `sqpstep` from the solution of Exercise 8.9 so that it accepts the parameter $\bar{x}_0$. You would need to update the upper and lower bounds on w accordingly. Test it and make sure that it works.

(b) In order to visualize the generalized tangential predictor, call the `sqpstep` method with different values for $\bar{x}_0$ while resetting the variable vector $\bar{w}$ to its initial value (zero) between each call. Use a linear interpolation for $\bar{x}_0$ with 100 points between zero and the value $(10, 0)'$, i.e., set $\bar{x}_0 = \lambda[10\ 0]'$ for $\lambda \in [0, 1]$. Plot the first control u_0 as a function of λ and keep your plot.

(c) To compute the exact solution manifold with relatively high accuracy, perform now the same procedure for the same 100 increasing values of λ, but this time perform for each value of λ multiple Gauss-Newton iterations, i.e., replace each call to `sqpstep` with, e.g., 10 calls without changing $\bar{x}_0$. Plot the obtained values for u_0 and compare with the tangential predictor from the previous task by plotting them in the same plot.

(d) In order to see how the real-time iterations work in a more realistic setting, let the values of λ jump faster from 0 to 1, e.g., by doing only 10 steps, and plot the result again into the same plot.

(e) Modify the previous algorithm as follows: after each change of λ by 0.1, keep it constant for nine iterations, before you do the next jump. This results in about 100 consecutive real-time iterations. Interpret what you see.

(f) Now we do the first *closed-loop simulation*: set the value of $\bar{x}_0^{[1]}$ to $[10\ 0]'$ and initialize $w^{[0]}$ at zero, and perform the first real-time iteration by calling `sqpstep`. This iteration yields the new solution guess $w^{[1]}$ and corresponding control $u_0^{[1]}$. Use this control at the "real plant," i.e., generate the next value of $\bar{x}_0$, which we denote $\bar{x}_0^{[2]}$, by calling the one-step simulation function, $\bar{x}_0^{[2]} := \Phi(\bar{x}_0^{[1]}, u_0^{[1]})$. Close the loop by calling `sqpstep` using $w^{[1]}$ and $\bar{x}_0^{[2]}$, etc., and perform 100 iterations. For better observation, plot after each real-time iteration the control and state variables on the whole prediction horizon. (It is interesting to note that the state trajectory is not necessarily feasible).

Also observe what happens with the states $\bar{x}_0$ during the scenario, and plot them in another plot against the time index. Do they converge, and if yes, to what value?

(g) Now we make the control problem more difficult by treating the pendulum in an upright position, which is unstable. This is simply done by changing the sign in front of the sine in the differential equation, i.e., our model is now

$$f(x(t), u(t)) = \begin{bmatrix} v(t) \\ C\sin(p(t)/C) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t) \tag{8.65}$$

Start your real-time iterations again at $w^{[0]} = 0$ and set $\bar{x}_0^{[1]}$ to $[10\ 0]'$, and perform the same closed-loop simulation as before. Explain what happens.

Bibliography

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

J. Albersmeyer and M. Diehl. The lifted Newton method and its application in optimization. *SIAM Journal on Optimization*, 20(3):1655–1684, 2010.

J. Andersson. *A General-Purpose Software Framework for Dynamic Optimization*. PhD thesis, KU Leuven, October 2013.

J. Andersson, J. Akesson, and M. Diehl. CasADi – a symbolic package for automatic differentiation and optimal control. In *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 297–307. Springer, 2012.

U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, Philadelphia, 1998.

D. Axehill. Controlling the level of sparsity in MPC. *Systems & Control Letters*, 76:1–7, 2015.

D. Axehill and M. Morari. An alternative use of the Riccati recursion for efficient optimization. *Systems & Control Letters*, 61(1):37–40, 2012.

I. Bauer, H. G. Bock, S. Körkel, and J. P. Schlöder. Numerical methods for optimum experimental design in DAE systems. *J. Comput. Appl. Math.*, 120 (1-2):1–15, 2000.

A. Ben-Tal and A. Nemirovski. *Lectures on Modern Convex Optimization: Analysis, Algorithms, and Engineering Applications*. MPS-SIAM Series on Optimization. MPS-SIAM, Philadelphia, 2001.

D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, second edition, 1999.

J. T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming*. SIAM, Philadelphia, 2001.

L. T. Biegler. *Nonlinear Programming*. MOS-SIAM Series on Optimization. SIAM, 2010.

T. Binder, L. Blank, H. G. Bock, R. Bulirsch, W. Dahmen, M. Diehl, T. Kronseder, W. Marquardt, J. P. Schlöder, and O. V. Stryk. Introduction to model based optimization of chemical processes on moving horizons. *Online Optimization of Large Scale Systems: State of the Art, Springer*, pages 295–340, 2001.

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.

H. G. Bock. Recent advances in parameter identification techniques for ODE. In *Numerical Treatment of Inverse Problems in Differential and Integral Equations*, pages 95–121. Birkhäuser, 1983.

H. G. Bock and K. J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. In *Proceedings of the IFAC World Congress*, pages 242–247. Pergamon Press, 1984.

S. P. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical solution of initial-value problems in differential-algebraic equations*. Classics in Applied Mathematics 14. SIAM, Philadelphia, 1996.

A. E. Bryson and Y. Ho. *Applied Optimal Control*. Hemisphere Publishing, New York, 1975.

A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 13:117–119, 1974.

P. Deuflhard. *Newton methods for nonlinear problems: affine invariance and adaptive algorithms*, volume 35. Springer, 2011.

M. Diehl. *Real-Time Optimization for Large Scale Nonlinear Processes*. PhD thesis, Universität Heidelberg, 2001.

M. Diehl, H. G. Bock, J. P. Schlöder, R. Findeisen, Z. Nagy, and F. Allgöwer. Real-time optimization and nonlinear model predictive control of processes governed by differential-algebraic equations. *J. Proc. Cont.*, 12(4):577–585, 2002.

M. Diehl, H. J. Ferreau, and N. Haverbeke. Efficient numerical methods for nonlinear MPC and moving horizon estimation. In L. Magni, M. D. Raimondo, and F. Allgöwer, editors, *Nonlinear model predictive control*, volume 384 of *Lecture Notes in Control and Information Sciences*, pages 391–417. Springer, 2009.

A. Domahidi. *Methods and Tools for Embedded Optimization and Control*. PhD thesis, ETH Zürich, 2013.

H. J. Ferreau, C. Kirches, A. Potschka, H. G. Bock, and M. Diehl. qpOASES: a parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, 6(4):327–363, 2014.

R. Franke. *Integrierte dynamische Modellierung und Optimierung von Systemen mit saisonaler Wärmespeicherung*. PhD thesis, Technische Universität Ilmenau, Germany, 1998.

J. V. Frasch, S. Sager, and M. Diehl. A parallel quadratic programming method for dynamic optimization problems. *Mathematical Programming Computations*, 7(3):289–329, 2015.

G. Frison. *Algorithms and Methods for High-Performance Model Predictive Control*. PhD thesis, Technical University of Denmark (DTU), 2015.

A. H. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47:629–705, 2005.

M. Gerdts. *Optimal Control of ODEs and DAEs*. Berlin, Boston: De Gruyter, 2011.

P. Gill, W. Murray, and M. Saunders. SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization. *SIAM Review*, 47(1):99–131, 2005.

J. Gillis. *Practical methods for approximate robust periodic optimal control of nonlinear mechanical systems*. PhD thesis, KU Leuven, 2015.

A. Griewank and A. Walther. *Evaluating Derivatives*. SIAM, 2 edition, 2008.

S. Gros and M. Diehl. *Numerical Optimal Control*. 2018. (In preparation).

J. Guddat, F. G. Vasquez, and H. T. Jongen. *Parametric Optimization: Singularities, Pathfollowing and Jumps*. Teubner, Stuttgart, 1990.

E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I*. Springer Series in Computational Mathematics. Springer, Berlin, 2nd edition, 1993.

E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations II – Stiff and Differential-Algebraic Problems*. Springer Series in Computational Mathematics. Springer, Berlin, 2nd edition, 1996.

A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM Trans. Math. Soft.*, 31:363–396, 2005.

B. Houska, H. J. Ferreau, and M. Diehl. ACADO toolkit – an open source framework for automatic control and dynamic optimization. *Optimal Cont. Appl. Meth.*, 32(3):298–312, 2011.

D. H. Jacobson and D. Q. Mayne. *Differential dynamic programming*, volume 24 of *Modern Analytic and Computational Methods in Science and Mathematics*. American Elsevier Pub. Co., 1970.

M. R. Kristensen, J. B. Jørgensen, P. G. Thomsen, and S. B. Jørgensen. An ESDIRK method with sensitivity analysis capabilities. *Computers and Chemical Engineering*, 28:2695–2707, 2004.

D. B. Leineweber, I. Bauer, A. A. S. Schäfer, H. G. Bock, and J. P. Schlöder. An Efficient Multiple Shooting Based Reduced SQP Strategy for Large-Scale Dynamic Process Optimization (Parts I and II). *Comput. Chem. Eng.*, 27: 157–174, 2003.

W. Li and E. Todorov. Iterative linear quadratic regulator design for nonlinear biological movement systems. In *Proceedings of the 1st International Conference on Informatics in Control, Automation and Robotics*, 2004.

W. C. Li and L. T. Biegler. Multistep, Newton-Type Control Strategies for Constrained Nonlinear Processes. *Chem. Eng. Res. Des.*, 67:562–577, 1989.

D. Q. Mayne. A second-order gradient method for determining optimal trajectories of non-linear discrete-time systems. *Int. J. Control*, 3(1):85–96, 1966.

Y. Nesterov. *Introductory lectures on convex optimization: a basic course*, volume 87 of *Applied Optimization*. Kluwer Academic Publishers, 2004.

J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, second edition, 2006.

T. Ohtsuka. A continuation/GMRES method for fast computation of nonlinear receding horizon control. *Automatica*, 40(4):563–574, 2004.

G. Pannocchia, J. B. Rawlings, D. Q. Mayne, and G. Mancuso. Whither discrete time model predictive control? *IEEE Trans. Auto. Cont.*, 60(1):246–252, January 2015.

L. R. Petzold, S. Li, Y. Cao, and R. Serban. Sensitivity analysis of differential-algebraic equations and partial differential equations. *Computers and Chemical Engineering*, 30:1553–1559, 2006.

R. Quirynen. *Numerical Simulation Methods for Embedded Optimization*. PhD thesis, KU Leuven and University of Freiburg, 2017.

R. Quirynen, S. Gros, B. Houska, and M. Diehl. Lifted collocation integrators for direct optimal control in ACADO toolkit. *Mathematical Programming Computation*, pages 1–45, 2017a.

R. Quirynen, B. Houska, and M. Diehl. Efficient symmetric Hessian propagation for direct optimal control. *Journal of Process Control*, 50:19–28, 2017b.

S. M. Robinson. Strongly Regular Generalized Equations. *Mathematics of Operations Research, Vol. 5, No. 1 (Feb., 1980), pp. 43-62*, 5:43–62, 1980.

A. Sideris and J. Bobrow. An efficient sequential linear quadratic algorithm for solving unconstrained nonlinear optimal control problems. *IEEE Transactions on Automatic Control*, 50(12):2043–2047, 2005.

M. J. Tenny, S. J. Wright, and J. B. Rawlings. Nonlinear Model Predictive Control via Feasibility-Perturbed Sequential Quadratic Programming. *Comp. Optim. Appl.*, 28:87–121, 2004.

Q. Tran-Dinh, C. Savorgnan, and M. Diehl. Adjoint-based predictor-corrector sequential convex programming for parametric nonlinear optimization. *SIAM J. Optimization*, 22(4):1258–1284, 2012.

C. F. Van Loan. Computing integrals involving the matrix exponential. *IEEE Trans. Automat. Control*, 23(3):395–404, 1978.

A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Prog.*, 106(1):25–57, 2006.

A. Zanelli, A. Domahidi, J. Jerez, and M. Morari. FORCES NLP: An efficient implementation of interior-point methods for multistage nonlinear nonconvex programs. *International Journal of Control*, 2017.