Numerical Optimal Control

$\boxed{\text{DRAFT}}$

Moritz Diehl
Optimization in Engineering Center (OPTEC)
and ESAT-SCD, K.U. Leuven,
Kasteelpark Arenberg 10,
3001 Leuven, Belgium
`moritz.diehl@esat.kuleuven.be`

June 1, 2011

Foreword

Optimal control regards the *optimization* of *dynamic systems*. Thus, it bridges two large and active research communities of applied mathematics, each with their own journals and conferences. A scholar of numerical optimal control has to acquire basic numerical knowledge within both fields, i.e. numerical optimization on the one hand, and system theory and numerical simulation on the other hand. Within this text, we start by rehearsing basic concepts from both fields. Hereby, we give numerical optimization the larger weight, as dynamic system simulation is often covered rather well in engineering and applied mathematics curricula, and basic optimization concepts such as convexity or optimality conditions and Lagrange multipliers play a crucial role in numerical methods for optimal control. The course is intended for students of engineering and the exact sciences as well as for interested PhD students and besides the abovementioned fields requires only knowledge of linear algebra and numerical analysis. The course should be accompanied by computer exercises, and its aim is to give an introduction into numerical methods for solution of optimal control problems, in order to prepare the students for using and developing these methods themselves for specific applications in science and engineering.

The course is divided into four major parts [for each of which we also give a few references for further reading].

- Introduction and Background [63, 22]
- Discrete Time Optimal Control [10]
- Continuous Time Optimal Control [25, 13]
- Real-Time Optimization [30]

This script was written while the author was visiting ETH Zurich in spring 2011 to give a one semester master course on numerical optimal control.

Moritz Diehl,
Leuven,
June 2011.

moritz.diehl@esat.kuleuven.be

Contents

Part I

Introduction and Background

Chapter 1

Dynamic Systems and Their Simulation

Optimal control regards the optimization of dynamic systems, i.e. processes that are evolving with time and that are characterized by a *state* x that allows us to predict the future system behaviour. Often, the dynamic system can be controlled by suitable choice of some inputs that we denote as *controls* u in this course.

Dynamic systems and their mathematical models can come in many variants:

- dynamic systems might be *autonomous*, i.e. only evolving according to their intrinsic laws of motion and independent of time, or *non-autonomous*, i.e. influenced by the outside world or depending on time. When controls are involved we have per definition a non-autonomous system, which in this course we usually call a *controlled dynamic system.* Note that its dynamics can still be time-independent or time-dependent depending on the presence of uncontrollable influences from the outside world.
- dynamic systems might be *stochastic*, like the movements of assets on the stockmarket, or *deterministic*, like the motion of planets in the solar system.
- dynamic systems might also appear as a game where the evolution is neither deterministic nor stochastic but determined by one or multiple adverse players, which is treated in the fields of robust optimal control and game theory.
- the state and control spaces in which x and u live might be *discrete*, like in a chess game or on a digital computer, or *continuous*, like the motion of a car.
- finally, the time in which the system evolves might be *discrete*, like in an automaton or any other iterative process, or *continuous*, like the physical time.

In this course, we have a strong focus on deterministic systems with continuous state and control spaces, and regard both discrete time as well as continuous time systems, in two major parts of the course. The only exception to this is found in the chapter on dynamic programming (DP) where we derive and illustrate the method also for discrete state and control spaces and show that DP can be applied not only to deterministic problems but also to stochastic systems and games. The main reason for the restriction to optimal control of deterministic systems with continuous state and control spaces is that these optimal control problems can be treated by derivative-based optimization methods and are thus tremendously easier to solve than most other classes, both in terms of the solvable system sizes as well as the computational speed. Most crucial, these continuous optimal control problems comprise the important class of convex optimal control problems, which usually allow us to find a global solution reliably and fast. Convex optimal control problems are important in their own right, but also serve as an approximation of nonconvex optimal control problems within Newton-type optimization methods.

1.1 Controlled Discrete Time Systems

In the first large part of the course we will regard discrete time dynamical systems on a time horizon of length N that are characterized by the time-dependent dynamics

$$.x_{k+1} \quad = \quad f(x_k, u_k), \quad k = 0, 1, \ldots, N-1. \tag{1.1}$$

with u_k the "controls" or "inputs" and x_k the "states". Let $x_k \in \mathbb{R}^{n_x}$ and let $u_k \in \mathbb{R}^{n_u}$ with $k = 0, \ldots, N-1$.

If we know the initial state x_0 and the controls $u_0, \ldots, u_{N-1}$ we could recursively call the function f in order to obtain all other states, $x_1, \ldots, x_N$. We call this a *forward simulation* of the system dynamics.

Definition 1.1 (Forward Simulation)
We define the so called "forward simulation routine" to be the computer program that generates a map

$$\begin{array}{rrcl} f_{\text{sim}}: & \mathbb{R}^{n_x+Nn_u} & \rightarrow & \mathbb{R}^{Nn_x} \\ & (x_0; u_0, u_1, \ldots, u_{N-1}) & \mapsto & (x_1, x_2, \ldots, x_N) \end{array} \tag{1.2}$$

that is recursively defined by the Equation (1.1).

The inputs of the forward simulation routine are both the initial value and the controls. In many practical problems we can only choose the controls while the initial value is fixed. Though this is a very natural assumption, it is not the only possible one. In optimization, we might have very different requirements: We might, for example, have a free initial value that we want to choose in an optimal way. Or we might have both a fixed initial state and a fixed terminal state that we want to reach. We might also look for periodic sequences with $x_0 = x_N$, but do not know x_0 beforehand. All these desires on the initial and the terminal state can be expressed by suitable constraints. For the purpose of this course it is important to note that the only constraints that are characterizing an optimal control problem are the system dynamics stated in Equation (1.1), but no initial value constraint, which is optional in optimal control.

1.1.1 Linear Time Invariant Systems

A special class of tremendous importance are the linear time invariant (LTI) systems

$$x_{k+1} \quad = \quad Ax_k + Bu_k, \quad k = 0, 1, \dots, N-1. \tag{1.3}$$

with matrices $A \in \mathbb{R}^{n_x \times n_x}$ and $B \in \mathbb{R}^{n_x \times n_u}$. LTI systems are one of the principal interests in the field of automatic control and a vast literature exists on LTI systems. Many important notions such as controllability or stabilizability, and computational results such as the step or frequency response functions can be defined in terms of the matrices A and B alone. Note that in the field of linear system analysis and control, often output equations $y_k = Cx_k + Du_k$ are added, and the outputs y_k are regarded the only physically relevant output quantities while the state x_k has more of a virtual existence and different state space realizations of the same input-output behaviour exist. In contrast to this, in this course, we assume that the state is a fundamental entity within the optimal control problem and outputs are for simplicity not explicitly added as they can be subsumed in the optimal control problem formulation, if necessary.

1.1.2 Affine Systems and Linearizations along Trajectories

An important generalization of linear systems are affine time-varying systems of the form

$$x_{k+1} \quad = \quad A_k x_k + B_k u_k + c_k, \quad k = 0, 1, \dots, N-1, \tag{1.4}$$

These often appear as linearizations of nonlinear dynamic systems along a given reference trajectory. To see this latter point, let us regard a nonlinear dynamic system and some given reference trajectory values $\bar{x}_0, \dots, \bar{x}_{N-1}$ as well as $\bar{u}_0, \dots, \bar{u}_{N-1}$. Then the Taylor expansion of each function f_k at the reference value $(\bar{x}_k, \bar{u}_k)$ is given by

$$(x_{k+1} - \bar{x}_{k+1}) \quad \approx \quad \frac{\partial f_k(\bar{x}_k, \bar{u}_k)}{\partial x}(x_k - \bar{x}_k) + \frac{\partial f_k(\bar{x}_k, \bar{u}_k)}{\partial u}(u_k - \bar{u}_k) + (f_k(\bar{x}_k, \bar{u}_k) - \bar{x}_{k+1}) \tag{1.5}$$

thus resulting in affine time-varying dynamics of the form (1.4) for the differences $(x_k - \bar{x}_k)$ and $(u_k - \bar{u}_k)$. Note that even for a time-invariant nonlinear system the linearized dynamics becomes time-variant due to the different linearization points on the reference trajectory.

1.2 Controlled Continuous Time Dynamic Systems

Most systems of interest in science and engineering are described in form of differential equations which live in continuous time, while all numerical simulation methods have to discretize the time interval of interest and thus effectively generate discrete time systems of different forms. This section only scratches the surface of numerical simulation methods, and in the remainder of this course numerical solvers for solution of initial value problems will be treated as black boxes. In this section, only a very brief introduction into the most relevant concepts from simulation of differential equations is given. Good introductory books on numerical simulation of dynamic systems described by ordinary differential (ODE) or differential algebraic equations (DAE) are e.g. [23, 24, 3].

1.2.1 Ordinary Differential Equations

A controlled dynamic system in continuous time can be described by an ordinary differential equation (ODE) on a horizon $[0, T]$ with controls $u(t)$ by

$$\dot{x}(t) = f(x(t), u(t), t), \quad t \in [0, T] \tag{1.6}$$

where t is the continuous time and $x(t)$ the state. However, the dependence of f on the fixed controls $u(t)$ is for the purpose of this section equivalent to a direct time-dependence of f, if we redefine the ODE as $\dot{x} = \tilde{f}(x, t)$ with $\tilde{f}(x, t) := f(x, u(t), t)$. Thus, from now on we will leave away the dependence on the controls for notational simplicity, and just regard the time-dependent ODE:

$$\dot{x}(t) = f(x(t), t), \quad t \in [0, T] \tag{1.7}$$

An initial value problem (IVP) is given by (1.7) and the initial value constraint $x(0) = x_0$ with some fixed parameter x_0.

Existence of a solution to an IVP is guaranteed under continuity of f w.r.t. to x and t according to a theorem that is due to G. Peano (1858-1932). But existence alone is of limited interest as the solutions might be non-unique. For example, the scalar ODE $\dot{x}(t) = \sqrt{|x(t)|}$ can stay for an undetermined duration in the point $x = 0$ before leaving it at an arbitrary time t_0 and then following a trajectory $x(t) = (t - t_0)^2/4$. This ODE is continuous at the origin, but its slope approaches infinity, which causes the non-uniqueness.

More important is thus the following theorem by Picard (1890) and Lindelöf (1894):

Theorem 1.1 (Existence and Uniqueness of IVP)**:** *Regard the initial value problem* (1.7) *with* $x(0) = x_0$*, and assume that* f *is continuous with respect to* x *and* t*. Furthermore, assume that* f *is Lipschitz continuous w.r.t.* x*, i.e. that there exists a constant* L *such that for all* x, y *and all* $t \in [0, T]$

$$\|f(x, t) - f(y, t)\| \leq L\|x - y\|.$$

Then there exists a unique solution $x(t)$ *of the IVP in a neighbourhood of* $(x_0, 0)$.

Note that this theorem can be extended to the case that there are finitely many discontinuities of f w.r.t. t, in which case the solutions are still unique, but the ODE solution has to be defined in the weak sense. The fact that unique solutions still exist in the case of discontinuities is important because (a) many optimal control problems have discontinuous control trajectories $u(t)$ in their solution, and (b) many algorithms, the so called *direct methods*, first discretize the controls, typically as piecewise constant functions which have jumps at the interval boundaries. This does not cause difficulties for existence and uniqueness of the IVPs.

1.2.2 Numerical Integration: Explicit One-Step Methods

Numerical integration methods are used to approximately solve a well-posed IVP that satisfies the conditions of Theorem 1.1. They come in many different variants, and can be categorized along two major categories, on the one hand the one-step vs. the multistep methods, on the other hand the explicit vs. the implicit methods. Let us start in this section with the explicit one-step methods which basically approximate the continuous dynamic system by a discrete time dynamic system.

All numerical integration methods start by discretizing w.r.t. time t. Let us for simplicity assume a fixed stepsize $\Delta t = T/N$, with N an integer. We then divide the interval $[0, T]$ into N subintervals $[t_k, t_{k+1}]$, $k = 0, \dots N-1$, each of length Δt, i.e. we have $t_k := k\Delta t$. Then, the solution is approximated on the grid points t_k by values s_k that shall satisfy $s_k \approx x(t_k)$, $k = 0, \dots, N-1$, where $x(t)$ is the exact solution to the IVP.

Numerical integration methods differ in the ways how they approximate the solution on the grid points and between, but they all shall have the property that if $N \to \infty$ then $s_k \to x(t_k)$. This is called *convergence*. Methods differ in the speed of convergence, and one says that a method is convergent with order p if

$$\max_{k=0,\dots,N} \|s_k - x(t_k)\| = O(\Delta t^p).$$

The simplest integrator is the explicit Euler method. It first sets $s_0 := x_0$ and then recursively computes, for $k = 0, \dots, N-1$:

$$s_{k+1} := s_k + \Delta t\, f(s_k, t_k). \tag{1.8}$$

It is a first order method, and due to this low order it is very inefficient and should not be used in practice.

However, with a few extra evaluations of f in each step, higher order one-step methods can easily be obtained, the *explicit Runge-Kutta (RK) methods* due to Runge (1895) and Kutta (1901). On each discretization interval $[t_k, t_{k+1}]$ these use not only one but m evaluations of f at the intermediate states $s_k^{(i)}$, $i = 1, \dots, m$, that live on a grid of intermediate time points $t_k^{(i)} := t_k + c_i\,\Delta t$ with suitably chosen c_i that satisfy $0 \leq c_i \leq 1$. Then one RK step is obtained by performing the following intermediate steps:

$$\begin{aligned}
s_k^{(1)} &:= s_k && (1.9)\\
s_k^{(2)} &:= s_k + \Delta t\, a_{21} f(s_k^{(1)}, t_k^{(1)}) && (1.10)\\
s_k^{(3)} &:= s_k + \Delta t \left(a_{31} f(s_k^{(1)}, t_k^{(1)}) + a_{32} f(s_k^{(2)}, t_k^{(2)})\right) && (1.11)\\
&\vdots && (1.12)\\
s_k^{(i)} &:= s_k + \Delta t \sum_{j=1}^{i-1} a_{ij} f(s_k^{(j)}, t_k^{(j)}) && (1.13)\\
&\vdots && (1.14)\\
s_k^{(m)} &:= s_k + \Delta t \sum_{j=1}^{m-1} a_{mj} f(s_k^{(j)}, t_k^{(j)}) && (1.15)\\
s_{k+1} &:= s_k + \Delta t \sum_{j=1}^{m} b_j f(s_k^{(j)}, t_k^{(j)}) && (1.16)
\end{aligned}$$

Each RK method is characterized by its so-called Butcher tableau,

$$\begin{array}{c|cccc}
c_1 & & & & \\
c_2 & a_{21} & & & \\
c_3 & a_{31} & a_{32} & & \\
\vdots & \ddots & \ddots & & \\
c_m & a_{m1} & \cdots & a_{m,m-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_m
\end{array}$$

and by a smart choice of these coefficients a high order of the method can be achieved. The explicit Euler integrator has the choice $m = 1$ $c_1 = 0$, $b_1 = 1$, and a widespread method of order $m = 4$ has the tableau

$$\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{1}{2} & 0 & \frac{1}{2} & & \\
1 & 0 & 0 & 1 & \\
\hline
 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array} \qquad (1.17)$$

Note that practical RK methods also have stepsize control, i.e. adapt Δt depending on estimates of the local error, which are obtained by comparing two RK steps of different order. Particularly efficient methods of this adaptive type, the *Runge-Kutta-Fehlberg* methods, reuse as many evaluations of f as possible between the two RK steps.

Why is the Euler method not used in practice, and a high order so important? To get an intuitive idea let us assume that we want to simulate an ODE on the interval $[0, 1]$ with a very low accuracy of $\epsilon = 10^{-3}$ and that a first order method gives an accuracy $\epsilon = 10\Delta t$. Then it needs $\Delta t = 10^{-4}$, i.e. 10000 steps. If a fourth order method gives the accuracy $\epsilon = 10(\Delta t)^4$, it needs $\Delta t = 0.1$, i.e. only 10 steps for the same accuracy! Given this enourmous difference, the four times higher cost per RK step for the higher order method is more than outweighed, and it is still 250 times faster than the first order Euler method! In practice, RK integrators with orders up to 8 are used, but the Runge-Kutta-Fehlberg method of fourth order (+fifth order for error estimation) is the most popular one.

1.2.3 Stiff Systems and Implicit Integrators

When an explicit integrator is applied to a very stable system, its steps very easily overshoot. For example, when we want to simulate for a very large $\lambda \gg 1$ the ODE

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

it is clear that this system is super stable and converges very quickly to zero. If we now use an explicit Euler method with stepsize Δt, then the discrete time dynamic system

$$s_{k+1} = s_k - \Delta t\, \lambda s_k = (1 - \Delta t\, \lambda) s_k$$

is obtained. This system gets unstable if $\Delta t > \frac{2}{\lambda}$, which might be very small when λ is very big. Note that such a small stepsize is not necessary to obtain a high accuracy, but is only necessary to render the integrator stable. It turns out that all explicit methods suffer from the fact that super stable systems necessitate excessively short step sizes. This becomes particularly annoying if a system has both slow and fast decaying modes, i.e. if some of the eigenvalues of the Jacobian $\frac{\partial f}{\partial x}$ are only slightly negative while others are strongly negative, i.e. represent very stable dynamics. Such systems are called stiff systems.

Instead of using explicit integrators, stiff systems can be much better treated by implicit integrators. The simplest of them is the implicit Euler integrator, which in each integrator step solves the nonlinear equation in the variable s_{k+1}

$$s_{k+1} = s_k + \Delta t\, f(s_{k+1}, t_{k+1}). \tag{1.18}$$

If applied to the super stable test system from above, for which this equation can explicitly be solved, the implicit Euler yields the dynamics

$$s_{k+1} = s_k - \Delta t\, \lambda s_{k+1} \Leftrightarrow s_{k+1} = s_k / (1 + \Delta t\, \lambda)$$

which is stable for any $\Delta t > 0$ and will always converge to zero, like the true solution of the ODE. The implicit Euler is stable for this example. It turns out that the idea can easily be generalized to RK methods which then just obtain Butcher tableaus which are full square and not only lower triangular. An implicit RK method has to solve in each iteration the nonlinear equation system

$$s_k^{(1)} = s_k + \Delta t \sum_{j=1}^{m} a_{1j} f(s_k^{(j)}, t_k^{(j)}) \tag{1.19}$$

$$\vdots \tag{1.20}$$

$$s_k^{(i)} = s_k + \Delta t \sum_{j=1}^{m} a_{ij} f(s_k^{(j)}, t_k^{(j)}) \tag{1.21}$$

$$\vdots \tag{1.22}$$

$$s_k^{(m)} = s_k + \Delta t \sum_{j=1}^{m} a_{mj} f(s_k^{(j)}, t_k^{(j)}) \tag{1.23}$$

and then sets the next step to

$$s_{k+1} := s_k + \Delta t \sum_{j=1}^{m} b_j f(s_k^{(j)}, t_k^{(j)})$$

The nonlinear system needs typically to be solved by a Newton method. Note that the system is of size $(m \times n_x)$.

1.2.4 Orthogonal Collocation

A specific variant of implicit RK methods is called *collocation* and is derived as follows: the solution $x(t)$ on the collocation interval $t \in [t_k, t_{k+1}]$ is approximated by a polynomial of mth order, $p(t; v) = \sum_{j=0}^{m} v_j t^j$, and a grid of *collocation points* $t_k^{(i)} = t_k + c_i \Delta t$ is chosen as above, using $0 \leq c_1 < c_2 \ldots < c_m \leq 1$. The $(m+1)$ unknown vector coefficients $v_0, \ldots, v_m$ are then determined via the $(m+1)$ vector equations that require that the polynomial starts at s_k and its derivative at each collocation point matches the function f at the corresponding value of the polynomial.

$$s_k = p(t_k; v) = \sum_{j=0}^{m} v_j t_k^j \tag{1.24a}$$

$$f(p(t_k^{(1)}; v), t_k^{(1)}) = p'(t_k^{(1)}; v) = \sum_{j=1}^{m} j\, v_j (t_k^{(1)})^{j-1} \tag{1.24b}$$

$$\vdots \tag{1.24c}$$

$$f(p(t_k^{(m)}; v), t_k^{(m)}) = p'(t_k^{(m)}; v) = \sum_{j=1}^{m} j\, v_j (t_k^{(m)})^{j-1} \tag{1.24d}$$

$$\tag{1.24e}$$

Finally, once all v_i are known, the full step is given by evaluation of the polynomial at the end point of the interval:

$$s_{k+1} = \sum_{j=0}^{m} v_j (t_{k+1})^j.$$

It can be easily shown that the above procedure can be formulated as an implicit RK method with a Butcher tableau that is specific to the choice of the collocation points. Very important is the observation that a smart choice of collocation points leads to very high orders, using the principle of Gauss-quadrature. This is achieved by first noticing that the exact trajectory $x(t)$ satisfies the equation

$$x(t_{k+1}) = x(t_k) + \int_{t_k}^{t_{k+1}} f(x(t), t)\,\mathrm{d}t.$$

In collocation, the trajectory $x(t)$ is approximated by the polynomial $p(t; v)$ for which holds $p'(t_k^{(i)}; v) = f(p(t_k^{(i)}; v), t_k^{(i)})$ at the collocation points $t_k^{(i)}$. Using this fact, we obtain for the polynomial the identity

$$p(t_{k+1}; v) = p(t_k; v) + \Delta t \cdot \sum_{i=1}^{m} \omega_i f(p(t_k^{(i)}; v), t_k^{(i)})$$

with the quadrature weights ω_i corresponding to the choice of collocation points. If we use Gauss-quadrature for this integral, i.e. choose the collocation points $t_k^{(i)}$ as the zeros of orthogonal Legendre polynomials on the corresponding interval $[t_k, t_{k+1}]$, then this integration is exact for polynomials up to degree $2m - 1$, implying that the collocation step $s_{k+1} - s_k$ would be exact if the exact solution would have a derivative $\dot{x}(t)$ that is a polynomial of order $2m - 1$, i.e. if the solution $x(t)$ would be a polynomial of order $2m$. This is called *Gauss-Legendre collocation*. It is the collocation method with the highest possible order, $2m$. Note that all its collocation points are in the interior of the collocation interval and symmetric around the midpoint, see the table below. Another popular collocation method sacrifices one order and chooses a set of collocation points that includes the end point of the interval. It is called *Gauss-Radau collocation* and has a desirable property for stiff systems called *stiff decay*. The relative collocation point locations $\xi_i = (t_k^{(i)} - t_k)/(t_{k+1} - t_k)$ for Gauss-Legendre and Gauss-Radau collocation are given in the table below, citing from [13].

m	Gauss-Legendre collocation	Gauss-Radau collocation
1	0.500000	1.000000
2	0.211325 0.788675	0.333333 1.000000
3	0.112702 0.500000 0.887298	0.155051 0.644949 1.000000
4	0.069432 0.330009 0.669991 0.930568	0.088588 0.409467 0.787659 1.000000

1.2.5 Linear Multistep Methods and Backward Differentiation Formulae

A different approach to obtain a high order are the linear multistep methods that use a linear combination of the past M steps $s_{k-M+1}, \ldots, s_k$ and their function values $f(s_{k-M+1}), \ldots$ in order to obtain the next state, s_{k+1}. They are implicit, if they also use the function value $f(s_{k+1})$. A major issue with linear multistep methods is stability, and their analysis needs to regard a dynamic system with an enlarged state space consisting of all M past values. A very popular and successful class of implicit multistep methods are called the *backward differentiation formulae (BDF)* methods. In each step, they formulate an implicit equation in the variable s_{k+1} by constructing the interpolation polynomial $p_k(t; s_{k+1})$ of order M that interpolates the known values $s_{k-M+1}, \ldots, s_k$ as well the unknown s_{k+1}, and then equates the derivative of this polynomial with the function value, i.e. solves the nonlinear equation

$$p_k'(t_{k+1}; s_{k+1}) = f(s_{k+1}, t_{k+1})$$

in the unknown s_{k+1}. Note that the fact that only a nonlinear system of size n_x needs to be solved in each step of the BDF method is in contrast to m-stage implicit RK methods, which need to solve a system of size $(m \times n_x)$. Still, the convergence of the BDF method is of order M. It is, however, not possible to construct stable BDF methods of arbitrary orders, as their stability regions shrink, i.e. they become unstable even for stable systems and very short step lengths Δt. The highest possible order for a BDF method is $M = 6$, while the BDF method with $M = 7$ is not stable anymore: if it is applied to the testequation $\dot{x} = -\lambda x$ with $\lambda > 0$ it diverges even if an arbitrarily small step size Δt is used. It is interesting to compare linear multistep methods with the sequence of Fibonacci numbers that also use a linear combination of the last two numbers in order to compute the next one (i.e. $M = 2$). While the Fibonacci numbers do not solve a differential equation, the analysis of their growth is equivalent to the analysis of the stability of linear multistep methods. For more details, the reader is referred e.g. to [23, 24, 3].

1.2.6 Differential Algebraic Equations

A more general class than ODE are the Differential Algebraic Equations (DAE) which in their easiest form are semi-explicit, i.e. can be written as

$$\dot{x} = f(x, z, t) \tag{1.25}$$
$$0 = g(x, z, t). \tag{1.26}$$

with *differential states* $x \in \mathbb{R}^{n_x}$ and *algebraic states* $z \in \mathbb{R}^{n_z}$ and the algebraic equations with $g(x, z, t) \in \mathbb{R}^{n_z}$, i.e. the Jacobian $\frac{\partial g}{\partial z}$ is a square matrix. A DAE is called of *index one* if this Jacobian is invertible at all relevant points. The existence and unqueness results can be generalized to this case by eliminating z as a function of (x, t) and reducing the DAE integration problem to the ODE case. Note that only initial values for the differential states can be imposed, i.e. the initial condition is $x(0) = x_0$, while $z(0)$ is implicitly defined by (1.26).

Some numerical methods also proceed by first eliminating z numerically and then solving an ODE, which has the advantage that any ODE integrator, even an explicit one, can be applied. On the other hand, this way needs nested Newton iterations and usually destroys any sparsity in the Jacobians.

Fortunately, most implicit numerical integration methods can easily be generalized to the case of DAE systems of index one, e.g. the implicit Euler method would solve in each step the following enlarged system in the unknowns (s^x_{k+1}, s^z_{k+1}):

$$\frac{s^x_{k+1} - s^x_k}{\Delta t} = f(s^x_{k+1}, s^z_{k+1}, t_{k+1}) \tag{1.27}$$
$$0 = g(s^x_{k+1}, s^z_{k+1}, t_{k+1}) \tag{1.28}$$

DAEs can more generally be of fully implicit type and the distinction between differential and algebraic states could not be given a priori, in which case they would just be written as a set of equations $F(\dot{x}, x, t) = 0$. Before solving this case numerically, an analysis and possible index reduction has to be performed, and to be decided, which initial conditions can be imposed without causing inconsistencies. If, however, index one can be guaranteed and a division into algebraic and differential states exists, then it is perfectly possible to generalize implicit integration methods to fully implicit equations of the form $F(\dot{x}, x, z, t) = 0$.

1.2.7 Solution Map and Sensitivities

In the context of optimal control, derivatives of the dynamic system simulation are needed for the numerical algorithms. Following Theorem 1.1 we know already that a unique ODE (or DAE) solution exists to the IVP $\dot{x} = f(x,t), x(0) = x_0$ under mild conditions, namely Lipschitz continuity of f w.r.t. x and continuity w.r.t. t. This solution exists locally, i.e. if the time $T > 0$ is chosen small enough, on the whole interval $[0, T]$. Note that for nonlinear continuous time systems – in contrast to discrete time systems – it is very easily possibly even with innocently looking functions f to obtain an "explosion", i.e. a solution that tends to infinity for finite times. For illustration, regard the example $\dot{x} = x^2, x(0) = 1$ which has the explicit solution $x(t) = 1/(1 - t)$ tending to infinity for $t \to 1$. This is why we cannot guarantee an ODE/DAE solution on any given interval $[0, T]$ for arbitrary T, but have to possibly reduce the length of the interval.

In order to discuss the issue of derivatives, which in the dynamic system context are often called "sensitivities", let us now regard an ODE with some parameters $p \in \mathbb{R}^{n_p}$ that enter the function f and assume that f satisfies the assumptions of Theorem 1.1. We regard some values $\bar{x}_0$, $\bar{p}$, T such that the ODE

$$\dot{x} = f(x, p, t), \quad t \in [0, T] \tag{1.29}$$

with $p = \bar{p}$ and $x(0) = \bar{x}_0$ has a unique solution on the whole interval $[0, T]$. For small perturbations of the values $(\bar{p}, \bar{x}_0)$, due to continuity, we will still have a unique solution on the whole interval $[0, T]$. Let us restrict ourselves to a neighborhood $\mathcal{N}$ of $(\bar{p}, \bar{x}_0)$. For each fixed $t \in [0, T]$, we can now regard the well defined and unique solution map $x(t; \cdot) : \mathcal{N} \to \mathbb{R}^{n_x}$, $(p, x_0) \mapsto x(t; p, x_0)$. This map gives the value $x(t; p, x_0)$ of the unique solution trajectory at time t for given parameter p and initial value x_0. A natural question to ask is if this map is differentiable. Fortunately, it is possible to show that if f is m-times continuously differentiable with respect to both x and p, then the solution map $x(t; \cdot)$ is also m-times continuously differentiable.

To regard a simple and important example: for linear continuous time systems

$$\dot{x} = Ax + Bp$$

the map $x(t; p, x_0)$ is explicitly given as

$$x(t; p, x_0) = \exp(At)x_0 + \int_0^t \exp(A(t - \tau))Bp\mathrm{d}\tau,$$

where $\exp(A)$ is the matrix exponential. Like the function f, this map is infinitely many times differentiable (and even well defined for all times t, as linear systems cannot explode). In the general nonlinear case, the map $x(t; p, x_0)$ can only be generated by a numerical simulation routine. The computation of derivatives of this numerically generated map is a delicate issue that we discuss in detail in the third part of the course. To mention already the main difficulty, note that all practical numerical integration routines are adaptive, i.e. might choose to do different numbers of integration steps for different IVPs. This renders the numerical approximation of the map $x(t; p, x_0)$ typically non-differentiable. Thus, multiple calls of a black-box integrator and application of finite differences might result in very wrong derivative approximations.

Chapter 2

Nonlinear Optimization

Nonlinear optimization algorithms are at the basis of many numerical optimal control methods. Traditionally, this field is called *Nonlinear Programming*. In this chapter we briefly review the most important concepts from this area that are needed throughout the course. Also, in order to choose the right algorithm for a practical problem, we should know how to classify it and which mathematical structures can be exploited. Replacing an inadequate algorithm by a suitable one can make solution times many orders of magnitude shorter. An important such structure is convexity which allows us to to find global minima by searching for local minima only.

For anyone not really familiar with the concepts of nonlinear optimization that are only very briefly outlined here, it is highly recommended to have a look at the excellent Springer text book "Numerical Optimization" by Jorge Nocedal and Steve Wright [63]. Who likes to know more about convex optimization than the much too brief outline given in this script is recommended to have a look at the equally excellent Cambridge University Press text book "Convex Optimization" by Stephen Boyd and Lieven Vandenberghe [22], whose PDF is freely available.

2.1 Nonlinear Programming (NLP)

In this course we mainly need algorithms for general Nonlinear Programming Problems (NLP), which are given in the form

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) && \text{(2.1a)} \\ \text{subject to} \quad & g(x) = 0, && \text{(2.1b)} \\ & h(x) \leq 0, && \text{(2.1c)} \end{aligned}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^{n_g}$, $h : \mathbb{R}^n \to \mathbb{R}^{n_h}$, are assumed to be continuously differentiable at least once, often twice and sometimes more. Differentiability of all problem functions allows us to use alorithms that are based on derivatives, in particular the so called "Newton-type optimization methods" which are the basis of most optimal control algorithms presented in this course. But many problems have more structure, which we should recognize and exploit in order to solve problems faster.

Note that in this chapter, as in the previous chapter, we use again the variable name x and the function names f, and g, but all with a completely different meaning. Also, as the variable space of x is omnipresent, we drop the subindex in n_x so that $x \in \mathbb{R}^n$ in this chapter.

2.1.1 Definitions

Let us start by making a few definitions to clarify the language.

Definition 2.1
The "feasible set" Ω is defined to be the set $\Omega := \{x \in \mathbb{R}^n | g(x) = 0,\ h(x) \leq 0\}$.

Definition 2.2
The point $x^* \in \mathbb{R}^n$ is a "global minimizer" if and only if (iff) $x^* \in \Omega$ and $\forall x \in \Omega :\ f(x) \geq f(x^*)$.

Definition 2.3
The point $x^* \in \mathbb{R}^n$ is a "local minimizer" iff $x^* \in \Omega$ and there exists a neighborhood $\mathcal{N}$ of x^* (e.g. an open ball around x^*) so that $\forall x \in \Omega \cap \mathcal{N} :\ f(x) \geq f(x^*)$.

Definition 2.4 (Active/Inactive Constraint)
An inequality constraint $h_i(x) \leq 0$ is called "active" at $x^* \in \Omega$ iff $h_i(x^*) = 0$ and otherwise "inactive".

Definition 2.5 (Active Set)
The index set $\mathcal{A}(x^*) \subset \{1, \ldots, n_h\}$ of active constraints is called the "active set".

2.2 Convex Optimization Problems

"The great watershed in optimization is not between linearity and nonlinearity, but convexity and nonconvexity"

R. Tyrrell Rockafellar

What is convexity, and why is it so important for optimizers?

Definition 2.6 (Convex Set)
A set $\Omega \subset \mathbb{R}^n$ is convex if

$$\forall x, y \in \Omega, t \in [0,1]: \; x + t(y - x) \in \Omega. \tag{2.2}$$

("all connecting lines lie inside set")

Definition 2.7 (Convex Function)
A function $f : \Omega \to \mathbb{R}$ is convex, if Ω is convex and if

$$\forall x, y \in \Omega, t \in [0,1]: \; f(x + t(y - x)) \leq f(x) + t(f(y) - f(x)). \tag{2.3}$$

("all secants are above graph"). This definition is equivalent to saying that the Epigraph of f, i.e. the set $\{(x, s) \in \mathbb{R}^n \times \mathbb{R} | x \in \Omega, \; s \geq f(x)\}$, is a convex set.

Definition 2.8 (Concave Function)
A function $f : \Omega \to \mathbb{R}$ is called "concave" if $(-f)$ is convex.

Note that the feasible set Ω of an optimization problem is convex if the function g is affine and the functions h_i are convex, as supported by the following theorem that is easy to prove.

Theorem 2.1 (Convexity of Sublevel Sets)**:** *The sublevel set $\{x \in \Omega | h(x) \leq 0\}$ of a convex function $h : \Omega \to \mathbb{R}$ is convex.*

Definition 2.9 (Convex Optimization Problem)
An optimization problem with convex feasible set Ω and convex objective function $f : \Omega \to \mathbb{R}$ is called a "convex optimization problem".

Theorem 2.2 (Local Implies Global Optimality for Convex Problems)**:** *For a convex optimization problem, every local minimum is also a global one.*

2.2.1 How to detect convexity of functions?

There exists a whole algebra of operations that preserve convexity of functions and sets, which is excellently explained in the text books on convex optimization [8, 22]. Here we only mention an important fact that is related to the positive curvature of a function. Before we proceed, we introduce an important definition often used in this course.

Definition 2.10 (Generalized Inequality for Symmetric Matrices)
We write for a symmetric matrix $B = B^T$, $B \in \mathbb{R}^{n\times n}$ that "$B\succcurlyeq 0$" if and only if B is *positive semi-definite* i.e., if $\forall z \in \mathbb{R}^n: \; z^T Bz \geq 0$, or, equivalently, if all (real) eigenvalues of the symmetric matrix B are non-negative:

$$B\succcurlyeq 0 \iff \min \text{eig}\,(B) \geq 0.$$

We write for two such symmetric matrices that "$A\succcurlyeq B$" iff $A-B\succcurlyeq 0$, and "$A\preccurlyeq B$" iff $B\succcurlyeq A$. We say $B\succ 0$ iff B is *positive definite*, i.e. if $\forall z \in \mathbb{R}^n \setminus \{0\} : z^T Bz > 0$, or, equivalently, if all eigenvalues of B are positive

$$B\succ 0 \iff \min \text{eig}(B) > 0.$$

Theorem 2.3 (Convexity for C^2 Functions)**:** *Assume that $f : \Omega \to \mathbb{R}$ is twice continuously differentiable and Ω convex and open. Then holds that f is convex if and only if for all $x \in \Omega$ the Hessian is positive semi-definite, i.e.*

$$\forall x \in \Omega: \quad \nabla^2 f(x)\succcurlyeq 0. \tag{2.4}$$

Example 2.1 (Quadratic Function)**:** The function $f(x) = c^T x + \frac{1}{2}x^T Bx$ is convex if and only if $B\succcurlyeq 0$, because $\forall x \in \mathbb{R}^n: \; \nabla^2 f(x) = B$.

Software: An excellent tool to formulate and solve convex optimization problems in a MATLAB environment is CVX, which is available as open-source code and easy to install.

2.3 Important Special Classes of Optimization Problems

2.3.1 Quadratic Programming (QP)

If in the general NLP formulation (2.1) the constraints g, h are affine, and the objective is a linear-quadratic function, we call the resulting problem a Quadratic Programming Problem or Quadratic Program (QP). A general QP can be formulated as follows.

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad c^T x + \frac{1}{2}x^T Bx \tag{2.5a}$$

$$\text{subject to} \quad Ax - b = 0, \tag{2.5b}$$

$$Cx - d \leq 0. \tag{2.5c}$$

Here, the problem data are $c \in \mathbb{R}^n, A \in \mathbb{R}^{n_g\times n}, b \in \mathbb{R}^{n_g}, C \in \mathbb{R}^{n_h\times n}, d \in \mathbb{R}^{n_h}$, as well as the "Hessian matrix" $B \in \mathbb{R}^{n\times n}$. Its name stems from the fact that $\nabla^2 f(x) = B$ for $f(x) = c^T x + \frac{1}{2}x^T Bx$. If $B = 0$ the QP degenerates to a linear program (LP).

The eigenvalues of B decide on convexity or non-convexity of a QP, i.e. the possibility to solve it in polynomial time to global optimality, or not. If $B \succcurlyeq 0$ we speak of a convex QP, and if $B \succ 0$ we speak of a strictly convex QP. The latter class has the agreeable property that it always has unique minimizers.

Software for solving a QP Problem: MATLAB: quadprog. Commercial: CPLEX, MOSEK. Open-source: CVX, qpOASES.

2.3.2 Mixed-Integer Programming (MIP)

A mixed integer programming problem or mixed integer program (MIP) is a problem with both real and integer decision variables. A MIP can be formulated as follows:

$$\begin{aligned}
&\underset{\substack{x \in \mathbb{R}^n \\ z \in \mathbb{Z}^m}}{\text{minimize}} \quad && f(x,z) && \text{(2.6a)} \\
&\text{subject to} \quad && g(x,z) = 0, && \text{(2.6b)} \\
& && h(x,z) \leq 0. && \text{(2.6c)}
\end{aligned}$$

Definition 2.11 (Mixed integer non-linear program (MINLP))
If f, g, h are twice differentiable in x and z we speak of a mixed integer non-linear program. Generally speaking, these problems are very hard to solve, due to the combinatorial nature of the variables z.

However, for special problem classes the search through the combinatorial tree can be made more efficient than pure enumeration. Two important examples of such problems are given in the following.

Definition 2.12 (Mixed integer linear program (MILP))
If f, g, h are affine in both x and z we speak of a mixed linear integer program. These problems can efficiently be solved with codes such as the commercial and very powerful code `CPLEX` or the free code `lp_solve` with a nice manual `http://lpsolve.sourceforge.net/5.5/` which is however not suitable for large problems. A famous problem in this class is the "travelling salesman problem", which has only discrete decision variables. Linear integer programming is often just called "Integer programming (IP)". It is by far the largest research area in the discrete optimization community.

Definition 2.13 (Mixed integer quadratic programs (MIQP))
If g, h are affine and f convex quadratic in both x and z we speak of a mixed integer QP (MIQP).

These problems are also efficiently solvable, mostly by commercial solvers (e.g. CPLEX).

In optimal control, mixed integer problems arise because some states or some controls can be integer valued, in which case the dynamic system is called a "hybrid system". Generally speaking, hybrid optimal control problems are more difficult to solve compared to problems with only continuous variables. One exception is the case when dynamic programming can be applied to a problem with purely discrete state and control spaces. Most of this lecture is concerned with continuous optimization.

2.4 First Order Optimality Conditions

An important question in continuous optimization is if a feasible point $x^* \in \Omega$ satisfies necessary first order optimality conditions. If it does not satisfy these conditions, x^* cannot be a local minimizer. If it does satisfy these conditions, it is a hot candidate for a local minimizer. If the problem is convex, these conditions are even *sufficient* to guarantee that it is a global optimizer. Thus, most algorithms for nonlinear optimization search for such points. The first order condition can only be formulated if a technical "constraint qualification" is satisfied, which in its simplest and numerically most attractive variant coms in the following form.

Definition 2.14 (LICQ)
The "linear independence constraint qualification" (LICQ) holds at $x^* \in \Omega$ iff all vectors $\nabla g_i(x^*)$ for $i \in \{1, \ldots, n_g\}$ & $\nabla h_i(x^*)$ for $i \in \mathcal{A}(x^*)$ are linearly independent.

To give further meaning to the LICQ condition, let us combine all active inequalities with all equalities in a map $\tilde{g}$ defined by stacking all functions on top of each other in a colum vector as follows:

$$\tilde{g}(x) = \begin{bmatrix} g(x) \\ h_i(x)(i \in \mathcal{A}(x^*)) \end{bmatrix}. \tag{2.7}$$

LICQ is then equivalent to full row rank of the Jacobian matrix $\frac{\partial \tilde{g}}{\partial x}(x^*)$.

2.4.1 The Karush-Kuhn-Tucker Optimality Conditions

This condition allows us to formulate the famous KKT conditions that are due to Karush [51] and Kuhn and Tucker [54].

Theorem 2.4 (KKT Conditions)**:** *If x^* is a local minimizer of the NLP* (2.1) *and LICQ holds* *at x^* then there exist so called multiplier vectors $\lambda \in \mathbb{R}^{n_g}$ and $\mu \in \mathbb{R}^{n_h}$ with*

$$\begin{aligned}
\nabla f(x^*) + \nabla g(x^*)\lambda^* + \nabla h(x^*)\mu^* &= 0 && (2.8a)\\
g(x^*) &= 0 && (2.8b)\\
h(x^*) &\leq 0 && (2.8c)\\
\mu^* &\geq 0 && (2.8d)\\
\mu_i^* h_i(x^*) &= 0, \quad i = 1, \ldots, n_h. && (2.8e)
\end{aligned}$$

Regarding the notation used in the first line above, please observe that in this script we use the gradient symbol ∇ also for functions g, h with multiple outputs, not only for scalar functions like f. While ∇f is a column vector, in ∇g we collect the gradient vectors of all output components in a matrix which is the transpose of the Jacobian, i.e. $\nabla g(x) := \frac{\partial g}{\partial x}(x)^T$. *Note:* The KKT conditions are the First order necessary conditions for optimality (FONC) for constrained optimization, and are thus the equivalent to $\nabla f(x^*) = 0$ in unconstrained optimization. In the special case of convex problems, the KKT conditions are not only *necessary* for a *local* minimizer, but even *sufficient* for a *global* minimizer. In fact, the following extremely important statement holds.

Theorem 2.5: *Regard a convex NLP and a point x^* at which LICQ holds. Then:*

$$x^* \textit{ is a global minimizer} \Longleftrightarrow \exists \lambda, \mu \textit{ so that the KKT conditions hold.}$$

2.4.2 The Lagrangian Function

Definition 2.15 (Lagrangian Function)
We define the so called "Lagrangian function" to be

$$\mathcal{L}(x, \lambda, \mu) = f(x) + \lambda^T g(x) + \mu^T h(x). \tag{2.9}$$

Here, we have used again the so called "Lagrange multipliers" or "dual variables" $\lambda \in \mathbb{R}^{n_g}$ and $\mu \in \mathbb{R}^{n_h}$. The Lagrangian function plays a crucial role in both convex and general nonlinear optimization, not only as a practical shorthand within the KKT conditions: using the definition of the Lagrangian, we have (2.8a) $\Leftrightarrow \nabla_x \mathcal{L}(x^*, \lambda^*, \mu^*) = 0$.

Remark 1: In the absence of inequalities, the KKT conditions simplify to $\nabla_x \mathcal{L}(x, \lambda) = 0$, $g(x) = 0$, a formulation that is due to Lagrange and was much earlier known than the KKT conditions.

Remark 2: The KKT conditions require the inequality multipliers μ to be positive, $\mu \geq 0$, while the sign of the equality multipliers λ is arbitrary. An interesting observation is that for a convex problem with f and all h_i convex and g affine, and for $\mu \geq 0$, the Lagrangian function is a convex function in x. This often allows us to explicitly find the unconstrained minimum of the Lagrangian for any given λ and $\mu \geq 0$, which is called the Lagrange dual function, and which can be shown to be an underestimator of the minimum. Maximizing this underestimator over all λ and $\mu \geq 0$ leads to the concepts of weak and strong duality.

2.4.3 Complementarity

The last three KKT conditions (2.8c)-(2.8e) are called the *complementarity* conditions. For each index i, they define an L-shaped set in the (h_i, μ_i) space. This set is not a smooth manifold but has a non-differentiability at the origin, i.e. if $h_i(x^*) = 0$ and also $\mu_i^* = 0$. This case is called a weakly active constraint. Often we want to exclude this case. On the other hand, an active constraint with $\mu_i^* > 0$ is called strictly active.

Definition 2.16
Regard a KKT point (x^*, λ^*, μ^*). We say that *strict complementarity* holds at this KKT point iff all active constraints are strictly active.

Strict complementarity is a favourable condition because, together with a second order condition, it implies that the active set is stable against small perturbations. It also makes many theorems easier to formulate and to prove, and is also required to prove convergence of some numerical methods.

2.5 Second Order Optimality Conditions

In case of strict complementarity at a KKT point (x^*, λ^*, μ^*), the optimization problem can locally be regarded to be a problem with equality constraints only, namely those within the function $\tilde{g}$ defined in Equation (2.7). Though more complex second order conditions can be formulated that are applicable even when strict complementarity does not hold, we restrict ourselves here to this special case.

Theorem 2.6 (Second Order Optimality Conditions)**:** *Let us regard a point x^* at which LICQ holds together with multipliers λ^*, μ^* so that the KKT conditions* (2.8a)-(2.8e) *are satisfied and let strict complementarity hold. Regard a basis matrix $Z \in \mathbb{R}^{n \times (n - n_{\tilde{g}})}$ of the null space of $\frac{\partial \tilde{g}}{\partial x}(x^*) \in \mathbb{R}^{n_{\tilde{g}} \times n}$, i.e. Z has full column rank and $\frac{\partial \tilde{g}}{\partial x}(x^*) Z = 0$.*

Then the following two statements hold:

(a) If x^* is a local minimizer, then $Z^T \nabla_x^2 \mathcal{L}(x^*, \lambda^*, \mu^*) Z \succcurlyeq 0$.
(Second Order Necessary Condition, short : SONC)

(b) If $Z^T \nabla_x^2 \mathcal{L}(x^*, \lambda^*, \mu^*) Z \succ 0$, then x^* is a local minimizer.
This minimizer is unique in its neighborhood, i.e., a strict local minimizer, and stable against small differentiable perturbations of the problem data. (Second Order Sufficient Condition, short: SOSC)

The matrix $\nabla_x^2 \mathcal{L}(x^*, \lambda^*, \mu^*)$ plays an important role in optimization algorithms and is called the *Hessian of the Lagrangian*, while its projection on the null space of the Jacobian, $Z^T \nabla_x^2 \mathcal{L}(x^*, \lambda^*, \mu^*) Z$, is called the *reduced Hessian.*

2.5.1 Quadratic Problems with Equality Constraints

To illustrate the above optimality conditions, let us regard a QP with equality constraints only.

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad c^T x + \frac{1}{2} x^T B x \tag{2.10a}$$

$$\text{subject to} \quad Ax + b = 0. \tag{2.10b}$$

We assume that A has full row rank i.e., LICQ holds. The Lagrangian is $\mathcal{L}(x, \lambda) = c^T x + \frac{1}{2} x^T B x + \lambda^T (Ax + b)$ and the KKT conditions have the explicit form

$$c + Bx + A^T \lambda = 0 \tag{2.11a}$$

$$b + Ax = 0. \tag{2.11b}$$

This is a linear equation system in the variable (x, λ) and can be solved if the so called *KKT matrix*

$$\begin{bmatrix} B & A^T \\ A & 0 \end{bmatrix}$$

is invertible. In order to assess if the unique solution (x^*, λ^*) of this linear system is a minimizer, we need first to construct a basis Z of the null space of A, e.g. by a full QR factorization of $A^T = QR$ with $Q = (Y|Z)$ square orthonormal and $R = (\bar{R}^T|0)^T$. Then we can check if the reduced Hessian matrix $Z^T B Z$ is positive semidefinite. If it is not, the objective function has negative curvature in at least one of the feasible directions and x^* cannot be a minimizer. If on the other hand $Z^T B Z \succ 0$ then x^* is a strict local minimizer. Due to convexity this would also be the global solution of the QP.

2.5.2 Invertibility of the KKT Matrix and Stability under Perturbations

An important fact is the following. If the second order sufficient conditions for optimality of Theorem 2.6 (b) hold, then it can be shown that the KKT-matrix

$$\begin{bmatrix} \nabla_x^2 \mathcal{L}(x^*, \lambda^*, \mu^*) & \frac{\partial \tilde{g}}{\partial x}(x^*)^T \\ \frac{\partial \tilde{g}}{\partial x}(x^*) & \end{bmatrix}$$

is invertible. This implies that the solution is stable against perturbations. To see why, let us regard a perturbed variant of the optimization problem (2.1)

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) + \delta_f^T x && (2.12a) \\ \text{subject to} \quad & g(x) + \delta_g = 0, && (2.12b) \\ & h(x) + \delta_h \leq 0, && (2.12c) \end{aligned}$$

with small vectors $\delta_f, \delta_g, \delta_h$ of appropriate dimensions that we summarize as $\delta = (\delta_f, \delta_g, \delta_h)$. If a solution exists for $\delta = 0$, the question arises if a solution exists also for small $\delta \neq 0$, and how this solution depends on the perturbation δ. This is is answered by the following theorem.

Theorem 2.7 (SOSC implies Stability of Solutions)**:** *Regard the family of perturbed optimization problems* (2.12) *and assume that for $\delta = 0$ exists a local solution $(x^*(0), \lambda^*(0), \mu^*(0))$ that satisfies LICQ, the KKT condition, strict complementarity, and the second order sufficient condition of Theorem 2.6 (b). Then there exists an $\epsilon > 0$ so that for all $\|\delta\| \leq \epsilon$ exists a unique local solution $(x^*(\delta), \lambda^*(\delta), \mu^*(\delta))$ that depends differentiably on δ. This local solution has the same active set as the nominal one, i.e. its inactive constraint multipliers remain zero and the active constraint multipliers remain positive. The solution does not depend on the inactive constraint perturbations. If $\tilde{g}$ is the combined vector of equalities and active inequalities, and $\tilde{\lambda}$ and $\tilde{\delta}_2$ the corresponding vectors of multipliers and constraint perturbations, then the derivative of the solution $(x^*(\delta), \tilde{\lambda}^*(\delta))$ with respect to $(\delta_1, \tilde{\delta}_2)$ is given by*

$$\frac{d}{d(\delta_1, \tilde{\delta}_2)} \begin{bmatrix} x^*(\delta) \\ \tilde{\lambda}^*(\delta) \end{bmatrix} \Bigg|_{\delta=0} = - \begin{bmatrix} \nabla_x^2 \mathcal{L}(x^*, \lambda^*, \mu^*) & \frac{\partial \tilde{g}}{\partial x}(x^*)^T \\ \frac{\partial \tilde{g}}{\partial x}(x^*) & \end{bmatrix}^{-1} \tag{2.13}$$

This differentiability formula follows from differentiation of the necessary optimality conditions of the parametrized optimization problems with respect to $(\delta_1, \tilde{\delta}_2)$

$$\nabla f(x^*(\delta)) + \frac{\partial \tilde{g}}{\partial x}(x^*)^T \tilde{\lambda} + \delta_1 = 0 \tag{2.14}$$

$$\tilde{g}(x^*(\delta)) + \tilde{\delta}_2 = 0 \tag{2.15}$$

Invertibility of the KKT matrix and stability of the solution under perturbations are very useful facts for the applicability of Newton-type optimization methods that are discussed in the next chapter.

Chapter 3

Newton-Type Optimization Algorithms

3.1 Equality Constrained Optimization

Let us first regard an optimization problem with only equality constraints,

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) \tag{3.1a}$$

$$\text{subject to} \quad g(x) = 0 \tag{3.1b}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^{n_g}$ are both smooth functions. The idea of the Newton-type optimization methods is to apply a variant of Newton's method to solve the nonlinear KKT conditions

$$\nabla_x \mathcal{L}(x, \lambda) = 0 \tag{3.2a}$$

$$g(x) = 0 \tag{3.2b}$$

In order to simplify notation, we define

$$w := \begin{bmatrix} x \\ \lambda \end{bmatrix} \text{ and } F(w) := \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda) \\ g(x) \end{bmatrix} \tag{3.3}$$

with $w \in \mathbb{R}^{n+n_g}$, $F : \mathbb{R}^{n+n_g} \to \mathbb{R}^{n+n_g}$, so that we can compactly formulate the above nonlinear root finding problem as

$$F(w) = 0. \tag{3.4}$$

Starting from an initial guess w_0, Newton's method generates a sequence of iterates $\{w_k\}_{k=0}^{\infty}$ by linearizing the nonlinear equation at the current iterate

$$F(w_k) + \frac{\partial F}{\partial w_k}(w_k)(w - w_k) = 0 \tag{3.5}$$

and obtaining the next iterate as its solution, i.e.

$$w_{k+1} \quad = \quad w_k - \frac{\partial F}{\partial w_k}(w_k)^{-1} F(w_k) \tag{3.6}$$

For equality constrained optimization, the linear system (3.5) has the specific form[1]

$$\begin{bmatrix} \nabla_x \mathcal{L}(x_k, \lambda_k) \\ g(x_k) \end{bmatrix} + \underbrace{\begin{bmatrix} \nabla_x^2 \mathcal{L}(x_k, \lambda_k) & \nabla g(x_k) \\ \nabla g(x_k)^T & 0 \end{bmatrix}}_{\text{KKT-matrix}} \begin{bmatrix} x - x_k \\ \lambda - \lambda_k \end{bmatrix} \quad = \quad 0 \tag{3.7}$$

Using the definition

$$\nabla_x \mathcal{L}(x_k, \lambda_k) \quad = \quad \nabla f(x_k) + \nabla g(x_k) \lambda_k \tag{3.8}$$

we see that the contributions depending on the old multiplier λ_k cancel each other, so that the above system is equivalent to

$$\begin{bmatrix} \nabla f(x_k) \\ g(x_k) \end{bmatrix} + \begin{bmatrix} \nabla_x^2 \mathcal{L}(x_k, \lambda_k) & \nabla g(x_k) \\ \nabla g(x_k)^T & 0 \end{bmatrix} \begin{bmatrix} x - x_k \\ \lambda \end{bmatrix} = 0. \tag{3.9}$$

This formulation shows that the data of the linear system only depend on λ_k via the Hessian matrix. We need not use the exact Hessian matrix, but can approximate it with different methods. This leads to the more general class of Newton-type optimization methods. Using any such approximation $B_k \approx \nabla_x^2 \mathcal{L}(x_k, \lambda_k)$, we finally obtain the Newton-type iteration as

$$\begin{bmatrix} x_{k+1} \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ 0 \end{bmatrix} - \begin{bmatrix} B_k & \nabla g(x_k) \\ \nabla g^T(x_k) & 0 \end{bmatrix}^{-1} \begin{bmatrix} \nabla f(x_k) \\ g(x_k) \end{bmatrix} \tag{3.10}$$

The general *Newton-type method* is summarized in Algorithm 1. If we use $B_k = \nabla_x^2 \mathcal{L}(x_k, \lambda_k)$, we recover the *exact Newton method*.

Algorithm 1 Equality constrained full step Newton-type method

```
Choose: initial guesses $x_0, \lambda_0$, and a tolerance $\epsilon$
Set: $k = 0$

while $\|\nabla \mathcal{L}(x_k, \lambda_k)\| \geq \epsilon$ or $\|g(x_k)\| \geq \epsilon$ do
    obtain a Hessian approximation $B_k$
    get $x_{k+1}, \lambda_{k+1}$ from (3.10)
    $k = k + 1$
end while
```

[1]Recall that in this script we use the convention $\nabla g(x) := \frac{\partial g}{\partial x}(x)^T$ that is consistent with the definition of the gradient $\nabla f(x)$ of a scalar function f being a column vector.

3.1.1 Quadratic Model Interpretation

It is easy to show that x_{k+1} and λ_{k+1} from (3.10) can equivalently be obtained from the solution of a QP:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad \nabla f(x_k)^T (x - x_k) + \frac{1}{2}(x - x_k)^T B_k (x - x_k) \tag{3.11a}$$

$$\text{subject to} \quad g(x_k) + \nabla g(x_k)^T (x - x_k) = 0 \tag{3.11b}$$

So we can interpret the Newton-type optimization method as a "Sequential Quadratic Programming" (SQP) method, where we find in each iteration the solution x^{QP} and λ^{QP} of the above QP and take it as the next NLP solution guess and linearization point x_{k+1} and λ_{k+1}. This interpretation will turn out to be crucial when we treat inequality constraints. But let us first discuss what methods exist for the choice of the Hessian approximation B_k.

3.1.2 The Exact Newton Method

The first and obvious way to obtain B_k is to use the exact Newton method and just set

$$B_k := \nabla_x^2 \mathcal{L}(x_k, \lambda_k)$$

But how can this matrix be computed? Many different ways for computing this second derivative exist. The most straightforward way is a finite difference approximation where we perturb the evaluation of $\nabla \mathcal{L}$ in the direction of all unit vectors $\{e_i\}_{i=1}^n$ by a small quantity $\delta > 0$. This yields each time one column of the Hessian matrix, as

$$\nabla_x^2 \mathcal{L}(x_k, \lambda_k) e_i = \frac{\nabla_x \mathcal{L}(x_k + \delta e_i, \lambda_k) - \nabla_x \mathcal{L}(x_k, \lambda_k)}{\delta} + O(\delta) \tag{3.12}$$

Unfortunately, the evaluation of the numerator of this quotient suffers from numerical cancellation, so that δ cannot be chosen arbitrarily small, and the maximum attainable accuracy for the derivative is $\sqrt{\epsilon}$ if ϵ is the accuracy with which the gradient $\nabla_x \mathcal{L}$ can be obtained. Thus, we loose half the valid digits. If $\nabla_x \mathcal{L}$ was itself already approximated by finite differences, this means that we have lost three quarters of the originally valid digits. More accurate and also faster ways to obtain derivatives of arbitrary order will be presented in the chapter on algorithmic differentiation.

Local convergence rate: The exact Newton method has a *quadratic convergence rate*, i.e. $\|w_{k+1} - w^*\| \leq \frac{\omega}{2} \|w_k - w^*\|^2$. This means that the number of accurate digits doubles in each iteration. As a rule of thumb, once a Newton method is in its area of quadratic convergence, it needs at maximum 6 iterations to reach the highest possible precision.

3.1.3 The Constrained Gauss-Newton Method

Let us regard the special case that the objective $f(x)$ has a nonlinear least-squares form, i.e. $f(x) = \frac{1}{2}\|R(x)\|_2^2$ with some function $R : \mathbb{R}^n \to \mathbb{R}^{n_R}$. In this case we can use a very powerful

Newton-type method which approximates the Hessian B_k using only first order derivatives. It is called the *Gauss-Newton method.* To see how it works, let us thus regard the nonlinear least-squares problem

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & \frac{1}{2}\|R(x)\|_2^2 && (3.13a) \\ \text{subject to} \quad & g(x) = 0 && (3.13b) \end{aligned}$$

The idea of the Gauss-Newton method is to linearize at a given iterate x_k both problem functions R and g, in order to obtain the following approximation of the original problem.

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & \frac{1}{2}\|R(x_k) + \nabla R(x_k)^T(x - x_k)\|_2^2 && (3.14a) \\ \text{subject to} \quad & g(x_k) + \nabla g(x_k)^T(x - x_k) = 0 && (3.14b) \end{aligned}$$

This is a convex QP which can easily be seen by noting that the objective (3.14a) is equal to

$$\frac{1}{2}R(x_k)^T R(x_k) + (x - x_k)^T \underbrace{\nabla R(x_k) R(x_k)}_{=\nabla f(x_k)} + \frac{1}{2}(x - x_k)^T \underbrace{\nabla R(x_k) \nabla R(x_k)^T}_{=:B_k}(x - x_k)$$

which is convex because $B_k \succcurlyeq 0$. Note that the constant term does not influence the solution and can be dropped. Thus, the Gauss-Newton subproblem (3.14) is identical to the SQP subproblem (3.11) with a special choice of the Hessian approximation, namely

$$B_k := \nabla R(x_k) \nabla R(x_k)^T = \sum_{i=1}^{n_R} \nabla R_i(x_k) \nabla R_i(x_k)^T$$

Note that no multipliers λ_k are needed in order to compute B_k. In order to assess the quality of the Gauss-Newton Hessian approximation, let us compare it with the exact Hessian, that is given by

$$\begin{aligned} \nabla_x^2 \mathcal{L}(x, \lambda) &= \textstyle\sum_{i=1}^{n_R} \nabla R_i(x_k) \nabla R_i(x_k)^T + \displaystyle\sum_{i=1}^{n_F} R_i(x) \nabla^2 R_i(x) + \sum_{i=1}^{n_g} \lambda_i \nabla^2 g_i(x) && (3.15) \\ &= B_k + O(\|R(x_k)\|) + O(\|\lambda\|). && (3.16) \end{aligned}$$

One can show that in the solution of a problem holds $\|\lambda^*\| = O(\|R(x^*)\|)$. Thus, in the vicinity of the solution, the difference between the exact Hessian and the the Gauss-Newton approximation B_k is of order $O(\|R(x^*)\|)$.

Local convergence rate: The Gauss-Newton method converges *linearly*, $\|w_{k+1} - w^*\| \leq \kappa \|w_k - w^*\|$ with a contracton rate $\kappa = O(\|R(x^*)\|)$. Thus, it converges fast if the residuals $R_i(x^*)$ are small, or equivalently, if the objective is close to zero, which is our desire in least-squares problems. In estimation problems, a low objective corresponds to a "good fit". Thus the Gauss-Newton method is only attracted by local minima with a small function value, a favourable feature in practice.

3.1.4 Hessian Approximation by Quasi-Newton BFGS Updates

Besides the exact Hessian and the Gauss-Newton Hessian approximation, there is another widely used way to obtain a Hessian approximation B_k within the Newton-type framework. It is based on the observation that the evaluation of $\nabla_x \mathcal{L}$ at different points can deliver curvature information that can help us to estimate $\nabla_x^2 \mathcal{L}$, similar as it can be done by finite differences, cf. Equation (3.12), but without any extra effort per iteration besides the evaluation of $\nabla f(x_k)$ and $\nabla g(x_k)$ that we need anyway in order to compute the next step. Quasi-Newton Hessian update methods use the previous Hessian approximation B_k, the step $s_k := x_{k+1} - x_k$ and the gradient difference $y_k := \nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}) - \nabla_x \mathcal{L}(x_k, \lambda_{k+1})$ in order to obtain the next Hessian approximation B_{k+1}. As in the finite difference formula (3.12), this approximation shall satisfy the *secant condition*

$$B_{k+1} s_k = y_k \tag{3.17}$$

but because we only have one single direction s_k, this condition does not uniquely determine B_{k+1}. Thus, among all matrices that satisfy the secant condition, we search for the ones that minimize the distance to the old B_k, measured in some suitable norm. The most widely used Quasi-Newton update formula is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update that can be shown to minimize a weighted Frobenius norm. It is given by the explicit formula:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{s_k^T y_k}. \tag{3.18}$$

Local convergence rate: It can be shown that $B_k \to \nabla_x^2 \mathcal{L}(x^*, \lambda^*)$ in the relevant directions, so that *superlinear convergence* is obtained with the BFGS method, i.e. $\|w_{k+1} - w^*\| \leq \kappa_k \|w_k - w^*\|$ with $\kappa_k \to 0$.

3.2 Local Convergence of Newton-Type Methods

We have seen three examples for Newton-type optimization methods which have different rates of local convergence if they are started close to a solution. They are all covered by the following theorem that exactly states the conditions that are necessary in order to obtain local convergence.

Theorem 3.1 (Newton-Type Convergence)**:** *Regard the root finding problem*

$$F(w) = 0, \quad F : \mathbb{R}^n \to \mathbb{R}^n \tag{3.19}$$

with w^* a local solution satisfying $F(w^*) = 0$ and a Newton-type iteration $w_{k+1} = w_k - M_k^{-1} F(w_k)$ with $M_k \in \mathbb{R}^{n \times m}$ invertible for all k. Let us assume a Lipschitz condition on the Jacobian $J(w) := \frac{\partial F}{\partial w}(w)$ as follows:

$$\|M_k^{-1}(J(w_k) - J(w))\| \leq \omega \|w_k - w^*\| \tag{3.20}$$

Let us also assume a bound on the distance of approximation M_k from the true Jacobian $J(w_k)$:

$$\|M_k^{-1}(J(w_k) - M_k)\| \leq \kappa_k \tag{3.21}$$

where $\kappa_k \leq \kappa$ with $\kappa < 1$. Finally, we assume that the initial guess w_0 is sufficiently close to the solution w^*,

$$\|w_0 - w^*\| \leq \frac{2}{\omega}(1-\kappa). \tag{3.22}$$

Then $w_k \to w^*$ with the following linear contraction in each iteration:

$$\|w_{k+1} - w^*\| \quad \leq \quad \left(\kappa_k + \frac{\omega}{2}\|w_k - w^*\|\right) \quad \cdot \quad \|w_k - w^*\|. \tag{3.23}$$

If $\kappa_k \to 0$, this results in a superlinear convergence rate, and if $\kappa = 0$ quadratic convergence results.

Noting that in Newton-type optimization we have

$$J(w_k) = \begin{bmatrix} \nabla_x^2 \mathcal{L}(x_k, \lambda_k) & \frac{\partial g}{\partial x}(x_k)^T \\ \frac{\partial g}{\partial x}(x_k) & 0 \end{bmatrix} \tag{3.24}$$

$$M_k = \begin{bmatrix} B_k & \frac{\partial g}{\partial x}(x_k)^T \\ \frac{\partial g}{\partial x}(x_k) & 0 \end{bmatrix} \tag{3.25}$$

$$J(w_k) - M_k = \begin{bmatrix} \nabla_x^2 \mathcal{L}(\cdot) - B_k & 0 \\ 0 & 0 \end{bmatrix} \tag{3.26}$$

the above theorem directly implies the three convergence rates that we had already mentioned.

Corollary: *Newton-type optimization methods converge*

- *quadratically if $B_k = \nabla_x^2 \mathcal{L}(x_k, \lambda_k)$ (exact Newton),*
- *superlinearly if $B_k \to \nabla_x^2 \mathcal{L}(x_k, \lambda_k)$ (BFGS),*
- *linearly if $\|B_k - \nabla_x^2 \mathcal{L}(x_k, \lambda_k)\|$ is small (Gauss-Newton).*

3.3 Inequality Constrained Optimization

When a nonlinear optimization problem with inequality constraints shall be solved, two big families of methods exist, first, nonlinear interior point (IP), and second, sequential quadratic programming (SQP) methods. Both aim at solving the KKT conditions (2.8) which include the non-smooth complementarity conditions, but have different ways to deal with this non-smoothness.

3.3.1 Interior Point Methods

The basic idea of an interior point method is to replace the non-smooth L-shaped set resulting from the complementarity conditions with a smooth approximation, typically a hyberbola. Thus, a smoothing constant $\tau > 0$ is introduced and the KKT conditions are replaced by the smooth equation system

$$\nabla f(x^*) + \nabla g(x^*)\lambda^* + \nabla h(x^*)\mu^* = 0 \tag{3.27a}$$

$$g(x^*) = 0 \tag{3.27b}$$

$$\mu_i^* h_i(x^*) + \tau = 0, \quad i = 1, \ldots, n_h. \tag{3.27c}$$

Note that the last equation ensures that $-h_i(x^*)$ and μ_i^* are both strictly positive and on a hyperbola.[2] For τ very small, the L-shaped set is very closely approximated by the hyperbola, but the nonlinearity is increased. Within an interior point method, we usually start with a large value of τ and solve the resulting nonlinear equation system by a Newton method, and then iteratively decrease τ, always using the previously obtained solution as initialization for the next one.

One way to interpret the above smoothened KKT-conditions is to use the last condition to eliminate $\mu_i^* = -\frac{\tau}{h_i(x^*)}$ and to insert this expression into the first equation, and to note that $\nabla_x (\log(-h_i(x))) = \frac{1}{h_i(x)} \nabla h_i(x))$. Thus, the above smooth form of the KKT conditions is nothing else than the optimality conditions of a *barrier problem*

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) - \tau \sum_{i=1}^{n_h} \log(-h_i(x)) \tag{3.28a}$$

$$\text{subject to} \quad g(x) = 0. \tag{3.28b}$$

Note that the objective function of this problem tends to infinity when $h_i(x) \to 0$. Thus, even for very small $\tau > 0$, the barrier term in the objective function will prevent the inequalities to be violated. The *primal barrier method* just solves the above barrier problem with a Newton-type optimization method for equality constrained optimization for each value of τ. Though easy to implement and to interpret, it is not necessarily the best for numerical treatment, among other because its KKT matrices become very ill-conditioned for small τ. This is not the case for the *primal-dual IP method* that solves the full nonlinear equation system (3.27) including the dual variables μ.

For convex problems, very strong complexity results exist that are based on *self-concordance* of the barrier functions and give upper bounds on the total number of Newton iterations that are needed in order to obtain a numerical approximation of the global solution with a given precision. When an IP method is applied to a general NLP that might be non-convex, we can of course only expect to find a local solution, but convergence to KKT points can still be proven, and these *nonlinear IP methods* perform very well in practice.

[2]In the numerical solution algorithms for this system, we have to ensure that the iterates do not jump to a second hyperbola of infeasible shadow solutions, by shortening steps if necessary to keep the iterates in the correct quadrant.

Software: A very widespread and successful implementation of the nonlinear IP method is the open-source code IPOPT [78, 77]. Though IPOPT can be applied to convex problems and will yield the global solution, dedicated IP methods for different classes of convex optimization problems can exploit more problem structure and will solve these problems faster and more reliably. Most commercial LP and QP solution packages such as CPLEX or MOSEK make use of IP methods, as well as many open-source implementations such as the sparsity exploiting QP solver OOQP.

3.3.2 Sequential Quadratic Programming (SQP) Methods

Another approach to address NLPs with inequalities is inspired by the quadratic model interpretation that we gave before for Newton-type methods. It is called *Sequential Quadratic Programming (SQP)* and solves in each iteration an inequality constrained QP that is obtained by linearizing the objective and constraint functions:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad \nabla f(x_k)^T(x - x_k) + \frac{1}{2}(x - x_k)^T B_k (x - x_k) \tag{3.29a}$$

$$\text{subject to} \quad g(x_k) + \nabla g(x_k)^T(x - x_k) = 0 \tag{3.29b}$$

$$h(x_k) + \nabla h(x_k)^T(x - x_k) \geq 0 \tag{3.29c}$$

Note that the active set is automatically discovered by the QP solver and can change from iteration to iteration. However, under strict complementarity, it will be the same as in the true NLP solution x^* once the SQP iterates x_k are in the neighborhood of x^*.

As before for equality constrained problems, the Hessian B_k can be chosen in different ways. First, in the *exact Hessian SQP method* we use $B_k = \nabla_x^2 \mathcal{L}(x_k, \lambda_k, \mu_k)$, and it can be shown that under the second order sufficient conditions (SOSC) of Theorem 2.6 (b), this method has locally quadratic convergence. Second, in the case of a least-squares objective $f(x) = \frac{1}{2}\|R(x)\|_2^2$, we can use the Gauss-Newton Hessian approximation $B_k = \nabla R(x_k) \nabla R(x_k)^T$, yielding linear convergence with a contraction rate $\kappa = O(\|R(x^*)\|)$. Third, quasi-Newton updates such as BFGS can directly be applied, using the Lagrange gradient difference $y_k := \nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}, \mu^{k+1}) - \nabla_x \mathcal{L}(x_k, \lambda_{k+1}, \mu^{k+1})$ in formula (3.18).

Note that in each iteration of an SQP method, an inequality constrained QP needs to be solved, but that we did not mention yet how this should be done. One way would be to apply an IP method tailored to QP problems. This is indeed done, in particular within SQP methods for large sparse problems. Another way is to use a QP solver that is based on an *active set method*, as sketched in the next subsection.

Software: A successful and sparsity exploiting SQP code is SNOPT [43]. Many optimal control packages such as MUSCOD-II [56] or the open-source package ACADO [49, 1] contain at their basis structure exploiting SQP methods. Also the MATLAB solver `fmincon` is based on an SQP algorithm.

3.3.3 Active Set Methods

Another class of algorithms to address optimization problems with inequalities, the *active set methods*, are based on the following observation: if we would know the active set, then we could solve directly an equality constrained optimization problem and obtain the correct solution. The main task is thus to find the correct active set, and an active set method iteratively refines a guess for the active set that is often called the *working set*, and solves in each iteration an equality constrained problem. This equality constrained problem is particularly easy to solve in the case of linear inequality constraints, for example in LPs and QPs. Many very successful LP solvers are based on an active set method which is called the *simplex algorithm*, whose invention by Dantzig [28] was one of the great breakthroughs in the field of optimization. Also many successful QP solvers are based on active set methods. A major advantage of active set strategies is that they can very efficiently be warm-started under circumstances where a series of related problems have to be solved, e.g. within an SQP method, within codes for mixed integer programming, or in the context of model predictive control (MPC) [39].

3.4 Globalisation Strategies

In all convergence results for the Newton-type algorithms stated so far, we had to assume that the initialization was sufficiently close to the true solution in order to make the algorithm converge, which is not always the case. An approach often used to overcome this problem is to use a *homotopy* between a problem we have already solved and the problem we want to solve: in this procedure, we start with the known solution and then proceed slowly, step by step modifying the relevant problem parameters, towards the problem we want to solve, each time converging the Newton-type algorithm and using the obtained solution as initial guess for the next problem. Applying a homotopy requires more user input than just the specification of the problem, so most available Newton-typ optimization algorithms have so called *globalisation strategies.* Most of these strategies can be interpreted as automatically generated homotopies.

In the ideal case, a globalisation strategy ensures *global convergence*, i.e. the Newton-type iterations converge to a local minimum from arbitrary initial guesses. Note that the terms *global convergence* and *globalisation strategies* have nothing to do with *global optimization*, which is concerned with finding global minima for non-convex problems.

Here, we only touch the topic of globalisation strategies very superficially, and for all details we refer to textbooks on nonlinear optimization and recommend in particular [63].

Two ingredients characterize a globalization strategy: first, a measure of progress, and second, a way to ensure that progress is made in each iteration.

3.4.1 Measuring Progress: Merit Functions and Filters

When two consecutive iterations of a Newton-type algorithm for solution of a constrained optimization problem shall be compared with each other it is not trivial to judge if progress is made by the step. The objective function might be improved, while the constraints might be violated more, or conversely. A *merit function* introduces a scalar measure of progress with the property that each local minimum of the NLP is also a local minimum of the merit function. Then, during the optimization routine, it can be monitored if the next Newton-type iteration gives a better merit function than the iterate before. If this is not the case, the step can be rejected or modified.

A widely used merit function is the *exact L1 merit function*

$$T_1(x) = f(x) + \sigma(\|g(x)\|_1 + \|h^+(x)\|_1)$$

with $f(x)$ the objective, $g(x)$ the residual vector of the equality constraints, and $h^+(x)$ the violations of the inequality constraints, i.e. $h_i^+(x) = \max(0, h_i(x))$ for $i = 1, \ldots, n_h$. Note that the L1 penalty function is non-smooth. If the penalty parameter σ is larger than the largest modulus of any Lagrange multiplier at a local minimum and KKT point (x^*, λ^*, μ^*), i.e. if $\sigma > \max(\|\lambda^*\|_\infty, \|\mu^*\|_\infty)$, then the L1 penalty is exact in the sense that x^* also is a local minimum of $T_1(x)$. Thus, in a standard procedure we require that in each iteration a descent is achieved, i.e. $T_1(x_{k+1}) < T_1(x_k)$, and if it is not the case, the step is rejected or modified, e.g. by a line search or a trust region method.

A disadvantage of requiring a descent in the merit function in each iteration is that the full Newton-type steps might be too often rejected, which can slow down the speed of convergence. Remedies to are e.g. a "watchdog technique" that starting at some iterate x_k allows up to $M - 1$ full Newton-type steps without merit function improvement if the Mth iterate is better, i.e. if at the end holds $T_1(x_{k+M}) < T_1(x_k)$, so that the generosity was justified. If this is not the case, the algorithm jumps back to x_k and enforces strict descent for a few iterations.

A different approach that avoids the arbitrary weighting of objective function and constraint violations within a merit function and often allows to accept more full Newton-steps comes in the form of *filter methods*. They regard the pursuit of a low objective function and low constraint violations as two equally important aims, and accept each step that leads to an improvement in at least one of the two, compared to all previous iterations. To ensure this, a so called *filter* keeps track of the best objective and constraint violation pairs that have been achieved so far, and the method rejects only those steps that are *dominated by the filter* i.e., for which one of the previous iterates had both, a better objective and a lower constraint violation. Otherwise the new iterate is accepted and added to the filter, possibly dominating some other pairs in the filter that can then be removed from the filter. Filter methods are popular because of the fact that they often allow the full Newton-step and still have a global convergence guarantee.

3.4.2 Ensuring Progress: Line Search and Trust-Region Methods

If a full Newton-type step does not lead to progress in the chosen measure, it needs to be rejected. But how can a step be generated that is acceptable? Two very popular ways for this exist, one called *line search*, the other *trust region*.

A line search method takes the result of the QP subproblem as a trial step only, and shortens the step if necessary. If $(x_k^{\mathrm{QP}}, \lambda_k^{\mathrm{QP}}, \mu_k^{\mathrm{QP}})$ is the solution of the QP at an SQP iterate x_k, it can be shown (if the QP multipliers are smaller than σ) that the step vector or *search direction* $(x_k^{\mathrm{QP}} - x_k)$ is a descent direction for the L1 merit function T_1, i.e. descent in T_1 can be enforced by performing, instead of the full SQP step $x_{k+1} = x_k^{\mathrm{QP}}$, a shorter step

$$x_{k+1} = x_k + t(x_k^{\mathrm{QP}} - x_k)$$

with a damping factor or *step length* $t \in (0, 1]$. One popular way to ensure global convergence with help of of a merit function is to require in each step the so called *Armijo condition*, a tightened descent condition, and to perform a *backtracking* line search procedure that starts by trying the full step ($t = 1$) first and iteratively shortens the step by a constant factor ($t \leftarrow t/\beta$ with $\beta > 1$) until this descent condition is satisfied. As said, the L1 penalty function has the desirable property that the search direction is a descent direction so that the Armijo condition will eventually be satisfied if the step is short enough. Line-search methods can also be combined with a filter as a measure of progress, instead of the merit function.

An alternative way to ensure progress is to modify the QP subproblem by adding extra constraints that enforce the QP solution to be in a small region around the previous iterate, the *trust region*. If this region is small enough, the QP solution shall eventually lead to an improvement of the merit function, or be acceptable by the filter. The underlying philosophy is that the linearization is only valid in a region around the linearization point and only here we can expect our QP approximation to be a good model of the original NLP. Similar as for line search methods with the L1 merit function, it can be shown for suitable combinations that the measure of progress can always be improved when the trust region is made small enough. Thus, a trust region algorithm checks in each iteration if enough progress was made to accept the step and adapts the size of the trust region if necessary.

As said above, a more detailed description of different globalisation strategies is given in [63].

Chapter 4

Calculating Derivatives

Derivatives of computer coded functions are needed everywhere in optimization. In order to just check optimality of a point, we need already to compute the gradient of the Lagrangian function. Within Newton-type optimization methods, we need the full Jacobian of the constraint functions. If we want to use an exact Hessian method, we even need second order derivatives of the Lagrangian.

There are many ways to compute derivatives: Doing it by hand is error prone and nearly impossible for longer evaluation codes. Computer algebra packages like Mathematica or Maple can help us, but require that the function is formulated in their specific language. More annoyingly, the resulting derivative code can become extremely long and slow to evaluate.

On the other hand, *finite differences* can always be applied, even if the functions are only available as black-box codes. They are easy to implement and relatively fast, but they necessarily lead to a loss of precision of half the valid digits, as they have to balance the numerical errors that originate from Taylor series truncation and from finite precision arithmetic. Second derivatives obtained by recursive application of finite differences are even more inaccurate. The best perturbation sizes are difficult to find in practice. Note that the computational cost to compute the gradient $\nabla f(x)$ of a scalar function $f : \mathbb{R}^n \to \mathbb{R}$ is $(n+1)$ times the cost of one function evaluation.

We will see that a more efficient way exists to evaluate the gradient of a scalar function, which is also more accurate. The technology is called *algorithmic differentiation (AD)* and requires in principle nothing more than that the function is available in the form of source code in a standard programming language such as C, C++ or FORTRAN.

4.1 Algorithmic Differentiation (AD)

Algorithmic differentiation uses the fact that each differentiable function $F : \mathbb{R}^n \to \mathbb{R}^{n_F}$ is composed of several *elementary operations*, like multiplication, division, addition, subtraction, sinefunctions, exp-functions, etc. If the function is written in a programming language like e.g. C, C++ or FORTRAN, special AD-tools can have access to all these elementary operations. They can process the code in order to generate new code that does not only deliver the function value, but also desired derivative information. Algorithmic differentiation was traditionally called *automatic differentiation*, but as this might lead to confusion with symbolic differentiation, most AD people now prefer the term *algorithmic differentiation*, which fortunately has the same abbreviation. A good and authoritative textbook on AD is [45].

In order to see how AD works, let us regard a function $F : \mathbb{R}^n \to \mathbb{R}^{n_F}$ that is composed of a sequence of m elementary operations. While the inputs $x_1, \dots, x_n$ are given before, each elementary operation ϕ_i, $i = 0, \dots, m-1$ generates another intermediate variable, x_{n+i+1}. Some of these intermediate variables are used as output of the code, but in principle we can regard all variables as possible outputs, which we do here. This way to regard a function evaluation is stated in Algorithm 2 and illustrated in Example 4.1 below.

Algorithm 2 User Function Evaluation via Elementary Operations

```
**Input:** $x_1, \dots, x_n$
**Output:** $x_1, \dots, x_{n+m}$

**for** $i = 0$ to $m-1$ **do**
    $x_{n+i+1} \leftarrow \phi_i(x_1, \dots, x_{n+i})$
**end for**

*Note:* each $\phi_i$ depends on only one or two out of $\{x_1, \dots, x_{n+i}\}$.
```

Example 4.1 (Function Evaluation via Elementary Operations)**:** Let us regard the simple scalar function

$$f(x_1, x_2, x_3) = \sin(x_1 x_2) + \exp(x_1 x_2 x_3)$$

with $n = 3$. We can decompose this function into $m = 5$ elementary operations, namely

$$\begin{array}{rcl} x_4 &=& x_1 x_2 \\ x_5 &=& \sin(x_4) \\ x_6 &=& x_4 x_3 \\ x_7 &=& \exp(x_6) \\ x_8 &=& x_5 + x_7 \end{array}$$

Thus, if the $n = 3$ inputs x_1, x_2, x_3 are given, the $m = 5$ elementary operations $\phi_0, \dots, \phi_4$ compute the $m = 5$ intermediate quantities, $x_4, \dots, x_8$, the last of which is our desired scalar output, x_{n+m}.

The idea of AD is to use the chain rule and differentiate each of the elementary operations ϕ_i separately. There are two modes of AD, on the one hand the “forward” mode of AD, and on the other hand the "backward", "reverse", or "adjoint" mode of AD. In order to present both of them in a consistent form, we first introduce an alternative formulation of the original user function, that uses augmented elementary functions, as follows[1]: we introduce new augmented states

$$\tilde{x}_0 = x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad \tilde{x}_1 = \begin{bmatrix} x_1 \\ \vdots \\ x_{n+1} \end{bmatrix}, \quad \ldots, \quad \tilde{x}_m = \begin{bmatrix} x_1 \\ \vdots \\ x_{n+m} \end{bmatrix} \tag{4.1}$$

as well as new augmented elementary functions $\tilde{\phi}_i : \mathbb{R}^{n+i} \to \mathbb{R}^{n+i+1}$, $\tilde{x}_i \mapsto \tilde{x}_{i+1} = \tilde{\phi}_i(\tilde{x}_i)$ with

$$\tilde{\phi}_i(\tilde{x}_i) = \begin{bmatrix} x_1 \\ \vdots \\ x_{n+i} \\ \phi_i(x_1, \ldots, x_{n+i}) \end{bmatrix}, \quad i = 0, \ldots, m-1. \tag{4.2}$$

Thus, the whole evaluation tree of the function can be summarized as a concatenation of these augmented functions followed by a multiplication with a "selection matrix" C that selects from $\tilde{x}_m$ the final outputs of the computer code.

$$F(x) = C \cdot \tilde{\phi}_{m-1}(\tilde{\phi}_{m-2}(\cdots \tilde{\phi}_1(\tilde{\phi}_0(x)))).$$

The full Jacobian of F, that we denote by $J_F = \frac{\partial F}{\partial x}$ is given by the chain rule as the product of the Jacobians of the augmented elementary functions$\tilde{J}_i = \frac{\partial \tilde{\phi}_i}{\partial \tilde{x}_i}$, as follows:

$$J_F = C \cdot \tilde{J}_{m-1} \cdot \tilde{J}_{m-2} \cdots \tilde{J}_1 \cdot \tilde{J}_0. \tag{4.3}$$

Note that each elementary Jacobian is given as a unit matrix plus one extra row. Also note that the extra row that is here marked with stars $*$ has at maximum two non-zero entries.

$$\tilde{J}_i = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \\ * & * & * & * \end{bmatrix}.$$

For the generation of first order derivatives, algorithmic differentiation uses two alternative ways to evaluate the product of these Jacobians, the *forward* and the *backward mode* as described in the next two sections.

4.2 The Forward Mode of AD

In forward AD we first define a *seed vector* $p \in \mathbb{R}^n$ and then evaluate the directional derivative $J_F p$ in the following way:

$$J_F p = C \cdot (\tilde{J}_{m-1} \cdot (\tilde{J}_{m-2} \cdots (\tilde{J}_1 \cdot (\tilde{J}_0 p)))). \tag{4.4}$$

[1]MD thanks Carlo Savorgnan for having outlined to him this way of presenting forward and backward AD

In order to write down this long matrix product as an efficient algorithm where the multiplications of all the ones and zeros do not cause computational costs, it is customary in the field of AD to use a notation that uses "dot quantities" $\dot{x}_i$ that we might think of as the velocity with which a certain variable changes, given that the input x changes with speed $\dot{x} = p$. We can interpret them as

$$\dot{x}_i \equiv \frac{dx_i}{dx} p.$$

In the augmented formulation, we can introduce dot quantities $\dot{\tilde{x}}_i$ for the augmented vectors $\tilde{x}_i$, for $i = 0, \ldots, m-1$, and the recursion of these dot quantities is just given by the initialization with the seed vector, $\dot{\tilde{x}}_i = p$, and then the recursion

$$\dot{\tilde{x}}_{i+1} = \tilde{J}_i(\tilde{x}_i)\dot{\tilde{x}}_i, \quad i = 0, 1, \ldots, m-1.$$

Given the special structure of the Jacobian matrices, most elements of $\dot{\tilde{x}}_i$ are only multiplied by one and nothing needs to be done, apart from the computation of the last component of the new vector $\dot{\tilde{x}}_{i+1}$. This last component is $\dot{x}_{n+i+1}$ Thus, in an efficient implementation, the forward AD algorithm works as the algorithm below. It first sets the seed $\dot{x} = p$ and then proceeds as follows.

Algorithm 3 Forward Automatic Differentiation

```
Input: $\dot{x}_1, \ldots, \dot{x}_n$ and all partial derivatives $\frac{\partial \phi_i}{\partial x_j}$
Output: $\dot{x}_1, \ldots, \dot{x}_{n+m}$

for $i = 0$ to $m-1$ do
    $\dot{x}_{n+i+1} \leftarrow \sum_{j=1}^{n+i} \frac{\partial \phi_i}{\partial x_j} \dot{x}_j$
end for

Note: each sum consist of only one or two non-zero entries.
```

In forward AD, the function evaluation and the derivative evaluation can be performed in parallel, which eliminates the need to store any internal information. This is best illustrated using an example.

Example 4.2 (Forward Automatic Differentiation)**:** We regard the same example as above, $f(x_1, x_2, x_3) = \sin(x_1 x_2) + \exp(x_1 x_2 x_3)$. First, each intermediate variable has to be computed, and then each line can be differentiated. For given x_1, x_2, x_3 and $\dot{x}_1, \dot{x}_2, \dot{x}_3$, the algorithm proceeds as follows:

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

The result is $\dot{x}_8 = (\dot{x}_1, \dot{x}_2, \dot{x}_3)\nabla f(x_1, x_2, x_3)$.

It can be proven that the computational cost of Algorithm 3 is smaller than two times the cost of Algorithm 2, or short

$$\text{cost}(J_F p) \leq 2\,\text{cost}(F).$$

If we want to obtain the full Jacobian of F, we need to call Algorithm 3 several times, each time with the seed vector corresponding to one of the n unit vectors in $\mathbb{R}^n$, i.e. we have

$$\text{cost}(J_F) \leq 2\,n\,\text{cost}(F).$$

AD in forward mode is slightly more expensive than numerical finite differences, but it is exact up to machine precision.

4.2.1 The "Imaginary trick" in MATLAB

An easy way to obtain high precision derivatives in MATLAB is closely related to AD in forward mode. It is based on the following observation: if $F : \mathbb{R}^n \to \mathbb{R}^{n_F}$ is analytic and can be extended to complex numbers as inputs and outputs, then for any $t > 0$ holds

$$J_F(x)p = \frac{\text{im}(F(x + itp))}{t} + O(t^2). \tag{4.5}$$

In contrast to finite differences, there is no subtraction in the numerator, so there is no danger of numerical cancellation errors, and t can be chosen extremely small, e.g. $t = 10^{-100}$, which means that we can compute the derivative up to machine precision. This "imaginary trick" can most easily be used in a programming language like MATLAB that does not declare the type of variables beforehand, so that real-valued variables can automatically be overloaded with complex-valued variables. This allows us to obtain high-precision derivatives of a given black-box MATLAB code. We only need to be sure that the code is analytic (which most codes are) and that matrix or vector transposes are not expressed by a prime `'` (which conjugates a complex number), but by `transp`.

4.3 The Backward Mode of AD

In backward AD we evaluate the product in Eq. (4.3) in the reverse order compared with forward AD. Backward AD does not evaluate forward directional derivatives. Instead, it evaluates *adjoint directional derivatives*: when we define a *seed vector* $\lambda \in \mathbb{R}^{n_F}$ then backward AD is able to evaluate the product $\lambda^T J_F$. It does so in the following way:

$$\lambda^T J_F = ((((\lambda^T C) \cdot \tilde{J}_{m-1}) \cdot \tilde{J}_{m-2}) \cdots \tilde{J}_1) \cdot \tilde{J}_0. \tag{4.6}$$

When writing this matrix product as an algorithm, we use "bar quantities" instead of the "dot quantities" that we used in the forward mode. These quantities can be interpreted as derivatives of the final output with respect to the respective intermediate quantity. We can interpret

$$\bar{x}_i \equiv \lambda^T \frac{dF}{dx_i}.$$

Each intermediate variable has a bar variable and at the start, we initialize all bar variables with the value that we obtain from $C^T\lambda$. Note that most of these seeds will usually be zero, depending on the output selection matrix C. Then, the backward AD algorithm modifies all bar variables. Backward AD gets most transparent in the augmented formulation, where we have bar quantities $\bar{\tilde{x}}_i$ for the augmented states $\tilde{x}_i$. We can transpose the above Equation (4.6) in order to obtain

$$J_F^T\lambda = \tilde{J}_0^T \cdot (\tilde{J}_1^T \cdots \underbrace{\tilde{J}_{m-1}^T \underbrace{(C^T\lambda)}_{=\bar{\tilde{x}}_m}}_{=\bar{\tilde{x}}_{m-1}}).$$

In this formulation, the initialization of the backward seed is nothing else than setting $\bar{\tilde{x}}_m = C^T\lambda$ and then going in reverse order through the recursion

$$\bar{\tilde{x}}_i = \tilde{J}_i(\tilde{x}_i)^T \bar{\tilde{x}}_{i+1}, \quad i = m-1, m-2, \ldots, 0.$$

Again, the multiplication with ones does not cause any computational cost, but an interesting feature of the reverse mode is that some of the bar quantities can get several times modified in very different stages of the algorithm. Note that the multiplication $\tilde{J}_i^T \bar{\tilde{x}}_{i+1}$ with the transposed Jacobian

$$\tilde{J}_i^T = \begin{bmatrix} 1 & & & * \\ & 1 & & * \\ & & \ddots & * \\ & & 1 & * \end{bmatrix}.$$

modifies at maximum two elements of the vector $\bar{\tilde{x}}_{i+1}$ by adding to them the partial derivative of the elementary operation multiplied with $\bar{x}_{n+i+1}$. In an efficient implementation, the backward AD algorithm looks as follows.

Algorithm 4 Reverse Automatic Differentiation

```
Input: seed vector $\bar{x}_1, \ldots, \bar{x}_{n+m}$ and all partial derivatives $\frac{\partial \phi_i}{\partial x_j}$
Output: $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$

for $i = m-1$ down to 0 do
  for all $j = 1, \ldots, n+i$ do
    $\bar{x}_j \leftarrow \bar{x}_j + \bar{x}_{n+i+1} \frac{\partial \phi_i}{\partial x_j}$
  end for
end for

Note: each inner loop will only update one or two bar quantities.
```

Example 4.3 (Reverse Automatic Differentiation)**:** We regard the same example as before, and want to compute the gradient $\nabla f(x) = (\bar{x}_1, \bar{x}_2, \bar{x}_3)^T$ given (x_1, x_2, x_3). We set $\lambda = 1$. Because the selection matrix C selects only the last intermediate variable as output, i.e. $C = (0, \cdots 0, 1)$, we initialize the seed vector with zeros apart from the last component, which is one. In the reverse mode, the algorithm first has to evaluate the function with all intermediate quantities, and only then it can compute the bar quantities, which it does in reverse order. At the end it obtains, among other, the desired quantitities $(\bar{x}_1, \bar{x}_2, \bar{x}_3)$. The full algorithm is the following.

```
// *** forward evaluation of the function ***
$x_4 = x_1 x_2$
$x_5 = \sin(x_4)$
$x_6 = x_4 x_3$
$x_7 = \exp(x_6)$
$x_8 = x_5 + x_7$

  // *** initialization of the seed vector ***
  $\bar{x}_i = 0, \quad i = 1, \ldots, 7$
  $\bar{x}_8 = 1$

      // *** backwards sweep ***
      // * differentiation of $x_8 = x_5 + x_7$
      $\bar{x}_5 = \bar{x}_5 + 1\,\bar{x}_8$
      $\bar{x}_7 = \bar{x}_7 + 1\,\bar{x}_8$
      // * differentiation of $x_7 = \exp(x_6)$
      $\bar{x}_6 = \bar{x}_6 + \exp(x_6)\bar{x}_7$
      // * differentiation of $x_6 = x_4 x_3$
      $\bar{x}_4 = \bar{x}_4 + x_3 \bar{x}_6$
      $\bar{x}_3 = \bar{x}_3 + x_4 \bar{x}_6$
      // * differentiation of $x_5 = \sin(x_4)$
      $\bar{x}_4 = \bar{x}_4 + \cos(x_4)\bar{x}_5$
      // differentiation of $x_4 = x_1 x_2$
      $\bar{x}_1 = \bar{x}_1 + x_2 \bar{x}_4$
      $\bar{x}_2 = \bar{x}_2 + x_1 \bar{x}_4$
```

The desired output of the algorithm is $(\bar{x}_1, \bar{x}_2, \bar{x}_3)$, equal to the three components of the gradient $\nabla f(x)$. Note that all three are returned in *only one* reverse sweep.

It can be shown that the cost of Algorithm 4 is less than 3 times the cost of Algorithm 2, i.e.,

$$\text{cost}(\lambda^T J_F) \leq 3\,\text{cost}(F).$$

If we want to obtain the full Jacobian of F, we need to call Algorithm 4 several times with the n_F seed vectors corresponding to the unit vectors in $\mathbb{R}^{n_F}$, i.e. we have

$$\text{cost}(J_F) \leq 3\, n_F\, \text{cost}(F).$$

This is a remarkable fact: it means that the backward mode of AD can compute the full Jacobian at a cost that is independent of the state dimension n. This is particularly advantageous if $n_F \ll n$, e.g. if we compute the gradient of a scalar function like the objective or the Lagrangian. The reverse mode can be much faster than what we can obtain by finite differences, where we always need $(n+1)$ function evaluations. To give an example, if we want to compute the gradient of a scalar function $f : \mathbb{R}^n \to \mathbb{R}$ with $n =$1 000 000 and each call of the function needs one second of CPU time, then 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 needs only 4 seconds (1 call of the function plus one backward sweep). Thus, besides being more accurate, backward AD can also be much faster than finite differences.

The only disadvantage of the backward mode of AD is that we have to store all intermediate variables and partial derivatives, in contrast to finite differences or forward AD. A partial remedy to this problem exist in form of *checkpointing* that trades-off computational speed and memory requirements. Instead of all intermediate variables, it only stores some "checkpoints" during the forward evaluation. During the backward sweep, starting at these checkpoints, it re-evaluates parts of the function to obtain those intermediate variables that have not been stored. The optimal number and location of checkpoints is a science of itself. Generally speaking, checkpointing reduces the memory requirements, but comes at the expense of runtime.

From a user perspective, the details of implementation are not too relevant, but it is most important to just know that the reverse mode of AD exists and that it allows in many cases a much more efficient derivative generation than any other technique.

4.3.1 Efficient Computation of the Hessian

A particularly important quantity in Newton-type optimization methods is the Hessian of the Lagrangian. It is the second derivative of the scalar function $\mathcal{L}(x, \lambda, \mu)$ with respect to x. As the multipliers are fixed for the purpose of differentiation, we can for notational simplicity just regard a function $f : \mathbb{R}^n \to \mathbb{R}$ of which we want to compute the Hessian $\nabla^2 f(x)$. With finite differences we would at least need $(n+2)(n+1)/2$ function evaluations in order to compute the Hessian, and due to round-off and truncation errors, the accuracy of a finite difference Hessian would be much lower than the accuracy of the function f: we loose three quarters of the valid digits.

In contrast to this, algorithmic differentiation can without problems be applied recursively, yielding a code that computes the Hessian matrix at the same precision as the function f itself, i.e. typically at machine precision. Moreover, if we use the reverse mode of AD at least once, e.g. by first generating an efficient code for $\nabla f(x)$ (using backward AD) and then using forward AD to obtain the Jacobian of it, we can reduce the CPU time considerably compared to finite differences. Using the above procedure, we would obtain the Hessian $\nabla^2 f$ at a cost of $2\,n$ times the cost of a gradient ∇f, which is about four times the cost of evaluating f alone. This means that we have the following runtime bound:

$$\mathrm{cost}(\nabla^2 f) \leq 8\,n\,\mathrm{cost}(f).$$

A compromise between accuracy and ease of implementation that is equally fast in terms of CPU time is to use backward AD only for computing the first order derivative $\nabla f(x)$, and then to use finite differences for the differentiation of $\nabla f(x)$.

4.4 Algorithmic Differentiation Software

Most algorithmic differentiation tools implement both forward and backward AD, and most are specific to one particular programming language. They come in two different variants: either they use *operator overloading* or *source-code transformation*.

The first class does not modify the code but changes the type of the variables and overloads the involved elementary operations. For the forward mode, each variable just gets an additional dot-quantity, i.e. the new variables are the pairs $(x_i, \dot{x}_i)$, and elementary operations just operate on these pairs, like e.g.

$$(x, \dot{x}) \cdot (y, \dot{y}) = (xy, x\dot{y} + y\dot{x}).$$

An interesting remark is that operator overloading is also at the basis of the imaginary trick in MATLAB were we use the overloading of real numbers by complex numbers and used the small imaginary part as dot quantity and exploited the fact that the extremely small higher order terms disappear by numerical cancellation.

A prominent and widely used AD tool for generic user supplied C++ code that uses operator overloading is ADOL-C. Though it is not the most efficient AD tool in terms of CPU time it is well documented and stable. Another popular tool in this class is CppAD.

The other class of AD tools is based on source-code transformation. They work like a text-processing tool that gets as input the user supplied source code and produces as output a new and very differently looking source code that implements the derivative generation. Often, these codes can be made extremely fast. Tools that implement source code transformations are ADIC for ANSI C, and ADIFOR and TAPENADE for FORTRAN codes.

In the context of ODE or DAE simulation, there exist good numerical integrators with forward and backward differentiation capabilities that are more efficient and reliable than a naive procedure that would consist of taking an integrator and processing it with an AD tool. Examples for integrators that use the principle of forward and backward AD are the code DAESOL-II or the open-source codes from the ACADO Integrators Collection or from the SUNDIALS Suite.

Part II

Discrete Time Optimal Control

Chapter 5

Discrete Time Optimal Control Formulations

Throughout this part of the script we regard for notational simplicity time-invariant dynamical systems with dynamics

$$x_{k+1} \quad = \quad f(x_k, u_k), \quad k = 0, \dots, N-1. \tag{5.1}$$

Recall that u_k are the *controls* and x_k the *states*, with $x_k \in \mathbb{R}^{n_x}$ and $u_k \in \mathbb{R}^{n_u}$.

As discussed in the first chapter, if we know the initial state x_0 and the controls $u_0, \dots, u_{N-1}$, we could simulate the system to obtain all other states. But in optimization, we might have different requirements than just a fixed initial state. We might, for example, have both a fixed initial state and a fixed terminal state that we want to reach. Or we might just look for periodic sequences with $x_0 = x_N$. All these desires on the initial and the terminal state can be expressed by a boundary constraint function

$$r(x_0, x_N) = 0. \tag{5.2}$$

For the case of fixed initial value, this function would just be

$$r(x_0, x_N) \quad = \quad x_0 - \bar{x}_0 \tag{5.3}$$

where $\bar{x}_0$ is the fixed initial value and not an optimization variable. Another example would be to have both ends fixed, resulting in a function r of double the state dimension, namely:

$$r(x_0, x_N) \quad = \quad \begin{bmatrix} x_0 - \bar{x}_0 \\ x_N - \bar{x}_N \end{bmatrix}. \tag{5.4}$$

Finally, periodic boundary conditions can be imposed by setting

$$r(x_0, x_N) \quad = \quad x_0 - x_N. \tag{5.5}$$

Other constraints that are usually present are *path constraint* inequalities of the form

$$h(x_k, u_k) \quad \leq \quad 0, \quad k = 0, \ldots, N-1. \tag{5.6}$$

In the case of upper and lower bounds on the controls, $u_{\min} \leq u_k \leq u_{\max}$, the function h would just be

$$h(x, u) = \begin{bmatrix} u - u_{\max} \\ u_{\min} - u \end{bmatrix}.$$

5.1 Optimal Control Problem (OCP) Formulations

Two major approaches can be distinguished to formulate and numerically solve a discrete time optimal control problem, the *simultaneous* and the *sequential* approach, which we will outline after having formulated the optimal control problem in its standard form.

5.1.1 Original Problem Formulation

Given the system model and constraints, a quite generic discrete time optimal control problem can be formulated as the following constrained NLP:

$$\begin{aligned}
\underset{x_0, u_0, x_1, \ldots, u_{N-1}, x_N}{\text{minimize}} \quad & \sum_{k=0}^{N-1} L(x_k, u_k) \; + \; E(x_N) && (5.7a)\\
\text{subject to} \quad & x_{k+1} - f(x_k, u_k) \; = \; 0, \quad \text{for} \quad k = 0, \ldots, N-1, && (5.7b)\\
& h(x_k, u_k) \; \leq \; 0, \quad \text{for} \quad k = 0, \ldots, N-1, && (5.7c)\\
& r(x_0, x_N) \; = \; 0. && (5.7d)
\end{aligned}$$

We remark that other optimization variables could be present as well, such as a free parameter p that can be chosen but is constant over time, like e.g. the size of a vessel in a chemical reactor or the length of a robot arm. Such parameters could be added to the optimisation formulation above by defining dummy states $\{p_k\}_{k=1}^{N}$ that satisfy the dummy dynamic model equations

$$p_{k+1} \quad = \quad p_k, \quad k = 0, \ldots, N-1. \tag{5.8}$$

Note that the initial value of p_0 is not fixed by these constraints and thus we would have obtained our aim of having a time constant parameter vector that is free for optimization.

5.1.2 The Simultaneous Approach

The nonlinear program (5.7) is large and structured and can thus in principle be solved by any NLP solver. This is called the *simultaneous approach* to optimal control and requires the use of a structure exploiting NLP solver in order to be efficient. Note that in this approach, all original variables, i.e. u_k and x_k remain optimization variables of the NLP. Its name stems from the fact that the NLP solver has to simultaneously solve both, the simulation and the optimization problem. It is interesting to remark that the model equations (5.7b) will for most NLP solvers only be satisfied once the NLP iterations are converged. The simultaneous approach is therefore sometimes referred to as an *infeasible path* approach. The methods *direct multiple shooting* and *direct collocation* that we explain in the third part of this script are simultaneous approaches.

5.1.3 The Reduced Formulation and the Sequential Approach

On the other hand, we know that we could eliminate nearly all states by a forward simulation, and in this way we could reduce the variable space of the NLP. The idea is to keep only x_0 and $U = [u_0^T, \dots, u_{N-1}^T]^T$ as variables. The states $x_1, \dots, x_N$ are eleminated recursively by

$$\begin{aligned} \bar{x}_0(x_0, U) &= x_0 && (5.9a) \\ \bar{x}_{k+1}(x_0, U) &= f(\bar{x}_k(x_0, U), u_k), \quad k = 0, \dots, N-1. && (5.9b) \end{aligned}$$

Then the optimal control problem is equivalent to a *reduced problem* with much less variables, namely the following nonlinear program:

$$\begin{aligned} \underset{x_0, U}{\text{minimize}} \quad & \sum_{k=0}^{N-1} L(\bar{x}_k(x_0, U), u_k) + E(\bar{x}_k(x_0, U)) && (5.10a) \\ \text{subject to} \quad & h(\bar{x}_k(x_0, U), u_k) \leq 0, \quad \text{for} \quad k = 0, \dots, N-1, && (5.10b) \\ & r(x_0, \bar{x}_N(x_0, U)) = 0. && (5.10c) \end{aligned}$$

Note that the model Equation (5.9b) is implicitly satisfied by definition, but is not anymore a constraint of the optimization problem. This reduced problem can now be addressed again by Newton-type methods, but the exploitation of sparsity in the problem is less important. This is called the *sequential* approach, because the simulation problem and optimization problem are solved sequentially, one after the other. Note that the user can observe during all iterations of the optimization procedure what is the resulting state trajectory for the current iterate, as the model equations are satisfied by definition.

If the initial value is fixed, i.e. if $r(x_0, x_N) = x_0 - \bar{x}_0$, one can also eliminate $x_0 \equiv \bar{x}_0$, which reduces the variables of the NLP further.

5.2 Analysis of a Simplified Optimal Control Problem

In order to learn more about the structure of optimal control problems and the relation between the simultaneous and the sequential approach, we regard in this section a simplified optimal control problem in discrete time:

$$
\begin{aligned}
\underset{x_0, u_0, x_1, \ldots, u_{N-1}, x_N}{\text{minimize}} \quad & \sum_{k=0}^{N-1} L(x_k, u_k) \ + \ E(x_N) && \text{(5.11a)}\\
\text{subject to} \quad & f(x_k, u_k) - x_{k+1} \ = \ 0 \quad \text{for} \quad k = 0, \ldots, N-1 && \text{(5.11b)}\\
& r(x_0, x_N) \ = \ 0 && \text{(5.11c)}
\end{aligned}
$$

5.2.1 KKT Conditions of the Simplified Problem

We first summarize the variables as $w = (x_0, u_0, x_1, u_1, \ldots, u_{N-1}, x_N)$ and summarize the multipliers as $\lambda = (\lambda_1, \ldots, \lambda_N, \lambda_r)$. Then the above optimal control problem can be summarized as

$$
\begin{aligned}
\underset{w}{\text{minimize}} \quad & F(w) && \text{(5.12a)}\\
\text{subject to} \quad & G(w) \ = \ 0. && \text{(5.12b)}
\end{aligned}
$$

Here, the objective $F(w)$ is just copied from (5.11a) while $G(w)$ collects all constraints:

$$
G(w) \ = \ \begin{bmatrix} f(x_0, u_0) - x_1 \\ f(x_1, u_1) - x_2 \\ \vdots \\ f(x_{N-1}, u_{N-1}) - x_N \\ r(x_0, x_N) \end{bmatrix}. \qquad \text{(5.12c)}
$$

The Lagrangian function has the form

$$
\begin{aligned}
\mathcal{L}(w, \lambda) \ &= \ F(w) + \lambda^T G(w) \\
&= \ \sum_{k=0}^{N-1} L(x_k, u_k) + E(x_N) + \sum_{k=0}^{N-1} \lambda_{k+1}^T (f(x_k, u_k) - x_{k+1}) \\
& \quad +\lambda_r^T r(x_0, x_N), \qquad \text{(5.13)}
\end{aligned}
$$

and the summarized KKT-conditions of the problem are

$$
\begin{aligned}
\nabla_w \mathcal{L}(w, \lambda) \ &= \ 0 && \text{(5.14a)}\\
G(w) \ &= \ 0. && \text{(5.14b)}
\end{aligned}
$$

But let us look at these KKT-conditions in more detail. First, we evaluate the derivative of $\mathcal{L}$ with respect to all state variables x_k, one after the other. We have to treat $k = 0$ and $k = N$ as special cases. For $k = 0$ we obtain:

$$
\nabla_{x_0} \mathcal{L}(w, \lambda) \ = \ \nabla_{x_0} L(x_0, u_0) + \frac{\partial f}{\partial x_0}(x_0, u_0)^T \lambda_1 + \frac{\partial r}{\partial x_0}(x_0, x_N)^T \lambda_r = 0. \qquad \text{(5.15a)}
$$

Then the case for $k = 1, \ldots, N-1$ is treated

$$
\nabla_{x_k} \mathcal{L}(w, \lambda) \ = \ \nabla_{x_k} L(x_k, u_k) - \lambda_k + \frac{\partial f}{\partial x_k}(x_k, u_k)^T \lambda_{k+1} = 0. \qquad \text{(5.15b)}
$$

Last, the special case $k = N$

$$\nabla_{x_N}\mathcal{L}(w,\lambda) \quad = \quad \nabla_{x_N}E(x_N) - \lambda_N + \frac{\partial r}{\partial x_N}(x_0,x_N)^T\lambda_r = 0. \tag{5.15c}$$

Second, let us calculate the derivative of the Lagrangian with respect to all controls u_k, for $k = 0,\dots,N-1$. Here, no special cases need to be considered, and we obtain the general formula

$$\nabla_{u_k}\mathcal{L}(w,\lambda) \quad = \quad \nabla_{u_k}L(x_k,u_k) + \frac{\partial f}{\partial u_k}(x_k,u_k)^T\lambda_{k+1} = 0. \tag{5.15d}$$

Until now, we have computed in detail the components of the first part of the KKT-condition (5.14a), i.e. $\nabla_w\mathcal{L}(w,\lambda) = 0$. The other part of the KKT-condition, $G(w) = 0$, is trivially given by

$$\begin{aligned} f(x_k,u_k) - x_{k+1} &= 0, \quad k = 0,\dots,N-1 & (5.15e)\\ r(x_0,x_N) &= 0 & (5.15f)\end{aligned}$$

Thus, collecting all equations (5.15a) to (5.15f), we have stated the KKT-conditions of the OCP. They can be treated by Newton-type methods in different ways. The *simultaneous approach* addresses equations (5.15a) to (5.15f) directly by a Newton-type method in the space of all variables (w,λ). In contrast to this, the *sequential approach* approach eliminates all the states $x_1,\dots,x_N$ in (5.15e) by a forward simulation, and if it is implemented efficiently, it also uses Eqs. (5.15c) and (5.15b) to eliminate all multipliers $\lambda_N,\dots,\lambda_1$ in a backward simulation, as discussed in the following subsection.

5.2.2 Computing Gradients in the Sequential Approach

A naive implementation of the sequential approach would start by coding routines that evaluate the objective and constraint functions, and then passing these routines as black-box codes to a generic NLP solver, like `fmincon` in MATLAB. But this would not be the most efficient way to implement the sequential approach. The reason is the generation of derivatives, which a generic NLP solver will compute by finite differences. On the other hand, many generic NLP solvers allow the user to deliver explicit functions for the derivatives as well. This allows us to compute the derivatives of the reduced problem functions more efficiently. The key technology here is algorithmic differentiation in the backward mode, as explained in Chapter 4.

To see how this relates to the optimality conditions (5.15a) to (5.15f) of the optimal control problem, let us simplify the setting even more by assuming a fixed initial value and no constraint on the terminal state, i.e. $r(x_0,x_N) = \bar{x}_0 - x_0$. In this case, the KKT conditions simplify to the following set of equations, which we bring already into a specific order:

$$\begin{aligned} x_0 &= \bar{x}_0 & (5.16a)\\ x_{k+1} &= f(x_k,u_k), \quad k = 0,\dots,N-1, & (5.16b)\\ \lambda_N &= \nabla_{x_N}E(x_N) & (5.16c)\\ \lambda_k &= \nabla_{x_k}L(x_k,u_k) + \frac{\partial f}{\partial x_k}(x_k,u_k)^T\lambda_{k+1}, & \\ &\qquad k = N-1,\dots,1, & (5.16d)\\ \nabla_{u_k}L(x_k,u_k) + \frac{\partial f}{\partial u_k}(x_k,u_k)^T\lambda_{k+1} &= 0, \quad k = 0,\dots,N-1. & (5.16e)\end{aligned}$$

It can easily be seen that the first four equations can trivially be satisfied, by a forward sweep to obtain all x_k and a backward sweep to obtain all λ_k. Thus, x_k and λ_k can be made explicit functions of $u_0, \ldots, u_{N-1}$. The only equation that is non-trivial to satisfy is the last one, the partial derivatives of the Lagrangian w.r.t. the controls $u_0, \ldots, u_{N-1}$. Thus we could decide to eliminate x_k and λ_k and only search with a Newton-type scheme for the variables $U = (u_0, \ldots, u_{N-1})$ such that these last equations are satisfied. It turns out that the left hand side residuals (5.16e) are nothing else than the derivative of the reduced problem's objective (5.10a), and the forward-backward sweep algorithm described above is nothing else than the reverse mode of algorithmic differentiation. It is much more efficient than the computation of the gradient by finite differences.

The forward-backward sweep is well known in the optimal control literature and often introduced without reference to the reverse mode of AD. On the other hand, it is good to know the general principles of AD in forward or backward mode, because AD can also be beneficial in other contexts, e.g. for the evaluation of derivatives of the other problem functions in (5.10a)-(5.10c). Also, when second order derivatives are needed, AD can be used and more structure can be exploited, but this is most easily derived in the context of the simultaneous approach, which we do in the following chapter.

Chapter 6

Sparsity Structure of the Optimal Control Problem

Let us in this chapter regard a very general optimal control problem in the original formulation, i.e. the NLP that would be treated by the simultaneous approach.

$$
\begin{aligned}
\underset{x_0, u_0, x_1, \ldots, u_{N-1}, x_N}{\text{minimize}} \quad & \sum_{k=0}^{N-1} L_k(x_k, u_k) + E(x_N) && \text{(6.1a)} \\
\text{subject to} \quad & f_k(x_k, u_k) - x_{k+1} = 0, \quad \text{for} \quad k = 0, \ldots, N-1, && \text{(6.1b)} \\
& \sum_{k=0}^{N-1} r_k(x_k, u_k) + r_N(x_N) = 0, && \text{(6.1c)} \\
& h_k(x_k, u_k) \leq 0, \quad \text{for} \quad k = 0, \ldots, N-1, && \text{(6.1d)} \\
& h_N(x_N) \leq 0. && \text{(6.1e)}
\end{aligned}
$$

Compared to the OCP (5.7) in the previous chapter, we now allow indices on all problem functions making the system time dependent; also, we added terminal inequality constraints (6.1e), and as boundary conditions we now allow now very general coupled multipoint constraints (6.1c) that include the cases of fixed initial or terminal values or periodicity, but are much more general. Note that in these boundary constraints terms arising from different time points are only coupled by addition, because this allows us to maintain the sparsity structure we want to exploit in this chapter.

Collecting all variables in a vector w, the objective in a function $F(w)$, all equalities in a function $G(w)$ and all inequalities in a function $H(w)$, the optimal control problem could be summarized as

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & F(w) && && \text{(6.2a)} \\ \text{subject to} \quad & G(w) &&= 0, && \text{(6.2b)} \\ & H(w) &&\leq 0. && \text{(6.2c)} \end{aligned}$$

Its Lagrangian function is given by

$$\mathcal{L}(w, \lambda, \mu) = F(w) + \lambda^T G(w) + \mu^T H(w).$$

But this summarized form does not reveal any of the structure that is present in the problem.

6.1 Partial Separability of the Lagrangian

In fact, the above optimal control problem is a very sparse problem because each of its functions depends only on very few of its variables. This means for example that the Jacobian matrix of the equality constraints has many zero entries. But not only first order derivatives are sparse, also the second order derivative that we need in Newton-type optimization algorithms, namely the Hessian of the Lagrangian, is a very sparse matrix. This is due to the fact that the Lagrangian is a *partially separable* function [44].

Definition 6.1 (Partial Separability)
A function $f : \mathbb{R}^n \to \mathbb{R}$ is called partially separable if it can be decomposed as a sum of m functions $f_j : \mathbb{R}^{n_j} \to \mathbb{R}$ with $n_j < n$ for all $j = 1, \ldots, m$. This means that for each j exists a subset I_j of indices from $\{1, \ldots, n\}$ and subvectors x_{I_j} of x such that

$$f(x) = \sum_{j=1}^{m} f_j(x_{I_j}).$$

The Lagrangian function of the above optimization problem can explicitly be decomposed into subfunctions that each depend on some of the multipliers and only on the variables (x_k, u_k) with the same index k. Let us collect again all variables in a vector w but decompose it as[1] $w = (w_1, \ldots, w_N)$ with $w_k = (x_k, u_k)$ for $k = 0, \ldots, N-1$ and $w_N = x_N$. Collecting all equality multipliers in a vector $\lambda = (\lambda_1, \ldots, \lambda_N, \lambda_r)$ and the inequality multipliers in a vector $\mu = (\mu_0, \ldots, \mu_N)$ we obtain for the Lagrangian

$$\mathcal{L}(w, \lambda, \mu) = \sum_{k=0}^{N} \mathcal{L}_k(w_k, \lambda, \mu)$$

[1]Note that for notational beauty we omit here and in many other occasions the transpose signs that would be necessary to make sure that the collection of column vectors is again a column vector, when this is clear from the context.

with the local Lagrangian subfunctions defined as follows. The first subfunction is given as

$$\mathcal{L}_0(w_0,\lambda,\mu) = L_0(x_0,u_0) + \lambda_1^T f_0(x_0,u_0) + \mu_0^T h_0(x_0,u_0) + \lambda_r^T r_0(x_0,u_0)$$

and for $k = 1,\dots,N-1$ we have the subfunctions

$$\mathcal{L}_k(w_k,\lambda,\mu) = L_k(x_k,u_k) + \lambda_{k+1}^T f_k(x_k,u_k) - \lambda_k^T x_k + \mu_k^T h_k(x_k,u_k) + \lambda_r^T r_k(x_k,u_k)$$

while the last subfunction is given as

$$\mathcal{L}_N(w_N,\lambda,\mu) = E(x_N) - \lambda_N^T x_N + \mu_N^T h_N(x_N) + \lambda_r^T r_N(x_N).$$

In fact, while each of the equality multipliers appears in several $(\lambda_1,\dots,\lambda_N)$ or even all problem functions (λ_r), the primal variables of the problem do not have any overlap in the subfunctions. This leads to the remarkable observation that the Hessian matrix $\nabla_w^2\mathcal{L}$ is *block diagonal*, i.e. it consists only of small symmetric matrices that are located on its diagonal. All other second derivatives are zero, i.e.

$$\frac{\partial^2\mathcal{L}}{\partial w_i \partial w_j}(w,\lambda,\mu) = 0, \quad \text{for any} \quad i \neq j.$$

This block diagonality of the Hessian leads to several very favourable facts, namely that (i) the Hessian can be approximated by *high-rank* or *block updates* within a BFGS method [44, 21], and (ii) that the QP subproblem in all Newton-type methods has the same decomposable objective function as the original optimal control problem itself.

6.2 The Sparse QP Subproblem

In order to analyse the sparsity structure of the optimal control problem, let us regard the quadratic subproblem that needs to be solved in one iteration of an exact Hessian SQP method. In order not to get lost in too many indices, we disregard the SQP iteration index completely. We regard the QP that is formulated at a current iterate (x,λ,μ) and use the SQP step $\Delta w = (\Delta x_0, \Delta u_0, \dots, \Delta x_N)$ as the QP variable. This means that in the summarized formulation we would have the QP subproblem

$$\underset{\Delta w}{\text{minimize}} \quad \nabla F(w)^T \Delta w + \frac{1}{2}\Delta w^T \, \nabla_w^2 \, \mathcal{L}(w,\lambda,\mu)\Delta w \tag{6.3a}$$

$$\text{subject to} \quad G(w) + \nabla G(w)^T \Delta w = 0, \tag{6.3b}$$

$$H(w) + \nabla H(w)^T \Delta w \leq 0. \tag{6.3c}$$

Let us now look at this QP subproblem in the detailed formulation. It is remarkably similar to the original OCP. To reduce notational overhead, let us define a few abbreviations: first, the diagonal blocks of the Hessian of the Lagrangian

$$Q_k = \nabla_{w_k}^2 \mathcal{L}(w,\lambda,\mu), \quad k = 0,\dots,N,$$

second, the objective gradients

$$g_k = \nabla_{(x,u)} L(x_k,u_k), \quad k = 0,\dots,N-1, \quad \text{and} \quad g_N = \nabla_x E(x_N),$$

third the system discontinuities (that can be non-zero in the simultaneous approach)

$$a_k = f_k(x_k, u_k) - x_{k+1}, \quad k = 0, \ldots, N-1,$$

and fourth the transition matrices

$$A_k = \frac{\partial f_k}{\partial x_k}(x_k, u_k), \quad B_k = \frac{\partial B_k}{\partial u_k}(x_k, u_k), \quad k = 0, \ldots, N-1,$$

fifth the residual of the coupled constraints

$$r = \sum_{k=0}^{N-1} r_k(x_k, u_k) \qquad + \quad r_N(x_N),$$

as well as its derivatives

$$R_k = \frac{\partial r_k}{\partial (x_k, u_k)}(x_k, u_k), \quad k = 0, \ldots, N-1, \quad \text{and} \quad R_N = \frac{\partial r_N}{\partial x}(x_N),$$

and last the inequality constraint residuals and their derivatives

$$h_k = h_k(x_k, u_k), \quad H_k = \frac{\partial h_k}{\partial (x_k, u_k)}(x_k, u_k) \quad \text{and} \quad h_N = h_N(x_N), \quad H_N = \frac{\partial h_N}{\partial x}(x_N).$$

With all the above abbreviations, the detailed form of the QP subproblem is finally given as follows.

$$\underset{\Delta x_0, \Delta u_0, \ldots, \Delta x_N}{\text{minimize}} \quad \frac{1}{2}\sum_{k=0}^{N-1} \begin{bmatrix} \Delta x_k \\ \Delta u_k \end{bmatrix}^T Q_k \begin{bmatrix} \Delta x_k \\ \Delta u_k \end{bmatrix} + \frac{1}{2}\Delta x_N^T Q_N \Delta x_N + \sum_{k=0}^{N} \begin{bmatrix} \Delta x_N \\ \Delta u_N \end{bmatrix}^T g_k + \Delta x_N^T g_N \tag{6.4}$$

$$\begin{aligned} \text{subject to} \quad a_k + A_k \Delta x_k + B_k \Delta u_k - \Delta x_{k+1} &= 0, \quad \text{for} \quad k = 0, \ldots, N-1, && (6.5) \\ r + \sum_{k=0}^{N-1} R_k \begin{bmatrix} \Delta x_k \\ \Delta u_k \end{bmatrix} + R_N \Delta x_N &= 0, \\ h_k + H_k \begin{bmatrix} \Delta x_k \\ \Delta u_k \end{bmatrix} &\leq 0, \quad \text{for} \quad k = 0, \ldots, N-1, \\ h_N + H_N \Delta x_N &\leq 0. && (6.6) \end{aligned}$$

This is again an optimal control problem, but a linear-quadratic one. It is a convex QP if the Hessian blocks Q_k are positive definite, and can be solved by a variety of sparsity exploiting QP solvers.

6.3 Sparsity Exploitation in QP Solvers

When regarding the QP (6.4) one way would be to apply a sparse interior point QP solver like OOQP to it, or a sparse active set method. This can be very efficient. Another way would be to first reduce, or *condense*, the variable space of the QP, and then apply a standard dense QP solver to the reduced problem. Let us treat this way first.

6.3.1 Condensing

When we regard the linearized dynamic system equations (6.5) they correspond to an affine time variant system in the steps Δx_k, namely

$$\Delta x_{k+1} = a_k + A_k \Delta x_k + B_k \Delta u_k. \tag{6.7}$$

If the values for Δx_0 as well as for all $\{\Delta u_k\}_{k=0}^{N-1}$ would be known, then also the values for $\{\Delta x_k\}_{k=1}^{N}$ can be obtained by a forward simulation of this linear system. Due to its linearity, the resulting map will be linear, i.e. we can write

$$\begin{aligned}
\begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_N \end{bmatrix} &= v \quad + \quad M \begin{bmatrix} \Delta x_0 \\ \Delta u_0 \\ \vdots \\ \Delta u_{N-1} \end{bmatrix}, && (6.8) \\
&\Leftrightarrow \\
\Delta w_{\text{dep}} &= v + M \Delta w_{\text{ind}} && (6.9)
\end{aligned}$$

with a vector $v \in \mathbb{R}^{N \cdot n_x}$ and a matrix $M \in \mathbb{R}^{(N \cdot n_x) \times (n_x + N \cdot n_u)}$, and dividing the variables into a dependent and an independent part, $\Delta w = (\Delta w_{\text{dep}}, \Delta w_{\text{ind}})$.

The vector v can be generated recursively by simulating the affine dynamic system (6.7) with all inputs set to zero, i.e. $\Delta w_{\text{ind}} = 0$. This yields the forward recursion

$$v_1 = a_0, \quad v_{k+1} = a_k + A_k v_k, \quad k = 1, \dots, N-1$$

for the components of the vector $v = (v_1, \dots, v_N)$. The subblocks of the matrix M can be obtained recursively as well in a straightforward way. Note that the matrix is lower triangular because the states Δx_j do not depend on Δu_k if $k \geq j$. On the other hand, if $k < j$, the corresponding matrix blocks are $A_{j-1} \cdots A_{k+1} B_k$. Finally, the dependence of Δx_j on Δx_0 is $A_{j-1} \cdots A_0$. In this way, all blocks of the matrix M are defined.

To get a notationally different, but equivalent view on condensing, note that the linear dynamic system equations (6.5) are nothing else than the linear system

$$\begin{bmatrix} A_0 & B_0 & -\mathbb{I} & & & & & & \\ & & A_1 & B_1 & -\mathbb{I} & & & & \\ & & & & \ddots & & & & \\ & & & & & & A_{N-1} & B_{N-1} & -\mathbb{I} \end{bmatrix} \begin{bmatrix} \Delta x_0 \\ \Delta u_0 \\ \Delta x_1 \\ \Delta u_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_{N-1} \\ \Delta u_{N-1} \\ \Delta x_N \end{bmatrix} = - \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{bmatrix} \tag{6.10}$$

After reordering the variables into dependent and independent ones, this system can be written as

$$\left[\begin{array}{cccc|cccc} A_0 & B_0 & & & -\mathbb{I} & & & \\ & B_1 & & & A_1 & -\mathbb{I} & & \\ & & \ddots & & & \ddots & \ddots & \\ & & & B_{N-1} & & & A_{N-1} & -\mathbb{I} \end{array}\right] \left[\begin{array}{c} \Delta x_0 \\ \Delta u_0 \\ \vdots \\ \Delta u_{N-1} \\ \hline \Delta x_1 \\ \vdots \\ \Delta x_N \end{array}\right] = - \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{bmatrix} \tag{6.11}$$

which we can summarize as

$$[X|Y] \begin{bmatrix} \Delta w_{\text{ind}} \\ \Delta w_{\text{dep}} \end{bmatrix} = -a$$

so that we get the explicit solution

$$\Delta w_{\text{dep}} = \underbrace{(-Y^{-1}a)}_{=v} + \underbrace{(-Y^{-1}X)}_{=M} \Delta w_{\text{ind}}.$$

Note that the submatrix Y is always invertible due the fact that it is lower triangular and has (negative) unit matrices on its diagonal.

Once the vector v and matrix M are computed, we can formulate a *condensed QP* which has only the independent variables Δw_{ind} as degrees of freedom. This condensed QP can be solved by a dense QP solver, and the resulting solution Δw^*_{ind} can be expanded again to yield also the QP solution for $w^*_{\text{dep}} = v + M\Delta w^*_{\text{ind}}$. The QP multipliers $\lambda_{\text{dep}} = (\lambda_1, \ldots, \lambda_N)$ for the constraints (6.5) can be obtained from the dense QP solution in a slightly more complex way. The trick is to regard the Lagrangian of the original QP (6.5), $\mathcal{L}^{\text{QP}}(\Delta w_{\text{ind}}, \Delta w_{\text{dep}}, \lambda_{\text{dep}}, \lambda_r, \mu)$ and note that the condensed QP yields also the multipliers λ^*_r, μ^*, which turn out to be the correct multipliers also for the uncondensed QP. Thus, the only missing quantity is λ^*_{dep}. It can be obtained by using the follwing two observations: first, for the true QP solution must hold that the Lagrange gradient is zero, also with respect to Δw_{dep}. Second, this Lagrange gradient depends linearly on the unknown multipliers λ_{dep} which contribute to it via the term $Y^T \lambda_{\text{dep}}$, i.e. we have

$$0 = \nabla_{\Delta w_{\text{dep}}} \mathcal{L}^{\text{QP}}(\Delta w^*_{\text{ind}}, \Delta w^*_{\text{dep}}, \lambda^*_{\text{dep}}, \lambda^*_r, \mu^*) = \nabla_{\Delta w_{\text{dep}}} \mathcal{L}^{\text{QP}}(\Delta w^*_{\text{ind}}, \Delta w^*_{\text{dep}}, 0, \lambda^*_r, \mu^*) + Y^T \lambda^*_{\text{dep}}.$$

It is a favourable fact that the Lagrange gradient depends on the missing multipliers via the matrix Y^T, because this matrix is invertible. Thus, we obtain an explicit equation for obtaining the missing multipliers, namely

$$\lambda^*_{\text{dep}} = -Y^{-T} \nabla_{\Delta w_{\text{dep}}} \mathcal{L}^{\text{QP}}(\Delta w^*_{\text{ind}}, \Delta w^*_{\text{dep}}, 0, \lambda^*_r, \mu^*).$$

Note that the multipliers would not be needed within a Gauss-Newton method.

Summarizing, condensing reduces the original QP to a QP that has the size of the QP in the sequential approach. Nearly all sparsity is lost, but the dimension of the QP is much reduced. Condensing is favourable if the horizon length N and the control dimension n_u are relatively small compared to the state dimension n_x. If the initial value is fixed, then also Δx_0 can be eliminated from the condensed QP before passing it to a dense QP solver, further reducing the dimension.

On the other hand, if the state dimension n_x is very small compared to $N \cdot n_u$, condensing is not favourable due to the fact that it destroys sparsity. This is most easily seen in the Hessian. In the original sparse QP, the block sparse Hessian has $N(n_x + n_u)^2 + n_x^2$ nonzero elements. This is linear in N. In contrast to this, the condensed Hessian is dense and has $(n_x + Nn_u)^2$ elements, which is quadratic in N. Thus, if N is large, not only might the condensed Hessian need more (!) storage than the original one, also the solution time of the QP becomes cubic in N (factorization costs of the Hessian).

6.3.2 Sparse KKT System

A different way to exploit the sparsity present in the QP (6.4) is to keep all variables in the problem and use within the QP solver linear algebra routines that exploit sparsity of matrices. This can be realized within both, interior point (IP) methods as well as in active set methods, but is much easier to illustrate at the example of IP methods. For illustration, let us assume a problem without coupled constraints (6.6) and assume that all inequalities have been transformed into primal barrier terms that are added to the objective. Then, in each interior point iteration, an equality constrained QP of the following simple form needs to be solved.

$$\underset{\Delta x_0, \Delta u_0, \dots, \Delta x_N}{\text{minimize}} \quad \frac{1}{2}\sum_{k=0}^{N-1} \begin{bmatrix} \Delta x_k \\ \Delta u_k \end{bmatrix}^T \begin{bmatrix} Q_k^x & Q_k^{xu} \\ (Q_k^{xu})^T & Q_k^u \end{bmatrix} \begin{bmatrix} \Delta x_k \\ \Delta u_k \end{bmatrix} + \frac{1}{2}\Delta x_N^T Q_N \Delta x_N + \sum_{k=0}^{N} \begin{bmatrix} \Delta x_N \\ \Delta u_N \end{bmatrix}^T g_k + \Delta x_N^T g_N \tag{6.12}$$

$$\text{subject to} \quad a_k + A_k \Delta x_k + B_k \Delta u_k - \Delta x_{k+1} = 0, \quad \text{for} \quad k = 0, \dots, N \tag{6.13}$$

Formulating the Lagrangian of this QP and differentiating it with respect to all its primal and dual variables $y = (\Delta x_0, \Delta u_0, \lambda_1, \Delta x_1, \Delta u_1, \dots \lambda_N, \Delta x_N)$ in this order we obtain a linear system of the following block tridiagonal form

$$\begin{bmatrix}
Q_0^x & Q_0^{xu} & A_0^T & & & & & & & & & \\
(Q_0^{xu})^T & Q_0^u & B_0^T & & & & & & & & & \\
A_0 & B_0 & 0 & -\mathbb{I} & & & & & & & & \\
 & & -\mathbb{I} & Q_1^x & Q_1^{xu} & A_1^T & & & & & & \\
 & & & (Q_1^{xu})^T & Q_1^u & B_1^T & & & & & & \\
 & & & A_1 & B_1 & 0 & -\mathbb{I} & & & & & \\
 & & & & & -\mathbb{I} & & & & & & \\
 & & & & & & \ddots & \ddots & & & & \\
 & & & & & & & & A_{N-1} & B_{N-1} & 0 & -\mathbb{I} \\
 & & & & & & & & & & -\mathbb{I} & Q_N
\end{bmatrix}
\begin{bmatrix} \Delta x_0 \\ \Delta u_0 \\ \lambda_1 \\ \Delta x_1 \\ \Delta u_1 \\ \lambda_2 \\ \vdots \\ \lambda_N \\ \Delta x_N \end{bmatrix}
=
\begin{bmatrix} * \\ * \\ * \\ * \\ * \\ * \\ * \\ * \\ * \end{bmatrix} \tag{6.14}$$

This linear system can be solved with a banded direct factorization routine, whose runtime is proportional to $N(n_x + n_u)^3$. We will see in the next chapter that a particularly efficient way to solve the above linear system can be obtained by applying the principle of dynamic programming to the equality constrained quadratic subproblem (6.12).

Summarizing, the approach to directly solve the sparse QP without condensing is advantageous if Nn_u is large compared to n_x. It needs, however, sparse linear algebra routines within the QP solver. This is easier to implement in the case of IP methods than for active set methods.

Chapter 7

Dynamic Programming

Dynamic programming (DP) is a very different approach to solve optimal control problems than the ones presented previously. The methodology was developed in the fifties and sixties of the 19th century, most prominently by Richard Bellman [5] who also coined the term dynamic programming. Interestingly, dynamic programming is easiest to apply to systems with discrete state and control spaces, so that we will introduce this case first. When DP is applied to discrete time systems with continuous state spaces, some approximations have to be made, usually by discretization. Generally, this discretization leads to exponential growth of computational cost with respect to the dimension n_x of the state space, what Bellman called the "curse of dimensionality". It is the only but major drawback of DP and limits its practical applicability to systems with $n_x \approx 6$. In the continuous time case, DP is formulated as a partial differential equation in the state space, the Hamilton-Jacobi-Bellman (HJB) equation, suffering from the same limitation; but this will be treated in Chapter 9. On the positive side, DP can easily deal with all kinds of hybrid systems or non-differentiable dynamics, and it even allows us to treat stochastic optimal control with recourse, or minimax games, without much additional effort. An excellent textbook on discrete time optimal control and dynamic programming is [9]. Let us now start with discrete control and state spaces.

7.1 Dynamic Programming in Discrete State Space

Let us regard a dynamic system

$$x_{k+1} = f(x_k, u_k)$$

with $f : \mathbb{X} \times \mathbb{U} \to \mathbb{X}$, i.e. $x_k \in \mathbb{X}$ and $u_k \in \mathbb{U}$, where we do not have to specify the sets $\mathbb{X}$ and $\mathbb{U}$ yet. We note, however, that we need to assume they are finite for a practical implementation of DP. Thus, let us in this section assume they are finite with $n_{\mathbb{X}}$ and $n_{\mathbb{U}}$ elements, respectively. Let us also define a stage cost $L(x, u)$ and terminal cost $E(x)$ that take values from $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$, where infinity denotes infeasible pairs (x, u) or x. The optimal control problem that we first address can be stated as

$$\begin{aligned}
\underset{x_0, u_0, x_1, \ldots, u_{N-1}, x_N}{\text{minimize}} \quad & \sum_{k=0}^{N-1} L(x_k, u_k) \; + \; E(x_N) && (7.1a)\\
\text{subject to} \quad & f(x_k, u_k) - x_{k+1} \; = \; 0, \quad \text{for} \quad k = 0, \ldots, N-1, && (7.1b)\\
& \bar{x}_0 - x_0 \; = \; 0. && (7.1c)
\end{aligned}$$

Given the fact that the initial value is fixed and the controls $\{u_k\}_{k=0}^{N-1}$ are the only true degrees of freedom, and given that each $u_k \in \mathbb{U}$ takes one of the $n_{\mathbb{U}}$ elements of $\mathbb{U}$, there exist exactly $n_{\mathbb{U}}^N$ different trajectories, each with a specific value of the objective function, where infinity denotes an infeasible trajectory. Assuming that the evaluation of f and of L takes one computational unit, and noting that each trajectory needs N such evaluations, the overall complexity of simple enumeration is $O(Nn_{\mathbb{U}}^N)$. Simple enumeration of all possible trajectories thus has a complexity that grows exponentially with the horizon length N.

Dynamic programming is just a more intelligent way to enumerate all possible trajectories. It starts from the *principle of optimality*, i.e. the observation that each subtrajectory of an optimal trajectory is an optimal trajectory as well. More specifically, in DP we define the *value function* or *cost-to-go function* as the optimal cost that would be obtained if at time $k \in \{0, \ldots, N\}$ and at state $\bar{x}_k$ we solve the optimal control problem on a shortened horizon:

$$\begin{aligned}
J_k(\bar{x}_k) \quad = \quad \underset{x_k, u_k, \ldots, u_{N-1}, x_N}{\min} \quad & \sum_{i=k}^{N-1} L(x_i, u_i) \; + \; E(x_N) && (7.2a)\\
\text{subject to} \quad & f(x_i, u_i) - x_{i+1} \; = \; 0, \quad \text{for} \quad i = k, \ldots, N-1, && (7.2b)\\
& \bar{x}_k - x_k \; = \; 0. && (7.2c)
\end{aligned}$$

Thus, each function $J_k : \mathbb{X} \to \mathbb{R}_\infty$ summarizes the cost-to-go to the end when starting at a given state. For the case $k = N$ we trivially have $J_N(x) = E(x)$. The principle of optimality states now that for any $k \in \{0, \ldots, N-1\}$ holds

$$J_k(\bar{x}_k) = \min_u L(\bar{x}_k, u) + J_{k+1}(f(\bar{x}_k, u)). \tag{7.3}$$

This immediately allows us to perform a recursion to compute all functions J_k one after the other, starting with $k = N-1$ and then reducing k in each recursion step by one, until we have obtained J_0. This recursion is called the *dynamic programming recursion.* Once all the value functions J_k are computed, the *optimal feedback control* for a given state x_k at time k is given by

$$u_k^*(x_k) = \arg\min_u L(x_k, u) + J_{k+1}(f(x_k, u))$$

This allows us to reconstruct the optimal trajectory by a forward simulation that starts at $x_0 = \bar{x}_0$ and then proceeds as follows:

$$x_{k+1} = f(x_k, u_k^*(x_k)), \quad k = 0, \ldots, N-1.$$

In this way, DP allows us to solve the optimal control problem up to global optimality, but with a different complexity than simple enumeration. To assess its complexity, let us remark that the most cost intensive step is the generation of the N cost-to-go functions J_k. Each recursion step (7.3) needs to go through all $n_{\mathbb{X}}$ states x. For each state it needs to test $n_{\mathbb{U}}$ controls u by evaluating once the system $f(x,u)$ and stage cost $L(x,u)$, which by definition costs one computational unit. Thus, the overall computational complexity is $O(Nn_{\mathbb{X}}n_{\mathbb{U}})$. Compared with simple enumeration, where we had $O(Nn_{\mathbb{U}}^N)$, DP is often much better even for moderately sized horizons N. Let us for example assume an optimal control problem with $n_{\mathbb{U}} = 10$, $n_{\mathbb{X}} = 1000$, $N = 100$. Then simple enumeration has a cost of 10^{102} while DP has a cost of 10^6.

One of the main advantages of dynamic programming, that can likewise be defined for continuous state spaces, is that we do not need to make any assumptions (such as differentiability or convexity) on the functions f, L, E defining the problem, and still it solves the problem up to global optimality. On the other hand, if it shall be applied to a continuous state space, we have to represent the functions J_k on the computer, e.g. by tabulation on a grid in state space. If the continuous state space $\mathbb{X}_{\text{cont}}$ is a box in dimension n_x, and if we use a rectangular grid with m intervals in each dimension, then the total number of grid points is m^{n_x}. If we perform DP on this grid, then the above complexity estimate is still valid, but with $n_{\mathbb{X}} = m^{n_x}$. Thus, when DP is applied to systems with continuous state spaces, it has exponential complexity in the dimension of the state space; it suffers from what Bellman called the *curse of dimensionality*. There exist many ways to approximate the value function, e.g. by neural networks or other functional representations [11], but the global optimality guarantee of dynamic programming is lost in these cases. On the other hand, there exists one special case where DP can be performed exactly in continuous state spaces, that we treat next.

7.2 Linear Quadratic Problems

Let us regard now linear quadratic optimal control problems of the form

$$\underset{x,\,u}{\text{minimize}} \quad \sum_{i=0}^{N-1} \begin{bmatrix} x_i \\ u_i \end{bmatrix}^T \begin{bmatrix} Q_i & S_i^T \\ S_i & R_i \end{bmatrix} \begin{bmatrix} x_i \\ u_i \end{bmatrix} \; + \; x_N^T P_N x_N \tag{7.4}$$

$$\begin{aligned} \text{subject to} \quad & \\ x_0 - \bar{x}_0 &= 0, \\ x_{i+1} - A_i x_i - B_i u_i &= 0, \quad i = 0, \ldots, N-1. \end{aligned}$$

Let us apply dynamic programming to this case. In each recursion step, we have to solve, for a time varying stage cost $L_k(x,u) = \begin{bmatrix} x_k \\ u_k \end{bmatrix}^T \begin{bmatrix} Q_k & S_k^T \\ S_k & R_k \end{bmatrix} \begin{bmatrix} x_k \\ u_k \end{bmatrix}$ and a dynamic system $f_k(x,u) = A_k x + B_k u$ the recursion step

$$J_k(x) = \min_u L_k(x,u) + J_{k+1}(f_k(x,u)),$$

where we start with $J_N(x) = x^T P_N x$. Fortunately, it can be shown that under these circumstances, each J_k is quadratic, i.e. it again has the form $J_k(x) = x^T P_k x$. More specifically, the following theorem holds, where we drop the index k for simplicity.

Theorem 7.1 (Quadratic Representation of Value Function)**:** *If $R + B^T P B$ is positive definite,* *then the minimum $J_{\text{new}}(x)$ of one step of the DP recursion*

$$J_{\text{new}}(x) = \min_u \quad \begin{bmatrix} x \\ u \end{bmatrix}^T \left(\begin{bmatrix} Q & S^T \\ S & R \end{bmatrix} + [A \,|\, B]^T P \, [A \,|\, B] \right) \begin{bmatrix} x \\ u \end{bmatrix}$$

is a quadratic function explicity given by $J_{\text{new}}(x) = x^T P_{\text{new}} \, x$ with

$$P_{\text{new}} = Q + A^T PA - (S^T + A^T PB)(R + B^T PB)^{-1}(S + B^T PA). \tag{7.5}$$

The proof starts by noting that the optimization problem for a specific x is given by

$$J_{\text{new}}(x) = \min_u \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} Q + A^T PA & S^T + A^T PB \\ S + B^T PA & R + B^T PB \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix}.$$

Then it uses the fact that for invertible $\bar{R} = R + B^T PB$ this problem can be solved explicitly, yielding the formula (7.5), by a direct application of the *Schur complement lemma*, that can easily be verified by direct calculation.

Lemma 7.2 (Schur Complement Lemma)**:** *If $\bar{R}$ is positive definite then*

$$\min_u \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} \bar{Q} & \bar{S}^T \\ \bar{S} & \bar{R} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} = x^T \left(\bar{Q} - \bar{S}^T \bar{R}^{-1} \bar{S} \right) x \tag{7.6}$$

and the minimizer $u^*(x)$ is given by $u^*(x) = -\bar{R}^{-1}\bar{S}x$.

The above theorem allows us to solve the optimal control problem by first computing explicitly all matrices P_k, and then performing the forward closed loop simulation. More explicitly, starting with P_N, we iterate for $k = N-1, \ldots, 0$ backwards

$$P_k = Q_k + A_k^T P_{k+1} A_k - (S_k^T + A_k^T P_{k+1} B_k)(R_k + B_k^T P_{k+1} B_k)^{-1}(S_k + B_k^T P_{k+1} A_k). \tag{7.7}$$

This is sometimes called the *Difference Riccati Equation.* Then, we obtain the optimal feedback $u_k^*(x_k)$ by

$$u_k^*(x_k) = -(R_k + B_k^T P_{k+1} B_k)^{-1}(S_k + B_k^T P_{k+1} A_k) x_k,$$

and finally, starting with $x_0 = \bar{x}_0$ we perform the forward recursion

$$x_{k+1} = A_k x_k + B_k u_k^*(x_k),$$

which delivers the complete optimal trajectory of the linear quadratic optimal control problem.

An important and more general case are problems with linear quadratic costs and affine linear systems, i.e. problems of the form

$$\underset{x,\,u}{\text{minimize}} \quad \sum_{i=0}^{N-1} \begin{bmatrix} 1 \\ x_i \\ u_i \end{bmatrix}^T \begin{bmatrix} * & q_i^T & s_i^T \\ q_i & Q_i & S_i^T \\ s_i & S_i & R_i \end{bmatrix} \begin{bmatrix} 1 \\ x_i \\ u_i \end{bmatrix} \quad + \quad \begin{bmatrix} 1 \\ x_N \end{bmatrix}^T \begin{bmatrix} * & p_N^T \\ p_N & P_N \end{bmatrix} \begin{bmatrix} 1 \\ x_N \end{bmatrix} \tag{7.8}$$

subject to

$$\begin{aligned} x_0 - x_0^{\text{fix}} &= 0, \\ x_{i+1} - A_i x_i - B_i u_i - c_i &= 0, \quad i = 0, \dots, N-1. \end{aligned}$$

These optimization problems appear at many occasions, for example as linearizations of nonlinear optimal control problems, as in Chapter 6, in reference tracking problems with $L_i(x_i, u_i) = \|x_i - x_i^{\text{ref}}\|_Q^2 + \|u_i\|_R^2$, or in *moving horizon estimation* (MHE) with cost $L_i(x_i, u_i) = \|Cx_i - y_i^{\text{meas}}\|_Q^2 + \|u_i\|_R^2$. They can be treated by exactly the same recursion formulae as above, by augmenting the system states x_k to

$$\tilde{x}_k = \begin{bmatrix} 1 \\ x_k \end{bmatrix}$$

and replacing the dynamics by

$$\tilde{x}_{k+1} = \begin{bmatrix} 1 & 0 \\ c_k & A_k \end{bmatrix} \tilde{x}_k + \begin{bmatrix} 0 \\ B_k \end{bmatrix} u_k$$

with initial value

$$\tilde{x}_0^{\text{fix}} = \begin{bmatrix} 1 \\ x_0^{\text{fix}} \end{bmatrix}$$

Then the problem (7.8) can be reformulated in the form of problem (7.4) and can be solved using exactly the same difference Riccati equation formula as before!

7.3 Infinite Horizon Problems

Dynamic programming can easily be generalized to infinite horizon problems of the form

$$\underset{x,\,u}{\text{minimize}} \quad \sum_{i=0}^{\infty} L(x_i, u_i)$$

subject to

$$\begin{aligned} x_0 - \bar{x}_0 &= 0, \\ x_{i+1} - f(x_i, u_i) &= 0, \quad i = 0, \dots, \infty. \end{aligned}$$

Interestingly, the cost-to-go function $J_k(x_k)$ defined in Equation (7.2) becomes independent of the index k, i.e it holds that $J_k = J_{k+1}$ for all k. This directly leads to the *Bellman Equation*:

$$J(x) = \min_u \underbrace{L(x, u) + J(f(x, u))}_{=\tilde{J}(x,u)}$$

The optimal controls are obtained by the function

$$u^*(x) = \arg\min_u \tilde{J}(x, u).$$

This feedback is called the *stationary optimal feedback control.* It is a static state feedback law.

7.4 The Linear Quadratic Regulator

An important special case is again the case of a linear system with quadratic cost. It is the solution to an infinite horizon problem with a linear system $f(x,u) = Ax + Bu$ and quadratic cost

$$L(x,u) = \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} Q & S^T \\ S & R \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix}.$$

For its solution, we just require a stationary solution of the Riccati recursion (7.7), setting $P_k = P_{k+1}$, which yields the so called *algebraic Riccati equation in discrete time*

$$P = Q + A^T PA - (S^T + A^T PB)(R + B^T PB)^{-1}(S + B^T PA).$$

This is a nonlinear matrix equation in the symmetric matrix P, i.e. with $n_x(n_x + 1)/2$ unknowns. It can either be solved by an iterative application of the difference Riccati recursion (7.7) starting with e.g. a zero matrix $P = 0$, or by faster converging procedures such as Newton-type methods, where, however, care has to be taken to avoid possible shadow solutions that are not positive definite. Once the solution matrix P is found, the optimal feedback control $u^*(x)$ is given by

$$u^*(x) = -\underbrace{(R + B^T PB)^{-1}(S + B^T PA)}_{=K} x$$

This feedback is called the *Linear Quadratic Regulator (LQR)*, and K is the LQR gain.

7.5 Robust and Stochastic Dynamic Programming

One of its most interesting characteristics is that DP can easily be applied to games like chess, or to *robust optimal control problems*. Here, an adverse player choses counter-actions, or disturbances, w_k against us. They influence both the stage costs L_k as well as the system dynamics f_k and while we want to minimize, our adversary wants to maximize. The robust DP recursion for such a minimax game is simply:

$$J_k(x) = \min_u \underbrace{\max_w L_k(x,u,w) + J_{k+1}(f_k(x,u,w))}_{=\tilde{J}_k(x,u)}$$

starting with

$$J_N(x) = E(x).$$

The solution obtained by DP takes into account that we can react to the actions by the adversary, i.e. that we can apply feedback, and in the model predictive control (MPC) literature such a feedback law is sometimes called Closed-Loop Robust Optimal Control [7].

Alternatively, we might have a stochastic system and the aim is to find the feedback law that gives us the best expected value. Here, instead of the maximum, we take an *expectation* over the disturbances w_k. The stochastic DP recursion is simply given by

$$J_k(x) = \min_u \underbrace{\mathbb{E}_w\{L_k(x,u,w) + J_{k+1}(f_k(x,u,w))\}}_{=\tilde{J}_k(x,u)}$$

where $\mathbb{E}_w\{\cdot\}$ is the expectation operator, i.e. the integral over w weighted with the probability density function $\rho(w|x,u)$ of w given x and u:

$$\mathbb{E}_w\{\phi(x,u,w)\} = \int \phi(x,u,w)\rho(w|x,u)dw.$$

In case of finitely many disturbances, this is just a weighted sum. Note that DP avoids the combinatorial explosion of scenario trees that are often used in stochastic programming, but of course suffers from the curse of dimensionality. It is the preferred option for long horizon problems with small state spaces.

7.6 Interesting Properties of the DP Operator

Let us define the *dynamic programming operator* T_k acting on one value function, J_{k+1}, and giving another one, J_k, by

$$T_k[J](x) = \min_u L_k(x,u) + J(f_k(x,u)).$$

Note that the operator T_k maps from the space of functions $\mathbb{X} \to \mathbb{R}_\infty$ into itself. With this operator, the dynamic programming recursion is compactly written as $J_k = T_k[J_{k+1}]$, and the stationary Bellman equation would just be $J = T[J]$. Let us for notational simplicity drop the index k in the following. An interesting property of the DP operator T is its *monotonicity*, as follows.

Theorem 7.3 (Monotonicity of DP)**:** *Regard two value functions J and J'. If $J \geq J'$ in the sense that for all $x \in \mathbb{X}$ holds that $J(x) \geq J'(x)$ then also*

$$T[J] \geq T[J'].$$

The proof is

$$T[J](x) = \min_u L(x,u) + \underbrace{J(f(x,u))}_{\geq J'(f(x,u)))} \geq \min_u L(x,u) + J'(f(x,u)) = T[J'](x)$$

This monotonicity property holds also for robust or stochastic dynamic programming, and is for example used in existence proofs for solutions of the stationary Bellman equation, or in stability proofs of model predictive control (MPC) schemes [61].

Another interesting observation is that certain DP operators T preserve convexity of the value function J.

Theorem 7.4 (Convex dynamic programming)**:** *If the system is affine in (x,u), i.e. $f(x,u,w) = A(w)x + B(w)u + c(w)$, and if the stage cost $L(x,u,w)$ is convex in (x,u), then the DP, the robust DP, and the stochastic DP operators T preserve convexity of J, i.e. if J is a convex function, then $T[J]$ is again a convex function.*

It is interesting to note that no restrictions are given on how the functions depend on w. The proof of the convexity preservation starts by noting that for fixed w, $L(x, u, w) + J(f(x, u, w))$ is a convex function in (x, u). Because also the maximum over all w, or the positively weighted sum of an expectation value computation, preserve convexity, the function $\tilde{J}(x, u)$ is in all three cases convex in both x and u. Finally, the minimization of a convex function over one of its arguments preserves convexity, i.e. the resulting value function $T[J]$ defined by

$$T[J](x) = \min_u \tilde{J}(x, u)$$

is convex. □

But why would convexity be important in the context of DP? First, convexity of $\tilde{J}(x, u)$ implies that the computation of the feedback law $\arg\min_u \tilde{J}(x, u)$ is a convex parametric program and could reliably be solved by local optimization methods. Second, it might be possible to represent the value function $J(x)$ more efficiently than by tabulation on a grid, for example as the pointwise maximum of affine functions

$$J(x) = \max_i a_i^T \begin{bmatrix} 1 \\ x \end{bmatrix}.$$

It is an interesting fact that that for piecewise linear convex costs and constraints and polyhedral uncertainty this representation is exact and leads to an exact robust DP algorithm that might be called *polyhedral DP* [7, 31]. The polyhedral convex representability of the cost-to-go for linear systems with piecewise linear cost is indirectly exploited in some explicit MPC approaches [66, 6]. Polyhedral representations with a limited number of facets can also be used to approximate a convex cost-to-go and still yield some guarantees on the closed-loop system [16, 17, 50]. Finally, note that also the linear quadratic regulator is a special case of convex dynamic programming.

7.7 The Gradient of the Value Function

The meaning of the cost-to-go, or the value function, J_k is that it is the cost incurred on the remainder of the horizon for the best possible strategy. In order to make an interesting connection between the value function and the multipliers λ_k that we encountered in derivative based optimization methods, let us now regard a discrete time optimal control problem as in the previous chapters, but without coupled constraints, as these cannot directly be treated with dynamic programming. We assume further that the initial value is fixed and that all inequality and terminal constraints are subsumed in the stage cost $L(x, u)$ and terminal cost $E(x_N)$ by barrier functions that take infinite values outside the feasible domain but are differentiable inside. For terminal equality constraints, e.g. a fixed terminal state, assume for the moment that these are approximated by a terminal region of non-zero volume on which again a barrier can be defined. Thus, we regard the following problem.

$$\underset{x_0, u_0, x_1, \ldots, u_{N-1}, x_N}{\text{minimize}} \quad \sum_{k=0}^{N-1} L(x_k, u_k) \; + \; E(x_N) \tag{7.9a}$$

$$\text{subject to} \quad f(x_k, u_k) - x_{k+1} \; = \; 0, \quad \text{for} \quad k = 0, \ldots, N-1, \tag{7.9b}$$

$$\bar{x}_0 - x_0 \; = \; 0. \tag{7.9c}$$

The dynamic programming recursion for this problem is given by:

$$J_N(x) = E(x), \quad J_k(x) = \min_u L(x,u) + J_{k+1}(f(x,u)), \quad k = N-1, \dots, 0. \tag{7.10}$$

We remember that we obtained the optimal solution by the forward recursion

$$x_0 = \bar{x}_0, \quad x_{k+1} = f(x_k, u_k), \quad k = 0, \dots, N-1,$$

where u_k is defined by

$$u_k = \arg\min_u L(x_k, u) + J_{k+1}(f(x_k, u)). \tag{7.11}$$

The solution of this optimization problem in u necessarily satisfies the first order necessary optimality condition

$$\nabla_u L(x_k, u_k) + \frac{\partial f}{\partial u}(x_k, u_k)^T \nabla J_{k+1}(f(x_k, u_k)) = 0 \tag{7.12}$$

which defines u_k locally if the problem is locally strictly convex, i.e., it objective has a positive definite Hessian at (x_k, u_k). We now formulate simple conditions on x_k and u_k that follow necessarily from the DP recursion. For this aim we first note that on the optimal trajectory holds $x_{k+1} = f(x_k, u_k)$ and that we trivially obtain along the optimal trajectory

$$J_N(x_N) = E(x_N), \quad J_k(x_k) = L(x_k, u_k) + J_{k+1}(x_{k+1}), \quad k = N-1, \dots, 0.$$

This implies for example that the value function remains constant on the whole trajectory for problems with zero stage costs. However, it is even more interesting to regard the gradient $\nabla J_k(x_k)$ along the optimal state trajectory. If we differentiate (7.10) at the point x_k with respect to x we obtain

$$\nabla J_N(x_k) = \nabla E(x_k), \quad \nabla J_k(x_k)^T = \frac{d}{dx} \underbrace{L(x_k, u_k) + J_{k+1}(f(x_k, u_k))}_{=: \tilde{J}_k(x_k, u_k)} \quad k = N-1, \dots, 0. \tag{7.13}$$

In the evaluation of the total derivative it is needed to observe that the optimal u_k is via (7.12) an implicit function of x_k. However, it turns out that the derivative does not depend on $\frac{du_k}{dx_k}$ because of

$$\frac{d}{dx}\tilde{J}_k(x_k, u_k) = \frac{\partial \tilde{J}_k}{\partial x}(x_k, u_k) + \underbrace{\frac{\partial \tilde{J}_k}{\partial u}(x_k, u_k)}_{=0} \frac{du_k}{dx_k}, \tag{7.14}$$

where the partial derivative with respect to u is zero because of (7.12). Thus, the gradients of the value function at the optimal trajectory have to satisfy the recursion

$$\nabla J_k(x_k) = \nabla_x L(x_k, u_k) + \frac{\partial f}{\partial x}(x_k, u_k)^T \nabla J_{k+1}(x_{k+1}) \quad k = N-1, \dots, 0. \tag{7.15}$$

This recursive condition on the gradients $\nabla J_k(x_k)$ is equivalent to the first order necessary condition (FONC) for optimality that we obtained previously for differentiable optimal control problems, if we identify the gradients with the multipliers, i.e. set

$$\lambda_k = \nabla J_k(x_k). \tag{7.16}$$

This is a very important interpretation of the multipliers λ_k: they are nothing else than the gradients of the value function along the optimal trajectory!

7.8 A Discrete Time Minimum Principle

Collecting all necessary conditions of optimality that we just derived, but substituting $\nabla J_k(x_k)$ by λ_k we arrive indeed exactly to the same conditions (5.16) that we derived in Chapter 5 in a completely different way.

$$
\begin{aligned}
x_0 &= \bar{x}_0 && (7.17a)\\
x_{k+1} &= f(x_k, u_k), \quad k = 0, \ldots, N-1, && (7.17b)\\
\lambda_N &= \nabla_{x_N} E(x_N) && (7.17c)\\
\lambda_k &= \nabla_x L(x_k, u_k) + \frac{\partial f}{\partial x}(x_k, u_k)^T \lambda_{k+1}, \quad k = N-1, \ldots, 1, && (7.17d)\\
0 &= \nabla_u L(x_k, u_k) + \frac{\partial f}{\partial u}(x_k, u_k)^T \lambda_{k+1}, \quad k = 0, \ldots, N-1. && (7.17e)
\end{aligned}
$$

In the context of continuous time problems, we will arrive at a very similar formulation, which has the interesting features that the recursion for λ becomes a differential equation that can be integrated forward in time if desired, and that the optimization problem in (7.11) does only depend on the gradient of J. This will facilitate the formulation and numerical solution of the necessary optimality conditions as a boundary value problem.

Part III

Continuous Time Optimal Control

Chapter 8

Continuous Time Optimal Control Problems

When we are confronted with a problem whose dynamic system lives in continuous time and whose control inputs are a function, we speak of a *continuous time optimal control problem.* This type of problem is the focus of this third part of this script. We will encounter variations of the same concepts as in the discrete time setting, such as Lagrange multipliers λ, the value function J, or the difference between sequential or simultaneous methods. Some numerical methods and details, however, are only relevant in the continuous time setting, such as the indirect methods and Pontryagin's Maximum Principle described in Chapter 10, or the ODE solvers with sensitivity generation described in Section 11.4.

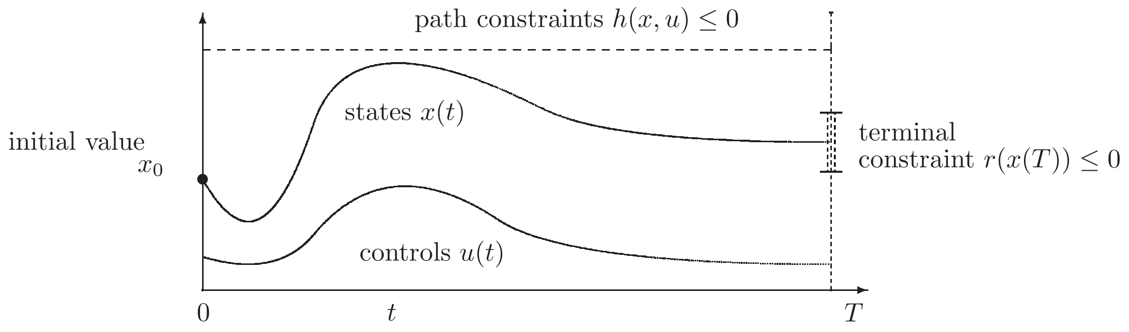
8.1 Formulation of Continuous Time Optimal Control Problems

In an ODE setting, a continuous time optimal control problem can be stated as follows.

$$\begin{array}{rcll}
\underset{x(\cdot),\, u(\cdot)}{\text{minimize}} & & \displaystyle\int_0^T L(x(t),u(t))\, dt \quad + \quad E\,(x(T)) & \qquad (8.1) \\
\text{subject to} & & & \\
x(0) - x_0 & = & 0, & \text{(fixed initial value)} \\
\dot{x}(t) - f(x(t),u(t)) & = & 0, \qquad t \in [0,T], & \text{(ODE model)} \\
h(x(t),u(t)) & \leq & 0, \qquad t \in [0,T], & \text{(path constraints)} \\
r\,(x(T)) & \leq & 0, & \text{(terminal constraints).}
\end{array}$$

The problem and its variables are visualized in Figure 8.1.

The integral cost contribution $L(x,u)$ is sometimes called the *Lagrange term* (which should not


path constraints $h(x,u) \leq 0$
states $x(t)$
initial value x_0
terminal constraint $r(x(T)) \leq 0$
controls $u(t)$
0
t
T
Figure 8.1: The variables and constraints of a continuous time optimal control problem.

be confused with the Lagrange function) and the terminal cost $E(x(T))$ is sometimes called a *Mayer term*. The combination of both, like here, is called a *Bolza objective*.

Note that any Lagrange objective term can be reformulated as a Mayer term, if we add an additional "cost state" c that has to satisfy the differential equation $\dot{c} = L(x,u)$, and then simply take $c(T)$ as the terminal Mayer cost term. Conversely, every differentiable Mayer term can be replaced by by a Lagrange term, namely by $L(x,u) = \nabla E(x)^T f(x,u)$, as the cost integral then satisfies the equality $\int_0^T L(x,u)dt = \int_0^T \frac{dE}{dt}dt = E(x(T)) - E(x_0)$. These two equivalences mean that it would be no restriction of generality to take only one of the two cost contributions, Lagrange or Mayer term, in the above formulation; however, in this script we choose to use the full Bolza objective.

So far, we wrote all functions L, E, f, h independent of time t or of parameters p, and we will leave both of these generalizations away in the remainder of this script. However, all the methods presented in the following chapters can easily be adapted to these two cases, using again state augmentation, as follows. If time dependence occurs, we just introduce a "clock state" t with differential equation $\dot{t} = 1$, and work with the augmented system $\dot{\tilde{x}} = \tilde{f}(\tilde{x}, u)$:

$$\tilde{x} = \begin{bmatrix} x \\ t \end{bmatrix}, \tilde{f}(\tilde{x}, u) = \begin{bmatrix} f(x,u,t) \\ 1 \end{bmatrix}$$

Likewise, in the case that time constant, but free optimization parameters p occur, they can be incorporated as "parameter state" p with differential equation $\dot{p} = 0$ and free initial value.

Another interesting case that is specific to continuous time problems is when the duration T of the problem is free. As an example, we might think of a robot arm that should move an object in minimal time from its current state to some desired terminal position. In this case, we might rescale the time horizon to the interval $[0,1]$ by a time constant but free variable T that is treated like an optimization parameter. Then we regard a scaled problem $\dot{\tilde{x}} = \tilde{f}(\tilde{x}, u)$

$$\tilde{x} = \begin{bmatrix} x \\ T \end{bmatrix}, \tilde{f}(\tilde{x}, u) = \begin{bmatrix} T \cdot f(x,u) \\ 0 \end{bmatrix}$$

with pseudo time $\tau \in [0,1]$, where the initial condition $T(0)$ for the "state" T is free and T satisfies again $\dot{T} = 0$.

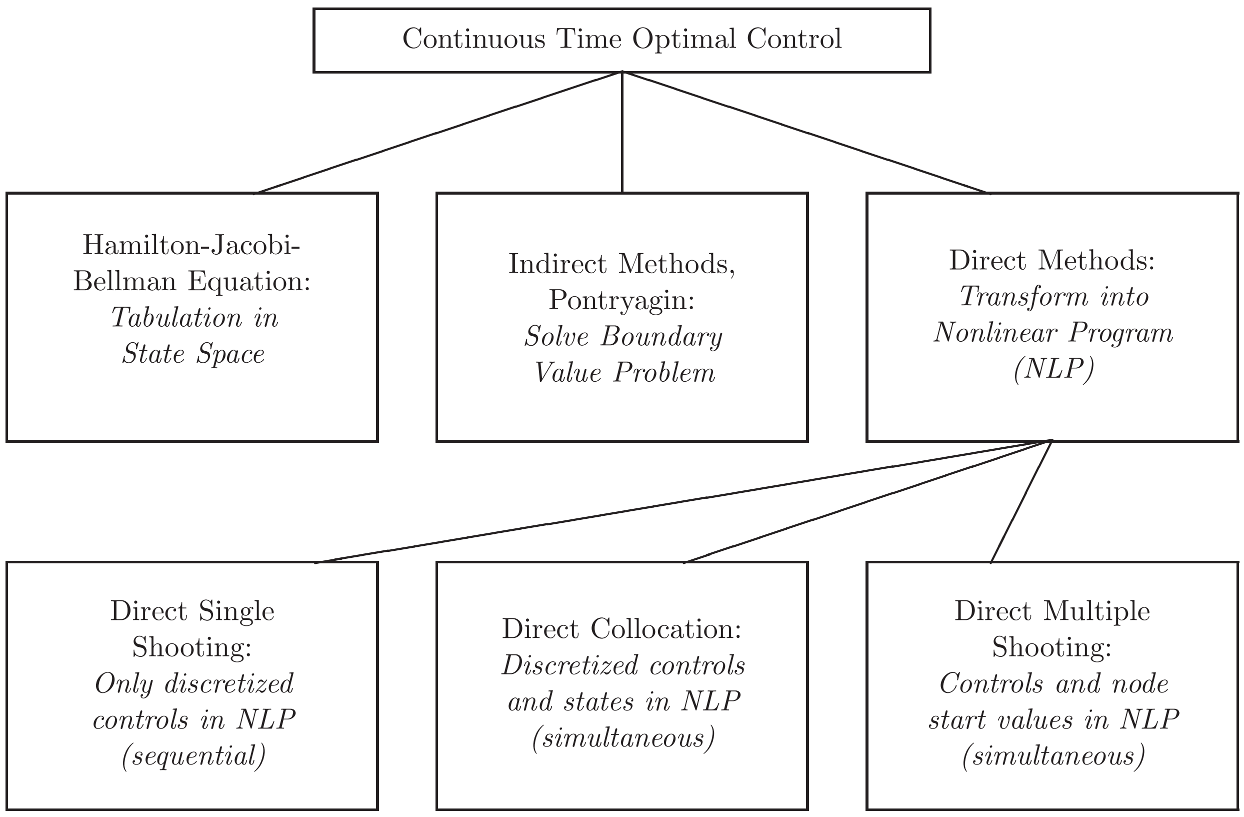

Continuous Time Optimal Control
Hamilton-Jacobi-Bellman Equation: *Tabulation in State Space*
Indirect Methods, Pontryagin: *Solve Boundary Value Problem*
Direct Methods: *Transform into Nonlinear Program (NLP)*
Direct Single Shooting: *Only discretized controls in NLP (sequential)*
Direct Collocation: *Discretized controls and states in NLP (simultaneous)*
Direct Multiple Shooting: *Controls and node start values in NLP (simultaneous)*

Figure 8.2: The optimal control family tree.

We note that although all the above reformulations make it possible to transfer the methods in this script to the respective special cases, an efficient numerical implementation should exploit the structures inherent in these special cases.

8.2 Overview of Numerical Approaches

Generally speaking, there are three basic families of approaches to address continuous time optimal control problems, (a) state-space, (b) indirect, and (c) direct approaches, cf. the top row of Fig. 8.2. We follow here the outline given in [38].

State-space approaches use the principle of optimality that states that each subarc of an optimal trajectory must be optimal. While this was the basis of dynamic programming in discrete time, in the continuous time case this leads to the so-called *Hamilton-Jacobi-Bellman (HJB) equation*, a partial differential equation (PDE) in the state space. Methods to numerically compute solution approximations exist, but the approach severely suffers from Bellmans "curse of dimensionality" and is restricted to small state dimensions. This approach is briefly sketched in Chapter 9.

Indirect Methods use the necessary conditions of optimality of the infinite problem to derive a boundary value problem (BVP) in ordinary differential equations (ODE). This BVP must numerically be solved, and the approach is often sketched as “first optimize, then discretize”. The class of indirect methods encompasses also the well known calculus of variations and the Euler-Lagrange differential equations, and the so-called *Pontryagin Maximum Principle*. The numerical solution of the BVP is performed by shooting techniques or by collocation. The two major drawbacks are that the underlying differential equations are often difficult to solve due to strong nonlinearity and instability, and that changes in the control structure, i.e. the sequence of arcs where different constraints are active, are difficult to handle: they usually require a completely new problem setup. Moreover, on so called singular arcs, higher index differential-algebraic equations (DAE) arise which necessitate specialized solution techniques. This approach is briefly sketched in Chapter 10.

Direct methods transform the original infinite optimal control problem into a finite dimensional nonlinear programming problem (NLP) which is then solved by structure exploiting numerical optimization methods. Roughly speaking, direct methods transform the continuous time dynamic system into a discrete time system and then proceed as described in the first two parts of this script. The approach is therefore often sketched as “first discretize, then optimize”. One of the most important advantages of direct compared to indirect methods is that they can easily treat inequality constraints, like the inequality path constraints in the formulation above. This is because structural changes in the active constraints during the optimization procedure are treated by well developed NLP methods that can deal with inequality constraints and active set changes. All direct methods are based on a finite dimensional parameterization of the control trajectory, but differ in the way the state trajectory is handled, cf. the bottom row of Fig. 8.2. For solution of constrained optimal control problems in real world applications, direct methods are nowadays by far the most widespread and successfully used techniques, and are therefore the focus of this script. Brief descriptions of three of the direct methods – single shooting, multiple shooting, and collocation – and some algorithmic details are given in Chapter 11, while we point out that the first two parts of the script covering finite dimensional optimization and discrete time dynamic systems have already covered most of the algorithmic ideas relevant for direct approaches to optimal control.

Chapter 9

The Hamilton-Jacobi-Bellman Equation

In this short chapter we give a very brief sketch of how the concept of dynamic programming can be utilized in continuous time, leading to the so called Hamilton-Jacobi-Bellman (HJB) Equation. For this aim we regard the following simplified optimal control problem:

$$\begin{aligned}
\underset{x(\cdot),\, u(\cdot)}{\text{minimize}} \quad & \int_0^T L(x(t), u(t))\, dt \quad + \quad E\left(x(T)\right) && (9.1)\\
\text{subject to} \quad & \\
x(0) - \bar{x}_0 \quad &= \quad 0, && \text{(fixed initial value)}\\
\dot{x}(t) - f(x(t), u(t)) \quad &= \quad 0, \qquad t \in [0, T]. && \text{(ODE model)}
\end{aligned}$$

Note that we might approximate all inequality constraints by differentiable barrier functions that tend to infinity when the boundary of the feasible set is reached.

9.1 Dynamic Programming in Continuous Time

In order to motivate the HJB equation, we start by an Euler discretization of the above optimal control problem. Though we would in numerical practice never employ an Euler discretization due to its low order, it is helpful for theoretical purposes, like here. We introduce a timestep $h = \frac{T}{N}$ and then address the following discrete time OCP:

$$\underset{x,\, u}{\text{minimize}} \qquad \sum_{i=0}^{N-1} hL(x_i, u_i) \quad + \quad E\left(x_N\right)$$

$$\begin{aligned} &\text{subject to} \\ x_0 - \bar{x}_0 &= 0, \\ x_{i+1} &= x_i + hf(x_i, u_i) \quad i = 0, \ldots, N-1, \end{aligned}$$

Dynamic programming applied to this optimization problem yields:

$$J_k(x) = \min_u \quad hL(x,u) + J_{k+1}(x + hf(x,u)).$$

Replacing the index k by time points $t_k = kh$ and identifying $J_k(x) = J(x, t_k)$, we obtain

$$J(x, t_k) = \min_u \quad hL(x,u) + J(x + hf(x,u), t_k + h).$$

Assuming differentiability of $J(x,t)$ in (x,t) and Taylor expansion yields

$$J(x,t) = \min_u \quad hL(x,u) + J(x,t) + h\nabla_x J(x,t)^T f(x,u) + h\frac{\partial J}{\partial t}(x,t) + O(h^2).$$

Finally, bringing all terms independent of u to the left side and dividing by $h \to 0$ we obtain already the *Hamilton-Jacobi-Bellman (HJB) Equation*:

$$-\frac{\partial J}{\partial t}(x,t) = \min_u \quad L(x,u) + \nabla_x J(x,t)^T f(x,u).$$

This partial differential equation (PDE) describes the evolution of the value function over time. We have to solve it backwards for $t \in [0,T]$, starting at the end of the horizon with

$$J(x,T) = E(x).$$

The optimal feedback control for the state x at time t is then obtained from

$$u^*_{\text{feedback}}(x,t) = \arg\min_u \quad L(x,u) + \nabla_x J(x,t)^T f(x,u).$$

It is a remarkable fact that the optimal feedback control does not depend on the absolute value, but only on the gradient of the value function, $\nabla_x J(x,t)$. Abbreviating this gradient with $\lambda \in \mathbb{R}^{n_x}$, one introduces the *Hamiltonian function*

$$H(x,\lambda,u) := L(x,u) + \lambda^T f(x,u).$$

Using the new notation and regarding λ as the relevant input of the Hamiltonian, the control can be expressed as an explicit function of x and λ:

$$u^*_{\text{explicit}}(x,\lambda) = \arg\min_u \quad H(x,\lambda,u).$$

Then we can explicitly compute the so called *true Hamiltonian*

$$H^*(x,\lambda) := \min_u H(x,\lambda,u) = H(x,\lambda,u^*_{\text{explicit}}(x,\lambda)),$$

where the control does not appear as input anymore. Using the true Hamiltonian, we can write the Hamilton-Jacobi-Bellman Equation compactly as:

$$-\frac{\partial J}{\partial t}(x,t) = H^*(x, \nabla_x J(x,t))$$

Like dynamic programming, the solution of the HJB Equation also suffers from the "curse of dimensionality" and its numerical solution is very expensive in larger state dimensions. In addition, differentiability of the value function is not always guaranteed such that even the existence of solutions is generally difficult to prove. However, some special cases exist that can analytically be solved, most prominently, again, linear quadratic problems.

9.2 Linear Quadratic Control and Riccati Equation

Let us regard a linear quadratic optimal control problem of the following form.

$$\begin{array}{ll} \underset{x(\cdot),\,u(\cdot)}{\text{minimize}} & \displaystyle\int_0^T \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} Q(t) & S(t)^T \\ S(t) & R(t) \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} dt \;+\; x(T)^T P_{\mathrm{T}} x(T) \end{array}$$

$$\begin{array}{rcll} \text{subject to} & & & \\ x(0) - x_0 & = & 0, & \text{(fixed initial value)} \\ \dot{x} - A(t)x - B(t)u & = & 0, \quad t \in [0, T]. & \text{(linear ODE model)} \end{array}$$

As in discrete time, the value function is quadratic for this type of problem. In order to see this, let us assume that $J(x,t) = x^T P(t) x$. Under this assumption, the HJB Equation reads as

$$-\frac{\partial J}{\partial t}(x,t) = \min_u \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} Q(t) & S(t)^T \\ S(t) & R(t) \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} + 2x^T P(t)(A(t)x + B(t)u).$$

Symmetrizing, the right hand side is given by

$$\min_u \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} Q + PA + A^T P & S^T + PB \\ S + B^T P & R \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix}.$$

By the Schur Complement Lemma, Lemma 7.2, this yields

$$-\frac{\partial J}{\partial t} = x^T \Big(Q + PA + A^T P - (S^T + PB) R^{-1} (S + B^T P) \Big) x,$$

which is again a quadratic term. Thus, if J was quadratic, as assumed, it remains quadratic during the backwards evolution. The resulting matrix differential equation

$$-\dot{P} = Q + PA + A^T P - (S^T + PB) R^{-1} (S + B^T P)$$

with terminal condition

$$P(T) = P_{\mathrm{T}}$$

is called the *differential Riccati equation.* Integrating it backwards allows us to compute the cost-to-go function for the above optimal control problem. The corresponding feedback law is by the Schur complement lemma given as:

$$u^*_{\text{feedback}}(x,t) = -R(t)^{-1}(S(t) + B(t)^T P(t))x.$$

9.3 Infinite Time Optimal Control

Let us now regard an infinite time optimal control problem, as follows.

$$\underset{x(\cdot),\,u(\cdot)}{\text{minimize}} \quad \int_0^\infty L(x(t), u(t))\, dt \tag{9.2}$$

$$\begin{array}{rcll} & \text{subject to} & & \\ x(0) - x_0 & = & 0, & \\ \dot{x}(t) - f(x(t), u(t)) & = & 0, & t \in [0, \infty]. \end{array}$$

The principle of optimality states that the value function of this problem, if it is finite and it exists, must be stationary, i.e. independent of time. Setting $\frac{\partial J}{\partial t}(x,t) = 0$ leads to the stationary HJB equation

$$0 = \min_u \quad L(x,u) + \nabla_x J(x)^T f(x,u)$$

with stationary optimal feedback control law $u^*_{\text{feedback}}(x) = \arg\min_u \quad L(x,u) + \nabla_x J(x)^T f(x,u)$.

This equation is easily solvable in the linear quadratic case, i.e., in the case of an infinite horizon linear quadratic optimal control with time independent cost and system matrices. The solution is again quadratic and obtained by setting

$$\dot{P} = 0$$

and solving

$$0 = Q + PA + A^T P - (S^T + PB) R^{-1} (S + B^T P).$$

This equation is called the *algebraic Riccati equation in continuous time.* Its feedback law is a static linear gain:

$$u^*_{\text{feedback}}(x) = -\underbrace{R^{-1}(S + B^T P)}_{=K} x.$$

Chapter 10

Pontryagin and the Indirect Approach

The indirect approach is an extremely elegant and compact way to characterize and compute solutions to optimal control problems. Its origins date back to the calculus of variations and the classical work by Euler and Lagrange. However, its full generality was only developed in 1950s and 1960s, starting with the seminal work of Pontryagin and coworkers [67]. One of the major achievements of their approach compared to the previous work was the possibility to treat inequality path constraints, which appear in most relevant applications of optimal control, notably in time optimal problems. *Pontryagin's Maximum Principle* describes the necessary optimality conditions for optimal control in continuous time. Using these conditions in order to eliminate the controls from the problem and then numerically solving a boundary value problem (BVP) is called the *indirect approach* to optimal control. It was widely used when the Sputnik and Apollo space missions where planned and executed, and is still very popular in aerospace applications. The main drawbacks of the indirect approach are the facts, (a) that it must be possible to eliminate the controls from the problem by algebraic manipulations, which is not always straightforward or might even be impossible, (b) that the optimal controls might be a discontinuous function of x and λ, such that the BVP suffers from a non-smooth differential equation, and (c) that the differential equation might become very nonlinear and unstable and not suitable for a forward simulation. All these issues of the indirect approach can partially be addressed, and most important, it offers an exact and elegant characterization of the solution of optimal control problems in continuous time.

10.1 The HJB Equation along the Optimal Solution

In order to derive the necessary optimality conditions stated in Pontryagin's Maximum Principle, let us again regard the simplified optimal control problem stated in Equation (9.1), and let us recall that the Hamiltonian function was defined as $H(x, \lambda, u) = L(x, u) + \lambda^T f(x, u)$ and the Hamilton-Jacobi-Bellman equation was formulated as: $-\frac{\partial J}{\partial t}(x, t) = \min_u H(x, \nabla J(x, t), u)$ with terminal condition $J(x,T) = E(x)$. We already made the important observation that the optimal feedback controls

$$u^*_{\text{feedback}}(x,t) = \arg\min_u H(x, \nabla_x J(x,t), u)$$

depend only on the gradient $\nabla_x J(x,t)$, not on J itself. Thus, we might introduce the so called *adjoint variables* or *costates* λ that we identify with this gradient. If we would know the state $x^*(t)$ and costate $\lambda^*(t)$ at a point on the optimal trajectory, then we would obtain the optimal controls $u^*(t)$ from $u^*(t) = u^*_{\text{explicit}}(x^*(t), \lambda^*(t))$ where the explicit control law is defined again by

$$u^*_{\text{explicit}}(x,\lambda) = \arg\min_u H(x,\lambda,u). \tag{10.1}$$

For historical reasons, the characterization of the optimal controls resulting from this pointwise minimum is called *Pontryagin's Maximum Principle*, but we might also refer to it as the *minimum principle* when convenient.

One major question remains, however: how can we characterize and obtain the optimal states and costates $x^*(t)$ and $\lambda^*(t) = \nabla_x J(x^*(t),t)$? The idea is to assume that the trajectory is known, and to differentiate the HJB Equation along this optimal trajectory. Let us regard the HJB Equation

$$-\frac{\partial J}{\partial t}(x,t) = \min_u H(x, \nabla_x J(x,t), u) = H(x, \nabla_x J(x,t), u^*_{\text{explicit}}(x, \nabla_x J(x,t)))$$

and differentiate it totally with respect to x. Note that the right hand side depends via $\nabla_x J(x,t)$ and u^*_{explicit} indirectly on x. Fortunately, we know that $\frac{\partial H}{\partial u}(x^*,\lambda^*,u^*) = 0$ due to the minimum principle, so that we obtain

$$-\frac{\partial^2 J}{\partial x \partial t}(x^*,t) = \frac{\partial H}{\partial x}(x^*,\lambda^*,u^*) + \underbrace{\frac{\partial H}{\partial \lambda}(x^*,\lambda^*,u^*)}_{=f(x^*,u^*)^T} \nabla_x^2 J(x^*,t)$$

where we drop for notational convenience the time dependence for $x^*(t), \lambda^*(t), u^*(t)$. Using $\dot{x}^* = f(x^*,u^*)$ and reordering yields

$$\underbrace{\frac{\partial}{\partial t}\nabla J(x^*,t) + \nabla_x^2 J(x^*,t)\, \dot{x}^*}_{=\frac{d}{dt}\nabla_x J(x^*,t)} = \dot{\lambda}^* = -\nabla_x H(x^*,\lambda^*,u^*)$$

This is a differential equation for the costate λ^*. Finally, we differentiate $J(x,T) = E(x)$ and obtain the terminal boundary condition

$$\lambda(T) = \nabla E(x(T)).$$

Thus, we have derived necessary conditions that the optimal trajectory must satisfy. We combine them with the constraints of the optimal control problem and summarize them as:

$$\begin{array}{rcllr} x^*(0) &=& \bar{x}_0, & & \text{(initial value)} \\ \dot{x}^*(t) &=& f(x^*(t),u^*(t)), & t \in [0,T], & \text{(ODE model)} \\ \dot{\lambda}^*(t) &=& -\nabla_x H(x^*(t),\lambda^*(t),u^*(t)), & t \in [0,T], & \text{(adjoint equations)} \\ u^*(t) &=& \arg\min\limits_u H(x^*(t),\lambda^*(t),u), & t \in [0,T], & \text{(minimum principle)} \\ \lambda^*(T) &=& \nabla E(x^*(T)). & & \text{(adjoint final value)} \end{array} \tag{10.2}$$

Due to the fact that boundary conditions are given both at the start and the end of the time horizon, these necessary optimality conditions form and two-point *boundary value problem (BVP)*. These conditions can either be used to check if a given trajectory can possibly be a solution; alternatively, and more interestingly, we can solve the BVB numerically in order to obtain candidate solutions to the optimal control problem. Note that this is possible due to the fact that the number and type of the conditions matches the number and type of the unknowns: u^* is determined by the minimum principle, while x^* and λ^* are obtained by the ODE and the adjoint equations, i.e. an ODE in $\mathbb{R}^{2n_x}$, in combination with the corresponding number of boundary conditions, n_x at the start for the initial value and n_x at the end for the adjoint final value. But before we discuss how to numerically solve such a BVP we have to address the question of how we can eliminate the controls from the BVP.

10.2 Obtaining the Controls on Regular and on Singular Arcs

Let us in this section discuss how to derive an explicit expression for the optimal control that are formally given by

$$u^*_{\text{explicit}}(x, \lambda) = \arg\min_u H(x, \lambda, u). \tag{10.3}$$

In this section we discuss two cases, first the standard case, and second the case of so called *singular arcs*.

In the benevolent standard case, the optimal controls are simply determined by the equation

$$\frac{\partial H}{\partial u}(x, \lambda, u^*) = 0.$$

In this case, the analytic expression of the derivative has an explicit appearance of the controls, and we can transform the equation in order to obtain the implicit function $u^*_{\text{explicit}}(x, \lambda)$. Let us illustrate this with an example.

Example 10.1 (Linear Quadratic Control with Regular Cost)**:** Regard $L(x, u) = \frac{1}{2}(x^T Q x + u^T R u)$ with positive definite R and $f(x, u) = Ax + Bu$. Then

$$H(x, \lambda, u) = \frac{1}{2}(x^T Q x + u^T R u) + \lambda^T (Ax + Bu)$$

and

$$\frac{\partial H}{\partial u} = u^T R + \lambda^T B.$$

Thus, $\frac{\partial H}{\partial u} = 0$ implies that

$$u^*_{\text{explicit}}(x, \lambda) = -R^{-1} B^T \lambda.$$

Note that the explicit expression only depends on λ here. For completeness, let us also compute the derivative of the Hamiltonian with respect to x, which yields

$$\frac{\partial H}{\partial x} = x^T Q + \lambda^T A,$$

so that the evolution of the costate is described by the adjoint equation

$$\dot{\lambda} = -\frac{\partial H}{\partial x}^T = -A^T\lambda - Qx.$$

If we would have an optimal control problem with fixed initial value $\bar{x}_0$ and quadratic terminal cost, i.e. $E(x) = \frac{1}{2}x^T Px$, then the BVP that we would need to solve is given by

$$\begin{array}{rcll} x^*(0) &=& \bar{x}_0, & \text{(initial value)} \\ \dot{x}^*(t) &=& Ax^*(t) - BR^{-1}B^T\lambda^*(t), \quad t\in[0,T], & \text{(ODE model)} \\ \dot{\lambda}^*(t) &=& -A^T\lambda^*(t) - Qx^*(t) \qquad t\in[0,T], & \text{(adjoint equations)} \\ \lambda^*(T) &=& Px. & \text{(adjoint final value)} \end{array} \tag{10.4}$$

The second and more complicated case occurs if the relation

$$\frac{\partial H}{\partial u}(x,\lambda,u^*) = 0$$

is not invertible with respect to u^*. We then speak of a *singular arc.* This e.g. occurs if $L(x,u)$ is independent of u and $f(x,u)$ is linear in u, as then $\frac{\partial H}{\partial u}$ does not depend explicitly on u. Roughly speaking, singular arcs are due to the fact that *singular perturbations* of the controls – that go up and down infinitely fast – would not matter in the objective and yield exactly the same optimal solution as the well-behaved piecewise continuous control in which we are usually interested. Note that the controls still influence the trajectory on a singular arc, but that this influence is only indirectly, via the evolution of the states.

This last fact points out to a possible remedy: if $\frac{\partial H}{\partial u}$ is zero along the singular arc, then also its total time derivative along the trajectory should be zero. Thus, we differentiate the condition totally with respect to time

$$\frac{d}{dt}\frac{\partial H}{\partial u}(x(t),\lambda(t),u) = 0,$$

which yields

$$\frac{\partial}{\partial x}\frac{\partial H}{\partial u}\underbrace{\dot{x}}_{=f(x,u)} + \frac{\partial}{\partial \lambda}\frac{\partial H}{\partial u}\underbrace{\dot{\lambda}}_{=-\nabla_x H} = 0.$$

We substitute the explicit expressions for $\dot{x}$ and $\dot{\lambda}$ into this equation and hope that now u appears explicitly. If this is still not the case, we differentiate even further, until we have found an $n > 1$ such that the relation

$$\left(\frac{d}{dt}\right)^n \frac{\partial H}{\partial u}(x(t),\lambda(t),u) = 0$$

explicitly depends on u. Then we can invert the relation and finally have an explicit equation for u^*. Let us illustrate this with another example.

Example 10.2 (Linear Quadratic Control with Singular Cost)**:** Regard $L(x,u) = x^T Qx$ and $f(x,u) = Ax + Bu$. Then

$$H(x,\lambda,u) = \frac{1}{2}x^T Qx + \lambda^T(Ax + Bu).$$

and

$$\frac{\partial H}{\partial u} = \lambda^T B.$$

This does not explicitly depend on u and thus u^* can not easily be obtained. Therefore, we differentiate totally with respect to time:

$$\frac{d}{dt}\frac{\partial H}{\partial u} = \dot{\lambda}^T B = -\frac{\partial H}{\partial x} B = -(x^T Q + \lambda^T A)B.$$

This still does not explicitly depend on u. Once more differentiating yields:

$$\frac{d}{dt}\frac{d}{dt}\frac{\partial H}{\partial u} = -\dot{x}^T QB - \dot{\lambda}^T AB = -(Ax + Bu)^T QB + (x^T Q + \lambda^T A)AB.$$

Setting this to zero and transposing it, we obtain the equation

$$-B^T QAx - B^T QBu + B^T A^T Qx + B^T A^T A^T \lambda = 0,$$

and inverting it with respect to u we finally obtain the desired explicit expression

$$u^*_{\text{explicit}}(x, \lambda) = (B^T QB)^{-1} B^T \left(A^T Qx - QAx + A^T A^T \lambda\right).$$

10.3 Pontryagin with Path Constraints

In case of path constraints of the form $h(x(t), u(t)) \leq 0$ for $t \in [0, T]$ the same formalism as developed before is still applicable. In this case, it can be shown that for given x and λ, we need to determine the optimizing u from

$$u^*_{\text{explicit}}(x, \lambda) = \arg\min_u H(x, \lambda, u) \text{ s.t. } h(x, u) \leq 0. \tag{10.5}$$

This is easiest in the case of pure control constraints, i.e. if we have only $h(u) \leq 0$. When mixed state control or pure state constraints occur, the formalism becomes more complicated. In the case of mixed constraints with regular solution of the above optimization problem (10.5), we only have to adapt the adjoint differential equation to $-\dot{\lambda} = \nabla_x H(x, \lambda, u) + \nabla_x h(x, u)\mu^*$ where μ^* is the corresponding solution multiplier. In the case of pure state constraints, if the corresponding state is controllable, we usually have a singular situation and have to regard higher order derivatives in order to obtain feasible trajectories along the active state constraint; in the case of uncontrollable state constraints, we will only have a touching point and the adjoints will typically jump at this point. Let us leave all complications away and illustrate in this section only the nicest case, the one of pure control constraints.

Example 10.3 (Linear Quadratic Problem with Control Constraints)**:** Let us regard constraints $h(u) = Gu + b \leq 0$ and the Hamiltonian $H(x, \lambda, u) = \frac{1}{2}x^T Qx + u^T Ru + \lambda^T(Ax + Bu)$ with R invertible. Then

$$u^*_{\text{explicit}}(x, \lambda) = \arg\min_u H(x, \lambda, u) \text{ s.t. } h(u) \leq 0$$

is equal to

$$u^*_{\text{explicit}}(x,\lambda) = \arg\min_u \frac{1}{2}u^T R u + \lambda^T B u \text{ s.t. } Gx + b \leq 0$$

which is a strictly convex parametric quadratic program (pQP) which has a piecewise affine, continuous solution.

A special and more specific case of the above class is the following.

Example 10.4 (Scalar Bounded Control)**:** Regard scalar u and constraint $|u| \leq 1$, with Hamiltonian

$$H(x,\lambda,u) = \frac{1}{2}u^2 + v(x,\lambda)u + w(x,\lambda)$$

Then, with

$$\tilde{u}(x,\lambda) = -v(x,\lambda)$$

we have

$$u^*_{\text{explicit}}(x,\lambda) = \max\{-1, \min\{1, \tilde{u}(x,\lambda)\}\}$$

Attention: this simple "saturation" trick is only applicable in the case of one dimensional QPs.

10.4 Hamiltonian System Properties

The combined forward and adjoint differential equations have a particular structure: they form a *Hamiltonian system.* In order to see this, first note for notational simplicity that we can directly use the true Hamiltonian $H^*(x,\lambda)$ in the differential equation, and second recall that

$$\nabla_\lambda H^*(x,\lambda) = f(x, u^*_{\text{explicit}}(x,\lambda))$$

Thus,

$$\frac{d}{dt}\begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} \nabla_\lambda H^*(x,\lambda) \\ -\nabla_x H^*(x,\lambda) \end{bmatrix}$$

which is a Hamiltonian system. We might abbreviate the system dynamics as $\dot{y} = \tilde{f}(y)$ with

$$y = \begin{bmatrix} x \\ \lambda \end{bmatrix}, \quad \text{and} \quad \tilde{f}(y) = \begin{bmatrix} \nabla_\lambda H^*(x,\lambda) \\ -\nabla_x H^*(x,\lambda) \end{bmatrix}. \tag{10.6}$$

The implications of this specific structure are, first, that the Hamiltonian is conserved. This can be easily seen by differentiating H totally with respect to time.

$$\begin{aligned} \frac{d}{dt}H^*(x,\lambda) &= \nabla_x H^*(x,\lambda)^T \dot{x} + \nabla_\lambda H^*(x,\lambda)^T \dot{\lambda} && (10.7) \\ &= \nabla_x H^*(x,\lambda)^T \nabla_\lambda H^*(x,\lambda) - \nabla_\lambda H^*(x,\lambda)^T \nabla_x H^*(x,\lambda) && (10.8) \\ &= 0 && (10.9) \end{aligned}$$

Second, by Liouville's Theorem, the fact that the system $\dot{y} = \tilde{f}(y)$ is a Hamiltonian system also means that the volume in the phase space of $y = (x, \lambda)$ is preserved. The implication of this is that even if the dynamics of x is very stable and contracting fast, which is usally not a problem for stiff integration routines, then the dynamics of λ must be expanding and is very unstable. This is an unfortunate fact for numerical approaches to solve the BVP that are based on a forward simulation of the combined differential equation system, like single shooting: if the system $\dot{x} = f(x, u)$ has either some *very unstable* or some *very stable* modes, in both cases the forward simulation of the combined system is an ill-posed problem. In this case, the indirect approach is still applicable when other numerical approaches such as collocation are employed, but looses much of its appeal.

10.5 Numerical Solution of the Boundary Value Problem

In this section we address the question of how we can compute a solution of the boundary value problem (BVP) in the indirect approach. The remarkable observation is that the only non-trivial unknown is the initial value for the adjoints, $\lambda(0)$. Once this value has been found, the complete optimal trajectory can in principle be recovered by a forward simulation of the combined differential equation. Let us first recall that the BVP that we want to solve is given as

$$\begin{aligned} x(0) - \bar{x}_0 &= 0, && (10.10)\\ \lambda(T) - \nabla E(x(T)) &= 0, && (10.11)\\ \dot{x}(t) - \nabla_\lambda H^*(x(t), \lambda(t)) &= 0, \quad t \in [0, T], && (10.12)\\ \dot{\lambda}(t) + \nabla_x H^*(x(t), \lambda(t)) &= 0, \quad t \in [0, T]. && (10.13) \end{aligned}$$

Using the shorthands (10.6) the equation system can be summarized as

$$\begin{aligned} r_0(y(0)) + r_{\mathrm{T}}(y(T)) &= 0, && (10.14)\\ \dot{y}(t) - \tilde{f}(y(t)) &= 0, \quad t \in [0, T]. && (10.15) \end{aligned}$$

This BVP has $2n_x$ differential equations $\dot{y} = \tilde{f}$, and $2n_x$ boundary conditions and is therefore usually well-defined. We explain three approaches to solve this BVP numerically, *single shooting*, *collocation*, and *multiple shooting*.

Single shooting starts with the following idea: for any guess of the initial value y_0, we can use a numerical integration routine in order to obtain the trajectory $y(t; y_0)$ for all $t \in [0, T]$, as a function of y_0. This is visualized in Figure 10.1. The result is that the differential equation (10.15) is by definition already satisfied. Thus, we only need to check the boundary condition (10.14), which we can do using the terminal trajectory value $y(T; y_0)$:

$$\underbrace{r_0(y_0) + r_{\mathrm{T}}(y(T; y_0))}_{=:F(y_0)} = 0.$$

This equation might or might not be satisfied for the given guess y_0. If it is not satisfied, we might iteratively refine the guess y_0 using Newton's method for root finding of $F(y_0) = 0$ which iterates

$$y_0^{k+1} = y_0^k - \left(\frac{\partial F}{\partial y_0}(y_0^k)\right)^{-1} F(y_0^k).$$

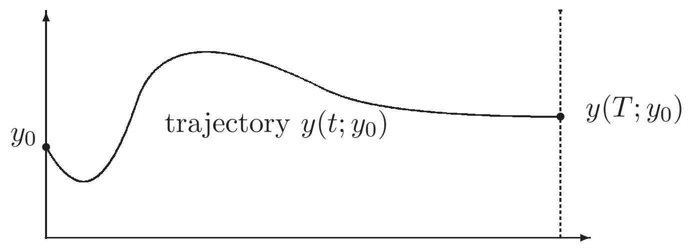


Figure 10.1: Single shooting obtains the trajectory by a forward integration that starts at y_0.

It is important to note that in order to evaluate $\frac{\partial F}{\partial y_0}(y_0^k)$ we have to compute the ODE sensitivities $\frac{\partial y(T;y_0)}{\partial y_0}$.

In some cases, as said above, the forward simulation of the combined ODE might be an ill-conditioned problem so that single shooting cannot be employed. In this case, as alternative approach is to use *simultaneous collocation* that can be sketched as follows. First, we discretize the combined states on a grid with node values $s_i \approx y(t_i)$. Second, we replace the infinite ODE

$$0 = \dot{y}(t) - \tilde{f}(y(t)), \quad t \in [0, T],$$

by finitely many equality constraints

$$c_i(s_i, s_{i+1}) = 0, \quad i = 0, \dots, N-1,$$

e.g. with $c_i(s_i, s_{i+1}) := \frac{s_{i+1}-s_i}{t_{i+1}-t_i} - \tilde{f}\left(\frac{s_i+s_{i+1}}{2}\right)$. Note that one would usually use higher order collocation schemes with several collocation points in each collocation interval. In any case, after discretization, we obtain a large scale, but sparse nonlinear equation system:

$$\begin{array}{rcll} r_0(s_0) + r_{\mathrm{T}}(s_N) &=& 0, & \text{(boundary conditions)} \\ c_i(s_i, s_{i+1}) &=& 0, \qquad i = 0, \dots, N-1. & \text{(discretized ODE model)} \end{array}$$

We can solve this system again with Newton's method. In this case, it is crucial that we exploit the sparsity in the linear system setup and its solution, because of the large dimension of the system. Note that the user has to choose the discretization grid in a way that ensures sufficient numerical accuracy.

A third numerical method that can be regarded a hybrid method between the two previous approaches is called *multiple shooting*, originally due to Osborne [65]. Like single shooting, it uses a forward ODE solver; but like collocation, it divides the time horizon into N subintervals, e.g. of length $\Delta t = T/N$. On each subinterval, it integrates the ODE starting at an initial value s, i.e. it solves the initial value problem on a short horizon

$$\dot{y}(t) = \tilde{f}(y(t)), t \in [0, \Delta t], \quad y(0) = s$$

in order to generate the map $\Phi(s) := y(\Delta t; s)$. Using this map, the nonlinear equation system that needs to be solved in multiple shooting – which is equivalent to the root finding system of single shooting – is given by

$$\begin{array}{rcll} r_0(s_0) + r_{\mathrm{T}}(s_N) & = & 0, & \text{(boundary conditions)} \\ \Phi(s_i) - s_{i+1} & = & 0, \quad i = 0, \ldots, N-1. & \text{(continuity conditions)} \end{array}$$

At first sight multiple shooting seems to combine the disadvantages of both previous methods: like single shooting, it cannot handle strongly unstable systems as it relies on a forward integration, and like collocation, it leads to a large scale equation system and needs sparse treatment of the linear algebra. On the other hand, it also inherits the advantages of the other two methods: like single shooting, it can rely on existing forward solvers with inbuilt adaptivity so that it avoids the question of numerical discretization errors: the choice N is much less important than in collocation and typically, one chooses an N between 5 and 50 in multiple shooting. Also, multiple shooting can be implemented in a way that allows one to perform in each Newton iteration basically the same computational effort as in single shooting, by using a condensing technique. Finally, like collocation, it allows one to deal better with unstable and nonlinear systems than single shooting. These last facts, namely that a *lifted Newton method* can solve the large "lifted" equation system (e.g. of multiple shooting) at the same cost per Newton iteration as the small scale nonlinear equation system (e.g. of single shooting) to which it is equivalent, but with faster local convergence rates, is in detail investigated in [2] where also a literature review on such lifted methods is given.

Chapter 11

Direct Approaches to Continuous Optimal Control

Direct methods to continuous optimal control finitely parameterize the infinite dimensional decision variables, notably the controls $u(t)$, such that the original problem is approximated by a finite dimensional nonlinear program (NLP). This NLP can then be addressed by structure exploiting numerical NLP solution methods. For this reason, the approach is often characterized as "First discretize, then optimize." The direct approach connects easily to all optimization methods developed in the continuous optimization community, such as the methods described in Chapter 2. Most successful direct methods even parameterize the problem such that the resulting NLP has the structure of a discrete time optimal control problem, such that all the techniques and structures described in Chapters 5 and 6 are applicable. For this reason, the current chapter is kept relatively short; its major aim is to outline the major concepts and vocabulary in the field.

We start by describing *direct single shooting*, *direct multiple shooting*, and *direct collocation* and a variant *pseudospectral methods*. We also discuss how sensitivities are computed in the context of shooting methods. The optimization problem formulation we address in this chapter is the same as (8.1) in Chapter 8. The direct methods differ in how they transcribe this problem into a finite NLP. The problem (8.1) has a fixed initial value which simplifies in particular the single shooting method, but all concepts can in a straightforward way be generalized to other OCP formulations with free initial values.

11.1 Direct Single Shooting

All shooting methods use an embedded ODE or DAE solver in order to eliminate the continuous time dynamic system. They do so by first parameterizing the control function $u(t)$, e.g. by polynomials, by piecewise constant functions, or, more generally, by piecewise polynomials. We denote the finite control parameters by the vector q, and the resulting control function by $u(t; q)$.

The most widespread parameterization are piecewise constant controls, for which we choose a fixed grid $0 = t_0 < t_1 < \ldots < t_N = T$, and N parameters $q_i \in \mathbb{R}^{n_u}$, $i = 0, \ldots, N-1$, and then we set

$$u(t;q) = q_i \quad \text{if} \quad t \in [t_i, t_{i+1}].$$

Thus, the dimension of the vector $q = (q_0, \ldots, q_{N-1})$ is Nn_u. In single shooting, which is a *sequential approach* earliest presented in [48, 68], we then regard the states $x(t)$ on $[0,T]$ as dependent variables that are obtained by a forward integration of the dynamic system, starting at x_0 and using the controls $u(t;q)$. We denote the resulting trajectory as $x(t;q)$. In order to discretize inequality path constraints, we choose a grid, typically the same as for the control discretization, at which we check the inequalities. Thus, in single shooting, we transcribe the OCP (8.1) into the following NLP, that is visualized in Figure 11.1.

$$\underset{q \in \mathbb{R}^{Nn_u}}{\text{minimize}} \quad \int_0^T L(x(t;q), u(t;q))\, dt \;+\; E\left(x(T;q)\right) \tag{11.1}$$

subject to

$$\begin{array}{rcll} h(x(t_i;q), u(t_i;q)) & \leq & 0, & i = 0, \ldots, N-1, \quad \text{(discretized path constraints)} \\ r\left(x(T;q)\right) & \leq & 0. & \quad \text{(terminal constraints)} \end{array}$$

As the only variable of this NLP is the vector $q \in \mathbb{R}^{Nn_u}$ that influences nearly all problem functions, the above problem can usually be solved by a dense NLP solver in a black-box fashion. As the problem functions and their derivatives are expensive to compute, while a small QP is cheap to solve, often Sequential Quadratic Programming (SQP) is used, e.g. the codes NPSOL or SNOPT. Let us first assume the Hessian needs not be computed but can be obtained e.g. by BFGS updates.

The computation of the derivatives can be done in different ways with a different complexity: first, we can use forward derivatives, using finite differences or algorithmic differentiation. Taking the computational cost of integrating one time interval as one computational unit, this means that one complete forward integration costs N units. Given that the vector q has Nn_u components, this means that the computation of all derivatives costs $(Nn_u + 1)N$ units when implemented in the most straightforward way. This number can still be reduced by one half if we take into account that controls at the end of the horizon do not influence the first part of the trajectory. We might call this way the *reduced derivative computation* as it computes directly only the reduced quantities needed in each reduced QP.

Second, if the number of output quantities such as objective and inequality constraints is not big, we can use the principle of reverse automatic differentiation in order to generate the derivatives. In the extreme case that no inequality constraints are present and we only need the gradient of the objective, this gradient can cheaply be computed by reverse AD, as done in the so called *gradient methods*. Note that in this case the same adjoint differential equations of the indirect approach can be used for reverse computation of the gradient, but that in contrast to the indirect method we do not eliminate the controls, and we integrate the adjoint equations backwards in time. The complexity for one gradient computation is only $4N$ computational units. However, each additional state constraint necessitates a further backward sweep.

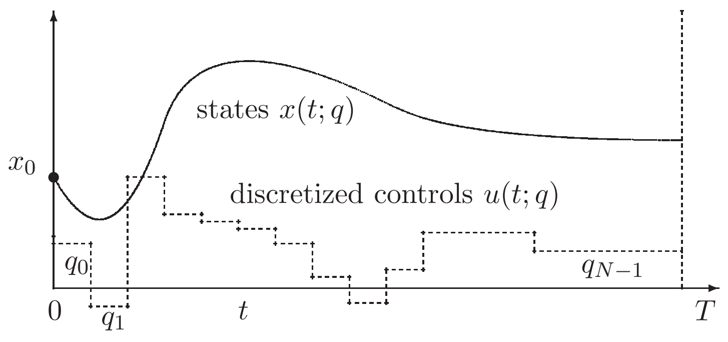


Figure 11.1: The NLP variables in the direct single shooting method.

Third, in the case that we have chosen piecewise controls, as here, we might use the fact that after the piecewise control discretization we have basically transformed the continuous time OCP into a discrete time OCP (see next section). Then we can compute the derivatives with respect to both s_i and q_i on each interval separately, which costs $(n_x + n_u + 1)$ units. This means a total derivative computation cost of $N(n_x + n_u + 1)$ units. In contrast to the second (adjoint) approach, this approach can handle an arbitrary number of path inequality constraints, like the first one. Note that it has the same complexity that we obtain in the standard implementation of the multiple shooting approach, as explained next. We remark here already that both shooting methods can each implement all the above ways of derivative generation, but differ in one respect only, namely that single shooting is a sequential and multiple shooting a simultaneous approach.

11.2 Direct Multiple Shooting

The direct multiple shooting method that was originally developed by Bock and Plitt [21] performs first a piecewise control discretization on a grid, exacly as we did in single shooting, i.e. we set

$$u(t) = q_i \quad \text{for} \quad t \in [t_i, t_{i+1}].$$

But then, it solves the ODE separately on each interval $[t_i, t_{i+1}]$, starting with artificial initial values s_i:

$$\begin{aligned} \dot{x}_i(t; s_i, q_i) &= f(x_i(t; s_i, q_i), q_i), \quad t \in [t_i, t_{i+1}], \\ x_i(t_i; s_i, q_i) &= s_i. \end{aligned}$$

Thus, we obtain trajectory pieces $x_i(t; s_i, q_i)$. Likewise, we numerically compute the integrals

$$l_i(s_i, q_i) := \int_{t_i}^{t_{i+1}} L(x_i(t_i; s_i, q_i), q_i) dt.$$

Finally, we choose a grid at which we check the inquality path constraints; here we choose the same as for the controls and states, but note that a much finer sampling would be possible as well, which, however, requires continuous output from the integrator. Thus, the NLP that is solved in multiple shooting and that is visualized in Figure 11.2 is given by

$$\underset{s,\,q}{\text{minimize}} \quad \sum_{i=0}^{N-1} l_i(s_i, q_i) \;+\; E\,(s_N) \tag{11.2}$$

$$\begin{array}{rcll}
\text{subject to} & & & \\
x_0 - s_0 & = & 0, & & \text{(initial value)} \\
x_i(t_{i+1}; s_i, q_i) - s_{i+1} & = & 0, & i = 0, \ldots, N-1, & \text{(continuity)} \\
h(s_i, q_i) & \leq & 0, & i = 0, \ldots, N, & \text{(discretized path constraints)} \\
r\,(s_N) & \leq & 0. & & \text{(terminal constraints)}
\end{array}$$

Note that by setting $f_i(s_i, q_i) := x_i(t_{i+1}; s_i, q_i)$ the continuity conditions can be interpreted a discrete time dynamic system $s_{i+1} = f_i(s_i, q_i)$ and the above optimal control problem has exactly the same structure as the discrete time optimal control problem (6.1) discussed in detail in Chapter 6. Most important, we can and should employ a sparsity exploiting NLP solver. Regarding the derivative computation, nearly all cost resides in the derivatives of the discrete time dynamic system, i.e. the matrices A_i and B_i in (6.5). If again the simulation on one interval, i.e. one evaluation of f_i, costs one unit, then the computation of these matrices by finite differences costs $(n_x + n_u + 1)$, and as we need N of them, we have a total derivative computation cost of $N(n_x + n_u + 1)$ per Newton-type iteration.

Remark on Schlöder's Reduction Trick: We point out here that the derivatives of the condensed QP could also directly be computed, using the reduced way, as explained as first variant in the context of single shooting. It exploits the fact that the initial value x_0 is fixed in the NMPC problem, changing the complexity of the derivative computations. It is only advantageous for large state but small control dimensions as it has a complexity of $N^2 n_u$. It was originally developed by Schlöder [71] in the context of Gauss-Newton methods and generalized to general SQP shooting methods by [70]. A further generalization of this approach to solve a "lifted" (larger, but equivalent) system with the same computational cost per iteration is the so called *lifted Newton method* [2] where also an analysis of the benefits of lifting is made.

The main advantages of lifted Newton approaches such as multiple shooting compared with single shooting are the facts that (a) we can also initialize the state trajectory, and (b), that they show superior local convergence properties in particular for unstable systems. An interesting remark is that if the original system is linear, continuity is perfectly satisfied in all SQP iterations, and single and multiple shooting would be identical. Also, it is interesting to recall that the Lagrange multipliers λ_i for the continuity conditions are an approximation of the adjoint variables, and that they indicate the costs of continuity.

Finally, it is interesting to note that a direct multiple shooting algorithm can be made a single shooting algorithm easily: we only have to overwrite, before the derivative computation, the states s by the result of a forward simulation using the controls q obtained in the last Newton-type iteration. From this perspective, we can regard single shooting as a variant of multiple shooting where we perturb the result of each iteration by a "feasibility improvement" that makes all continuity conditions feasible by the forward simulation, implicitly giving priority to the control guess over the state guess [74].

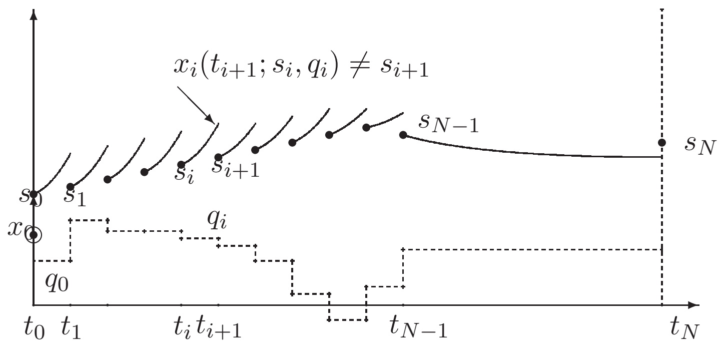


Figure 11.2: The NLP variables in the direct multiple shooting method.

11.3 Direct Collocation

A third important class of direct methods are the direct transcription methods, most notably *direct collocation.* Here we discretize the infinite OCP in both controls and states on a fixed and relatively fine grid $\{t_k\}_{k=0}^{N}$; recall that each collocation interval corresponds to an integrator step. We denote the states on the grid points by $s_k \approx x(t_k)$. We choose a parameterization of the controls on the same grid, e.g. piecewise constant or piecewise polynomials, with control parameters q_k that yield on each interval a function $u_k(t; q)$

On each collocation interval $[t_k, t_{k+1}]$ a set of m collocation points $t_k^{(1)}, \dots, t_k^{(m)}$ is chosen and the trajectory is approximated by a polynomial $p_k(t; v_k)$ with coefficient vector v_k. As equalities of the optimization problem we now require that the collocation conditions (1.24) are met at the collocation points.

$$s_k = p_k(t_k; v_k) \tag{11.3a}$$

$$f(p_k(t_k^{(1)}; v_k), u_k(t_k^{(1)}; q_k)) = p_k'(t_k^{(1)}; v) \tag{11.3b}$$

$$\vdots \tag{11.3c}$$

$$f(p_k(t_k^{(m)}; v_k), u_k(t_k^{(m)}; q_k)) = p_k'(t_k^{(m)}; v) \tag{11.3d}$$

We summarize this system by the vector equation $c_k(s_k, v_k, q_k) = 0$ that has as many components as the vector v_i. Additionally, we also require continuity accross interval boundaries, i.e. we add the constraints $p_k(t_{k+1}; v_k) - s_{k+1} = 0$. We also approximate the integrals $\int_{t_k}^{t_{k+1}} L(x, u)dt$ on the collocation intervals by a quadrature formula using the same collocation points, which we denote by the a term $l_k(s_k, v_k, q_k)$. Path constraints are enforced on a grid, e.g. the interval boundaries, which we do here. We point out, that much finer sampling is possible as well, e.g. on the collocation nodes or even more often. Thus, we obtain a large scale, but sparse NLP:

$$\underset{s,v,q}{\text{minimize}} \quad \sum_{k=0}^{N-1} l_k(s_k, v_k, q_k) \; + \; E(s_N) \tag{11.4}$$

$$\begin{array}{rcll}
\text{subject to} & & & \\
s_0 - x_0 & = & 0, & & \text{(fixed initial value)} \\
c_k(s_k, v_k, q_k) & = & 0, & k = 0, \ldots, N-1, & \text{(collocation conditions)} \\
p_k(t_{k+1}; v_k) - s_{k+1} & = & 0, & k = 0, \ldots, N-1, & \text{(continuity conditions)} \\
h(s_k, q_k) & \leq & 0, & k = 0, \ldots, N-1, & \text{(discretized path constraints)} \\
r\,(s_N) & \leq & 0. & & \text{(terminal constraints)}
\end{array}$$

This large sparse NLP needs to be solved by structure exploiting solvers, and due to the fact that the problem functions are typically relatively cheap to evaluate compared with the cost of the linear algebra, nonlinear interior point methods are often the most efficient approach here. A widespread combination is to use collocation with IPOPT using the AMPL interface. It is interesting to note that, like in direct multiple shooting, the multipliers of the continuity conditions are again an approximation of the adjoint variables.

An interesting variant of orthogonal collocation methods that is often called the *pseudospectral optimal control method* uses only one collocation interval but on this interval it uses an extremly high order polynomial. State constraints are then typically enforced at all collocation points.

11.4 Sensitivity Computation in Shooting Methods

In all shooting methods we need to compute derivatives of the result of an ODE integration routine, or, in the more general case, of a DAE solver, on a given time interval. Let us for notational simplicity regard just the autonomous ODE case $\dot{x} = f(x)$ on a time interval $[0, T]$. The case of control or other parameters on which this ODE depends as well as time dependence can conceptually be covered by state augmentation. Thus, we regard a starting point s and the evolution of the ODE

$$\dot{x} = f(x), \quad t \in [0, T], \quad x(0) = s. \tag{11.5}$$

This gives a solution $x(t; s)$, $t \in [0, T]$, and we are most interested in the terminal value $x(T; s)$ and in the sensitivity matrix

$$G(t) = \frac{\partial x(t; s)}{\partial s}, \quad t \in [0, T],$$

and in particular its terminal value. This matrix $G(T) \in \mathbb{R}^{n_x \times n_x}$ can be computed in many different ways, five of which we briefly sketch here.

1. External Numerical Differentiation (END)
2. Solution of the Variational Differential Equations
3. Algorithmic Differentiation (AD) of the Integrator
4. Internal Algorithmic Differentiation within the Integrator
5. Internal Numerical Differentiation (IND)

In all five methods we assume that the integrator to be differentiated is a state-of-the-art integrator with inbuilt error control and adaptive step-size selection.

The first approach, *External Numerical Differentiation (END)*, just treats the integrator as a black box function and uses finite differences. We perturb s by some quantity $\epsilon > 0$ in the direction of the unit vectors e_i and call the integrator several times in order to compute directional derivatives by finite differences:

$$G(T)e_i \approx \frac{x(T; s + \epsilon e_i) - x(T; q)}{\epsilon} \tag{11.6}$$

The cost of this approach to compute $G(T)$ is $(n_x + 1)$ times the cost of a forward simulation. The approach is very easy to implement, but suffers from one serious problem: due to integrator adaptivity, each call might have a different discretization grid. This error control of each trajectory does not only create an overhead, but worse, it might result in discontinuous perturbations even for small ϵ. It is important to note that due to adaptivity, the output $x(T; s)$ is not a differentiable function in s, but only guaranteed to be close to the true solution within the integrator accuracy TOL, e.g. TOL $= 10^{-4}$. Thus, we need to use, as a rule of thumb, $\epsilon = \sqrt{\text{TOL}}$ in order to make large enough perturbations. As finite differences always mean that we loose half the digits of accuracy, we might easily end e.g. with a derivative that has only two valid digits.

A completely different approach is to formulate and solve the *variational differential equations* along with the nominal trajectory. This means that we solve, together with $\dot{x} = f(x)$, the additional matrix differential equation

$$\dot{G}(t) = \frac{\partial f}{\partial x}(x(t))G(t), \ t \in [0, T], \quad G(0) = \mathbb{I}.$$

This is much more accurate than the first approach at a similar computational cost, but we have to get analytic expressions for $\frac{\partial f}{\partial x}(x(t))$. Also, it is interesting to note that the computed sensitivity $G(T)$ might not be 100% identical with the derivative of the (discretized) integrator result $x(T; s)$.

This last disadvantage is avoided in the third approach, *Algorithmic Differentiation (AD) of the Integrator*, where we first freeze the discretization scheme at the current nominal trajectory and then apply an AD tool to the whole integrator. This is up to machine precision 100% identical with the derivative of the numerical solution $x(T; s)$ for a given fixed discretization grid. In a practical implementation, the integrator and right hand side function $f(x)$ need to be in the same or in compatible computer languages that are treated by the corresponding AD tool (e.g. C++ when using ADOL-C). Also, if an implicit integrator is used, it should be noted that the underlying Newton iterations will differentiated, which might create considerable and avoidable overhead compared to the variational differential equation approach.

A fourth approach, *Internal Algorithmic Differentiation (AD) of the Integrator* can be seen as a combination of the variational differential equation and AD. Here, AD is applied to each step of the integrator in a custom implementation of the integrator, but care is taken that no components of the algorithm are differentiated that need not be differentiated, such as Newton matrices. The approach is illustrated for an Euler scheme (where it is identical to both the variational differential equation and external AD). If the grid is given by $\{t_k\}_{k=0}^N$ and the Euler iterates

$$x_{k+1} = x_k + (t_{k+1} - t_k)f(x_k), \quad k = 0, \ldots, N-1, \quad x_0 = s.$$

then this approach generates matrices

$$G_{k+1} = G_k + (t_{k+1} - t_k)\,\frac{\partial f}{\partial x}(x_k)\,G_k, \quad k = 0, \ldots, N-1, \quad G_0 = \mathbb{I}.$$

This approach is usually the most computationally efficient of the exact differentiation approaches but requires a custom implementation of an ODE/DAE solver that is explicitly designed for the generation of sensitivities. Note that as in the previous two approaches, we cannot deal with black-box right hand side functions $f(x)$ as we need to compute their derivatives symbolically or algorithmically, though the matrix $\frac{\partial f}{\partial x}(x_k)$ could of course also be computed by finite differences.

This last idea can be generalized to the concept of *Internal Numerical Differentiation (IND)* [18]. At first sight it is similar to END, but needs a custom implementation and differs in several respects. First, all trajectories are computed simultaneously; only the nominal trajectory is adaptive, while the perturbed trajectories use the nominal, frozen grid. In implicit methods, also matrix factorizations etc. will be frozen. At the end of the interval, we use the finite difference formula (11.6) but with a much smaller perturbation, namely $\epsilon = \sqrt{\text{PREC}}$ where PREC is the machine precision, typically 10^{-16} The derivatives will have the accuracy $\sqrt{\text{PREC}}$, i.e. usually 10^{-8}, which is much higher than for END.

Again, we illustrate IND at hand of the explicit Euler integration scheme, where each perturbed trajectory with index $i = 1, \ldots, n_x$ just satisfies

$$x^i_{k+1} = x^i_k + (t_{k+1} - t_k) f(x^i_k), \quad k = 0, \ldots, N-1, \quad x^i_0 = s + \epsilon e_i.$$

Note that due to the fact that adaptivity and possible matrix factorizations are switched off for the perturbed trajectories, IND is not only more accurate, but also cheaper than END.

11.5 A Classification of Direct Optimal Control Methods

It is an interesting exercise to try to classify Newton type optimal control algorithms, where we follow the presentation given in [34]. Let us have a look at how nonlinear optimal control algorithms perform their major algorithmic components, each of which comes in several variants:

(a) Treatment of Inequalities: Nonlinear IP vs. SQP

(b) Nonlinear Iterations: Simultaneous vs. Sequential

(c) Derivative Computations: Full vs. Reduced

(d) Linear Algebra: Banded vs. Condensing

In the last two of these categories, we observe that the first variants each exploit the specific structures of the simultaneous approach, while the second variant reduces the variable space to the one of the sequential approach. Note that reduced derivatives imply condensed linear algebra, so the combination [Reduced,Banded] is excluded. In the first category, we might sometimes distinguish two variants of SQP methods, depending on how they solve their underlying QP problems, via active set QP solvers (SQP-AS) or via interior point methods (SQP-IP).

Based on these four categories, each with two alternatives, and one combination excluded, we obtain 12 possible combinations. In these categories, the classical single shooting method [68] could be classified as [SQP,Sequential,Reduced] or as [SQP,Sequential,Full,Condensing] because some variants compute directly the reduced derivatives $\bar{R}^u$ in (13.12b), while others compute first the stagewise derivative matrices A_i and B_i and condense then. Tenny's feasibility perturbed SQP method [74] could be classified as [SQP,Sequential,Full,Banded], and Bock's multiple shooting [21] as well as the classical reduced SQP collocation methods [75, 14, 12] as [SQP,Simultaneous,Full,Condensing]. The band structure exploiting SQP variants from Steinbach [73] and Franke [42] are classified as [SQP-IP,Simultaneous,Full,Banded], while the widely used interior point direct collocation method in conjunction with IPOPT by Biegler and Wächter [76] as [IP,Simultaneous,Full,Banded]. The reduced Gauss-Newton method of Schlöder [71] would here be classified as [SQP,Simultaneous,Reduced].

Part IV

Real-Time Optimization

Chapter 12

Nonlinear Model Predictive Control

So far, we have regarded one single optimal control problem and focussed on ways to numerically solve this problem. Once we have computed such a solution, we might try to control the corresponding real process with the obtained control trajectory. This approach to use a precomputed control trajectory is called *open-loop control*. Unfortunately, the result will most probably be very dissatisfying, as the real process will typically not coincide completely with the model that we have used for optimization. If we wanted for example move a robot arm to a terminal point, the robot arm might end at a very different location than the model predicted. This is due to the difference of the model with the reality, sometimes called *model-plant-mismatch*. This mismatch might be due to modelling errors or external, unforeseen disturbances.

On the other hand, we might be able to observe the real process during its time development, and notice, for example, that the robot arm moves differently than predicted. This will allow us to correct the control inputs online in order to get a better performance; this procedure is called *feedback control* or *closed-loop control*. Feedback allows us to improve the practical performance of optimal control enormously. In its most basic form, we could use ad-hoc implementations of feedback that react to deviations from the planned state trajectory by basic control schemes such as a *proportional-integral (PI)* controller. On the other hand, we might use again optimal control techniques in order to react to disturbances of the state, by using *optimal feedback control*, which we had outlined in the Chapters 7 and 9 on dynamic programming (DP) and the HJB Equation. In the case of the moving robot arm this would result in the following behaviour: if during its motion the robot arm is strongly pushed by an external disturbance, it will not try to come back to its planned trajectory but instead adapt to the new situation and follow the new optimal trajectory. This is straightforward in the case of DP or HJB, where we have the optimal feedback control precomputed for all possible states. But as said, these approaches are impossible to use for nontrivial state dimensions, i.e. systems with more than, say, 3-8 states. Thus, typically we cannot precompute the optimal feedback control in advance.

A possible remedy is to compute the optimal feedback control in *real-time*, or *online*, during the runtime of the process. In the case of the robot arm this means that after the disturbance, we would call our optimization solver again in order to quickly compute the new optimal trajectory.

If we could solve this problem exactly and infinitely fast, we would get exactly the same feedback as in optimal feedback control. In reality, we have to work with approximations: first, we might simplify the optimal control problem in order to allow faster computation, e.g. by predicting only a limited amount of time into the future, and second, we might adapt our algorithms to the new task, namely that we have to solve optimization problems again and again. This task is called *real-time optimization* or *embedded optimization*, due to the fact that in many cases, the numerical optimization will be carried out on *embedded hardware*, i.e. processors that reside not in a desktop computer but e.g. in a feedback control system.

While this idea of *optimal feedback control via real-time optimization* sounds challenging or even impossible for the fast motion of robot arms, it is since decades industrial practice in the process control industry under the name of *Model Predictive Control (MPC)*. There, time scales are often in the range of minutes and allow ample time for each optimization. The main stream implementation of MPC can in discrete time roughly be formulated as follows: (1) observe the current state of the system $\bar{x}_0$, (2) predict and optimize the future behaviour of the process on a limited time window of N steps by solving an open-loop optimization problem starting at the state $\bar{x}_0$, (3) implement the first control action u_0^* at the real process, (4) move the optimization horizon one time step forward and repeat the procedure. MPC is sometimes also called *receding horizon control* due to this movement of the *prediction horizon*. The name *nonlinear MPC*, short *NMPC*, is reserved for the special case of MPC with underlying nonlinear dynamic systems, while linear MPC refers to MPC with linear system models. Note that NMPC leads typically to non-convex optimization problems while nearly all linear MPC formulations use convex cost and constraints.

Note that in the case of a time-invariant system and cost, the subsequent optimization problems differ only by the initial value $\bar{x}_0$ and nothing else, and thus, the MPC feedback is time-invariant as well. If we would be able to solve the problem with an infinite prediction horizon, we would obtain the stationary optimal feedback control. The limitation of the horizon to a finite length N allows us to solve the problem numerically. If we choose N large enough, it will be a good approximation to the infinite horizon problem.

In this script, we do not focus on the different ways to formulate the MPC problem, but on its numerical solution by suitable real-time optimization methods. This and the next chapter follows the presentation given in [34] and [30] and focusses on the MPC optimal control problem.

12.1 NMPC Optimization Problem

Let us in this chapter regard the following simplified optimal control problem in discrete time augmented with algebraic equations.

$$\begin{aligned}
\underset{x,\,z,\,u}{\text{minimize}} \quad & \sum_{i=0}^{N-1} L(x_i, z_i, u_i) \quad + \quad E\,(x_N) && \text{(12.1a)}\\
\text{subject to} \quad x_0 - \bar{x}_0 &= 0, && \text{(12.1b)}\\
x_{i+1} - f(x_i, z_i, u_i) &= 0, \quad i = 0, \ldots, N-1, && \text{(12.1c)}\\
g(x_i, z_i, u_i) &= 0, \quad i = 0, \ldots, N-1, && \text{(12.1d)}\\
h(x_i, z_i, u_i) &\leq 0, \quad i = 0, \ldots, N-1, && \text{(12.1e)}\\
r\,(x_N) &\leq 0. && \text{(12.1f)}
\end{aligned}$$

Here, $x_i \in \mathbb{R}^{n_x}$ is the differential state, $z_i \in \mathbb{R}^{n_z}$ the algebraic state, and $u_i \in \mathbb{R}^{n_u}$ is the control. Functions f and g are assumed twice differentiable and map into $\mathbb{R}^{n_x}$ and $\mathbb{R}^{n_z}$, respectively. The algebraic state z_i is uniquely determined by (12.1d) when x_i and u_i are fixed, as we assume that $\frac{\partial g}{\partial z}$ is invertible everywhere.

We choose to regard this difference-algebraic system form because it covers several parametrization schemes for continuous time dynamic systems in differential algebraic equation (DAE) form, in particular direct multiple shooting with DAE relaxation [55] and direct collocation [75, 14]. Note that in the case of collocation, all collocation equations on a collocation interval would be collected within the function g and the collocation node values in the variables z_i, see the formulation in formula (11.4).

Here, the free variables are the differential state vector $x = (x_0^T, x_1^T \ldots, x_{N-1}^T, x_N^T)^T$ at all considered time points and the algebraic and control vector on all but the last time points: $z = (z_0^T, z_1^T \ldots, z_{N-1}^T)^T$ and $u = (u_0^T, u_1^T \ldots, u_{N-1}^T)^T$.

The task in real-time optimization for NMPC is now the following: for a given value of $\bar{x}_0$, we need to approximately solve the above optimization problem as fast as possible, and of the obtained solution, it is the optimal value u_0 that we need fastest in order to provide the NMPC feedback. We might call the exact solution $u_0^*(\bar{x}_0)$ in order to express its dependence on the initial value $\bar{x}_0$. The only reason why we formulate and optimize the large optimization problem is because it delivers us this map $u_0^* : \mathbb{R}^{n_x} \to \mathbb{R}^{n_u}$, which is an approximation to the optimal feedback control.

Remark on fixed and free parameters: In most NMPC applications there are some *constant* parameters $\bar{p}$ that are assumed constant for the NMPC optimization, but that change for different problems, like $\bar{x}_0$. We do not regard them here for notational convenience, but note that they can be treated by state augmentation, i.e. regarded as constant system states with fixed initial value $\bar{p}$.

12.2 Nominal Stability of NMPC

Very often, one is interested in stabilizing the nonlinear dynamic system at a given set point for states and controls, which we might without loss of generality set to zero here. This steady state, that satisfies $f(0, 0, 0) = 0$, $g(0, 0, 0) = 0$ must be assumed to be feasible, i.e. $h(0, 0, 0) \leq 0$. One then often uses as stage cost the quadratic deviation from this set point, i.e., $L(x, u) = x^T Qx + u^T Ru$ with positive definite matrices Q, R. It is important to note that this function is positive definite, i.e., $L(0, 0) = 0$ and $L(x, u) > 0$ other wise. In this case, one would ideally like to solve the infinite horizon problem with $N = \infty$ in order to obtain the true stationary optimal feedback control; this would automatically ensure stability, as the value function $J(x)$ can be shown to decrease along the trajectory of the nominal system in each time step by $-L(x_0, u^*(x_0))$ and can thus serve as a Lyapunov function. But as we have in practice to choose a finite N, the question arises how we can ensure nominal stability of NMPC nevertheless. One way due to [52, 60] is to impose a *zero terminal constraint* i.e. to require $x_N = 0$ as terminal boundary condition (12.1f) in the NMPC problem and to employ no terminal cost, i.e. $E(x_N) = 0$.

In this case of a zero terminal constraint, it can be shown that the value function J_0 of the finite horizon problem is a Lyapunov function that decreases by at least $-L(\bar{x}_0, u^*(\bar{x}_0))$ in each time step. To prove this, let us assume that $(x_0^*, z_0^*, u_0^*, x_1^*, z_1^*, u_1^*, \ldots, x_N^*)$ is the solution of the NMPC problem (12.1a)-(12.1f) starting with initial value $\bar{x}_0$. After application of this feedback to the nominal system, i.e. without model-plant-mismatch, the system will evolve exactly as predicted, and for the next NMPC problem the initial value $\bar{x}_0'$ will be given by $\bar{x}_0' = x_1^*$. For this problem, the *shifted* version of the previous solution $(x_1^*, z_1^*, u_1^*, \ldots, x_N^*, 0, 0, 0)$ is a feasible point, and due to the zero values at the end, no additional cost arises at the end of the horizon. However, because the first stage cost term moved out of the horizon, we have that the cost of this feasible point of the next NMPC problem is reduced by exactly $-L(\bar{x}_0, u^*(\bar{x}_0))$. After further optimization, the cost can only be further reduced. Thus, we have proven that the value function J_0 is reduced along the trajectory, i.e. $J_0(\bar{x}_0') \leq J_0(\bar{x}_0) - L(\bar{x}_0, u^*(\bar{x}_0))$. More generally, one can relax the zero terminal constraint and construct combinations of terminal cost $E(x_N)$ and terminal inequalities $r(x_N) \leq 0$ that have the same property but are less restrictive, cf. e.g. [27, 29, 61].

12.3 Online Initialization via Shift

For exploiting the fact that NMPC requires the solution of a whole sequence of neighboring NLPs and not just a number of stand-alone problems, we have first the possibility to *initialize* subsequent problems efficiently based on previous information.

A first and obvious way to transfer solution information from one solved NMPC problem to the initialization of the next one is employing the shift that we used already in the proof of nominal stability above. It is motivated by the principle of optimality of subarcs, which, in our context, states the following: Let us assume we have computed an optimal solution $(x_0^*, z_0^*, u_0^*, x_1^*, z_1^*, u_1^*, \ldots, x_N^*)$ of the NMPC problem (12.1a)-(12.1f) starting with initial value $\bar{x}_0$. If we regard a shortened NMPC problem without the first interval, which starts with the initial value $\bar{x}_1$ chosen to be x_1^*, then for this shortened problem the vector $(x_1^*, z_1^*, u_1^*, \dots, x_N^*)$ is the optimal solution.

Based on the expectation that the measured or observed true initial value for the shortened NMPC problem differs not much from x_1^* – i.e. we believe our prediction model and expect no disturbances – this “shrinking” horizon initialization is canonical, and it is used in MPC of batch or finite time processes, see e.g. [47, 32].

However, in the case of moving horizon problems, the horizon is not only shortened by removing the first interval, but also prolonged at the end by appending a new terminal interval – i.e. the horizon is moved forward in time. In the moving horizon case, the principle of optimality is thus not strictly applicable, and we have to think about how to initialize the appended new variables z_N, u_N, x_{N+1}. Often, they are obtained by setting $u_N := u_{N-1}$ or setting u_N as the steady state control. The states z_N and x_{N+1} are then obtained by forward simulation. In the case that zero is the steady state and we had a zero terminal constraint, this would just result in zero values to be appended, as in the proof in the previous section. In any case, this transformation of the variables from one problem to the next is called “shift initialization”. It is not as canonical as the “shrinking horizon” case, because the shifted solution is not optimal for the new undisturbed problem. However, in the case of long horizon lengths N we can expect the shifted solution to be a good initial guess for the new solution. Moreover, for most NMPC schemes with stability guarantee (for an overview see e.g. [61]) there exists a canonical choice of u_N that implies feasibility (but not optimality) of the shifted solution for the new, undisturbed problem. The shift initialization is very often used e.g. in [58, 15, 62, 37].

A comparison of shifted vs. non-shifted initializations was performed in [20] with the result that for autonomous NMPC problems that shall regulate a system to steady state, there is usually no advantage of a shift initialization compared to the “primitive” warm start initialization that leaves the variables at the previous solution. In the extreme case of short horizon lengths, it turns out to be even advantageous NOT to shift the previous solution, as subsequent solutions are less dominated by the initial values than by the terminal conditions. On the other hand, shift initialization are a crucial prerequisite in periodic tracking applications [37] and whenever the system or cost function are not autonomous.

12.4 Outline of Real-Time Optimization Strategies

In NMPC we would dream to have the solution to a new optimal control problem instantly, which is impossible due to computational delays. Several ideas help us to deal with this issue.

Offline precomputations: As consecutive NMPC problems are similar, some computations can be done once and for all before the controller starts. In the extreme case, this leads to an explict precomputation of the NMPC control law that has raised much interest in the linear MPC community [6], or a solution of the Hamilton-Jacobi-Bellman Equation, both of which are prohibitive for state and parameter dimensions above ten. But also when online optimization is used, code optimization for the model routines is often essential, and it is in some cases even possible to precompute and factorize Hessians or even Jacobians in Newton type Optimization routines, in particular in the case of neighboring feedback control along reference trajectories [53, 26]. Also, pre-optimized compilable computer code can be auto-generated that is specific to the family of optimization problems, which is e.g. in convex optimization pursued in [59].

Delay compensation by prediction: When we know how long our computations for solving an NMPC problem will take, it is a good idea *not* to address a problem starting at the current state but to simulate at which state the system will be when we will have solved the problem. This can be done using the NMPC system model and the open-loop control inputs that we will apply in the meantime [41]. This feature is used in many practical NMPC schemes with non-negligible computation time.

Division into preparation and feedback phase: A third ingredient of several NMPC algorithms is to divide the computations in each sampling time into a preparation phase and a feedback phase [33]. The more CPU intensive preparation phase (a) is performed with an old predicted state $\bar{x}_0$ before the new state estimate, say $\bar{x}_0'$, is available, while the feedback phase (b) then delivers quickly an *approximate* solution to the optimization problem for $\bar{x}_0'$. Often, this approximation is based on one of the tangential predictors discussed in the next chapter.

Iterating while the problem changes: A fourth important ingredient of some NMPC algorithms is the idea to work on the optimization problem while it changes, i.e., to never iterate the Newton type procedure to convergence for an NMPC problem getting older and older during the iterations, but to rather work with the most current information in each new iteration. This idea is used in [58, 33, 64].

As a historical note, one of the first true online algorithms for nonlinear MPC was the *Newton-Type Controller of Li and Biegler* [57]. It is based on a sequential optimal control formulation, thus it iterates in the space of controls $u = (u_0, u_1, \ldots, u_{N-1})$ only. It uses an SQP type procedure with Gauss-Newton Hessian and line search, and in each sampling time, only one SQP iteration is performed. The transition from one problem to the next uses a shift of the controls $u^{\text{new}} = (u_1, \ldots, u_{N-1}, u_N^{\text{new}})$. The result of each SQP iterate is used to give an approximate feedback to the plant. As a sequential scheme without tangential predictor, it seems to have had sometimes problems with nonlinear convergence, though closed-loop stability was proven for open-loop stable processes [58].

In the next chapter, we will discuss several other real-time optimization algorithms in more detail that are all based on ideas from the field of parametric nonlinear optimization. s

Chapter 13

Parametric Nonlinear Optimization

In the shift initialization discussed in the previous chapter we did assume that the new initial value corresponds to the model prediction. This is of course never the case, because exactly the fact that the initial state is subject to disturbances motivates the use of MPC. By far the most important changes from one optimization problem to the next one are thus the unpredictable changes in the initial value $\bar{x}_0$. Is there anything we can do about this in the initialization of a new problem? It turns out that the concept of *parametric sensitivities* helps us here. In order to understand this concept, in this chapter we will regard the task of real-time optimization from a different perspective than before, namely from the point of view of *parametric optimization*, which is a subfield of nonlinear optimization [4, 46].

13.1 Parametric Nonlinear Optimization

The NMPC problem as stated in Equations (12.1a)-(12.1f) in the previous chapter is a specially structured case of a generic parametric nonlinear program (pNLP) with variables $Y = (x, z, u)$ that depends on the parameter $\bar{x}_0$. This pNLP has the form

$$\mathrm{pNLP}(\bar{x}_0): \qquad \underset{Y}{\text{minimize}}\ F(Y) \quad \text{s.t.} \quad \left\{ \begin{array}{rcl} G(\bar{x}_0, Y) & = & 0 \\ H(Y) & \leq & 0 \end{array} \right. \tag{13.1}$$

We recall that under mild assumptions, any locally optimal solution Y^* of this problem has to satisfy the Karush-Kuhn-Tucker (KKT) conditions: there exist multiplier vectors λ^* and μ^* so that the following equations hold:

$$\nabla_Y \mathcal{L}(Y^*, \lambda^*, \mu^*) = 0 \tag{13.2a}$$

$$G(\bar{x}_0, Y^*) = 0 \tag{13.2b}$$

$$0 \geq H(Y^*) \quad \perp \quad \mu^* \geq 0. \tag{13.2c}$$

Here we have used the definition of the Lagrange function

$$\mathcal{L}(Y, \lambda, \mu) = F(Y) + G(\bar{x}_0, Y)^T \lambda + H(Y)^T \mu \tag{13.3}$$

and the symbol $\perp$ between the two vector valued inequalities in Eq. (13.2c) states that also the complementarity condition

$$H_i(Y^*)\,\mu_i^* = 0, \quad i = 1, \ldots, n_H, \tag{13.4}$$

shall hold.

Remark on Initial Value Embedding: Due to the fact that the parameter $\bar{x}_0$ enters G linearly in our formulation, the Jacobian of G and thus also the Lagrange gradient does not depend on $\bar{x}_0$. We can therefore identify $\nabla_Y G(\bar{x}_0, Y) = \nabla_Y G(Y)$. The fact that all derivatives are independent of the parameter $\bar{x}_0$ will make the description of the path-following algorithms in the coming sections easier. Note that this particular formulation of the parameter dependence can in all parametric optimization problems be achieved by introducing the parameter x_0 as a variable and constraining it by a constraint $\bar{x}_0 - x_0 = 0$, as we have done in (12.1a)-(12.1f). We call this in the general case a *parameter embedding*. In the context of MPC, like here, we speak of the *initial value embedding* [30].

The primal-dual points $W = (Y, \lambda, \mu)$ that satisfy the KKT conditions for different values of $\bar{x}_0$ form the *solution manifold*; due to the non-smoothness of the complementarity condition, this manifold is in general not differentiable. However, if we would have no inequality constraints, the solution manifold is in general smooth, and we treat this case first.

13.2 Predictor-Corrector Pathfollowing Methods

In the equality constrained case, we have $W = (Y, \lambda)$, and the first two KKT conditions (13.2a)-(13.2b) form a nonlinear equation system depending on the parameter $\bar{x}_0$ that we can summarize as $R(\bar{x}_0, W) = 0$. The solution $W^*(\bar{x}_0)$ that satisfies these conditions for a given $\bar{x}_0$ is in general a smooth map; more precisely, it is smooth at all points at which the Jacobian $\frac{\partial R}{\partial W}$ is invertible. Note that this Jacobian is nothing else than the matrix that we called the *KKT matrix* in Chapter 2, and that the KKT matrix is invertible whenever the second order sufficient optimality conditions of Theorem 2.6 hold, which we can assume here. The derivative of the solution map $W^*(\bar{x}_0)$ is by the implicit function theorem given by

$$\frac{\partial W^*}{\partial \bar{x}_0}(\bar{x}_0) = -\left(\frac{\partial R}{\partial W}(\bar{x}_0, W^*(\bar{x}_0))\right)^{-1} \frac{\partial R}{\partial \bar{x}_0}(\bar{x}_0, W^*(\bar{x}_0)). \tag{13.5}$$

In the real-time optimization context, we might have solved a problem with parameter $\bar{x}_0$ with solution $W = W^*(\bar{x}_0)$ and want to solve next the problem for a new parameter $\bar{x}_0'$. The tangential predictor W' for this new solution $W^*(\bar{x}_0')$ is given by

$$W' = W + \frac{\partial W^*}{\partial \bar{x}_0}(\bar{x}_0)(\bar{x}_0' - \bar{x}_0) = W - \left(\frac{\partial R}{\partial W}(\bar{x}_0, W)\right)^{-1} \frac{\partial R}{\partial \bar{x}_0}(\bar{x}_0, W)(\bar{x}_0' - \bar{x}_0).$$

Note the similarity with one step of a Newton method. In fact, a combination of the tangential predictor and the corrector due to a Newton method proves to be useful in the case that W was not the exact solution of $R(\bar{x}_0, W) = 0$, but only an approximation. In this case, linearization at $(\bar{x}_0, W)$ yields a formula that one step of a *predictor-corrector pathfollowing method* needs to satisfy:

$$R(\bar{x}_0, W) + \frac{\partial R}{\partial \bar{x}_0}(\bar{x}_0, W)(\bar{x}_0' - \bar{x}_0) + \frac{\partial R}{\partial W}(\bar{x}_0, W)(W' - W) = 0. \tag{13.6}$$

Written explicitly, it delivers the solution guess W' for the next parameter $\bar{x}_0'$ as

$$W' = W \underbrace{- \left(\frac{\partial R}{\partial W}(\bar{x}_0, W)\right)^{-1} \frac{\partial R}{\partial \bar{x}_0}(\bar{x}_0, W)(\bar{x}_0' - \bar{x}_0)}_{=\Delta W_{\text{predictor}}} \underbrace{- \left(\frac{\partial R}{\partial W}(\bar{x}_0, W)\right)^{-1} R(\bar{x}_0, W)}_{=\Delta W_{\text{corrector}}}.$$

Structure due to Initial Value Embedding: We can use the fact that $\bar{x}_0$ enters R linearly due to the *initial value embedding* in order to simplify the formulae. First, we can omit the dependence of the derivatives on $\bar{x}_0$ and second, we can write $\frac{\partial R}{\partial \bar{x}_0}(W)(\bar{x}_0' - \bar{x}_0) = R(\bar{x}_0', W) - R(\bar{x}_0, W)$. Thus, the Equation (13.6) that the predictor-corrector step needs to satisfy simplifies to

$$R(\bar{x}_0', W) + \frac{\partial R}{\partial W}(W)(W' - W) = 0. \tag{13.7}$$

It follows that the predictor-corrector step can be easily obtained by just applying one standard Newton step to the new problem pNLP$(\bar{x}_0')$ initialized at the past solution guess W, if we employed the initial value embedding in the problem formulation. This is convenient in particular in the context of inequality constrained optimization.

In order to devise pathfollowing methods for the case of inequality constraints, there exist two different approaches. The first and easier one is closely related to nonlinear interior point (IP) methods and approximates the KKT system by a smooth equation, while the second one is related to sequential quadratic programming (SQP) methods and treats the non-smooth complementarity conditions in a different way.

13.3 Interior Point Pathfollowing Methods

Let us first recall that a *nonlinear interior point method* addresses the solution of the KKT system by replacing the last nonsmooth KKT condition in Eq. (13.2c) by a smooth nonlinear approximation, with $\tau > 0$:

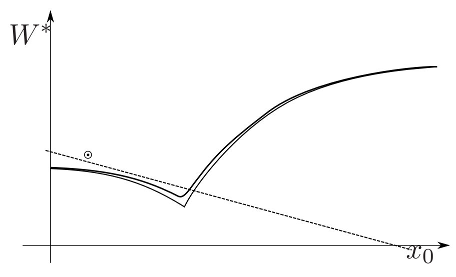
$$\begin{aligned} \nabla_Y \mathcal{L}(Y^*, \lambda^*, \mu^*) &= 0 & \text{(13.8a)} \\ G(\bar{x}_0, Y^*) &= 0 & \text{(13.8b)} \\ H_i(Y^*)\, \mu_i^* + \tau &= 0, \quad i = 1, \ldots, n_H. & \text{(13.8c)} \end{aligned}$$

If we regard this system for a fixed parameter τ, it is just a nonlinear equation that determines the unknowns $W = (Y, \lambda, \mu)$ and depends on the parameter $\bar{x}_0$, and which we summarize again as

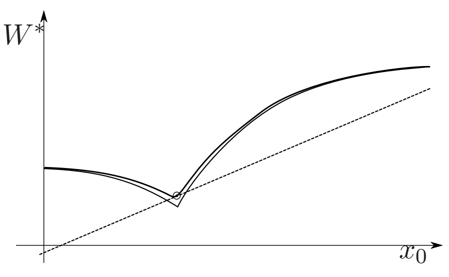
$$R(\bar{x}_0, W) = 0. \tag{13.9}$$

This equation system implicitly defines the smooth *interior point (IP) solution manifold* $W^*(\bar{x}_0)$ in which we are interested in the real-time optimization context. As it is a smooth equation, we can in principle apply the pathfollowing predictor-corrector method of the previous section. For decreasing τ, this IP solution manifold approximates closer and closer the true solution manifold of the parametric NLP.

Remark on IP Sensitivities at Active Set Changes: Unfortunately, for small τ, the interior point solution manifold is strongly nonlinear at points where the active set changes, and the tangential predictor is not a good approximation when we linearize at such points, as visualized in Fig. 13.1(b). One remedy would be to increase the path parameter τ, which decreases the nonlinearity, but comes at the expense of generally less accurate solutions. This is illustrated in Figs. 13.2(a) and 13.2(b) for the same two linearization points as before. In Fig. 13.2(b) we see that the tangent is approximating the IP solution manifold well in a larger area around the linearization point, but that the IP solution itself is more distant to the true NLP solution. Thus, the tangential predictor is of limited use across active set changes.

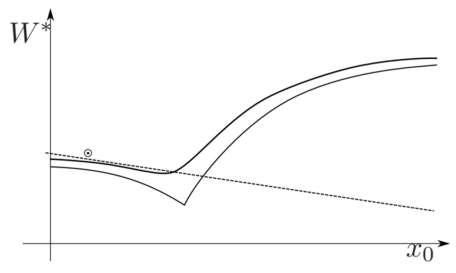


(a) Linearizing at approximate solution

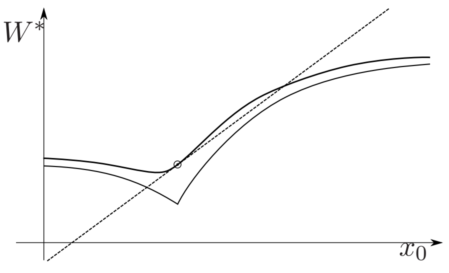


(b) Linearizing at active set change

Figure 13.1: Tangential predictors for interior point method using a small τ.



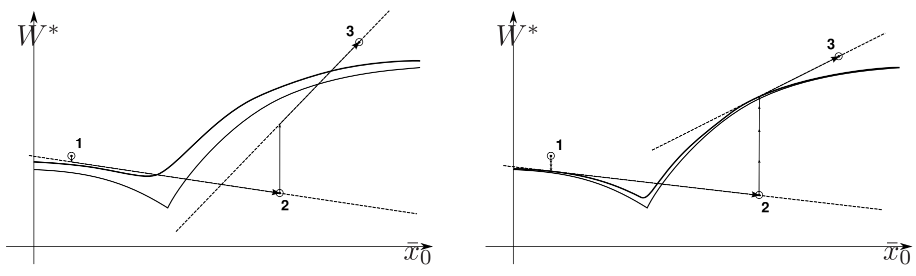
(a) Linearizing at approximate solution



(b) Linearizing at active set change

Figure 13.2: Tangential predictors for interior point method using a larger τ.

The Continuation/GMRES Method of Ohtsuka [64]: The Continuation/GMRES method performs one predictor-corrector Newton type iteration in each sampling time, and is based on a sequential formulation. It is based on an IP treatment of the inequalities with fixed path parameter $\tau > 0$ it uses an exact Hessian, andmit uses the iterative GMRES method for linear system solution in each Newton step. Most important, it makes use of the tangential predictor described in Eq. (13.7). This features seems to allow it to follow the nonlinear IP solution manifold well – which is strongly curved at active set changes. For a visualization, see Fig. 13.3(a). In each sampling time, only one linear system is built and solved by the GMRES method, leading to a predictor-corrector pathfollowing method. The closed-loop stability of the method is in principle covered by the stability analysis for the real-time iterations without shift given in [35]. A variant of the method is given in [72], which uses a simultanous approach and condensing and leads to improved accuracy and lower computational cost in each Newton type iteration.



(a) Ohtsuka's C/GMRES method (b) Advanced Step Controller

Figure 13.3: Subsequent solution approximations.

Advanced Step Controller by Zavala and Biegler [81]: In order to avoid the convergence issues of predictor-corrector pathfollowing methods, in the "advanced step controller" of Zavala and Biegler a more conservative choice is made: in each sampling time, a complete Newton type IP procedure is iterated to convergence (with $\tau \to 0$). In this respect, it is just like offline optimal control – IP, simultaneous, full derivatives with exact Hessian, structure exploiting linear algebra. However, two features qualify it as an online algorithm: first, it takes computational delay into account by solving an "advanced" problem with the expected state $\bar{x}_0$ as initial value – similar as in the real-time iterations with shift – and (b), it applies the obtained solution not directly, but computes first the tangential predictor which is correcting for the differences between expected state $\bar{x}_0$ and the actual state $\bar{x}_0'$, as described in Eq. (13.7) with $R(W, \bar{x}_0) = 0$. Note that in contrast to the other online algorithms, several Newton iterations are performed in part (a) of each sampling time, the "preparation phase". The tangential predictor (b) is computed in the "feedback phase" by only one linear system solve based on the last Newton iteration's matrix factorization. As in the C/GMRES method, the IP predictor cannot "jump over" active set changes as easily as the SQP based predictor of the real-time iteration. Roughly speaking, the advanced step controller gives lower priority to sudden active set changes than to system nonlinearity. As the advanced step controller solves each expected problem exactly, classical NMPC stability theory [61] can relatively easily be extended to this scheme [81].

13.4 SQP Pathfollowing Methods

In fact, if inequalities are present, the true NLP solution is not determined by a smooth root finding problem (13.8a)–(13.8c), but by the KKT conditions (13.2a)–(13.2c). It is a well-known fact from parametric optimization, cf. [46], that the solution manifold has smooth parts when the active set does not change (and bifurcations are excluded), but that non-differentiable points occur whenever the active set changes. Is there anything we can do in order to "jump" over these non-smooth points in a way that delivers better predictors than the IP predictors discussed before?

At points with weakly active constraints, we have to regard *directional* derivatives of the solution manifold, or "generalized tangential predictors". These can be computed by suitable quadratic programs [46, Thm 3.3.4] and are visualized in Fig. 13.4(b). The theoretical results can be made a practical algorithm by the procedure proposed in [30]: first, we have to make sure that the parameter $\bar{x}_0$ enters the NLP linearly, via the initial value embedding, cf. Eq. (12.1b). Second, we address the problem with an exact Hessian SQP method. Third, we just take our current solution guess W^k for a problem $\bar{x}_0$, and then solve a parametric QP subproblem

$$\mathrm{pQP}(\bar{x}_0', W^k): \qquad \underset{Y}{\text{minimize}}\; F_{\mathrm{QP}}^k(Y) \quad \text{s.t.} \quad \left\{ \begin{array}{rcl} G(\bar{x}_0', Y^k) + \nabla G(Y^k)^T (Y - Y^k) & = & 0 \\ H(Y^k) + \nabla H(Y^k)^T (Y - Y^k) & \leq & 0 \end{array} \right. \tag{13.10}$$

with objective function

$$F_{\mathrm{QP}}^k(Y) = \nabla F(Y^k)^T Y + \frac{1}{2}(Y - Y^k)^T \nabla_Y^2 \mathcal{L}(Y^k, \lambda^k, \mu^k)(Y - Y^k). \tag{13.11}$$

for the new parameter value $\bar{x}_0'$, but initialized at W^k. It can be shown [30, Thm. 3.6] that this "initial value embedding" procedure delivers exactly the generalized tangential predictor when started at a solution $W^k = W^*(\bar{x}_0)$, as in Fig. 13.4(b). It is important to remark that the predictor becomes approximately tangential when (a) we do not start on the solution manifold, see Fig. 13.4(a), or (b) we do not use an exact Hessian or Jacobian matrix. In practical NMPC applications, very often a Gauss-Newton Hessian provides an excellent positive definite approximation of the Hessian.

Condensing: Let us recall that the states can be eliminated from the above parametric QP, resulting in a smaller, *condensed* quadratic program of the form

$$\mathrm{pQPcond}(\bar{x}_0', W^k): \qquad \underset{u}{\text{minimize}} \qquad f_{\mathrm{condQP},k}(\bar{x}_0', u) \tag{13.12a}$$

$$\text{subject to} \qquad \bar{r}_k + \bar{R}_k^{x_0} \bar{x}_0 + \bar{R}_k^u u \;\leq\; 0. \tag{13.12b}$$

If the dimension of the vector $u = (u_0^T, u_1^T, \ldots, u_{N-1}^T)^T$ is not too large, this QP can be solved fast using dense general purpose QP solvers. The importance of the condensed QP in the real-time optimization context is that it can very quickly be solved but still contains the explicit dependence on the parameter $\bar{x}_0$ as well as the controls, in particular the first one, u_0, which we need for the next MPC feedback.

The Real-Time Iteration Scheme [33]: Based on the above ideas, the real-time iteration scheme presented in [30, 33] performs one SQP type iteration with Gauss-Newton Hessian per sampling time. However, it employs a simultaneous NLP parameterization, Bock's direct multiple shooting method, with full derivatives and condensing. Moreover, it uses the generalized tangential predictor of the "initial value embedding" to correct for the mismatch between the expected state $\bar{x}_0$ and the actual state $\bar{x}_0'$. In contrast to the C/GMRES method by Ohtsuka, where the predictor is based on one linear system solve from Eq. (13.7), here an inequality constrained QP is solved. The computations in each iteration are divided into a long "preparation phase" (a), in which the system linearization and condensing are performed, and a much shorter "feedback phase" (b), see the visualization in Fig. 13.5. The feedback phase solves just one condensed QP (13.12a)–(13.12b). Depending on the application, the feedback phase can be several orders of magnitude shorter than the feedback phase. The iterates of the scheme are visualized in Fig. 13.6(a). The same iterates are obtained with a variant of the scheme that uses Schlöder's trick for reducing the costs of the preparation phase in the case of large state dimensions [69]. Note that only one system linearization and one QP solution are performed in each sampling time, and that the QP corresponds to a linear MPC feedback along a time varying trajectory. In contrast to IP formulations, the real-time iteration scheme gives priority to active set changes and works well when the active set changes faster than the linearized system matrices. In the limiting case of a linear system model it gives the same feedback as linear MPC. Error bounds and closed loop stability of the scheme have been established for shrinking horizon problems in [32] and for NMPC with shifted and non-shifted initializations in [36] and [35].

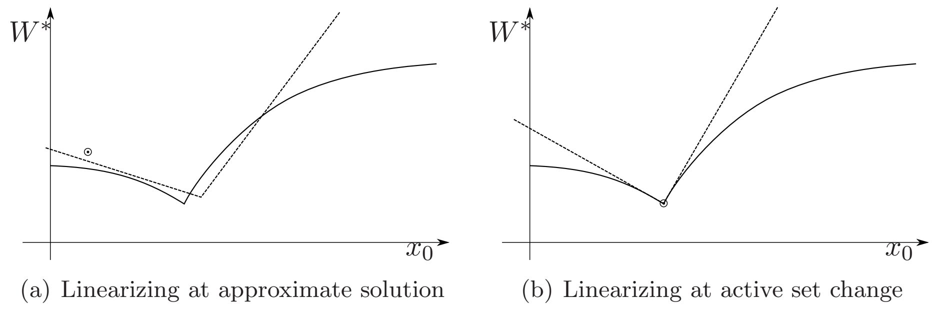


Figure 13.4: Generalized tangential predictors for SQP method.

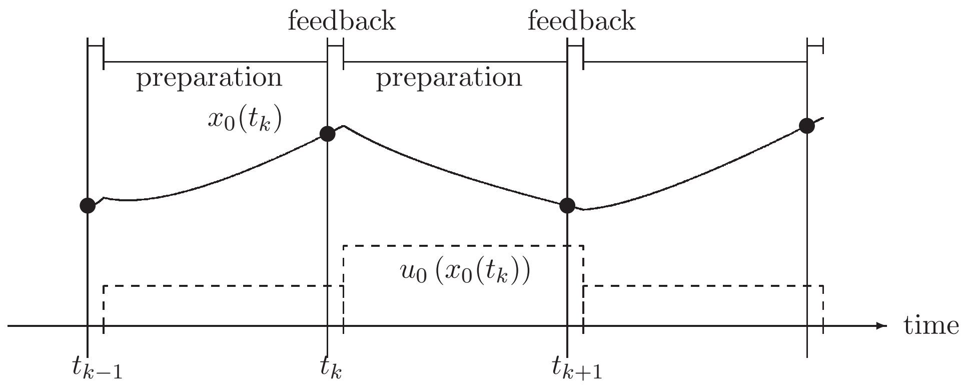


Figure 13.5: Division of one real-time iteration into preparation and feedback phase.

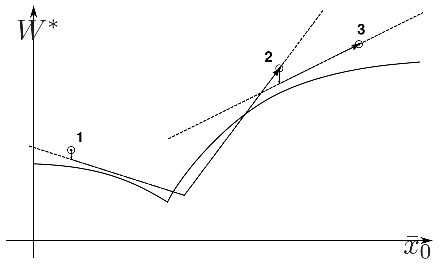
Adjoint-Based Multi-Level Real-Time Iterations [19]: A variant of real-time iterations was presented in [19], where even cheaper calculations are performed in each sampling time than one Newton or Gauss-Newton step usually requires. Within the *Adjoint-Based Multi-Level Real-Time Iterations*, at the lowest level (A), only one condensed QP (13.12a)–(13.12b) is solved, for the most current initial value $\bar{x}_0$. This provides a form of linear MPC at the base level, taking at least active set changes into account with a very high sampling frequency. On the next two intermediate levels, that are performed less often than every sampling time, only the nonlinear constraint residuals are evaluated (B), allowing for feasibility improvement, cf. also [26], or the Lagrange gradient is evaluated (C), allowing for optimality improvement. This level C is based on the following QP with inexact matrices

$$\underset{Y}{\text{minimize}}\ F^k_{\text{adjQP}}(Y) \quad \text{s.t.} \quad \left\{ \begin{array}{rcl} G(\bar{x}_0', Y^k) + B_k^T(Y - Y^k) & = & 0 \\ H(Y^k) + C_k^T(Y - Y^k) & \leq & 0. \end{array} \right. \tag{13.13}$$

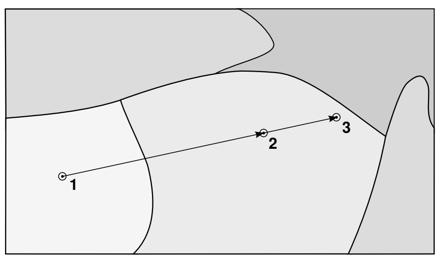
with the QP objective

$$F^k_{\text{adjQP}}(Y) = Y^T \underbrace{\left(\nabla_Y \mathcal{L}(Y^k, \lambda^k, \mu^k) - B_k\lambda^k - C_k\mu^k\right)}_{\text{"modified gradient"}} + \frac{1}{2}(Y - Y^k)^T A_k (Y - Y^k). \tag{13.14}$$

A crucial ingredient of this level is the fact that the Lagrange gradient can be evaluated efficiently by the reverse mode of automatic differentiation. Note that in all three levels A, B, and C mentioned so far, no new QP matrices are computed and that even system factorizations can be reused again and again. Level C iterations are still considerably cheaper than one full SQP iteration [79], but also for them optimality and NMPC closed-loop stability can be guaranteed by the results in [35] – as long as the system matrices are accurate enough to guarantee Newton type contraction. Only when this is not the case anymore, an iteration on the highest level, D, has to be performed, which includes a full system linearization and is as costly as a usual Newton type iteration.



(a) Real-Time Iteration scheme



(b) Critical regions of a parametric NLP

Figure 13.6: Subsequent solution approximations (left), and critical regions (right).

13.5 Critical Regions and Online Active Set Strategies

It is interesting to have a look at the parameter space $\bar{x}_0$ visualized in Fig.13.6(b). The picture shows the "critical regions" on each of which the active set in the solution is stable. It also shows three consecutive problems on a line that correspond to the scenario used in Figures 13.3(a), 13.6(a), and 13.3(b). Between problem 1 and 2 there is one active set change, while problems 2 and 3 have the same active set, i.e., are in the same critical region. The C/GMRES method and Advanced Step Controller exploit the smoothness on each critical region in order to obtain the conventional Newton predictor that, however, looses validity when a region boundary is crossed. The real-time iteration basically "linearizes" the critical regions which then become polytopic, by using the more accurate, but also more expensive QP predictor.

As the QP cost can become non-negligible for fast MPC applications, a so-called online active set strategy was proposed in [39]. This strategy goes on a straight line in the space of linearized regions from the old to the new QP problem. As long as one stays within one critical region, the QP solution depends affinely on $\bar{x}_0$ – exactly as the conventional Newton predictor. Only if the homotopy crosses boundaries of critical regions, the active set is updated accordingly. The online active set strategy is available in the open-source QP package qpOASES [40], and is particularly suitable in combination with real-time iterations of level A, B, and C, where the QP matrices do not change, see [80].

Bibliography

[1] ACADO Toolkit Homepage. http://www.acadotoolkit.org, 2009–2011.

[2] J. Albersmeyer and M. Diehl. The Lifted Newton Method and its Application in Optimization. SIAM Journal on Optimization, 20(3):1655–1684, 2010.

[3] U.M. Ascher and L.R. Petzold. Computer Methods for Ordinary Differential Equations and Differential–Algebraic Equations. SIAM, Philadelphia, 1998.

[4] B. Bank, J. Guddat, D. Klatte, B. Kummer, and K. Tammer. Non-Linear Parametric Optimization. Birkhauser Verlag, 1983.

[5] R. Bellman. Dynamic programming. Dover, 1965.

[6] A. Bemporad, F. Borrelli, and M. Morari. Model Predictive Control Based on Linear Programming - The Explicit Solution. IEEE Transactions on Automatic Control, 47(12):1974–1985, 2002.

[7] A. Bemporad, F. Borrelli, and M. Morari. Min-max Control of Constrained Uncertain Discrete-Time Linear Systems. IEEE Transactions on Automatic Control, 2003. in press.

[8] A. Ben-Tal and A. Nemirovski. Lectures on Modern Convex Optimization: Analysis, Algorithms, and Engineering Applications, volume 3 of MPS/SIAM Series on Optimization. SIAM, 2001.

[9] D. Bertsekas. Dynamic Programming and Optimal Control, volume 1. Athena Scientific, 3rd edition, 2005.

[10] D.P. Bertsekas. Dynamic Programming and Optimal Control, volume 1 and 2. Athena Scientific, Belmont, MA, 1995.

[11] D.P. Bertsekas and J.N. Tsitsiklis. Neuro-Dynamic Programming. Athena Scientific, Belmont, MA, 1996.

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

[13] Lorenz T. Biegler. Nonlinear Programming. MOS-SIAM Series on Optimization. SIAM, 2010.

[14] L.T. Biegler. Solution of dynamic optimization problems by successive quadratic programming and orthogonal collocation. Computers and Chemical Engineering, 8:243–248, 1984.

[15] L.T. Biegler and J.B Rawlings. Optimization approaches to nonlinear model predictive control. In W.H. Ray and Y. Arkun, editors, Proc. 4th International Conference on Chemical Process Control - CPC IV, pages 543–571. AIChE, CACHE, 1991.

[16] J. Björnberg and M. Diehl. Approximate robust dynamic programming and robustly stable MPC. Automatica, 42(5):777–782, May 2006.

[17] J. Björnberg and M. Diehl. Approximate Dynamic Programming for Generation of Robustly Stable Feedback Controllers. In Modeling, Simulation and Optimization of Complex Processes. International Conference on High Performance Scientific Computing, pages 69–86, Heidelberg, 2008. Springer.

[18] H.G. Bock. Randwertproblemmethoden zur Parameteridentifizierung in Systemen nichtlinearer Differentialgleichungen, volume 183 of Bonner Mathematische Schriften. Universität Bonn, Bonn, 1987.

[19] H.G. Bock, M. Diehl, E.A. Kostina, and J.P. Schlöder. Constrained Optimal Feedback Control of Systems Governed by Large Differential Algebraic Equations. In L. Biegler, O. Ghattas, M. Heinkenschloss, D. Keyes, and B. van Bloemen Waanders, editors, Real-Time and Online PDE-Constrained Optimization, pages 3 – 22. SIAM, 2007.

[20] H.G. Bock, M. Diehl, D.B. Leineweber, and J.P. Schlöder. Efficient direct multiple shooting in nonlinear model predictive control. In F. Keil, W. Mackens, H. Voß, and J. Werther, editors, Scientific Computing in Chemical Engineering II, volume 2, pages 218–227, Berlin, 1999. Springer.

[21] H.G. Bock and K.J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. In Proceedings 9th IFAC World Congress Budapest, pages 243–247. Pergamon Press, 1984.

[22] S. Boyd and L. Vandenberghe. Convex Optimization. University Press, Cambridge, 2004.

[23] K.E. Brenan, S.L. Campbell, and L.R. Petzold. The Numerical Solution of Initial Value Problems in Ordinary Differential-Algebraic Equations. North Holland Publishing Co., Amsterdam, 1989.

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

[25] A.E. Bryson and Y.-C. Ho. Applied Optimal Control. Wiley, New York, 1975.

[26] C. Büskens and H. Maurer. SQP-methods for solving optimal control problems with control and state constraints: adjoint variables, sensitivity analysis and real-time control. Journal of Computational and Applied Mathematics, 120:85–108, 2000.

[27] H. Chen and F. Allgöwer. A quasi-infinite horizon nonlinear model predictive control scheme with guaranteed stability. Automatica, 34(10):1205–1218, 1998.

[28] G. B. Dantzig. Linear Programming and Extensions. Princeton University Press, 1963.

[29] G. De Nicolao, L. Magni, and R. Scattolini. Stability and Robustness of Nonlinear Receding Horizon Control. In F. Allgöwer and A. Zheng, editors, Nonlinear Predictive Control, volume 26 of Progress in Systems Theory, pages 3–23, Basel Boston Berlin, 2000. Birkhäuser.

[30] M. Diehl. Real-Time Optimization for Large Scale Nonlinear Processes, volume 920 of Fortschr.-Ber. VDI Reihe 8, Meß-, Steuerungs- und Regelungstechnik. VDI Verlag, Düsseldorf, 2002. Download also at: http://www.ub.uni-heidelberg.de/archiv/1659/.

[31] M. Diehl and J. Björnberg. Robust Dynamic Programming for Min-Max Model Predictive Control of Constrained Uncertain Systems. IEEE Transactions on Automatic Control, 49(12):2253–2257, December 2004.

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

[33] 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. Contr., 12(4):577–585, 2002.

[34] M. Diehl, H. J. Ferreau, and N. Haverbeke. Nonlinear model predictive control, volume 384 of Lecture Notes in Control and Information Sciences, chapter Efficient Numerical Methods for Nonlinear MPC and Moving Horizon Estimation, pages 391–417. Springer, 2009.

[35] M. Diehl, R. Findeisen, and F. Allgöwer. A Stabilizing Real-time Implementation of Nonlinear Model Predictive Control. In L. Biegler, O. Ghattas, M. Heinkenschloss, D. Keyes, and B. van Bloemen Waanders, editors, Real-Time and Online PDE-Constrained Optimization, pages 23–52. SIAM, 2007.

[36] M. Diehl, R. Findeisen, F. Allgöwer, H.G. Bock, and J.P. Schlöder. Nominal Stability of the Real-Time Iteration Scheme for Nonlinear Model Predictive Control. IEE Proc.-Control Theory Appl., 152(3):296–308, 2005.

[37] M. Diehl, L. Magni, and G. De Nicolao. Online NMPC of unstable periodic systems using approximate infinite horizon closed loop costing. IFAC Annual Reviews in Control, 28:37–45, 2004.

[38] Moritz Diehl, Hans Georg Bock, Holger Diedam, and Pierre-Brice Wieber. Fast Direct Multiple Shooting Algorithms for Optimal Robot Control. Lecture Notes in Control and Information Sciences, 340, 2006.

[39] H. J. Ferreau, H. G. Bock, and M. Diehl. An online active set strategy to overcome the limitations of explicit MPC. International Journal of Robust and Nonlinear Control, 18(8):816–830, 2008.

[40] H.J. Ferreau. qpOASES User's Manual, 2007–2011. http://www.qpOASES.org/.

[41] R. Findeisen and F. Allgöwer. Computational Delay in Nonlinear Model Predictive Control. Proc. Int. Symp. Adv. Control of Chemical Processes, ADCHEM, 2003.

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

[43] P.E. Gill, W. Murray, and M.A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. Technical report, Numerical Analysis Report 97-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 1997.

[44] A. Griewank and Ph.L. Toint. Partitioned variable metric updates for large structured optimization problems. Numerische Mathematik, 39:119–137, 1982.

[45] A. Griewank and A. Walther. Evaluating Derivatives. SIAM, 2008.

[46] J. Guddat, F. Guerra Vasquez, and H.T. Jongen. Parametric Optimization: Singularities, Pathfollowing and Jumps. Teubner, Stuttgart, 1990.

[47] A. Helbig, O. Abel, and W. Marquardt. Model Predictive Control for On-line Optimization of Semi-batch Reactors. In Proc. Amer. Contr. Conf., pages 1695–1699, Philadelphia, 1998.

[48] G.A. Hicks and W.H. Ray. Approximation methods for optimal control systems. Can. J. Chem. Engng., 49:522–528, 1971.

[49] B. Houska, H.J. Ferreau, and M. Diehl. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. Optimal Control Applications and Methods, 2011. DOI: 10.1002/oca.939 (in print).

[50] C.N. Jones and M. Morari. Polytopic approximation of explicit model predictive controllers. Automatic Control, IEEE Transactions on, 55(11):2542–2553, 2010.

[51] W. Karush. Minima of functions of several variables with inequalities as side conditions. Master's thesis, Department of Mathematics, University of Chicago, 1939.

[52] S.S. Keerthi and E.G. Gilbert. Optimal infinite-horizon feedback laws for a general class of constrained discrete-time systems: Stability and moving-horizon approximations. Journal of Optimization Theory and Applications, 57(2):265–293, 1988.

[53] P. Krämer-Eis and H.G. Bock. Numerical Treatment of State and Control Constraints in the Computation of Feedback Laws for Nonlinear Control Problems. In P. Deuflhard et al., editor, Large Scale Scientific Computing, pages 287–306. Birkhäuser, Basel Boston Berlin, 1987.

[54] H.W. Kuhn and A.W. Tucker. Nonlinear programming. In J. Neyman, editor, Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, 1951. University of California Press.

[55] 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). Computers and Chemical Engineering, 27:157–174, 2003.

[56] D.B. Leineweber, 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. Part II: Software Aspects and Applications. Computers and Chemical Engineering, 27:167–174, 2003.

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

[58] W.C. Li and L.T. Biegler. Newton-Type Controllers for Constrained Nonlinear Processes with Uncertainty. Industrial and Engineering Chemistry Research, 29:1647–1657, 1990.

[59] J. Mattingley, Y. Wang, and Stephen Boyd. Code generation for receding horizon control. In Proceedings of the IEEE International Symposium on Computer-Aided Control System Design, Yokohama, Japan, 2010.

[60] D. Q. Mayne and H. Michalska. Receding horizon control of nonlinear systems. IEEE Transactions on Automatic Control, 35(7):814–824, 1990.

[61] D.Q. Mayne, J.B. Rawlings, C.V. Rao, and P.O.M. Scokaert. Constrained model predictive control: stability and optimality. Automatica, 26(6):789–814, 2000.

[62] A. M'hamdi, A. Helbig, O. Abel, and W. Marquardt. Newton-type Receding Horizon Control and State Estimation. In Proc. 13rd IFAC World Congress, pages 121–126, San Francisco, 1996.

[63] J. Nocedal and S.J. Wright. Numerical Optimization. Springer Series in Operations Research and Financial Engineering. Springer, 2 edition, 2006.

[64] T. Ohtsuka. A Continuation/GMRES Method for Fast Computation of Nonlinear Receding Horizon Control. Automatica, 40(4):563–574, 2004.

[65] M.R. Osborne. On shooting methods for boundary value problems. Journal of Mathematical Analysis and Applications, 27:417–433, 1969.

[66] E. N. Pistikopoulos, V. Dua, N. A. Bozinis, A. Bemporad, and M. Morari. On-line optimization via off-line parametric optimization tools. Computers and Chemical Engineering, 24:183–188, 2000.

[67] L.S. Pontryagin, V.G. Boltyanski, R.V. Gamkrelidze, and E.F. Miscenko. The Mathematical Theory of Optimal Processes. Wiley, Chichester, 1962.

[68] R.W.H. Sargent and G.R. Sullivan. The development of an efficient optimal control package. In J. Stoer, editor, Proceedings of the 8th IFIP Conference on Optimization Techniques (1977), Part 2, Heidelberg, 1978. Springer.

[69] A. Schäfer, P. Kühl, M. Diehl, J.P. Schlöder, and H.G. Bock. Fast reduced multiple shooting methods for Nonlinear Model Predictive Control. Chemical Engineering and Processing, 46(11):1200–1214, 2007.

[70] A.A.S. Schäfer. Efficient reduced Newton-type methods for solution of large-scale structured optimization problems with application to biological and chemical processes. PhD thesis, Universität Heidelberg, 2005.

[71] J.P. Schlöder. Numerische Methoden zur Behandlung hochdimensionaler Aufgaben der Parameteridentifizierung, volume 187 of Bonner Mathematische Schriften. Universität Bonn, Bonn, 1988.

[72] Y. Shimizu, T. Ohtsuka, and M. Diehl. A Real-Time Algorithm for Nonlinear Receding Horizon Control Using Multiple Shooting and Continuation/Krylov Method. International Journal of Robust and Nonlinear Control, 19:919–936, 2009.

[73] M.C. Steinbach. A structured interior point SQP method for nonlinear optimal control problems. In R. Bulirsch and D. Kraft, editors, Computation Optimal Control, pages 213–222, Basel Boston Berlin, 1994. Birkhäuser.

[74] M.J. Tenny, S.J. Wright, and J.B. Rawlings. Nonlinear model predictive control via feasibility-perturbed sequential quadratic programming. Computational Optimization and Applications, 28(1):87–121, April 2004.

[75] T.H. Tsang, D.M. Himmelblau, and T.F. Edgar. Optimal control via collocation and non-linear programming. International Journal on Control, 21:763–768, 1975.

[76] A. Wächter. An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering. PhD thesis, Carnegie Mellon University, 2002.

[77] A. Wächter and L. Biegler. IPOPT - an Interior Point OPTimizer. https://projects.coin-or.org/Ipopt, 2009.

[78] A. Wächter and L.T. Biegler. On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. Mathematical Programming, 106:25–57, 2006.

[79] L. Wirsching. An SQP Algorithm with Inexact Derivatives for a Direct Multiple Shooting Method for Optimal Control Problems. Master's thesis, University of Heidelberg, 2006.

[80] L. Wirsching, H.J. Ferreau, H.G. Bock, and M. Diehl. An Online Active Set Strategy for Fast Adjoint Based Nonlinear Model Predictive Control. In Preprints of the 7th Symposium on Nonlinear Control Systems (NOLCOS), Pretoria, 2007.

[81] V. M. Zavala and L.T. Biegler. The Advanced Step NMPC Controller: Optimality, Stability and Robustness. Automatica, 45:86–93, 2009.