Lecture Notes on Optimal Estimation and Control

Moritz Diehl

May 26, 2014

Contents

Chapter 1

Introduction

Optimal control regards the *optimization* of *dynamic systems*. In this lecture we identify dynamic systems with processes that are evolving with time and that can be characterized by *states* x that allow us to predict the future behavior of the system. If the state is not known, we first need to *estimate* it based on the available measurement information. The estimation process is very often optimization-based and uses the same optimization methods that are used for optimal control. This is the reason why this lecture bundles both optimal control and estimation in one single course. Often, a dynamic system can be controlled by a suitable choice of inputs that we denote as *controls* u in this script. In optimal control, these controls shall be chosen optimally in order to optimize some *objective function* and respect some *constraints*.

For optimal control, we might think of an electric train where the state x consists of the current position and velocity, and where the control u is the engine power that the train driver can choose at each moment. We might regard the motion of the train on a time interval $[t_{\mathrm{init}}, t_{\mathrm{fin}}]$, and the objective could be to minimize the consumed energy to drive from Station A to Station B, and one of the constraints would be that the train should arrive in Station B at the fixed final time, t_{fin}.

On the other hand, in optimization-based estimation, we treat unknown disturbances as inputs, and the objective function is the misfit between the actual measurements and their model predictions. The resulting optimization problems are mathematically of the same form as the problems of optimal control, with the disturbances as controls. For this reason, we focus the larger part of the course on the topic "optimal control". At several occasions we specialize to estimation problems as a special case.

A typical property of a dynamic system is that knowledge of an *initial state* x_{init} and a *control input trajectory* $u(t)$ for all $t \in [t_{\mathrm{init}}, t_{\mathrm{fin}}]$ allows one to determine the whole *state trajectory* $x(t)$ for $t \in [t_{\mathrm{init}}, t_{\mathrm{fin}}]$. As the motion of a train can very well be modelled by Newton's laws of motion, the usual description of this dynamic system is deterministic and in continuous time and with continuous states.

But dynamic systems and their mathematical models can come in many variants, and it is useful to properly define the names given commonly to different dynamic system classes, which we do in the next section. Afterwards, we will discuss two important classes, continuous time and discrete time systems, in more mathematical

detail, before we give an overview of optimization problem classes and finally outline the contents of the lecture chapter by chapter.

1.1 Dynamic System Classes

In this section, let us go, one by one, through the many dividing lines in the field of dynamic systems.

Continuous vs Discrete Time Systems

Any dynamic system evolves over time, but time can come in two variants: while the physical time is continuous and forms the natural setting for most technical and biological systems, other dynamic systems can best be modelled in discrete time, such as digitally controlled sampled-data systems, or games.

We call a system a *discrete time system* whenever the time in which the system evolves only takes values on a predefined time grid, usually assumed to be integers. If we have an interval of real numbers, like for the physical time, we call it a *continuous time system*. In this lecture, we usually denote the continuous time by the variable $t \in \mathbb{R}$ and write for example $x(t)$. In case of discrete time systems, we use an index, usually $k \in \mathbb{N}$, and write x_k for the state at time point k.

Continuous vs Discrete State Spaces

Another crucial element of a dynamic system is its state x, which often lives in a continuous state space, like the position of the train, but can also be discrete, like the position of the figures on a chess game. We define the *state space* $\mathbb{X}$ to be the set of all values that the state vector x may take. If $\mathbb{X}$ is a subset of a real vector space such as $\mathbb{R}^{n_x}$ or another differentiable manifold, we speak of a *continuous state space*. If $\mathbb{X}$ is a finite or a countable set, we speak of a *discrete state space*. If the state of a system is described by a combination of discrete and continuous variables we speak of a *hybrid state space*.

A *multi-stage system* is the special case of a system with hybrid state space that develops through a sequence of stages and where the state space on each stage is continuous. An example for a multi-stage system is walking, where consecutive stages are characterized by the number of feet that are on the ground at a given moment. For multi-stage systems, the time instant when one stage ends and the next one starts can often be described by a *switching function*. This function is positive on one and negative on the other stage, and assumes the value zero at the time instant that separates the stages.

Another special case are systems that develop in a continuous state space and in continuous time, but are sometimes subject to discontinuous jumps, such as bouncing billiard balls. These can often be modelled as multi-stage systems with switching functions, plus so called *jump conditions* that describe the discontinuous state evolution at the time instant between the stages.

Finite vs Infinite Dimensional State Spaces

The class of continuous state spaces can be further subdivided into the finite dimensional ones, whose state can be characterized by a finite set of real numbers, and the infinite dimensional ones, which have a state that lives in function spaces. The evolution of finite dimensional systems in continuous time is usually described by *ordinary differential equations (ODE)* or their generalizations, such as *differential algebraic equations (DAE)*.

Infinite dimensional systems are sometimes also called *distributed parameter systems*, and in the continuous time case, their behaviour is typically described by *partial differential equations (PDE)*. An example for a controlled infinite dimensional system is the evolution of the airflow and temperature distribution in a building that is controlled by an air-conditioning system.

Continuous vs Discrete Control Sets

We denote by $\mathbb{U}$ the set in which the controls u live, and exactly as for the states, we can divide the possible control sets into *continuous control sets* and *discrete control sets*. A mixture of both is a *hybrid control set*. An example for a discrete control set is the set of gear choices for a car, or any switch that we can can choose to be either on or off, but nothing in between.

In the systems and control community, the term *hybrid system* denotes a dynamic system which has either a hybrid state or hybrid control space, or both. Generally speaking, hybrid systems are more difficult to optimize than systems with continuous control and state spaces.

However, an interesting and relevant class are hybrid systems that have continuous time and continuous states, but discrete controls. They might be called hybrid systems with *external switches* or *integer controls* and turn out to be tremendously easier to optimize than other forms of hybrid systems, if treated with the right numerical methods.

Time-Variant vs Time-Invariant Systems

A system whose dynamics depend on time is called a *time-variant system*, while a dynamic system is called *time-invariant* if its evolution does not depend on the time and date when it is happening. As the laws of physics are time-invariant, most technical systems belong to the latter class, but for example the temperature evolution of a house with hot days and cold nights might best be described by a time-variant system model. While the class of time-variant systems trivially comprises all time-invariant systems, it is an important observation that also the other direction holds: each time-variant system can be modelled by a nonlinear time-invariant system if the state space is augmented by an extra state that takes account of the advancement of time, and which we might call the “clock state”.

Linear vs Nonlinear Systems

If the state trajectory of a system depends linearly on the initial value and the control inputs, it is called a *linear system*. If the dependence is affine, one should ideally speak

of an *affine system*, but often the term linear is used here as well. In all other cases, we speak of a *nonlinear system*.

A particularly important class of linear systems are *linear time invariant (LTI)* systems. An LTI system can be completely characterized in at least three equivalent ways: first, by two matrices that are typically called A and B; second, by its *step response function*; and third, by its *frequency response function*. A large part of the research in the control community is devoted to the study of LTI systems.

Controlled vs Uncontrolled Dynamic Systems

While we are in this lecture mostly interested in *controlled dynamic systems*, i.e. systems that have a control input that we can choose, it is good to remember that there exist many systems that cannot be influenced at all, but that only evolve according to their intrinsic laws of motion. These *uncontrolled systems* have an empty control set, $\mathbb{U} = \emptyset$. If a dynamic system is both uncontrolled and time-invariant it is also called an *autonomous system*.

Note that an autonomous system with discrete state space that also lives in discrete time is often called an *automaton*.

Within the class of controlled dynamic systems, of special interest are the so called *controllable systems*, which have the desirable property that their state vector x can be steered from any initial state x_{init} to any final state x_{fin} in a finite time with suitably chosen control input trajectories. Many controlled systems of interest are not completely controllable because some parts of their state space cannot be influenced by the control inputs. If these parts are stable, the system is called *stabilizable*.

Stable vs Unstable Dynamic Systems

A dynamic system whose state trajectory remains bounded for bounded initial values and controls is called a *stable system*, and an *unstable system* otherwise. For autonomous systems, *stability* of the system around a fixed point can be defined rigorously: for any arbitrarily small neighborhood $\mathcal{N}$ around the fixed point there exists a region so that all trajectories that start in this region remain in $\mathcal{N}$. *Asymptotic stability* is stronger and additionally requires that all considered trajectories eventually converge to the fixed point. For autonomous LTI systems, stability can be computationally characterized by the eigenvalues of the system matrix.

Deterministic vs Stochastic Systems

If the evolution of a system can be predicted when its initial state and the control inputs are known, it is called a *deterministic system*. When its evolution involves some random behaviour, we call it a *stochastic system*.

The movements of assets on the stockmarket are an example for a stochastic system, whereas the motion of planets in the solar system can usually be assumed to be deterministic. An interesting special case of deterministic systems with continuous state space are *chaotic systems*. These systems are so sensitive to their initial values that even knowing these to arbitrarily high, but finite, precisions does not allow one to

predict the complete future of the system: only the near future can be predicted. The partial differential equations used in weather forecast models have this property, and one well-known chaotic system of ODE, the *Lorenz attractor*, was inspired by these.

Note that also games like chess can be interpreted as dynamic systems. Here the evolution is neither deterministic nor stochastic, but determined by the actions of an adverse player. If we assume that the adversary always chooses the worst possible control action against us, we enter the field of *game theory*, which in continuous state spaces and engineering applications is often denoted by *robust optimal control*.

Open-Loop vs Closed-Loop Controlled Systems

When choosing the inputs of a controlled dynamic system, one first way is decide in advance, before the process starts, which control action we want to apply at which time instant. This is called *open-loop control* in the systems and control community, and has the important property that the control u is a function of time only and does not depend on the current system state.

A second way to choose the controls incorporates our most recent knowledge about the system state which we might observe with the help of measurements. This knowledge allows us to apply feedback to the system by adapting the control action according to the measurements. In the systems and control community, this is called *closed-loop control*, but also the more intuitive term *feedback control* is used. It has the important property that the control action does depend on the current state. The map from the state to the control action is called a *feedback control policy*. In case this policy optimizes our optimization objective, it is called the *optimal feedback control policy*.

Open-loop control can be compared to a cooking instruction that says: cook the potatos for 25 minutes in boiling water. A closed-loop, or feedback control of the same process would for example say: cook the potatos in boiling water until they are so soft that they do not attach anymore to a fork that you push into them. The feedback control approach promises the better result, but requires more work as we have to take the measurements.

This lecture is mainly concerned with numerical methods of how to compute optimal open-loop controls for given objective and constraints. But the last part of the lecture is concerned with a powerful method to approximate the optimal feedback control policy: *model predictive control*, a feedback control technique that is based on the repeated solution of open-loop optimal control problems.

Focus of This Script

In this script we have a strong focus on deterministic systems with continuous state and control spaces. Mostly, we consider discrete time systems, while in a follow up lecture on numerical optimal control we discuss the treatment of continuous time systems in much more detail.

The main reason for our focus on continuous state and control spaces is that the resulting optimal control problems can efficiently be treated by derivative-based optimization methods. They are thus tremendously easier to solve than most other classes, both in terms of the solvable system sizes and of computational speed. Also, these

continuous optimal control problems comprise the important class of convex optimal control problems, which 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.

1.2 Continuous Time Systems

Most systems of interest in science and engineering are described in form of differential equations which live in continuous time. On the other hand, all numerical simulation methods have to discretize the time interval of interest in some form or the other and thus effectively generate discrete time systems. We will thus only briefly sketch some relevant properties of continuous time systems in this section, and sketch how they can be transformed into discrete time systems. Throughout the lecture, we will mainly be concerned with discrete time systems, while we occasionally come back to the continuous time case.

Ordinary Differential Equations

A controlled dynamic system in continuous time can in the simplest case be described by an ordinary differential equation (ODE) on a time interval $[t_{\text{init}}, t_{\text{fin}}]$ by

$$\dot{x}(t) = f(x(t), u(t), t), \quad t \in [t_{\text{init}}, t_{\text{fin}}] \tag{1.1}$$

where $t \in \mathbb{R}$ is the time, $u(t) \in \mathbb{R}^{n_u}$ are the controls, and $x(t) \in \mathbb{R}^{n_x}$ is the state. The function f is a map from states, controls, and time to the rate of change of the state, i.e. $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times [t_{\text{init}}, t_{\text{fin}}] \to \mathbb{R}^{n_x}$. Due to the explicit time dependence of the function f, this is a time-variant system.

We are first interested in the question if this differential equation has a solution if the initial value $x(t_{\text{init}})$ is fixed and also the controls $u(t)$ are fixed for all $t \in [t_{\text{init}}, t_{\text{fin}}]$. In this context, the dependence of f on the fixed controls $u(t)$ is equivalent to a a further time-dependence of f, and we can redefine the ODE as $\dot{x} = \tilde{f}(x, t)$ with $\tilde{f}(x, t) := f(x, u(t), t)$. Thus, let us first leave away the dependence of f on the controls, and just regard the time-dependent uncontrolled ODE:

$$\dot{x}(t) = f(x(t), t), \quad t \in [t_{\text{init}}, t_{\text{fin}}]. \tag{1.2}$$

Initial Value Problems

An initial value problem (IVP) is given by (1.2) and the initial value constraint $x(t_{\text{init}}) = x_{\text{init}}$ with some fixed parameter x_{init}. Existence of a solution to an IVP is guaranteed under continuity of f with respect to to x and t according to a theorem from 1886 that is due to Giuseppe Peano. But existence alone is of limited interest as the solutions might be non-unique.

Example 1 (Non-Unique ODE Solution) *The scalar ODE with $f(x) = \sqrt{|x(t)|}$ can stay for an undetermined duration in the point $x = 0$ before leaving it at an arbitrary*

time t_0. It then follows a trajectory $x(t) = (t - t_0)^2/4$ that can be easily shown to satisfy the ODE (1.2)*. We note that the ODE function f is continuous, and thus existence of the solution is guaranteed mathematically. However, at the origin, the derivative of f approaches infinity. It turns out that this is the reason which causes the non-uniqueness of the solution.*

As we are only interested in systems with well-defined and deterministic solutions, we would like to formulate only ODE with unique solutions. Here helps the following theorem by Charles Émile Picard (1890) and Ernst Leonard Lindelöf (1894).

Theorem 1 (Existence and Uniqueness of IVP) *Regard the initial value problem* (1.2) *with $x(t_{\text{init}}) = x_{\text{init}}$, and assume that $f : \mathbb{R}^{n_x} \times [t_{\text{init}}, t_{\text{fin}}] \to \mathbb{R}^{n_x}$ is continuous with respect to x and t. Furthermore, assume that f is Lipschitz continuous with respect to x, i.e., that there exists a constant L such that for all $x, y \in \mathbb{R}^{n_x}$ and all $t \in [t_{\text{init}}, t_{\text{fin}}]$*

$$\|f(x,t) - f(y,t)\| \leq L\|x - y\|. \tag{1.3}$$

Then there exists a unique solution $x : [t_{\text{init}}, t_{\text{fin}}] \to \mathbb{R}^{n_x}$ of the IVP.

Lipschitz continuity of f with respect to x is not easy to check. It is much easier to verify if a function is differentiable. It is therefore a helpful fact that every function f that is differentiable with respect to x is also locally Lipschitz continuous, and one can prove the following corollary to the Theorem of Picard-Lindelöf.

Corollary 1 (Local Existence and Uniqueness) *Regard the same initial value problem as in Theorem 1, but instead of global Lipschitz continuity, assume that f is continuously differentiable with respect to x for all $t \in [t_{\text{init}}, t_{\text{fin}}]$. Then there exists a possibly shortened, but non-empty interval $[t_{\text{init}}, t'_{\text{fin}}]$ with $t'_{\text{fin}} \in (t_{\text{init}}, t_{\text{fin}}]$ on which the IVP has a unique solution.*

Note that for nonlinear continuous time systems – in contrast to discrete time systems – it is very easily possible to obtain an "explosion", i.e., a solution that tends to infinity for finite times, even with innocently looking and smooth functions f.

Example 2 (Explosion of an ODE) *Regard the scalar example $f(x) = x^2$ with $t_{\text{init}} = 0$ and $x_{\text{init}} = 1$, and let us regard the interval $[t_{\text{init}}, t_{\text{fin}}]$ with $t_{\text{fin}} = 10$. The IVP has the explicit solution $x(t) = 1/(1-t)$, which is only defined on the half open interval $[0, 1)$, because it tends to infinity for $t \to 1$. Thus, we need to choose some $t'_{\text{fin}} < 1$ in order to have a unique and finite solution to the IVP on the shortened interval $[t_{\text{init}}, t'_{\text{fin}}]$. The existence of this local solution is guaranteed by the above corollary. Note that the explosion in finite time is due to the fact that the function f is not globally Lipschitz continuous, so Theorem 1 is not applicable.*

Discontinuities with Respect to Time

It is important to note that the above theorem and corollary can be extended to the case that there are finitely many discontinuities of f with respect to t. In this case the ODE solution can only be defined on each of the continuous time intervals separately, while the derivative of x is not defined at the time points at which the discontinuities of f

occur, at least not in the strong sense. But the transition from one interval to the next can be determined by continuity of the state trajectory, i.e. we require that the end state of one continuous initial value problem is the starting value of the next one.

The fact that unique solutions still exist in the case of discontinuities is important because, first, many optimal control problems have discontinuous control trajectories $u(t)$ in their solution, and, second, many algorithms discretize the controls as piecewise constant functions which have jumps at the interval boundaries. Fortunately, this does not cause difficulties for existence and uniqueness of the IVPs.

Linear Time Invariant (LTI) Systems

A special class of tremendous importance are the linear time invariant (LTI) systems. These are described by an ODE of the form

$$\dot{x} \quad = \quad Ax + Bu \tag{1.4}$$

with fixed 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. Note that the function $f(x, u) = Ax + Bu$ is Lipschitz continuous with respect to x with Lipschitz constant $L = \|A\|$, so that the global solution to any initial value problem with a piecewise continuous control input can be guaranteed.

Many important notions such as *controllability* or *stabilizability*, and computational results such as the *step response* or *frequency response function* can be defined in terms of the matrices A and B alone. Note that in the field of linear system analysis and control, usually also output equations $y = Cx$ are present, where the outputs y may be the only physically relevant quantities. Only the linear operator from u to y - the input-output-behaviour - is of interest, while the state x is just an intermediate quantity. In that context, the states are not even unique, because different state space realizations of the same input-output behavior exist. In this lecture, however, we are not interested in input-outputs-behaviours, but assume that the state is the principal quantity of interest. Output equations are not part of the models in this lecture. If one wants to make the connection to the LTI literature, one might set $C = \mathbb{I}$.

Zero Order Hold and Solution Map

In the age of digital control, the inputs u are often generated by a computer and implemented at the physical system as piecewise constant between two sampling instants. This is called *zero order hold*. The grid size is typically constant, say of fixed length $\Delta t > 0$, so that the sampling instants are given by $t_k = k \cdot \Delta t$. If our original model is a differentiable ODE model, but we have piecewise constant control inputs with fixed values $u(t) = u_k$ wtih $u_k \in \mathbb{R}^{n_u}$ on each interval $t \in [t_k, t_{k+1}]$, we might want to regard the transition from the state $x(t_k)$ to the state $x(t_{k+1})$ as a discrete time system. This is indeed possible, as the ODE solution exists and is unique on the interval $[t_k, t_{k+1}]$ for each initial value $x(t_k) = x_{\text{init}}$.

If the original ODE system is time-invariant, it is enough to regard one initial value problem with constant control $u(t) = u_{\text{const}}$

$$\dot{x}(t) = f(x(t), u_{\text{const}}), \quad t \in [0, \Delta t], \quad \text{with} \quad x(0) = x_{\text{init}}. \tag{1.5}$$

The unique solution $x : [0, \Delta t] \to \mathbb{R}^{n_x}$ to this problem is a function of both, the initial value x_{init} and the control u_{const}, so we might denote the solution by

$$x(t; x_{\text{init}}, u_{\text{const}}), \quad \text{for} \quad t \in [0, \Delta t]. \tag{1.6}$$

This map from $(x_{\text{init}}, u_{\text{const}})$ to the state trajectory is called the *solution map*. The final value of this short trajectory piece, $x(\Delta t; x_{\text{init}}, u_{\text{const}})$, is of major interest, as it is the point where the next sampling interval starts. We might define the transition function $f_{\text{dis}} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$ by $f_{\text{dis}}(x_{\text{init}}, u_{\text{const}}) = x(\Delta t; x_{\text{init}}, u_{\text{const}})$. This function allows us to define a discrete time system that uniquely describes the evolution of the system state at the sampling instants t_k:

$$x(t_{k+1}) = f_{\text{dis}}(x(t_k), u_k). \tag{1.7}$$

Solution Map of Linear Time Invariant Systems

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

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

with initial value x_{init} at $t_{\text{init}} = 0$, and constant control input u_{const}, the solution map $x(t; x_{\text{init}}, u_{\text{const}})$ is explicitly given as

$$x(t; x_{\text{init}}, u_{\text{const}}) = \exp(At)x_{\text{init}} + \int_0^t \exp(A(t-\tau))Bu_{\text{const}}\mathrm{d}\tau,$$

where $\exp(A)$ is the matrix exponential. It is interesting to note that this map is well defined for all times $t \in \mathbb{R}$, as linear systems cannot explode. The corresponding discrete time system with sampling time Δt is again a linear time invariant system, and is given by

$$f_{\text{dis}}(x_k, u_k) = A_{\text{dis}}x_k + B_{\text{dis}}u_k \tag{1.8}$$

with

$$A_{\text{dis}} = \exp(A\Delta t) \quad \text{and} \quad B_{\text{dis}} = \int_0^{\Delta t} \exp(A(\Delta t - \tau))B\mathrm{d}\tau. \tag{1.9}$$

One interesting observation is that the discrete time system matrix A_{dis} resulting from the solution of an LTI system in continuous time is by construction an invertible matrix, with inverse $A_{\text{dis}}^{-1} = \exp(-A\Delta t)$. For systems with strongly decaying dynamics, however, the matrix A_{dis} might have some very small eigenvalues and will thus be nearly singular.

Sensitivities

In the context of optimal control, derivatives of the dynamic system simulation are needed for nearly all numerical algorithms. Following Theorem 1 and Corollary 1 we know that the solution map to the IVP (1.5) exists on an interval $[0, \Delta t]$ and is unique under mild conditions even for general nonlinear systems. But is it also differentiable with respect to the initial value and control input?

In order to discuss the issue of derivatives, which in the dynamic system context are often called *sensitivities*, let us first ask what happens if we call the solution map with different inputs. For small perturbations of the values $(x_{\text{init}}, u_{\text{const}})$, we still have a unique solution $x(t; x_{\text{init}}, u_{\text{const}})$ on the whole interval $t \in [0, \Delta t]$. Let us restrict ourselves to a neighborhood $\mathcal{N}$ of fixed values $(x_{\text{init}}, u_{\text{const}})$. For each fixed $t \in [0, \Delta t]$, we can now regard the well defined and unique solution map $x(t; \cdot) : \mathcal{N} \to \mathbb{R}^{n_x},\ (x_{\text{init}}, u_{\text{const}}) \mapsto x(t; x_{\text{init}}, u_{\text{const}})$. 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 u, then the solution map $x(t; \cdot)$, for each $t \in [0, \Delta t]$, is also m-times continuously differentiable with respect to $(x_{\text{init}}, u_{\text{const}})$.

In the general nonlinear case, the solution map $x(t; x_{\text{init}}, u_{\text{const}})$ 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 a follow up course on numerical optimal control. To mention already the main difficulty, note that most 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 $x(t; x_{\text{init}}, u_{\text{const}})$ typically non-differentiable in the inputs $x_{\text{init}}, u_{\text{const}}$. Thus, multiple calls of a black-box integrator and application of finite differences might result in very wrong derivative approximations.

Numerical Integration Methods

A numerical simulation routine that approximates the solution map is often called an *integrator*. A simple but very crude way to generate an approximation for $x(t; x_{\text{init}}, u_{\text{const}})$ for $t \in [0, \Delta t]$ is to perform a linear extrapolation based on the time derivative $\dot{x} = f(x, u)$ at the initial time point:

$$\tilde{x}(t; x_{\text{init}}, u_{\text{const}}) = x_{\text{init}} + t f(x_{\text{init}}, u_{\text{const}}), \quad t \in [0, \Delta t]. \tag{1.10}$$

This is called one *Euler integration step*. For very small Δt, this approximation becomes very good. In fact, the error $\tilde{x}(\Delta t; x_{\text{init}}, u_{\text{const}}) - x(\Delta t; x_{\text{init}}, u_{\text{const}})$ is of second order in Δt. This motivated Leonhard Euler to perform several steps of smaller size, and propose what is now called the *Euler integration method*. We subdivide the interval $[0, \Delta t]$ into M subintervals each of length $h = \Delta t / M$, and perform M such linear extrapolation steps consecutively, starting at $\tilde{x}_0 = x_{\text{init}}$:

$$\tilde{x}_{j+1} = \tilde{x}_j + h f(\tilde{x}_j, u_{\text{const}}), \quad j = 0, \ldots, M-1. \tag{1.11}$$

It can be proven that the Euler integration method is *stable*, i.e. that the propagation of local errors is bounded with a constant that is independent of the step size h. Therefore,

the approximation becomes better and better when we decrease the step size h: since the *consistency* error in each step is of order h^2, and the total number of steps is of order $\Delta t/h$, the accumulated error in the final step is of order $h\Delta t$. As this is linear in the step size h, we say that the Euler method has the *order one*. Taking more steps is more accurate, but also needs more computation time. One measure for the computational effort of an integration method is the number of evaluations of f, which for the Euler method grows linearly with the desired accuracy.

In practice, the Euler integrator is rarely competitive, because other methods exist that deliver the desired accuracy levels at much lower computational cost. We discuss several numerical simulation methods later, but present here already one of the most widespread integrators, the *Runge-Kutta Method of Order Four*, which we will often abbreviate as *RK4*. One step of the RK4 method needs four evaluations of f and stores the results in four intermediate quantities $k_i \in \mathbb{R}^{n_x}$, $i = 1, \ldots, 4$. Like the Euler integration method, the RK4 also generates a sequence of values $\tilde{x}_j$, $j = 0, \ldots, M$, with $\tilde{x}_0 = x_{\text{init}}$. At $\tilde{x}_j$, and using the constant control input u_{const}, one step of the RK4 method proceeds as follows:

$$
\begin{aligned}
k_1 &= f(\tilde{x}_j, u_{\text{const}}) && \text{(1.12a)}\\
k_2 &= f(\tilde{x}_j + \frac{h}{2} k_1, u_{\text{const}}) && \text{(1.12b)}\\
k_3 &= f(\tilde{x}_j + \frac{h}{2} k_2, u_{\text{const}}) && \text{(1.12c)}\\
k_4 &= f(\tilde{x}_j + h\, k_3, u_{\text{const}}) && \text{(1.12d)}\\
\tilde{x}_{j+1} &= \tilde{x}_j + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) && \text{(1.12e)}
\end{aligned}
$$

One step of RK4 is thus as expensive as four steps of the Euler method. But it can be shown that the accuracy of the final approximation $\tilde{x}_M$ is of order $h^4\Delta t$. In practice, this means that the RK4 method usually needs tremendously fewer function evaluations than the Euler method to obtain the same accuracy level.

From here on, and throughout the major part of the lecture, we will leave the field of continuous time systems, and directly assume that we control a discrete time system $x_{k+1} = f_{\text{dis}}(x_k, u_k)$. Let us keep in mind, however, that the transition map $f_{\text{dis}}(x_k, u_k)$ is usually not given as an explicit expression but can instead be a relatively involved computer code with several intermediate quantities. In the exercises of this lecture, we will usually discretize the occuring ODE systems by using only one Euler or RK4 step per control interval, i.e. use $M = 1$ and $h = \Delta t$. The RK4 step often gives already a sufficient approximation at relatively low cost.

1.3 Discrete Time Systems

Let us now discuss in more detail the discrete time systems that are at the basis of the control problems in the first part of this lecture. In the general time-variant case, these systems are characterized by the dynamics

$$
x_{k+1} = f_k(x_k, u_k), \quad k = 0, 1, \ldots, N-1 \qquad \text{(1.13)}
$$

on a time horizon of length N, with N control input vectors $u_0, \ldots, u_{N-1} \in \mathbb{R}^{n_u}$ and $(N+1)$ state vectors $x_0, \ldots, x_N \in \mathbb{R}^{n_x}$.

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

Definition 1 (Forward simulation) *The* forward simulation *is the map*

$$\begin{array}{rccc} f_{\text{sim}}: & \mathbb{R}^{n_x+Nn_u} & \to & \mathbb{R}^{(N+1)n_x} \\ & (x_0; u_0, u_1, \ldots, u_{N-1}) & \mapsto & (x_0, x_1, x_2, \ldots, x_N) \end{array} \tag{1.14}$$

that is defined by solving Equation (1.13) *recursively for all* $k = 0, 1, \ldots, N-1$.

The inputs of the forward simulation routine are the initial value x_0 and the controls u_k for $k = 0, \ldots, N-1$. 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 manuscript it is important to note that the fundamental equation that is characterizing a dynamic optimization problem are the system dynamics stated in Equation (1.13), but no initial value constraint, which is optional.

Linear Time Invariant (LTI) Systems

As discussed already for the continuous time case, linear time invariant (LTI) systems are not only one of the simplest possible dynamic system classes, but also have a rich and beautiful history. In the discrete time case, they are determined by the system equation

$$x_{k+1} \quad = \quad Ax_k + Bu_k, \quad k = 0, 1, \ldots, N-1. \tag{1.15}$$

with fixed matrices $A \in \mathbb{R}^{n_x \times n_x}$ and $B \in \mathbb{R}^{n_x \times n_u}$. An LTI system is asymptotically stable if all eigenvalues of the matrix A are strictly inside the unit disc of the complex plane, i.e. have a modulus smaller than one. It is easy to show that the forward simulation map for an LTI system on a horizon with length N is given by

$$f_{\text{sim}}(x_0; u_0, \ldots, u_{N-1}) = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} x_0 \\ Ax_0 + Bu_0 \\ A^2x_0 + ABu_0 + Bu_1 \\ \vdots \\ A^Nx_0 + \sum_{k=0}^{N-1} A^{N-1-k}Bu_k \end{bmatrix}$$

In order to check controllability, due to linearity, one might ask the question if after N steps any terminal state x_N can be reached from $x_0 = 0$ by a suitable choice of control

inputs. Because of

$$x_N = \underbrace{\begin{bmatrix} A^{N-1}B & A^{N-2}B & \cdots & B \end{bmatrix}}_{=\mathcal{C}_N} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{bmatrix}$$

this is possible if and only if the matrix $\mathcal{C}_N \in \mathbb{R}^{n_x \times N n_u}$ has the rank n_x. Increasing N can only increase the rank, but one can show that the maximum possible rank is already reached for $N = n_x$, so it is enough to check if the so called *controllability matrix* $\mathcal{C}_{n_x}$ has the rank n_x.

Eigenvalues and Eigenvectors of LTI Systems

Every square matrix $A \in \mathbb{R}^{n_x \times n_x}$ can be brought into the Jordan canonical form $A = QJQ^{-1}$ with non-singular $Q \in \mathbb{C}^{n_x \times n_x}$ and J block diagonal, consisting of m-Jordan blocks J_i. Thus, it holds that

$$J = \begin{bmatrix} J_1 & & & \\ & J_2 & & \\ & & \ddots & \\ & & & J_m \end{bmatrix} \quad \text{with} \quad J_i = \begin{bmatrix} \lambda_i & 1 & & \\ & \lambda_i & 1 & \\ & & \ddots & \ddots \\ & & & \lambda_i \end{bmatrix}.$$

Many of the Jordan blocks might just have size one, i.e. $J_i = [\lambda_i]$. To better understand the uncontrolled system evolution with dynamics $x_{k+1} = Ax_k$ and initial condition $x_0 = x_{\text{init}}$, one can regard the solution map $x_N = A^N x_0$ in the eigenbasis, which yields the expression

$$x_N = Q\ J^N\ (Q^{-1}x_0)$$

First, it is seen that all Jordan blocks evolve independently, after the initial condition is represented in the eigenbasis. Second, a simple Jordan block J_i will just result in the corresponding component being multiplied by a factor λ_i^N. Third, for nontrivial Jordan blocks, one obtains more complex expressions with N upper diagonals of the form

$$J_i^N = \begin{bmatrix} \lambda_i^N & N\lambda_i^{N-1} & \cdots & 1 & \\ & \lambda_i^N & N\lambda_i^{N-1} & & \ddots \\ & & & \ddots & \\ & & & & \lambda_i^N \end{bmatrix}.$$

If one eigenvalue has a larger modulus $|\lambda_i|$ than all others, the Jordan block J_i^N will grow faster (or shrink slower) than the others for increasing N. The result is that the corresponding eigenvector(s) will dominate the final state x_N for large N, while all others "die out". Here, the second largest eigenvalues will result in the most slowly

decaying components, and their corresponding eigenvectors will keep a visible contribution in x_N the longest.

Interestingly, complex eigenvalues as well as eigenvectors appear in complex conjugate pairs. If an eigenvalue λ_i is complex, the (real part of) the corresponding eigenvector will perform oscillatory motion. To understand the behaviour of complex eigenvectors, let us regard a complex conjugate pair of simple eigenvalues λ_i and $\lambda_j = \bar{\lambda}_i$, and their eigenvectors $v_i, v_j \in \mathbb{C}^{n_x}$, i.e. $Av_i = \lambda_i v_i$ and $Av_j = \bar{\lambda}_i v_j$. It is easy to see that, because A is real, $v_j = \bar{v}_i$ is a possible choice for the eigenvector corresponding to $\bar{\lambda}_i$. Then holds that $\mathrm{Re}\{v_i\} = \frac{1}{2}(v_i + v_j)$. Therefore,

$$A^N \,\mathrm{Re}\{v_i\} = \frac{1}{2}(\lambda_i^N v_i + \lambda_j^N v_j) = \frac{1}{2}(\lambda_i^N v_i + \bar{\lambda}_i^N \bar{v}_i) = \mathrm{Re}\{\lambda_i^N v_i\}.$$

If we represent λ_i as $\lambda_i = re^{\phi i}$ (where the i in the exponent is the imaginary unit while the other i remains just an integer index), then $\lambda_i^N = r^N e^{N\phi i}$. If ϕ is a fraction of 2π, there is an N such that $N\phi = 2\pi$, and after N iterations we will obtain the same real part as in the original eigenvector, but multiplied with r^N. We can conclude that the real part of the eigenvector to a complex eigenvalue $re^{\phi i}$ performs a form of damped or growing oscillatory motion with period duration $N = 2\pi/\phi$ and growth constant r.

Affine Systems and Linearizations along Trajectories

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

$$x_{k+1} \quad = \quad A_k x_k + B_k u_k + c_k, \quad k = 0, 1, \ldots, N-1. \tag{1.16}$$

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

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

thus resulting in affine time-varying dynamics of the form (1.16). 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.

It is an important fact that the forward simulation map of an affine system (1.16) is again an affine function of the initial value and the controls. More specifically, this affine map is for any $N \in \mathbb{N}$ given by:

$$x_N = (A_{N-1} \cdots A_0)\, x_0 + \sum_{k=0}^{N-1} \left(\Pi_{j=k+1}^{N-1} A_j\right)(B_k u_k + c_k)\,.$$

1.4 Optimization Problem Classes

Mathematical optimization refers to finding the best, or *optimal* solution among a set of possible decisions, where optimality is defined with the help of an *objective function*.

Some solution candidates are *feasible*, others not, and it is assumed that *feasibility* of a solution candidate can be checked by evaluation of some *constraint functions* that need for example be equal to zero. Like the field of dynamic systems, the field of mathematical optimization comprises many different problem classes, which we will briefly try to classify in this section.

Historically, optimization has been identified with programming, where a program was understood as a deterministic plan, e.g., in logistics. For this reason, many of the optimization problem classes have been given names that contain the words *program* or *programming*. In this script we will often use these names and their abbreviations, because they are still widely used. Thus, we use e.g. the term *linear program (LP)* as a synonym for a *linear optimization problem*. It is interesting to note that the major society for mathematical optimization, which had for decades the name *Mathematical Programming Society (MPS)*, changed its name in 2011 to *Mathematical Optimization Society (MOS)*, while it decided not to change the name of its major journal, that still is called *Mathematical Programming*. In this script we chose a similarly pragmatic approach to the naming conventions.

Finite vs Infinite Dimensional Optimization

An important divididing line in the field of optimization regards the dimension of the space in which the decision variable, say x, is chosen. If x can be represented by finitely many numbers, e.g. $x \in \mathbb{R}^n$ with some $n \in \mathbb{N}$, we speak of a *finite dimensional optimization problem*, otherwise, of an *infinite dimensional optimization problem*. The second might also be referred to as *optimization in function spaces*. Discrete time optimal control problems fall into the first, continuous time optimal control problems into the second class.

Besides the dimension of the decision variable, also the dimension of the constraint functions can be finite or infinite. If an infinite number of inequality constraints is present while the decision variable is finite dimensional, one speaks of a *semi-infinite optimization problem*. This class naturally arises in the context of *robust optimization*, where one wants to find the best choice of the decision variable that satisfies the constraints for all possible values of an unknown but bounded disturbance.

Continuous vs Integer Optimization

A second dividing line concerns the type of decision variables. These can be either *continuous*, like for example real valued vectors $x \in \mathbb{R}^n$, or any other elements of a smooth manifold. On the other hand, the decision variable can be *discrete*, or *integer valued*, i.e. we have $z \in \mathbb{Z}^n$, or, when a set of binary choices has to be made, $z \in \{0, 1\}^n$. In this case one often also speaks of *combinatorial optimization*. If an optimization problem has both, continuous and integer variables, it is called a *mixed-integer optimization problem*.

An important class of continuous optimization problems are the so called *nonlinear*

programs (NLP). They can be stated in the form

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) && && \text{(1.17a)} \\ \text{subject to} \quad & g(x) &=& \ 0, && \text{(1.17b)} \\ & h(x) &\leq& \ 0, && \text{(1.17c)} \end{aligned}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^{n_g}$, and $h : \mathbb{R}^n \to \mathbb{R}^{n_h}$ are assumed to be at least once continuously differentiable. Note that we use function and variable names such as f and x with a very different meaning than before in the context of dynamic systems. In the first part of the lecture we discuss algorithms to solve this kind of optimization problems, and the discrete time optimal control problems treated in this lecture can also be regarded as a specially structured form of NLPs. Two important subclasses of NLPs are the *linear programs (LP)*, which have affine problem functions f, g, h, and the *quadratic programs (QP)*, which have affine constraint functions g, h and a more general linear quadratic objective $f(x) = c^T x + \frac{1}{2} x^T H x$ with a symmetric matrix $H \in \mathbb{R}^{n \times n}$.

A large class of mixed-integer optimization problems are the so called *mixed integer nonlinear programs (MINLP)*, which can be stated as

$$\begin{aligned} \underset{\substack{x \in \mathbb{R}^n \\ z \in \mathbb{Z}^m}}{\text{minimize}} \quad & f(x, z) && && \text{(1.18a)} \\ \text{subject to} \quad & g(x, z) &=& \ 0, && \text{(1.18b)} \\ & h(x, z) &\leq& \ 0. && \text{(1.18c)} \end{aligned}$$

Among the MINLPs, an important special case arises if the problem functions f, g, h are affine in both variables, x and z, which is called a *mixed integer linear program (MILP)*. If the objective is allowed to be linear quadratic, one speaks of a *mixed integer quadratic program (MIQP)*. If in an MILP only integer variables are present, one usually just calls it an *integer program (IP)*. The field of (linear) integer programming is huge and has powerful algorithms available. Most problems in logistics fall into this class, a famous example being the *travelling salesman problem*, which concerns the shortest closed path that one can travel through a given number of towns, visiting each town exactly once.

An interesting class of mixed-integer optimization problems arises in the context of optimal control of hybrid dynamic systems, which in the discrete time case can be regarded a special case of MINLP. In continuous time, we enter the field of infinite dimensional mixed-integer optimization, often also called *Mixed-integer optimal control problems (MIOCP)*.

Convex vs Nonconvex Optimization

Arguably the most important dividing line in the world of optimization is between convex and nonconvex optimization problems. Convex optimization problems are a

subclass of the continuous optimization problems and arise if the objective function is a convex function and the set of feasible points a convex set. In this case one can show that any *local solution*, i.e. values for the decision variables that lead to the best possible objective value in a neighborhood, is also a *global solution*, i.e. has the best possible objective value among all feasible points. Practically very important is the fact that convexity of a function or a set can be checked just by checking convexity of its building blocks and if they are constructed in a way that preserves convexity.

Several important subclasses of NLPs are convex, such as LPs. Also QPs are convex if they have a convex objective f. Another example are *Quadratically Constrained Quadratic Programs (QCQP)* which have quadratic inequalities and whose feasible set is the intersection of ellipsoids. Some other optimization problems are convex but do not form part of the NLP family. Two widely used classes are *second-order cone programs (SOCP)* and *semi-definite programs (SDP)* which have linear objective functions but more involved convex feasible sets: for SOCP, it is the set of vectors which have one component that is larger than the Euclidean norm of all the other components and which it is called the *second order cone*, and for SDP it is the set of symmetric matrices that are positive semi-definite, i.e. have all eigenvalues larger than zero. SDPs are often used when designing linear feedback control laws. Also infinite dimensional optimization problems such as optimal control problems in continuous time can be convex under fortunate circumstances.

In this context, it is interesting to note that a sufficient condition for convexity of an optimal control problem is that the underlying dynamic system is linear and that the objective and constraints are convex in controls and states. On the other hand, optimal control problems with underlying nonlinear dynamic systems, which are the focus of this lecture, are usually nonconvex.

Optimization problems with integer variables can never be convex due to the nonconvexity of the set of integers. However, it is of great algorithmic advantage if mixed-integer problems have a convex substructure in the sense that convex problems arise when the integer variables are allowed to also take real values. These so called *convex relaxations* are at the basis of nearly all competitive algorithms for mixed-integer optimization. For example, linear integer programs can be solved very efficiently because their convex relaxations are just linear programs, which are convex and can be solved very efficiently.

1.5 Overview and Notation

The chapters of these lecture notes can roughly be divided into six major areas.

- Introduction
- Optimization Background
- Discrete Time Optimal Control
- Continuous Time Optimal Control
- Model Predictive Control and Moving Horizon Estimation

Notation

Within this lecture we use $\mathbb{R}$ for the set of real numbers, $\mathbb{R}_+$ for the non-negative ones and $\mathbb{R}_{++}$ for the positive ones, $\mathbb{Z}$ for the set of integers, and $\mathbb{N}$ for the set of natural numbers including zero, i.e. we identify $\mathbb{N} = \mathbb{Z}_+$. The set of real-valued vectors of dimension n is denoted by $\mathbb{R}^n$, and $\mathbb{R}^{n\times m}$ denotes the set of matrices with n rows and m columns. By default, all vectors are assumed to be column vectors, i.e. we identify $\mathbb{R}^n = \mathbb{R}^{n\times 1}$. We usually use square brackets when presenting vectors and matrices elementwise. Because will often deal with concatenations of several vectors, say $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, yielding a vector in $\mathbb{R}^{n+m}$, we abbreviate this concatenation sometimes as (x, y) in the text, instead of the correct but more clumsy equivalent notations $[x^\top, y^\top]^\top$ or

$$\begin{bmatrix} x \\ y \end{bmatrix}.$$

Square and round brackets are also used in a very different context, namely for intervals in $\mathbb{R}$, where for two real numbers $a < b$ the expression $[a, b] \subset \mathbb{R}$ denotes the closed interval containing both boundaries a and b, while an open boundary is denoted by a round bracket, e.g. (a, b) denotes the open interval and $[a, b)$ the half open interval containing a but not b.

When dealing with norms of vectors $x \in \mathbb{R}^n$, we denote by $\|x\|$ an arbitrary norm, and by $\|x\|_2$ the Euclidean norm, i.e. we have $\|x\|_2^2 = x^\top x$. We denote a weighted Euclidean norm with a positive definite weighting matrix $Q \in \mathbb{R}^{n\times n}$ by $\|x\|_Q$, i.e. we have $\|x\|_Q^2 = x^\top Q x$. The L_1 and L_∞ norms are defined by $\|x\|_1 = \sum_{i=1}^n |x_i|$ and $\|x\|_\infty = \max\{|x_1|, \ldots, |x_n|\}$. Matrix norms are the induced operator norms, if not stated otherwise, and the Frobenius norm $\|A\|_{\mathrm{F}}$ of a matrix $A \in \mathbb{R}^{n\times m}$ is defined by $\|A\|_{\mathrm{F}}^2 = \mathrm{trace}(AA^\top) = \sum_{i=1}^n \sum_{j=1}^m A_{ij} A_{ij}$.

When we deal with derivatives of functions f with several real inputs and several real outputs, i.e. functions $f : \mathbb{R}^n \to \mathbb{R}^m, x \mapsto f(x)$, we define the Jacobian matrix $\frac{\partial f}{\partial x}(x)$ as a matrix in $\mathbb{R}^{m\times n}$, following standard conventions. For scalar functions with $m = 1$, we denote the gradient vector as $\nabla f(x) \in \mathbb{R}^n$, a column vector, also following standard conventions. Slightly less standard, we generalize the gradient symbol to all functions $f : \mathbb{R}^n \to \mathbb{R}^m$ even with $m > 1$, i.e. we generally define in this lecture

$$\nabla f(x) = \frac{\partial f}{\partial x}(x)^\top \in \mathbb{R}^{n\times m}.$$

Using this notation, the first order Taylor series is e.g. written as

$$f(x) = f(\bar{x}) + \nabla f(\bar{x})^\top (x - \bar{x})) + o(\|x - \bar{x}\|)$$

The second derivative, or Hessian matrix will only be defined for scalar functions $f : \mathbb{R}^n \to \mathbb{R}$ and be denoted by $\nabla^2 f(x)$.

For square symmetric matrices of dimension n we sometimes use the symbol $\mathbb{S}_n$, i.e. $\mathbb{S}_n = \{A \in \mathbb{R}^{n\times n} | A = A^\top\}$. For any symmetric matrix $A \in \mathbb{S}_n$ we write $A \succcurlyeq 0$ if it is a positive semi-definite matrix, i.e. all its eigenvalues are larger or equal to zero, and $A \succ 0$ if it is positive definite, i.e. all its eigenvalues are positive. This notation is also used for *matrix inequalities* that allow us to compare two symmetric matrices $A, B \in \mathbb{S}_n$, where we define for example $A \succcurlyeq B$ by $A - B \succcurlyeq 0$.

When using logical symbols, $A \Rightarrow B$ is used when a proposition A implies a proposition B. In words the same is expressed by "If A then B". We write $A \Leftrightarrow B$ for "A if and only if B", and we sometimes shorten this to "A iff B", with a double "f", following standard practice.

Chapter 2

Nonlinear Optimization

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

In this first part of the book we discuss several concepts from the field of mathematical optimization that are important for optimal control. Our focus is on quickly arriving at a point where the specific optimization methods for dynamic systems can be treated, while the same material can be found in much greater detail in many excellent textbooks on numerical optimization such as [NW06].

The reason for keeping this part on optimization self-contained and without explicit reference to optimal control is that this allows us to separate between the general concepts of optimization and those specific to optimal control. For this reason, we use in this part the language and notation that is customary in mathematical optimization. The optimization problem with which we are concerned in this part is the standard *Nonlinear Program (NLP)* that was already stated in the introduction:

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

where $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^{n_g}$, and $h : \mathbb{R}^n \to \mathbb{R}^{n_h}$ are assumed to be twice continuously differentiable. Function f is called the *objective function*, function g is the vector of *equality constraints*, and h the vector of *inequality constraints*. We start with some fundamental definitions. First, we collect all points that satisfy the constraints in one set.

Definition 2 (Feasible set) *The* feasible set Ω *is the set*

$$\Omega := \{x \in \mathbb{R}^n \mid g(x) = 0,\ h(x) \leq 0\}\,.$$

The points of interest in optimization are those feasible points that minimize the objective, and they come in two different variants.

Definition 3 (Global minimum) *The point $x^* \in \mathbb{R}^n$ is a* global minimizer *if and only if (iff) $x^* \in \Omega$ and $\forall x \in \Omega: \ f(x) \geq f(x^*)$. The value $f(x^*)$ is the global minimum.*

Unfortunately, the global minimum is usually difficult to find, and most algorithms allow us to only find *local minimizers*, and to verify optimality only locally.

Definition 4 (Local minimum) *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 $\forall x \in \Omega \cap \mathcal{N}: \ f(x) \geq f(x^*)$. The value $f(x^*)$ is a local minimum.*

In order to be able to state the optimality conditions that allow us to check if a candidate point x^* is a local minimizer or not, we need to describe the feasible set in the neighborhood of x^*. It turns out that not all inequality constraints need to be considered locally, but only the *active* ones.

Definition 5 (Active Constraints and Active Set) *An inequality constraint $h_i(x) \leq 0$ is called* active *at $x^* \in \Omega$ iff $h_i(x^*) = 0$ and otherwise* inactive*. The index set $\mathcal{A}(x^*) \subset \{1, \ldots, n_h\}$ of active inequality constraint indices is called the "active set".*

Often, the name *active set* also comprises all equality constraint indices, as equalities could be considered to be always active.

Problem (2.1) is very generic. In Section 2.1 we review some special cases, which still yield large classes of optimization problems. 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 reduce solution times by orders of magnitude. E.g., an important structure is convexity. It allows us to to find global minima by searching for local minima only.

For the general case we review the first and second order conditions of optimality in Sections 2.2 and 2.3, respectively.

2.1 Important Special Classes

Linear Optimization

An obvious special case occurs when the functions f, g, and h in (2.1) are linear, resulting in a linear optimization problem (or Linear Program, LP)

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^\top x && \text{(2.2a)} \\ \text{subject to} \quad & Ax - b = 0, && \text{(2.2b)} \\ & Cx - d \leq 0. && \text{(2.2c)} \end{aligned}$$

Here, the problem data are $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{n_g \times n}$, $b \in \mathbb{R}^{n_g}$, $C \in \mathbb{R}^{n_h \times n}$, and $d \in \mathbb{R}^{n_h}$.

It is easy to show that one optimal solution of any LP – if the LP does have a solution and is not unbounded – has to be a vertex of the polytope of feasible points. Vertices can be represented and calculated by means of basis solution vectors, with a basis of *active inequality constraints*. Thus, there are only finitely many vertices, giving rise to Simplex algorithms that compare all possible solutions in a clever way. However, naturally also the optimality conditions of Section 2.2 are valid and can be used for algorithms, in particular interior point methods.

Quadratic Optimization

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 Optimization Problem or Quadratic Program (QP). A general QP can be formulated as follows.

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^\top x + \frac{1}{2} x^\top B x && \text{(2.3a)} \\ \text{subject to} \quad & Ax - b = 0, && \text{(2.3b)} \\ & Cx - d \leq 0. && \text{(2.3c)} \end{aligned}$$

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 $f(x) = c^\top x + \frac{1}{2} x^\top B x$.

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 property that it always has unique minimizers.

Convex Optimization

Roughly speaking, a set is convex, if all connecting lines lie inside the set:

Definition 6 (Convex Set) *A set $\Omega \subset \mathbb{R}^n$ is convex if*

$$\forall x, y \in \Omega, t \in [0,1] : \; x + t(y - x) \in \Omega. \tag{2.4}$$

A function is convex, if all secants are above the graph:

Definition 7 (Convex Function) *A function $f : \Omega \to \mathbb{R}$ is convex, if Ω is convex and if*

$$\forall x, y \in \Omega, t \in [0,1] : \; f(x + t(y - x)) \leq f(x) + t(f(y) - f(x)). \tag{2.5}$$

Note that 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 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 (2.1) is convex if the function g is affine and the functions h_i are convex, as supported by the following theorem.

Theorem 2 (Convexity of Sublevel Sets) *The sublevel set $\{x \in \Omega \mid h(x) \leq 0\}$ of a convex function $h : \Omega \to \mathbb{R}$ is convex.*

Definition 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 3 (Local Implies Global Optimality for Convex Problems) *For a convex optimization problem, every local minimum is also a global one.*

We leave the proofs of Theorems 2 and 3 as an exercise.

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 [BTN01, BV04]. Here we only mention an important fact that is related to the positive curvature of a function.

Theorem 4 (Convexity for C^2 Functions) *Assume that $f : \Omega \to \mathbb{R}$ is twice continuously differentiable and Ω convex and open. Then f is convex if and only if for all $x \in \Omega$ the Hessian is positive semi-definite, i.e.,*

$$\forall x \in \Omega : \quad \nabla^2 f(x) \succcurlyeq 0. \tag{2.6}$$

Again, we leave the proof as an exercise. As an example, the quadratic objective function $f(x) = c^\top x + \frac{1}{2} x^\top B x$ of (2.3) is convex if and only if $B \succcurlyeq 0$, because $\forall x \in \mathbb{R}^n : \nabla^2 f(x) = B$.

2.2 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 comes in the following form.

Definition 10 (LICQ) *The* linear independence constraint qualification *(LICQ) holds at $x^* \in \Omega$ iff all vectors $\nabla g_i(x^*)$ for $i \in \{1, \ldots, n_g\}$ and $\nabla h_i(x^*)$ 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 $\frac{\partial \tilde{g}}{\partial x}(x^*)$.

The Karush-Kuhn-Tucker Optimality Conditions

This condition allows us to formulate the famous KKT conditions that are due to Karush [Kar39] and Kuhn and Tucker [KT51].

Theorem 5 (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*

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

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., $\nabla g(x) := \frac{\partial g}{\partial x}(x)^\top$.

Definition 11 (KKT Point) *We call a triple (x^*, λ^*, μ^*) a "KKT Point" if it satisfies LICQ and the KKT conditions Eqs.* (2.8a)-(2.8e)*.*

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

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

The Lagrangian Function

Definition 12 (Lagrangian Function) *We define the so called "Lagrangian function" to be*

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

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, which we omit here for brevity.

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 13 *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.

2.3 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 7 (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 $Z^\top \nabla_x^2 \mathcal{L}(x^*, \lambda^*, \mu^*) Z \succcurlyeq 0$.
(Second Order Necessary Condition, short : SONC)

(b) If $Z^\top \nabla_x^2 \mathcal{L}(x^*, \lambda^*, \mu^*) Z \succ 0$, 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, $Z^\top \nabla_x^2 \mathcal{L}(x^*, \lambda^*, \mu^*) Z$, is called the *reduced Hessian*.

Quadratic Problems with Equality Constraints

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

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^\top x + \frac{1}{2} x^\top B x && \text{(2.10a)} \\ \text{subject to} \quad & Ax + b = 0. && \text{(2.10b)} \end{aligned}$$

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

$$\begin{aligned} c \quad + \quad Bx \quad + \quad A^\top \lambda \quad &= \quad 0 && \text{(2.11a)} \\ b \quad + \quad Ax \quad\quad\quad\quad &= \quad 0. && \text{(2.11b)} \end{aligned}$$

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

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

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 $A^\top = QR$ with $Q = (Y|Z)$ square orthonormal and $R = (\bar{R}^\top | 0)^\top$. Then we can check if the reduced Hessian matrix $Z^\top B Z$ 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 $Z^\top B Z \succ 0$ then x^* is a strict local minimizer. Due to convexity this would also be the global solution of the QP.

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 7 (b) hold, then it can be shown that the KKT-matrix

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

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)

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

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 8 (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 7 (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*

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

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

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

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

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.

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.

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

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 [NW06]. 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 [BV04], whose PDF is freely available.

Chapter 3

Newton-Type Optimization Algorithms

"Nature and nature's laws lay hid in night;
God said "Let Newton be" and all was light."
Alexander Pope

3.1 Equality Constrained Optimization

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

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) && \text{(3.1a)} \\ \text{subject to} \quad & g(x) = 0 && \text{(3.1b)} \end{aligned}$$

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

$$\begin{aligned} \nabla_x \mathcal{L}(x, \lambda) &= 0 && \text{(3.2a)} \\ g(x) &= 0 && \text{(3.2b)} \end{aligned}$$

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

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

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*.

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:

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

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

[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.

Algorithm 1 Equality constrained full step Newton-type method

Choose: initial guesses x_0, λ_0, and a tolerance ϵ
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}, λ_{k+1} from (3.10)
  $k = k + 1$
end while

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

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

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.

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

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

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

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 9 (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 2 *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

$$\begin{aligned} \nabla f(x^*) + \nabla g(x^*)\lambda^* + \nabla h(x^*)\mu^* &= 0 && \text{(3.27a)} \\ g(x^*) &= 0 && \text{(3.27b)} \\ \mu_i^* h_i(x^*) + \tau &= 0, \quad i = 1, \ldots, n_h. && \text{(3.27c)} \end{aligned}$$

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 $\nabla_x \left(\log(-h_i(x))\right) = \frac{1}{h_i(x)} \nabla h_i(x)$). Thus, the above smooth form of the KKT conditions is nothing else than the optimality conditions of a *barrier problem*

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) - \tau \sum_{i=1}^{n_h} \log\left(-h_i(x)\right) && \text{(3.28a)} \\ \text{subject to} \quad & g(x) = 0. && \text{(3.28b)} \end{aligned}$$

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 [WB06, WB09]. 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 7 (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 [GMS97]. Many optimal control packages such as MUSCOD-II [LSBS03] or the open-source package ACADO [HFD11] 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 [Dan63] 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) [FBD08].

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 [NW06].

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 functio*n 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 [NW06].

Chapter 4

Calculating Derivatives

"Progress is measured by the degree of differentiation within a society."
Herbert Read

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.

Input: $x_1, \ldots, x_n$
Output: $x_1, \ldots, x_{n+m}$

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

Note: each ϕ_i depends on only one or two out of $\{x_1, \ldots, x_{n+i}\}$.

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, sine-functions, 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 [GW08].

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 $x_1, \ldots, x_n$ are given before, each elementary operation ϕ_i, $i = 0, \ldots, m-1$ 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 4.1 and illustrated in Example 3 below.

Example 3 (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

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

Thus, if the $n = 3$ inputs x_1, x_2, x_3 are given, the $m = 5$ elementary operations $\phi_0, \ldots, \phi_4$ compute the $m = 5$ intermediate quantities, $x_4, \ldots, x_8$, 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.

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

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.

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:

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

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.

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 (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 4.2 is smaller than two times the cost of Algorithm 4.1, 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 4.2 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

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

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

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

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.

Example 5 (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*

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.

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 4.3 is less than 3 times the cost of Algorithm 4.1, 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 4.3 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.

Bibliography

[BTN01] 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.

[BV04] S. Boyd and L. Vandenberghe. *Convex Optimization*. University Press, Cambridge, 2004.

[Dan63] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[FBD08] 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.

[GMS97] 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.

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives*. SIAM, 2 edition, 2008.

[HFD11] B. Houska, H.J. Ferreau, and M. Diehl. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

[Kar39] W. Karush. Minima of Functions of Several Variables with Inequalities as Side Conditions. Master's thesis, Department of Mathematics, University of Chicago, 1939.

[KT51] 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.

[LSBS03] 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.

[NW06] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2 edition, 2006.

[WB06] 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(1):25–57, 2006.

[WB09] A. Wächter and L. Biegler. IPOPT - an Interior Point OPTimizer. https://projects.coin-or.org/Ipopt, 2009.