Numerical Optimal Control
(Draft)

Moritz Diehl and Sébastien Gros

October 11, 2024

Contents

Preface

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

This manuscript is based on lecture notes of courses on optimal control that the authors gave since 2011 at various universities (ETH Zurich, KU Leuven, Trento, Freiburg, Trondheim, Linkoping and Chalmers University of Technology). It profited already from feedback by many students, but is still work in progress and not yet error free. Special thanks go to Sebastian Sager for inspiring discussions on how best to present optimal control, and for suggesting some of the quotes at the start of each chapter. Both authors want to thank Jesus Lago Garcia who helped, during a student job contract, with the Latex editing of text and formulae, who suggested and implemented valuable changes in the organization of the chapters, and who in particular collected and re-edited nearly all of the exercises of this book. MD also wants to thank

Florian Messerer and Armin Nurkanović who helped giving the master course on Numerical Optimal Control at Freiburg University in the past years, and gave valuable feedback. MD also wants to thank James B. Rawlings for valuable advice on textbook writing in general, and for continuing to point out the need for a new textbook on numerical optimal control with focus on direct methods.

The present version of the manuscript is not yet complete, and it is not yet proofread carefully. However, we decided to put the PDF already online so that we can refer to the manuscript in courses we teach and recommend it to interested persons. Feedback is most welcome, in particular at this stage of the writing process.

Santa Barbara and Trondheim — Moritz Diehl and Sébastien Gros
March 2024

Please send feedback and ideas for improvement to
`moritz.diehl@imtek.uni-freiburg.de`
and
`sebastien.gros@ntnu.no`.

1

Introduction: Dynamic Systems and Optimization

Optimal control regards the optimization of dynamic systems. We identify dynamic systems with processes that are evolving in time and that can be characterized by *states* x that allow us to predict the future behavior of the system. Often, the dynamic system can be controlled by a suitable choice of inputs that we denote as *controls* u in this textbook. Typically, these controls shall be chosen optimally in order to optimize some *objective function* and respect some *constraints*. The process of finding the optimal control inputs requires numerical methods, and these methods are the focus of the book.

As an example of an optimal control problem, 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 $[0, T]$, 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.

A typical property of a dynamic system is that knowledge of an *initial state* x_0 and a *control input trajectory* $u(t)$ for all $t \in [0, T]$ allows one to determine the whole *state trajectory* $x(t)$ for $t \in [0, T]$. [1] 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

[1] For ease of notation, and without loss of generality, we use time $t = 0$ as start and $t = T$ as end of most time intervals in this book.

more mathematical detail, before we give an overview of optimization problem classes and finally outline the contents of the book 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 book, 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 Continuous 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 [60].

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 book 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_0 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 book 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 book is concerned with a powerful method to approximate the optimal feedback control policy: *nonlinear model predictive control*, a feedback control technique that is based on the repeated solution of open-loop optimal control problems.

Focus of This Book: Deterministic Systems with Continuous States

In this textbook we have a strong focus on deterministic systems with continuous state and control spaces. In Chapters 7 and we consider discrete time systems, and in Chapters 9 to 14 we discuss continuous time systems.

The main reason for this 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. After this section, and throughout the first two parts of the book,

we will exclusively be concerned with discrete time systems, before we will finally come back to the continuous time case in Chapter 9.

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 $[0, T]$ by

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

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. $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times [0, T] \to \mathbb{R}^{n_x}$. 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 $x(0)$ is fixed and also the controls $u(t)$ are fixed for all $t \in [0, T]$. 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:

$$\dot{x}(t) = f(x(t), t), \quad t \in [0, T]. \tag{1.1}$$

Initial Value Problems

An initial value problem (IVP) is given by (1.1) and the initial value constraint $x(0) = x_0$ with some fixed parameter x_0. Existence of a solution to an IVP is guaranteed under continuity of f 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 1.1 (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 (1.1). 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 1.2 (Existence and Uniqueness of IVP)**.** *Regard the initial value problem* (1.1) *with* $x(0) = x_0$*, and assume that* $f : \mathbb{R}^{n_x} \times [0, T] \to \mathbb{R}^{n_x}$ *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* $t \in [0, T]$

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

Then there exists a unique solution $x : [0, T] \to \mathbb{R}^{n_x}$ *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 1.3 (Local Existence and Uniqueness)**.** *Regard the same initial value problem as in Theorem 1.2, but instead of global Lipschitz continuity, assume that* f *is continuously differentiable with respect to* x *for all* $t \in [0, T]$*. Then there exists a possibly shortened, but non-empty interval* $[0, T']$ *with* $T' \in (0, T]$ *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 even with innocently looking and smooth functions f to obtain an "explosion", i.e., a solution that tends to infinity for finite times.

Example 1.4 (Explosion of an ODE)**.** Regard the scalar example $f(x) = x^2$ with $x_0 = 1$, and let us regard the interval $[0, T]$ with $T = 10$. 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 $T' < 1$ in order to have a unique and finite solution to the IVP on the shortened interval $[0, T']$. 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 1.2 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

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

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 book, 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 book. 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 $x(t_k) = x_0$.

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

$$\dot{x}(t) = f(x(t), u_{\text{const}}), \quad t \in [0, \Delta t], \quad \text{with} \quad x(0) = x_0. \tag{1.2}$$

The unique solution $x : [0, \Delta t] \to \mathbb{R}^{n_x}$ to this problem is a function of both, the initial value x_0 and the control u_{const}, so we might denote the solution by

$$x(t; x_0, u_{\text{const}}), \quad \text{for} \quad t \in [0, \Delta t].$$

This map from (x_0, u_{const}) to the state trajectory is called the *solution map*. The final value of this short trajectory piece, $x(\Delta t; x_0, u_{\text{const}})$, 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 $f_{\text{dis}}(x_0, u_{\text{const}}) = x(\Delta t; x_0, u_{\text{const}})$. 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:

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

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_0 at $t = 0$, and constant control input u_{const}, the solution map $x(t; x_0, u_{\text{const}})$ is explicitly given as

$$x(t; x_0, u_{\text{const}}) = \exp(At)x_0 + \int_0^t \exp(A(t - \tau))Bu_{\text{const}} \mathrm{d}\tau,$$

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

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

with

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

Sensitivities

In the context of optimal control, derivatives of the dynamic system simulation are needed for nearly all numerical algorithms. Following Theorem 1.2 and Corollary 1.3 we know that the solution map to the IVP (1.2) 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_0, u_{const}), we still have a unique solution $x(t; x_0, u_{\text{const}})$ on the whole interval $t \in [0, \Delta t]$. Let us restrict ourselves to a neighborhood $\mathcal{N}$ of fixed values (x_0, u_{const}). For each fixed $t \in [0, \Delta t]$, we can now regard the well defined and unique solution map $x(t; \cdot) : \mathcal{N} \to \mathbb{R}^{n_x},\ (x_0, u_{\text{const}}) \mapsto x(t; x_0, u_{\text{const}})$. 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_0, u_{const}).

In the general nonlinear case, the solution map $x(t; x_0, u_{\text{const}})$ 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 the third part of the book. 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 $x(t; x_0, u_{\text{const}})$ typically non-differentiable in the inputs x_0, u_{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 $x(t; x_0, u_{\text{const}})$ 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:

$$\tilde{x}(t; x_0, u_{\text{const}}) = x_0 + t f(x_0, u_{\text{const}}), \quad t \in [0, \Delta t].$$

This is called one *Euler integration step*. For very small Δt, this approximation becomes very good. In fact, the error $\tilde{x}(\Delta t; x_0, u_{\text{const}}) - x(\Delta t; x_0, u_{\text{const}})$ 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 $\tilde{x}_0 = x_0$:

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

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 $\tilde{x}_0 = x_0$. At $\tilde{x}_j$, and using the constant control input u_{const}, one step of the RK4 method proceeds as follows:

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

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 first part of the book, 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 the first part of this book, 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 Chapters 7 and 8 of this book. In the general time-variant case, these systems are characterized by the dynamics

$$x_{k+1} = f_k(x_k, u_k), \quad k = 0, 1, \ldots, N-1 \tag{1.3}$$

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 1.5 (Forward simulation)**.** The *forward simulation* is the map

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

that is defined by solving Equation (1.3) 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 textbook it is important to note that the fundamental equation that is characterizing a dynamic optimization problem are the system dynamics stated in Equation (1.3), 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

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

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 stable if all eigenvalues of the matrix A are in the unit disc of the complex plane, i.e. have a modulus smaller or equal to one, and *asymptotically stable* if all moduli are strictly 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

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

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.

Affine Systems and Linearizations along Trajectories

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

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

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 (1.4). 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 (1.4) 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 book 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 book 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

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

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 Chapters 2 to **??** we discuss algorithms to solve this kind of optimization problems, and the discrete time optimal control problems treated in Chapters 7 and 8 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 $f(x) = c^\top x + \frac{1}{2} x^\top B x$ with a symmetric matrix $B \in \mathbb{R}^{n \times n}$.

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

$$\begin{aligned} \underset{\substack{x \in \mathbb{R}^n \\ z \in \mathbb{Z}^m}}{\text{minimize}} \quad & f(x, z) \\ \text{subject to} \quad & g(x, z) = 0, \\ & h(x, z) \leq 0. \end{aligned} \tag{1.5}$$

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 book, 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, Exercises and Notation

As said before, the book is divided into four major parts. Below we list the topics which are treated in each part.

- Numerical Optimization: Newton-type optimization methods in many variants.
- Discrete Time Optimal Control: problem formulations, sparsity structure exploitation and dynamic programming.
- Continuous Time Optimal Control: numerical simulation, indirect methods and Hamilton-Jacobi-Bellman equation based approaches, direct collocation, differential-algebraic equations.
- Online Optimal Control: parametric optimization, online quadratic and nonlinear programming, efficient initializations, real-time iterations.

The four parts build on each other, so it is advisable to read and work on them in the order in which they are presented in the book.

Exercises

At the end of each chapter there is a collection of exercises. Some of the exercises are solvable by pen and paper, but many exercises need the use of a computer. In this case, very often we require the use of the following software:

- MATLAB (`www.mathworks.com`) or the open-source alternative OCTAVE (`https://www.gnu.org/software/octave/`).
- The open source packages:
  - CasADi (`https://github.com/casadi/casadi/wiki`).
  - ACADO (`http://acado.github.io/`).
  - qpOASES (`https://projects.coin-or.org/qpOASES`).

Sometimes exercises can only be done with help of data or template files, which can all be downloaded on the webpage that is accompanying this book (`http://www.syscop.de/numericaloptimalcontrol`).

Notation

Within this book 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 $\|A\|_F$ of a matrix $A \in \mathbb{R}^{n\times m}$ is defined by $\|A\|_F^2 = \text{trace}(AA^\top) = \sum_{i=1}^{n}\sum_{j=1}^{m} A_{ij}A_{ij}$.

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 book

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

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.

Exercises

1.1 Consider a linear model of some coutry population with the state vector $x \in \mathbb{R}^{100}$ representing the population of each age group. Let $x_i(k)$ mean the number of people of age i during year k. For instance, $x_6(2014)$ would be the number of people who are 6 years old in year 2014. Each year babies (0-year-olds) are formed depending on a linear birth rate:

$$x_0(k+1) = \sum_{j=0}^{99} \beta_j x_j(k)$$

Each year most of the population ages by one year, except for a fraction who die according to mortality rate μ:

$$x_{i+1}(k+1) = x_i(k) - \mu_i x_i(k) \qquad i = 0, \ldots, 98$$

(a) Download the file `birth_mortality_rates.m` from the book website to obtain the birth rate β and mortality rate μ. Plot them as a function of the population age.

(b) Write the discrete time model in the form of

$$x(k+1) = A\,x(k)$$

(c) Lord of the Flies: Setting an initial population of 100 four-year-olds, and no other people, simulate the system for 150 years. Make a 3-d plot of the population, with axes {year, age, population}.

(d) Eigen decomposition: Plot the eigenvalues of A in the complex plane. Plot the real part of the two eigenvectors of A which have largest eigenvalue magnitude

Is this system stable? What is the significance of these eigenvectors with large eigenvalues?

(e) Run two simulations: in each simulation, use for $x(0)$ the real part of an eigenvector from the previous question. What is the significance of this result?

1.2 Consider a two-dimensional model of an airplane with states $x = [p_x, p_z, v_x, v_z]$ where position $\vec{p} = [p_x, p_z]$ and velocity $\vec{v} = [v_x, v_z]$ are vectors in the $x - z$ directions. We will use the standard aerospace convention that $\hat{x}$ is forward and $\hat{z}$ is DOWN, so altitude is $-p_z$. The system has one control $u = [\alpha]$, where α is the aerodynamic angle of attack in radians. The system dynamics are:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} p_x \\ p_z \\ v_x \\ v_z \end{pmatrix} = \begin{pmatrix} v_x \\ v_z \\ F_x/m \\ F_z/m \end{pmatrix}$$

where $m = 2.0$ is the mass of the airplane. The forces $\vec{F}$ on the airplane are

$$\vec{F} = \vec{F}_{\text{lift}} + \vec{F}_{\text{drag}} + \vec{F}_{\text{gravity}}$$

Lift force $\vec{F}_{\text{lift}}$ is

$$\vec{F}_{\text{lift}} = \frac{1}{2}\rho\|\vec{v}\|^2 C_L(\alpha) S_{\text{ref}}\,\hat{e}_L$$

where lift direction $\hat{e}_L = [v_z, -v_x]/\|\vec{v}\|$, and lift coefficient $C_L = 2\pi\alpha\frac{10}{12}$. S_{ref} is the wing aerodynamic reference area. The drag force $\vec{F}_{\text{drag}}$ is

$$\vec{F}_{\text{drag}} = \frac{1}{2}\rho\|\vec{v}\|^2 C_D(\alpha) S_{\text{ref}}\, \hat{e}_D$$

Drag direction $\hat{e}_D = -\vec{v}/\|\vec{v}\|$, and drag coefficient $C_D = 0.01 + \frac{C_L^2}{\text{AR}\pi}$. The gravitational force is

$$\vec{F}_{\text{gravity}} = [0, m\, g]$$

Use AR = 10, $\rho = 1.2$, $g = 9.81$, $S_{\text{ref}} = 0.5$.

(a) Write the continuous time model in the form of

$$\frac{d}{dt}x = f(x, u) \tag{1.6}$$

(b) Simulate the system for 10 seconds using the `ode45` MATLAB function. Use $\alpha = 3°$, and initial conditions $p_x = p_z = v_z = 0$, $v_x = 10$. Plot p_x, p_z, v_x, v_z vs. time, and p_x vs. altitude.

(c) Convert the system to the discrete time form

$$x(k+1) = f_d(x(k), u(k))$$

using a forward Euler integrator. Simulate this system and compare to `ode45`. Estimating the accuracy by eye, how small do you have to make the time step so that results are similar accuracy to `ode45`? Using the MATLAB functions `tic` and `toc`, how much time does `ode45` take compared to forward Euler for similar accuracy?

(d) Re-do the previous item using 4th order Runge-Kutta (RK4) instead of forward Euler. Which is faster (for similar accuracy) among the three methods?

(e) Linearize the discrete time RK4 system to make an approximate system of the form

$$x(k+1) \approx f(\tilde{x}, \tilde{u}) + \underbrace{\frac{\partial f}{\partial x}(\tilde{x}, \tilde{u})}_{A}(x(k) - \tilde{x}) + \underbrace{\frac{\partial f}{\partial u}(\tilde{x}, \tilde{u})}_{B}(u(k) - \tilde{u})$$

using a first order Taylor expansion around the point $\tilde{x} = [10, 3, 11, 5]^\top$, $\tilde{u} = 5°$.

The Jacobian is given by

$$\frac{\partial f}{\partial x} = \left(\frac{\partial f}{\partial p_x}, \frac{\partial f}{\partial p_z}, \frac{\partial f}{\partial v_x}, \frac{\partial f}{\partial v_z}\right).$$

You can approximate the Jacobian by doing small variations in all

directions of x and u (finite differences). For example, in the direction of p_x the derivative $\frac{\partial f}{\partial p_x}$ is given by:

$$\frac{\partial f}{\partial p_x}(\tilde{x}, \tilde{u}) \approx \frac{f(\tilde{x} + [\delta, 0, 0, 0]^\top, \tilde{u}) - f(\tilde{x}, \tilde{u})}{\delta}.$$

(f) Plot the Eigenvalues of A in the complex plane. Is the system stable? Is this a problem?

1.3 **Introduction to CasADi 1:** CasADi is an open-source software tool for solving optimization problems in general and optimal control problems in particular. In its most typical usage, it leaves it to the user to formulate the problem as a standard form constrained optimization problem of the form:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & \underline{x} \le x \le \overline{x} \\ & \underline{g} \le g(x) \le \overline{g}, \end{aligned} \tag{1.7}$$

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

In this exercise, f is a convex quadratic function and g is a linear function, in which case we refer to problem (13.14) as a (convex) quadratic program (QP). To solve a QP with CasADi, start by creating a struct containing expressions for x, f and g:

- MATLAB:

```
x = SX.sym('x',n);
f = (some expression of x)
g = (some expression of x)
prob = struct('x',x,'f',f,'g',g);
```

- Python:

```
x = SX.sym('x',n)
f = (some expression of x)
g = (some expression of x)
prob = {'x':x,'f':f,'g':g}
```

This symbolic representation of the problem is then used to construct a QP solver as follows:

- MATLAB:

```
solver = qpsol('solver','qpoases',prob);
```

- Python:

```
solver = qpsol('solver','qpoases',prob)
```

where the arguments are, respectively, the *display name* of the solver `s`, the solver *plugin* – here the open-source QP solver qpOASES – and and the above symbolic problem formulation. A set of algorithmic options can be passed as an optional forth argument. Optimization solvers are *functions* in CasADi that are evaluated to get the solution:

- MATLAB:

```
res = solver('x0',x0,'lbx',lbx,'ubx',ubx,
   'lbg',lbg,'ubg',ubg);
```

- Python:

```
res = solver(x0:=0,lbx = lbx,ubx=ubx,
   lbg=lbg,ubg=ubg)
```

Where `lbx`, `ubx`, `lbg` and `ubg` are the bounds of x and $g(x)$ and `x0` is an initial guess for x (less important for convex QPs, since the solution is unique).

Exercise example: Hanging Chain We want to model a chain attached to two supports and hanging in between. Let us discretize it with N mass points connected by $N-1$ springs. Each mass i has position (y_i, z_i), $i = 1, \ldots, N$. The equilibrium point of the system minimises the potential energy. The potential energy of each spring is

$$V_{\text{el}}^i = \frac{1}{2} D_i \left((y_i - y_{i+1})^2 + (z_i - z_{i+1})^2 \right).$$

The gravitational potential energy of each mass is

$$V_{\text{g}}^i = m_i \, g_0 \, z_i.$$

The total potential energy is thus given by:

$$V_{\text{chain}}(y, z) = \frac{1}{2} \sum_{i=1}^{N-1} D_i \left((y_i - y_{i+1})^2 + (z_i - z_{i+1})^2 \right) + g_0 \sum_{i=1}^{N} m_i \, z_i, \quad (1.8)$$

where $y = [y_1, \cdots, y_N]^T$ and $z = [z_1, \cdots, z_N]^T$.

We wish to solve

$$\underset{y,z}{\text{minimize}} \qquad V_{\text{chain}}(y, z) \tag{1.9}$$

subject to constraints modeling the ground, to be introduced below.

(a) Go to the CasADi website and locate the user guide. Make sure the version of the user guide matches the version of CasADi used in the book (3.0.0). Then, with a Python or MATLAB interpreter in front of you, read Chapter 3 as well as Sections 4.1-4.3 in Chapter 4 of the user guide.

(b) From the course website, you will find solution scripts for Python and MATLAB that solve the unconstrained problem using $N = 40$, $m_i = 40/N$ kg, $D_i = 70N$ N/m, $g_0 = 9.81$ m/s^2 with the first and last mass point fixed to $(-2, 1)$ and $(2, 1)$, respectively. Go through the script and make sure you understand the steps.

(c) Introduce ground constraints: $z_i \geq 0.5$ and $z_i - 0.1\, y_i \geq 0.5$, for $i = 2, \cdots, N-2$. Solve your QP again, plot the result and compare it with the previous one.

2

Root-Finding with Newton-Type Methods

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

In this first part of the book we discuss several concepts from the field of numerical analysis and 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 [55]. 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 numerical analysis and optimization on the one hand, and those specific to optimal control on the other hand. We slightly adapt the notation, however, in order to prepare the interface to optimal control later.

In essence, optimization is about finding the inputs for some possibly nonlinear function that make the output of the function achieve some desired properties. In the simplest case, one demands that the function output should have a certain value, and assumes that the function has exactly as many inputs as it has outputs. Many problems in numerical analysis – in particular in optimization – can be formulated as such root-finding problems. Newton's method and its variants are at the basis of virtually all methods for their solution. Throughout this chapter, let us therefore consider a continuously differentiable function $R : \mathbb{R}^n \to \mathbb{R}^n$, $z \mapsto R(z)$, where our aim is to solve the nonlinear equation system

$$R(z) = 0.$$

Newton's idea was to start with an initial guess z_0, and recursively generate a

sequence of iterates $\{z_k\}_{k=0}^{\infty}$ by linearizing the nonlinear equation at the current iterate:

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

We can explicitly compute the next iterate by solving the linear system:

$$z_{k+1} = z_k - \left(\frac{\partial R}{\partial z}(z_k)\right)^{-1} R(z_k).$$

Note that we have to assume that the Jacobian $J(z_k) := \frac{\partial R}{\partial z}(z)$ is invertible.

More general, we can use an invertible approximation M_k of the Jacobian $\frac{\partial R}{\partial z}(z_k)$. The general Newton type iteration is

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

Depending on how closely M_k approximates $J(z_k)$, the local convergence can be fast or slow, or the sequence may even not converge.

Example 2.1. Regard $R(z) = z^{16} - 2$, where $\frac{\partial R}{\partial z}(z) = 16z^{15}$. The Newton method iterates:

$$z_{k+1} = z_k - (16z_k^{15})^{-1}(z_k^{16} - 2).$$

The iterates quickly converge to the solution z^* with $R(z^*) = 0$. In fact, the convergence rate of Newton's method is *q-quadratic*. Alternatively, we could use a Jacobian approximation, e.g. the constant value $M_k = 16$ corresponding to the true Jacobian at $z = 1$. The resulting iteration would be

$$z_{k+1} = z_k - (16)^{-1}(z_k^{16} - 2).$$

This approximate method might or might not converge. This might or might not depend on the initial value z_0. If the method converges, what will be its convergence rate? We investigate the conditions on $R(z)$, z_0 and M_k that we need to ensure local convergence in the following sections.

2.1 Local Convergence Rates

Definition 2.2 (Different types of convergence rates)**.** *Assume $z_k \in \mathbb{R}^n$, $z_k \to \overline{z}$. Then the sequence z_k is said to converge:*

(i) *Q-linearly* $\Leftrightarrow$

$$\|z_{k+1} - \overline{z}\| \leqslant C\|z_k - \overline{z}\| \text{ with } C < 1 \tag{2.1}$$

holds for all $k \geqslant k_0$. The "Q" in Q-linearly means the "Q" of "quotient". Another equivalent definition is:

$$\limsup_{k\to\infty} \frac{\|z_{k+1} - \overline{z}\|}{\|z_k - \overline{z}\|} < 1.$$

(ii) *Q-superlinearly* $\Leftrightarrow$

$$\|z_{k+1} - \overline{z}\| \leqslant C_k \|z_k - \overline{z}\| \text{ with } C_k \to 0.$$

This is equivalent to:

$$\limsup_{k\to\infty} \frac{\|z_{k+1} - \overline{z}\|}{\|z_k - \overline{z}\|} = 0.$$

(iii) *Q-quadratically* $\Leftrightarrow$

$$\|z_{k+1} - \overline{z}\| \leqslant C \|z_k - \overline{z}\|^2 \text{ with } C < \infty$$

which is equivalent to:

$$\limsup_{k\to\infty} \frac{\|z_{k+1} - \overline{z}\|}{\|z_k - \overline{z}\|^2} < \infty.$$

Example 2.3 (Convergence rates)**.** Consider examples with $z_k \in \mathbb{R}$, $z_k \to 0$ and $\overline{z} = 0$.

(i) $z_k = \frac{1}{2^k}$ converges q-linearly: $\frac{z_{k+1}}{z_k} = \frac{1}{2}$.
(ii) $z_k = 0.99^k$ also converges q-linearly: $\frac{z_{k+1}}{z_k} = 0.99$. This example converges very slowly to $\overline{z}$. In practice we desire C in equation (2.1) be smaller than, say, $\frac{1}{2}$.
(iii) $z_k = \frac{1}{k!}$ converges Q-superlinearly, as $\frac{z_{k+1}}{z_k} = \frac{1}{k+1}$.
(iv) $z_k = \frac{1}{2^{2^k}}$ converges Q-quadratically, because $\frac{z_{k+1}}{(z_k)^2} = \frac{(2^{2^k})^2}{2^{2^{k+1}}} = 1 < \infty$. For $k = 6$, $z^k = \frac{1}{2^{64}} \approx 0$, so in practice convergence up to machine precision is reached after roughly 6 iterations.

2.2 A Local Contraction Theorem

Theorem 2.4 (Local Contraction)**.** *Regard a nonlinear differentiable function $R : \mathbb{R}^n \to \mathbb{R}^n$ and a solution point $z^* \in \mathbb{R}^n$ with $R(z^*) = 0$, and the Newton type iteration $z_{k+1} = z_k - M_k^{-1} R(z_k)$ that is started at the initial value z_0. The sequence z_k converges to z^* with contraction rate*

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

if there exist $\omega < \infty$ *and* $\kappa < 1$ *such that for all* z_k *and* z *holds*

$$\|M_k^{-1}(J(z_k) - J(z))\| \leq \omega \|z_k - z\| \qquad \textit{(Lipschitz, or "omega", condition),}$$
$$\|M_k^{-1}(J(z_k) - M_k)\| \leq \kappa_k \leq \kappa \qquad \textit{(compatibility, or "kappa", condition)}$$

and if $\|z_0 - z^*\|$ *is sufficiently small, namely* $\|z_0 - z^*\| < \frac{2(1-\kappa)}{\omega}$.

Note: $\kappa = 0$ for exact Newton.

Proof

$$\begin{aligned}
z_{k+1} - z^* &= z_k - z^* - M_k^{-1} R(z_k) \\
&= z_k - z^* - M_k^{-1}(R(z_k) - R(z^*)) \\
&= M_k^{-1}(M_k(z_k - z^*)) \\
&\quad - M_k^{-1} \int_0^1 J(z^* + t(z_k - z^*))(z_k - z^*) \mathrm{d}t \\
&= M_k^{-1}(M_k - J(z_k))(z_k - z^*) \\
&\quad - M_k^{-1} \int_0^1 \Big[J(z^* + t(z_k - z^*)) - J(z_k) \Big](z_k - z^*) \mathrm{d}t.
\end{aligned}$$

Taking the norm of both sides:

$$\begin{aligned}
\|z_{k+1} - z^*\| &\leq \kappa_k \|z_k - z^*\| \\
&\quad + \int_0^1 \omega \|z^* + t(z_k - z^*) - z_k\| \mathrm{d}t \; \|z_k - z^*\| \\
&= \Big(\kappa_k + \omega \underbrace{\int_0^1 (1-t) \mathrm{d}t}_{=\frac{1}{2}} \|z_k - z^*\| \Big) \|z_k - z^*\| \\
&= \Big(\kappa_k + \frac{\omega}{2} \|z_k - z^*\| \Big) \|z_k - z^*\|.
\end{aligned}$$

Convergence follows from the fact that the first contraction factor, $\Big(\kappa_0 + \frac{\omega}{2}\|z_0 - z^*\|\Big)$ is smaller than $\delta := \Big(\kappa + \frac{\omega}{2}\|z_0 - z^*\|\Big)$, and that $\delta < 1$ due to the assumption $\|z_0 - z^*\| < \frac{2(1-\kappa)}{\omega}$. This implies that $\|z_1 - z^*\| \leq \delta \|z_0 - z^*\|$, and recursively that all following contraction factors will be bounded by δ, such that we have the upper bound $\|z_k - z^*\| \leq \delta^k \|z_0 - z^*\|$. This means that we have at least linear convergence with contraction rate δ. Of course, the local contraction rate will typically be faster than this, depending on the values of κ_k. □

Remark: The above contraction theorem could work with slightly weaker assumptions. First, we could restrict the validity of the "omega and kappa conditions" to a norm ball around the solution z^*, namely to the set $\{z \mid \|z - z^*\| <$

$\frac{2(1-\kappa)}{\omega}\}$. Second, in the omega and kappa conditions, we could have used slightly weaker conditions, as follows:

$$\|M_k^{-1}(J(z_k) - J(z_k + t(z^* - z_k)))(z^* - z_k)\| \leq \omega t \|z_k - z^*\|^2 \quad \text{(weaker } \omega \text{ cond.)}$$
$$\|M_k^{-1}(J(z_k) - M_k)(z_k - z^*)\| \leq \kappa_k \|z_k - z^*\| \quad \text{(weaker } \kappa \text{ cond.)}.$$

The above weaker conditions turn out to be invariant under affine transformations of the variables z as well as under linear transformations of the root finding residual function function $R(z)$. For this reason, they are in general preferable over the assumptions which we used the above theorem, which are only invariant under linear transformations of $R(z)$, but simpler to write down and to remember. Let us discuss the concept of affine invariance in the following section.

2.3 Affine Invariance

An iterative method to solve a root finding problem $R(z) = 0$ is called "affine invariant" if affine basis transformations of the equations or of the variables will not change the resulting iterations. This is an important property in practice. Regard, for example, the case where we would like to generate a method for finding an equilibrium temperature in a chemical reaction system. You can formulate your equations measuring the temperature in Kelvin, in Celsius or in Fahrenheit, which each will give different numerical values denoting the same physical temperature. Fortunately, the three values can be obtained by affine transformations from each other. For example, to get the value in Kelvin from the value in Celsius you just have to add the number 273.15, and for the transition from Celsius to Fahrenheit you have to multiply the Celsius value with 1.8 and add 32 to it. Also, you might think of examples where you indicate distances using kilometers or nanometers, respectively, resulting in very different numerical values that are obtained by a multiplication or division by the factor 10^{12}, but have the same physical meaning. The fact that the choice of units or coordinate system will result just in a affine transformation, applies to many other root finding problems in science and engineering. It is not unreasonable to ask that a good numerical method should behave the same if it is applied to problems formulated in different units or coordinate systems. This property we call "affine invariance".

More mathematically, given two invertible matrices $A, B \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, we regard the following root finding problem

$$\tilde{R}(y) := AR(b + By) = 0.$$

Clearly, if we have a solution z^* with $R(z^*) = 0$, then we can easily construct from it a y^* such that $\tilde{R}(y^*) = 0$, by inverting the relation $z^* = b + By^*$, i.e. $y^* = B^{-1}(z^* - b)$. Let us now regard an iterative method that, starting from an initial guess z_0, generates iterates $z_0, z_1, \ldots$ towards the solution of $R(z) = 0$. The method is called "affine invariant" if, when it is applied to the problem $\tilde{R}(y) = 0$ and started with the initial guess $y_0 = B^{-1}(z_0 - b)$ (i.e. the same point in the new coordinate system), it results in iterates $y_0, y_1, \ldots$ that all satisfy the relation $y_k = B^{-1}(z_k - b)$ for $k = 0, 1, \ldots$.

It turns out that the exact Newton method is affine invariant, and many other Newton type optimization methods like the Gauss-Newton method share this property, but not all. Practically speaking, to come back to the conversion from Celsius to Fahrenheit, Newton's method would perform exactly as well in America as in Europe. In contrast to this, some other methods, like for example the gradient method, would depend on the chosen units and thus perform different iterates in America than in Europe. More severely, a method that is not affine invariant usually needs very careful scaling of the model equations and decision variables in order to work well, while an affine invariant method works (usually) well, independent of the chosen scaling.

2.4 Tight Conditions for Local Convergence

The local contraction theorem of this chapter gives sufficient conditions for local convergence. Here, the omega condition is not restrictive, because ω can be arbitrarily large, and is satisfied on any compact set if the function R is twice continuously differentiable (ω is given by the maximum of the norm of the second derivative tensor, a continuous function, on the compact set). Also, we could start the iterations arbitrarily close to the solution, so the condition $\kappa + \frac{\omega}{2}\|z_0 - z^*\| < 1$ can always be met as long as $\kappa < 1$. Thus, the only really restrictive condition is the condition that the iteration matrices M_k should be similar enough to the true Jacobians $J(z_k)$, so that a $\kappa < 1$ exists. Unfortunately, the similarity measure of the kappa-condition might not be tight, so if we cannot find such a κ, it is not clear if the iterations converge or not.

In this section we want to formulate a sufficient condition for local convergence that is tight, and even find a necessary condition for local convergence of Newton-type methods. For this aim, we only have to make one assumption, namely that the iteration matrices M_k are given by a continuously differentiable matrix valued function $M : \mathbb{R}^n \to \mathbb{R}^{n\times n}$, i.e. that we have $M_k = M(z_k)$. This is for example the case for an exact Newton method, as well as for any method with fixed iteration matrix M (the function is just constant in this case).

It is also the case for the Gauss-Newton method for nonlinear least squares optimization. We need to use a classical result from nonlinear systems theory, which we will not prove here.

Lemma 2.5 (Linear Stability Analysis)**.** *Regard an iteration of the form $z_{k+1} = G(z_k)$ with G a continuously differentiable function in a neighborhood of a fixed point $G(z^*) = z^*$. If all Eigenvalues of the Jacobian $\frac{\partial G}{\partial z}(z^*)$ have a modulus smaller than one, i.e. if the spectral radius $\rho\left(\frac{\partial G}{\partial z}(z^*)\right)$ is smaller than one, then the fixed point is asymptotically stable and the iterates converge to z^* with a Q-linear convergence rate with asymptotic contraction factor $\rho\left(\frac{\partial G}{\partial z}(z^*)\right)$. On the other hand, if one of the Eigenvalues has a modulus larger than one, i.e. if $\rho\left(\frac{\partial G}{\partial z}(z^*)\right) > 1$, then the fixed point is unstable and the iterations can move away from z^* even if we have an initial guess z_0 that is arbitrarily close to z^*.*

Here, we use the definition of the spectral radius $\rho(A)$ of a square matrix A, as follows:

$$\rho(A) := \max\{|\lambda| \mid \lambda \text{ is Eigenvalue of } A\}.$$

We will not prove the lemma here, but only give some intuition. For this aim regard the Taylor series of G at the fixed point z^*, which yields

$$\begin{aligned} z_{k+1} - z^* &= G(z_k) - z^* \\ &= G(z^*) + \frac{\partial G}{\partial z}(z^*)(z_k - z^*) + O(\|z_k - z^*\|^2) - z^* \\ &= \frac{\partial G}{\partial z}(z^*)(z_k - z^*) + O(\|z_k - z^*\|^2). \end{aligned}$$

Thus, up to first order, the nonlinear system dynamics of $z_{k+1} = G(z_k)$ are determined by the Jacobian $A := \frac{\partial G}{\partial z}(z^*)$. A recursive application of the relation $(z_{k+1} - z^*) \approx A \cdot (z_k - z^*)$ yields $(z_k - z^*) = A^k \cdot (z_0 - z^*) + O(\|z_0 - z^*\|^2)$. Now, the matrix product A^k shrinks to zero with increasing k if $\rho(A) < 1$, and it grows to infinity if $\rho(A) > 1$.

When we apply the lemma to the continously differentiable map $G(z) := z - M(z)^{-1}R(z)$, then we can establish the following theorem, which is the main result of this section.

Theorem 2.6 (Sufficient and Necessary Conditions for Local Newton Type Convergence)**.** *Regard a Newton type iteration of the form $z_{k+1} = z_k - M(z_k)^{-1}R(z_k)$, where $R(z)$ is twice continuously differentiable with Jacobian $J(z)$ and $M(z)$ once continuously differentiable and invertible in a neighborhood of a solution z^* with $R(z^*) = 0$. If all Eigenvalues of the matrix $I - M(z^*)^{-1}J(z^*)$ have a*

modulus smaller than one, i.e. if the spectral radius

$$\kappa_{\text{exact}} := \rho\left(I - M(z^*)^{-1} J(z^*)\right)$$

is smaller than one, then this fixed point is asymptotically stable and the iterates converge to z^* with a Q-linear convergence rate with asymptotic contraction factor κ_{exact}. On the other hand, if $\kappa_{\text{exact}} > 1$, then the fixed point z^* is unstable.

Proof We prove the theorem based on the lemma, applied to the map $G(z) := z - M(z)^{-1}R(z)$. We first check that indeed $z^* = G(z^*)$, due to the fact that $R(z^*) = 0$. Second, we need to compute the Jacobian of G at z^*:

$$\begin{aligned}\frac{\partial G}{\partial z}(z^*) &= I - \frac{\partial (M^{-1})}{\partial z}(z^*) \underbrace{R(z^*)}_{=0} - M(z^*)^{-1}\frac{\partial R}{\partial z}(z^*) \\ &= I - M(z^*)^{-1} J(z^*).\end{aligned}$$

□

In summary, the spectral radius of the matrix $I - M(z^*)^{-1}J(z^*)$ is a tight criterion for local convergence. If it is larger than one, the Newton type method diverges, if it is smaller than one, the method converges.

Remark: The local contraction rate κ_{exact} directly depends on the difference between the exact and the approximate Jacobian, due to the trivial matrix identity

$$I - M(z^*)^{-1}J(z^*) = M(z^*)^{-1}(M(z^*) - J(z^*)).$$

For Newton's method itself, the two matrices are identical, $M(z^*) = J(z^*)$, and the linear contraction rate is zero: $\kappa_{\text{exact}} = 0$. This should be expected due to the fact that Newton's method converges quadratically. For Newton-type methods with nonzero κ_{exact}, the convergence will be linear only, but it can be very fast linear convergence if the approximate Jacobian is close to the exact one and $\kappa_{\text{exact}} \ll 1$. On the other hand, if the difference between the two matrices is too large, the spectral radius κ_{exact} might become larger than one, making the Newton-type method divergent.

2.5 Globalization

When the initial guess z_0 for starting a Newton-type iteration is too far from the solution, the iterates usually do not converge. In order to be able to reach the

region of local convergence, most Newton-type methods use a form of globalization for ensuring *global convergence*, i.e., convergence from any starting point. Here, we only give one example for a globalization technique for an exact Newton method, one that is based on line search.

Globalization by Armijo backtracking line search. To design a simple globalization procedure for a Newton method to solve $R(z) = 0$, we regard the function $V(z) = (1/2)\,\|R(z)\|^2$ as the merit function. Because its gradient is given by $\nabla V(z) = J(z)^\top R(z)$, the exact Newton step $p(z) := -J(z)^{-1}R(z)$ is a descent direction for any point with $R(z) \neq 0$, as can be seen by computing the scalar product $\nabla V(z)^\top p(z) = -R(z)^\top J(z)J(z)^{-1}R(z) = -\|R(z)\|^2 < 0$. This means that there exists a step length $\alpha \in (0, 1]$ such that $V(z + \alpha p(z)) < V(z)$. To ensure sufficient decrease of the merit function in each iteration, we can even impose the stronger *Armijo condition* that requires

$$V(z + \alpha p(z)) \leq V(z) + \alpha\gamma\nabla V(z)^\top p(z) \tag{2.2}$$

for some fixed $\gamma \in (0, 1/2)$, e.g., $\gamma = 0.01$. By choosing any step length α that satisfies the Armijo condition, one can prevent the iterates from jumping between points of nearly equal merit-function value without making progress. To prevent the steps from becoming infinitely small, one can use the backtracking algorithm. First, one checks if the step length $\alpha = 1$ satisfies the Armijo condition. If not, one reduces α by a constant factor, i.e., one reduces α to $\beta\alpha$ with a fixed value $\beta \in (0, 1)$, e.g., $\beta = 0.8$, and checks the Armijo condition again. If it is satisfied, one accepts the step length; if not, one reduces the value of α further, each time by the constant factor β. For descent directions and continuously differentiable merit functions, the backtracking algorithm always terminates and delivers a step length larger than zero. In fact, it delivers the largest value $\alpha \in \{1, \beta, \beta^2, \ldots\}$ that satisfies the Armijo condition (2.2). We denote the selected step length by $\alpha(z)$ in order to express its implicit dependence on z.

In summary, the globalized Newton's method iterates according to the system dynamics $z^+ = f(z)$ with $f(z) = z + \alpha(z)p(z)$. Note that while the merit function $V(z)$ is continuous and even differentiable, the discrete time system f is not continuous due to the state dependent switches in the backtracking procedure. Under mild assumptions on the function $R(z)$, one can ensure global convergence of the damped Newton procedure to a stationary point of the merit function, i.e., to a point z^* with $\nabla V(z^*) = 0$, which can either be a solution with $R(z^*) = 0$ or a point where $J(z^*)$ is singular.

Exercises

2.1 Sketch the root finding residual function $R : \mathbb{R} \to \mathbb{R}$, $R(z) := z^{16} - 2$ and its tangent at $z_0 = 1$, and locate the first Newton iterate z_1 in the graph.

2.2 For the root finding problem above, regard a Newton-type method with fixed iteration matrix $M := 20$ and locate the first Newton-type iterate in the graph. Also draw the corresponding Taylor-type approximation that is given by the linear function $\tilde{R}(z) := z_0 + M(z - z_0)$.

2.3 Define the iteration map $G(z) := z - M(z)^{-1}R(z)$ for $R(z) := z^{16} - 2$ with two different choices for M: first, with $M_{\text{Newton}}(z) = J(z)$ (exact Newton), and second, with $M_{\text{fixed}} := 20$ (fixed Jacobian approximation). Draw both iteration maps on the positive orthant. Also draw the diagonal line corresponding to the identity map, and sketch the first three Newton-type iterates for both methods.

2.4 As above, plot the iteration map for the fixed Jacobian method, but now for different values of M_{fixed}. For which values of M_{fixed} do you expect divergence? How would you justify your expectation analytically, and how can it be interpreted visually?

2.5 Write a computer program for Newton-type optimization in $\mathbb{R}^n$, that takes as inputs a function $F(z)$, a Jacobian approximation $M(z)$, and a starting point $z_0 \in \mathbb{R}^n$, and which outputs the first 20 Newton type iterations. Test your program with $R(z) = z^{16} - 2$ and exact Jacobian starting at different positive initial guesses. How many iterations do you typically need in order to obtain a solution that is exact up to machine precision?

2.6 An equivalent problem to $z^{16} - 2 = 0$ can be obtained by *lifting* it to a higher dimensional space [2], as follows:

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

Implement Newton's method for this lifted problem and start it at $z_0 = [1, 1, 1, 1]^\top$. Also implement the Newton method for the unlifted problem, and compare the convergence of the two algorithms.

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

2.8 Regard the problem of finding a solution to the nonlinear equation system $x = e^y$ and $x^4 + y^4 = 4$ in the two variables $x, y \in \mathbb{R}$. Sketch the solution sets to the two individual equations as curves in $\mathbb{R}^2$ and locate the intersection points. Now regard the solution of this system with Newton's method, initialized at the point $x = y = 2$. Based on the system linearization at this initial guess, sketch the solution sets of the two linear equations that define the first Newton iterate, and locate this iterate graphically.

2.9 Regard the two dimensional root finding problem from Question 2.8 above and solve it with your implementation of Newton's method from Question 2.5, using different initial guesses. Does it always converge, and if it converges, does it always converge to the same solution?

2.10 Consider the following optimization problem:

$$\underset{z}{\text{minimize}} \quad \underbrace{(1 - z_1^2) + 100\,(z_2 - z_1^2)^2}_{=:f(z)}$$

where the objective $f : \mathbb{R}^2 \to \mathbb{R}$ is the famous Rosenbrock function. A solution to this problem z^* can be obtained by solution of the nonlinear system $\nabla f(z) = 0$. Compute the gradient of the Rosenbrock function, $R(z) := \nabla f(z)$ and the Jacobian of R (i.e. the Hessian matrix of f) on paper. Implement Newton's method. Start with different initial guesses and observe the convergence.

2.11 Solve the previous exercise with the a simple Newton-type method, where you use a fixed Jacobian approximation that is given by $M = \begin{bmatrix} 200 & 0 \\ 0 & 200 \end{bmatrix}$.

2.12 A hanging chain can be modeled as a set of N balls (each with mass m) connected by $N - 1$ massless rods (each of length L). We assume that the two endpoints of the chain are fixed, and are interested in the equilibrium positions of all balls in between. In this exercise, we compute these positions by using the equilibrium of forces and Newton's method.

Applying the equilibrium conditions to each ball produces the following set of equations, for $i = 1, \ldots, N - 1$.

$$F_{i+1} = mg \begin{bmatrix} 0 \\ 1 \end{bmatrix} + F_i \tag{2.3}$$

where g is the gravitational acceleration and $F_i \in \mathbb{R}^2$ is defined as the force between the balls i and $i + 1$. On the other hand, considering the geometry of the chain, the following relation between the ball positions

p_i can be obtained:

$$p_{i+1} = p_i + L\frac{F_i}{||F_i||_2}. \tag{2.4}$$

Here, $p_i \in \mathbb{R}^2$ represents the position of the ball i. Assume that $N = 15$, $L_i = 1$ [m] and $m = 5$ [kg].

(a) Fixing the position of the first mass p_1 to $\begin{bmatrix} 0 \\ 10 \end{bmatrix}$, knowing the force F_1 and using Equations (2.3) and (2.4), we can create a forward map and compute all the forces F_i and positions p_i. Implement a function that uses as input F_1 and outputs the positions $p_1, \ldots, p_{15}$ of every mass.

(b) Now we want to fix also the position of the last mass p_{15} to $\begin{bmatrix} 10 \\ 10 \end{bmatrix}$. The function from the previous task generates p_{15} as a function of the initial force F_1. Form a root finding problem $R(z) = 0$, with $z := F_1$ and $R(z) := p_{15}(z) - \begin{bmatrix} 10 \\ 10 \end{bmatrix}$.

(c) In order to apply Newton's method to $R(z)$, we have to compute its derivative. Finite differences provide an easy method for this. Defining the Jacobian of $R(z)$ at a point z as $J(z)$, finite differences use the fact that:

$$J(z)p \approx \frac{R(z + \epsilon p) - R(z)}{\epsilon}$$

where we can use e.g. $\epsilon = 10^{-6}$. If using first $p = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and then $p = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, the Jacobian can be computed after three calls of $R(z)$. Implement the computation of the Jacobian of $J(z)$ at an arbitrary point z by finite differences.

(d) Implement Newton's method to obtain the F_1^* that satisfies the equilibrium of forces and solves the root finding problem. Use the forward map computed on the first task and plot the position of every mass under equilibrium conditions.

(e) Can you formulate and solve an equivalent "lifted" root finding problem for computing the rest position of the chain?

3
Nonlinear Optimization

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

The optimization problem with which we are concerned in this and the following chapters is the standard *Nonlinear Program (NLP)* that was already stated in the introduction:

$$\begin{aligned} \underset{w \in \mathbb{R}^n}{\text{minimize}} \quad & f(w) \\ \text{subject to} \quad & g(w) = 0, \\ & h(w) \leq 0, \end{aligned} \tag{3.1}$$

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 3.1 (Feasible set)**.** The *feasible set* Ω is the set

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

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

Definition 3.2 (Global minimum)**.** The point $w^* \in \mathbb{R}^n$ is a *global minimizer* if and only if (iff) $w^* \in \Omega$ and $\forall w \in \Omega :\ f(w) \geq f(w^*)$. The value $f(w^*)$ 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 3.3 (Local minimum)**.** The point $w^* \in \mathbb{R}^n$ is a *local minimizer* iff $w^* \in \Omega$ and there exists a neighborhood $\mathcal{N}$ of w^* (e.g., an open ball around w^*) so that $\forall w \in \Omega \cap \mathcal{N} : \ f(w) \geq f(w^*)$. The value $f(w^*)$ is a *local minimum*.

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

Definition 3.4 (Active Constraints and Active Set)**.** An inequality constraint $h_i(w) \leq 0$ is called *active* at $w^* \in \Omega$ iff $h_i(w^*) = 0$ and otherwise *inactive*. The index set $\mathcal{A}(w^*) \subset \{1, \ldots, n_h\}$ 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 (3.1) is very generic. In Section 3.1 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 3.2 and 3.3, respectively.

3.1 Important Special Classes

Linear Optimization

An obvious special case occurs when the functions f, g, and h in (3.1) are linear, resulting in a linear optimization problem (or Linear Program, LP)

$$\begin{aligned} \underset{w \in \mathbb{R}^n}{\text{minimize}} \quad & c^\top w \\ \text{subject to} \quad & Aw - b = 0, \\ & Cw - d \leq 0. \end{aligned}$$

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 3.2 are valid and can be used for algorithms, in particular interior point methods.

Quadratic Optimization

If in the general NLP formulation (3.1) 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.

$$\begin{aligned} \underset{w \in \mathbb{R}^n}{\text{minimize}} \quad & c^\top w + \frac{1}{2} w^\top B w \\ \text{subject to} \quad & Aw - b = 0, \\ & Cw - d \leq 0. \end{aligned} \tag{3.2}$$

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

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

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

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

Definition 3.6 (Convex Function)**.** A function $f : \Omega \to \mathbb{R}$ is convex, if Ω is convex and if

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

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 3.7 (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 (3.1) is convex if the function g is affine and the functions h_i are convex, as supported by the following theorem.

Theorem 3.8 (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 3.9 (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 3.10 (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 3.8 and 3.10 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 [7, 19]. Here we only mention an important fact that is related to the positive curvature of a function. Before we proceed, we introduce an important definition often used in this book.

Definition 3.11 (Generalized Inequality for Symmetric Matrices)**.** We write for a symmetric matrix $B = B^\top$, $B \in \mathbb{R}^{n \times n}$ that "$B \succcurlyeq 0$" if and only if B is *positive semi-definite* i.e., if $\forall z \in \mathbb{R}^n :\ z^\top B z \geq 0$, or, equivalently, if all (real) eigenvalues of the symmetric matrix B are non-negative:

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

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

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

Theorem 3.12 (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.,*

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

Again, we leave the proof as an exercise. As an example, the quadratic objective function $f(x) = c^\top x + \frac{1}{2}x^\top Bx$ of (3.2) is convex if and only if $B \succcurlyeq 0$, because $\forall x \in \mathbb{R}^n : \nabla^2 f(x) = B$.

3.2 First Order Optimality Conditions

An important question in continuous optimization is if a feasible point $w^* \in \Omega$ satisfies necessary first order optimality conditions. If it does not satisfy these conditions, w^* 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 3.13 (LICQ)**.** The *linear independence constraint qualification* (LICQ) holds at $w^* \in \Omega$ iff all vectors $\nabla g_i(w^*)$ for $i \in \{1, \ldots, n_g\}$ and $\nabla h_i(w^*)$ for $i \in \mathcal{A}(w^*)$ 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:

$$\tilde{g}(w) = \begin{bmatrix} g(w) \\ h_i(w)(i \in \mathcal{A}(w^*)) \end{bmatrix}. \tag{3.3}$$

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

The Karush-Kuhn-Tucker Optimality Conditions

This condition allows us to formulate the famous KKT conditions that are due to Karush [43] and Kuhn and Tucker [46].

Theorem 3.14 (KKT Conditions)**.** *If w^* is a local minimizer of the NLP* (3.1) *and LICQ holds at w^* then there exist so called multiplier vectors $\lambda \in \mathbb{R}^{n_g}$ and $\mu \in \mathbb{R}^{n_h}$ with*

$$\nabla f(w^*) + \nabla g(w^*)\lambda^* + \nabla h(w^*)\mu^* = 0 \tag{3.4a}$$
$$g(w^*) = 0 \tag{3.4b}$$
$$h(w^*) \leq 0 \tag{3.4c}$$
$$\mu^* \geq 0 \tag{3.4d}$$
$$\mu_i^* h_i(w^*) = 0, \quad i = 1, \ldots, n_h. \tag{3.4e}$$

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., $\nabla g(w) := \frac{\partial g}{\partial w}(w)^\top$. *Note:* The KKT conditions are the first order necessary conditions for optimality (FONC) for constrained optimization, and are thus the equivalent to $\nabla f(w^*) = 0$ 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 3.15. *Regard a convex NLP and a point w^* at which LICQ holds. Then:*

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

The Lagrangian Function

Definition 3.16 (Lagrangian Function)**.** We define the so called "Lagrangian function" to be

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

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 (3.4a) $\Leftrightarrow \nabla_w \mathcal{L}(w^*, \lambda^*, \mu^*) = 0$.

Remark 1: In the absence of inequalities, the KKT conditions simplify to $\nabla_w \mathcal{L}(w, \lambda) = 0$, $g(w) = 0$, 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 w. 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.

Complementarity

The last three KKT conditions (3.4c)-(3.4e) 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 $h_i(w^*) = 0$ 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 3.17. Regard a KKT point (w^*, 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.

3.2.1 Interpretation of the KKT conditions

It is extremely useful to equip ourselves with an interpretation of the KKT conditions (3.4). We present here the *physical* interpretation, where we see the KKT conditions as a *force balance* between the objective function and the constraints. It is easiest to construct this interpretation on a two-dimensional problem. The objective function can then be seen as a landscape with hills and depressions, and the optimal solution can be seen as a ”ball” rolling towards the lowest point in that landscape. The force exerted by the cost function on the solution corresponds to the *slope* of the cost function, i.e.:

$$-\nabla f(w^*).$$

In this picture, equality constraints can be seen as a ”rail” (or as a surface in dimensions higher than two) along which the ”ball” is forced to move. Inequality constraints can be seen as ”barriers” that divide the landscape and contain the ”ball” in a restrained domain. The constraints then exert forces on the ball, maintaining it on the rail and on the correct side of the barriers.

Equality constraints, the rail in our landscape, are described by the manifold $g(w) = 0$. The ”ball” is free to move along the rail but cannot leave it. The rail then exerts a force on the ”ball” only in directions orthogonal to the rail. Such directions are readily described by $\nabla g(w)$. The KKT condition (3.4a) for pure equality constraints reads as:

$$\nabla f(w^*) + \nabla g(w^*)\lambda^* = 0$$

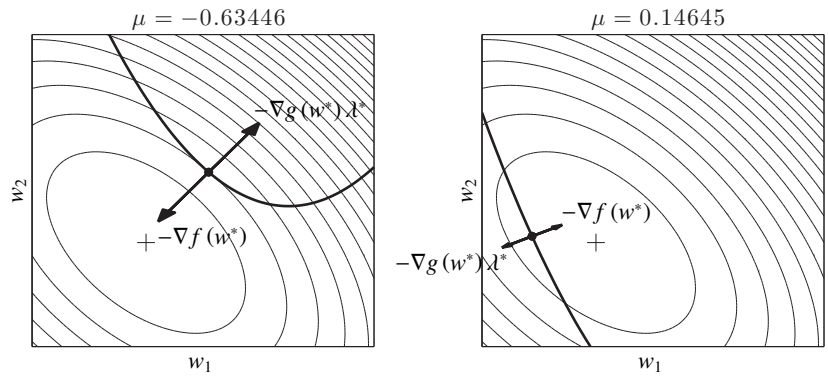


Figure 3.1 Illustration of the KKT conditions for an equality-constrained NLP. The "slope" of the cost function $-\nabla f(w)$ pushes the "ball" towards its lowest point. The "ball" is maintained on the "rail", i.e. the equality constraints $g(w) = 0$, via the force $-\nabla g(w)\lambda$, but is free to move along the rail. At the solution w^*, λ^*, the forces exerted by the rail and the cost function even out.

and prescribes that at the solution w^*, λ^*, the force exerted by the cost function $-\nabla f(w^*)$ and the force exerted by the rail i.e. $-\nabla g(w^*)\lambda^*$ are in balance. The rail will exert whatever force (in the orthogonal direction) is required to maintain the "ball" on the rail, hence the role of the Lagrange multipliers λ^* is to adjust the force of the rail in order to balance out the gradient of the cost function. This interpretation is illustrated in Figure 3.1.

Similarly, inequality constraints, the barriers in our landscape, are described by the manifold $h(w) \leq 0$, and can exert a force on the "ball" only in directions orthogonal to the barrier, i.e. $\nabla h(w)$, and *only* towards the interior of the feasible domain. The sign constraint (3.4b) on the Lagrange multipliers μ associated to the inequality constraints is then needed to ensure that the barrier can only "push" the "ball" into the feasible domain, but cannot force it to remain in contact with the barrier. The complementarity slackness condition (3.4e) essentially means that the barrier can exert a force on the "ball" if and only if the "ball" is in contact with the barrier. This interpretation is illustrated in Figure 3.2.

Finally the LICQ condition also has a physical interpretation. In the two-dimensional case, when the LICQ fails, some constraints exert forces that are collinear at the solution, resulting in infinite forces. This interpretation is illustrated in Figure 3.3.

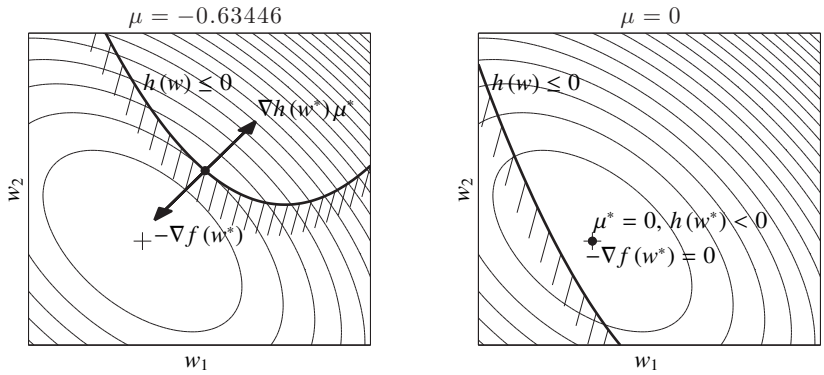


Figure 3.2 Illustration of the KKT conditions for an inequality-constrained NLP. The "slope" of the cost function $-\nabla f(w)$ pushes the solution towards its lowest point. The solution contained by the "barrier", i.e. the inequality constraints $h(w) \leq 0$ to remain within the feasible domain via the force $-\nabla h(w)\mu$, but is free to move along the barrier and towards the interior of the feasible domain. At the solution w^*, μ^*, the forces exerted by the barrier and the cost function even out. If the solution is in contact with the barrier, then the force is non-zero and pushes towards the interior of the feasible domain, i.e. $h(w^*) = 0$, $\mu > 0$ (left graph). Otherwise, the barrier exerts no force on the solution, i.e. $h(w^*) < 0$, $\mu = 0$ (right graph).

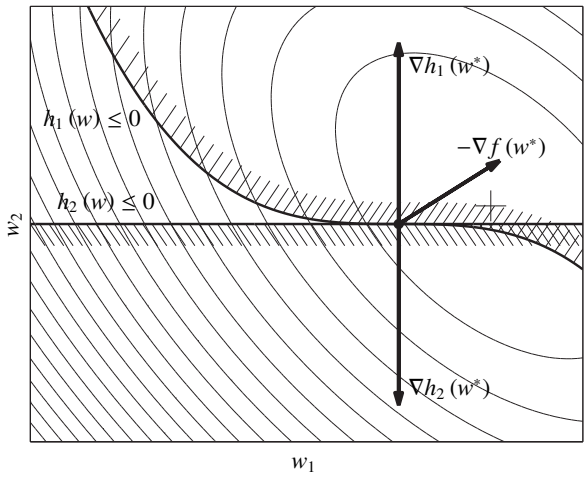


Figure 3.3 Failure of the LICQ condition. The optimal solution is not a KKT point. In this case, the forces exerted by the constraints $h_1(w)$ and $h_2(w)$ are collinear, and cannot balance the slope of the cost function $-\nabla f(w)$, even though the constraints prevent the solution from moving further toward the minimum of the cost function.

3.3 Second Order Optimality Conditions

In case of strict complementarity at a KKT point (w^*, 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 (3.3). 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 3.18 (Second Order Optimality Conditions)**.** *Let us regard a point w^* at which LICQ holds together with multipliers λ^*, μ^* so that the KKT conditions* (3.4a)-(3.4e) *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 $\frac{\partial \tilde{g}}{\partial w}(w^*) \in \mathbb{R}^{n_{\tilde{g}} \times n}$, i.e., Z has full column rank and $\frac{\partial \tilde{g}}{\partial w}(w^*)Z = 0$.*

Then the following two statements hold:

(a) If w^* is a local minimizer, then $Z^\top \nabla_w^2 \mathcal{L}(w^*, \lambda^*, \mu^*)Z \geqslant 0$.
(Second Order Necessary Condition, short : SONC)

(b) If $Z^\top \nabla_w^2 \mathcal{L}(w^*, \lambda^*, \mu^*)Z \succ 0$, then w^* 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 $\nabla_w^2 \mathcal{L}(w^*, \lambda^*, \mu^*)$ 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, $Z^\top \nabla_w^2 \mathcal{L}(w^*, \lambda^*, \mu^*)Z$, 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.

$$\begin{aligned} \underset{w \in \mathbb{R}^n}{\text{minimize}} \quad & c^\top w + \frac{1}{2} w^\top B w \\ \text{subject to} \quad & Aw + b = 0. \end{aligned}$$

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

$$\begin{array}{ccccccc} c & + & Bw & + & A^\top \lambda & = & 0 \\ b & + & Aw & & & = & 0. \end{array}$$

This is a linear equation system in the variable (w, 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 (w^*, 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 $Q = (Y \mid Z)$ square orthonormal and $R = (\bar{R}^\top \mid 0)^\top$. Then we can check if the reduced Hessian matrix $Z^\top BZ$ is positive semidefinite. If it is not, the objective function has negative curvature in at least one of the feasible directions and w^* cannot be a minimizer. If on the other hand $Z^\top BZ \succ 0$ then w^* 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 3.18 (b) hold, then it can be shown that the KKT-matrix

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

is invertible. This implies that the solution is stable against perturbations. To see why, let us regard a perturbed variant of the optimization problem (3.1)

$$\begin{aligned} \underset{w \in \mathbb{R}^n}{\text{minimize}} \quad & f(w) + \delta_f^\top w \\ \text{subject to} \quad & g(w) + \delta_g = 0, \\ & h(w) + \delta_h \leq 0, \end{aligned} \tag{3.5}$$

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 3.19 (SOSC implies Stability of Solutions)**.** *Regard the family of perturbed optimization problems* (3.5) *and assume that for $\delta = 0$ exists a local solution $(w^*(0), \lambda^*(0), \mu^*(0))$ that satisfies LICQ, the KKT condition, strict complementarity, and the second order sufficient condition of Theorem 3.18 (b). Then there exists an $\epsilon > 0$ so that for all $\|\delta\| \leq \epsilon$ exists a unique local solution $(w^*(\delta), \lambda^*(\delta), \mu^*(\delta))$ 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 $(w^*(\delta), \tilde{\lambda}^*(\delta))$ with respect to $(\delta_1, \tilde{\delta}_2)$ is given by

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

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

$$\begin{aligned} \nabla f(w^*(\delta)) + \frac{\partial \tilde{g}}{\partial w}(w^*)^\top \tilde{\lambda} + \delta_1 &= 0 \\ \tilde{g}(w^*(\delta)) + \tilde{\delta}_2 &= 0 \end{aligned}$$

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.

Multipliers as Shadow Costs of the Constraints

One immediate consequence of the above sensitivity result is that the gradient of the objective function $f(w^*(\delta))$ with respect to the perturbation parameter δ is due to the chain rule given by

$$\frac{\mathrm{d}}{\mathrm{d}\delta} f(w^*(\delta)) \Big|_{\delta=0}^\top = -\begin{bmatrix} \nabla_w^2 \mathcal{L}(w^*, \lambda^*, \mu^*) & \frac{\partial \tilde{g}}{\partial w}(w^*)^\top \\ \frac{\partial \tilde{g}}{\partial w}(w^*) & \end{bmatrix}^{-1} \begin{bmatrix} \nabla_w f(w^*) \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \tilde{\lambda}^* \end{bmatrix}$$

The last equality can be derived by noting that the KKT matrix is invertible, thus the solution unique, and that the gradient of the Lagrangian is zero at the solution. The interpretation of the result is twofold: first, due to the leading zeros, it can be seen that the objective value $f(w^*(\delta))$ is completely insensitive against perturbations δ_f in the gradient of the objective, or more general, in perturbations of the objective function. This remarkable observation is a consequence of the fact that the minimizer is in a flat region of the reduced objective, thus feasible changes in $w^*(\delta)$ do not change the objective up to first order. The second interpretation is equally interesting: the appearance of the multipliers in the gradient means that changes $\delta_{\tilde{g}}$ to the constraints lead directly to an increase or decrease in the cost $f(w^*(\delta))$. Thus, for positive multipliers, the cost increases for positive perturbations of the constraints, and the increase in the cost is directly given by the multiplier values $\tilde{\lambda}$. Note that this is consistent with the fact that the inequality multipliers are restricted to be positive: an increase in δ_h will tighten the inequality constraint, i.e., reduce the feasible set, such

that the objective function can only increase. Also note that the physical units of the multipliers are given by the objective unit divided by the corresponding constraint units, i.e., for an objective in Euro and a constraint that restricts some distance in meter, the multiplier would have the unit Euro/meter. This is the famous interpretation of multipliers as "shadow costs" of the constraints.

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 [55]. 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 [19], whose PDF is freely available.

Exercises

3.1 Consider the following NLP:

$$\begin{aligned} \underset{w \in \mathbb{R}^N}{\text{minimize}} \quad & \frac{1}{2} w^\top w \\ \text{subject to} \quad & N - w^\top w \leq 0. \end{aligned}$$

What is the solution of the above problem? Is it a KKT point ? Is it regular ? Does it fulfil the SOSC ? Justify and explain.

3.2 Solve the same questions of the previous tasks on the modified NLP:

$$\begin{aligned} \underset{w \in \mathbb{R}^2}{\text{minimize}} \quad & \frac{1}{2} w^\top w \\ \text{subject to} \quad & w_1 w_2 - 1 = 0, \\ & 2 - w^\top w \leq 0. \end{aligned}$$

3.3 A colleague of yours wants to solve the following problem:

$$\underset{w \in \mathbb{R}^2}{\text{minimize}} \quad w_1 + w_2 \tag{3.6a}$$

$$\text{subject to} \quad w_1 + w_2 = aw_1^2 + bw_2^2 + c \tag{3.6b}$$

with $a, b > 0$. He observes the equality constraint (3.6b) and the cost (3.6a) and concludes that solving (3.6) is equivalent to solving:

$$\underset{w \in \mathbb{R}^2}{\text{minimize}} \quad aw_1^2 + bw_2^2 + c$$

which takes the trivial solution $x,\ y = 0$. He then realizes that something is wrong with his approach, but he cannot explain what goes wrong. Help him.

3.4 Prove that the unconstrained optimization problem

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

with $f : \mathbb{R}^n \to \mathbb{R}$ a continuous, coercive function, has a global minimum point.

Hint: Use the Weierstrass Theorem and the following definition.

Definition (Coercive functions). A continuous function $f(x)$ that is defined on $\mathbb{R}^n$ is coercive if

$$\lim_{\|x\| \to \infty} f(x) = +\infty$$

or equivalently, if $\forall\ M\ \exists R : \|x\| > R \Rightarrow \|f(x)\| > M$.

3.5 Determine and explain whether the following functions are convex or not:

(a) $f(x) = c^\top x + x^\top A^\top A x$

(b) $f(x) = -c^\top x - x^\top A^\top A x$

(c) $f(x) = \log(c^\top x) + \exp(b^\top x)$

(d) $f(x) = -\log(c^\top x) - \exp(b^\top x)$

(e) $f(x_1, x_2) = 1/(x_1 x_2)$ on $\mathbb{R}^2_{++}$.

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

3.6 Determine and explain whether the following sets are convex or not:

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

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

$$B(x_c, r) = \{x \mid \|x - x_c\| \leq r\} = \{x \mid (x - x_c)^\top (x - x_c) \leq r^2\}$$

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

$$C = \{(x, t) \mid \| x \| \leq t\}$$

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

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

(e) A polyhedra:

$$\{x \in \mathbb{R}^n \mid Ax \leq b,\ Cx \leq d\}$$

(f) The set of points closer to one set than another:

$$C := \{x \in \mathbb{R}^n | \operatorname{dist}(x, \mathcal{S}) \leq \operatorname{dist}(x, \mathcal{T})\},$$
$$\text{with } \operatorname{dist}(x, \mathcal{S}) := \inf\{\|x - z\|_2 \mid z \in \mathcal{S}\}$$

3.7 Consider the following *mixed-integer quadratic program* (MIQP):

$$\begin{array}{ll} \underset{x \in \{0,1\}^n}{\text{minimize}} & x^\top Q x + q^\top x \\ \text{subject to} & Ax \geq b \end{array}$$

where the optimization variables x_i are restricted to take values in $\{0, 1\}$. Solving mixed-integer problems is in general a challenging task, thus it is common practice to reformulate them as the following:

$$\begin{array}{ll} \underset{x \in \{0,1\}^n}{\text{minimize}} & x^\top Q x + q^\top x \\ \text{subject to} & Ax \geq b, \\ & x_i(1 - x_i) = 0 \quad i = 0, \cdots, n-1. \end{array}$$

(a) Is this reformulation continuous?
(b) Is this reformulation convex?
(c) Is this reformulation a QP problem?
(d) Compute the Lagrangian function $\mathcal{L}(x, \lambda, \mu)$.

(e) Derive the first and second order optimality conditions for this specific problem.

3.8 Regard, first just on paper, the following NLP:

$$\begin{aligned} \underset{x \in \mathbb{R}^2}{\text{minimize}} \quad & x_2 \\ \text{subject to} \quad & x_1^2 + 4x_2^2 \leq 4, \\ & x_1 \geq -2, \\ & x_1 = 1 \end{aligned}$$

(a) How many degrees of freedom, how many equality, and how many inequality constraints does this problem have?

(b) Sketch the feasible set Ω of this problem. What is the optimal solution?

(c) Bring this problem into the NLP standard form

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

by defining the dimension n and the functions f, g, h along with their dimensions appropriately.

(d) Now formulate three MATLAB functions f, g, h for the above NLP, choose an initial guess for x, and solve the problem using `fmincon`. Check that the output corresponds to what you expected.

3.9 We want to model a chain attached to two supports and hanging in between. Let us discretise it with N mass points connected by $N-1$ springs. Each mass i has position (y_i, z_i), $i = 1, \ldots, N$. The equilibrium point of the system minimises the potential energy. We know that the potential energy of each spring is given by

$$V_{\text{el}}^i = \frac{1}{2} D_i \left((y_i - y_{i+1})^2 + (z_i - z_{i+1})^2 \right),$$

and that the gravitational potential energy of each mass by

$$V_{\text{g}}^i = m_i \, g_0 \, z_i.$$

As a result, the total potential energy is given by:

$$V_{\text{chain}}(y,z) = \frac{1}{2}\sum_{i=1}^{N-1} D_i\left((y_i - y_{i+1})^2 + (z_i - z_{i+1})^2\right) + g_0 \sum_{i=1}^{N} m_i\, z_i,$$

Considering $y = [y_1, \cdots, y_N]^\top$ and $z = [z_1, \cdots, z_N]^\top$, the problem that we wish to solve is given by:

$$\underset{y,z}{\text{minimize}} \qquad V_{\text{chain}}(y,z),$$

with optional additional inequality constraints which model a plane that the chain can not touch. This problem can be formulated by a QP as:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^\top H x + g^\top x \\ \text{subject to} \quad & x_{\text{lb}} \le x \le x_{\text{ub}}, \\ & a_{\text{lb}} \le Ax \le a_{\text{ub}} \end{aligned}$$

where $x = [y_1, z_1, \ldots, y_N, z_N]^\top$. In this representation, you get an equality constraint by having upper and lower bound equal, i.e. $a_{\text{lb}}^{(k)} = a_{\text{ub}}^{(k)}$ for some k.

(a) Formulate the problem using $N = 40$, $m_i = 4/N$ kg, $D_i = 70N$ N/m, $g_0 = 9.81$ m/s^2 with the first and last mass point fixed to $(-2, 1)$ and $(2, 1)$, respectively.

(b) Solve the problem using the function *quadprog* from MATLAB.

(c) Visualize the solution by plotting (y, z).

(d) Introduce ground constraints: $z_i \ge 0.5$ and $z_i - 0.1\, y_i \ge 0.5$. Solve your QP again and plot the result. Compare the result with the previous one.

(e) What would happen if you add instead of the piecewise linear ground constraints, the nonlinear ground constraints $z_i \ge y_i^2$ to your problem? The resulting problem is no longer a QP, but is it convex?

(f) What would happen if you add instead the nonlinear ground constraints $z_i \ge -y_i^2$ to your problem? Is the problem convex?

(g) **Introduction to CasADi 2:** Based on the template solution of Exercise 1.3, implement the above problem in CasaADi using IPOPT instead of qpOASES. To do that, call the function `nlpsol` instead of `qpsol` and leave the rest identical:

- MATLAB

```
solver = nlpsol('solver','ipopt',prob);
```

- Python

```
solver = nlpsol('solver','ipopt',prob)
```

3.10 The following function has multiple local minima in the domain $[-1, 1]\times[-1, 1]$:

$$f(x, y) = \exp(-x^2 - y^2)\ \sin(4\,(x + y + x * y^2))$$

(a) Plot and visualize the function in $[-1, 1] \times [-1, 1]$.

(b) Find the unconstrained minimizer of the function starting at different initial points, e.g. $[0, 0]$, $[0.9, 0, 9]$, $[-0.9, -0, 9]$. Use the function *fminunc* from MATLAB. What do you see?

3.11 Using the same aircraft model from Exercise 1.2, we provide a set of real measurements of an aircraft's flight. This data set contains position measurements $\hat{p}_{x,k}$ and $\hat{p}_{z,k}$ but not velocity. Since it's possible to measure some aircraft parameters with a scale and ruler, you know that mass is 2.5, S_{sref} is 0.7, and aspect ratio AR is 14. You don't know the angle of attack α or initial state x_0 so they need to be estimated.

(a) Dowload the file `flight_data.m` from the book website to obtain the dataset. Plot the noisy measurements as a function of time considering that the measurements were recorded during a time interval of 20 s.

(b) For Exercise 1.2, you wrote a simulation function which you can think of as taking initial state x_0 and angle of attack α as inputs, and returning the simulated states over the trajectory $\bar{x}_k = [\bar{p}_{x,k}|\bar{p}_{z,k}|\bar{v}_{x,k}|\bar{v}_{z,k}], k = 0 \ldots N-1$ as outputs:

$$[\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_{N-1}] = f_{\text{sim}}(x_0, \alpha)$$

Estimate angle of attack α and initial state x_0 by solving the following NLP:

$$\min_{x_0,\alpha} \sum_{k=0}^{N-1} (\bar{p}_{x,k}(x_0, \alpha) - \hat{p}_{x,k})^2 + (\bar{p}_{z,k}(x_0, \alpha) - \hat{p}_{z,k})^2$$

Use the RK4 fixed-step integrator from Exercise 1.2. You may need to adjust the initial guess in order to find the correct local minimum. A good way to tweak the initial guess is to simulate and plot the simulated trajectory against the data. You may also need to play with bounds on your design variables.

(c) Plot the final estimated trajectory against the noisy data.

3.12 **Introduction to CasADi 3:** Recalling the Rosenbrock problem from Exercise 2.10:

(a) Formulate and solve the following version:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x_1^2 + 100\, x_3^2 \\ \text{subject to} \quad & x_3 + (1 - x_1)^2 - x_2 = 0 \end{aligned}$$

Using IPOPT and $x = [2.5, 3.0, 0.75]$ as a starting point. How many iterations does the solver need to converge to the solution? Does it change if we instruct IPOPT to use a limited-memory BFGS approximation? This can be done by passing the following options dictionary as the forth argument to `nlpsol`:

- MATLAB

```
opts = struct;
opts.ipopt.hessian_approximation = 'limited-memory';
```

- Python

```
opts = {'ipopt.hessian_approximation': 'limited-memory'}
```

(b) Manually eliminate x_3 from the problem formulation using the constraint equation and resolve the now unconstrained problem with only two variables. How does the number of iterations change?

(c) **Nonlinear root-finding problems in CasADi** A special case of an NLP is a root-finding problem. We will write them in the form:

$$\begin{aligned} g_0(z, x_1, x_2, \ldots, x_n) &= 0 \\ g_1(z, x_1, x_2, \ldots, x_n) &= y_1 \\ g_2(z, x_1, x_2, \ldots, x_n) &= y_2 \\ &\vdots \\ g_m(z, x_1, x_2, \ldots, x_n) &= y_m, \end{aligned}$$

where the first equation uniquely defines z as a function of $x_1, \ldots, x_n$ by the *implicit function theorem* and the remaining equations define the auxiliary outputs $y_1, \ldots, y_m$. Given a function g for evaluating g_0, $\ldots$, g_m, we can use CasADi to automatically formulate a (differentiable) function $G : \{z_{\text{guess}}, x_1, x_2, \ldots, x_n\} \to \{z, y_1, y_2, \ldots, y_m\}$. This function includes a guess for z to handle the case when the solution is non-unique. The syntax for this, assuming $n = m = 1$, is:

- MATLAB

```
z = SX.sym('x',nz);
x = SX.sym('x',nx);
g0 = (some expression of x and z)
g1 = (some expression of x and z)
g = Function('g', {z, x}, {g0, g1});
G = rootfinder('G', 'newton', g);
```

- Python

```
z = SX.sym('x',nz)
x = SX.sym('x',nx)
g0 = (some expression of x & z)
g1 = (some expression of x & z)
g = Function('g', [z, x], [g0, g1])
G = rootfinder('G', 'newton', g)
```

where the `rootfinder` function, similar to `nlpsol` and `qpsol`, expects a display name, the name of a solver plugin (here a simple full-step Newton method) and the problem formulation, here expressed as a residual function.

Starting with the unconstrained version of the Rosenbrock problem use CasADi's `gradient` function to get a new expression for the gradient of the objective function. According to the first order necessary conditions for optimality, this gradient must be zero. Formulate and solve this as a root-finding problem in CasADi. Use the same initial condition as before.

4

Newton-Type Optimization Algorithms

It can be programmed in an afternoon if one has a quadratic programing subroutine available [...]
— Michael J. D. Powell (1936-2015), [29]

4.1 Equality Constrained Optimization

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

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

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

$$\begin{aligned} \nabla_w \mathcal{L}(w, \lambda) = \quad & 0 \\ g(w) = \quad & 0. \end{aligned}$$

In order to simplify notation, we define

$$z := \begin{bmatrix} w \\ \lambda \end{bmatrix} \text{ and } F(z) := \begin{bmatrix} \nabla_w \mathcal{L}(w, \lambda) \\ g(w) \end{bmatrix}$$

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

$$F(z) = 0.$$

Starting from an initial guess z_0, Newton's method generates a sequence of iterates $\{z_k\}_{k=0}^{\infty}$ by linearizing the nonlinear equation at the current iterate

$$F(z_k) + \frac{\partial F}{\partial z_k}(z_k)(z - z_k) = 0 \tag{4.1}$$

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

$$z_{k+1} = z_k - \frac{\partial F}{\partial z_k}(z_k)^{-1} F(z_k).$$

For equality constrained optimization, the linear system (4.1) has the specific form[1]

$$\begin{bmatrix} \nabla_w \mathcal{L}(w_k, \lambda_k) \\ g(w_k) \end{bmatrix} + \underbrace{\begin{bmatrix} \nabla_w^2 \mathcal{L}(w_k, \lambda_k) & \nabla g(w_k) \\ \nabla g(w_k)^\top & 0 \end{bmatrix}}_{\text{KKT-matrix}} \begin{bmatrix} w - w_k \\ \lambda - \lambda_k \end{bmatrix} = 0.$$

Using the definition

$$\nabla_w \mathcal{L}(w_k, \lambda_k) = \nabla f(w_k) + \nabla g(w_k)\lambda_k$$

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

$$\begin{bmatrix} \nabla f(w_k) \\ g(w_k) \end{bmatrix} + \begin{bmatrix} \nabla_w^2 \mathcal{L}(w_k, \lambda_k) & \nabla g(w_k) \\ \nabla g(w_k)^\top & 0 \end{bmatrix} \begin{bmatrix} w - w_k \\ \lambda \end{bmatrix} = 0.$$

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 $B_k \approx \nabla_w^2 \mathcal{L}(w_k, \lambda_k)$, we finally obtain the Newton-type iteration as

$$\begin{bmatrix} w_{k+1} \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} w_k \\ 0 \end{bmatrix} - \begin{bmatrix} B_k & \nabla g(w_k) \\ \nabla g^\top(w_k) & 0 \end{bmatrix}^{-1} \begin{bmatrix} \nabla f(w_k) \\ g(w_k) \end{bmatrix}. \tag{4.2}$$

The general *Newton-type method* is summarized in Algorithm 4.1. If we use $B_k = \nabla_w^2 \mathcal{L}(w_k, \lambda_k)$, we recover the *exact Newton method*.

Algorithm 4.1 (Equality constrained full step Newton-type method)**.**

Choose: initial guesses w_0, λ_0, and a tolerance ϵ

Set: $k = 0$

while $\|\nabla \mathcal{L}(w_k, \lambda_k)\| \geq \epsilon$ or $\|g(w_k)\| \geq \epsilon$ **do**

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

obtain a Hessian approximation B_k
get w_{k+1}, λ_{k+1} from (4.2)
$k = k + 1$
end while

4.1.1 Quadratic Model Interpretation

It is easy to show that w_{k+1} and λ_{k+1} from (4.2) can equivalently be obtained from the solution of a QP:

$$\begin{aligned} \underset{w \in \mathbb{R}^n}{\text{minimize}} \quad & \nabla f(w_k)^\top (w - w_k) + \frac{1}{2}(w - w_k)^\top B_k (w - w_k) \\ \text{subject to} \quad & g(w_k) + \nabla g(w_k)^\top (w - w_k) = 0. \end{aligned} \tag{4.3}$$

So we can interpret the Newton-type optimization method as a "Sequential Quadratic Programming" (SQP) method, where we find in each iteration the solution w^{QP} and λ^{QP} of the above QP and take it as the next NLP solution guess and linearization point w_{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.

4.1.2 The Exact Newton Method

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

$$B_k := \nabla^2_w \mathcal{L}(w_k, \lambda_k).$$

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

$$\nabla^2_w \mathcal{L}(w_k, \lambda_k) e_i = \frac{\nabla_w \mathcal{L}(w_k + \delta e_i, \lambda_k) - \nabla_w \mathcal{L}(w_k, \lambda_k)}{\delta} + O(\delta). \tag{4.4}$$

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 $\nabla_w \mathcal{L}$ can be obtained. Thus, we loose half the valid digits. If $\nabla_w \mathcal{L}$ 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* in a neighbourhood of the optimal solution z^*, i.e. $\|z_{k+1} - z^*\| \leq \frac{\omega}{2}\|z_k - z^*\|^2$ when z_k is sufficiently close to z^*. 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.

4.1.3 The Constrained Gauss-Newton Method

Let us regard the special case that the objective $f(w)$ has a nonlinear least-squares form, i.e. $f(w) = \frac{1}{2}\|R(w)\|_2^2$ 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

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

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

$$\underset{w \in \mathbb{R}^n}{\text{minimize}} \quad \frac{1}{2}\|R(w_k) + \nabla R(w_k)^\top (w - w_k)\|_2^2 \tag{4.5a}$$

$$\text{subject to} \quad g(w_k) + \nabla g(w_k)^\top (w - w_k) = 0. \tag{4.5b}$$

This is a convex QP which can easily be seen by noting that the objective (4.5a) is equal to

$$\frac{1}{2}R(w_k)^\top R(w_k) + (w - w_k)^\top \underbrace{\nabla R(w_k) R(w_k)}_{=\nabla f(w_k)} + \frac{1}{2}(w - w_k)^\top \underbrace{\nabla R(w_k) \nabla R(w_k)^\top}_{=:B_k} (w - w_k)$$

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 (4.5) is identical to the SQP subproblem (4.3) with a special choice of the Hessian approximation, namely

$$B_k := \nabla R(w_k) \nabla R(w_k)^\top = \sum_{i=1}^{n_R} \nabla R_i(w_k) \nabla R_i(w_k)^\top.$$

Note that the multipliers λ_k are not needed in order to compute the Gauss-Newton Hessian approximation 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

$$\begin{aligned}\nabla_w^2 \mathcal{L}(w, \lambda) &= \sum_{i=1}^{n_R} \nabla R_i(w_k) \nabla R_i(w_k)^\top + \sum_{i=1}^{n_R} R_i(w) \nabla^2 R_i(w) + \sum_{i=1}^{n_g} \lambda_i \nabla^2 g_i(w) \\ &= B_k + O(\|R(w_k)\|) + O(\|\lambda\|).\end{aligned}$$

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

Local convergence rate: The Gauss-Newton method converges *linearly*, $\|z_{k+1} - z^*\| \leq \kappa \|z_k - z^*\|$ with a contraction rate $\kappa = O(\|R(w^*)\|)$ in a neighbourhood of the solution z^*. Thus, it converges fast if the residuals $R_i(w^*)$ 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.

4.1.4 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 $\nabla_w \mathcal{L}$ at different points can deliver curvature information that can help us to estimate $\nabla_w^2 \mathcal{L}$, similar as it can be done by finite differences, cf. Equation (4.4), but without any extra effort per iteration besides the evaluation of $\nabla f(w_k)$ and $\nabla g(w_k)$ 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 $s_k := w_{k+1} - w_k$ and the gradient difference $y_k := \nabla_w \mathcal{L}(w_{k+1}, \lambda_{k+1}) - \nabla_w \mathcal{L}(w_k, \lambda_{k+1})$ in order to obtain the next Hessian approximation B_{k+1}. As in the finite difference formula (4.4), this approximation shall satisfy the *secant condition*

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

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:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^\top B_k}{s_k^\top B_k s_k} + \frac{y_k y_k^\top}{s_k^\top y_k}. \tag{4.6}$$

Local convergence rate: It can be shown that $B_k \to \nabla_w^2 \mathcal{L}(w^*, \lambda^*)$ in the relevant directions, so that *superlinear convergence* is obtained with the BFGS method in a neighbourhood of the solution z^*, i.e. $\|z_{k+1} - z^*\| \leq \kappa_k \|z_k - z^*\|$ with $\kappa_k \to 0$.

4.2 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 4.2 (Newton-Type Convergence)**.** *Regard the root finding problem*

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

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

$$\|M_k^{-1}(J(z_k) - J(z))\| \leq \omega \|z_k - z^*\|.$$

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

$$\|M_k^{-1}(J(z_k) - M_k)\| \leq \kappa_k$$

where $\kappa_k \leq \kappa$ with $\kappa < 1$. Finally, we assume that the initial guess z_0 is sufficiently close to the solution z^*,

$$\|z_0 - z^*\| \leq \frac{2}{\omega}(1 - \kappa).$$

Then $z_k \to z^*$ with the following linear contraction in each iteration:

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

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

Noting that in Newton-type optimization we have

$$J(z_k) = \begin{bmatrix} \nabla_w^2 \mathcal{L}(w_k, \lambda_k) & \frac{\partial g}{\partial w}(w_k)^\top \\ \frac{\partial g}{\partial w}(w_k) & 0 \end{bmatrix}$$
$$M_k = \begin{bmatrix} B_k & \frac{\partial g}{\partial w}(w_k)^\top \\ \frac{\partial g}{\partial w}(w_k) & 0 \end{bmatrix}$$
$$J(z_k) - M_k = \begin{bmatrix} \nabla_w^2 \mathcal{L}(\cdot) - B_k & 0 \\ 0 & 0 \end{bmatrix}$$

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

Corollary 4.3. *Newton-type optimization methods converge*

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

Proof of Theorem 4.2

We will show that $\|z_{k+1} - z^*\| \leq \delta_k \|z_k - z^*\|$ with $\delta_k := \left(\kappa_k + \frac{\omega}{2}\|z_k - z^*\|\right)$ and that for all k holds $\delta_k < 1$. For this aim let us regard

$$\begin{aligned} z_{k+1} - z^* &= z_k - z^* - M_k^{-1} F(z_k) \\ &= z_k - z^* - M_k^{-1}(F(z_k) - F(z^*)) \\ &= M_k^{-1}(M_k(z_k - z^*)) - M_k^{-1} \int_0^1 J(z^* + t(z_k - z^*))(z_k - z^*) dt \\ &= M_k^{-1}(M_k - \nabla^2 f(z_k))(z_k - z^*) \\ &\quad - M_k^{-1} \int_0^1 \left[\nabla^2 f(z^* + t(z_k - z^*)) - \nabla^2 f(z_k)\right](z_k - z^*) dt. \end{aligned}$$

Taking the norm of both sides:

$$\begin{aligned} \|z_{k+1} - z^*\| &\leq \kappa_k \|z_k - z^*\| + \int_0^1 \omega \|z^* + t(z_k - z^*) - z_k\| dt \quad \|z_k - z^*\| \\ &= \left(\kappa_k + \omega \underbrace{\int_0^1 (1-t) dt}_{=\frac{1}{2}} \|z_k - z^*\|\right) \|z_k - z^*\| \\ &= \underbrace{\left(\kappa_k + \frac{\omega}{2}\|z_k - z^*\|\right)}_{=\delta_k} \|z_k - z^*\|. \end{aligned}$$

The proof that for all k we have that $\delta_k < 1$ proceeds inductively: as $\delta_0 < 1$ by the assumptions of Theorem 4.2, we can conclude that $\|z_1 - z^*\| \leq \|z_0 - z^*\|$. This in turn implies that $\delta_1 \leq \delta_0$. The same reasoning can be made for each of the following steps, implying that all $\delta_k < 1$. Thus, the proof is nearly complete. To obtain the specific convergence rates, we distinguish three cases depending on the value of κ respectively κ_k:

(i) $\|z_{k+1} - z^*\| \leq \frac{\omega}{2}\|z_k - z^*\|^2$, Q-quadratic convergence if $\kappa = 0$,

(ii) $\|z_{k+1} - z^*\| \leq \underbrace{(\kappa_k + \frac{\omega}{2}\|z_k - z^*\|)}_{\to 0}\|z_k - z^*\|$, Q-superlinear if $\kappa_k \to 0$,

(iii) $\|z_{k+1} - z^*\| \leq (\underbrace{\kappa}_{<1} + \underbrace{\frac{\omega}{2}\|z_k - z^*\|}_{\to 0})\|z_k - z^*\|$, Q-linear if κ_k do not converge to zero.

4.3 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 (3.4) which include the non-smooth complementarity conditions, but have different ways to deal with this non-smoothness.

4.3.1 Interior Point Methods

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

$$\nabla f(w^*) + \nabla g(w^*)\lambda^* + \nabla h(w^*)\mu^* = 0 \tag{4.7a}$$

$$g(w^*) = 0 \tag{4.7b}$$

$$\mu_i^* h_i(w^*) + \tau = 0, \quad i = 1, \ldots, n_h. \tag{4.7c}$$

Note that the last equation ensures that $-h_i(w^*)$ 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

[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.

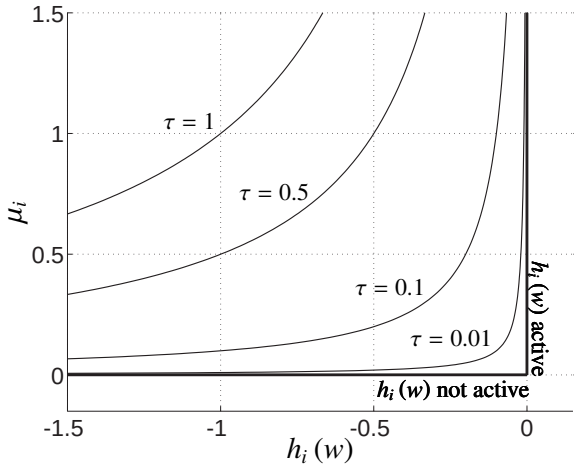


Figure 4.1 Relaxation of the complementarity slackness condition. We display here the manifold $\mu_i h_i(w) + \tau = 0$ for various values of τ. The original non-smooth manifold $\mu_i h_i(w) = 0$ arising in the KKT conditions is displayed as the thick lines.

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 $\mu_i^* = -\frac{\tau}{h_i(w^*)}$ and to insert this expression into the first equation, and to note that $\nabla_w(\log(-h_i(w))) = \frac{1}{h_i(w)}\nabla h_i(w)$). Thus, the above smooth form of the KKT conditions is nothing else than the optimality conditions of a *barrier problem*

$$\begin{aligned} \underset{w \in \mathbb{R}^n}{\text{minimize}} \quad & f(w) - \tau \sum_{i=1}^{n_h} \log(-h_i(w)) \\ \text{subject to} \quad & g(w) = 0. \end{aligned} \tag{4.8}$$

Note that the objective function of this problem tends to infinity when $h_i(w) \to 0$. 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. One can observe that the barrier problem (4.8) and the primal-dual (4.7) deliver the same solution w_τ for any given value of τ. It is also important to know that the error between

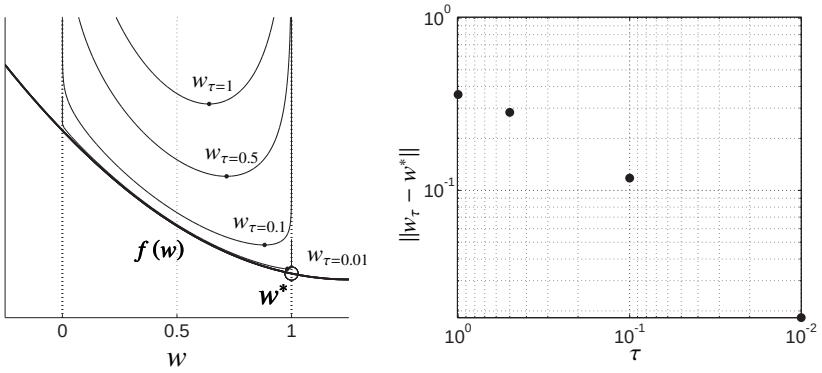


Figure 4.2 Illustration of the primal barrier method presented in (4.8). The left graph displays an illustrative cost function $f(w)$ (thick curve), and simple bounds $0 \leq w \leq 1$. The various objective functions with barrier $f(w) - \tau \sum_{i=1}^{n_h} \log(-h_i(w))$ are displayed for various values of τ, alongside their respective minima w_τ. The right graph displays the error between the actual solution to the problem w^*, and the solutions w_τ obtained from the barrier problem (4.8) for various values of τ.

the solution delivered by Interior-Point methods and the exact solution of the original problem is of the order $\mathcal{O}(\tau)$, i.e. the error introduced by the Interior-Point methods decreases linearly with τ.

Though easy to implement and to interpret, Interior-Point methods are not necessarily the best in terms of 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 (4.7) 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.

Most IP solvers treat the relaxed complementarity conditions (4.7c) using a slack formulation, where a set of "artificial" or *slack* variables s_i, $i = 1, ..., n_h$

is added to the problem in order to reformulate it. The equivalent system:

$$\nabla f(w^*) + \nabla g(w^*)\lambda^* + \nabla h(w^*)\mu^* = 0 \tag{4.9a}$$
$$g(w^*) = 0 \tag{4.9b}$$
$$\mu_i^* s_i^* - \tau = 0, \quad i = 1, \ldots, n_h \tag{4.9c}$$
$$h_i(w^*) + s_i^* = 0, \quad i = 1, \ldots, n_h \tag{4.9d}$$

is solved instead of (4.7). Though the form (4.9) is equivalent to (4.7) and delivers the same solution, it offers several advantages over (4.7), in particular:

- the Newton iteration on system (4.9) can be started with an initial guess w that is infeasible with respect to the inequality constraints, i.e. $h_i(w) > 0$ for some i, as long as the slack variables s_i are initiated and kept positive throughout the iterations. Hence one does not need to provide a feasible initial guess. In the course of the Newton iterations, the inequality constraints are brought to feasibility via the equality constraints (4.9d).
- when a Newton iteration is deployed on system (4.7), one must ensure that $h(w) < 0$ thoughout the iterations, which requires a careful backtracking, i.e. a reduction of the size of the step provided by the Newton iteration (see Section 4.4 for more details) until $h(w) < 0$ is ensured. When $h(w)$ is expensive to evaluate, such backtracking can be time consuming. In contrast, ensuring that $s > 0$, $\mu > 0$ is trivial to do when the form (4.9) is used. The step-size ensuring the positivity of s and μ then provides an inexpensive upper-bound to the actual step-size that ought to be used.

Software: A very widespread and successful implementation of the nonlinear IP method is the open-source code IPOPT [68, 67]. 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.

4.3.2 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:

$$\begin{aligned} \underset{w \in \mathbb{R}^n}{\text{minimize}} \quad & \nabla f(w_k)^\top (w - w_k) + \frac{1}{2}(w - w_k)^\top B_k (w - w_k) \\ \text{subject to} \quad & g(w_k) + \nabla g(w_k)^\top (w - w_k) = 0, \\ & h(w_k) + \nabla h(w_k)^\top (w - w_k) \leq 0. \end{aligned}$$

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 w^* once the SQP iterates w_k are in the neighborhood of w^*.

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 $B_k = \nabla^2_w \mathcal{L}(w_k, \lambda_k, \mu_k)$, and it can be shown that under the second order sufficient conditions (SOSC) of Theorem 3.18 (b), this method has locally quadratic convergence. Second, in the case of a least-squares objective $f(w) = \frac{1}{2}\|R(w)\|_2^2$, we can use the Gauss-Newton Hessian approximation $B_k = \nabla R(w_k) \nabla R(w_k)^\top$, yielding linear convergence with a contraction rate $\kappa = O(\|R(w^*)\|)$. Third, quasi-Newton updates such as BFGS can directly be applied, using the Lagrange gradient difference $y_k := \nabla_w \mathcal{L}(w_{k+1}, \lambda_{k+1}, \mu^{k+1}) - \nabla_w \mathcal{L}(w_k, \lambda_{k+1}, \mu^{k+1})$ in formula (4.6).

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 [36]. Many optimal control packages such as MUSCOD-II [48] or the open-source package ACADO [41, 1] contain at their basis structure exploiting SQP methods. Also the MATLAB solver `fmincon` is based on an SQP algorithm.

4.3.3 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 [24] 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) [33].

4.4 Globalization 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. Indeed, the Newton iteration using the SQP approach is based on solving successive quadratic problems which approximate locally the original problem. The Newton step then takes the minimum of the current quadratic problem as a guess for the minimum of the original problem. However, the Newton step can be large, and leave the region of validity of the quadratic model. In such cases, the Newton step can be counterproductive for improving the optimality and/or feasibility of the iterate. We illustrate this problem for the unconstrained case in Figure 4.3

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-type optimization algorithms have so called *globalization strategies*. Most of these strategies can be interpreted as automatically generated homotopies.

In the ideal case, a globalization 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 *globalization strategies* have nothing to do with *global optimization*, which is concerned with finding global minima for non-convex problems.

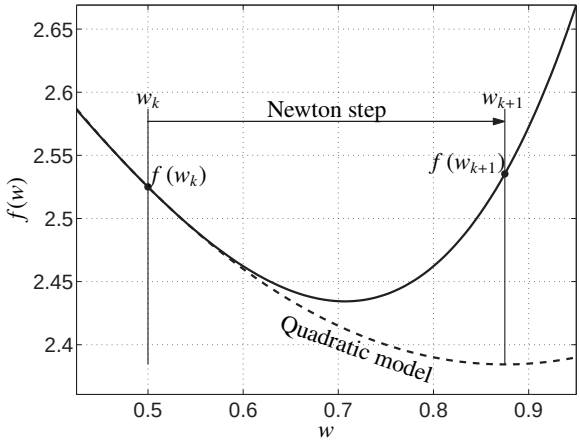


Figure 4.3 Illustration of the failure of the full Newton step. The Newton iteration is based on solving successive quadratic problems, which model locally the original optimization problem. If the Newton step provided by the quadratic model leaves its region of validity, it can provide a worse point w_{k+1} than the previous one, i.e., w_k. In this example, the Newton step going from w_k to w_{k+1} increases the cost function.

Here, we only touch the topic of globalization strategies very superficially, and for all details we refer to textbooks on nonlinear optimization and recommend in particular [55].

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

4.4.1 Measuring Progress: Merit Functions and Filters

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

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

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

with $f(w)$ the objective, $g(w)$ the residual vector of the equality constraints, and $h^+(w)$ the violations of the inequality constraints, i.e., $h_i^+(w) = \max\{0, h_i(w)\}$ 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 (w^*, λ^*, μ^*), i.e., if $\sigma > \max\{\|\lambda^*\|_\infty, \|\mu^*\|_\infty\}$, then the L1 penalty is exact in the sense that w^* also is a local minimum of $T_1(w)$. Thus, in a standard procedure we require that in each iteration a descent is achieved, i.e., $T_1(w_{k+1}) < T_1(w_k)$, 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 this are e.g. a "watchdog technique" that starting at some iterate w_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 $T_1(w_{k+M}) < T_1(w_k)$, so that the generosity was justified. If this is not the case, the algorithm jumps back to w_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.

4.4.2 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 $(w_k^{\mathrm{QP}}, \lambda_k^{\mathrm{QP}}, \mu_k^{\mathrm{QP}})$ is the solution of the QP at an SQP iterate w_k, it can be shown (if the QP multipliers are smaller than

σ) that the step vector or *search direction* $(w_k^{\mathrm{QP}} - w_k)$ 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 $w_{k+1} = w_k^{\mathrm{QP}}$, a shorter step

$$w_{k+1} = w_k + t(w_k^{\mathrm{QP}} - w_k)$$

with a damping factor or *step length* $t \in (0, 1]$. One popular way to ensure global convergence with help 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 globalization strategies is given in [55].

Exercises

4.1 Prove that a regularized Newton-type step $x_{k+1} = x_k - (B_k + \alpha I)^{-1}\nabla f(x_k)$ with B_k a Hessian approximation, α a positive scalar and I the identity matrix of suitable dimensions, converges to a small gradient step $x_{k+1} = x_k - \frac{1}{\alpha}\nabla f(x_k)$ as $\alpha \to \infty$.

4.2 Show that the Newton method is guaranteed to converge to a root (if it exists) of any monotonically increasing convex differentiable function

$F : \mathbb{R} \to \mathbb{R}$.

4.3 Let f be a twice continuously differentiable function satisfying $LI \geq \nabla^2 f(x) \geq mI$ for some $L > m > 0$ and let x^* be the unique minimizer of f over $\mathbb{R}^n$.

(a) Show that for any $x \in \mathbb{R}^n$:

$$f(x) - f(x^*) \geq \frac{m}{2}\|x - x^*\|^2.$$

(b) Let $\{x_k\}_{k\geq 0}$ be the sequence generated by the damped Newton's method with constant stepsize $t_k = \frac{m}{L}$. Show that:

$$f(x_k) - f(x_{k+1}) \geq \frac{m}{2L}\nabla f(x_k)^\top (\nabla^2 f(x_k))^{-1} \nabla f(x_k).$$

(c) Show that $x_k \to x^*$ as $k \to \infty$.

4.4 Prove the following theorem on the convergence of the exact Newton method.

If we apply the exact Newton method on the nonlinear set of equations $r(w) = 0$ and the following properties on the Jacobian hold:

- Boundedness: $\|\nabla r(w)\| \geq m, \ \forall w \in \mathbb{R}^n$,
- Lipschitz continuity: $\|\nabla r(w) - \nabla r(x)\| \leq L\|w - x\|, \quad \forall w, \ x \in \mathbb{R}^n$,

then the Newton iteration converges (locally) with the rate:

$$\|r(w + \Delta w)\| \leq \frac{L}{2m^2}\|r(w)\|^2.$$

Hint: use the integration by parts formula:

$$r(w + \Delta w) = r(w) + \int_0^1 \nabla r(w + t\Delta w)^\top \Delta w \cdot \mathrm{d}t.$$

4.5 The goal of this exercise is to Implement different Newton-type methods that minimize the nonlinear function:

$$f(x, y) = \frac{1}{2}(x - 1)^2 + \frac{1}{2}(10(y - x^2))^2 + \frac{1}{2}y^2. \tag{4.10}$$

(a) Derive, first on paper, the gradient and Hessian matrix of the function in (4.10). Then, re-write it in the form $f(x, y) = \frac{1}{2}\|R(x, y)\|_2^2$ where $R : \mathbb{R}^2 \to \mathbb{R}^3$ is the residual function. Derive the Gauss-Newton Hessian approximation and compare it with the exact one. When do the two matrices coincide?

(b) Implement your own Newton method with exact Hessian information and full steps. Start from the initial point $(x_0, y_0) = (-1, 1)$ and use as termination condition $\|\nabla f(x_k, y_k)\|_2 \leq 10^{-3}$. Keep track of the iterates (x_k, y_k) and use the provided function to plot the results.

(c) Update your code to use the Gauss-Newton Hessian approximation instead. Compare the performance of the two algorithms and plot the difference between exact and approximate Hessian as a function of the iterations.

(d) Now try to implement the BFGS formula for calculating the Hessian. Compare the results with the previous algorithms.

(e) Check how the steepest descent method performs on this example. Your Hessian now becomes simply αI where α is a positive scalar and I the identity matrix. Try $\alpha = 100, 200$ and 500. For which values does your algorithm converge? How does its performance compare with the previous methods?

(f) Imagine you remove the term $\frac{1}{2}y^2$ from $f(x, y)$ and compare the exact Newton's method with the Gauss-Newton. What do you expect?

4.6 Consider an NLP of the form:

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & \Phi(w) \\ \text{subject to} \quad & g(w) = 0. \end{aligned} \tag{4.11}$$

Prove that, under a condition on matrix H that you should specify, the primal Newton direction for (4.11) is provided by solving the QP:

$$\begin{aligned} \underset{\Delta w}{\text{minimize}} \quad & \frac{1}{2}\Delta w^\top H \Delta w + \nabla\Phi(w)^\top \Delta w \\ \text{subject to} \quad & g(w) + \nabla g(w)^\top \Delta w = 0 \end{aligned} \tag{4.12}$$

and the Lagrange multipliers of QP (4.12) provide the update for the Lagrange multipliers of (4.11).

4.7 Write a NLP solver for a problem of the type (4.11) using the Newton method.

- Have the option between using an exact Hessian and a Gauss-Newton Hessian approximation.
- Implement a line-search based on the Armijo condition.
- Use $\| \begin{bmatrix} \nabla \mathcal{L} & g \end{bmatrix} \|_1 \leq \text{tol}$ as an exit condition.

Hint: use the matlab symbolic toolbox to automatically compute your sensitivities ∇g, $\nabla \Phi$, and H, generate a function computing them using "matlabFunction". You will then be able to easily deploy your code to any NLP of the form (4.11)*, that will save you a lot of time in the following question. Test your code on a strictly convex Quadratic Program first, it should converge in one full Newton step.*

4.8 Try the following problem:

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & \frac{1}{2} w^\top w + \mathbf{1}^\top w \\ \text{subject to} \quad & w^\top w = 1 \end{aligned}$$

where $\mathbf{1}$ is a standing vector of ones of adequate dimension, and $w \in \mathbb{R}^n$. Prepare your solver for $n = 2$, and plots of:

- The trajectory of w_1, w_2 in a 2D plot, plot the unit circle representing the constraint.
- A semi-log plot of your exit criterion $\| \begin{bmatrix} \nabla \mathcal{L} & g \end{bmatrix} \|_1 \leq \text{tol}$ over the iterations.
- Your step size t over the iterations.

(a) Run the code using the parameters $\alpha = \beta = 0.5$ for the line-search parameters and $\nu = 1$ for the T_1 merit function. Use a tolerance of 10^{-8}. Use $\lambda = 0$ for the dual initial guess and try the following primal initial guesses:

$$w = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \text{and} \quad w = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \quad \text{and} \quad w = \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

Explain what you see.

(b) Try now the initial guess

$$w = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \lambda = 0.$$

What happens ? Explain.

(c) Same question for the initial guess

$$w = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \text{and} \quad \lambda = 0.$$

(d) Finally what happens with the initial guess

$$w = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix} \quad \text{and} \quad \lambda = 0.$$

Can you fix it ?

4.9 We now turn to the following NLP

$$\begin{aligned} \underset{w \in \mathbb{R}^3}{\text{minimize}} \quad & \frac{1}{2} w^\top w \\ \text{subject to} \quad & w_1^2 - 2w_2^3 - w_2 - 10w_3 = 0, \\ & w_2 + 10w_3 = 0. \end{aligned} \tag{4.13}$$

Deploy your NLP solver on problem (4.13). You can e.g. use the initial guess:

$$w = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \quad \text{and} \quad \lambda = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and a tolerance of 10^{-8}. What do you observe ? Explain.

4.10 Re-use your code to write an SQP solver for the general problems of the form:

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & \Phi(w) \\ \text{subject to} \quad & g(w) = 0, \\ & h(w) \leq 0. \end{aligned} \tag{4.14}$$

You can use the Matlab function *quadprog* as a QP solver. Make sure you include a check of the QP solver output (check for infeasibility, and non-convexity). Verify your code by setting up a QP problem in (4.14), you should observe a one-step convergence.

4.11 We will again use the aircraft model of Exercise 1.2 with the aircraft's flight noisy data of Exercise 3.11 to estimate a model for the flight trajectory. The data can be obtained in the book website as `flight_data.m`, and as before, it represents the position $\hat{p}_{x,k}$ and $\hat{p}_{z,k}$ during a 20 s flight. For this exercise, we will assume that the solution trajectory can be modeled by a fifth order polynomial as:

$$\bar{p}_{x,k}(\theta_x) = \theta_{x,1} + t_k\theta_{x,2} + \theta_{x,3}\sin\left(\theta_{x,4} + t_k\theta_{x,5}\right)e^{-\theta_{x,6}t_k}$$
$$\bar{p}_{z,k}(\theta_z) = \theta_{z,1} + t_k\theta_{z,2} + \theta_{z,3}\sin\left(\theta_{z,4} + t_k\theta_{z,5}\right)e^{-\theta_{z,6}t_k}$$

Then, in order to model the airplane flight trajectory, we can estimate the polynomial coefficients by solving the following optimization problem:

$$\underset{\theta_x,\,\theta_z}{\text{minimize}} \quad \sum_{k=0}^{N-1} \left(\bar{p}_{x,k}(\theta_x,\theta_z)) - \hat{p}_{x,k}\right)^2 + \left(\bar{p}_{z,k}(\theta_x,\theta_z) - \hat{p}_{z,k}\right)^2$$

(a) Write down the objective function in the form of Gauss-Newton:

$$\underset{\theta}{\text{minimize}} \quad \frac{1}{2}F(\theta)^\top F(\theta) \tag{4.15}$$

(b) Linearize $F(\theta)$ analytically to solve for F_0, J, where:

$$F(\theta) \approx F_0 + J\Delta\theta$$

(c) Use Newton's method with the Gauss-Newton Hessian approximation to solve (4.15).

(d) Plot p_x vs $-p_z$, $-p_z$ vs time, and p_x vs time. It will probably be very useful in debugging to plot each iteration of the algorithm.

(e) In Exercise 1.2, you used a RK4 integrator to minimize the measurement errors but for estimating α and the initial state. Now put that probem in the Gauss-Newton form. Only write a `MATLAB` function for F, not J. A function for computing J from F is provided in the book website as `finite_difference_jacob.m`. You will call this function with a command something like:

```
[F0,J] = finite_difference_jacob(@(x)Fobj(x),x0);
```

Solve this problem using Newton's method with the Gauss-Newton Hessian approximation. For initial guess, use any initial state you want, and use $\alpha = 3°$.

4.12 **CasADi Exercise: SQP Implementation** Regard the following optimization problem:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) := \frac{1}{2}(x_1 - 1)^2 + \frac{1}{2}(10(x_2 - x_1^2))^2 + \frac{1}{2}x_2^2 \\ \text{subject to} \quad & g(x) := x_1 + (1 - x_2)^2 = 0, \\ & h(x) := 0.2 + x_1^2 - x_2 \leq 0 \end{aligned} \tag{4.16}$$

(a) Re-write on paper the objective function in nonlinear least-square form $f(x) = \frac{1}{2}\|R(x)\|_2^2$ and derive the Gauss-Newton approximation of the Hessian of the Lagrangian.

(b) We will start by implementing an SQP solver for the unconstrained problem obtained by removing both g and h from (4.16). Using the template provided in the website, implement the CasADi functions `f` and `Jf` that return evaluations of f and its Jacobian. Use the numerical values given in the template to check that your implementation is correct. Do the same for the residual function R and its Jacobian.

(c) Using the Jacobian of f and R build the Gauss-Newton objective function

$$f_{gn} = \frac{1}{2}\Delta x^T \nabla R(x^k)\nabla R(x^k)^T \Delta x + \nabla_x f(x^k)^T \Delta x.$$

Then, allocate an instance of the QP solver qpOASES using CasADi and use it to solve the local quadratic approximations in the SQP iterations. Plot the results using the template. Where do the iterates converge to?

(d) Include now the equality constraints. Define two CasADi functions `G` and `Jg` that return evaluations of g and its Jacobian and use them to define the linearized equality constraint

$$g_l = g(x^k) + \nabla g_x(x^k)^T \Delta x.$$

Include this constraint in the QP formulation and run the simulation again. Does the solution change?

(e) Finally, include the inequality constraints. As in Task 5.4, define `H` and `Jh` and use them to define the linearized inequality constraints. Include them in the QP formulation and run the finalized version of the SQP solver.

5
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.

5.1 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 [38].

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 5.1 and illustrated in Example 5.2 below.

Algorithm 5.1 (User Function Evaluation via Elementary Operations)**.**
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 $\{x_1, \ldots, x_{n+i}\}$.

Example 5.2 (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

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

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

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

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

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

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:

$$J_F = C \cdot \tilde{J}_{m-1} \cdot \tilde{J}_{m-2} \cdots \tilde{J}_1 \cdot \tilde{J}_0. \tag{5.1}$$

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.

5.2 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:

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

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, $\dot{\tilde{x}}_0 = p$, and then the recursion

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

Given the special structure of the Jacobian matrices, most elements of $\dot{\tilde{x}}_i$ are only multiplied by one and nothing needs to be done, apart from the computation of the last component of the new vector $\dot{\tilde{x}}_{i+1}$. This last component is $\dot{x}_{n+i+1}$

Thus, in an efficient implementation, the forward AD algorithm works as the algorithm below. It first sets the seed $\dot{x} = p$ and then proceeds as follows.

Algorithm 5.3 (Forward Automatic Differentiation)**.**
Input: $\dot{x}_1, \ldots, \dot{x}_n$ and all partial derivatives $\frac{\partial \phi_i}{\partial x_j}$
Output: $\dot{x}_1, \ldots, \dot{x}_{n+m}$

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

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

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

Example 5.4 (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 14.15 is smaller than two times the cost of Algorithm 5.1, 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 14.15 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.

5.2.1 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

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

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 dot-prime `.'` or `transpose()`.

5.3 The Backward Mode of AD

In backward AD we evaluate the product in Eq. (5.1) 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 $\lambda^\top J_F$. It does so in the following way:

$$\lambda^\top J_F = ((((\lambda^\top C) \cdot \tilde{J}_{m-1}) \cdot \tilde{J}_{m-2}) \cdots \tilde{J}_1) \cdot \tilde{J}_0. \tag{5.2}$$

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

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

Each intermediate variable has a bar variable and at the start, we initialize all bar variables with the value that we obtain from $C^\top \lambda$. 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 (5.2) in order to obtain

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

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

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

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 $\tilde{J}_i^\top \bar{\tilde{x}}_{i+1}$ with the transposed Jacobian

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

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

Algorithm 5.5 (Reverse Automatic Differentiation)**.**
Input: seed vector $\bar{x}_1, \ldots, \bar{x}_{n+m}$ and all partial derivatives $\frac{\partial \phi_i}{\partial x_j}$
Output: $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$

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

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

Example 5.6 (Reverse Automatic Differentiation)**.** We regard the same example as before, and want to compute the gradient $\nabla f(x) = (\bar{x}_1, \bar{x}_2, \bar{x}_3)^\top$ given (x_1, x_2, x_3). We set $\lambda = 1$. Because the selection matrix C selects only the last intermediate variable as output, i.e. $C = (0, \cdots 0, 1)$, we initialize the seed vector with zeros apart from the last component, which is one. In the reverse mode, the algorithm first has to evaluate the function with all intermediate quantities, and only then it can compute the bar quantities, which it does in reverse order. At the end it obtains, among other, the desired quantitities $(\bar{x}_1, \bar{x}_2, \bar{x}_3)$. The full algorithm is the following.

// *** forward evaluation of the function ***

$x_4 = x_1 x_2$

$x_5 = \sin(x_4)$

$x_6 = x_4 x_3$

$x_7 = \exp(x_6)$

$x_8 = x_5 + x_7$

// *** initialization of the seed vector ***

$\bar{x}_i = 0, \quad i = 1, \ldots, 7$

$\bar{x}_8 = 1$

// *** backwards sweep ***

// * differentiation of $x_8 = x_5 + x_7$

$\bar{x}_5 = \bar{x}_5 + 1\ \bar{x}_8$

$\bar{x}_7 = \bar{x}_7 + 1\ \bar{x}_8$

// * differentiation of $x_7 = \exp(x_6)$

$\bar{x}_6 = \bar{x}_6 + \exp(x_6)\bar{x}_7$

// * differentiation of $x_6 = x_4 x_3$

$\bar{x}_4 = \bar{x}_4 + x_3 \bar{x}_6$

$\bar{x}_3 = \bar{x}_3 + x_4 \bar{x}_6$

// * differentiation of $x_5 = \sin(x_4)$

$\bar{x}_4 = \bar{x}_4 + \cos(x_4)\bar{x}_5$

// differentiation of $x_4 = x_1 x_2$

$\bar{x}_1 = \bar{x}_1 + x_2 \bar{x}_4$

$\bar{x}_2 = \bar{x}_2 + x_1 \bar{x}_4$

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

It can be shown that the cost of Algorithm 5.5 is less than 3 times the cost of Algorithm 5.1, i.e.,

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

If we want to obtain the full Jacobian of F, we need to call Algorithm 5.5 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.

5.3.1 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:

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

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 $\nabla f(x)$.

5.4 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.

Exercises

5.1 Assume we have a twice continuously differentiable function $f : \mathbb{R} \to \mathbb{R}$ and we want to evaluate its derivative $f'(x_0)$ at x_0 with finite differences. Further assume that in a neighborhood $\mathcal{N}(x_0)$ it holds:

$$|f''(x)| \leq f''_{\max}, \quad |f(x)| \leq f_{\max} \tag{5.3}$$

with $\mathcal{N}(x_0) := \{x \,|x_0 - \delta \leq x \leq x_0 + \delta\}$, $\delta > t$ and t the perturbation in the finite difference approximation. The function $f(x)$ can be represented on a computing system with an accuracy ϵ_{mach}, i.e., it is perturbed by noise $\epsilon(x)$:

$$\tilde{f}(x) = f(x)(1 + \epsilon(x)) \quad |\epsilon(x)| \leq \epsilon_{\text{mach}}.$$

(a) Compute a bound ψ on the error of the finite difference approximation

of $f'(x_0)$

$$\left| \frac{\tilde{f}(x_0 + t) - \tilde{f}(x_0)}{t} - f'(x_0) \right| \leq \psi(t, f_{\max}, f''_{\max}, \epsilon_{\text{mach}}).$$

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

(c) Do a similar analysis for the central differences where $\tilde{f}'(x_0) = \frac{\tilde{f}(x_0+t)-\tilde{f}(x_0-t)}{2t}$. *Hint: you can assume that also the third derivative is bounded in* $[x_0 - t, x_0 + t]$.

5.2 Consider a two-dimensional model of an airplane with states $x = [p_x, p_z, v_x, v_z]$ where position $\vec{p} = [p_x, p_z]$ and velocity $\vec{v} = [v_x, v_z]$ are vectors in the $x-z$ directions. We will use the standard aerospace convention that $\hat{x}$ is forward and $\hat{z}$ is down, so altitude is $-p_z$. The system has one control $u = [\alpha]$, where α is the aerodynamic angle of attack in radians. A Matlab function has been provided for you which integrates the system in time, implementing:

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

where the continuous time system dynamics have the form:

$$f(x, u) = \begin{pmatrix} v_x \\ v_z \\ F_x/m \\ F_z/m \end{pmatrix}$$

with

$$\vec{F} = \vec{F}_{\text{lift}} + \vec{F}_{\text{drag}} + \vec{F}_{\text{gravity}}.$$

As well as outputting x_{k+1}, this function also provides $\frac{\partial x_{k+1}}{\partial x_k}$ and $\frac{\partial x_{k+1}}{\partial u_k}$. This function is available as `integrate_airplane_ode.m` at the book webpage. In this exercise we want to find controls for the airplane so that it gets a maximum velocity in upwards direction at the end of the horizon (after 2 seconds) using $h = 0.02$ and a horizon length of $N = 100$. In particular, we will optimize the following NLP:

$$\begin{aligned} &\underset{U \in \mathbb{R}^{100}}{\text{minimize}} \quad \phi(U) = v_{z,N}(U) \\ &\text{subject to} \quad -1^\circ \leq U_k \leq 10^\circ, \quad k = 0 \ldots N-1 \end{aligned}$$

A matlab function `function [phi, grad_phi, X] = phi_obj(U)` has been provided at the webpage in the file `phi_obj.m`. This function

computes $v_{z,N}$ (`phi`) and a time history of states (`X`). It also returns $\frac{\partial v_{z,N}}{\partial U}$ (`grad_phi`), but this part is incomplete - you will implement it yourself.

(a) Use `phi_obj.m` to solve the NLP using `fmincon` letting Matlab estimate derivatives. Your `fmincon` call should look like:

```
opts = optimset('display','iter','algorithm',...
             'interior-point','MaxFunEvals',100000);
alphasOpt = fmincon(@phi_obj, alphas0, [], [],...
                            [], [], lb, ub, [], opts);
```

Use $\alpha_k = 0, \;\; k = 0 \ldots N-1$ as an initial guess. Plot p_x vs $-p_z$ and α vs time. How much time and iterations does the solver need to converge?

(b) Using reverse mode AD, complete the missing part of `phi_obj.m` to compute `grad_phi`.

(c) Solve the NLP with `phi_obj.m` and `fmincon` again, this time using exact derivatives. Your `fmincon` call should look like:

```
opts = optimset('GradObj','on','display','iter',...
                       'algorithm','interior-point');
alphasOpt = fmincon(@phi_obj, alphas0, [], [],...
                            [], [], lb, ub, [], opts);
```

Use $\alpha_k = 0, \;\; k = 0 \ldots N-1$ as an initial guess. Plot p_x vs $-p_z$ and α vs time. How much time and iterations does the solver need to converge?

7
Discrete Optimal Control

A lot of times it's up to our discretion.
— Joe Jimenez

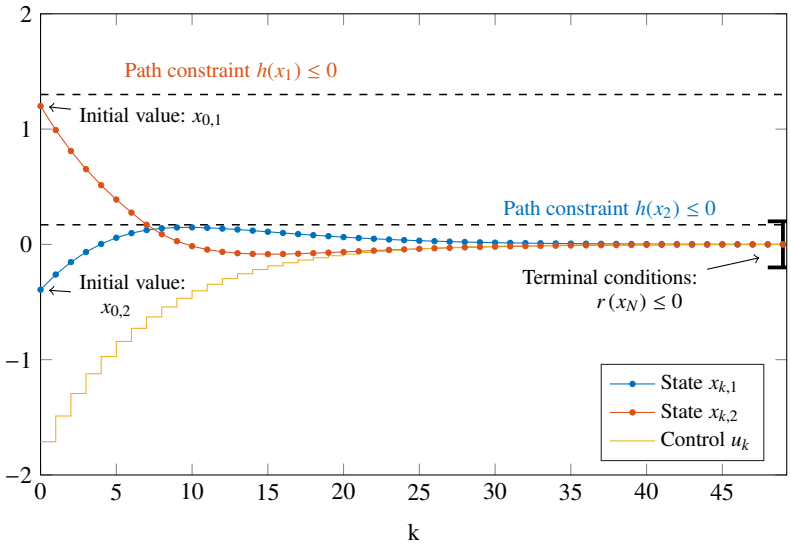


Figure 7.1 Variables of a discrete optimal control problem with $N = 49$

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

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

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

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

$$r(x_0, x_N) = 0.$$

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

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

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

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

Finally, periodic boundary conditions can be imposed by setting

$$r(x_0, x_N) = x_0 - x_N.$$

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

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

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

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

7.1 Optimal Control Problem (OCP) Formulations

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

7.1.1 Original Problem Formulation

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

$$\underset{\substack{x_0,u_0,x_1,\ldots,\\ u_{N-1},x_N}}{\text{minimize}} \quad \sum_{k=0}^{N-1} L(x_k,u_k) + E(x_N) \tag{7.1a}$$

$$\text{subject to} \quad x_{k+1} - f(x_k,u_k) = 0, \quad k = 0,\ldots,N-1, \tag{7.1b}$$

$$h(x_k,u_k) \leq 0, \quad k = 0,\ldots,N-1, \tag{7.1c}$$

$$r(x_0,x_N) = 0. \tag{7.1d}$$

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

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

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

7.1.2 The Simultaneous Approach

The nonlinear program (7.1) is large and structured and can thus in principle be solved by any NLP solver. This is called the *simultaneous approach* to optimal control and requires the use of a structure exploiting NLP solver in order to be efficient. Note that in this approach, all original variables, i.e., u_k and x_k, remain optimization variables of the NLP. Its name stems from the fact that the NLP solver has to simultaneously solve both, the simulation and the optimization problem. It is interesting to remark that the model equations (7.1b) will for most NLP solvers only be satisfied once the NLP iterations are converged. The simultaneous approach is therefore sometimes referred to as an *infeasible path* approach. The methods *direct multiple shooting* and *direct collocation* that we explain in the third part of this script are simultaneous approaches.

7.1.3 The Reduced Formulation and the Sequential Approach

On the other hand, we know that we could eliminate nearly all states by a forward simulation, and in this way we could reduce the variable space of the

NLP. The idea is to keep only x_0 and $U = [u_0^\top, \ldots, u_{N-1}^\top]^\top$ as variables. The states $x_1, \ldots, x_N$ are eleminated recursively by

$$\begin{aligned} \bar{x}_0(x_0, U) &= x_0 \\ \bar{x}_{k+1}(x_0, U) &= f(\bar{x}_k(x_0, U), u_k), \quad k = 0, \ldots, N-1. \end{aligned} \tag{7.2}$$

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

$$\begin{aligned} \underset{x_0, U}{\text{minimize}} \quad & \sum_{k=0}^{N-1} L(\bar{x}_k(x_0, U), u_k) + E(\bar{x}_k(x_0, U)) && \text{(7.3a)} \\ \text{subject to} \quad & r(x_0, \bar{x}_N(x_0, U)) = 0, && \text{(7.3b)} \\ & h(\bar{x}_k(x_0, U), u_k) \leq 0, \quad k = 0, \ldots, N-1. && \text{(7.3c)} \end{aligned}$$

Note that the model Equation (7.2) is implicitly satisfied by definition, but is not anymore a constraint of the optimization problem. This reduced problem can now be addressed again by Newton-type methods, but the exploitation of sparsity in the problem is less important. This is called the *sequential* approach, because the simulation problem and optimization problem are solved sequentially, one after the other. Note that the user can observe during all iterations of the optimization procedure what is the resulting state trajectory for the current iterate, as the model equations are satisfied by definition.

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

7.2 Analysis of a Simplified Optimal Control Problem

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

$$\begin{aligned} \underset{\substack{x_0, u_0, x_1, \ldots, \\ u_{N-1}, x_N}}{\text{minimize}} \quad & \sum_{k=0}^{N-1} L(x_k, u_k) + E(x_N) && \text{(7.4a)} \\ \text{subject to} \quad & x_{k+1} - f(x_k, u_k) = 0, \quad k = 0, \ldots, N-1, && \text{(7.4b)} \\ & r(x_0, x_N) = 0. && \text{(7.4c)} \end{aligned}$$

7.2.1 KKT Conditions of the Simplified Problem

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

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

Here, the objective $F(w)$ is just copied from (7.4a) while $G(w)$ collects all constraints:

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

The Lagrangian function has the form

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

and the summarized KKT-conditions of the problem are

$$\nabla_w \mathcal{L}(w, \lambda) = 0 \tag{7.5a}$$

$$G(w) = 0. \tag{7.5b}$$

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

$$\nabla_{x_0} \mathcal{L}(w, \lambda) = \nabla_{x_0} L(x_0, u_0) + \frac{\partial f}{\partial x_0}(x_0, u_0)^\top \lambda_1 + \frac{\partial r}{\partial x_0}(x_0, x_N)^\top \lambda_r = 0. \tag{7.6a}$$

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

$$\nabla_{x_k} \mathcal{L}(w, \lambda) = \nabla_{x_k} L(x_k, u_k) - \lambda_k + \frac{\partial f}{\partial x_k}(x_k, u_k)^\top \lambda_{k+1} = 0. \tag{7.6b}$$

Last, the special case $k = N$

$$\nabla_{x_N} \mathcal{L}(w, \lambda) = \nabla_{x_N} E(x_N) - \lambda_N + \frac{\partial r}{\partial x_N}(x_0, x_N)^\top \lambda_r = 0. \tag{7.6c}$$

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

$$\nabla_{u_k} \mathcal{L}(w, \lambda) = \nabla_{u_k} L(x_k, u_k) + \frac{\partial f}{\partial u_k}(x_k, u_k)^\top \lambda_{k+1} = 0. \tag{7.6d}$$

Until now, we have computed in detail the components of the first part of the KKT-condition (7.5a), i.e., $\nabla_w \mathcal{L}(w, \lambda) = 0$. The other part of the KKT-condition, $G(w) = 0$, is trivially given by

$$f(x_k, u_k) - x_{k+1} = 0, \quad k = 0, \ldots, N-1 \tag{7.6e}$$

$$r(x_0, x_N) = 0. \tag{7.6f}$$

Thus, collecting all equations (7.6a) to (7.6f), we have stated the KKT-conditions of the OCP. They can be treated by Newton-type methods in different ways. The *simultaneous approach* addresses equations (7.6a) to (7.6f) directly by a Newton-type method in the space of all variables (w, λ). In contrast to this, the *sequential approach* approach eliminates all the states $x_1, \ldots, x_N$ in (7.6e) by a forward simulation, and if it is implemented efficiently, it also uses Eqs. (7.6c) and (7.6b) to eliminate all multipliers $\lambda_N, \ldots, \lambda_1$ in a backward simulation, as discussed in the following subsection.

7.2.2 Computing Gradients in the Sequential Approach

A naive implementation of the sequential approach would start by coding routines that evaluate the objective and constraint functions, and then passing these routines as black-box codes to a generic NLP solver, like `fmincon` in MATLAB. But this would not be the most efficient way to implement the sequential approach. The reason is the generation of derivatives, which a generic NLP solver will compute by finite differences. On the other hand, many generic NLP solvers allow the user to deliver explicit functions for the derivatives as well. This allows us to compute the derivatives of the reduced problem functions more efficiently. The key technology here is algorithmic differentiation in the backward mode, as explained in Chapter 5.

To see how this relates to the optimality conditions (7.6a) to (7.6f) of the optimal control problem, let us simplify the setting even more by assuming a fixed initial value and no constraint on the terminal state, i.e., $r(x_0, x_N) = \bar{x}_0 - x_0$.

In this case, the KKT conditions simplify to the following set of equations, which we bring already into a specific order:

$$x_0 = \bar{x}_0 \tag{7.7a}$$
$$x_{k+1} = f(x_k, u_k), \qquad k = 0, \dots, N-1, \tag{7.7b}$$
$$\lambda_N = \nabla_{x_N} E(x_N) \tag{7.7c}$$
$$\lambda_k = \nabla_{x_k} L(x_k, u_k) + \frac{\partial f}{\partial x_k}(x_k, u_k)^\top \lambda_{k+1}, \qquad k = N-1, \dots, 1, \tag{7.7d}$$
$$0 = \nabla_{u_k} L(x_k, u_k) + \frac{\partial f}{\partial u_k}(x_k, u_k)^\top \lambda_{k+1}, \qquad k = 0, \dots, N-1. \tag{7.7e}$$

It can easily be seen that the first four equations can trivially be satisfied, by a forward sweep to obtain all x_k and a backward sweep to obtain all λ_k. Thus, x_k and λ_k can be made explicit functions of $u_0, \dots, u_{N-1}$. The only equation that is non-trivial to satisfy is the last one, the partial derivatives of the Lagrangian w.r.t. the controls $u_0, \dots, u_{N-1}$. Thus we could decide to eliminate x_k and λ_k and only search with a Newton-type scheme for the variables $U = (u_0, \dots, u_{N-1})$ such that these last equations are satisfied. It turns out that the left hand side residuals (7.7e) are nothing else than the derivative of the reduced problem's objective (7.3a), and the forward-backward sweep algorithm described above is nothing else than the reverse mode of algorithmic differentiation. It is much more efficient than the computation of the gradient by finite differences.

The forward-backward sweep is well known in the optimal control literature and often introduced without reference to the reverse mode of AD. On the other hand, it is good to know the general principles of AD in forward or backward mode, because AD can also be beneficial in other contexts, e.g. for the evaluation of derivatives of the other problem functions in (7.3a)-(7.3b). Also, when second order derivatives are needed, AD can be used and more structure can be exploited, but this is most easily derived in the context of the simultaneous approach, which we do in the following section.

7.3 Sparsity Structure of the Optimal Control Problem

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

$$\underset{\substack{x_0,u_0,x_1,\ldots,\\ u_{N-1},x_N}}{\text{minimize}} \quad \sum_{k=0}^{N-1} L_k(x_k,u_k) + E(x_N) \tag{7.8a}$$

$$\text{subject to} \qquad f_k(x_k,u_k) - x_{k+1} = 0, \quad k = 0,\ldots,N-1, \tag{7.8b}$$

$$\sum_{k=0}^{N-1} r_k(x_k,u_k) + r_N(x_N) = 0, \tag{7.8c}$$

$$h_k(x_k,u_k) \le 0, \quad k = 0,\ldots,N-1, \tag{7.8d}$$

$$h_N(x_N) \le 0. \tag{7.8e}$$

Compared to the OCP (7.1) in the previous sections, we now allow indices on all problem functions making the system time dependent; also, we added terminal inequality constraints (7.8e), and as boundary conditions we now allow now very general coupled multipoint constraints (7.8c) that include the cases of fixed initial or terminal values or periodicity, but are much more general. Note that in these boundary constraints, terms arising from different time points are only coupled by addition, because this allows us to maintain the sparsity structure we want to exploit in this section.

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

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

Its Lagrangian function is given by

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

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

7.3.1 Partial Separability of the Lagrangian

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

Hessian of the Lagrangian, is a very sparse matrix. This is due to the fact that the Lagrangian is a *partially separable* function [37].

Definition 7.1 (Partial Separability)**.** A function $f : \mathbb{R}^n \to \mathbb{R}$ is called partially separable if it can be decomposed as a sum of m functions $f_j : \mathbb{R}^{n_j} \to \mathbb{R}$ with $n_j < n$ for all $j = 1, \ldots, m$. This means that for each j exists a subset I_j of indices from $\{1, \ldots, n\}$ and subvectors x_{I_j} of x such that

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

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

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

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

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

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

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

while the last subfunction is given as

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

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

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

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

This block diagonality of the Hessian leads to several very favourable facts, namely that (i) the Hessian can be approximated by *high-rank* or *block updates* within a BFGS method [37, 17], and (ii) that the QP subproblem in all Newton-type methods has the same decomposable objective function as the original optimal control problem itself.

7.3.2 The Sparse QP Subproblem

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

$$\begin{aligned} \underset{\Delta w}{\text{minimize}} \quad & \nabla F(w)^\top \Delta w + \frac{1}{2} \Delta w^\top \nabla_w^2 \mathcal{L}(w, \lambda, \mu) \Delta w \\ \text{subject to} \quad & G(w) + \nabla G(w)^\top \Delta w = 0, \\ & H(w) + \nabla H(w)^\top \Delta w \leq 0. \end{aligned}$$

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

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

second, the objective gradients

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

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

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

and fourth the transition matrices

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

fifth the residual of the coupled constraints

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

as well as its derivatives

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

and last the inequality constraint residuals and their derivatives

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

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

$$\underset{\substack{\Delta x_0, \Delta u_0, \dots, \\ \Delta x_N}}{\text{minimize}} \quad \frac{1}{2}\sum_{k=0}^{N-1}\begin{bmatrix}\Delta x_k \\ \Delta u_k\end{bmatrix}^\top Q_k \begin{bmatrix}\Delta x_k \\ \Delta u_k\end{bmatrix} + \frac{1}{2}\Delta x_N^\top Q_N \Delta x_N + \sum_{k=0}^{N}\begin{bmatrix}\Delta x_k \\ \Delta u_k\end{bmatrix}^\top g_k + \Delta x_N^\top g_N \tag{7.9a}$$

$$\text{subject to} \quad a_k + A_k \Delta x_k + B_k \Delta u_k - \Delta x_{k+1} = 0, \quad k = 0, \dots, N-1, \tag{7.9b}$$

$$r + \sum_{k=0}^{N-1} R_k \begin{bmatrix}\Delta x_k \\ \Delta u_k\end{bmatrix} + R_N \Delta x_N = 0, \tag{7.9c}$$

$$h_k + H_k \begin{bmatrix}\Delta x_k \\ \Delta u_k\end{bmatrix} \le 0, \quad k = 0, \dots, N-1, \tag{7.9d}$$

$$h_N + H_N \Delta x_N \le 0. \tag{7.9e}$$

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

7.3.3 Sparsity Exploitation in QP Solvers

When regarding the QP (7.9) one way would be to apply a sparse interior point QP solver like OOQP to it, or a sparse active set method. This can be very efficient. Another way would be to first reduce, or *condense*, the variable space of the QP, and then apply a standard dense QP solver to the reduced problem. Let us treat this way first.

Condensing

When we regard the linearized dynamic system equations (7.9b) they correspond to an affine time variant system in the steps Δx_k, namely

$$\Delta x_{k+1} = a_k + A_k \Delta x_k + B_k \Delta u_k. \tag{7.10}$$

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

$$\begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_N \end{bmatrix} = v + M \begin{bmatrix} \Delta x_0 \\ \Delta u_0 \\ \vdots \\ \Delta u_{N-1} \end{bmatrix},$$

$$\Leftrightarrow$$

$$\Delta w_{\text{dep}} = v + M \Delta w_{\text{ind}}$$

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

The vector v can be generated recursively by simulating the affine dynamic system (7.10) with all inputs set to zero, i.e., $\Delta w_{\text{ind}} = 0$. This yields the forward recursion

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

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

To get a notationally different, but equivalent view on condensing, note that the linear dynamic system equations (7.9b) are nothing else than the linear system

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

After reordering the variables into dependent and independent ones, this sys-

tem can be written as

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

which we can summarize as

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

so that we get the explicit solution

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

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

Once the vector v and matrix M are computed, we can formulate a *condensed QP* which has only the independent variables Δw_{ind} as degrees of freedom. This condensed QP can be solved by a dense QP solver, and the resulting solution Δw^*_{ind} can be expanded again to yield also the QP solution for $w^*_{\text{dep}} = v + M\Delta w^*_{\text{ind}}$. The QP multipliers $\lambda_{\text{dep}} = (\lambda_1, \ldots, \lambda_N)$ for the constraints (7.9b) can be obtained from the dense QP solution in a slightly more complex way. The trick is to regard the Lagrangian of the original QP (7.9b), $\mathcal{L}^{\text{QP}}(\Delta w_{\text{ind}}, \Delta w_{\text{dep}}, \lambda_{\text{dep}}, \lambda_r, \mu)$ and note that the condensed QP yields also the multipliers λ^*_r, μ^*, which turn out to be the correct multipliers also for the uncondensed QP. Thus, the only missing quantity is λ^*_{dep}. It can be obtained by using the follwing two observations: first, for the true QP solution must hold that the Lagrange gradient is zero, also with respect to Δw_{dep}. Second, this Lagrange gradient depends linearly on the unknown multipliers λ_{dep} which contribute to it via the term $Y^\top \lambda_{\text{dep}}$, i.e. we have

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

It is a favourable fact that the Lagrange gradient depends on the missing multipliers via the matrix $Y^\top$, because this matrix is invertible. Thus, we obtain an

explicit equation for obtaining the missing multipliers, namely

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

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

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

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

Sparse KKT System

A different way to exploit the sparsity present in the QP (7.9) is to keep all variables in the problem and use within the QP solver linear algebra routines that exploit sparsity of matrices. This can be realized within both, interior point (IP) methods as well as in active set methods, but is much easier to illustrate at the example of IP methods. For illustration, let us assume a problem without coupled constraints (7.9c) and assume that all inequalities have been transformed into primal barrier terms that are added to the objective. Then, in each interior point iteration, an equality constrained QP of the following simple form needs to be solved.

$$\underset{\substack{\Delta x_0, \Delta u_0, \ldots, \\ \Delta x_N}}{\text{minimize}} \quad \frac{1}{2}\sum_{k=0}^{N-1}\begin{bmatrix}\Delta x_k \\ \Delta u_k\end{bmatrix}^\top \begin{bmatrix} Q_k^x & Q_k^{xu} \\ (Q_k^{xu})^\top & Q_k^u \end{bmatrix}\begin{bmatrix}\Delta x_k \\ \Delta u_k\end{bmatrix} + \frac{1}{2}\Delta x_N^\top Q_N \Delta x_N \\ + \sum_{k=0}^{N}\begin{bmatrix}\Delta x_N \\ \Delta u_N\end{bmatrix}^\top g_k + \Delta x_N^\top g_N \tag{7.11}$$

$$\text{subject to} \quad a_k + A_k\Delta x_k + B_k\Delta u_k - \Delta x_{k+1} = 0, \quad k = 0, \ldots, N-1.$$

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

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

This linear system can be solved with a banded direct factorization routine, whose runtime is proportional to $N(2n_x + n_u)^3$. We will see in the next chapter that a particularly efficient way to solve the above linear system can be obtained by applying the principle of dynamic programming to the equality constrained quadratic subproblem (7.11).

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

Exercises

7.1 Consider a simple pendulum defined by the system state $x = [\phi, \omega]^\top$ where ϕ is the angle and ω the angular velocity of the pendulum, and

where $\phi = 0$, represents the pendulum in its inverse position, e.g. the mass is at the top. The system dynamics are given by:

$$\begin{aligned}\dot{\phi} &= \omega \\ \dot{\omega} &= 2\sin(\phi) + u\end{aligned}$$

In this problem, we will solve a Discrete Optimal Control Problem formulated as an Non-Linear Program by discretization of the dynamics. In particular, we will use the Matlab fmincon function to solve the following NLP:

$$\begin{aligned}\underset{\substack{x_0,\ldots,x_N,\\ u_0,\ldots,u_{N-1}}}{\text{minimize}} \quad & \sum_{k=0}^{N-1}(\phi_k^2 + u_k^2) \\ \text{subject to} \quad & \bar{x}_0 - x_0 = 0, \\ & f(x_k, u_k) - x_{k+1} = 0, \qquad k = 0, \ldots, N-1, \\ & \omega_{\min} \le \omega_k \le \omega_{\max}, \quad k = 0, \ldots, N, \\ & u_{\min} \le u_k \le u_{\max}, \quad k = 0, \ldots, N,\end{aligned}$$

where the discrete time system dynamics are obtain by a Runge-Kutta integrator of order 4 with a timestep of $h = 0.2$. The horizon of the optimal control problem is $N = 60$, the given initial state is $\bar{x}_0 = [-\pi, 0]$, and the bounds are given by $\omega_{\min} = -\pi$, $\omega_{\max} = \pi$, $u_{\min} = -1.1$, $u_{\max} = 1.1$.

In order to pass the optimal control problem to the solver we first have to formulate it as an NLP. The variables of the optimal control problem are summarized in a vector $y = (x_0, u_0, \ldots, u_{N-1}, x_N)^\top$. Then the NLP has the following form:

$$\begin{aligned}\underset{y}{\text{minimize}} \quad & \psi(y) \\ \text{subject to} \quad & G(y) = 0, \\ & y_{\min} \le y \le y_{\max}.\end{aligned} \tag{7.12}$$

(a) Write down the objective function $\psi(y)$ and the constraints on paper. Use the same order for the constraints as in the optimal control problem.

$$G(y) = \begin{bmatrix} \\ \vdots \\ \\ \end{bmatrix} \quad y_{\max} = \begin{bmatrix} \\ \\ \end{bmatrix} \quad y_{\min} = \begin{bmatrix} \\ \\ \end{bmatrix}$$

Implement the objective and the equality constraints as Matlab functions

(b) Check if your function $G(y)$ does what you want by writing a forward simulation function `[y]=simulate(x0,U)` that simulates, for a given initial value x_0 and control profile $U = (u_0, \ldots, u_{N-1})$, the whole trajectory $x_1, \ldots, x_N$ and constructs from this the full vector $y = (x_0, u_0, x_1, \ldots, x_N)$. If you generate for any x_0 and U a vector y and then you call your function $G(y)$ with this input, almost all of your residuals should be zero. Which ones are not zero?
As a test, simulate e.g. with $x_0 = [0, 0.5]^\top$ and $u_k = 1, k = 0, \ldots, N-1$ in order to generate y, and then call $G(y)$, to test that your function G is correct. Specify the norm of the residuals $G(y)$.

(c) Use `fmincon` to solve the NLP:

```
options=optimoptions(@fmincon, 'display',
                 'final/iter','MaxFunEvals',100000);
y=fmincon(@objective,y0,[],[],[],[],lby,uby,
                           @nonlconstraints,options);
```

As an initialization for y_0 you can use $\bar{x}_0$ for all state variables and zero for all control variables.
How many iteration does the solver need to converge (use display option `iter`)? How long does the call to the minimizer take (use `tic/toc` and the display option `final`)? Plot the evolution of the state and the applied controls in time. Make an animation to see if the pendulum swings up.
Hint: You can call the following function several times to create the animation:

```
function plot_pendulum(x)
   phi = x(1);
   plot([0;sin(phi)], [0;cos(phi)], '-o')
   xlim([-1,1])
   ylim([-1,1])
```

```
end
```

(d) Do a RK4 simulation of the pendulum and apply the optimal controls of part (c) open-loop. Does the pendulum swing up? Does the resulting state trajectory differ from the output of the solver? Why?

(e) Play with other options of the solvers like the type of finite differences for computing the Jacobian, stopping criteria, and tolerances for violating the constraints. How do they influences computation time, number of iterations, and precision of the solution.

7.2 In this exercise, we will again use the same pendulum to solve the NLP given by (7.12). This time however, we will solve (7.12) by a self written Sequential Quadratic Programming (SQP) solver with Gauss-Newton Hessian.

In particular, we will first prepare the calculation of the Jacobian that is needed in the SQP iterations and we will test the correctness of the Jacobian by passing it to the `fmincon` solver and find a faster way to compute the Jacobian. Then, we will implement the SQP solver by solving the Quadratic Programs (QP) in each iteration with then Matlab `quadprog` function.

Hint: It is recommended to start the problem by re-using the code from the previous task.

(a) The Jacobian of the non-linear equality constraints $J_G(y) = \frac{\partial G}{\partial y}(y)$ can be passed directly to the `fmincon` function by including it as an output of the constrains function (see Matlab constraints documentation), and by activating the correct `fmincon` option:

```
options=optimoptions(@fmincon,\dots,'GradConstr',
                                          'on',\dots);
```

Calculate the Jacobian $J_G(y)$ by finite differences, perturbing all 182 directions one after the other using $\delta = 10^{-4}$. This needs in total 183 calls of G. Give your routine e.g. the name `[jac,Gy]=GJacSlow(y)`. Compute J_G for w_0 and look at the structure of this matrix by making a plot using the command `spy(J)`. Use this Jacobian to solve the OCP. How many iterations and how much time does the solver need to converge?

(b) By looking at the structure of J_G, we see that the matrix is very sparse can be calculated much more efficiently. The Jacobian $J_G(y) = \frac{\partial G}{\partial y}(y)$ is block sparse with as blocks either (negative) unit matrices or the partial derivatives $A_k = \frac{\partial f}{\partial x}(x_k, u_k)$ and $B_k = \frac{\partial f}{\partial u}(x_k, u_k)$. Fill in the

corresponding blocks in the following matrix

$$J_G(y) = \begin{bmatrix} & \ddots & \ddots & \ddots & \\ & & & & \\ & & & & \end{bmatrix}$$

(c) With this knowledge you can construct the Jacobian in a computationally much more efficient way, as follows:

- First write a function `[A,B]=RK4stepJac(x,u)` using finite differences with a step size of $\delta = 10^{-4}$. Here, $A = \frac{\partial f}{\partial x}(x,u)$ and $B = \frac{\partial f}{\partial u}(x,u)$.
- Using this function `[A,B]=RK4stepJac(x,u)`, implement a function `[jac,Gy]=GJacFast(y)`.
- Compare if the result is correct by taking the difference of the Jacobians you obtain by `[jac,Gy]=GJacFast(y)` and `[jac,Gy]=GJacSlow(y)`.

  Pass this Jacobian to your constraints function. How many iterations and how much time does the solver need now?

(d) The SQP with Gauss-Newton Hessian (also called *constrained Gauss-Newton method*) solves a linearized version of this problem in each iteration. More specific, if the current iterate is $\bar{y}$, the next iterate is the solution of the following Quadratic Program (QP):

$$\begin{aligned} \underset{y}{\text{minimize}} \quad & y^\top H y \\ \text{subject to} \quad & G(\bar{y}) + J_G(\bar{y})(y - \bar{y}) = 0, \\ & y_{\min} \leq y \leq y_{\max}. \end{aligned} \tag{7.13}$$

Define what H_x and H_u need to be in the Hessian

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

(e) Write a function `[ybar_next]=GNStep(ybar)` that performs one SQP-Gauss-Newton step by first calling `[jac,Gy] = GJacFast(y)` and then solving the resulting QP (7.13) using the MATLAB Qp

solver `quadprog`. Note that the QP is very sparse but that this sparsity is not exploited in full during the call of `quadprog`.

(f) Write a loop around your function `GNStep`, initialize the GN procedure at at y_0, and stop the iterations when $\|y_{k+1} - y_k\|$ gets smaller than 10^{-4}. Plot the iterates as well as the vector G during the iterations. How many iterations do you need? How much time does your SQP solver need to converge? Plot the evolution of the state and the applied controls in time.

(g) Find out how to exploit sparsity in the `quadprog` solver and solve the SQP with the sparse QP solver. How much time does your SQP solver need to converge now?

7.3 The aim of this exercise is to bring a harmonic oscillator to rest with minimal control effort. For this aim we regard the linear discrete time dynamic system:

$$\begin{bmatrix} p_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} p_k \\ v_k \end{bmatrix} + \Delta t \left(\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} p_k \\ v_k \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_k \right), \quad k = 1, \ldots, N-1 \tag{7.14}$$

with $p_1 = 10, v_1 = 0, \Delta t = 0.2, N = 51$. Denote for simplicity from now on $x_k = (p_k, v_k)^\top$.

(a) Write a MATLAB routine `[xN]=oscisim(U)` that computes x_N as a function of the control inputs $U = (u_1, \ldots, u_{N-1})^\top$. Mathematically, we will denote this function by $f_{\text{oscisim}} : \mathbb{R}^{N-1} \to \mathbb{R}^2$.

(b) To verify that your routine does what you want, plot the simulated positions $p_1, \ldots, p_N$ within this routine for the input $U = 0$.

(c) Now we want to solve the optimal control problem

$$\begin{aligned} \underset{U \in \mathbb{R}^N}{\text{minimize}} \quad & \|U\|_2^2 \\ \text{subject to} \quad & f_{\text{oscisim}}(U) = 0 \end{aligned}$$

Formulate and solve this problem with `fmincon`. Plot the solution vector U as well as the trajectory of the positions in the solution.

(d) Now add inequalities to the problem, limiting the inputs u_k in amplitude by an upper bound $|u_k| \leq u_{\max}, k = 1, \ldots, N-1$. This adds $2(N-1)$ inequalities to your problem. Which?

(e) Formulate the problem with inequalities in `fmincon`. Experiment with different values of $u_{\max}$, starting with big ones and making it smaller. If it is very big, the solution will not be changed at all. At which critical value of $u_{\max}$ does the solution start to change? If it is too small,

the problem will become infeasible. At which critical value of $u_{\max}$ does this happen?

(f) Both of the above problems are convex, i.e. each local minimum is also a global minimum. Note that the equality constraint of the optimal control problems is just a linear function at the moment. Make this constraint nonlinear and thus make the problem nonconvex. One way is to add a small nonlinearity into the dynamic system (7.14) by making the spring nonlinear, i.e. replacing the term -1 in the lower left corner of the system matrix by $-(1+\mu p_k^2)$ with a small μ, and solving the problem again. At which value of μ does the solver `fmincon` need twice as many iterations as before?

7.4 We regard again the optimal control problem from Exercise 7.3. We had previously used the Euler integrator, so let's now we use a RK4 integrator because it is more accurate. Furthermore, instead of using `fmincon`, you will write your Newton-type optimization method. For notation simplicity, let's denote f_{oscisim} by g_{sim}.

The necessary optimality conditions (KKT conditions) for the above problem are

$$2U^* + \frac{\partial g_{\text{sim}}}{\partial U}(U^*)^\top \lambda^* = 0$$
$$g_{\text{sim}}(U^*) = 0.$$

Let us introduce a shorthand for the Jacobian matrix:

$$J_{\text{sim}}(U) := \frac{\partial g_{\text{sim}}}{\partial U}(U)$$

By linearization of the constraint at some given iterate (U_k, λ_k) and neglecting its second order derivatives, we get the following (Gauss-Newton) approximation of the KKT conditions:

$$\begin{bmatrix} 2U_k \\ g_{\text{sim}}(U_k) \end{bmatrix} + \begin{bmatrix} 2\mathbb{I} & J_{\text{sim}}(U_K)^\top \\ J_{\text{sim}}(U_k) & 0 \end{bmatrix} \begin{bmatrix} U_{k+1} - U_k \\ \lambda_{k+1} \end{bmatrix} = 0$$

This system can be solved easily by a linear solve in order to obtain a new iterate U_{k+1}. But in order to do this, we need first to compute the Jacobian $J_{\text{sim}}(U)$.

(a) Implement a routine that uses finite differences, i.e. calls the function g_{sim} $(N + 1)$ times, once at the nominal value and then with each component slightly perturbed by e.g. $\delta = 10^{-4}$ in the direction of each unit vector e_k, so that we get the approximations

$$\frac{\partial g_{\text{sim}}}{\partial u_k}(U) \approx \frac{g_{\text{sim}}(U + \delta e_k) - g_{\text{sim}}(U)}{\delta}.$$

We denote the resulting function that gives the full Jacobian matrix of g_{sim} by $J_{\text{sim}} : \mathbb{R}^N \to \mathbb{R}^{2\times N}$.

(b) Now, we implement the Gauss-Newton scheme from above, but as we are not interested in the multipliers we just implement it as follows:

$$U_{k+1} = U_k - \begin{bmatrix} \mathbb{I} & 0 \end{bmatrix} \begin{bmatrix} 2\mathbb{I} & J_{\text{sim}}(U_k)^\top \\ J_{\text{sim}}(U_k) & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2U_k \\ g_{\text{sim}}(U_k) \end{bmatrix}$$

Choose an initial guess for the controls, e.g. $U = 0$, and start your iteration and stop when $\|U_{k+1} - U_k\|$ is very small. How many iterations do you need to converge? Do you have an idea why?

7.5 Throughout this exercise, we make our controlled oscillator from the previous problems slightly nonlinear by making it a pendulum and setting

$$\frac{\mathrm{d}}{\mathrm{d}t} \begin{bmatrix} p(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} v(t) \\ -C \sin(p(t)/C) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t), \quad t \in [0, T],$$

with $C := 180/\pi/4$. We again abbreviate the ODE as $\dot{x} = f(x, u)$ with $x = (p, v)^\top$, and choose again the fixed initial value $x_0 = (10, 0)^\top$ and $T = 10$. Note that p now measures the deviation from the equilibrium state in multiples of 4 degrees (i.e. we start with 40 degrees).

We also regard again the optimal control problem from the last two problems:

$$\begin{aligned} &\underset{U \in \mathbb{R}^N}{\text{minimize}} && \|U\|_2^2 \\ &\text{subject to} && g_{\text{sim}}(U) = 0 \end{aligned} \tag{7.15}$$

and we use again RK4 and do again $N = 50$ integrator steps to obtain the terminal state x_N as a function of the controls $U = [u_0, \ldots, u_{N-1}]$.

(a) Run again your Gauss-Newton scheme from the last problem, i.e. use in each iteration finite differences to compute the Jacobian matrix

$$J_{\text{sim}}(U) := \frac{\partial g_{\text{sim}}}{\partial U}(U)$$

and iterate

$$U_{k+1} = U_k - \begin{bmatrix} \mathbb{I} & 0 \end{bmatrix} \begin{bmatrix} 2\mathbb{I} & J_{\text{sim}}(U_k)^\top \\ J_{\text{sim}}(U_k) & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2U_k \\ g_{\text{sim}}(U_k) \end{bmatrix}$$

How many iterations do you need now with the nonlinear oscillator? Plot the vector U_k and the resulting trajectory of p in each Gauss-Newton iteration so that you can observe the Gauss-Newton algorithm at work.

(b) Modify your Gauss-Newton scheme so that you also obtain the multiplier vectors, i.e. iterate with $B_k = 2\mathbb{I}$ as follows:

$$\begin{bmatrix} U_{k+1} \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} U_k \\ 0 \end{bmatrix} - \begin{bmatrix} B_k & J_{\text{sim}}(U_k)^\top \\ J_{\text{sim}}(U_k) & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2U_k \\ g_{\text{sim}}(U_k) \end{bmatrix}$$

Choose as your stopping criterion now that the norm of the residual

$$\text{KKTRES}_k := \left\| \begin{bmatrix} \nabla_U \mathcal{L}(U_k, \lambda_k) \\ g_{\text{sim}}(U_k) \end{bmatrix} \right\| = \left\| \begin{bmatrix} 2U_k + J_{\text{sim}}(U_k)^\top \lambda_k \\ g_{\text{sim}}(U_k) \end{bmatrix} \right\|$$

shall be smaller than a given tolerance, e.g. TOL $= 10^{-4}$. Store the values KKTRES_k and plot their logarithms against the iteration number k. Which converge rate does it show?

(c) Now use a different Hessian approximation, namely the BFGS update, i.e. start with a unit Hessian, $B_0 = \mathbb{I}$ and then update the Hessian according to

$$B_{k+1} := B_k - \frac{B_k s_k s_k^\top B_k}{s_k^\top B_k s_k} + \frac{y_k y_k^\top}{s_k^\top y_k}.$$

with $s_k := U_{k+1} - U_k$ and $y_k := \nabla_U \mathcal{L}(U_{k+1}, \lambda_{k+1}) - \nabla_U \mathcal{L}(U_k, \lambda_{k+1})$. Devise your BFGS algorithm so that you need to evaluate the expensive Jacobian $J_{\text{sim}}(U_k)$ only once per BFGS iteration. Tipp: remember the old Jacobian $J_{\text{sim}}(U_k)$, then evaluate the new one $J_{\text{sim}}(U_{k+1})$, and only then compute B_{k+1}.

(d) Observe the BFGS iterations and regard the logarithmic plot of the norm of the residual KKTRES_k. How many iterations do you need now? Can you explain the form of the plot? What happens if you make your initial Hessian guess B_0 equal to the Gauss-Newton Hessian, ie. $B_0 = 2\mathbb{I}$?

*** In the remainder of this exercise, we want to compute the Jacobian $J_{\text{sim}}(U)$ in a more efficient way inspired by the reverse mode of algorithmic differentiation (AD). This part of the exercise sheet is optional and you should only do it if you feel motivated enough. ***

(e) For a start, save your old routine for $J_{\text{sim}}(U)$ in a separate folder to be able to compare the results of your new routine with it later.

(f) Then, note that the RK4 integrator step can be summarized in a function Φ so that the last state x_N, i.e. the output of the function $g_{\text{sim}}(U)$, is obtained by the recursion

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

Along the simulated trajectory $\{(x_k, u_k)\}_{k=0}^{N-1}$, this system can be linearized as

$$\delta x_{k+1} = A_k \delta x_k + B_k \delta u_k, \quad k = 0, \ldots, N-1,$$

where the matrices

$$A_k := \frac{\partial \Phi}{\partial x}(x_k, u_k) \quad \text{and} \quad B_k := \frac{\partial \Phi}{\partial u}(x_k, u_k),$$

can be computed by finite differences. Note that we use the symbol B_k here for coherence with the notation of linear system theory, but that this symbol B_k here has nothing to do with the Hessian matrix B_k used in the other questions.
To become specific: modify your integrator so that

- Your RK4 step is encapsulated in a single function:

  `[xnew]=RK4step(x,u)`

- You also write a function `[xnew,A,B]=RK4stepJac(x,u)` using finite differences with a step size of $\delta = 10^{-4}$
- Your integrator stores and outputs both the trajectory of states $\{x_k\}_{k=0}^{N-1}$ and the trajectory of matrices $\{(A_k, B_k)\}_{k=0}^{N-1}$. Use three dimensional tensors like `Atraj(i,j,k)`.

The interface of the whole routine could be:

`[x,Atraj,Btraj]=forwardsweep(U)`

(g) Now, using the matrices A_k, B_k, we want to compute $J_{\text{sim}}(U)$, i.e. write a routine with the interface `[Jsim]=backwardsweep(Atraj,Btraj)`. For this aim we observe that

$$\frac{\partial g_{\text{sim}}}{\partial u_k}(U) = \underbrace{(A_{N-1} A_{N-2} \cdots A_{k+1})}_{=:G_{k+1}} B_k$$

In order to compute all derivatives $\frac{\partial g_{\text{sim}}}{\partial u_k}(U)$ in an efficient way, we compute the matrices $G_{k+1} = (A_{N-1} A_{N-2} \cdots A_{k+1})$ in reverse order, i.e. we start with $k = N-1$ and then go down to $k = 0$. We start by $G_N := \mathbb{I}$ and then compute

$$G_k := G_{k+1} A_k, \quad k = N-1, \ldots, 0$$

(h) Combining the forward and the backward sweep from the previous two questions, and write a new function for $J_{\text{sim}}(U)$. It is efficient to combine it with the computation of $g_{\text{sim}}(U)$, i.e. have the interface `[gsim,Jsim]=gsimJac(U)`. Compare the result with the numerical Jacobian calculation from before by taking `norm(Jsimold-Jsimnew)`.

(i) How do the computation times of old and the new Jacobian routine scale with N? This question can be answered without numerical experiments, just by thinking.

(j) Now run your Gauss-Newton algorithm again and verify that it gives the same solution and same number of iterations as before.

7.6 In this exercise we regard again the discrete time system

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

that is generated by one RK4 step applied to the controlled nonlinear pendulum with a time step $\Delta t = 0.2$. Its state is $x = (p, v)^\top$ and the ODE $\dot{x} = f(x, u)$ is with $C := 180/\pi/10$ given as

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

The key difference respect to the previous exercise is that now, you will use the simultaneous approach with SQP and a Gauss-Newton Hessian to solve the optimal control problem.

(a) Write the function $\Phi(x_k, u_k)$ as a MATLAB code encapsulated in a single function `[xnew]=RK4step(x,u)`

(b) Let's define the OCP that we aim to solve in this section. We start by considering that the initial value is again $\bar{x}_0 = (10, 0)^\top$ and that $N = 50$. Furthermore, we take into account that we define bounds on p, v, and u, namely $p_{\max} = 10$, $v_{\max} = 10$, i.e. $x_{\max} = (p_{\max}, v_{\max})^\top$, and $u_{\max} = 3$. Finally, we can regard the OCP that we solved in the previous Exercises and that is given by Equation (7.15) as well the specific structure of the simultaneous approach, so that as a result the specific OCP is given by:

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

Formulate the nonlinear function $G(w)$, the Hessian matrix H, and

bounds $w_{\max}$, where $w = (x_0, u_0, \ldots, u_{N-1}, x_N) \in \mathbb{R}^n$ and $n = 152$, so that the above OCP can be written in the following form:

$$\begin{aligned} \underset{w \in \mathbb{R}^{152}}{\text{minimize}} \quad & w^\top H w \\ \text{subject to} \quad & G(w) = 0, \\ & -w_{\max} \leq w \leq w_{\max}. \end{aligned}$$

Define what H_x and H_u need to be in the Hessian

$$H = \begin{bmatrix} H_x & & & \\ & H_u & & \\ & & \ddots & \\ & & & H_x \end{bmatrix}, \quad H_x = \begin{bmatrix} \quad \end{bmatrix} \quad , H_u = \begin{bmatrix} \quad \end{bmatrix}.$$

Construct the matrix H and vector $w_{\max}$ in MATLAB, and write a MATLAB function `[G]=Gfunc(w)`.

(c) Check if your function $G(w)$ does what you want by writing a forward simulation function `[w]=simulate(x0,U)` that simulates, for a given initial value x_0 and control profile $U = (u_0, \ldots, u_{N-1})$, the whole trajectory $x_1, \ldots, x_N$ and constructs from this the full vector $w = (x_0, u_0, x_1, \ldots, x_N)$. If you generate for any x_0 and U a vector w and then you call your function $G(w)$ with this input, nearly all your residuals should be zero. Which components will not be zero?

As a test, simulate e.g. with $x_0 = (5, 0)$ and $u_k = 1$, $k = 0, \ldots, N-1$ in order to generate w, and then call $G(w)$, to test that your function G is correct.

(d) The SQP with Gauss-Newton Hessian solves a linearized version of this problem in each iteration. More specific, if the current iterate is $\bar{w}$, the next iterate is the solution of the following QP:

$$\begin{aligned} \underset{w \in \mathbb{R}^{152}}{\text{minimize}} \quad & w^\top H w \\ \text{subject to} \quad & G(\bar{w}) + J_G(\bar{w})(w - \bar{w}) = 0, \\ & -w_{\max} \leq w \leq w_{\max}. \end{aligned} \tag{7.16}$$

In order to implement the Gauss-Newton method we need the Jacobian $J_G(w) = \frac{\partial G}{\partial w}(w)$. Considering that $J_G(w)$ is block sparse, where the the blocks are either (negative) unit matrices, partial derivatives $A_k = \frac{\partial \Phi}{\partial x}(x_k, u_k)$ or partial derivatives $B_k = \frac{\partial \Phi}{\partial u}(x_k, u_k)$, fill in the corre-

sponding blocks in the following matrix:

$$J_G(w) = \begin{bmatrix} & & & & \\ & \ddots & \ddots & \ddots & \\ & & & & \end{bmatrix}$$

(e) We compute the Jacobian $J_G(w)$ by finite differences, i.e. perturbing all 152 directions one after the other. Give your routine e.g. the name `[G,J]=GfuncJacSlow(w)`. Compute J_G for a given w (e.g. the one from above) and look at the structure this matrix, e.g. using the command `spy(J)`.

(f) Write a function `[wplus]=GNStep(w)` that performs one SQP-Gauss-Newton step by first calling `[G,J]=GfuncJac(w)` and then solving the resulting QP (7.16) using `quadprog` from MATLAB. Note that the QP is a very sparse QP but that this sparsity is not exploited in full during the call of `quadprog`.

(g) Write a loop around your function `GNStep`, initialize the GN procedure at at $w = 0$, and stop the iterations when $\|w_{k+1} - w_k\|$ gets smaller than 10^{-4}. Plot the iterates as well as the vector G during the iterations. How many iterations do you need?

(h) A different algorithm is obtained if we overwrite before each call of the function `GNStep` the values for the states within `w` by the result of a forward simulation, using the corresponding controls and the initial value $\bar{x}_0$. Run it again with this modification, using the same zero initialization for the controls. How many iterations do you need now? Do you know to which of the algorithms from the previous exercises this new method is equivalent?

(i) Finally, you can construct the Jacobian in a much more computationally efficient way:

- First write a function `[xnew,A,B]=RK4stepJac(x,u)` using finite differences with a step size of $\delta = 10^{-4}$. Here, $A = \frac{\partial \Phi}{\partial x}(x,u)$ and $B = \frac{\partial \Phi}{\partial u}(x,u)$.
- Using this function `[xnew,A,B]=RK4stepJac(x,u)`, implement a function `[G,J]=GfuncJacFast(w)`.
- Compare if the result is correct by taking the difference of the Jacobians you obtain by `[G,J]=GfuncJacFast(w)` and `[G,J]=GfuncJacSlow(w)`.

8
Dynamic Programming

In view of all that we have said in the foregoing sections, the many obstacles we appear to have surmounted. What casts the pall over our victory celebration? It is the curse of dimensionality, a malediction that has plagued the scientist from earliest days.
— Richard E. Bellman

Dynamic programming (DP) is a very different approach to solve optimal control problems than the ones presented previously. The methodology was developed in the fifties and sixties of the 20th century, most prominently by Richard Bellman [4] who also coined the term dynamic programming. Interestingly, dynamic programming is easiest to apply to systems with discrete state and control spaces, so that we will introduce this case first. When DP is applied to discrete time systems with continuous state spaces, some approximations have to be made, usually by discretization. Generally, this discretization leads to exponential growth of computational cost with respect to the dimension n_x of the state space, what Bellman called the "curse of dimensionality". It is the only but major drawback of DP and limits its practical applicability to systems with $n_x \approx 6$. In the continuous time case, DP is formulated as a partial differential equation in the state space, the Hamilton-Jacobi-Bellman (HJB) equation, suffering from the same limitation; but this will be treated in Chapter 11. On the positive side, DP can easily deal with all kinds of hybrid systems or non-differentiable dynamics, and it even allows us to treat stochastic optimal control with recourse, or minimax games, without much additional effort. An excellent textbook on discrete time optimal control and dynamic programming is [8]. Let us now start with discrete control and state spaces.

8.1 Dynamic Programming in Discrete State Space

Let us regard a dynamic system

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

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

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

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

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

$$\begin{aligned} J_k(\bar{x}_k) \;=\; \underset{\substack{x_k, u_k, \ldots, \\ u_{N-1}, x_N}}{\text{minimize}} \quad & \sum_{i=k}^{N-1} L(x_i, u_i) + E(x_N) \\ \text{subject to} \quad & f(x_i, u_i) - x_{i+1} = 0, \quad i = k, \ldots, N-1, \\ & \bar{x}_k - x_k = 0. \end{aligned} \tag{8.1}$$

Thus, each function $J_k : \mathbb{X} \to \mathbb{R}_\infty$ summarizes the cost-to-go to the end

when starting at a given state. For the case $k = N$ we trivially have $J_N(x) = E(x)$. The principle of optimality states now that for any $k \in \{0, \dots, N-1\}$ holds

$$J_k(\bar{x}_k) = \underset{u}{\text{minimize}} \quad L(\bar{x}_k, u) + J_{k+1}(f(\bar{x}_k, u)). \tag{8.2}$$

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

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

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

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

In this way, DP allows us to solve the optimal control problem up to global optimality, but with a different complexity than simple enumeration. To assess its complexity, let us remark that the most cost intensive step is the generation of the N cost-to-go functions J_k. Each recursion step (8.2) needs to go through all $n_{\mathbb{X}}$ states x. For each state it needs to test $n_{\mathbb{U}}$ controls u by evaluating once the system $f(x, u)$ and stage cost $L(x, u)$, which by definition costs one computational unit. Thus, the overall computational complexity is $O(Nn_{\mathbb{X}}n_{\mathbb{U}})$. Compared with simple enumeration, where we had $O(Nn_{\mathbb{U}}^N)$, DP is often much better even for moderately sized horizons N. Let us for example assume an optimal control problem with $n_{\mathbb{U}} = 10$, $n_{\mathbb{X}} = 1000$, $N = 100$. Then simple enumeration has a cost of 10^{102} while DP has a cost of 10^6.

One of the main advantages of dynamic programming, that can likewise be defined for continuous state spaces, is that we do not need to make any assumptions (such as differentiability or convexity) on the functions f, L, E defining the problem, and still it solves the problem up to global optimality. On the other hand, if it shall be applied to a continuous state space, we have to represent the functions J_k on the computer, e.g., by tabulation on a grid in state space. If the continuous state space $\mathbb{X}_{\text{cont}}$ is a box in dimension n_x, and if we use a rectangular grid with m intervals in each dimension, then the total number of grid points is m^{n_x}. If we perform DP on this grid, then the above complexity estimate is still valid, but with $n_{\mathbb{X}} = m^{n_x}$. Thus, when DP is applied to systems with continuous state spaces, it has exponential complexity in the dimension of the state space; it suffers from what Bellman called the *curse*

of dimensionality. There exist many ways to approximate the value function, e.g., by neural networks or other functional representations [9], but the global optimality guarantee of dynamic programming is lost in these cases. On the other hand, there exists one special case where DP can be performed exactly in continuous state spaces, that we treat next.

8.2 Linear Quadratic Problems

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

$$\begin{aligned} \underset{x,\,u}{\text{minimize}} \quad & \sum_{k=0}^{N-1} \begin{bmatrix} x_k \\ u_k \end{bmatrix}^\top \begin{bmatrix} Q_k & S_k^\top \\ S_k & R_k \end{bmatrix} \begin{bmatrix} x_k \\ u_k \end{bmatrix} + x_N^\top P_N x_N \\ \text{subject to} \quad & x_0 - \bar{x}_0 = 0, \\ & x_{k+1} - A_k x_k - B_k u_k = 0, \quad k = 0, \dots, N-1. \end{aligned} \tag{8.3}$$

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

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

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

Theorem 8.1 (Quadratic Representation of Value Function)**.** *If $R + B^\top P B$ is positive definite, then the minimum $J_{\text{new}}(x)$ of one step of the DP recursion*

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

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

$$P_{\text{new}} = Q + A^\top P A - (S^\top + A^\top P B)(R + B^\top P B)^{-1}(S + B^\top P A). \tag{8.4}$$

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

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

Then it uses the fact that for invertible $\bar{R} = R + B^\top P B$ this problem can be solved explicitly, yielding the formula (8.4), by a direct application of the *Schur complement lemma*, that can easily be verified by direct calculation.

Lemma 8.2 (Schur Complement Lemma)**.** *If $\bar{R}$ is positive definite then*

$$\min_u \begin{bmatrix} x \\ u \end{bmatrix}^\top \begin{bmatrix} \bar{Q} & \bar{S}^\top \\ \bar{S} & \bar{R} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} = x^\top \big(\bar{Q} - \bar{S}^\top \bar{R}^{-1} \bar{S} \big) x$$

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

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

$$P_k = Q_k + A_k^\top P_{k+1} A_k - (S_k^\top + A_k^\top P_{k+1} B_k)(R_k + B_k^\top P_{k+1} B_k)^{-1}(S_k + B_k^\top P_{k+1} A_k). \tag{8.5}$$

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

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

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

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

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

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

$$\begin{aligned} \underset{x,\,u}{\text{minimize}} \quad & \sum_{i=0}^{N-1} \begin{bmatrix} 1 \\ x_k \\ u_k \end{bmatrix}^\top \begin{bmatrix} * & q_k^\top & s_k^\top \\ q_k & Q_k & S_k^\top \\ s_k & S_k & R_k \end{bmatrix} \begin{bmatrix} 1 \\ x_k \\ u_k \end{bmatrix} + \begin{bmatrix} 1 \\ x_N \end{bmatrix}^\top \begin{bmatrix} * & p_N^\top \\ p_N & P_N \end{bmatrix} \begin{bmatrix} 1 \\ x_N \end{bmatrix} \\ \text{subject to} \quad & x_0 - x_0^{\text{fix}} = 0, \\ & x_{k+1} - A_k x_k - B_k u_k - c_k = 0, \quad k = 0, \dots, N-1. \end{aligned} \tag{8.6}$$

These optimization problems appear at many occasions, for example as linearizations of nonlinear optimal control problems, as in Chapter 7.3, in reference tracking problems with $L_k(x_k, u_k) = \|x_k - x_k^{\text{ref}}\|_Q^2 + \|u_k\|_R^2$, or in *moving horizon estimation* (MHE) with cost $L_k(x_k, u_k) = \|C x_k - y_k^{\text{meas}}\|_Q^2 + \|u_k\|_R^2$. They

can be treated by exactly the same recursion formulae as above, by augmenting the system states x_k to

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

and replacing the dynamics by

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

with initial value

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

Then the problem (8.6) can be reformulated in the form of problem (8.3) and can be solved using exactly the same difference Riccati equation formula as before!

8.3 Infinite Horizon Problems

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

$$\begin{aligned} \underset{x,\,u}{\text{minimize}} \quad & \sum_{k=0}^{\infty} L(x_k, u_k) \\ \text{subject to} \quad & x_0 - \bar{x}_0 = 0, \\ & x_{k+1} - f(x_k, u_k) = 0, \qquad k = 0, \ldots, \infty. \end{aligned}$$

Interestingly, the cost-to-go function $J_k(x_k)$ defined in Equation (8.1) becomes independent of the index k, i.e, it holds that $J_k = J_{k+1}$ for all k. This directly leads to the *Bellman Equation*:

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

The optimal controls are obtained by the function

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

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

8.4 The Linear Quadratic Regulator

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

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

For its solution, we just require a stationary solution of the Riccati recursion (8.5), setting $P_k = P_{k+1}$, which yields the so called *algebraic Riccati equation in discrete time*

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

This is a nonlinear matrix equation in the symmetric matrix P, i.e., with $n_x(n_x + 1)/2$ unknowns. It can either be solved by an iterative application of the difference Riccati recursion (8.5) starting with, e.g., a zero matrix $P = 0$, or by faster converging procedures such as Newton-type methods, where, however, care has to be taken to avoid possible shadow solutions that are not positive definite. Once the solution matrix P is found, the optimal feedback control $u^*(x)$ is given by

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

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

8.5 Robust and Stochastic Dynamic Programming

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

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

starting with

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

The solution obtained by DP takes into account that we can react to the actions by the adversary, i.e., that we can apply feedback, and in the model predictive control (MPC) literature such a feedback law is sometimes called Closed-Loop Robust Optimal Control [6].

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

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

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

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

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

8.6 Interesting Properties of the DP Operator

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

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

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

Theorem 8.3 (Monotonicity of DP)**.** *Regard two value functions J and J'. If $J \geq J'$ in the sense that for all $x \in \mathbb{X}$ holds that $J(x) \geq J'(x)$ then also*

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

The proof is

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

This monotonicity property holds also for robust or stochastic dynamic programming, and is for example used in existence proofs for solutions of the stationary Bellman equation, or in stability proofs of model predictive control (MPC) schemes [53].

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

Theorem 8.4 (Convex dynamic programming)**.** *If the system is affine in* (x, u)*, i.e.* $f(x, u, w) = A(w)x + B(w)u + c(w)$*, and if the stage cost* $L(x, u, w)$ *is convex in* (x, u)*, then the DP, the robust DP, and the stochastic DP operators* T *preserve convexity of* J*, i.e. if* J *is a convex function, then* $T[J]$ *is again a convex function.*

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

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

is convex. □

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

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

It is an interesting fact that that for piecewise linear convex costs and constraints and polyhedral uncertainty this representation is exact and leads to an exact robust DP algorithm that might be called *polyhedral DP* [6, 27]. The polyhedral convex representability of the cost-to-go for linear systems with piecewise linear cost is indirectly exploited in some explicit MPC approaches [58, 5]. Polyhedral representations with a limited number of facets can also be used to approximate a convex cost-to-go and still yield some guarantees on the closed-

loop system [14, 15, 42]. Finally, note that also the linear quadratic regulator is a special case of convex dynamic programming.

8.7 The Gradient of the Value Function

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

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

The dynamic programming recursion for this problem is given by:

$$J_N(x) = E(x), \quad J_k(x) = \min_u L(x, u) + J_{k+1}(f(x, u)), \quad k = N-1, \ldots, 0. \tag{8.7}$$

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

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

where u_k is defined by

$$u_k = \arg\min_u L(x_k, u) + J_{k+1}(f(x_k, u)). \tag{8.8}$$

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

$$\nabla_u L(x_k, u_k) + \frac{\partial f}{\partial u}(x_k, u_k)^\top \nabla J_{k+1}(f(x_k, u_k)) = 0 \tag{8.9}$$

which defines u_k locally if the problem is locally strictly convex, i.e., its objective has a positive definite Hessian at (x_k, u_k). We now formulate simple

conditions on x_k and u_k that follow necessarily from the DP recursion. For this aim we first note that on the optimal trajectory holds $x_{k+1} = f(x_k, u_k)$ and that we trivially obtain along the optimal trajectory

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

This implies for example that the value function remains constant on the whole trajectory for problems with zero stage costs. However, it is even more interesting to regard the gradient $\nabla J_k(x_k)$ along the optimal state trajectory. If we differentiate (8.7) at the point x_k with respect to x we obtain

$$\begin{aligned} \nabla J_N(x_N) &= \nabla E(x_N), \\ \nabla J_k(x_k)^\top &= \frac{\mathrm{d}}{\mathrm{d}x} \underbrace{L(x_k, u_k) + J_{k+1}(f(x_k, u_k))}_{=:\tilde{J}_k(x_k, u_k)} \quad k = N-1, \dots, 0. \end{aligned}$$

In the evaluation of the total derivative it is needed to observe that the optimal u_k is via (8.9) an implicit function of x_k. However, it turns out that the derivative does not depend on $\frac{\mathrm{d}u_k}{\mathrm{d}x_k}$ because of

$$\frac{\mathrm{d}}{\mathrm{d}x} \tilde{J}_k(x_k, u_k) = \frac{\partial \tilde{J}_k}{\partial x}(x_k, u_k) + \underbrace{\frac{\partial \tilde{J}_k}{\partial u}(x_k, u_k)}_{=0} \frac{\mathrm{d}u_k}{\mathrm{d}x_k},$$

where the partial derivative with respect to u is zero because of (8.9). Thus, the gradients of the value function at the optimal trajectory have to satisfy the recursion

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

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

$$\lambda_k = \nabla J_k(x_k).$$

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

8.8 A Discrete Time Minimum Principle

Collecting all necessary conditions of optimality that we just derived, but substituting $\nabla J_k(x_k)$ by λ_k we arrive indeed exactly to the same conditions (7.7)

that we derived in Chapter 7 in a completely different way.

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

In the context of continuous time problems, we will arrive at a very similar formulation, which has the interesting features that the recursion for λ becomes a differential equation that can be integrated forward in time if desired, and that the optimization problem in (8.8) does only depend on the gradient of J. This will facilitate the formulation and numerical solution of the necessary optimality conditions as a boundary value problem.

8.9 Iterative Dynamic Programming

8.10 Differential Dynamic Programming

Exercises

8.1 Consider a very simple system with state $x \in \{1, 2, \dots, 10\}$ and controls $u \in \{-1, 0, 1\}$ and time invariant dynamics $f(x, u) = x + u$ and stage cost $L(x, u) = |u|$ on a horizon of length $N = 3$. The terminal cost $E(x)$ is given by zero if $x = 5$ and by 100 otherwise. Take pen and paper and compute and sketch the cost to go functions J_3, J_2, J_1, J_0.

8.2 Use dynamic programming to solve the following simple discrete time OCP with one state and one control by hand. On the way towards the solution, explicitly state the cost to go functions $J_2(x), J_1(x), J_0(x)$ and feedback control laws $u_0^*(x)$ and $u_1^*(x)$.

$$
\begin{aligned}
\underset{\substack{x_0, x_1, x_2, \\ u_0, u_1}}{\text{minimize}} \quad & \sum_{k=0}^{1} u_k^2 \; + \; 10 x_2^2 \\
\text{subject to} \quad & x_0 = 5, \\
& x_{k+1} = x_k + u_k, \quad k = 0, 1.
\end{aligned}
$$

8.3 Regard the discrete time damped-spring system

$$x_{k+1} = \begin{pmatrix} 1 & 0.02 \\ -0.1 & 0.992 \end{pmatrix} x_k + \begin{pmatrix} 0 \\ 0.02 \end{pmatrix} u_k$$

over the horizon of $N = 600$, with initial state $x_0 = [10, 0]$.

(a) Simulate and plot the uncontrolled system ($u = 0$) as a baseline.

(b) Using dynamic programming, minimize the cost function:

$$\sum_{k=0}^{N-1} \left(x_k^\top Q x_k + u_k^\top R u_k \right) + x_N^\top P_N x_N$$

with

$$Q = \begin{pmatrix} \frac{1}{2^2} & 0 \\ 0 & \frac{1}{3^2} \end{pmatrix}, \quad R = \begin{pmatrix} \frac{1}{6^2} \end{pmatrix}, \quad P_N = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Plot the two states and control against the uncontrolled system.

(c) Consider the infinite-horizon system ($N \to \infty$) with cost function:

$$\sum_{k=0}^{\infty} \left(x_k^\top Q x_k + u_k^\top R u_k \right)$$

What control policy will minimize this cost function? Implement this control policy and simulate for $N = 600$. Plot this in state and control against the previous two trajectories.

8.4 In this Exercise we regard again the discrete time system

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

that is generated by one RK4 step applied to the controlled nonlinear pendulum, as defined in Exercise 7.6. Furthermore, we assume that you have the MATLAB function `[xnew]=RK4step(x,u)` available. If not, refer to Exercise 7.6 for implementation.

We regard the same optimal control problem as last time, with the initial value $\bar{x}_0 = (10, 0)^\top$ and $N = 50$ time steps, and bounds $p_{\max} = 10$, $v_{\max} = 10$, i.e. $x_{\max} = (p_{\max}, v_{\max})^\top$, and $u_{\max} = 10$. In contrast to last time and because of approximation errors, we now have to relax the terminal constraint $x_N = 0$ to a small box $-x_{N,\max} \leq x_N \leq x_{N,\max}$ with $x_{N,\max} = (5, 5)^\top$. We also add a small terminal cost term $10\|x_N\|_2^2$. As a result, the optimization problem we want to solve is given by

$$
\begin{aligned}
= 0 \underset{\substack{x_0,u_0,x_1,\ldots,\\ u_{N-1},x_N}}{\text{minimize}} \quad & \sum_{k=0}^{N-1} \|u_k\|_2^2 + 10\|x_N\|_2^2 \\
\text{subject to} \quad & \Phi(x_k, u_k) - x_{k+1} = 0, \qquad k = 0, \ldots, N-1, \\
& -x_{\max} \;\le\; x_k \le x_{\max}, \quad k = 0, \ldots, N-1, \\
& -x_{N,\max} \;\le\; x_N \le x_{N,\max}, \\
& -u_{\max} \;\le\; u_k \le u_{\max}, \quad k = 0, \ldots, N-1
\end{aligned}
$$

(a) Discretize the state and control spaces choosing step sizes in all dimensions of size 1. Considering that $\{x \in \mathbb{R}^2 | -x_{\max} \le x \le x_{\max}\}$ and $\{u \in \mathbb{R} | -u_{\max} \le u \le u_{\max}\}$ this would result in $n = 21 \cdot 21 = 441$ state grid points and $m = 7$ control grid points. Let us denote the gridded space and control spaces by $\mathbb{X}$ and $\mathbb{U}$. As a first step, define the terminal cost-to-go function $J_N(x)$ on the n state grid points $x \in \mathbb{X}$, i.e. define all elements of 21 by 21 matrix that you might call `Jmat`. For infeasible values, i.e. those that exceed the tight bounds of the terminal state, choose a very large number, e.g. 10^6. For later use, you might also define a three dimensional tensor `JmatTen(i,j,kp1)` with `kp1`$= 1, \ldots, 51$ in order to store all J_k.

(b) The next problem in implementing dynamic programming is that we cannot expect that we exactly hit with $\Phi(x, u)$ any of the grid points, i.e. unfortunately we have even for $x \in \mathbb{X}$ and $u \in \mathbb{U}$ that usually $\Phi(x, u) \notin \mathbb{X}$. We can resolve this issue by rounding the value of the output of Φ to the next grid point (this corresponds to a piecewise constant representation of J_{k+1} in the DP equation). Let us denote this function by $\tilde{\Phi} : \mathbb{X} \times \mathbb{U} \to \mathbb{X}$. Thus, write a MATLAB function $[xnew] = RK4round(x, u)$ which has the property that it always maps to the next grid point in $\mathbb{X}$. Note that we introduce uncontrolled discretiation errors here.

(c) Last, implement the dynamic programming recursion, i.e. write two nested loop: the outer loop goes through each $x \in \mathbb{X}$ and solves the optimization problem

$$J_k(x) = \min_{u \in \mathbb{U}} \|u\|_2^2 + J_{k+1}(\tilde{\Phi}(x, u))$$

by enumerating over all $u \in \mathbb{U}$ in the inner loop. Summarize your dynamic programming operator T in a function `Jmatplus=DPoper(Jmat)`.

(d) Starting at J_N, generate all fifty more matrices $J_{N-1}, \ldots, J_0$, and visualize your cost-to-go functions J_k by plotting the matrix entries as a

two dimensional function (cutting away the "infinitely" high values). Compare J_N and J_0. Can you interpret the form of them ?

(e) In order for DP to be useful, we need to generate control actions for a given state. They are easily obtained by

$$u_k^*(x) = \arg\min_{u \in \mathbb{U}} \|u\|_2^2 + J_{k+1}(\tilde{\Phi}(x, u))$$

Write a function `u=DPcont(x,J)` that gives you the dynamic programming feedback.

(f) If you want to generate the optimal trajectory for a given inititial state x_0, we can do a *closed-loop* simulation, i.e. we simulate

$$x_{k+1} = \Phi(x_k, u^*(x_k, J_{k+1})).$$

Note that we do *not* use $\tilde{\Phi}$ in this forward simulation, but Φ, because we want the feedback to compensate for our discretization errors. Generate the trajectories for x and u for the above optimal control problem.

(g) Assume a small bounded perturbation against which you want to robustify your controller. Assume for this that your function Φ is perturbed by a perturbation $w_k \in [-1, 1]^2$ as follows

$$x_{k+1} = \underbrace{\Phi(x_k, u_k) + 0.1 u_k w_k}_{=:\Phi_{\text{rob}}(x_k, u_k, w_k)}$$

Discretize the cube in which w lives e.g. by a 3 by 3 grid $\mathbb{W}$. Also, you need again to round the result so that you have a function $\tilde{\Phi}_{\text{rob}} : \mathbb{X} \times \mathbb{U} \times \mathbb{W} \to \mathbb{X}$. Now solve instead of the nominal DP recursion the robust DP recursion

$$J_k(x) = \min_{u \in \mathbb{U}} \max_{w \in \mathbb{W}} \|u\|_2^2 + J_{k+1}(\tilde{\Phi}_{\text{rob}}(x, u, w)).$$

Generate the nominal trajectory, i.e. with all $w_k = 0$ by the closed-loop simulation. Plot the result. What is different now?

(h) Last, generate a random scenario of values w_k inside the cube of perturbations, and simulate your closed-loop system again. Verify that the terminal constraint is still satisfied.

8.5 In this task we are using Dynamic Programming to find optimal controls to swing up a pendulum. The state of the system is $x = [\phi, \omega]^\top$ where ϕ is the angle and ω the angular velocity of the pendulum. When $\phi = 0$, the pendulum is in its inverse position, e.g. the mass is at the top. The system dynamics are given by

$$\dot{\phi} = \omega$$
$$\dot{\omega} = 2\sin(\phi) + u$$

where $\omega \in [\omega_{\min}, \omega_{\max}]$, and $u \in [u_{\min}, u_{\max}]$.

To find the controls for the pendulum swing-up, we are solving the following optimal control problem:

$$\begin{aligned} \underset{\substack{x_0,\dots,x_N,\\ u_0,\dots,u_{N-1}}}{\text{minimize}} \quad & \sum_{k=0}^{N-1} (\phi_k^2 + u_k^2) \\ \text{subject to} \quad & \bar{x}_0 = x_0, \\ & f(x_k, u_k) - x_{k+1} = 0, \qquad k = 0, \dots, N-1, \\ & u_{\min} \le u_k \le u_{\max}, \quad k = 0, \dots, N-1, \\ & \omega_{\min} \le \omega_k \le \omega_{\max}, \quad k = 0, \dots, N. \end{aligned} \tag{8.10}$$

Dynamic Programming requires a system which is discrete in space and in time. We already prepared the discretization of the continuous system for you in the file `pendulum_template.m` on the course webpage. The discretization is done in the following way:

The discrete versions of ϕ, ω and u live in the integer space $\mathbb{Z}$ and thus are denoted by ϕ_Z, ω_Z and u_Z. The conversion from real space to integer space is done by projection of the variables into N_ϕ, N_ω, N_u equally spaced bins in the range of the variables. In the template file you find predefined functions to convert the variables between integer and real numbers, e.g. `phiZ_to_phi` and `phi_to_phiZ`.

To complete the tasks, fill in the missing parts of the template file `pendulum_template.m`.

(a) Use the function `integrate_Z` to simulate the system in discrete space with $x_0 = [0.4, 0]^\top$ and $u = 0$. Do $N = 60$ integration steps and use a timestep of $h = 0.12$. Plot the evolution of ϕ and ω in time (in continuous state space). Make a plot (animation) that shows the motion of the pendulum. Assume a rod length of $1m$. You don't need to submit the animation in the pdf, it's just for you to see if the system behaves well.

(b) In the following section of the template file you see the precalculation of all integrations in the discrete state space to avoid unnecessary computations during Dynamic Programming. For a given combination of ϕ_Z, ω_Z, and u_Z, the resulting state from integration $[\phi_Z^+, \omega_Z^+]$

is stored in the lookup tables `PhiNext` and `WNext` and the costs are stored in the table `L`.

Use the lookup tables to do the same simulation as in Task 1. Plot the evolution of ϕ and ω in time (in continuous state space).

(c) Implement the backward pass (recursion) of Dynamic Programming, i.e. calculate the cost-to-go function $J_k(x_k)$ going from $k = N$ to $k = 1$. For $k = N$, the cost-to-go is initialized zero for all states (no terminal cost). Fill in the missing lines in the template file for this task. Use $N = 60$ and $h = 0.12$.

(d) Simulate the system using the optimal controls which are given by

$$u_k^*(x_k) = \arg\min_u \quad \phi_k^2 + u^2 + J_{k+1}(f(x_k, u))$$

starting from $x_0 = [-\pi, 0]^\top$. Plot the evolution of ϕ and ω in time (in continuous state space). Make a plot (animation) that shows the motion of the pendulum to see if the pendulum swings up.

8.6 Consider the inverted pendulum problem defined by the optimal control problem given by Equation (8.10). This time, we will try to find an approximate solution to the optimal control problem by linearising the original non-linear problem and finding a solution to the corresponding linear-quadratic problem of the form:

$$\begin{aligned} \underset{\substack{x_0,\ldots,x_N,\\ u_0,\ldots,u_{N-1}}}{\text{minimize}} \quad & \sum_{k=0}^{N-1} \begin{bmatrix} x_k \\ u_k \end{bmatrix}^\top \begin{bmatrix} Q_k & S_k^\top \\ S_k & R_k \end{bmatrix} \begin{bmatrix} x_k \\ u_k \end{bmatrix} + x_N^\top P_N x_N \\ \text{subject to} \quad & x_0 = \bar{x}_0, \\ & x_{k+1} = A_k x_k + B_k u_k, \quad k = 0, \ldots, N-1. \end{aligned}$$

In order to solve the different tasks download the `lqr_template.m` from the website.

(a) Linearise the system dynamics around the point $x = [0, 0]^\top$, $u = 0$ analytically by differentiation to obtain a continuous time system of the form: $\dot{x} = A_c x + B_c u$. Then you can obtain the corresponding discrete time system $x_{k+1} = Ax_k + Bu_k$ with a timestep of 0.1 using the Matlab commands:

```
sysc = ss(Ac,Bc,eye(2),0);
sysd = c2d(sysc,0.1);
A = sysd.a;
B = sysd.b;
```

Specify the continuous time and discrete time system matrices A_c, B_c, A, B.

(b) Bring the objective of the optimal control problem into a quadratic form and specify the matrices Q, R, S.

(c) Calculate recursively the P_k matrices (Difference Ricatti Equation) for $k = N-1, \ldots, 0$ with $N = 60$ starting from $P_N = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$.

(d) Calculate the first cost-to-go J_0 for the linear system for each state of the dynamic programming exercise of exercise sheet 3. Make a 3D contour plot with a fine level-step of J_0 using the matlab commands:

```
[C,handle] = contour3(J0);
set(handle,'LevelStep',get(handle,'LevelStep')*0.2)
```

Compare the contour plot with the contour plot of the non-linear cost-to-go function of the dynamic programming exercise the you can get with the `get_first_J()` function in the template.
What are the similarities/differences of the two contour plots?

(e) Starting from $x_0 = [-\frac{\pi}{8}, 2]^\top$, calculate the optimal feedback and the complete optimal trajectory of the linear quadratic optimal control problem by forward recursion.
Make a plot with the evolution of the state in time and a plot of the optimal feedback controls vs. time. Does the controller bring the system to the steady state?

(f) Apply the same optimal controls (open-loop) to the non-linear pendulum system. Start from the same initial state x_0 and simulate the system using the `integrate_rk4` function. Make a plot of state evolution and controls as before. Does the controller bring the system to the steady state? Discuss the result.

(g) Implement a feedback controller, i.e. calculate the optimal feedback for the current state and simulate the non-linear system using `integrate_rk4`. Make a plot of state evolution and controls as before. Does the controller bring the system to the steady state?

(h) Solve the Algebraic Ricatti Equation by iteratively calculating the P_k matrices until convergence. We say, convergence is reached if the Frobenius norm of differences between the current matrix P_{cur} and the next matrix P_{next} is below 10^{-5}:

```
norm(Pnext-Pcur, 'fro') <= 1e-5
```

(i) Use the solution to the Algebraic Ricatti Equation to implement a Linear-Quadratic-Regulator(LQR). Simulate the system with `integrate_rk4`.

Make a plot of state evolution and controls as before. Does the controller stabilize the system at the steady state?

8.7 We shall consider a simple OCP with two states (x_1, x_2) and one control (u):

$$\begin{aligned}
\underset{x,\,u}{\text{minimize}} \quad & \int_0^{\top} x_1(t)^2 + x_2(t)^2 + u(t)^2 \, dt \\
\text{subject to} \quad & \dot{x}_1 = (1 - x_2^2)\, x_1 - x_2 + u, \qquad x_1(0) = 0, \\
& \dot{x}_2 = x_1, \qquad\qquad\qquad\qquad x_2(0) = 1, \\
& -1 \le x_1(t) \le 1, \\
& -1 \le x_2(t) \le 1, \\
& -1 \le\ u(t)\ \le 1,
\end{aligned}$$

with $T = 10$.

To be able to solve the problem using dynamic programming, we parameterize the control trajectory into $N = 20$ piecewise constant intervals. On each interval, we then take 1 step of a RK4 integrator in order to get a discrete-time OCP of the form:

$$\begin{aligned}
\underset{x,\,u}{\text{minimize}} \quad & \sum_{k=0}^{N-1} F_0(x_1^{(k)}, x_2^{(k)}, u^{(k)}) \sum_{k=0}^{N-1} F_0(x_1^{(k)}, x_2^{(k)}, u^{(k)}) \\
\text{subject to} \quad & x_1^{(k+1)} = F_1(x_1^{(k)}, x_2^{(k)}, u^{(k)}), \quad k = 0, \ldots, N-1, \quad x_1^{(0)} = 0, \\
& x_2^{(k+1)} = F_2(x_1^{(k)}, x_2^{(k)}, u^{(k)}), \quad k = 0, \ldots, N-1, \quad x_2^{(0)} = 1, \\
& -1 \le x_1^{(k)} \le 1, \qquad k = 0, \ldots, N, \\
& -1 \le x_2^{(k)} \le 1, \qquad k = 0, \ldots, N, \\
& -1 \le u^{(k)} \le 1, \qquad k = 0, \ldots, N-1.
\end{aligned}$$

(a) Implement the RK4 integrator for the system dynamics. Take a look to Chapter 10 if you need help.

(b) The continuous x_1, x_2 and u are uniformly discretized in 101 values. Create the vectors containing the discrete values of the variables. Modify the integrator so that the dynamics round up to the closest discrete value.

(c) Using the stage cost and starting at $x_1(T)$, $x_2(T)$, recursively compute the cost of every possible state $(x_1^{(k)}, x_2^{(k)}, u^{(k)})$.

(d) Using the initial conditions solve the problem using dynamic programming.

(e) Add the additional end-point constraint $x_1(T) = -0.5$ and $x_2(T) = -0.5$. How does the solution change?

9

Continuous Time Optimal Control Problems

When we are confronted with a problem whose dynamic system lives in continuous time and whose control inputs are a continuous profile, i.e. functions of time living in an ∞-dimensional functional space, we speak of a *continuous time optimal control problem*. This type of problem is the focus of this third part of this script. We will encounter variations of the same concepts as in the discrete time setting, such as Lagrange multipliers λ, the value function J, or the difference between sequential or simultaneous methods. Some numerical methods and details, however, are only relevant to the continuous time setting, such as the indirect methods and Pontryagin's Maximum Principle described in Chapter 12, or the ODE solvers with sensitivity generation described in Section 10.4.

9.1 Formulation of Continuous Time Optimal Control Problems

In an ODE setting, many continuous-time optimal control problem can be stated as follows:

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

The problem and its variables are visualized in Figure 9.1.

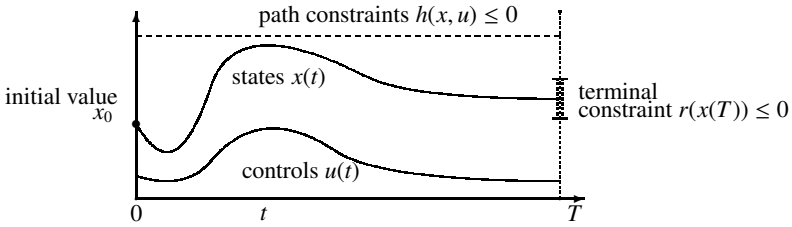


Figure 9.1 The variables and constraints of a continuous time optimal control problem.

The integral cost contribution $L(x,u)$ is sometimes called the *Lagrange term* (which should not be confused with the Lagrange function) and the terminal cost $E(x(T))$ is sometimes called a *Mayer term*. The combination of both, like here, is called a *Bolza objective*.

Note that any Lagrange objective term can be reformulated as a Mayer term, if we add an additional "cost state" c that has to satisfy the differential equation $\dot{c} = L(x,u)$, and then simply add $c(T)$ to the terminal Mayer cost term. Conversely, every differentiable Mayer term can be replaced by by a Lagrange term, namely by $L(x,u) = \nabla E(x)^\top f(x,u)$, as the cost integral then satisfies the equality $\int_0^\top L(x,u)dt = \int_0^\top \frac{\mathrm{d}E}{\mathrm{d}t}dt = E(x(T)) - E(x_0)$. These two equivalences entail that formulating a problem involving only a Lagrange term or only a Mayer term present no loss of generality. However, in this script we will use the full Bolza objective.

9.2 Problem reformulation

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

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

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

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

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

with pseudo time $\tau \in [0, 1]$ yielding the dynamics

$$\dot{\tilde{x}} \equiv \frac{\mathrm{d}}{\mathrm{d}\tau}\tilde{x} = \tilde{f}(\tilde{x}, u)$$

and where T is treated as a parameter, i.e. the initial condition $T(0)$ for the "state" T is free and T satisfies $\dot{T} = 0$.

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

9.3 Multi-stage Problems

A special class of continuous-time optimal control problems are multi-stage Problems, where the problem formulation can "switch" in the course of the horizon $[0, T]$. Such problems occur when e.g. the system dynamics, the cost function or the constraints change discontinuously at some time instant. The time instant at which the switching occurs can be fixed, free or even event-dependent (i.e. occurring when the system states fulfil a specific condition). Classical examples of multiple-stage optimal control problems stem from contact problem such as e.g. walking robots, shocks (e.g. a bouncing ball), the landing pattern of airliners (where the configuration of the plane has to be adjusted according to prescribed rules, yielding event-based changes in the dynamics), or industrial robots picking up and releasing objects. However, we will assume hereafter that the *ordering* in which the changes occur is prescribed and independent of the system evolution. This excludes e.g. handling problems such as gear-shifting in vehicles, where the order in which the gears are shifted is not prescribed but depends on the vehicle trajectory.

A fairly simple way of framing a multi-stage problem mathematically is to consider each *stage* of the problem as an optimal control problem of its own,

and to link the different stages by matching the terminal state of a state to the initial condition of the following stage. An N-stage multi-stage problem can then be formulated as:

$$
\begin{aligned}
\underset{x(\cdot),\, u(\cdot),\, T_{1,\ldots N}}{\text{minimize}} \quad & \sum_{k=0}^{N-1} \int_{T_k}^{T_{k+1}} L_k(x_k(t), u_k(t))dt + E_k\left(x_k(T_{k+1})\right) \\
\text{subject to} \quad & x_0(T_0) - \bar{x}_0 = 0, && \text{(initial value)}, \\
& x_k(T_k) - x_{k-1}(T_k) = 0, && \text{(state continuity)}, \\
& \dot{x}_k(t) - f_k(x_k(t), u_k(t)) = 0, \quad t \in [T_k, T_{k+1}], && \text{(ODE model)}, \\
& h_k(x_k(t), u_k(t)) \leq 0, \quad t \in [T_k, T_{k+1}], && \text{(path constraints)}, \\
& r_k\left(x_k(T_k)\right) \leq 0, && \text{(terminal const.)}, \\
& T_k - T_{k+1} \leq 0, && \text{(time ordering)}
\end{aligned}
$$

where $\bar{x}_0$ are the assigned initial conditions for the multi-stage problem, and the variables $T_{0,\ldots,N}$ are the switching times between stages. Clearly, if they are prescribed, they should then be excluded from the variables of the optimal control problem. Each stage can then be treated as a separate free end time problem, apart from the constraints $x_k(T_k) - x_{k-1}(T_k)$ linking the state trajectories between stages. Many variations of the above formulation are possible and useful to tackle various kinds of multi-stage problems.

9.4 Overview of Numerical Approaches

Generally speaking, there are three basic families of approaches to address continuous-time optimal control problems, (a) state-space, (b) indirect, and (c) direct approaches, cf. the top row of Fig. 9.2. We follow here the outline given in [31].

State-space approaches use the principle of optimality that states that each subarc of an optimal trajectory must be optimal. While this was the basis of dynamic programming in discrete time, in the continuous time case this leads to the so-called *Hamilton-Jacobi-Bellman (HJB) equation*, a partial differential equation (PDE) in the state space. Methods to numerically compute solution approximations exist, but the approach severely suffers from Bellman's "curse of dimensionality" and is restricted to small state dimensions. This approach is briefly sketched in Chapter 11.

Indirect Methods use the necessary conditions of optimality of the infinite dimensional problem to derive a boundary value problem (BVP) in ordinary

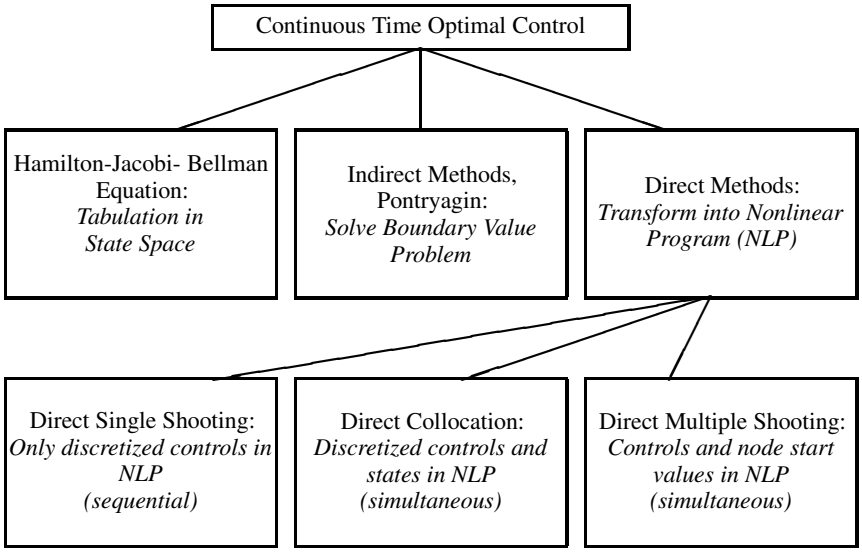


Figure 9.2 The optimal control family tree.

differential equations (ODE). This BVP must numerically be solved, and the approach is often sketched as "first optimize, then discretize", as the conditions of optimality are first written in continuous time for the given problem, and then discretized in one way or another in order for computing a numerical solution. The class of indirect methods encompasses also the well known calculus of variations and the Euler-Lagrange differential equations, and the so-called *Pontryagin Maximum Principle*. The numerical solution of the BVP is performed by shooting techniques or by collocation. The two major drawbacks are that the underlying differential equations are often difficult to solve due to strong nonlinearity and instability, and that changes in the control structure, i.e. the sequence of arcs where different constraints are active, are difficult to handle: they usually require a completely new problem setup. Moreover, on so called singular arcs, higher index differential-algebraic equations (DAE) arise which necessitate specialized solution techniques. This approach is briefly sketched in Chapter 12.

Direct methods transform the original infinite-dimensional optimal control problem into a finite-dimensional nonlinear programming problem (NLP) which is then solved by structure-exploiting numerical optimization methods. Roughly speaking, direct methods transform (typically via numerical methods) the continuous-

time dynamic system into a discrete-time system and then proceed as described in the first two parts of this script. The approach is therefore often sketched as "first discretize, then optimize", as the problem is first converted into a discrete one, on which optimization techniques are then deployed. One of the most important advantages of direct methods over indirect ones is that they can easily treat all sorts of constraints, such as e.g. the inequality path constraints in the formulation above. This ease of treatment stems from the fact that the activation and de-activation of the inequality constraints, i.e. structural changes in active constraints, occurring during the optimization procedure are treated by well-developed NLP methods that can efficiently deal with such active set changes. All direct methods are based on one form or another of finite-dimensional parameterization of the control trajectory, but differ significantly in the way the state trajectory is handled, cf. the bottom row of Fig. 9.2. For solution of constrained optimal control problems in real world applications, direct methods are nowadays by far the most widespread and successfully used techniques, and are therefore the focus of this script. Brief descriptions of three of the direct methods – single shooting, multiple shooting, and collocation – and some algorithmic details are given in Chapter 13, while we point out that the first two parts of the script covering finite dimensional optimization and discrete time dynamic systems have already covered most of the algorithmic ideas relevant for direct approaches to optimal control.

10
Numerical Simulation

Deploying optimal control on problems involving non-trivial continuous-time dynamics hinges on having efficient and accurate numerical simulations tools, which allow for building discretizations of these continuous dynamics. This chapter provides a brief but crucial exploration of these tools.

The existence of a solution to an Ordinary Differential Equation (ODE) with defined initial conditions, also called Initial-Value Problem (IVP), is guaranteed under continuity of f with respect to to x and t according to a theorem from 1886 due to Giuseppe Peano. But existence alone is of limited interest as the solutions might be non-unique. For example, the scalar ODE $\dot{x}(t) = \sqrt{|x(t)|}$ with $x(0) = 0$ admits as solution:

$$x(t) = \begin{cases} 0 & \text{for} \quad t < t_0, \\ \frac{1}{4}(t - t_0)^2 & \text{for} \quad t \geq t_0, \end{cases}$$

for any $t_0 \geq 0$, such that its solution is not unique. This ODE is continuous at the origin, but its slope approaches infinity, which causes the non-uniqueness. More important than the existence of the ODE solution is therefore its (local) uniqueness discussed in the following theorem by Charles Émile Picard (1890) and Ernst Leonard Lindelöf (1894):

Theorem 10.1 (Existence and Uniqueness of IVP)**.** *Regard the initial value problem* (1.1) *with* $x(0) = x_0$*, and assume that* f *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 *and all* $t \in [0, T]$

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

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

Note that this theorem can be extended to the case where $f(x, t)$ has finitely many discontinuities with respect to t, in which case the solutions are still

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

Following Theorem 10.1 we know that a unique ODE (or DAE) solution exists to the IVP $\dot{x} = f(x,t), x(0) = x_0$ under mild conditions, namely the Lipschitz continuity of f with respect to the state x and continuity with respect to the time t. This solution exists on the whole interval $[0,T]$ if the time $T > 0$ is chosen small enough. Note that for nonlinear continuous time systems – in contrast to discrete time systems – it is very easily possibly even with innocently-looking functions f to obtain an "explosion" in the solution of the ODE, i.e., a solution that tends to infinity in finite time. E.g. the trivial ODE $\dot{x} = x^2, x(0) = 1$ has the explicit solution $x(t) = 1/(1-t)$ tending to infinity for $t \to 1$. This simple example reveals that we cannot guarantee the existence of the solution to a differential equation on any given interval $[0,T]$ for arbitrary T, but only on sufficiently small time intervals.

10.1 Numerical Integration: Explicit One-Step Methods

Numerical integration methods are used to approximately solve a well-posed IVP that satisfies the conditions of Theorem 10.1. They come in many different variants, and can be categorized according to two major branches, on the one hand the one-step vs. the multistep methods, on the other hand the explicit vs. the implicit methods.

In the following of our exploration of numerical optimal control we will need to discuss the numerical integration over arbitrary time intervals, e.g. $[t_0, t_f]$. Let us start in this section with the explicit one-step methods, which is arguably the most basic numerical integration method. All numerical integration methods start by discretizing the state trajectories over a discretization time grid over the integration interval $[t_0, t_f]$. For the sake of simplicity, let us assume a uniform time grid, i.e. having fixed interval sizes of $\Delta t = (t_0 - t_f)/N$, where N is a positive integer. The discretization time grid is then setup as $t_k := t_0 + k\Delta t$ with $k = 0, \ldots N$, and divides the time interval $[t_0, t_f]$ into N subintervals $[t_k, t_{k+1}]$, each of length Δt. Then, the solution is approximated on the grid points t_k by discrete values s_k that shall satisfy $s_k \approx x(t_k)$, for $k = 0, \ldots, N$, where $x(t)$ is the exact solution to the IVP.

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

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

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

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

It is a first-order method, i.e. $p = 1$, and due to this low order it is very inefficient and should not be used in practice. Indeed, a few extra evaluations of f in each step can easily yield higher-order methods. E.g. the *explicit Runge-Kutta (RK) methods* due to Runge (1895) and Kutta (1901) use on each discretization interval $[t_k, t_{k+1}]$ not only one but m evaluations of f. They then hold intermediate state values $s_{k,i}$, $i = 1, \ldots, m$ *within* each interval $[t_k, t_{k+1}]$, which live on a grid of intermediate time points $t_{k,i} := t_k + c_i\,\Delta t$ with suitably chosen $c_i \in [0, 1]$. One RK step is then obtained via the following construction:

$$\begin{aligned}
s_{k,1} &:= s_k, \\
s_{k,2} &:= s_k + \Delta t\, a_{21} f\left(s_{k,1}, t_{k,1}\right), \\
s_{k,3} &:= s_k + \Delta t \left(a_{31} f\left(s_{k,1}, t_{k,1}\right) + a_{32} f\left(s_{k,2}, t_{k,2}\right)\right), \\
&\vdots \\
s_{k,i} &:= s_k + \Delta t \sum_{j=1}^{i-1} a_{ij} f(s_{k,j}, t_{k,j}), \\
&\vdots \\
s_{k,m} &:= s_k + \Delta t \sum_{j=1}^{m-1} a_{mj} f(s_{k,j}, t_{k,j}), \\
s_{k+1} &:= s_k + \Delta t \sum_{j=1}^{m} b_j f(s_{k,j}, t_{k,j}).
\end{aligned}$$

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

sion m:

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

An integration order of $m \leq 4$ is obtained from a Butcher tableau of dimension m for an adequate choice of a, b, c. It is however important to understand here that this equivalence between the dimension of the tableau and the integration order holds only for $m \leq 4$. In order to obtain an order of integration with $m \geq 5$, tableaus of dimension *larger* than 5 are needed

The explicit Euler integrator uses $m = 1$, $c_1 = 0$, $b_1 = 1$. A more efficient and widespread choice of Butcher tableau is

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

which yields a method of order $m = 4$, often simply referred to as the RK4 integration scheme.

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

Because of its simplicity, the Euler method may appear appealing in practice, however it is strongly recommended to favor higher-order methods. To get an intuitive idea of why it is so, let us assume that we want to simulate an ODE on the interval $[0, 1]$ with an accuracy of $\epsilon = 10^{-3}$ and that a first-order method gives an accuracy $\epsilon = 10\Delta t$. Then a time step of $\Delta t = 10^{-4}$ is required, i.e. $N = 10000$ steps are necessary in order to achieve the desired accuracy. If a fourth-order method gives the accuracy $\epsilon = 10(\Delta t)^4$, a time step of $\Delta t = 0.1$ is needed, i.e. only $N = 10$ steps are required for the same accuracy. Given this enormous difference, the fourfold cost per RK step required to deploy the fourth-order method is more than outweighed by the low number of steps required, such that it is actually 250 times cheaper than the first-order Euler method. In practice, RK integrators with orders up to 8 are used, but the Runge-

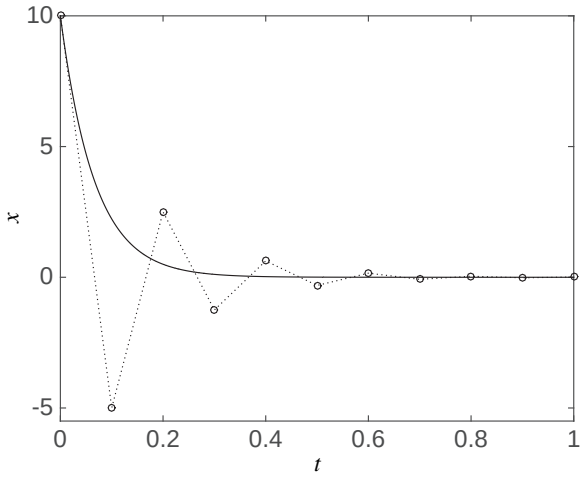


Figure 10.1 Numerical simulation of the first-order linear dynamics $\dot{x} = -15x$ using the explicit Euler method with $\Delta t = 0.1$, starting from the initial condition $x(0) = 10$. The exact solution is displayed as a plain curve, while the numerical solution is displayed using circles, connected by dotted lines. One can observe that due to the steep state derivative $\dot{x}$ in the early time of the integration, the explicit Euler scheme, which essentially computes the next state on the time grid via the tangent to the trajectory, significantly overshoots the exact solution of the ODE.

Kutta-Fehlberg method of fourth order (with fifth-order evaluation for error estimation and control) is the most popular one.

10.2 Stiff Systems and Implicit Integrators

When an explicit integrator is applied to a very stable system, its steps can overshoot the actual trajectory of the ODE solution, resulting in an inaccurate numerical integration, or even outright instability. The simple prototypical first-order system is often used to discuss these issues:

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

It takes the explicit, exact solution $x(t) = x(t_0)e^{-\lambda(t-t_0)}$. For a very large $\lambda \gg 1$ the ODE has a very fast stable mode decaying very quickly to zero. If we now use an explicit Euler method with stepsize Δt, then the trajectories of the discrete state s_k are defined by the discrete-time dynamic system:

$$s_{k+1} = s_k - \Delta t\, \lambda s_k = (1 - \Delta t\, \lambda) s_k, \quad s_0 = x(t_0),$$

which differs significantly from the exact trajectories $x(t)$, see Fig. 10.1 for an illustration. This discrete system actually becomes unstable if $\Delta t > \frac{2}{\lambda}$, which might be very small when λ is very large. Note that such a small stepsize is not necessary to obtain a high accuracy, but is only necessary to render the integrator stable.

It turns out that all explicit methods suffer from the fact that systems having very fast modes necessitate excessively short step sizes. This becomes particularly problematic if a system has both slow and fast decaying modes, i.e., if some of the eigenvalues of the Jacobian $\frac{\partial f}{\partial x}$ have a small magnitude while others are strongly negative, resulting in very quickly decaying dynamics. In such a case, one typically needs to perform fairly long simulations in order to capture the evolution of the slow dynamics, while very short steps are required in order to guarantee the stability and accuracy of the numerical integration due to the very fast modes. Such systems are called stiff systems.

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

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

One ought to observe the subtle yet crucial difference between this equation and the one used for deploying an explicit Euler integrator. While explicit Euler requires implementing an explicit rule, the equation above provides s_{k+1} implicitly. If applied to the fast, stable test system from above, for which this equation can be solved explicitly because of the linear dynamics, the implicit Euler scheme yields the discrete dynamics

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

which are stable for any $\Delta t > 0$ and always converge to zero, like the true solution of the ODE. Hence the implicit Euler scheme is always stable for this example. This idea can be easily generalized to RK methods, which then yield Butcher tableaus that are full squares and not only lower triangular, reflecting the implicit nature of the integration scheme. An implicit RK method has to

solve at each integration step k the nonlinear system of equations

$$\begin{aligned} s_{k,1} &= s_k + \Delta t \sum_{j=1}^{m} a_{1j} f\left(s_{k,j}, t_{k,j}\right) \\ &\vdots \\ s_{k,i} &= s_k + \Delta t \sum_{j=1}^{m} a_{ij} f\left(s_{k,j}, t_{k,j}\right) \\ &\vdots \\ s_{k,m} &= s_k + \Delta t \sum_{j=1}^{m} a_{mj} f\left(s_{k,j}, t_{k,j}\right) \end{aligned}$$

and then sets the next step to

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

The nonlinear system needs typically to be solved by a Newton method. Note that the system is of size $m \cdot n_x$, such that the computational complexity of performing the Newton iterations "naively" on an implicit RK method is unfortunately in general of order $O(m^3 n_x^3)$.

10.3 Orthogonal Collocation

Orthogonal collocation is a specific variant of implicit RK methods. The solution $x(t)$ on the collocation interval $t \in [t_k, t_{k+1}] \subseteq [t_0, t_f]$ is approximated by a d^{th}-order polynomial, labelled $p(t, v_k) \in \mathbb{R}^n$ in the following, where the polynomial depends linearly on the coefficients $v_k \in \mathbb{R}^{n(d+1)}$.

Interpolation polynomial The polynomials $p_k(t, v_k)$ used in orthogonal collocation methods are typically built as Lagrange polynomials. The Lagrange polynomial $p_k(t, v_k)$ for a time interval $[t_k, t_{k+1}]$ and a set of collocation times $t_{k,0} \ldots, t_{k,d}$ can be simply constructed using:

$$p_k(t, v_k) = \sum_{i=0}^{d} v_{k,i} \ell_{k,i}(t), \qquad \ell_{k,i}(t) = \prod_{j=0, j\neq i}^{d} \frac{t - t_{k,j}}{t_{k,i} - t_{k,j}} \in \mathbb{R},$$

where $v_{k,i} \in \mathbb{R}^{n_x}$ is a subset of the polynomial coefficients v_k having the size of the state vector of the system, and the collocation times are chosen as $t_{k,i} \in [t_k, t_{k+1}]$.

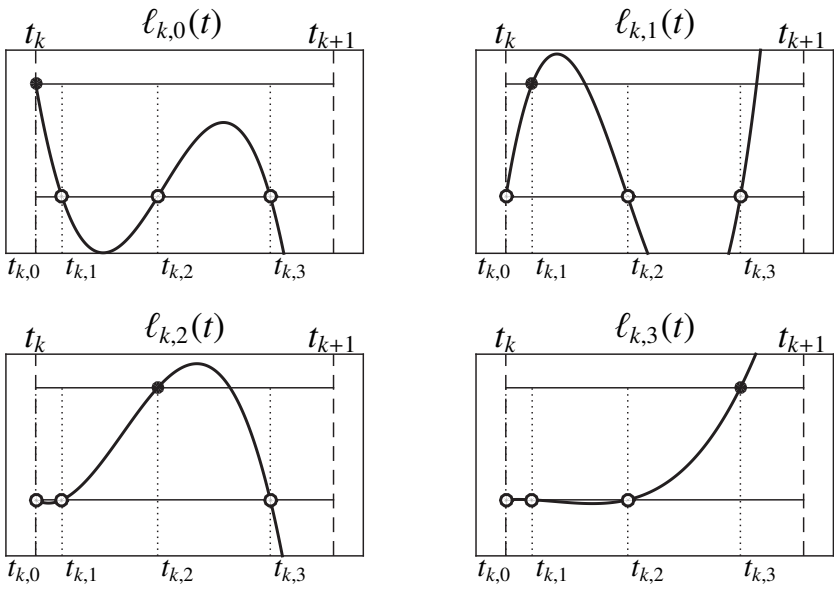


Figure 10.2 Illustration of the Lagrange polynomials $\ell_{k,i}(t)$ on the interval $[t_k,\ t_{k+1}]$. The property (10.1) is clearly visible here as each polynomial $\ell_{k,i}(t)$ take a unitary value at the collocation time $t_{k,i}$ (black dots) and a zero at all other times $t_{k,j\neq i}$ (white dots).

One can observe that the basis polynomials $\ell_{k,i}$ have by construction the property:

$$\ell_{k,i}\left(t_{k,j}\right) = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{if} \quad i \neq j \end{cases} \tag{10.1}$$

which we illustrate in Figure 10.2. They have the additional property of being orthogonal (though not orthonormal!), i.e.,

$$\int_{t_k}^{t_{k+1}} \ell_{k,i}(t)\ell_{k,j}(t)\,\mathrm{d}t = 0, \quad i \neq j. \tag{10.2}$$

Property (10.1) entails that the interpolation polynomial $p_k(t_{k,i}, v_k)$ "passes through" the interpolation points $v_{k,i}$, i.e.,

$$p_k(t_{k,i}, v_k) = v_{k,i} \tag{10.3}$$

holds for $i = 0, \ldots, d$. See Figure 10.3 for an illustration.

We detail later in this section the selection of the collocation times $t_{k,i}$. It is, however, useful to anticipate here with specifying that the first collocation

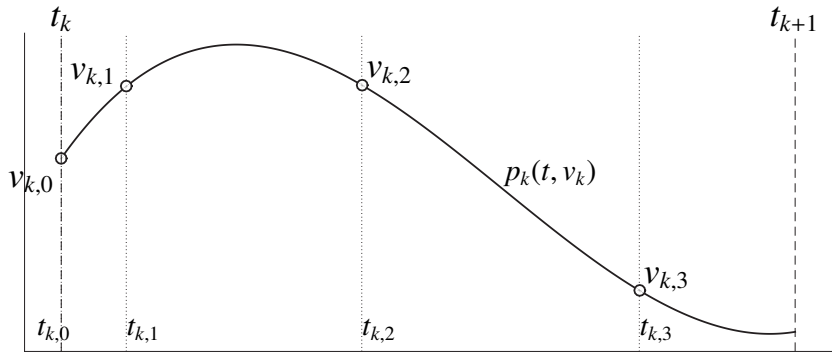


Figure 10.3 Illustration of the polynomial $p_k(t, v_k) = \sum_{i=0}^{d} v_{k,i}\ell_{k,i}(t)$ for $d = 3$ and for an arbitrary coefficient vector $v_k \in \mathbb{R}^4$, and $v_{k,i} \in \mathbb{R}$. One can observe the property (10.3), i.e. $p_k(t_{k,i}, v_k) = v_{k,i}$.

time $t_{k,0}$ is systematically chosen as $t_{k,0} = t_k$, such that $p_k(t_k, p_k) = v_{k,0}$ readily provides the initial value of the interpolation.

Collocation equations Using the polynomial $p_k(t, v_k)$ the integration over the time interval $[t_k, t_{k+1}]$ is performed via selecting adequate collocation variables $v_k \in \mathbb{R}^{n_x(d+1)}$. This selection occurs via solving a set of algebraic equations that ensure that the polynomial $p_k(t, v_k)$ is an accurate representation of the trajectories of the state. Assuming we have the initial value s_k at time t_k, the collocation equations for the simple ODE $\dot{x}(t) = f(x(t), t)$ then enforce the following $n_x(d + 1)$ conditions, see Figure 10.4:

(i) $p_k(t_k, v_k) = s_k$, i.e., the polynomials $p_k(t, v_k)$ must meet the initial condition at the beginning of the interval, i.e., at time $t_k = t_{k,0}$. It is worth observing here that since $p_k(t_k, v_k) = v_{k,0}$, satisfying the initial condition requires simply $v_{k,0} = s_k$ to hold.

(ii) $p_k(t_{k,i}, v_k)$ must satisfy the model dynamics on the remaining collocation times $t_{k,1}, \ldots, t_{k,d}$, i.e.:

$$\dot{p}_k(t_{k,i}, v_k) = f(\underbrace{p_k(t_{k,i}, v_k)}_{=v_{k,i}}, t_{k,i}) \tag{10.4}$$

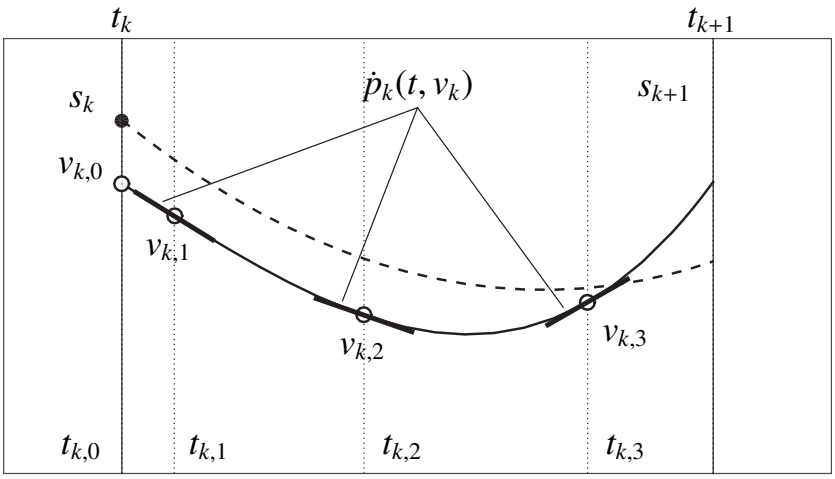


Figure 10.4 The NLP variables in the direct collocation method for $d = 3$, and for one specific time interval $[t_k,\ t_{k+1}]$. Here we illustrate the polynomial $p_k\,(t, v_k)$ (solid curve) vs. the actual state trajectories (dashed curve) when the collocation equations $c_k\,(s_k, v_k, t_{k,i}) = 0$ are not yet satisfied.

The integration of the system dynamics over a time interval $[t_k,\ t_{k+1}]$ is hence performed via solving the collocation equations:

$$c_k\,(v_k, t_{k,i}, s_k) = \begin{bmatrix} v_{k,0} - s_k \\ \dot{p}_k\,(t_{k,1}, v_k) - f(v_{k,1}, t_{k,i}) \\ \vdots \\ \dot{p}_k\,(t_{k,d}, v_k) - f(v_{k,d}, t_{k,i}) \end{bmatrix} = 0. \tag{10.5}$$

The end state of the simulation $x\,(t_{k+1})$ is then accurately approximated by $p(t_{k+1}, v_k)$. This principle is illustrated in Figure 10.5 for a single state.

We observe here that (10.5) is a system of $n(d + 1)$ equations in the $v_k \in \mathbb{R}^{n(d+1)}$ variables. We additionally observe that

$$\dot{p}_k(t, v_k) = \sum_{i=0}^{d} v_{k,i} \dot{\ell}_{k,i}(t),$$

such that, similarly to $p_k\,(t_{k,i}, v_k)$, the time derivatives of the polynomial, i.e. $\dot{p}_k\,(t_{k,i}, v_k)$ are linear in v_k. It can then be observed that for a linear dynamic model f, (10.5) is a linear system of equations. However, in general, (10.5) does not have an explicit solution.

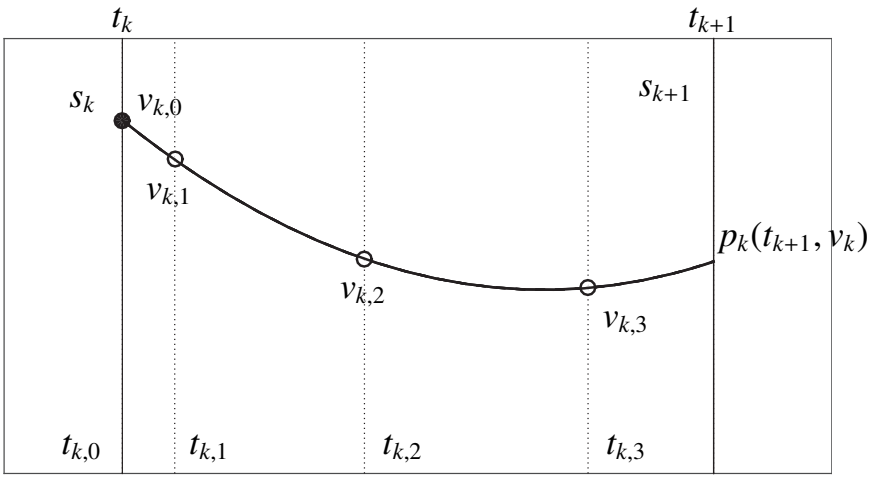


Figure 10.5 The NLP variables in the direct collocation method for $d = 3$, and for one specific time interval $[t_k, t_{k+1}]$. Here we illustrate the polynomial $p_k(t, v_k)$ when the collocation equations $c_k(s_k, v_k, t_{k,i}) = 0$ are satisfied, such that $p_k(t, v_k)$ captures accurately the system trajectory over the time interval $[t_k, t_{k+1}]$. The end state of the simulation $x(t_{k+1})$ is then accurately approximated by $p(t_{k+1}, v_k)$.

Selection of the collocation times $t_{k,i}$ It is very important to point out here that an adequate choice of collocation points leads to very high orders of integration. We can understand this point using the principle of Gauss-quadrature. Assuming that $p(t, v_k)$ is a polynomial of order $2d$, the Gaussian quadrature formula provides for any v_k the equality:

$$\int_{t_k}^{t_{k+1}} \dot{p}(t, v_k)\,\mathrm{d}t = (t_{k+1} - t_k) \cdot \sum_{i=1}^{m} \omega_i \cdot \dot{p}(t_{k,i}, v_k), \tag{10.6}$$

for an adequate choice of weights ω_i and of collocation points $t_{k,i}$.

The adequate choice of collocation time, i.e. the one that yields the Gauss quadrature formula (10.6) for polynomials of degree $2d$, is obtained by choosing the collocation points $t_{k,i}$ as the zeros of orthogonal Legendre polynomials on the corresponding interval $[t_k, t_{k+1}]$. This choice of collocation times is called *Gauss-Legendre collocation*. For the specific time interval $[0,\ 1]$, the collocation points $t_{k,i}$ for $i = 1, \ldots, 4$ are provided in Table 10.1 for $d = 1, \ldots, 4$. For an arbitrary time interval $[t_k,\ t_{k+1}]$, the adequate collocation times $t_{k,i}$ can be computed by identifying $\xi_i = (t_{k,i} - t_k)/(t_{k+1} - t_k)$ with the time points ξ_i in

Table 10.1 *Collocation times* $t_{k,1}, \ldots t_{k,d}$ *for* $d = 1, ..., 4$ *on the interval* $[0, 1]$.

d	Gauss-Legendre collocation ξ				Gauss-Radau collocation ξ			
1	0.50000							1
2	0.21132	0.78867			0.33333			1
3	0.11270	0.50000	0.88729		0.15505	0.64494		1
4	0.06943	0.33000	0.66999	0.93056	0.08858	0.40946	0.78765	1

Table 10.1, i.e. by computing the collocation times $t_{k,i}$ as

$$t_{k,i} = t_k + (t_{k+1} - t_k)\,\xi_i \in [t_k,\, t_{k+1}]\,. \tag{10.7}$$

An extra collocation point $t_{k,0}$ is systematically added to the collection $t_{k,1}, \ldots t_{k,d}$ in order to be able to enforce the initial value constraints $v_{k,0} = s_k$. Note that the collocation points $t_{k,1}, \ldots t_{k,d}$ are all in the interior of the collocation interval and symmetric around the midpoint.

Let us then observe that the exact state trajectory $x(t)$ of the ODE $\dot{x}(t) = f(x(t), t)$ satisfies the equation

$$x(t_{k+1}) = x(t_k) + \int_{t_k}^{t_{k+1}} f(x(t), t)\,\mathrm{d}t. \tag{10.8}$$

Let us then assume that the trajectory $x(t)$ can be exactly captured on the time interval $[t_k,\, t_{k+1}]$ by the polynomial $p\,(t, v_k)$ of degree $2d$. Using (10.5), (10.6) and (10.8), we obtain the identity

$$p(t_{k+1}, v_k) = p(t_k, v_k) + (t_{k+1} - t_k)\,\cdot \sum_{i=1}^{d} \omega_i f(p(t_{k,i}, v_k), t_{k,i})$$

such that $s_{k+1} = p(t_{k+1}, v_k) = x\,(t_{k+1})$ holds.

Using Gauss-Legendre collocation times, the integration is then exact if f is a polynomial of up to degree $2d - 1$. This implies that the collocation step $s_{k+1} - s_k$ is exact if the exact solution has a derivative $\dot{x}(t)$ that is a polynomial of order $2d - 1$, i.e., if the solution $x(t)$ is a polynomial of order $2d$. Gauss-Legendre collocation is the collocation method with the highest possible order for a given d, i.e. $2d$.

An alternative collocation setup sacrifices one order and chooses a set of collocation points that includes the end point of the interval. It is called *Gauss-Radau collocation* and has a desirable property for stiff systems called *stiff decay*. The relative collocation point locations $\xi_i = (t_{k,i} - t_k)/(t_{k+1} - t_k)$ for Gauss-Legendre and Gauss-Radau collocation are given in Table 10.1, see [12].

It is worth stressing here that the very high order of collocation methods hinges on using the collocation times prescribed by Table 10.1, ideally with

minimum rounding error. Indeed, numerical experiments show that using approximate collocation times can yield a fast loss of accuracy of collocation-based integrators.

10.3.1 Linear Multistep Methods and Backward Differentiation Formulae

A different approach to obtain a high order are the linear multistep methods that use a linear combination of the past M steps $s_{k-M+1}, \ldots, s_k$ and their function values $f(s_{k-M+1}), \ldots, f(s_k)$ in order to obtain the next state, s_{k+1}. They are implicit, if they also use the function value $f(s_{k+1})$. A major issue with linear multistep methods is stability, and their analysis needs to regard a dynamic system with an enlarged state space consisting of all M past values.

A very popular and successful class of implicit multistep methods are called the *backward differentiation formulae (BDF)* methods. In each step, an implicit equation is formulated in the variable s_{k+1} by constructing the interpolation polynomial $p_k(t, s_{k+1})$ of order M that interpolates the known values $s_{k-M+1}, \ldots, s_k$ as well the unknown s_{k+1}, and then equates the derivative of this polynomial with the function value, i.e., solves the nonlinear equation

$$\frac{\mathrm{d}}{\mathrm{d}t} p_k(t_{k+1}, s_{k+1}) = f(s_{k+1}, t_{k+1})$$

in the unknown s_{k+1}. Note that the fact that only a nonlinear system of size n_x needs to be solved in each step of the BDF method is in contrast to m-stage implicit RK methods, which need to solve a system of size $m \cdot n_x$. Still, the convergence of the BDF method is of order M. It is, however, not possible to construct stable BDF methods of arbitrary orders, as their stability regions shrink, i.e., they become unstable even for stable systems and very short step lengths Δt. The highest possible order for a BDF method is $M = 6$, while the BDF method with $M = 7$ is not stable anymore. If e.g. it is applied to the test equation $\dot{x} = -\lambda x$ with $\lambda > 0$ it diverges even if an arbitrarily small step size Δt is used. It is interesting to compare linear multistep methods with the sequence of Fibonacci numbers that also use a linear combination of the last two numbers in order to compute the next one (i.e., $M = 2$). While the Fibonacci numbers do not solve a differential equation, the analysis of their growth is equivalent to the analysis of the stability of linear multistep methods. For more details, the reader is referred to, e.g., [20, 21, 3].

10.3.2 Solution Map and Sensitivities

In the context of optimal control, derivatives of the simulation of the system dynamics with respect to the initial conditions and control inputs need to be provided to the numerical algorithms.

In order to discuss the issue of differentiating the solution of an ODE with respect to its initial conditions and possibly other parameters, which in the context of dynamic systems are often called *sensitivities*, let us now regard an ODE with some parameters $p \in \mathbb{R}^{n_p}$ that enter the function f and assume that f satisfies the assumptions of Theorem 10.1. We regard some values $\bar{x}_0$, $\bar{p}$, T such that the ODE

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

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

Let us illustrate this sensitivity question for linear, continuous-time systems

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

hence with $f(x, p, t) = Ax + Bp$, the map $x(t, p, x_0)$ is explicitly given as

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

where $e^{\cdot}$ is the matrix exponential function. Similarly to function f, this map is infinitely many times differentiable (and even well defined for all times t, as linear systems can be unstable but cannot "explode" in finite time). In this simple case, having an explicit solution map, the sensitivities of the solution can be explicitly computed, and read as:

$$\frac{\partial}{\partial x_0}x(t, p, x_0) = e^{At}, \qquad \frac{\partial}{\partial p}x(t, p, x_0) = \int_0^t e^{A(t-\tau)}B\mathrm{d}\tau$$

In the general nonlinear case, the map $x(t, p, x_0)$ can only be generated by a numerical simulation algorithm. The computation of derivatives of this numerically generated map is a delicate issue that we discuss in detail hereafter. To

mention already the main difficulty, note that most practical numerical integration algorithms are adaptive, i.e. they might choose to do different numbers of integration steps for different IVPs. This renders the numerical approximation of the map $x(t, p, x_0)$ typically non-differentiable as an infinitesimal perturbation in p or x_0 can trigger a discrete change in the number of integration steps. This feature makes multiple calls of a black-box integrator and application of finite differences problematic, as it often results in significantly wrong derivative approximations.

10.4 Sensitivity Computation for Integration Methods

Numerical optimal control methods require one to compute the derivatives of the result of an ODE or DAE integration algorithm, on a given time interval. Let us for notational simplicity regard just the autonomous ODE case $\dot{x} = f(x)$ on a time interval $[0, T]$. The case of constant control or other parameters on which this ODE depends as well as time dependency can conceptually be covered by state augmentation, i.e. one can e.g. rewrite the ODE $\dot{x} = f(x, p)$, $x(0) = x_0$ as:

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} f(x, v) \\ 0 \end{bmatrix}, \qquad \begin{bmatrix} x \\ v \end{bmatrix}(0) = \begin{bmatrix} x_0 \\ p \end{bmatrix}$$

Thus, we regard an initial condition x_0 and the evolution of the ODE

$$\dot{x} = f(x), \quad t \in [0, T], \quad x(0) = x_0.$$

giving a solution $x(t, x_0)$, $t \in [0, T]$. We are interested here in the *sensitivity matrix*

$$G(t) = \frac{\partial x(t, x_0)}{\partial x_0}, \quad t \in [0, T],$$

and in particular its terminal value. This matrix $G(T) \in \mathbb{R}^{n_x \times n_x}$ can be computed in many different ways, five of which we briefly sketch here.

(i) External Numerical Differentiation (END)
(ii) Solution of the Variational Differential Equations
(iii) Algorithmic Differentiation (AD) of the Integrator
(iv) Internal Algorithmic Differentiation within the Integrator
(v) Internal Numerical Differentiation (IND)

In all five methods we assume that the integrator to be differentiated is a state-of-the-art integrator with inbuilt error control and adaptive stepsize selection.

External Numerical Differentiation (END) The first approach, *External Numerical Differentiation (END)*, just treats the integrator as a black-box function and uses finite differences. We perturb x_0 by some quantity $\epsilon > 0$ in the direction of the unit vectors e_i and call the integrator several times in order to compute directional derivatives by finite differences:

$$G(T)e_i \approx \frac{x(T, x_0 + \epsilon e_i) - x(T, x_0)}{\epsilon}. \tag{10.9}$$

The cost of this approach to compute $G(T)$ is $(n_x + 1)$ times the cost of a forward simulation. The approach is very easy to implement, but suffers from one serious problem: due to integrator adaptivity, each call might have a different discretization grid. This error control of each trajectory does not only create an overhead, but worse, it might result in discontinuous perturbations even for small ϵ, when a perturbation $x_0 + \epsilon e_i$ triggers a discrete adaptation of integrator (e.g. a change in the number of steps). It is important to note that due to adaptivity, the output $x(T, x_0)$ is not a differentiable function in x_0, but only guaranteed to be close to the true solution within the integrator accuracy TOL. Thus, we need to use, as a rule of thumb, $\epsilon = \sqrt{\text{TOL}}$ in order to make large-enough perturbations. As finite differences always mean that we loose half the digits of accuracy, we might easily end e.g. with a derivative that has only two valid digits.

Variational Approach A completely different approach is to formulate and solve the *variational differential equations* along with the nominal trajectory. In this context, we define a matrix $G(t)$ with the property:

$$G(t) = \frac{\partial x(t, x_0)}{\partial x_0}$$

where $x(t, x_0)$ is the solution map of the ODE. Clearly, since $x(0, x_0) = x_0$ holds, $G(0) = \mathbb{I}$. Moreover, we observe that:

$$\dot{G}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial}{\partial x_0} x(t, x_0)\right) = \frac{\partial \dot{x}(t, x_0)}{\partial x_0} = \frac{\partial}{\partial x_0} f\left(x(t, x_0)\right) = \frac{\partial f}{\partial x}\left(x(t, x_0)\right) \underbrace{\frac{\partial x(t, x_0)}{\partial x_0}}_{G(t)}$$

This entails that we can obtain the sensitivities of the solution of the ODE by solving, together with $\dot{x} = f(x)$, the additional matrix differential equation

$$\frac{\mathrm{d}}{\mathrm{d}t} G(t) = \frac{\partial f}{\partial x}(x(t)) G(t), \; t \in [0, T], \quad G(0) = \mathbb{I}.$$

This approach is much more accurate than the previous one, at a similar computational cost. However, analytic expressions for $\frac{\partial f}{\partial x}$ are required. Also, it is

interesting to note that the computed sensitivity $G(T)$ might not be identical to the derivative of the (discretized) integrator result $x(T, x_0)$.

External Algorithmic Differentiation (EAD) The last disadvantage mentioned above is avoided in the third approach, *Algorithmic Differentiation (AD) of the Integrator* or *External AD*. The approach requires that the time steps and the order of the integrator are fixed at the current nominal trajectory. An AD tool is then deployed on the whole integrator code to generate the sensitivities. Up to machine precision, AD provides derivative that are identical to the ones of the numerical solution $x(T, x_0)$ for a given fixed discretization grid. In a practical implementation, the integrator and right hand side function $f(x)$ need to be in the same or in compatible computer languages that are treated by the corresponding AD tool (e.g. C++ when using ADOL-C).

If *External AD* is deployed on an implicit integrator, it should be noted that the underlying Newton iterations will be differentiated, which might create considerable and avoidable overhead compared to the variational differential equation approach.

Internal Algorithmic Differentiation (IAD) A fourth approach, labelled *Internal Algorithmic Differentiation (AD) of the Integrator* is a subtle variation of *External AD*. Here, AD is applied independently to each step of the integrator in a custom implementation of the integration algorithm, and care is taken that only the components of the algorithm that need to be differentiated are differentiated. The approach can be easily illustrated for an Euler scheme (in this specific case it internal AD is identical to both the variational differential equation and external AD). If the grid is given by $\{t_k\}_{k=0}^N$ and the Euler iterates as

$$x_{k+1} = x_k + (t_{k+1} - t_k) f(x_k), \quad k = 0, \ldots, N-1, \quad x_0 = s,$$

then this approach generates matrices

$$G_{k+1} = G_k + (t_{k+1} - t_k) \frac{\partial f}{\partial x}(x_k) G_k, \quad k = 0, \ldots, N-1, \quad G_0 = \mathbb{I}.$$

Internal AD can arguably be construed as a discrete variational equation deployed on the integration algorithm.

This approach is usually the most computationally efficient of the exact differentiation approaches but requires a custom implementation of an ODE/DAE solver that is explicitly designed for the generation of sensitivities. Note that as in the previous two approaches, dealing with black-box right hand side functions $f(x)$ would require that the matrix $\frac{\partial f}{\partial x}(x_k)$ must also be computed by finite differences at every integration step.

For the reader interested in implementing a rudimentary yet efficient integration scheme with sensitivity, we provide next the details of an efficient RK4 scheme with internal AD.

Algorithm 10.2. **Input:** $x_0,\ p$
Output: $f_{\mathrm{RK4}}(x_0, p)$, $\frac{\partial}{\partial x_0} f_{\mathrm{RK4}}(x_0, p)$, $\frac{\partial}{\partial p} f_{\mathrm{RK4}}(x_0, p)$
Set $x = x_0, \quad A = I, \quad B = 0$
for $n = 0 : N$ **do**

$k \leftarrow F(x, p), \qquad \Delta x \leftarrow k$
$k_x \leftarrow \nabla_x F(x, p), \quad \Delta x_x \leftarrow k_x$
$k_p \leftarrow \nabla_u F(x, p), \quad \Delta x_p \leftarrow k_p$

$k \leftarrow F(x + \frac{\Delta t}{2N} k, p), \qquad \Delta x \leftarrow \Delta x + 2k$
$k_x \leftarrow \nabla_x F(x, p)\left(I + \frac{\Delta t}{2N} k_x\right), \quad \Delta x_x \leftarrow \Delta x_x + 2k_x$
$k_p \leftarrow \nabla_u F(x, p) + \nabla_x F(x, p) \frac{\Delta t}{2N} k_p, \quad \Delta x_p \leftarrow \Delta x_p + 2k_p$

$k \leftarrow F(x + \frac{\Delta t}{2N} k, p), \qquad \Delta x \leftarrow \Delta x + 2k$
$k_x \leftarrow \nabla_x F(x, p)\left(I + \frac{\Delta t}{2N} k_x\right), \quad \Delta x_x \leftarrow \Delta x_x + 2k_x$
$k_p \leftarrow \nabla_u F(x, p) + \nabla_x F(x, p) \frac{\Delta t}{2N} k_p, \quad \Delta x_p \leftarrow \Delta x_p + 2k_p$

$k \leftarrow F(x + \frac{\Delta t}{N} k, p), \qquad \Delta x \leftarrow \Delta x + k$
$k_x \leftarrow \nabla_x F(x, p)\left(I + \frac{\Delta t}{N} k_x\right), \quad \Delta x_x \leftarrow \Delta x_x + k_x$
$k_p \leftarrow \nabla_u \nabla_u F(x, p) + \nabla_x F(x, p) \frac{\Delta t}{N} k_p, \quad \Delta x_p \leftarrow \Delta x_p + k_p$

$x \leftarrow x + \frac{\Delta t}{6N} \Delta x$
$A \leftarrow \left(I + \frac{\Delta t}{6N} \Delta x_x\right) A$
$B \leftarrow \left(I + \frac{\Delta t}{6N} \Delta x_x\right) B + \frac{\Delta t}{6N} \Delta x_p$

end for
Set $f_{\mathrm{RK4}}(x_0, p) = x, \quad \frac{\partial}{\partial x_0} f_{\mathrm{RK4}}(x_0, p) = A, \quad \frac{\partial}{\partial p} f_{\mathrm{RK4}}(x_0, p) = B$

Such an algorithm can e.g. be easily deployed in plain C and typically delivers high computational performances for a large class of ODEs.

This last idea can be generalized to the concept of *Internal Numerical Differentiation (IND)* [18]. At first sight it is similar to END, but needs a custom implementation and differs in several respects. First, all trajectories are computed simultaneously, only the nominal trajectory is adaptive, while the perturbed trajectories use the nominal, frozen grid. In implicit methods, also matrix factorizations etc. will be frozen. At the end of the interval, we use the

finite difference formula (10.9) but with a much smaller perturbation, namely $\epsilon = \sqrt{\text{PREC}}$ where PREC is the machine precision, typically 10^{-16}. The derivatives will have the accuracy $\sqrt{\text{PREC}}$, i.e. usually 10^{-8}, which is much higher than for END.

Again, we illustrate IND at hand of the explicit Euler integration scheme, where each perturbed trajectory with index $i = 1, \ldots, n_x$ just satisfies

$$x_{k+1}^i = x_k^i + (t_{k+1} - t_k) f(x_k^i), \quad k = 0, \ldots, N-1, \quad x_0^i = s + \epsilon e_i.$$

Note that due to the fact that adaptivity and possible matrix factorizations are switched off for the perturbed trajectories, IND is not only more accurate, but also cheaper than END.

10.4.1 Differentiation of Implicit Integrators

Internal Numerical Differentiation adequately deployed on implicit integrators is usually fairly simple and inexpensive as the Newton scheme used to solve the implicit equations underlying the scheme contain already most of the information required for computing the sensitivities. Implicit integrators are based on solving a set of equations defining on the time interval $[t_k, t_{k+1}]$ the end-state of the integrator x_{k+1} implicitly from the initial condition x_k and possible parameters p. One can write an implicit integrator in the general form

$$x(t_{k+1}, x_k) = \phi(w), \quad \text{with} \quad g(w, x_k, p) = 0,$$

where g captures implicitly the continuous dynamics of the system via an ad-hoc implicit integration scheme, w is a set of intermediate variables supporting the implicit integration and ϕ a function delivering the end state. The integration is the performed by running the following Newton iteration:

Algorithm 10.3. **Input:** x_k, p, w
Output: x_{k+1}, w
while $\|g(w, x_k, p)\| > \text{tol}$ **do**

$$w \leftarrow w - \left[\frac{\partial}{\partial w} g(w, x_k, p)\right]^{-1} g(w, x_k, p)$$

end while
Set $x_{k+1} = \phi(w)$

The sensitivities can then be obtained using the implicit function theorem,

by evaluating

$$\frac{\partial x_{k+1}}{\partial x_k} = -\frac{\partial \phi(w)}{\partial w}\left[\frac{\partial}{\partial w} g(w, x_k, p)\right]^{-1} \frac{\partial}{\partial x_k} g(w, x_k, p) \tag{10.10a}$$

$$\frac{\partial x_{k+1}}{\partial p} = -\frac{\partial \phi(w)}{\partial w}\left[\frac{\partial}{\partial w} g(w, x_k, p)\right]^{-1} \frac{\partial}{\partial p} g(w, x_k, p) \tag{10.10b}$$

at the output w of Algorithm 10.3. It is important here to observe that the computation of the sensitivities (10.10) can re-use the latest factorization of $\frac{\partial}{\partial w} g(w, x_k, p)$ formed in Algorithm 10.3, so that they require only the evaluation of $\frac{\partial}{\partial x_k} g(w, x_k, p)$, $\frac{\partial}{\partial p} g(w, x_k, p)$, $\frac{\partial \phi(w)}{\partial w}$ and the matrix multiplications required to evaluate (10.10). They can therefore be usually formed at a fairly low computational complexity.

10.5 Second-order sensitivities

We have detailed so far efficient methods to compute the first-order sensitivities of integrators, both in the explicit and implicit case. For a given simulation method $x(t, p, x_0)$, these computations aim at delivering the derivatives

$$\frac{\partial}{\partial x_0} x(t, p, x_0) \quad \text{and} \quad \frac{\partial}{\partial p} x(t, p, x_0) \tag{10.11}$$

which are essential in the context of numerical optimal control. However, numerical methods tackling the NLP underlying optimal control problems can also make use of the second-order information on the NLP, in the form of the Hessian of the Lagrange function. While approximations can be used to compute the exact Hessian, see Chapter **??**, providing the exact Hessian of the Lagrange function to the NLP solver can lead to a significantly better convergence of the NLP solver than when using Hessian approximations. As we will see later in further details, computing the exact Hessian of the NLP resulting from the discretization of a continuous optimal control problem will require the computation of the second-order derivatives of the simulation in some specific directions, i.e. we will be interested in computing

$$\frac{\partial^2}{\partial^2 x_0}\left(\mu^\top x(t, p, x_0)\right), \quad \frac{\partial^2}{\partial^2 p}\left(\mu^\top x(t, p, x_0)\right), \quad \frac{\partial^2}{\partial x_0 \partial p}\left(\mu^\top x(t, p, x_0)\right) \tag{10.12}$$

for some specific vector $\lambda \in \mathbb{R}^{n_x}$. We will present in the following some methods to approach the problem of computing (10.12) efficiently.

10.5.1 Second-order sensitivities for explicit integrators

In order to build a generic discussion here, let us describe explicit integrators as a generic recursion:

Algorithm 10.4.
Input: $x_0,\ p$
$s_0 = x_0$
for $i = 1 : N$ **do**
$s_i = \xi(s_{i-1}, p)$
end for
Return $x(t, x_0, p) = s_N$

We observe that this generic algorithm can represent any explicit integrator taking N steps in integrating the dynamics, depending on the choice of function ξ. E.g. a basic explicit Euler integrators would use the function $\xi(s_i, p) = s_i + \frac{t}{N} f(s_i, p)$.

Adjoint-mode sensitivities Let us then first consider the problem of computing the sensitivities not of the simulation $x(t, x_0, p)$ but of some *scalar function* of the simulation, i.e. we are interested in computing:

$$\frac{\partial}{\partial x_0}\zeta(x(t, p, x_0)) \quad \text{and} \quad \frac{\partial}{\partial p}\zeta(x(t, p, x_0)) \tag{10.13}$$

where ζ is a scalar function. Clearly, these sensitivities can be computed via a chain-rule, using the classical sensitivities (10.11). However, for this specific problem, the adjoint-mode offers a more straightforward approach. Let us define:

$$\lambda_i^\top = \frac{\partial \zeta(s_N)}{\partial s_i} \tag{10.14}$$

the sensitivity of the output s_N of Algorithm (10.4) to some of it intermediate state s_i. We then observe that:

$$\lambda_N^\top = \frac{\partial \zeta(s_N)}{\partial s_N} \tag{10.15a}$$

$$\lambda_{i-1}^\top = \frac{\partial \zeta(s_N)}{\partial s_{i-1}} = \underbrace{\frac{\partial \zeta(s_N)}{\partial s_i}}_{=\lambda_i^\top} \frac{\partial s_i}{\partial s_{i-1}} = \lambda_i^\top \frac{\partial \xi(s_{i-1}, p)}{\partial s_{i-1}} \tag{10.15b}$$

One can observe that (10.15) defines a *backward recursion* that can be computed provided that the *forward* state trajectories $s_{0,\dots,N}$ are available, e.g. via a deployment of Algorithm 10.4 with a storage of its intermediate values. The adjoint-mode sensitivity computation then reads as:

Algorithm 10.5.

Input: $s_{0,\dots,N}$, p
$\lambda \leftarrow \frac{\partial \zeta(s_N)}{\partial s_N}^\top$
for $i = N : 1$ **do**
$\lambda \leftarrow \frac{\partial \xi(s_{i-1},p)}{\partial s_{i-1}}^\top \lambda$
end for
Return $\frac{\partial}{\partial x_0}\zeta\left(x(t,p,x_0)\right) = \lambda^\top$

If the sensitivities are needed only in the form (10.13), as opposed to the entire sensitivities (10.11), then it is typically more efficient to deploy Algorithm 10.4 (*forward sweep*) and then Algorithm 10.5 (*Adjoint mode sensitivity*) in order to compute (10.13) rather than to compute the entire state sensitivities (10.11) to finally compute (10.13).

Forward over adjoints A classical approach to compute the second-order sensitivity of a scalar function of the form (10.12) is to perform a forward sensitivity computation over the adjoint-mode Algorithm 10.5, in order to compute the sensitivities $\frac{\partial \lambda_0}{\partial x_0}$. To that end, we label $H_i = \frac{\partial \lambda_i}{\partial x_0} \in \mathbb{R}^{n_x \times n_x}$, and we first observe that

$$H_N = \frac{\partial \lambda_N}{\partial x_0} = \frac{\partial \lambda_N}{\partial s_N}\frac{\partial s_N}{\partial x_0} = \frac{\partial^2 \zeta(s_N)}{\partial^2 s_N}\frac{\partial s_N}{\partial x_0}, \tag{10.16}$$

where $\frac{\partial s_N}{\partial x_0}$ can be obtained via a standard forward sensitivity computation of the explicit integrator. In the special case we consider here, where the scalar function ζ is linear in s_N, we observe that $H_N = 0$. We then observe that

$$H_{i-1} = \frac{\partial \lambda_{i-1}}{\partial x_0} = \frac{\partial \lambda_{i-1}}{\partial \lambda_i}\frac{\partial \lambda_i}{\partial x_0} + \frac{\partial \lambda_{i-1}}{\partial s_{i-1}}\frac{\partial s_{i-1}}{\partial x_0} \tag{10.17}$$

$$= \frac{\partial}{\partial \lambda_i}\left(\frac{\partial \xi(s_{i-1},p)}{\partial s_{i-1}}^\top \lambda_i\right) H_i + \frac{\partial}{\partial s_{i-1}}\left(\frac{\partial \xi(s_{i-1},p)}{\partial s_{i-1}}^\top \lambda_i\right)\frac{\partial s_{i-1}}{\partial x_0} \tag{10.18}$$

$$= \frac{\partial \xi(s_{i-1},p)}{\partial s_{i-1}}^\top H_i + \frac{\partial^2}{\partial s_{i-1}^2}\left(\lambda_i^\top \xi(s_{i-1},p)\right)\frac{\partial s_{i-1}}{\partial x_0}, \tag{10.19}$$

where, as before, $\frac{\partial s_{i-1}}{\partial x_0}$ can be obtained via a standard forward sensitivity computation of the explicit integrator. A prototype of an algorithm that computes the directional second-order sensitivities of an explicit integrator can then read as:

Algorithm 10.6.

Input: x_0, p and direction μ
Forward pass:
$s_0 = x_0$, $A_0 = I$

for $i = 1 : N$ **do**
$A_i = \frac{\partial \xi(s_{i-1}, p)}{\partial s_{i-1}} A_{i-1}$ *Storage needed*
$s_i = \xi(s_{i-1}, p)$ *Storage needed*
end for
Adjoint + Forward over adjoint:
$\lambda \leftarrow \mu, \quad H \leftarrow 0$
for $i = N : 1$ **do**
$H \leftarrow \frac{\partial \xi(s_{i-1}, p)}{\partial s_{i-1}}^{\top} H + \frac{\partial^2}{\partial s_{i-1}^2} \left(\lambda^{\top} \xi(s_{i-1}, p)\right) A_{i-1}$
$\lambda \leftarrow \frac{\partial \xi(s_{i-1}, p)}{\partial s_{i-1}}^{\top} \lambda$
end for
Return $x(t, x_0, p) = s_N, \quad \frac{\partial}{\partial x_0} x(t, x_0, p) = A_N$
and $\frac{\partial}{\partial x_0} \left(\mu^{\top} x(t, x_0, p)\right) = \lambda, \quad \frac{\partial^2}{\partial x_0^2} \left(\mu^{\top} x(t, x_0, p)\right) = H$

It is interesting to observe that this fairly simple algorithm generates a simulation of the dynamics with first-order sensitivities, as well as the first and second-order sensitivities of the scalar function $\mu^{\top} x(t, x_0, p)$ of the simulation $x(t, x_0, p)$ in the prescribed direction μ. However, it is important to underline here that the intermediate steps s_i and their sensitivities A_i need to be stored during the forward pass. In general, the storage requirements amounts to the storage of $Nn(n + 1)$ floats, which can be, unfortunately, fairly large.

Exercises

10.1 **Euler vs RK4:** Consider a controlled harmonic oscillator described by:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} p(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}\begin{bmatrix} p(t) \\ v(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t), \quad t \in [0, T].$$

We abbreviate this ODE as $\dot{x} = f(x, u)$ with $x = (p, v)^\top$. We choose the fixed initial value $x(0) = (10, 0)^\top$ and $T = 10$.

(a) We are interested in comparing the simulation results for $u(t) = 0$ that are obtained by two different integration schemes, namely the (explicit) Euler integrator and a Runge-Kutta integrator of 4th order. We regard in particular the value $p(10)$, and as the ODE is explicitly solvable, we know it exactly, which is useful for comparisons. What is the analytical expression for $p(10)$? Evaluate it numerically.

(b) Write a function named f using `def f(x,u)` that evaluates the right hand side of the ODE. Then, implement an explicit Euler method with $N = 50$ integrator steps, i.e. with a stepsize of $\Delta t = 10/50 = 0.2$. The central line in the Euler code reads

$$x_{k+1} = x_k + \Delta t \cdot f(x_k, u_k) \tag{10.20}$$

Plot your trajectories $\{(t_k, x_k)\}_1^{N+1}$ for $u_k = 0$.

(c) Now exchange in your Euler simulation code the line that generates the step (10.20) by the following five lines:

$$\begin{aligned} k_1 &= f(x_k, u_k) \\ k_2 &= f(x_k + \frac{1}{2}\Delta t \cdot k_1, u_k) \\ k_3 &= f(x_k + \frac{1}{2}\Delta t \cdot k_2, u_k) \\ k_4 &= f(x_k + \Delta t \cdot k_3, u_k) \\ x_{k+1} &= x_k + \Delta t \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

This is the classical Runge Kutta method of order four (RK4). Note that each integrator step is four times as expensive as an Euler step. What is the advantage of this extra effort? To get an idea, plot your trajectories $\{(t_k, x_k)\}_1^{N+1}$ for the same number N of integrator steps.

(d) Make both pieces of your integrating code reusable by creating functions named `euler` and `rk4` out them. Both should have arguments `x0`, `T` and `N` and return the state at the end point. Test your implementation by comparing with the plots.

(e) To make the comparison of Euler and RK4 quantitative, regard the different approximations of $p(10)$ that you obtain for different step-sizes, e.g. $\Delta t = 10^{-k}$ with $k = 0, \ldots, 5$. We call these approximations $\tilde{p}(10; \Delta t)$. Compute the errors $|p(10) - \tilde{p}(10; \Delta t)|$ and plot them doubly logarithmic. Use the norm function to calculate the norm of vectors. You should see a line for each integrator. Can you explain the different slopes?

10.2 Code a collocation-based integrator with sensitivities. We will use the following setup:

- Legendre polynomials of order K, using the time roots:

$$t = [0.0 \;\; 0.06943184420297355 \;\; 0.33000947820757187 \;\; 0.6699905217924282 \;\; 0.9305681557970262]$$

and build the Lagrange polynomials according to:

$$P_j(\tau) = \prod_{i \neq j} \frac{\tau - t_i}{t_j - t_i} \tag{10.21}$$

- For the sake of simplicity, we will use a single finite element per shooting interval. I.e. on the shooting interval $[t_k,\ t_{k+1}]$, the state trajectories are fully interpolated as:

$$x(\theta, t) = \sum_{j=0}^{K} \theta_j P_j\left(\frac{t - t_k}{t_{k+1} - t_k}\right), \qquad \forall t \in [t_k,\ t_{k+1}] \tag{10.22}$$

- The collocation equations then read as:

$$\begin{aligned} x(\theta, t_k) - x_k &= 0 \\ \frac{\partial}{\partial t} x(\theta, t)\Big|_{t=t_j} - F\left(x\left(\theta, t_j\right), u_k\right) &= 0, \qquad j = 1, ..., K \end{aligned} \tag{10.23}$$

where x_k is the initial condition for the shooting interval $[t_k,\ t_{k+1}]$. Note that $x\left(\theta, t_j\right) = \theta_j$.

Hints:

- *define your collocation polynomials $P_j(t)$ symbolically and compute their time derivatives $\dot{P}_j(t)$ by symbolic differentiation. Export them via matlabFunction to build the interpolations you need (i.e. for $\frac{\partial}{\partial t} x(\theta, t)\Big|_{t=t_j}$ and $x(\theta, t_k)$).*
- *Build the collocation constraints* (10.23) *also symbolically so that you can differentiate them automatically. Export all functions for the numerical part.*

- *Observe* (10.21) *and* (10.22)*, and reflect* ***carefully*** *on how and where the duration of your shooting intervals* $\Delta t = t_{k+1} - t_k$ *needs to be inserted. This is a classic source of error !*
- *Think also carefully of where/when to form the updates of your collocation variables in your integrator, such that your sensitivities match the collocation variables (and therefore the "final state") that your integrator delivers.*
- *It can be advantageous to use LU decompositions in your integrator. E.g. in Matlab you can form your factor using the function* $[L, U] = \mathrm{lu}(\nabla g^\top)$*. The solution to* $\nabla g^\top x + y = 0$ *is then given by* $y = -U \backslash (L \backslash x)$*. Observe that you can then re-use your factors for the sensitivity computations.*

Note: collocation-based integration is rather intensive coding-wise. Make sure you think through your coding strategy before starting.

(a) Deploy a (full-step) Newton scheme to solve the collocation equations (10.23). Have an "integrator tolerance" $\mathrm{tol}_{\mathrm{integ}}$ variable to control the accuracy of your Newton iterations.

(b) Introduce a computation of the integrator sensitivities.

(c) Validate your integrator by deploying it on an LTI. In this case, at steady-state your sensitivities will match the zero-order-hold linear discretisation of the dynamics (Matlab function "c2d") when $t_{k+1} - t_k \to 0$.

(d) Deploy your integrator on the pendulum dynamics built by the "WriteDynamics.m".

(e) Introduce your collocation-based integrator in your multiple-shooting code of the "Shooting" assignment. Use a terminal constraint to force your system to be at $x = 0$ at final time. You can build a "smart" initial guess for your integrators. Verify that your solver converges for moderate tolerances (try e.g. tol = 10^{-4})

(f) Test your solver at a tight tolerance (try e.g. tol = 10^{-10}), and experiment with the tolerance $\mathrm{tol}_{\mathrm{integ}}$ you set in the integrator. You will probably have to neutralise your line-search here, as it will become the numerically very sensitive. Explain what you observe.

10.3 **Adjoint-mode Differentiation** Consider the discrete dynamics

$$x_{k+1} = f(x_k, u_k), \quad k = 0, ..., N-1 \tag{10.24}$$

(a) Consider the cost function

$$\Phi = T(x_N) \tag{10.25}$$

Prove the following statement:

$$\nabla_{x_0}\Phi = \lambda_0 \tag{10.26}$$

where λ_0 is provided by the following recursion:

$$\lambda_{k-1} = \nabla_{x_k} f(x_k, u_k)\,\lambda_k \qquad \text{with} \qquad \lambda_N = \nabla_{x_N} T(x_N)$$

Hint: compute $\nabla_{x_k}\Phi$ by proceeding backward, starting from $\nabla_{x_N}\Phi$ and making your way to $\nabla_{x_0}\Phi$ via chain-ruling the dynamics.

(b) We now consider a cost function made of a stage and terminal cost:

$$\Phi = T(x_N) + \sum_{k=0}^{N-1} L(x_k, u_k) \tag{10.27}$$

Prove that (10.26) is still valid if λ_0 is provided by the following recursion:

$$\lambda_k = \nabla_{x_k}\mathcal{L}(x_k, u_k, \lambda_{k+1}) \qquad \lambda_N = \nabla_{x_N} T(x_N)$$

with $\mathcal{L}(x, u, \lambda) = L(x, u) + \lambda^\top f(x, u)$.

Hint: we use a similar strategy as in the previous question. However we need to be very careful here about the implicit and explicit dependencies of the functions on the variables. One way of handling this problem is to clearly distinguish between partial and total derivatives.

(c) A problem with a cost function of the form (10.27) and dynamics (10.24) can always be rewritten as:

$$\Phi = T(x_N) + x_N^{\mathrm{A}}$$

where $x_N^{\mathrm{A}} \in \mathbb{R}$ arises from the *state augmentation*:

$$\bar{x}_{k+1} = \begin{bmatrix} x_{k+1} \\ x_{k+1}^{\mathrm{A}} \end{bmatrix} = \begin{bmatrix} f(x_k, u_k) \\ x_k^{\mathrm{A}} + L(x_k, u_k) \end{bmatrix} = \bar{f}(\bar{x}_k, u_k)\,.$$

Such a reformulation allows one to get rid of the stage cost in any optimisation problem, and consider only problems with terminal cost without loss of generality. Reconcile formally this reformulation with the results established before. In particular, what happens to the multipliers of the formulation using a stage cost when switching to the reformulation ?

(d) Consider the discrete dynamics (10.24), and the cost function (10.25). Prove the more generic statement:

$$\nabla_p \Phi = \nabla_p x_0 (p) \lambda_0 + \sum_{k=0}^{N-1} \nabla_p \mathcal{L} (x_k, u_k (p), \lambda_{k+1})$$

for any variable p entering in the construction of the inputs u_k and/or initial conditions x_0, where the λ_k:s are given by the following recursion:

$$\lambda_k = \nabla_{x_k} \mathcal{L} (x_k, u_k, \lambda_{k+1}) \qquad \lambda_N = \nabla_{x_N} T (x_N)$$

and with $\mathcal{L} (x, u, \lambda) = \lambda^\top f (x, u)$.

Hint: this is a tricky one. You need to use the *augmented cost* function:

$$\Phi = T (x_N) + \sum_{k=0}^{N-1} \lambda_{k+1}^\top (f (x_k, u_k) - x_{k+1})$$

and take the Jacobian with respect to p. You will have to make ad-hoc simplifications and spot the telescopic sum (i.e. a sum where each term cancels out with the subsequent one).

11

The Hamilton-Jacobi-Bellman Equation

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

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

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

11.1 Dynamic Programming in Continuous Time

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

$$\begin{aligned}
\underset{x,\,u}{\text{minimize}} \quad & \sum_{i=0}^{N-1} hL(x_i,u_i) \;+\; E\left(x_N\right) \\
\text{subject to} \quad & x_0 - \bar{x}_0 = 0, \\
& x_{i+1} = x_i + hf(x_i,u_i), \quad i = 0,\ldots,N-1.
\end{aligned}$$

Dynamic programming applied to this optimization problem yields

$$J_k(x) \;=\; \underset{u}{\text{minimize}} \quad hL(x,u) + J_{k+1}(x + hf(x,u)).$$

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

$$J_k(x,t_k) \;=\; \underset{u}{\text{minimize}} \quad hL(x,u) + J(x + hf(x,u), t_k + h).$$

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

$$J(x,t) \;=\; \underset{u}{\text{minimize}} \quad hL(x,u) + J(x,t) + h\nabla_x J(x,t)^\top f(x,u) + h\frac{\partial J}{\partial t}(x,t) + O(h^2).$$

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

$$-\frac{\partial J}{\partial t}(x,t) \;=\; \underset{u}{\text{minimize}} \quad L(x,u) + \nabla_x J(x,t)^\top f(x,u).$$

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

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

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

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

One ought to observe that the optimal feedback control does not depend on the absolute value, but only on the gradient of the value function, $\nabla_x J(x,t)$. Introducing the variable $\lambda \in \mathbb{R}^{n_x}$ as this gradient, one can define the *Hamiltonian function*

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

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

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

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

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

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

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

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

11.2 Linear Quadratic Control and Riccati Equation

Let us consider a linear quadratic optimal control problem of the following form:

$$\begin{aligned}
\underset{x(\cdot),u(\cdot)}{\text{minimize}} \quad & \int_0^T \begin{bmatrix} x \\ u \end{bmatrix}^\top \begin{bmatrix} Q(t) & S(t)^\top \\ S(t) & R(t) \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} dt \;+\; x(T)^\top P_{\mathrm{T}} x(T) \\
\text{subject to} \quad & x(0) - x_0 = 0, && \text{(fixed initial value)}, \\
& \dot{x} - A(t)x - B(t)u = 0, \quad t \in [0,T], && \text{(linear ODE model)}.
\end{aligned}$$

As in discrete time, the value function is quadratic for this type of problem. In order to verify this statement, let us first observe that $J(x,T) = x^\top P_{\mathrm{T}} x$ is quadratic. Let us assume for now that $J(x,t)$ is quadratic for all time, i.e. $J(x,t) = x^\top P(t) x$ for some matrix $P(t)$. Under this assumption, the HJB Equation reads as

$$-\frac{\partial J}{\partial t}(x,t) \;=\; \underset{u}{\text{minimize}} \quad \begin{bmatrix} x \\ u \end{bmatrix}^\top \begin{bmatrix} Q(t) & S(t)^\top \\ S(t) & R(t) \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} + 2x^\top P(t)(A(t)x + B(t)u).$$

If symmetrized, the right reads as:

$$\underset{u}{\text{minimize}} \quad \begin{bmatrix} x \\ u \end{bmatrix}^\top \begin{bmatrix} Q + PA + A^\top P & S^\top + PB \\ S + B^\top P & R \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix}.$$

By the Schur Complement Lemma 8.2, this yields

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

which is again a quadratic term. Thus, as J is quadratic at a time T, it remains quadratic throughout the backwards evolution. The resulting matrix differential

equation

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

with terminal condition

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

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

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

11.3 Infinite Time Optimal Control

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

$$\begin{aligned} \underset{x(\cdot),\,u(\cdot)}{\text{minimize}} \quad & \int_0^\infty L(x(t),u(t))\,dt \\ \text{subject to} \quad & x(0) - x_0 = 0, \\ & \dot{x}(t) - f(x(t),u(t)) = 0, \quad t \in [0,\infty]. \end{aligned}$$

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

$$0 = \underset{u}{\text{minimize}} \quad L(x,u) + \nabla_x J(x)^\top f(x,u)$$

with stationary optimal feedback control law

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

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

$$\dot{P} = 0$$

and solving

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

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

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

Exercises

11.1 ...

12

Pontryagin and the Indirect Approach

The indirect approach is an extremely elegant and compact way to characterize and compute solutions to optimal control problems. Its origins date back to the calculus of variations and the classical work by Euler and Lagrange. However, its full generality was only developed in 1950s and 1960s, starting with the seminal work of Pontryagin and coworkers [59]. One of the major achievements of their approach compared to the previous work was the possibility to treat inequality path constraints, which appear in most relevant applications of optimal control, notably in time optimal problems. *Pontryagin's Maximum Principle* describes the necessary optimality conditions for optimal control in continuous time. Using these conditions in order to eliminate the controls from the problem and then numerically solving a boundary value problem (BVP) is called the *indirect approach* to optimal control. It was widely used when the Sputnik and Apollo space missions were planned and executed, and is still very popular in aerospace applications. The main drawbacks of the indirect approach are the facts, (a) that it must be possible to eliminate the controls from the problem by algebraic manipulations, which is not always straightforward or might even be impossible, (b) that the optimal controls might be a discontinuous function of x and λ, such that the BVP is possibly given by a non-smooth differential equation, and (c) that the differential equation might become very nonlinear and unstable and not suitable for a forward simulation. All these issues of the indirect approach can partially be addressed, and most importantly, it offers an exact and elegant characterization of the solution of optimal control problems in continuous time.

12.1 The HJB Equation along the Optimal Solution

In order to derive the necessary optimality conditions stated in Pontryagin's Maximum Principle, let us again regard the simplified optimal control problem:

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

and let us recall that the Hamiltonian function was defined as $H(x,\lambda,u) = L(x,u) + \lambda^\top f(x,u)$ and the Hamilton-Jacobi-Bellman equation was formulated as: $-\frac{\partial J}{\partial t}(x,t) = \min_u H(x,\nabla J(x,t),u)$ with terminal condition $J(x,T) = E(x)$. We already made the important observation that the optimal feedback controls

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

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

$$u^\star_{\text{exp}}(x,\lambda) \;=\; \arg\min_u \quad H(x,\lambda,u).$$

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

The problem of computing the optimal input is now reduced to the problem of finding the optimal state and costate trajectories $x^*(t)$ and $\lambda^*(t)$. The idea is to assume that the trajectory is known, and to differentiate the HJB Equation along this optimal trajectory. Let us regard the HJB Equation

$$-\frac{\partial J}{\partial t}(x,t) \;=\; \underset{u}{\text{minimize}}\; H\left(x,\nabla_x J(x,t),u\right) = H\left(x,\nabla_x J(x,t),u^\star_{\text{exp}}(x,\nabla_x J(x,t))\right)$$

and differentiate it totally with respect to x. Note that the right-hand side depends via $\nabla_x J(x,t)$ and $u^\star_{\text{exp}}$ indirectly on x. Fortunately, we know that $\frac{\partial H}{\partial u}(x^*,\lambda^*,u^*) = 0$ due to the minimum principle. Moreover, it is useful to remember here that

$\lambda(t) = \nabla_x J(x(t), t)$, such that $\frac{\partial \lambda}{\partial x} = \nabla_x^2 J(x(t), t)$. We then obtain

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

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

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

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

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

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

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

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

12.2 Obtaining the Controls on Regular and on Singular Arcs

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

$$u^\star_{\text{exp}}(x,\lambda) \;=\; \arg\min_u \quad H(x,\lambda,u).$$

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

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

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

In this case, the controls appear explicitly in the analytic expression of the derivative. We can then solve the implicit function and compute the control $u^\star_{\text{exp}}(x,\lambda)$ either numerically, or transform the equation in order to obtain it explicitly. Let us illustrate this with an example.

Example 12.1 (Linear Quadratic Control with Regular Cost)**.** Regard $L(x,u) = \frac{1}{2}(x^\top Qx + u^\top Ru)$ with positive definite R and $f(x,u) = Ax + Bu$. Then

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

and

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

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

$$u^\star_{\text{exp}}(x,\lambda) = -R^{-1}B^\top\lambda.$$

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

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

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

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

If a fixed initial value $\bar x_0$ is provided for the optimal control problem, and

quadratic terminal cost, i.e. $E(x) = \frac{1}{2}x^\top Px$ is present, then the TPBVP that we need to solve is given by

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

Note that this TPBVP is linear and therefore admits an explicit solution.

□

The second and more complicated case occurs in the case u^* is not provided by the implicit function

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

The implicit function theorem tells us that this occurs when $\frac{\partial^2 H}{\partial u^2}(x, \lambda, u)$ is rank-deficient (i.e. null in the single-input case). We then speak of a *singular arc*. This e.g. occurs if $L(x, u)$ is independent of u and $f(x, u)$ is linear in u, as then $\frac{\partial H}{\partial u}$ does not depend explicitly on u. Roughly speaking, singular arcs are due to the fact that *singular perturbations* of the controls – that go up and down infinitely fast – would not matter in the objective and yield exactly the same optimal solution as the well-behaved piecewise continuous control in which we are usually interested. Note that the controls still influence the trajectory on a singular arc, but that this influence occurs only indirectly, via the evolution of the states.

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

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

which yields

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

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

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

explicitly depends on u. Then we can invert the relation and finally have an explicit equation for u^*. It is interesting to observe here that an explicit dependence on u can occur only for n even. Let us illustrate this with another example.

Example 12.2 (Linear Quadratic Control with Singular Cost)**.** Regard $L(x,u) = x^\top Q x$ and $f(x,u) = Ax + Bu$. Then

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

and

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

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

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

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

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

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

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

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

$$u^\star_{\mathrm{exp}}(x,\lambda) = (B^\top QB)^{-1}B^\top \left(A^\top Qx - QAx + A^\top A^\top \lambda\right).$$

12.3 Pontryagin with Path Constraints

Let us consider here OCPs with path constraints:

$$\begin{aligned}
\underset{x(\cdot),\,u(\cdot)}{\text{minimize}} \quad & \int_0^T L(x(t),u(t))\,dt \;+\; E\left(x(T)\right) & & \\
\text{subject to} \quad & x(0) - \bar{x}_0 = 0 & & \text{(States initial value)},\\
& \dot{x}(t) - f(x(t),u(t)) = 0, & t\in[0,T] \quad & \text{(ODE model)},\\
& h\left(x(t),u(t)\right) \le 0, & t\in[0,T] \quad & \text{(Path Constraints)},
\end{aligned} \tag{12.1}$$

If path constraints of the form $h(x(t),u(t)) \le 0$ are to be satisfied by the solution of the optimal control problem for $t \in [0,T]$ the same formalism as

developed before is still applicable. In this case, it can be shown that for given x and λ, we need to determine the optimizing u from

$$\begin{aligned} u^{\star}_{\text{exp}}(x,\lambda) \;=\; & \underset{u}{\arg\min} \quad H(x,\lambda,u) \\ & \text{subject to} \quad h(x,u) \leq 0. \end{aligned} \tag{12.2}$$

This is simple to apply in the case of pure control constraints, i.e. if we have only $h(u) \leq 0$.

In the special case where the constraints $h(x,u)$ are not "fully controllable", a singular situation usually occurs. Higher-order derivatives of the state constraints ought then to be considered in order to describe the trajectories along the active state constraint at the solution; in the case of uncontrollable state constraints, we will only have a single time point where the state trajectory touches the constraint and the adjoints will typically jump at this point. Let us leave all complications away and illustrate in this section only the nicest case, the one of pure control constraints.

In the case of mixed constraints with regular solution of the above optimization problem (12.2), a simple way to describe the optimal solution is via the modified Hamiltonian function:

$$H(x,u,\lambda,\mu) = L(x,u) + \lambda^{\top} f(x,u) + \mu^{\top} h(x,u). \tag{12.3}$$

One ought to note that this modification affects both the adjoint differential equation and the Hamiltonian stationarity. Similarly to the KKT conditions, the adjoint variables μ must be positive at their solution, and a complementarity slackness condition must hold at every time $t \in [0,\, T]$. The resulting conditions of optimality can be written as

$$x(0) - \bar{x}_0 = 0, \tag{12.4a}$$
$$\lambda(T) - \nabla E(x(T)) = 0, \tag{12.4b}$$
$$\dot{x}(t) - \nabla_{\lambda} H^{*}(x(t),\lambda(t),\mu(t)) = 0, \quad t \in [0,T], \tag{12.4c}$$
$$\dot{\lambda}(t) + \nabla_{x} H^{*}(x(t),\lambda(t),\mu(t)) = 0, \quad t \in [0,T], \tag{12.4d}$$
$$h_i^{\star}(x(t),\lambda(t),\mu(t)) \cdot \mu_i(t) = 0, \quad t \in [0,T], \tag{12.4e}$$
$$h_i^{\star}(x(t),\lambda(t),\mu(t)) \leq 0, \qquad \mu(t) \geq 0 \quad t \in [0,T], \tag{12.4f}$$

where

$$H^{*}(x(t),\lambda(t),\mu(t)) = H\left(x(t), u^{\star}_{\text{exp}}(x(t),\lambda(t),\mu(t)), \lambda(t), \mu(t)\right) \tag{12.5}$$
$$h^{*}(x(t),\lambda(t),\mu(t)) = h\left(x(t), u^{\star}_{\text{exp}}(x(t),\lambda(t),\mu(t))\right) \tag{12.6}$$

and

$$u^\star_{\text{exp}}(x,\lambda,\mu) \;=\; \arg\min_{u} \quad H(x,\lambda,\mu,u).$$

is based on (12.3). We consider the problem of solving (12.4) numerically in Section 12.6.4. Let us consider here simple special cases.

Example 12.3 (Linear Quadratic Problem with Control Constraints)**.** Let us regard constraints $h(u) = Gu+b \leq 0$ and the Hamiltonian $H(x,\lambda,u) = \frac{1}{2}x^\top Qx + u^\top Ru + \lambda^\top(Ax+Bu)$ with R invertible. Then

$$\begin{aligned} u^\star_{\text{exp}}(x,\lambda) \;=\; & \arg\min_{u} \quad H(x,\lambda,u) \\ & \text{subject to} \quad h(u) \leq 0 \end{aligned}$$

is equal to

$$\begin{aligned} u^\star_{\text{exp}}(x,\lambda) \;=\; & \arg\min_{u} \quad \frac{1}{2}u^\top Ru + \lambda^\top Bu \\ & \text{subject to} \quad Gx + b \leq 0 \end{aligned}$$

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

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

Example 12.4 (Scalar Bounded Control)**.** Regard scalar u and constraint $|u| \leq 1$, with Hamiltonian

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

Then, with

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

we have

$$u^\star_{\text{exp}}(x,\lambda) = \max\{-1, \min\{1, \tilde{u}(x,\lambda)\}\}.$$

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

12.4 Properties of the Hamiltonian System

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

differential equation, and second recall that

$$\nabla_\lambda H^*(x,\lambda) = f\left(x, u^\star_{\text{exp}}(x,\lambda)\right).$$

Thus,

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

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

$$y = \begin{bmatrix} x \\ \lambda \end{bmatrix}, \quad \text{and} \quad \varphi(y) = \begin{bmatrix} \nabla_\lambda H^*(x,\lambda) \\ -\nabla_x H^*(x,\lambda) \end{bmatrix}. \tag{12.7}$$

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

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}H^*(x,\lambda) &= \nabla_x H^*(x,\lambda)^\top \dot{x} + \nabla_\lambda H^*(x,\lambda)^\top \dot{\lambda} \\ &= \nabla_x H^*(x,\lambda)^\top \nabla_\lambda H^*(x,\lambda) - \nabla_\lambda H^*(x,\lambda)^\top \nabla_x H^*(x,\lambda) = 0.\end{aligned}$$

Second, by Liouville's Theorem, the fact that the system $\dot{y} = \varphi(y)$ is a Hamiltonian system also means that the volume in the phase space of $y = (x, \lambda)$ is preserved. The implication of this is that even if the dynamics of x are stable and contracting fast, the dynamics of λ must be expanding and therefore unstable. We illustrate this effect in Fig. 12.1 for the optimal control problem

$$\begin{aligned}\underset{x(\cdot),\,u(\cdot)}{\text{minimize}} \quad & \frac{1}{2}\int_0^T x(t)^2 + u(t)^2 \, dt \\ \text{subject to} \quad & x(0) - 1 = 0 && \text{(fixed initial value)}, \\ & \dot{x}(t) + \sin x(t) - u(t) = 0, \quad t \in [0,T] && \text{(ODE model)}.\end{aligned}$$

yielding the state-costate equations:

$$\dot{x} = -\lambda - \sin x, \qquad \dot{\lambda} = \lambda \cos x - x$$

This is an unfortunate fact for numerical approaches that solve the TPBVP using a full simulation of the combined differential equation system, like single shooting. If the system $\dot{x} = f(x,u)$ has either some *very unstable* or some *very stable* modes, in both cases the forward simulation of the combined system is an ill-posed problem. In general, the conservation of volume in the state-costate space makes solving the TPBVP problem numerically very difficult with single shooting techniques. The indirect approach is however applicable using alternative numerical approaches, but it then looses some of its appeal.

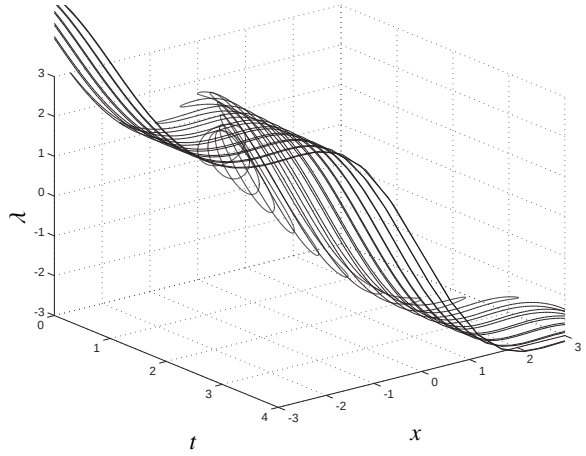


Figure 12.1 Illustration of the volume conservation in the state-costate space. Here an integration of the state-costate equations is displayed for a disc of initial conditions x, λ at time $t = 0$. The evolution of this disc of initial conditions is displayed for various time instants in the time interval $[0, 4]$. The area of the disc is preserved by the state-costate dynamics, such that a contraction of the area along a dimension yields an expansion in others.

Different numerical approaches for solving the TPBVP are presented further in Section 12.6.

12.5 Connection to the Calculus of Variations

Calculus of variations are fundamental to optimal control in general, and to indirect methods in particular. It offers powerful insights into the mathematics of optimal control, and also allows for explaining the behavior of direct methods in some special cases. Consider a simple optimal control problems, which we recast as a the functional:

$$\begin{aligned} J\left[u\left(\cdot\right)\right] &= \phi\left(x\left(t_{\mathrm{f}}\right)\right) + \int_{t_0}^{t_{\mathrm{f}}} L\left(x, u\right) \mathrm{d}t \\ &\text{where:} \quad \dot{x} = f\left(x, u\right), \quad x\left(t_0\right) = x_0 \end{aligned}$$

that maps an input profile $u\left(\cdot\right)$ into the corresponding cost $J\left[u\left(\cdot\right)\right] \in \mathbb{R}$. We can then define the Gâteaux derivative

$$\delta J\left[u\left(\cdot\right), \xi\left(\cdot\right)\right] = \lim_{\tau \to 0} \frac{J\left[u\left(\cdot\right) + \tau\xi\left(\cdot\right)\right] - J\left[u\left(\cdot\right)\right]}{\tau}.$$

Note that Gâteaux derivatives can be construed as the extension of directional derivatives to arbitrary vector spaces, including infinite-dimensional ones. Here we differentiate the functional $J[u(\cdot)]$ in the "direction" $\xi(\cdot)$. Optimality then requires that

$$\delta J\left[u^{\star}(\cdot), \xi(\cdot)\right] = 0, \quad \forall \xi(\cdot).$$

A useful interpretation of the stationarity of H is provided by the fundamental Lemma of Calculus of Variations:

$$\delta J[u(.), \xi(.)] = \int_0^{t_f} H_u(x(t), \lambda(t), u(t)) \cdot \xi(t)\, \mathrm{d}t.$$

In particular, it follows that at the optimal solution $u^{\star}(\cdot)$

$$\delta J\left[u^{\star}(\cdot), \xi(\cdot)\right] = \int_0^{t_f} H_u\left(x^{\star}(t), \lambda^{\star}(t), u^{\star}(t)\right) \cdot \xi(t)\, \mathrm{d}t = 0, \quad \forall \xi(t)$$

In the case the optimal input $u^{\star}(\cdot)$ is free to move locally at any time $t \in [0,\ t_f]$, the perturbation $\xi(t)$ is unrestricted and the condition of optimality becomes

$$H_u\left(x^{\star}(t), \lambda^{\star}(t), u^{\star}(t)\right) = 0$$

for all t, thus we recover the observations already made in the previous sections. A special case of the observation above will be of interest in the following. It stems from the restriction of the control profile $u(\cdot)$ to the Banach space (i.e. loosely speaking the notion of vector space extended to functions) of piecewise-constant functions. In such a case:

$$u(t) = u_k, \quad \xi(t) = \xi_k, \qquad \forall\, t \in [t_k,\ t_{k+1}].$$

This restriction allows one to discuss the zero-order hold discretization of the control input commonly used in direct methods framing it in the context of the Calculus of Variations. In this case, optimality requires

$$\begin{aligned}\delta J\left[u^{\star}(.), \xi(.)\right] &= \int_0^{t_f} H_u\left(x^{\star}(t), \lambda^{\star}(t), u^{\star}(t)\right) \cdot \xi(t)\, \mathrm{d}t \\ &= \sum_{k=0}^{N-1} \int_{t_k}^{t_{k+1}} H_u\left(x^{\star}(t), \lambda^{\star}(t), u_k^{\star}\right) \cdot \xi_k\, \mathrm{d}t = 0, \quad \forall \xi_k.\end{aligned}$$

Hence the optimality condition is

$$\int_{t_k}^{t_{k+1}} H_u\left(x^{\star}(t), \lambda^{\star}(t), u_k^{\star}\right) \mathrm{d}t = 0, \qquad \forall k. \tag{12.8}$$

For a problem with a bounded, scalar input, i.e. for an OCP of the form

$$\begin{aligned} J\left[u\left(.\right)\right] &= \phi\left(x\left(t_{\mathrm{f}}\right)\right) + \int_{t_0}^{t_{\mathrm{f}}} L\left(x,u\right) \mathrm{d}t \\ \text{subject to:} \quad & \dot{x} = f\left(x,u\right), \quad x\left(t_0\right) = x_0, \\ & u_{\min} \leq u \leq u_{\max} \end{aligned}$$

with $u \in \mathbb{R}$, the condition (12.8) then must hold at the optimum for all k for which $u_{\min} < u_k < u_{\max}$. These observations will be useful in Section 13.5 to discuss the numerical solutions to singular optimal control problems via direct methods.

12.6 Numerical Solution of the TPBVP

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

$$r_0 = x(0) - \bar{x}_0 = 0, \tag{12.9a}$$
$$r_T = \lambda(T) - \nabla E(x(T)) = 0, \tag{12.9b}$$
$$\dot{x}(t) - \nabla_\lambda H^*(x(t), \lambda(t)) = 0, \quad t \in [0,T], \tag{12.9c}$$
$$\dot{\lambda}(t) + \nabla_x H^*(x(t), \lambda(t)) = 0, \quad t \in [0,T]. \tag{12.9d}$$

Using the shorthands (12.7) and

$$b\left(y(0), y(T), \bar{x}_0\right) = \begin{bmatrix} r_0\left(y(0), \bar{x}_0\right) \\ r_T\left(y(T)\right) \end{bmatrix},$$

the system of equations can be summarized as:

$$b\left(y(0), y(T), \bar{x}_0\right) = 0, \tag{12.10a}$$
$$\dot{y}(t) - \varphi(y(t)) = 0, \quad t \in [0,T]. \tag{12.10b}$$

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

12.6.1 Single shooting

Single shooting starts with the following idea: for any guess of the initial value λ_0, we can use a numerical integration routine in order to obtain the state-costate trajectory as a function of $\lambda_0, \bar{x}_0$, i.e. $y(t, \lambda_0, \bar{x}_0)$ for all $t \in [0, T]$. This is visualized in Figure 12.2. The result is that the differential equation (12.10b) is by construction already satisfied, as well as the initial boundary condition (12.9a). Thus, we only need to enforce the boundary condition (12.9b), which we can do using the terminal trajectory value $y(T, \lambda_0, \bar{x}_0)$:

$$\underbrace{r_T\left(y(T, \lambda_0, \bar{x}_0)\right)}_{=:R_T(\lambda_0)} = 0.$$

For nonlinear dynamics φ, this equation can generally not be solved explicitly. We then use the Newton's method, starting from an initial guess λ_0, and iterating to the solution, i.e. we iterate

$$\lambda_0^{k+1} = \lambda_0^k - t_k \left(\frac{\partial R_T}{\partial \lambda_0}\left(\lambda_0^k\right)\right)^{-1} R_T\left(\lambda_0^k\right). \tag{12.11}$$

for some adequate step-size $t_k \in]0,\ 1]$. It is important to note that in order to evaluate $\frac{\partial R}{\partial \lambda_0}\left(\lambda_0^k\right)$ we have to compute the ODE sensitivities $\frac{\partial y(T, y_0)}{\partial \lambda_0}$.

In some cases, as said above, the forward simulation of the combined ODE might be an ill-conditioned problem so that single shooting cannot be employed. Even if the forward simulation problem is well-defined, the region of attraction of the Newton iteration on $R_T(\lambda_0) = 0$ can be very small, such that a good guess for λ_0 is often required. However, such a guess is typically unavailable. In the following example, we illustrate these observation on a simple optimal control problem.

Example 12.5. We consider the optimal control problem:

$$\begin{aligned} \underset{x(.),\,u(.)}{\text{minimize}} \quad & \int_0^T x_1(t)^2 + 10x_2(t)^2 + u(t)^2\, dt \\ \text{subject to} \quad & \dot{x}_1(t) = x_1(t)\, x_2(t) + u(t), \qquad x_1(0) = 0, \\ & \dot{x}_2(t) = x_1(t), \qquad\qquad\qquad x_2(0) = 1 \end{aligned} \tag{12.12}$$

with $T = 5$. This example does not hold a terminal cost or constraints, such that the terminal condition reads as $R_T(\lambda_0) = \lambda\,(T, \lambda_0, \bar{x}_0) = 0$. The state-costate trajectory at the solution is displayed in Figure 12.2. It is then interesting to build the function $\lambda_0 \mapsto \lambda\,(T, \lambda_0, \bar{x}_0)$ for various values of λ_0, see Figure 12.3. This function is very nonlinear, making it difficult for the Newton iteration to find the co-states' initial value λ_0 resulting in $\lambda\,(T, \lambda_0, \bar{x}_0) = 0$. More specifically,

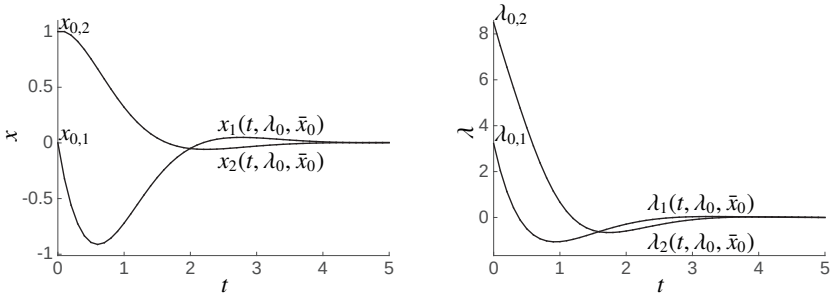


Figure 12.2 Illustration of the state and co-state trajectories for problem (12.12) at the solution delivering $\lambda(T, \lambda_0, \bar{x}_0) = 0$ for $T = 5$.

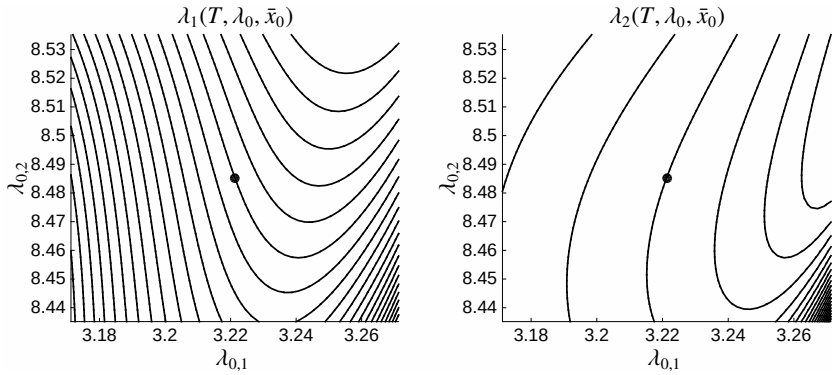


Figure 12.3 Illustration of the map $\lambda_0 \mapsto \lambda(T, \lambda_0, \bar{x}_0)$, in the form of level curves for $T = 5$. The black dot represents the solution of the TPBVP problem, where $R_T(\lambda_0) = \lambda(T, \lambda_0, \bar{x}_0) = 0$. One can observe that the map is very nonlinear, such that the Newton method can struggle to converge to the solution λ_0 of the TPBVP, unless a very good initial guess is provided.

the Newton iteration (full steps or reduced steps) converges only for a specific set of initial guesses λ_0^0 provided to the iteration (12.11), see Figure 12.4.

A crucial observation that will motivate an alternative to the single-shooting approach is illustrated in Figure 12.5, where the map $\lambda_0 \mapsto \lambda(t, \lambda_0, \bar{x}_0)$ is displayed for the integration times $t = 3$ and $t = 4$. The crucial observation here is that the map is fairly linear up to $t = 3$, and becomes increasingly nonlinear for larger integration times. This observation is general and motivates the core idea behind the alternatives to single shooting, namely that integration shall never be performed over long time intervals, so as to avoid creating strongly nonlinear functions in the TPBVP.

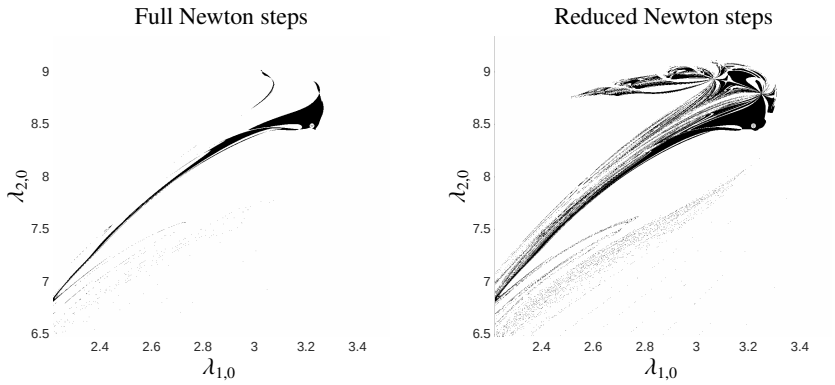


Figure 12.4 Illustration of the region of convergence of the Newton iteration (12.11) for problem (12.12) (in black, with full Newton steps on the left-hand side graph and with reduced steps on the right-hand side graph). Here we note $\lambda_{0,1}$, $\lambda_{0,2}$ the initial guess provided to the Newton iteration. The grey dots (at (3.22, 8.48)) depict the solution to the TPBVP. Only a fairly small, disconnected and highly convoluted set of initial guess for the co-states initial conditions leads to a convergence of the Newton iteration.

12.6.2 Multiple shooting

The nonlinearity of the integration map $\lambda_0 \mapsto y(t, \lambda_0, \bar{x}_0)$ for long integration times t motivates the "breaking down" of the full integration in small pieces, so as to avoid creating very nonlinear map in the TPBVP conditions. The idea is originally due to Osborne [57], and is based on dividing the time interval $[0,\ T]$ into (typically uniform) *shooting* intervals $[t_k,\ t_{k+1}] \subset [0,\ T]$, where the most common choice is $t_k = k\frac{T}{N}$. Let us then frame the integration over a short time interval $[t_k,\ t_{k+1}]$ with initial value s_k as the function $\Phi_k(s_k)$, defined as:

$$\Phi_k(s_k) = y(t_{k+1}) \quad \text{where} \tag{12.13a}$$

$$\dot{y}(t) - \varphi(y(t)) = 0, \quad t \in [t_k, t_{k+1}] \quad \text{and} \quad y(t_k) = s_k \tag{12.13b}$$

for $k = 0, \ldots, N-1$. We then rewrite the TPBVP conditions (12.10) as:

$$b(s_0, s_N, \bar{x}_0) = 0, \qquad \text{(boundary conditions)} \tag{12.14a}$$

$$\Phi_k(s_k) - s_{k+1} = 0, \quad k = 0, \ldots, N-1. \quad \text{(continuity conditions)} \tag{12.14b}$$

One can then rewrite the conditions (12.14) altogether as the function:

$$R_{\mathrm{MS}}(s, \bar{x}_0) = 0 \tag{12.15}$$

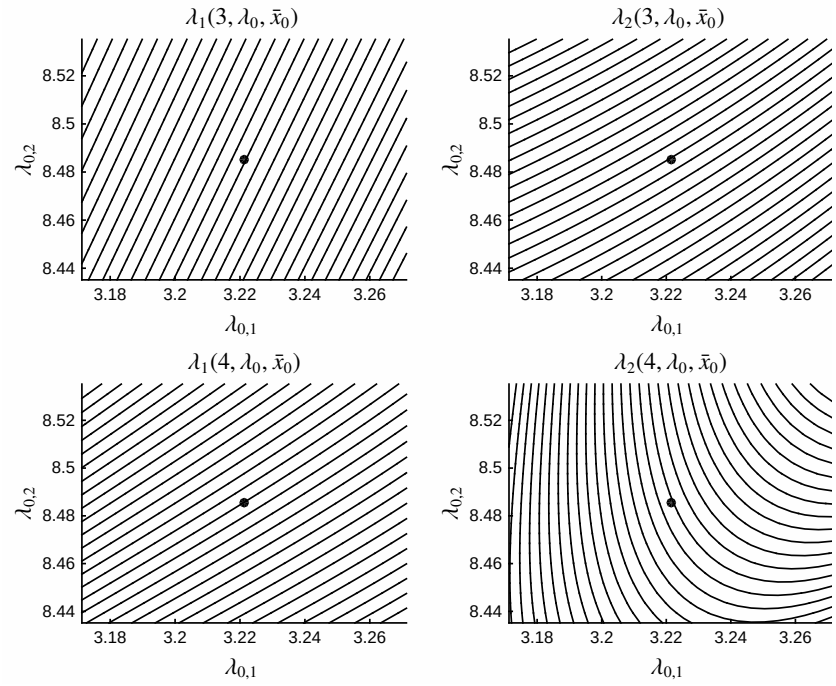


Figure 12.5 Illustration of the map $\lambda_0 \mapsto \lambda(t, \lambda_0, \bar{x}_0)$, in the form of level curves for different times t. The black dot represents the solution of the TPBVP problem, where $\lambda(T, \lambda_0, \bar{x}_0) = 0$. One can observe that the map is close to linear for "small" integration times t (upper graphs, where $t = 3$), and becomes increasingly nonlinear as the integration time increases (lower graph, where $t = 4$), until it reaches the final time $T = 5$, see Figure 12.3. This observation is general, and holds for most problems.

where we note $s = (s_0, \ldots, s_N)$. A Newton iteration can be then deployed on (12.15) to find the variables s, it reads as:

$$s^{k+1} = s^k - t_k \left(\frac{\partial R_{\text{MS}}}{\partial s} \left(s^k, \bar{x}_0 \right) \right)^{-1} R_{\text{MS}} \left(s^k, \bar{x}_0 \right). \tag{12.16}$$

for some step-size $t_k \in]0, 1]$. We illustrate the Multiple-Shooting approach in the following example.

Example 12.6. We consider the optimal control problem (12.12) from Example 12.5 with $T = 5$. If we denote $s_k = (x_k, \lambda_k)$, the boundary conditions for

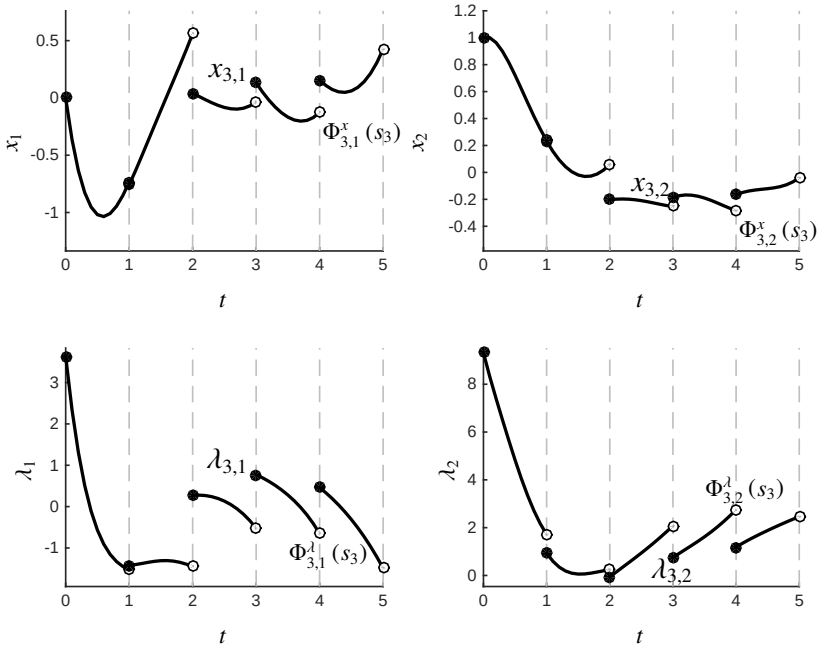


Figure 12.6 Illustration of the state and co-state trajectories for problem (12.12) during the multiple-shooting iterations (12.16), such that the conditions $\Phi_k(s_k) - s_{k+1} = 0$ are not yet fulfilled. Here, the discrete times t_k are depicted as grey dashed lines, the discrete state-costates $s_k = (x_{k,1}, x_{k,2}, \lambda_{k,1}, \lambda_{k,2})$ are depicted as black dots, and the resulting integrations $\Phi_k = \left(\Phi^x_{k,1}, \Phi^x_{k,2}, \Phi^\lambda_{k,1}, \Phi^\lambda_{k,2}\right)$ are depicted as white dots. The black curves represent the state-costate trajectories on the various time intervals $[t_k, t_{k+1}]$. At the solution (12.14), the conditions $\Phi_k(s_k) = s_{k+1}$ are enforced for $k = 1, \ldots, N-1$, such that the black and white dots coincide on each discrete time t_k.

this example then become:

$$x_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \qquad \lambda_N = 0. \tag{12.17}$$

We illustrate the Multiple-Shooting procedure (12.14) in Figure 12.6 for $N = 5$.

One ought to observe that the time intervals $[t_k, t_{k+1}]$ are of size $\frac{T}{N}$, and hence get shorter as N increases. Because one can "control" the length of the time interval over which the integration is performed via N, and because the functions $\Phi_k(s_k) - s_{k+1}$ become less nonlinear as the length of the time interval decreases, one can make them "arbitrarily" linear by increasing N. It follows that a sufficiently large N typically allows one to solve the Multiple-Shooting

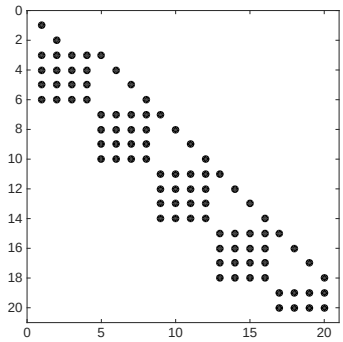


Figure 12.7 Illustration of sparsity pattern of the Jacobian matrix $\frac{\partial R_{\text{MS}}}{\partial s}$ in the Newton iteration (12.16) for the optimal control problem (12.12) approached via indirect multiple-shooting, for Example 12.6. Here we use $N = 5$. One can readily observe that the Jacobian matrix is sparse and highly structured. This structure arises via organising the algebraic conditions (12.15) and the variables s in time (i.e. in the order $k = 0, \dots, N$). Note that here the last variables s_N where eliminated using the equality $s_N = \Phi_{N-1}(s_{k-1})$. In the specific case of Example 12.6, the elimination has no impact on the Newton iteration because the boundary conditions $b(s_0, s_N, \bar{x}_0)$ are linear.

conditions (12.14) using a Newton iteration even if no good initial guess is available.

It is important to observe that the set of algebraic conditions (12.15) holds a large number of variables s, such that the Newton iteration (12.16) is deployed using large Jacobian matrices $\frac{\partial R_{\text{MS}}}{\partial s}$. However, these matrices are sparse, and if the algebraic conditions and variables are adequately organised, they are highly structured (see Figure 12.7), such that their factorization can be performed efficiently.

The second alternative to single-shooting is the object of the next section, and can be construed as an extreme case of Multiple-Shooting. We detail this next.

12.6.3 Collocation & Pseudo-spectral methods

The second alternative approach to single shooting is to use *simultaneous collocation* or Pseudo-spectral methods. As we will see next, the two approaches are fairly similar. The key idea behind these methods is to introduce *all* the variables involved in processing the integration of the dynamics, and the related algebraic conditions *into* the set of algebraic equations to be processed.

The most common implementation of this idea is based on the Orthogonal Collocation method presented in Section 10.3.

We consider the collocation-based integration of the state-costate dynamics on a time interval $[t_k,\, t_{k+1}]$ starting from the initial value s_k, as described in equation (10.5). The integration is then based on solving a set of collocation equations:

$$v_{k,0} = s_k \tag{12.18a}$$

$$\dot{p}\left(t_{k,i}, v_k\right) = \varphi(v_{k,i}, t_{k,i}), \qquad i = 1, \ldots, d \tag{12.18b}$$

for $k = 0, \ldots, N-1$, where $t_{k,i} \in [t_k,\, t_{k+1}]$ for $i = 0, \ldots, d$, and the variables $v_k \in \mathbb{R}^{2n_x \cdot (d+1)}$ hold the discretisation of the continuous state-costates dynamics. The TPBVP discretised using orthogonal collocation then holds the variables $v_{k,i}$ and s_k for $k = 0, \ldots, N-1$ and $i = 1, \ldots, d$, and the following constraints:

$$b\left(s_0, s_N, \bar{x}_0\right) = 0, \qquad \text{(boundary condition)}, \tag{12.19a}$$

$$p(t_{k+1}, v_k) - s_{k+1} = 0, \qquad \text{(continuity condition)}, \tag{12.19b}$$

$$v_{k,0} - s_k = 0, \qquad \text{(initial values)}, \tag{12.19c}$$

$$\dot{p}\left(t_{k,i}, v_k\right) - \varphi(v_{k,i}, t_{k,i}) = 0, \qquad \text{(dynamics)}. \tag{12.19d}$$

One can observe that equations (12.19b) and (12.19c) are linear, while equation (12.19d) is nonlinear when the dynamics are nonlinear. One can also observe that the variables $s_{0,\ldots,N-1}$ can actually be eliminated from (12.19), to yield a slightly more compact set of equation, with $k = 0, \ldots, N-1$ and $i = 1, \ldots, d$:

$$b\left(v_{k,0}, v_{N,0}, \bar{x}_0\right) = 0, \qquad \text{(boundary condition)}, \tag{12.20a}$$

$$p(t_{k+1}, v_k) - v_{k+1,0} = 0, \qquad \text{(continuity condition)}, \tag{12.20b}$$

$$\dot{p}\left(t_{k,i}, v_k\right) - \varphi(v_{k,i}, t_{k,i}) = 0, \qquad \text{(dynamics)}. \tag{12.20c}$$

This elimination does not modify the behavior of the Newton iteration. We can gather the algebraic conditions (12.20) and the variables s_k, v_k in the compact form

$$R_{\text{IC}}\left(w, \bar{x}_0\right) = 0, \tag{12.21}$$

where $w = \{v_{0,0}, \ldots, v_{0,d}, \ldots, v_{N-1,0}, \ldots, v_{N-1,d}, v_{N,0}\}$. A Newton iteration can be then deployed on (12.21) to find the variables w, it reads as

$$w^{k+1} = w^k - t_k \left(\frac{\partial R_{\text{IC}}}{\partial w}\left(w^k, \bar{x}_0\right) \right)^{-1} R_{\text{IC}}\left(w^k, \bar{x}_0\right), \tag{12.22}$$

for some step-size $t_k \in]0,\, 1]$. We illustrate the indirect collocation approach in the following example.

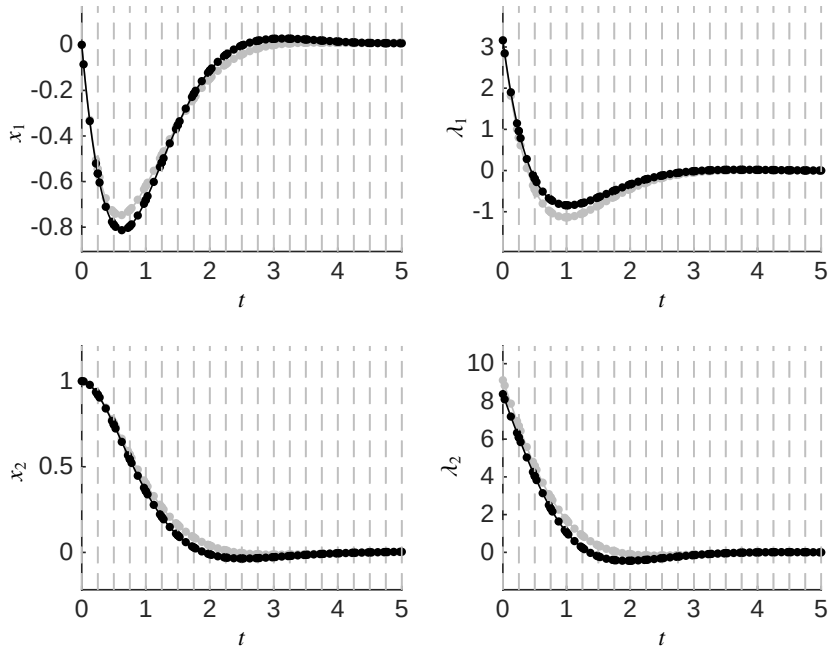


Figure 12.8 Illustration of the state and co-state trajectories for problem (12.12) using the orthogonal collocation approach with $N = 20$. The grey curves display the state-costate trajectories after the first full Newton step of (12.22), while the black curves report the state-costate trajectories at convergence. The discrete times t_k are depicted as grey dashed lines, the discrete state-costates on the time grid $t_{k,i}$ are depicted as dots. Note that the continuity conditions (12.19b) in the collocation method are linear in the variables w, such that the trajectories are continuous after the first full Newton step (hence the grey curves are continuous, even though the problem is not solved yet).

Example 12.7. We consider the optimal control problem (12.12) from Example 12.5 with $T = 5$. We illustrate the Orthogonal Collocation procedure (12.19) in Figure 12.8 for $N = 10$. The sparsity pattern of the Jacobian matrix $\frac{\partial R_{\text{IC}}}{\partial w}$ from the Newton iteration (12.22) is illustrated in Figure (12.9). The variables and constraints were ordered with respect to time. Even though it is large, the complexity of forming factorisations of the Jacobian matrix $\frac{\partial R_{\text{IC}}}{\partial w}$ is limited as it is sparse and highly structured.

Pseudo-spectral methods deploy a very similar approach to the one described here, to the exception that they skip the division of the time interval $[0,\ T]$ into subintervals $[t_k,\ t_{k+1}]$, and use a single set of basis functions spanning the entire

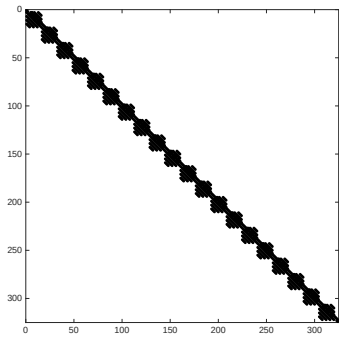


Figure 12.9 Illustration of the sparsity structure for the Jacobian $\frac{\partial R_{\text{IC}}}{\partial w}$ in the Newton iteration (12.22)

time interval $[0,\ T]$. Pseudo-spectral methods for the TPBVP problem (12.10) can then be framed as:

$$b\left(p(0,v),\, p(T,v),\, \bar{x}_0\right) = 0, \tag{12.23a}$$

$$\dot{p}\left(t_k, v\right) - \varphi(p(t_k,v), t_k) = 0, \qquad i = k, \ldots, n \tag{12.23b}$$

where $t_k \in [0,\ T]$, and the variables $v \in \mathbb{R}^{n_x \cdot n}$ hold the discretization of the continuous dynamics. Because they attempt to capture the state trajectories in a single function $p(t, v)$, with $t \in [0,\ T]$, the Newton iteration solving constraints (12.23) generally holds a dense Jacobian matrix, for which structure-exploiting linear algebra is generally inefficient.

12.6.4 Numerical solution of TPBVP with Path Constraints

In order to provide a fairly complete discussion on numerical solutions of the TPBVP problem for optimal control, we ought to consider the case of mixed path constraints arising in problem (12.1), resulting in a TPBVP of the form (12.4). As hinted in Section 12.3, the treatment of mixed path constraints in the context of indirect methods can be fairly involved.

The difficulty when solving the TPBVP (12.4) is very similar to the difficulty of solving the KKT conditions in the presence of inequality constraints, and stems from the non-smooth complementarity slackness condition (12.4e). Similarly to solving the non-smooth KKT conditions, we can consider here an approach similar to the Primal-Dual Interior-Point approach already detailed to solve the KKT conditions in the presence of inequality constraints, see Section 4.3.1. The Interior-point idea deployed on (12.4) yields the relaxation of the complementarity condition (12.4e) and the introduction of slack variables

$s(t)$ such that the following relaxed PMP conditions are numerically solved:

$$x(0) - \bar{x}_0 = 0, \tag{12.24a}$$
$$\lambda(T) - \nabla E(x(T)) = 0, \tag{12.24b}$$
$$\dot{x}(t) - \nabla_\lambda H^*(x(t), \lambda(t), \mu(t)) = 0, \quad t \in [0, T], \tag{12.24c}$$
$$\dot{\lambda}(t) + \nabla_x H^*(x(t), \lambda(t), \mu(t)) = 0, \quad t \in [0, T], \tag{12.24d}$$
$$h\left(x(t), u^\star_{\text{exp}}\right) + s(t) = 0, \quad t \in [0, T], \tag{12.24e}$$
$$s_i(t)\,\mu_i(t) - \tau = 0, \quad t \in [0, T], \quad i = 1, \ldots, n_h \tag{12.24f}$$

with the addition of the positivity conditions:

$$s(t) \geq 0, \qquad \mu(t) \geq 0 \quad t \in [0, T]. \tag{12.25}$$

One can observe that (12.24c)-(12.24f) is in fact an index-1 DAE (see Chapter 14), as the algebraic variables $s(t)$ and $\mu(t)$ can be eliminated using (12.24e) and (12.24f). In practice, problem (12.24) is best suited as it is for a numerical approach, as it allows to handle the positivity constraints (12.25) easily via taking the adequate step lengths in the Newton iterations.

The differential-algebraic conditions (12.24c)-(12.24f) can then be handled via a collocation method, yielding a large and sparse set of algebraic conditions that are simply added to the boundary conditions (12.24a)-(12.24b) to yield an algebraic system that we note as:

$$R_\tau(w) = 0, \tag{12.26}$$

where $w = (x, \lambda, \mu, s)$ gathers the discrete states x, costates λ, slack variables s and multipliers μ, discretized on the collocation time grid $t_{k,i}$ for $k = 0, \ldots, N-1$ and $i = 0, \ldots, d$. A prototype of interior-point algorithm then reads as follows.

Algorithm 12.8 (IP method for TPBVP with path constraints)**.**

Input: guess w, algorithmic parameters $\tau > 0$, $\gamma \in]0, 1[$, $\epsilon \in]0, 1[$, Tol > 0

while $\|R_\tau(w)\| \geq$ Tol **do**

$\Delta w = -\frac{\partial R_\tau(w)}{\partial w} R_\tau(w)$

Update $w = w + t\Delta w$, where $t \in]0, 1]$ ensures

$$s + t\Delta s \geq \epsilon s, \quad \mu + t\Delta\mu \geq \epsilon\mu$$

if $\|R_\tau(w)\|_X \leq 1$ **then** $\tau = \gamma\tau$

end while

where X is an ad-hoc norm on the residual R_τ. Let us consider the deployment of this Algorithm on the following example.

Example 12.9. We consider the optimal control problem (12.12) with the addition of simple mixed constraints in the form of state and input bounds:

$$\begin{aligned} \underset{x(.),\,u(.)}{\text{minimize}} \quad & \int_0^5 x_1(t)^2 + 10x_2(t)^2 + u(t)^2\,dt \\ \text{subject to} \quad & \dot{x}_1(t) = x_1(t)\,x_2(t) + u(t), && x_1(0) = 0, \\ & \dot{x}_2(t) = x_1(t), && x_2(0) = 1, \\ & u(t) \geq -1, \qquad x_1(t) \geq -0.6, && t \in [0,\ T]. \end{aligned} \tag{12.27}$$

with $T = 5$, which is similar to the optimal control problem treated in the previous examples, to the addition of the path constraint $x_1(t) \geq -0.6$ and $u(t) \geq -1$. We treat this problem using Algorithm 12.8. The resulting state-costate trajectory at the solution is displayed in Figure 12.10. The resulting optimal control input $u^\star_{\text{exp}}(x, \lambda, \mu)$, the slack variables s and the adjoint variables μ are displayed in Figure 12.11. The sparsity pattern of the Jacobian matrix $\frac{\partial R_\tau(w)}{\partial w}$ used in the Newton iterations in Algorithm 12.8 is illustrated in Figure 12.12.

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

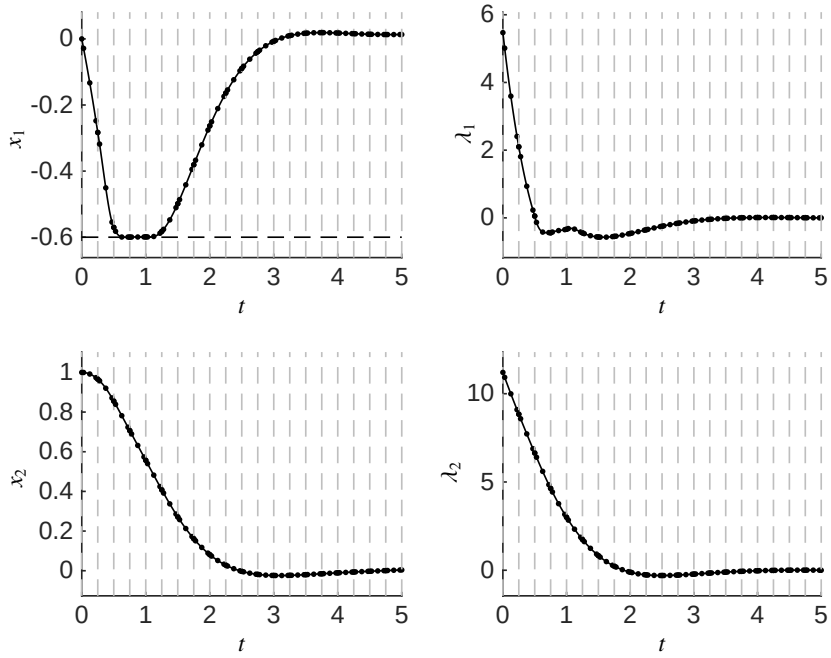


Figure 12.10 Illustration of the state and co-state trajectories for problem (12.12) using the orthogonal collocation approach with $N = 20$. The black curves report the state-costate trajectories at convergence. The discrete times t_k are depicted as grey dashed lines, the discrete state-costates on the time grid $t_{k,i}$ are depicted as dots.

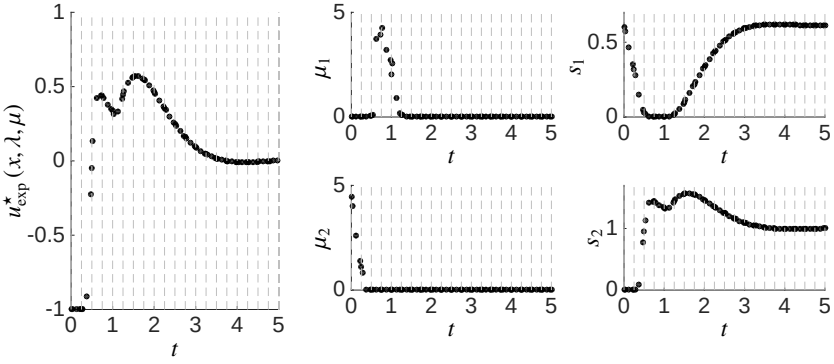


Figure 12.11 Illustration of the input $u^\star_{\text{exp}}(x, \lambda, \mu)$, slack s and adjoint variable μ at the solution of problem (12.12) using indirect collocation with an interior-point approach.

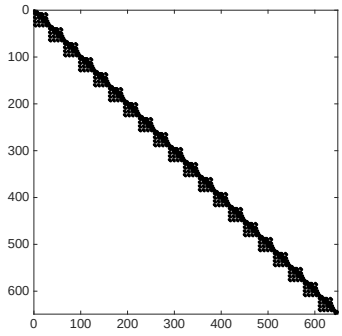


Figure 12.12 Illustration of the sparsity structure for the Jacobian $\frac{\partial R_\tau}{\partial w}$ in the Newton iteration deployed within Algorithm 12.8.

Exercises

12.1 In this exercise sheet, we regard the continuous time optimal control problem defined by:

$$\begin{aligned}
\underset{x(\cdot),\,u(\cdot)}{\text{minimize}} \quad & \int_0^\top u(t)^2 dt && \text{(12.28a)}\\
\text{subject to} \quad & x(0) = \bar{x}_0, && \text{(12.28b)}\\
& \dot{x}(t) = f(x(t), u(t)), \quad t \in [0, T], && \text{(12.28c)}\\
& X(T) = 0, && \text{(12.28d)}\\
& -u_{\max} \le u(t) \le u_{\max}, \quad t \in [0, T]. && \text{(12.28e)}
\end{aligned}$$

where the state is $x = (x_1, x_2)^\top$ and $\dot{x} = f(x, u)$ is given by:

$$f(x, u) = \begin{bmatrix} x_2(t) \\ -C \sin(x_1(t)/C) + u(t) \end{bmatrix}.$$

with $C := 180/\pi$.

We choose the initial value $\bar{x}_0 = (10, 0)^\top$, $T = 10$, and at first, we will leave away the control bound (12.28e) for first tasks.

(a) Considering that the Hamiltonian function for a general OCP with integral cost $L(x, u)$ is defined to be $H(x, \lambda, u) = L(x, u) + \lambda^\top f(x, u)$, and that in our case $L(x, u) = u^2$, write down explicitly the Hamiltonian function of the above optimal control problem as a function of the five variables $(x_1, x_2, \lambda_1, \lambda_2, u)$.

(b) Next, let us recall that the indirect approach eliminates the controls to obtain an explicit function $u^*(x, \lambda)$ that minimizes the Hamiltonian for

a given (x, λ). This optimal u^* can be computed by setting the gradient of the Hamiltonian w.r.t. u to zero, i.e. it must hold $\frac{\partial H}{\partial u}(x, \lambda, u^*) = 0$. Obtain an explicit expression for $u^*(x, \lambda)$.

(c) We will also need the derivatives w.r.t. x, so also calculate $\frac{\partial H}{\partial x_1}(x, \lambda, u)$ and $\frac{\partial H}{\partial x_2}(x, \lambda, u)$.

(d) Recall that the indirect approach formulates the Euler-Lagrange differential equations for the states and adjoints together. They are given by $\dot{x} = f(x, u^*(x, \lambda))$ and by $\dot{\lambda} = -\nabla_x H(x, \lambda, u^*(x, \lambda))$. For notational convenience, we define the vector $y = (x, \lambda)$ so that the Euler-Lagrange equation can be briefly written as the ODE $\dot{y} = \tilde{f}(y)$.
Collect all data from above to define explicitly the ODE right hand side $\tilde{f}$ as a function of the four components of the vector $y = (x_1, x_2, \lambda_1, \lambda_2)$

(e) The boundary value problem (BVP) that we now have to solve is given by

$$\begin{aligned} x(0) &= \bar{x}_0, \\ x(T) &= 0, \\ \dot{y}(t) &= \tilde{f}(y(t)), \quad t \in [0, T]. \end{aligned}$$

We will solve it by single shooting and a Newton procedure. The first step is to write an ODE simulator that for a given initial value $y_0 = (x_0, \lambda_0)$ simulates the ODE on the whole time horizon. Let us call the resulting trajectory $y(t; y_0)$, $t \in [0, T]$, and denote its terminal value by $y(T; y_0)$. Write a simulation routine that computes for given y_0 the value $y_N = y(T; y_0)$. Use a RK4 integrator with step size $\Delta t = 0.2$ and $N = 50$ time steps.

(f) Regard the initial value $y_0 = (x_0, \lambda_0)$. As the initial value for the states, x_0, is fixed to $\bar{x}_0$, we only need to find the right initial value for the adjoints, λ_0, i.e. we will fix $x_0 = \bar{x}_0$ and only keep $\lambda_0 \in \mathbb{R}^2$ as an unknown input to our simulator. Also, we have only to meet a terminal condition on $x(T)$, namely $x(T) = 0$, while $\lambda(T)$ is free. Thus, we are only interested in the map from λ_0 to $x(T)$, which we denote by $F(\lambda_0)$. Note that $F : \mathbb{R}^2 \to \mathbb{R}^2$. Using your simulator, write a MATLAB function `[x_end]=F(lambda_start)`.

(g) Add to your function functionality for plotting the trajectories of x_1, x_2, λ_1, λ_2. To do so, extend the output of your MATLAB simualtor to `[x_end,ytraj]=F(lambda_start)`.
For $\lambda_0 = 0$, call $F(\lambda_0)$ and plot the states and adjoints of your system. In this scenario, what is the numerical value of the final state $x(T) = F(0)$?

(h) The solution of the BVP is found if we have found λ_0^* such that $F(\lambda_0^*) = 0$. This system can be solved by Newton's method, that iterates, starting with some guess $\lambda_0^{[0]}$ (e.g. zero).

$$\lambda_0^{[k+1]} = \lambda_0^{[k]} - \left(\frac{\partial F}{\partial \lambda_0}(\lambda_0^{[k]})\right)^{-1} F(\lambda_0^{[k]})$$

First write a routine that computes the Jacobian $J_F(\lambda_0) = \frac{\partial F}{\partial \lambda_0}(\lambda_0)$ by finite differences using a perturbation $\delta = 10^{-4}$. Then implement a (full-step) Newton method that stops when $\|F(\lambda_0^{[k]})\| \leq \text{TOL}$ with $\text{TOL} = 10^{-3}$.

(i) For your obtained solution, plot the resulting state trajectories and verify by inspection that $x(T) = 0$.

(j) Using your function $u^*(x, \lambda)$, also plot the corresponding control trajectories $u(t)$.

(k) Add the control bounds (12.28e) with $u_{\max} = 3$. The only part in your whole algorithm that you need to change is the expression for $u^*(x, \lambda)$. The new constrained function

$$u^*_{\text{con}}(x, \lambda) = \arg\min_u H(x, \lambda, u) \quad \text{s.t.} \quad -u_{\max} \leq u \leq u_{\max},$$

is simply given by the "clipped" or "saturated" version of your old unconstrained function $u^*_{\text{unc}}(x, \lambda)$, namely by

$$u^*_{\text{con}}(x, \lambda) = \max\{-u_{\max}, \min\{u_{\max}, u^*_{\text{unc}}(x, \lambda)\}\}$$

Modify your differential equation $\tilde{f}$ by using this new expression for u^* and run your algorithm again. We remark that strictly speaking, the ODE right hand side is no longer differentiable so that the use of a RK4 is questionable as well as the computation of the Jacobian of F, but we cross our fingers and are happy that it works. For initialization of the Newton procedure, choose the multiplier λ_0^* from the unconstrained solution. In the solution, plot again the resulting trajectories for states, adjoints, and for $u(t)$, using of course your new function.

12.2 Consider the following two-point boundary-value problem, describing a person throwing a ball against a target:

$$\begin{bmatrix} \dot{p}_x \\ \dot{v}_x \\ \dot{p}_y \\ \dot{v}_y \end{bmatrix} = \begin{bmatrix} v_x \\ -\alpha\, v_x \sqrt{v_x^2 + v_y^2} \\ v_y \\ -\alpha\, v_y \sqrt{v_x^2 + v_y^2} - g_0 \end{bmatrix} \qquad \begin{cases} p_x(0) = 0, & p_x(T) = d \\ v_x(0) = v_{x,0}, & v_x(T) = v_{x,T} \\ p_y(0) = h, & p_y(T) = 0 \\ v_y(0) = v_{y,0}, & v_y(T) = v_{y,T} \end{cases}$$

The ball leaves the hand of the thrower with a velocity $(v_{x,0}, v_{y,0})$ a distance $h = 1.5$ m above the ground. It then follows an unguided trajectory determined by standard gravity $g_0 = 9.81$ m/s^2 and air friction $\alpha = 0.02$ hitting a target on the ground $d = 20$ m away after $T = 3$ s. The problem is to determine $(v_{x,0}, v_{y,0})$.

(a) Implement e RK4 integrator scheme with 20 steps to simulate the trajectory of the ball assuming assuming $v_{x,0} = v_{y,0} = 5$[m/s].

(b) Rewrite the integrator in order to get a function that given $v_0 := (v_{x,0}, v_{y,0})$ returns $p_T := (p_{x,T}, p_{y,T})$.

(c) Compute the Jacobian $\frac{\partial p_T}{\partial v_0}$ by finite differences.

(d) Write a full-step Newton method with 10 iterations to solve the root-finding problem:

$$p_T = F(v_0).$$

Verify the result by simulating the trajectory as in Task 4.1.

(e) Replace the quadratic friction terms $\alpha\, v_x \sqrt{v_x^2 + v_y^2}$ and $\alpha\, v_y \sqrt{v_x^2 + v_y^2}$ with the linear terms $\alpha\, v_x$ and $\alpha\, v_y$. How does this influence the number of Newton-iterations needed to solve the problem?

12.3 Regard again Exercise 8.7. We will solve now the modified version of that problem given by:

$$\begin{aligned} \underset{x,\,u}{\text{minimize}} \quad & \int_0^{\top} x_1(t)^2 + x_2(t)^2 + u(t)^2\, dt \\ \text{subject to} \quad & \dot{x}_1 = (1 - x_2^2)\, x_1 - x_2 + u, && x_1(0) = 0, \\ & \dot{x}_2 = x_1, && x_2(0) = 1, \\ & -1 \le\ u(t)\ \le 1, \end{aligned} \tag{12.29}$$

where $T = 10$. Notice the lack of state path constraints since they are difficult to handle with indirect methods.

(a) Introduce the costate $\lambda(t)$ and write down the Hamiltonian $H(x, \lambda, u)$ of (12.29):

(b) Use Pontryagin's maximum principle to derive an expression for the optimal control u^* as a function of x and λ. Note: $u(t)$ may only be a piecewise smooth function. Tip: How does u enter in the Hamiltonian?

(c) Derive the costate equations, i.e. $\dot{\lambda}(t) = \ldots$

(d) Derive the terminal conditions for the costate equations:

(e) Augment the original equations with the costate equation to form a two-point boundary-value problem (TPBVP) with four differential equations:

(f) **Casadi Part:** Solve the TPBVP with single-shooting. Use $[0, 0]$ as your initial guess for the initial costate. To integrate the system, our best chance is to use a variable stepsize integrator for stiff systems, such as the CVODES integrator from the SUNDIALS suite, available in CasADi. Note that the system is only piecewise smooth, which could potentially cause problems in the integrator, but we will ignore this and hope for the best. The resulting nonlinear system of equations is also challenging to solve, and in CasADi, our best bet is to use IPOPT with a dummy objective function ("minimize 0, subject to $g(x) = 0$"). We suggest allocating an instance of CVODES as follows:

- MATLAB

```
tf = SX.sym('tf');
dae = struct('x', aug, 'p', tf, 'ode',
  tf*augdot);
opts = struct('abstol', 1e-8, 'reltol',
  1e-8);
F = integrator('F', 'cvodes', dae, opts);
```

- Python

```
tf = SX.sym('tf')
dae = {'x':aug, 'p':tf, 'ode':tf*augdot}
opts = {'abstol':1e-8, 'reltol':1e-8}
F = integrator('F', 'cvodes', dae, opts)
```

where `aug` and `augdot` are expressions for the augmented state and augmented state derivative, respectively. We use a free parameter `tf` to scale the time horizon to $[0, t_f]$ instead of the default $[0, 1]$.

13

Direct Approaches to Continuous Optimal Control

Direct methods to continuous optimal control finitely parameterize the infinite dimensional decision variables, notably the controls $u(t)$, such that the original problem is approximated by a finite dimensional nonlinear program (NLP). This NLP can then be addressed by structure exploiting numerical NLP solution methods. For this reason, the approach is often characterized as "First discretize, then optimize." The direct approach connects easily to all optimization methods developed in the continuous optimization community, such as the methods described in Chapter 3. Most successful direct methods even parameterize the problem such that the resulting NLP has the structure of a discrete time optimal control problem, such that all the techniques and structures described in Chapters 7 and 7.3 are applicable. For this reason, the current chapter is kept relatively short; its major aim is to outline the major concepts and vocabulary in the field.

We start by describing *direct single shooting*, *direct multiple shooting*, and *direct collocation* and a variant *pseudospectral methods*. We also discuss how sensitivities are computed in the context of shooting methods. The optimization problem formulation we address in this chapter typically read as (but are not limited to):

$$
\begin{aligned}
\underset{x(.),\,u(.)}{\text{minimize}} \quad & \int_0^T L(x(t),u(t))\,dt \;+\; E\left(x(T)\right) \\
\text{subject to} \quad & x(0) - x_0 = 0, \quad \text{(initial value)}, \\
& \dot{x}(t) - f\left(x(t),u(t)\right) = 0, \quad \text{(system dynamics)}, \\
& h(x(t),u(t)) \leq 0, \quad \text{(path constraints)}, \\
& r\left(x(T)\right) \leq 0 \quad \text{(terminal constraints)}.
\end{aligned}
$$

For many OCPs, the system state derivatives $\dot{x}(t)$ are provided via an implicit function, or even via a Differential-Algebraic Equation (DAE). The methods

presented hereafter are applicable to all these cases with some minor modifications. The direct methods differ in how they transcribe this problem into a finite NLP. The optimal control problem above has a fixed initial value, which simplifies in particular the single shooting method, but all concepts can in a straightforward way be generalized to other OCP formulations with free initial values.

13.1 Direct Single Shooting

All shooting methods use an embedded ODE or DAE solver in order to eliminate the continuous time dynamic system. They do so by first parameterizing the control function $u(t)$, e.g. by polynomials, by piecewise constant functions, or, more generally, by piecewise polynomials. We denote the finite control parameters by the vector q, and the resulting control function by $u(t, q)$. The most widespread parameterization are piecewise constant controls, for which we choose a fixed time grid $0 = t_0 < t_1 < \ldots < t_N = T$, and N parameters $q_i \in \mathbb{R}^{n_u}$, $i = 0, \ldots, N-1$, and then we set

$$u(t, q) = q_k \quad \text{for} \quad t \in [t_k, t_{k+1}].$$

Thus, the dimension of the vector $q = (q_0, \ldots, q_{N-1})$ is of dimension Nn_u.

Single shooting is a *sequential approach* which has been earliest presented in [40, 61]. In single shooting, we regard the states $x(t)$ on $[0, T]$ as dependent variables that are obtained by a forward integration of the dynamic system, starting at x_0 and using the controls input $u(t, q)$. We denote the resulting trajectory as $x(t, q)$. In order to discretize inequality path constraints, we choose a grid, typically the same as for the control discretization, at which we check the inequalities. Thus, in single shooting, we transcribe the optimal control problem into the following NLP, that is visualized in Figure 13.1.

$$\begin{aligned}
&\underset{q \in \mathbb{R}^{Nn_u}}{\text{minimize}} && \int_0^T L(x(t,q), u(t,q))\,\mathrm{d}t \;+\; E\left(x(T,q)\right) && \\
&\text{subject to} && h(x(t_i,q), u(t_i,q)) \le 0, \quad i = 0, \ldots, N-1 && \text{(path constraints)},\\
& && r\left(x(T,q)\right) \le 0 && \text{(terminal constraints)}.
\end{aligned}$$

NLP structure in single shooting As the only variable of this NLP is the vector $q \in \mathbb{R}^{Nn_u}$ that influences nearly all problem functions, the above problem can usually be solved by a dense NLP solver in a black-box fashion. As the problem functions and their derivatives are expensive to compute, while a small

QP is cheap to solve, often Sequential Quadratic Programming (SQP) is used, e.g. the codes NPSOL or SNOPT. Let us first assume the Hessian needs not be computed but can be obtained e.g. by BFGS updates.

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

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

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

Example 13.1. Let us illustrate the single shooting method using the following

simple OCP:

$$\begin{aligned}
\underset{x(.),\,u(.)}{\text{minimize}} \quad & \int_0^5 x_1(t)^2 + 10x_2(t)^2 + u(t)^2\,dt \\
\text{subject to} \quad & \dot{x}_1(t) = x_1(t)\,x_2(t) + u(t), \qquad x_1(0) = 0, \\
& \dot{x}_2(t) = x_1(t), \qquad x_2(0) = 1, \\
& u(t) \geq -1, \qquad x_1(t) \geq -0.6, \qquad t \in [0,\ T],
\end{aligned} \tag{13.1}$$

which we used already in Example 12.9 of Section 12.6.4.

The resulting solution is illustrated in Figure 13.1, together with the sparsity patterns of the Jacobian of the inequality constraint function, i.e.

$$\frac{\partial}{\partial q} h(x(t_i, q), u(t_i, q)),$$

and the one of the Hessian of the Lagrange function.

Nonlinearity propagation in direct single shooting Unfortunately, direct single shooting often suffers from similar difficulties as the ones discussed in Section 12.6.1 for indirect single shooting. More specifically, when deploying single shooting in the context of direct optimal control a difficulty can arise from the nonlinearity of the "simulation" function $x(t, q)$ with respect to the control inputs q for a large simulation time t. We illustrate this problem using the following example:

$$\dot{x}_1 = 10\,(x_2 - x_1) \tag{13.2a}$$

$$\dot{x}_2 = x_1\,(q - x_3) - x_2 \tag{13.2b}$$

$$\dot{x}_3 = x_1 x_2 - 3x_3 \tag{13.2c}$$

where $x = (x_1, x_2, x_3) \in \mathbb{R}^3$ and $q \in \mathbb{R}$ is a constant control input. Note that the nonlinearities in this ODE result from apparently innocuous bilinear expressions. We are then interested in the relationship $q \to x(t, q)$ for different values of t. The initial conditions of the simulation were selected as $x(0) = (0, 0, 0)$ and $q \in [18,\ 38]$. The resulting relationship is displayed in Fig. 13.2. One can observe that while the relationship is not very nonlinear for small integration times t, it becomes extremely nonlinear for large times t, even though the ODE under consideration here appears simple and mildly nonlinear.

This example ought to warn the reader that the function $x(t, q)$ resulting from the simulation of nonlinear dynamics can be extremely nonlinear. As a result, functions such as the constraints and cost function in the NLP resulting form the discretization of an optimal control problem via single-shooting can be themselves extremely nonlinear functions of the input sequence q. Because

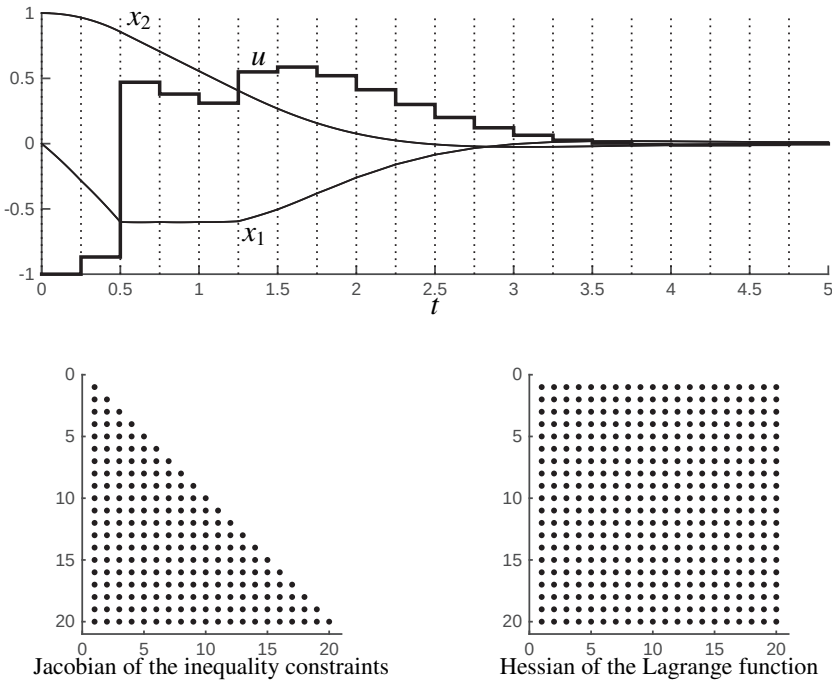


Figure 13.1 Solution to OCP (13.1) using a discretization based on single shooting, with $N = 20$ and using a 4-steps Runge-Kutta integrator of order 4. The upper graph reports the states and input trajectories. The lower graphs report the sparsity pattern of the Jacobian of the inequality constraints in the resulting NLP and the sparsity pattern of the Hessian of the Lagrange function.

most NLP solvers proceed to find a candidate solution via taking successive linearization of the KKT conditions of the problem at hand, the presence of very nonlinear functions in the NLP problem typically invalidates these approximations outside of a very small neighborhood of the linearization point, see Chapter 4 for more technical details on this issue.

These observations entails that in practice, when using single-shooting, a very good initial guess for q is often required. For many problems, such an initial guess is very difficult to construct. As in the context of indirect methods, these observations motivate the use of alternative transcription techniques.

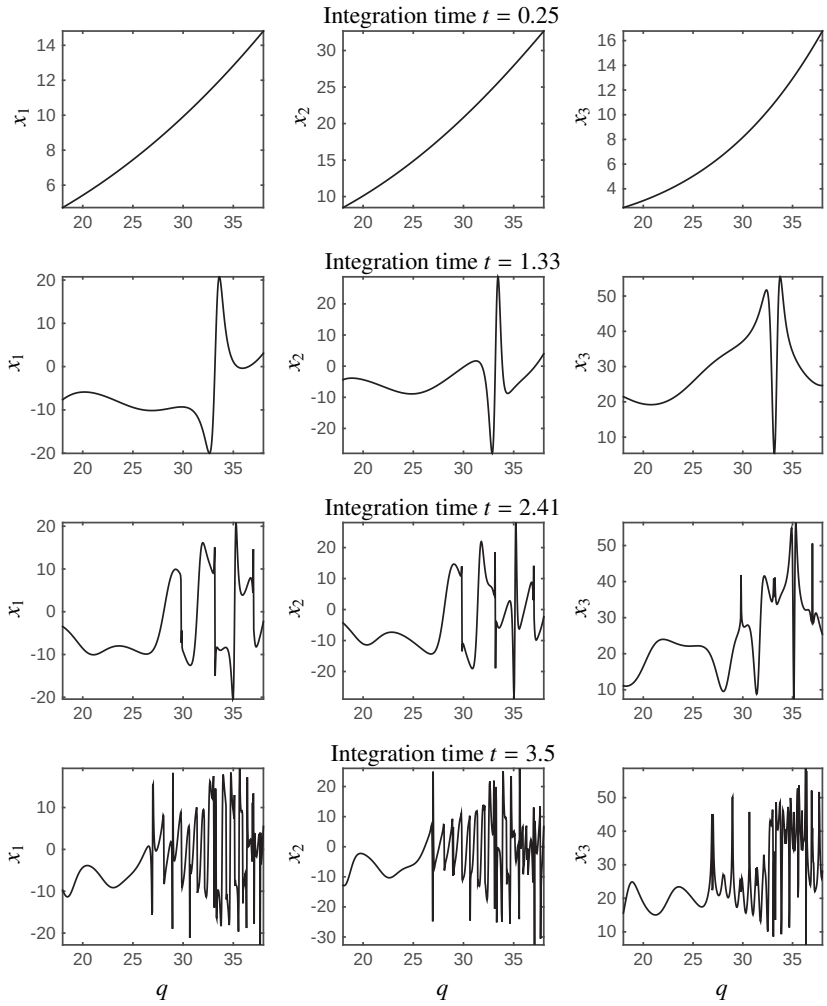


Figure 13.2 Illustration of the propagation of nonlinearities in the simple dynamic system (13.2). One can observe that for a short integration time $t = 0.25$ (first row), the relationship $q \rightarrow x(t, q)$ is close to linear. However, as the integration time increases to $t = 1.33, 2.41, 3.5$, the relationship $q \rightarrow x(t, q)$ becomes extremely nonlinear. While the effect of integration time is not necessarily as dramatic as for this specific example, large integration times yield strong nonlinear relationship $q \rightarrow x(t, q)$ for many nonlinear dynamics.

13.2 Direct Multiple Shooting

The direct multiple shooting method was originally developed by Bock and Plitt [17]. It follows similar ideas as the indirect multiple-shooting approach discussed in Section 12.6.1, but recast in the direct optimization framework, where the input profile is also discretized and part of the decision variables.

The idea behind the direct multiple-shooting approach stems from the observation that performing long integration of dynamics can be counterproductive for discretizing continuous optimal control problems into NLPs, and tackles the problem by limiting the integration over arbitrarily short time intervals. Direct multiple-shooting performs first a finite-dimensional discretization of the continuous control input $u(t)$, most commonly using a piecewise control discretization on a chosen time grid, exacly as we did in single shooting, i.e. we set

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

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

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

See Figure 13.3 for an illustration. Thus, we obtain trajectory pieces $x_i(t, s_i, q_i)$. Likewise, we numerically compute the integrals

$$l_i(s_i, q_i) := \int_{t_i}^{t_{i+1}} L\left(x_i(t, s_i, q_i), q_i\right) \mathrm{d}t.$$

The problem of piecing the trajectories together, i.e. ensuring the continuity condition $s_{i+1} = x_i(t_{i+1}, s_i, q_i)$ is left to the NLP solver. Finally, we choose a time grid on which the inequality path constraints are checked. It is common to choose the same time grid as for the discretization of the controls as piecewise constant, such that the constraints are checked based on the artificial initial values s_i. However, a much finer sampling is possible as well, provided that the numerical integrator building the simulations over the various time intervals $[t_k,\ t_{k+1}]$ report not only their final state $x_i(t_{i+1}, s_i, q_i)$, but also intermediate values. An integrator reporting the state (or some function of the state) over a refined or arbitrary time grid is sometimes labelled as *continuous-output* integrator.

The NLP arising from a discretization of an OCP based on multiple shooting

typically reads as:

$$\begin{aligned}
\underset{s,q}{\text{minimize}} \quad & \sum_{i=0}^{N-1} l_i(s_i, q_i) \;+\; E(s_N) \\
\text{subject to} \quad & x_0 - s_0 = 0, && \text{(initial value)}, \\
& x_i(t_{i+1}, s_i, q_i) - s_{i+1} = 0, \quad i = 0, \ldots, N-1 && \text{(continuity)}, \\
& h(s_i, q_i) \leq 0, \quad i = 0, \ldots, N && \text{(path constraints)}, \\
& r(s_N) \leq 0 && \text{(terminal constraints)}.
\end{aligned} \tag{13.3}$$

It is visualized in Figure 13.3. Let us illustrate the multiple shooting method using the OCP (13.1). Here the ordering of the equality constraints and variables is important in order to get structured sparsity patterns. In this example, the variables are ordered in time as:

$$s_{1,0},\ s_{2,0},\ q_0,\ s_{1,1},\ s_{2,1},\ q_1,\ \ldots,\ q_{N-1},\ s_{1,N},\ s_{2,N}$$

and the constraints are also ordered in time. The resulting solution is illustrated in Figure 13.4, together with the sparsity patterns of the Jacobian of the equality constraint function, and the one of the Hessian of the Lagrange function.

Note that by defining $f_i(s_i, q_i) := x_i(t_{i+1}, s_i, q_i)$, the continuity conditions can be interpreted as discrete time dynamic system $s_{i+1} = f_i(s_i, q_i)$ and the above optimal control problem has exactly the same structure as the discrete time optimal control problem (7.8) discussed in detail in Chapter 7.3. Most important, the sparsity structure arising from a discretization based on multiple-shooting (see Figure 13.4 for an illustration) ought to be exploited in the NLP solver.

Example 13.2. Let us tackle the OCP (13.1) of Example 13.1 via direct multiple-shooting. A 4-step RK4 integrator has been used here, deployed on $N = 20$ shooting intervals. The variables have been ordered as:

$$s_0,\ q_0,\ s_1,\ q_1, \ldots,\ s_{N-1},\ u_{N-1},\ s_N,$$

and the shooting constraints are also imposed time-wise.

The resulting solution is displayed in Figure 13.3, where one can observe the discrete state trajectories (black dots) at the discrete time instants $t_{0,\ldots,N}$ together with the simulations delivered by the integrators at the solution. One can also observe the very specific sparsity patterns of the Jacobian of the equality constraints and of the Hessian of the Lagrange function that arise from the direct multiple-shooting approach.

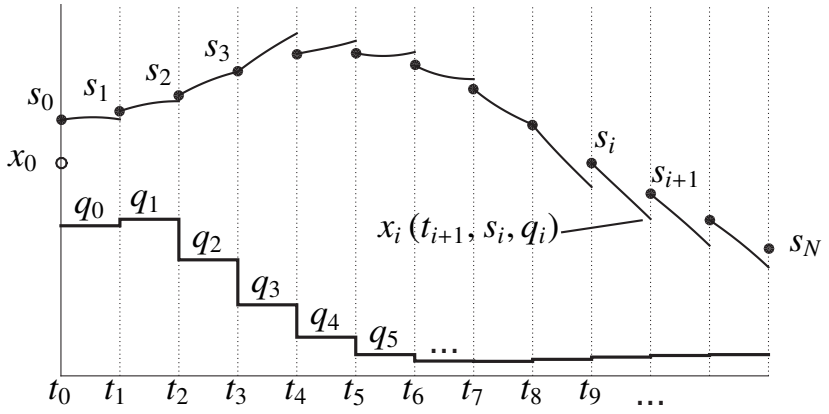


Figure 13.3 Illustration of the direct multiple shooting method. A piecewise-constant input profile parametrized by $q_{0,\ldots,N-1}$ is deployed on the time grid $t_{0,\ldots,N}$. The discrete states $s_{0,\ldots,N}$ act as "checkpoints" on the continuous state trajectories $x(t)$ at all discrete time points $t_{0,\ldots,N}$. Numerical integrators build the simulations $x_i\,(t, s_i, q_i)$ over each time interval $[t_i,\ t_{i+1}]$. The state trajectory held in the NLP solver becomes continuous only when the solution of the NLP is reached, where the continuity conditions $x_i\,(t_{i+1}, s_i, q_i) - s_{i+1}$ are enforced.

Remark on Schlöder's Reduction Trick: We point out here that the derivatives of the condensed QP could also directly be computed, using the reduced way, as explained as first variant in the context of single shooting. It exploits the fact that the initial value x_0 is fixed in the NMPC problem, changing the complexity of the derivative computations. It is only advantageous for large state but small control dimensions as it has a complexity of $N^2 n_u$. It was originally developed by Schlöder [63] in the context of Gauss-Newton methods and generalized to general SQP shooting methods by [62]. A further generalization of this approach to solve a "lifted" (larger, but equivalent) system with the same computational cost per iteration is the so called *lifted Newton method* [2] where also an analysis of the benefits of lifting is made.

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

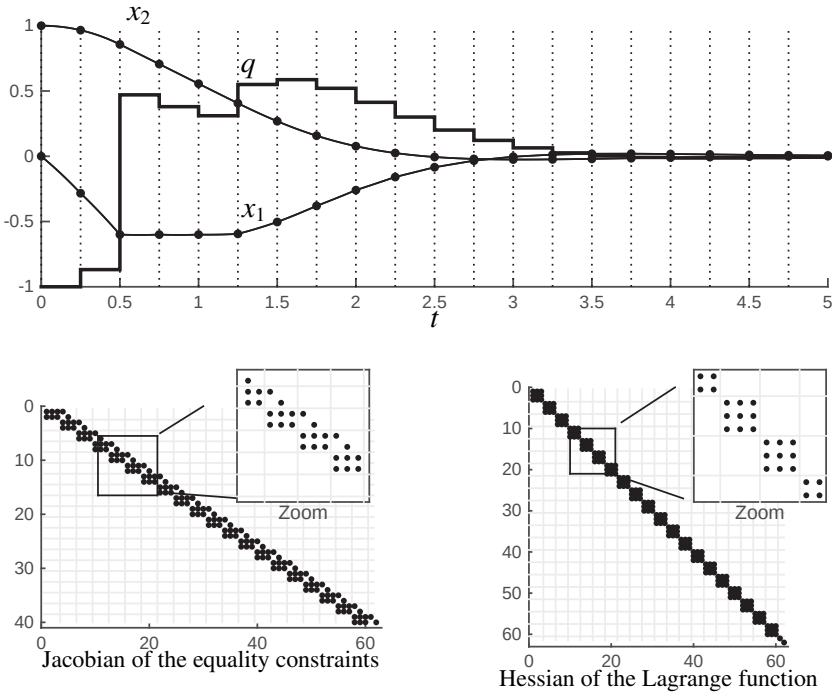


Figure 13.4 Solution to OCP (13.1) using a discretization based on multiple shooting, with $N = 20$ and using a 4-steps Runge-Kutta integrator of order 4. The upper graph reports the states and input trajectories at the solution, where the continuity condition holds. The lower graphs report the sparsity pattern of the Jacobian of the equality constraints in the resulting NLP and the sparsity pattern of the Hessian of the Lagrange function. The Hessian of the Lagrange function arising from multiple-shooting is block-diagonal, due to the separability of the Lagrange function. The Jacobian of the inequality constraints is diagonal in this example, and block-diagonal in general.

Finally, it is interesting to note that a direct multiple shooting algorithm can be made a single shooting algorithm easily: we only have to overwrite, before the derivative computation, the states s by the result of a forward simulation using the controls q obtained in the last Newton-type iteration. From this perspective, we can regard single shooting as a variant of multiple shooting where we perturb the result of each iteration by a "feasibility improvement" that makes all continuity conditions feasible by the forward simulation, implicitly giving priority to the control guess over the state guess [65].

13.3 Direct Collocation method

A third important class of direct methods are the so-called direct transcription methods, most notably *direct collocation*. The discretization method applied here is directly inspired from the collocation-based simulation already discussed in Chapter 10, Section 10.3, and very similar to the indirect collocation method discussed in Section 12.6.3.

Here we discretize the infinite OCP in both controls and states on a fixed and relatively fine grid t_k, with $k = 0, \ldots, N$. We denote the discrete states on the grid points t_k as s_k. We choose a parameterization of the controls on the same grid typically as piecewise constant, with control parameters q_k, which yields on each interval $[t_k, t_{k+1}]$ a constant control $u(t) = q_k$.

On each collocation interval $[t_k, t_{k+1}]$ a set of d collocation times $t_{k,i} \in [t_k, t_{k+1}]$ is chosen, with $i = 0, \ldots, d$. The trajectory of each state on the time interval $[t_k, t_{k+1}]$ is approximated by a polynomial $p_k(t, v_k) \in \mathbb{R}^{n_x}$ having the coefficients $v_k \in \mathbb{R}^{n_x(d+1)}$.

The collocation-based integration of the state dynamics on a time interval $[t_k, t_{k+1}]$ starting from the initial value s_k, as described in equation (10.5) hinges on solving the collocation equation:

$$c_k(v_k, s_k, q_k) = \begin{bmatrix} v_{k,0} - s_k \\ \dot{p}_k(t_{k,1}, v_k) - f(v_{k,1}, t_{k,1}, q_k) \\ \vdots \\ \dot{p}_k(t_{k,d}, v_k) - f(v_{k,d}, t_{k,d}, q_k) \end{bmatrix} = 0 \tag{13.4}$$

for the variables $v_{k,i} \in \mathbb{R}^{n_x}$, with $i = 0, \ldots, d$.

We now turn to building the NLP based on direct collocation. In addition to solving the collocation equations (13.4) for $k = 0, \ldots, N-1$, we also require continuity accross the interval boundaries, i.e. we require that

$$p_k(t_{k+1}, v_k) - s_{k+1} = 0$$

holds for $k = 0, \ldots, N$.

One finally ought to approximate the integrals $\int_{t_k}^{t_{k+1}} L(x, u)\mathrm{d}t$ on the collocation intervals by a quadrature formula using the same collocation points, which we denote by the a term $l_k(v_k, s_k, q_k)$. Path constraints can be enforced on the fine time grid $t_{k,i}$, though it is common to enforce them only on the interval boundaries t_k in order to reduce the amount of inequality constraints in the resulting NLP.

It is interesting to observe, that an arbitrary sampling of the state dynamics is possible by enforcing the path constraints at arbitrary time points t via the

interpolation $p_k(t, v_k)$. However, it is important to point out that the high integration order of collocation schemes holds only at the the main time grid t_k, such that interpolations at finer time grids, including the grid $t_{k,i}$, hold a lower numerical accuracy. In the following formulations, we will enforce the path constraints on the main time grid t_k.

Direct Collocation yields a large scale but sparse NLP, which can typically be written in the form

$$\begin{aligned}
\underset{v,\,s,\,q}{\text{minimize}} \quad & E(s_N) + \sum_{k=0}^{N-1} l_k(v_k, s_k, q_k) \\
\text{subject to} \quad & s_0 - x_0 = 0 && \text{(fixed initial value)}, \\
& c_k(v_k, s_k, q_k) = 0, \quad k = 0, \ldots, N-1 && \text{(collocation conditions)}, \\
& p_k(t_{k+1}, v_k) - s_{k+1} = 0, \quad k = 0, \ldots, N-1 && \text{(continuity conditions)}, \\
& h(s_k, q_k) \leq 0, \quad k = 0, \ldots, N-1, && \text{(path constraints)}, \\
& r(s_N) \leq 0 && \text{(terminal constraints)}.
\end{aligned}$$

One ought to observe that the discrete state variables s_k or alternatively the collocation variables $v_{k,0}$ can be eliminated via the first linear equality in each collocation equations $c_k(v_k, q_k, s_k) = 0$. It is in fact common to formulate the NLP arising from direct collocation without the s_k and enforcing continuity directly within the collocation equations. It then reads as follows:

$$\begin{aligned}
\underset{v,\,q}{\text{minimize}} \quad & E(v_{N,0}) + \sum_{k=0}^{N-1} l_k(v_k, q_k) \\
\text{subject to} \quad & v_{0,0} - x_0 = 0, \\
& \dot{p}_k(t_{k,i}, v_k) - f(v_{k,i}, q_k) = 0, \quad k = 0, \ldots, N-1,\ i = 1, \ldots, d, \\
& p_k(t_{k+1}, v_k) - v_{k+1,0} = 0, \quad k = 0, \ldots, N-1, \\
& h(v_{k,0}, q_k) \leq 0, \quad k = 0, \ldots, N-1, \\
& r(v_{N,0}) \leq 0.
\end{aligned} \tag{13.5}$$

We illustrate the variables and constraints of NLP (13.5) in Figure 13.5.

The direct collocation method offers two ways of increasing the numerical accuracy of the integration. We need to remember here that the integration error of a Gauss-Legendre collocation scheme is of $\mathcal{O}\left((t_{k+1} - t_k)^{2d}\right)$ (respectively $\mathcal{O}\left((t_{k+1} - t_k)^{2d-1}\right)$ for the Gauss-Radau collocation scheme). In order to gain accuracy, one can therefore increase d and thereby gain two orders in the integration error. Alternatively, one can reduce the size of the time intervals $[t_k,\, t_{k+1}]$ by e.g. a factor ξ and thereby reduce the order of the integration error

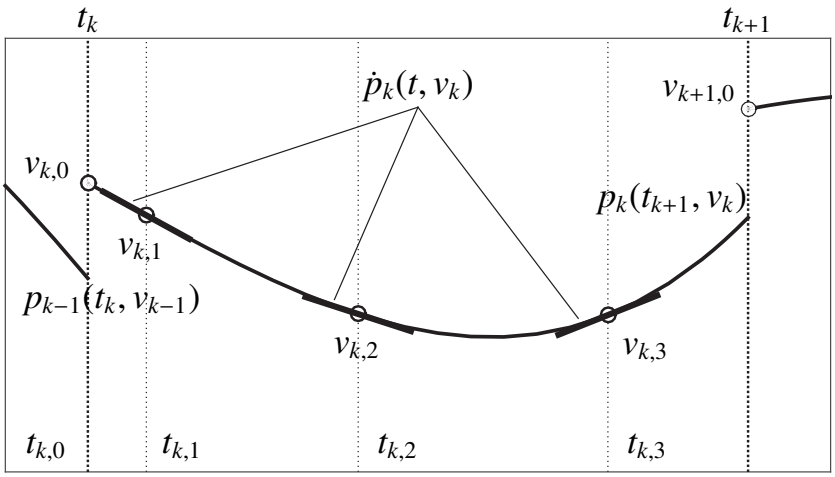


Figure 13.5 Illustration of the variables and constraints of NLP (13.5) for $d = 3$, and for one specific time interval $[t_k, t_{k+1}]$ before the constraints are fulfilled (early iteration). One can observe that the continuity conditions $p_k(t_{k+1}, v_k) - v_{k+1,0} = 0$ are not (yet) satisfied.

by a factor ξ^{2d} (respectively ξ^{2d-1} for the Gauss-Radau collocation scheme). However, numerical experiments often show that the conditioning of the linear algebra underlying the NLP resulting from direct collocation tends to worsen as d increases beyond relatively small orders. In practice, it often appears counterproductive to use $d > 4$ for complex optimal control problems.

One ought to observe here that discretizing an OCP using direct collocation allows for a fairly straightforward construction of the exact Hessian of the NLP. Indeed, one can observe that the nonlinear contributions to the constraints involved in the NLP arising from a discretization based on direct collocation are all explicitly given by the model continuous dynamics function f, the path constraints function h, and the terminal constraints function r. These functions are, in most OCPs, readily provided in their symbolic forms. It follows that assembling the Lagrange function and computing its first and second-order derivatives is fairly straightforward using any efficient symbolic computation tool such as e.g. AMPL or casADi.

Example 13.3. Let us tackle the OCP (13.1) of Example 13.1 via direct collocation. The direct collocation is implemented using a Gauss-Legendre direct collocation scheme with $d = 3$. Here again, the ordering of the equality con-

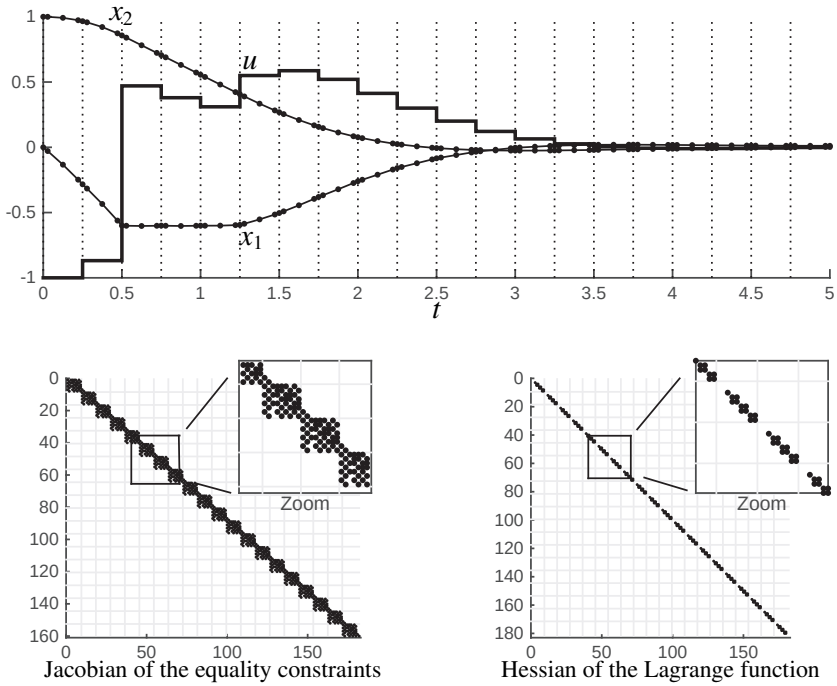


Figure 13.6 Solution to OCP (13.1) using a Gauss-Legendre direct collocation discretization scheme with $d = 3$, and $N = 20$. The upper graph reports the states and input trajectories. The collocated states $v_{k,i}$ are reported as the dots. The lower graphs report the sparsity pattern of the Jacobian of the equality constraints in the resulting NLP and the sparsity pattern of the Hessian of the Lagrange function. Observe that the Hessian is block diagonal, while the Jacobian has a block-diagonal pattern with some elements off the blocks corresponding to the continuity conditions. The Jacobian of the inequality constraints is diagonal in this example, and block-diagonal in general.

straints and variables is important in order to get structured sparsity patterns. In this example, the variables are ordered in time as:

$$v_{0,0},\ \ldots,\ v_{0,3},\ q_0,\ \ldots,\ v_{N-1,0},\ \ldots,\ v_{N-1,3},\ q_{N-1}$$

where $v_{k,i} \in \mathbb{R}^2$, and the constraints are also ordered in time. The resulting solution is illustrated in Figure 13.6, together with the sparsity patterns of the Jacobian of the equality constraint function, and the one of the Hessian of the Lagrange function.

The large NLP resulting from direct collocation need to be solved by struc-

ture exploiting solvers, and due to the fact that the problem functions are typically relatively cheap to evaluate compared to the cost of the linear algebra, nonlinear interior point methods are often the most efficient approach here. A widespread combination is to use collocation with IPOPT using the AMPL interface, or the casADi tool. It is interesting to note that, like in direct multiple shooting, the multipliers associated to the continuity conditions are again an approximation of the adjoint variables.

An interesting variant of orthogonal collocation methods that is often called the *pseudo-spectral optimal control method* uses only one collocation interval but on this interval it uses an extremely high order polynomial. State constraints are then typically enforced at all collocation points. Unfortunately, the constraints Jacobian and Lagrange Hessian matrices arising from the pseudo-spectral method are typically fairly dense and therefore more expensive to factorize than the ones arising in direct collocation.

Alternative input parametrization We have discussed to far the use of a piecewise-constant input parametrization in the context of direct methods. We ought to stress here that, while this choice is simple and very popular, it is also arbitrary. In fact, what qualifies direct methods is their use of a restriction of the continuous (and therefore ∞-dimensional) input profile $u(t)$ to a space of finite dimension, which can then be described via a finite set of numbers and therefore treated in the computer. In principle, any description of the continuous input $u(t)$ as a finite-dimensional object is possible, though some descriptions are less favorable than others. Indeed, it can e.g. be counterproductive to adopt an input discritization that destroys or degrades the sparsity patterns arising in the linear algebra of the various direct methods presented above. For this reason, it is typically preferable to adopt input discretizations that are "local" in time. Indeed, the sparsity patterns specific to the structure arising both in multiple-shooting and direct collocation hinge on the division of the overall time interval $[t_0,\, t_N]$ into the subintervals $[t_k,\, t_{k+1}]$, and the fact that the variables specific to one interval k, e.g. v_k, q_k in the direct collocation method have an impact only on the neighboring intervals ($k-1$ and $k+1$) via the continuity conditions. It would then be unwise to destroy this feature by using a discretization of the continuous input $u(t)$ where the input parameters q influence the input profile globally (i.e. at e.g. all time instants) such that an input parameter q_k would influence all intervals. This observation rules out the use of "global" input parametrizations such as e.g. parametrizing the inputs via a finite Fourier series or a polynomial basis over the whole interval $[t_0,\, t_N]$.

In the context of direct collocation, a fairly natural refinement of the continuous input parametrization consists in providing as many degrees of freedom

as the discretization of the optimal control problem allows. More specifically, one can readily observe that the standard piecewise input parametrization is enforced by construction of the collocation equations (13.4), where a single input value q_k is used on each collocation interval $[t_k,\ t_{k+1}]$. More degrees of freedom in the discretized input can, however, be readily added by allowing a different input $q_{k,i}$ at every collocation time point $t_{k,i}$, for $i = 1, \ldots, d$. The collocation equations for each interval $k = 0, \ldots, N-1$ then read as:

$$c_k(v_k, s_k, q_k) = \begin{bmatrix} v_{k,0} - s_k \\ \dot{p}_k(t_{k,i}, v_k) - f(v_{k,i}, t_{k,i}, q_{k,i}) \\ \vdots \\ \dot{p}_k(t_{k,d}, v_k) - f(v_{k,d}, t_{k,d}, q_{k,d}) \end{bmatrix} = 0. \tag{13.6}$$

and the NLP receives the decision variables

$$w = \{v_{0,0}, v_{0,1}, q_{0,1}, \ldots v_{0,d}, q_{0,d}, v_{1,0}, v_{1,1}, q_{1,1}, \ldots, v_{1,d}, q_{1,d}, \ldots\}.$$

It is important to observe here that the input is parametrized as $q_{k,i}$ with $k = 0, \ldots, N-1$ and $i = \underline{1}, \ldots, d$, i.e. *no degree of freedom* $q_{k,0}$ ought to be attributed to the discrete input on the first collocation times $t_{k,0}$, as only the continuity of the state trajectory is enforced on that collocation time.

13.4 A Classification of Direct Optimal Control Methods

It is an interesting exercise to try to classify Newton type optimal control algorithms, where we follow the presentation given in [30]. Let us have a look at how nonlinear optimal control algorithms perform their major algorithmic components, each of which comes in several variants:

(a) Treatment of Inequalities: Nonlinear IP vs. SQP.
(b) Nonlinear Iterations: Simultaneous vs. Sequential.
(c) Derivative Computations: Full vs. Reduced.
(d) Linear Algebra: Banded vs. Condensing.

In the last two of these categories, we observe that the first variants each exploit the specific structures of the simultaneous approach, while the second variant reduces the variable space to the one of the sequential approach. Note that reduced derivatives imply condensed linear algebra, so the combination [Reduced,Banded] is excluded. In the first category, we might sometimes distinguish two variants of SQP methods, depending on how they solve their underlying QP problems, via active set QP solvers (SQP-AS) or via interior point methods (SQP-IP).

Based on these four categories, each with two alternatives, and one combination excluded, we obtain 12 possible combinations. In these categories, the classical single shooting method [61] could be classified as [SQP, Sequential, Reduced] or as [SQP, Sequential, Full, Condensing] because some variants compute directly the reduced derivatives $\bar{R}^u$ in (**??**), while others compute first the stagewise derivative matrices A_i and B_i and condense then. Tenny's feasibility perturbed SQP method [65] could be classified as [SQP, Sequential, Full, Banded], and Bock's multiple shooting [17] as well as the classical reduced SQP collocation methods [66, 11, 10] as [SQP, Simultaneous, Full, Condensing]. The band structure exploiting SQP variants from Steinbach [64] and Franke [35] are classified as [SQP-IP, Simultaneous, Full, Banded], while the widely used interior point direct collocation method in conjunction with IPOPT by Biegler and Wächter [68] as [IP, Simultaneous, Full, Banded]. The reduced Gauss-Newton method of Schlöder [63] would here be classified as [SQP, Simultaneous, Reduced].

13.5 Direct Methods for Singular Optimal Control Problems

In this section, we want to discuss the implications of solving a singular OCP, as introduced in Section 12 using classical techniques from numerical optimal control. We will focus here on the classic choice of a piecewise constant input parametrization using a uniform, fixed time grid.

For the sake of simplicity, we will consider OCP having a scalar input $u \in \mathbb{R}$ with only input bounds:

$$\begin{aligned} \underset{x(\cdot),\, u(\cdot)}{\text{minimize}} \quad & \phi(x(t_\mathrm{f})) + \int_{t_0}^{t_\mathrm{f}} L\left(x\left(t\right), u\left(t\right)\right) dt \\ \text{subject to} \quad & \dot{x} = f\left(x, u\right), \quad x\left(t_0\right) = x_0, \\ & u_{\min} \leq u \leq u_{\max}. \end{aligned}$$

We will moreover consider dynamics that are affine in the input u:

$$\dot{x} = \varphi\left(x\right) + g\left(x\right) u \tag{13.7}$$

and a Lagrange term L that is either affine in input u. The Hamiltonian function reads as

$$H\left(x, \lambda, u\right) = L\left(x, u\right) + \lambda^\top f\left(x, u\right).$$

The PMP equations then read as:

$$\begin{aligned} u^\star(x,\lambda) = &\underset{u_{\min}\le u\le u_{\max}}{\operatorname{argmin}} \; L(x,u) + \lambda^\top f(x,u) \\ &\dot{x} = f\left(x,u^\star\right), \qquad\qquad x(t_0) = x_0, \\ &\dot{\lambda} = -\nabla_x H\left(x,u^\star,\lambda\right), \quad \lambda(t_\mathrm{f}) = \nabla_x \phi\left(x(t_\mathrm{f})\right). \end{aligned}$$

In particular, if L is a function of x only, $H_u(x,u,\lambda) = \lambda^\top g(x)$ such that the input profile reads as:

$$u^\star(x,\lambda) = \begin{cases} u_{\max} & \text{if} \quad \lambda^\top g(x) < 0 \\ u_{\min} & \text{if} \quad \lambda^\top g(x) > 0 \\ u_{\mathrm{sing}}(x,\lambda) & \text{if} \quad \lambda^\top g(x) = 0 \end{cases}. \tag{13.8}$$

As detailed in Section 12.2, the input $u_{\mathrm{sing}}(x,\lambda)$ is obtained via the time derivatives of H_u. For the simple case of a scalar input, it is interesting to note that for systems of the form (13.7), the time derivatives of H_u, up to where the dependence on the control input appears, are provided by the Lie derivatives over the vector fields f, g, i.e.,

$$\begin{aligned} \frac{\mathrm{d}^k}{\mathrm{d}t^k} H_u &= \lambda^\top \mathcal{L}_f^k g, \qquad k < 2\sigma \\ \frac{\mathrm{d}^{2\sigma}}{\mathrm{d}t^{2\sigma}} H_u &= \lambda^\top \mathcal{L}_f^{2\sigma} g + \lambda^\top \left[g, \mathcal{L}_f^{2\sigma-1} g\right] u, \end{aligned}$$

where σ is the degree of singularity of the OCP, and the Lie derivative operator $\mathcal{L}$ is defined in terms of the Lie bracket $[\cdot,\cdot]$, i.e. $\mathcal{L}_f g = [f,g]$. Here $\mathcal{L}^k$ stands for k applications of the Lie derivative operator on itself. The input u appearing at the differentiation 2σ can then be construed as a lack of commutativity of the vector field g with the k^{th}-order Lie derivative of the vector fields f, g. The singular input u_{sing} is provided by:

$$u_{\mathrm{sing}}(x,\lambda) = -\frac{\lambda^\top \mathcal{L}_f^k g}{\lambda^\top \mathcal{L}_g^k g}.$$

An interesting special case occurs when $2\sigma + 1$ equates the number of states present in the dynamics, then:

$$\left[H_u \quad \tfrac{\mathrm{d}}{\mathrm{d}t} H_u \quad \ldots \quad \tfrac{\mathrm{d}^{2\sigma}}{\mathrm{d}t^{2\sigma}} H_u\right] = \lambda^\top \left[g \quad \mathcal{L}_f g \quad \ldots \quad \mathcal{L}_f^{(2\sigma)} g + \left[g, \mathcal{L}_f^{2\sigma} g\right] u\right] = 0$$

uniquely defines the singular input u_{sing} via the condition:

$$\det\left(\left[g \quad \mathcal{L}_f g \quad \ldots \quad \mathcal{L}_f^{(2\sigma)} g + \left[g, \mathcal{L}_f^{2\sigma-1} g\right] u_{\mathrm{sing}}\right]\right) = 0.$$

The singular input then becomes a function of the states only, i.e. $u_{\mathrm{sing}} = u_{\mathrm{sing}}(x)$, and therefore becomes a pure feedback law. We turn next to analysing

the impact of using a piecewise-constant parametrization of the input profile $u(.)$.

13.5.1 Oscillations in singular optimal control solutions

It is important to observe here that the restriction of the input profile to a piecewise-constant input parametrization with a fixed time grid generally prevents the input profile from accurately capturing the switching times occurring in the optimal input profile $u^\star(.)$ given by (13.8). The optimal piecewise-constant input profile will then compensate for not switching at the exact time instant by "oscillating" around the singular arc. This phenomenon is arguably best explained in the light of the fundamental Lemma of the Calculus of Variations introduced in Section 12.5. For a piecewise-constant input parametrization, it states that the piecewise-constant optimal input profile

$$u^\star(t) = u_k^\star \quad \forall t \in [t_k,\, t_{k+1}]$$

satisfies:

$$\int_{t_k}^{t_{k+1}} H_u\left(x^\star(t), \lambda^\star(t)\right) \mathrm{d}t = 0, \quad \forall\, k \quad \text{such that} \quad u_{\min} < u_k^\star < u_{\max}. \tag{13.9}$$

For singular problems, H_u is "controlled" by u via $H_u^{(2\sigma)}$, i.e. a chain of 2σ integrators. The optimal input $u_k^\star$, when it is not in its bounds, is then determined by the initial conditions of this chain at t_k, i.e. by

$$H_u\left(x^\star(t_k), \lambda^\star(t_k)\right), \quad \ldots, \quad H_u^{(2\sigma-1)}\left(x^\star(t_k), \lambda^\star(t_k)\right).$$

via condition (13.9). Let us then define:

$$\dot{v} = \begin{bmatrix} H_u \\ H_u^{(1)} \\ \vdots \\ H_u^{(2\sigma)} \end{bmatrix} = Av + BH_u^{(2\sigma)}(x, \lambda, u)$$

where

$$A = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ & & \vdots & & \\ 0 & 0 & \ldots & 0 & 1 \\ 0 & 0 & \ldots & 0 & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}.$$

Whenever $u_{\min} < u_k^\star < u_{\max}$, the discrete optimal control input $u_k^\star$ on $[t_k,\, t_{k+1}]$ then enforces:

$$\int_{t_k}^{t_{k+1}} H_u\,(x,\lambda)\,d\tau = C\,(v\,(t_{k+1}) - v\,(t_k)) = 0 \tag{13.10}$$

where $C = \begin{bmatrix}1 & \ldots & 0 & 0\end{bmatrix}$. We have that:

$$v\,(t_{k+1}) = e^{A(t_{k+1}-t_k)}v(t_k) + \int_{t_k}^{t_{k+1}} e^{A(t_{k+1}-\tau)}BH_u^{(2\sigma)}\,(x,\lambda,u)\,\mathrm{d}\tau$$

such that:

$$\begin{aligned} C\,(v\,(t_{k+1}) - v\,(t_k)) &= C\left(e^{A(t_{k+1}-t_k)} - I\right)v(t_k) \\ &+ \int_{t_k}^{t_{k+1}} Ce^{A(t_{k+1}-\tau)}BH_u^{(2\sigma)}\,(x,\lambda,u)\,\mathrm{d}\tau = 0. \end{aligned} \tag{13.11}$$

Let us consider for the sake of simplicity that $H_u^{(2\sigma)} = u_k^\star$. In this special case, (13.11) defines the piecewise-constant optimal input in terms of a constant linear feedback law:

$$u_k^\star = -Kv\,(t_k)$$

such that the discrete dynamics of v is given by a constant transition matrix Φ

$$v\,(t_{k+1}) = \Phi v\,(t_k)\,.$$

It can be verified that Φ takes σ real, stable eigenvalues in $[-1,\, 0]$, which depend only on the degree of singularity σ of the OCP. These eigenvalues then yield a damped "oscillatory" trajectory for $v\,(t_k)$ in the direction of the corresponding eigenvectors. These oscillations translate directly into corresponding oscillations in the sequence of optimal control inputs $u_k^\star$. The oscillations observed in the piecewise-constant optimal input $u_k^\star$ when discretizing and solving a singular problem numerically is therefore not a numerical artefact, but a fundamental property of the piecewise-constant input parametrization of the input profile.

We illustrate these observations in the following example.

Example 13.4. Consider the linear-quadratic singular optimal control problem:

$$\begin{aligned} \underset{x\,(.)\,,\,u\,(.)}{\text{minimize}} \quad & \frac{1}{2}\int_0^1 x_1^2\,\mathrm{d}t \\ \text{subject to} \quad & \dot{x} = \begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}x + \begin{bmatrix}0\\1\end{bmatrix}u, \quad x\,(0) = \begin{bmatrix}0\\1\end{bmatrix}, \\ & -5 \le u \le 5. \end{aligned} \tag{13.12}$$

It can be verified that

$$H_u = \lambda_2, \qquad \frac{\mathrm{d}}{\mathrm{d}t}H_u = -\lambda_1, \qquad \frac{\mathrm{d}^2}{\mathrm{d}t^2}H_u = x_1, \qquad \frac{\mathrm{d}^3}{\mathrm{d}t^3}H_u = x_2, \qquad \frac{\mathrm{d}^4}{\mathrm{d}t^4}H_u = u.$$

The optimal input profile then reads as:

$$u^\star(t) = \begin{cases} u_{\min} & \text{if} \quad \lambda_2^\star(t) > 0 \\ u_{\max} & \text{if} \quad \lambda_2^\star(t) < 0 \\ 0 & \text{if} \quad x = \lambda = 0 \end{cases}$$

i.e. the solution is bang-bang until the states and co-states reach the zero manifold. We are interested in studying the solution to problem (13.12) when the optimal input profile $u^\star(t)$ is approximated by a piecewise-constant profile $u_0^\star, ..., u_{N-1}^\star$. Direct collocation was used to tackle (13.12), using Legendre polynomials with an integration order of 10. The NLP was solved using an interior-point method converged to machine precision.

The resulting optimal control solution $u_k^\star$ is reported in Fig. 13.7, together with the continuous optimal input profile $u^\star(\cdot)$. One can observe oscillations in the piecewise-constant input after the last switching time at 0.578 s, which is typical of singular optimal control problems. The corresponding state trajectories are reported in Figure 13.8. The trajectories of $v(t)$ for this problem are reported in Figure 13.9, for both the continuous optimal input profile $u^\star(\cdot)$ and its piecewise-constant input parametrization $u_k^\star$. One can observe in the upper-left graph that the optimality condition (13.9) is satisfied by the solution $u_k^\star$, which requires an oscillation in $v(t_k)$. Indeed, the stable eigenvalues of matrix Φ for problem (13.12) read as $-0.0431, -0.4306$. The oscillation of $v(t_k)$ in turn require a corresponding oscillation in $u_k^\star$. These oscillations are also observed in the states and co-states trajectories, which for problem (13.12) match H_u and its time derivatives.

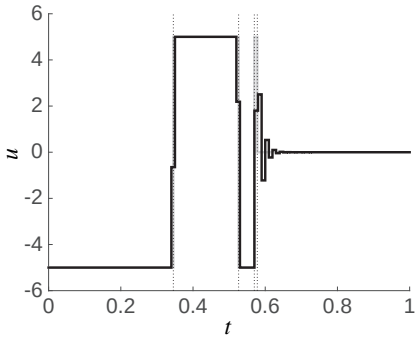


Figure 13.7 Optimal input profile (in grey) and piecewise-constant input profile obtained via direct collocation (in black) for problem (13.12), where the input is discretized as piecewise-constant over $N = 100$ uniform time intervals. The vertical dotted lines report the optimal switching times between $u = u_{\min}$, $u = u_{\max}$ and $u = 0$. The "oscillation" of the optimal piecewise-constant input is symptomatic of singular problems when the discretization of the input profile does not allow for capturing arbitrary switching times.

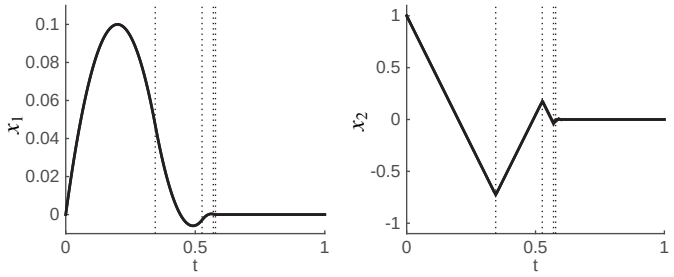


Figure 13.8 Optimal state trajectories for problem (13.12). Even though the optimal input obtained from direct collocation is significantly different from the optimal one, the respective resulting state trajectories are indistinguishable.

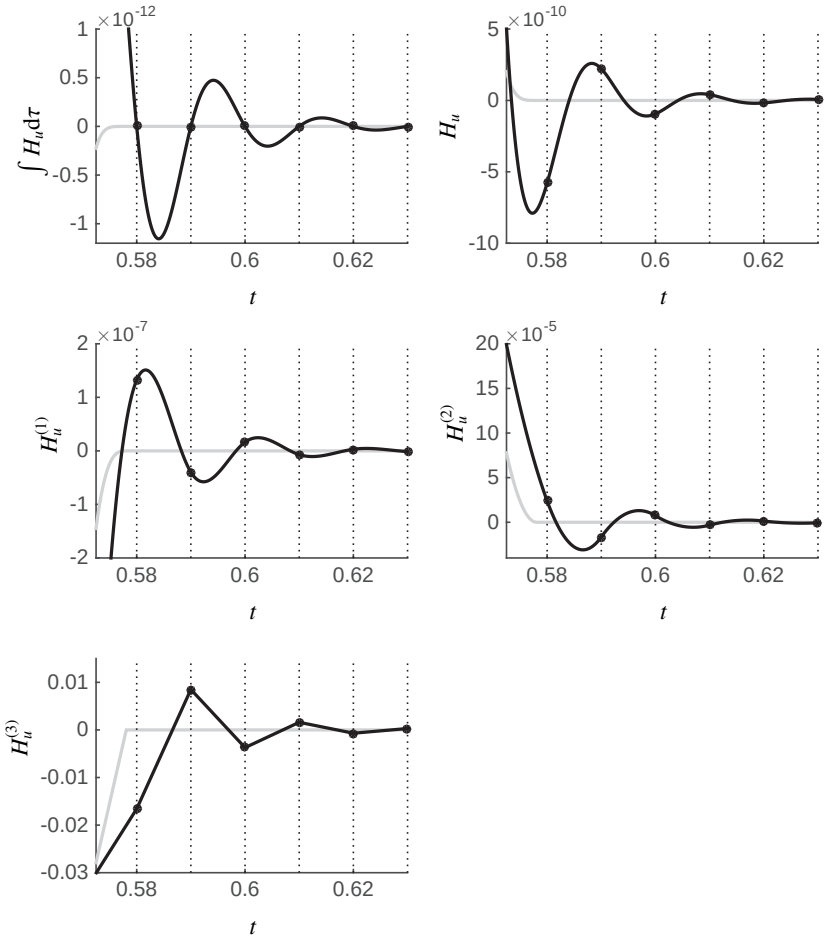


Figure 13.9 $\int H_u \mathrm{d}\tau$ and its time derivatives in the time interval [0.5725, 0.63]. The "oscillations" in the input profile obtained from direct collocation can be easily understood in the light of condition (13.10). The piecewise-constant optimal inputs $u_k^\star$ enforce $\int_{t_k}^{t_{k+1}} H_u \mathrm{d}\tau$ for all k where the input bounds are not active (see upper-left graph), which yield damped oscillations in $H_u^{(k)}\left(x^\star\left(t_k\right), \lambda^\star\left(t_k\right)\right)$.

Exercises

13.1 Let's regard again the OCP defined in Exercises 12.3 and 8.7:

$$\begin{aligned} \underset{x,\,u}{\text{minimize}} \quad & \int_0^\top x_1(t)^2 + x_2(t)^2 + u(t)^2\, dt \\ \text{subject to} \quad & \dot{x}_1 = (1 - x_2^2)\, x_1 - x_2 + u, \qquad x_1(0) = 0, \\ & \dot{x}_2 = x_1, \qquad\qquad\qquad\qquad x_2(0) = 1, \\ & -1 \leq\ u(t)\ \leq 1, \end{aligned} \tag{13.13}$$

where $T = 10$ as earlier.

(a) Implement a RK4 integrator for the system dynamics.

(b) Use the integrator to create a function $x(x_0, u, T)$ that simulates the system in a time interval T, and where x_0 is the initial state and u a set of piecewise constant controls defined in a time grid with constant step size Δt.

(c) Use the previous function to solve the OCP using single shooting and $N = 101$. Approximate the cost function using the trapezoidal rule between the nodes where u is defined. Use *fmincon* from MATLAB to solve the NLP.

(d) Modify the script to solve the same problem using direct multiple shooting. The control parametrization and the definition of the integrator can remain the same.

(e) How did the change from direct single shooting to direct multiple shooting influence the following features?

- The number of iterations.
- The number of nonzeros in the Jacobian of the constraints.
- The number of nonzeros in the Hessian of the Lagrangian.
- The total solution time.

13.2 In the previous problem, we solved the NLP using *fmincon*. In the following, we will write our own simple SQP code to solve (13.13). As a quick reminder, SQP employs a sequence of quadratic approximations to solve the NLP and solves these with a QP solver. For an NLP of the

form:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x_{\text{lb}} \leq x, \\ & x_{\text{ub}} \geq x, \\ & g(x) = 0, \end{aligned}$$

these quadratic approximations take the form:

$$\begin{aligned} \underset{\Delta x}{\text{minimize}} \quad & \frac{1}{2} \Delta x^\top \nabla_x^2 \mathcal{L}(x^{(k)}, \lambda^{(k)}) \Delta x + \nabla_x f(x^{(k)})^\top \Delta x \\ \text{subject to} \quad & x_{\text{lb}} - x^{(k)} \leq \Delta x, \\ & x_{\text{ub}} - x^{(k)} \geq \Delta x, \\ & g(x^{(k)}) + \frac{\partial g}{\partial x}(x^{(k)}) \Delta x = 0. \end{aligned} \tag{13.14}$$

where $(x^{(k)}, \lambda^{(k)})$ is a guess of the primal-dual solution to (13.14) and $\mathcal{L}(x, \lambda) = f(x) + \lambda^\top g(x)$ is the Lagrangian. The solution of this QP gives the step in Δx and a new approximation of the multipliers λ.

(a) For problems with a quadratic objective function $f(x) = \frac{1}{2} \|F(x)\|_2^2$, like the NLPs arrising from both direct single shooting and direct multiple shooting transcription of (13.13), a popular variant is to use a *Gauss-Newton* approximation of the Hessian of the Lagrangian:

$$\nabla_x^2 \mathcal{L}\left(x^{(k)}, \lambda^{(k)}\right) \approx \frac{\partial F}{\partial x}\left(x^{(k)}\right)^\top \frac{\partial F}{\partial x}\left(x^{(k)}\right)$$

and $\nabla_x f(x) = \frac{\partial F}{\partial x}\left(x^{(k)}\right)^\top F\left(x^{(k)}\right)$.
What are the main advantages and disadvantages of such an approximation?

(b) Implement a Gauss-Newton method to solve the problem. Use algorithmic differentiation or finite differences to calculate $\frac{\partial F}{\partial x}$ and $\frac{\partial g}{\partial x}$ and solve the QP subproblem using the `quadprog` tool from MATLAB.

13.3 Jebediah Kerman is an astronaut that has gone for an aerospace walk and lost track of time. He can't remember when atmospheric re-entry is scheduled, but he believes it is very soon. He needs to get back to his spaceship as quickly as possible. He has mass 30kg including space suit but not including fuel. He is currently carrying 10kg of fuel. He is 50m away from his ship, with zero relative velocity. He wants to return to the ship as quickly as possible (to have equal position and zero relative velocity), while still conserving 4kg of fuel for emergencies.

Jebediah ca be modeled as having three states: position p, velocity v, and fuel mass m_F. Moreover, the space suit has a rocket booster (control u) which can fire forwards or reverse. As a result, the equation of motion of his body are:

$$\frac{d}{dt}\begin{pmatrix} p \\ v \\ m_F \end{pmatrix} = \begin{pmatrix} v \\ u/\left(30 + m_F\right) \\ -u^2 \end{pmatrix} \tag{13.15}$$

(a) Write down the continuous time optimal control problem with a minimum time objective T.

(b) Discretize this problem using direct multiple shooting, and write down the NLP. Use the shooting function $x_{k+1} = f_{\text{rk4}}(x_k, u_k, \Delta t)$ with $\Delta t = \frac{T}{N}$ being an optimization variable, so your vector of optimization variables is $y = [x_0, u_0, \ldots, u_{N-1}, x_N, \Delta t]^\top$.

(c) Using an RK4 integrator, implement this NLP with `fmincon` and solve it. Use $N = 40$ as the number of control intervals and think of a proper initialization. Plot p, v, m_F, and u versus time.

(d) Make a sketch of the Hessian of the Lagrange function. You will see that the Hessian is sparse but not block diagonal. Can you find a problem reformulation with a block diagonal Hessian? Make a sketch of the new Hessian.

Hint: Introduce multiple copies of you timestep Δt and make it a pseudo state.

13.4 **CasADi Exercise:** Consider the following continuous-time infinite dimensional problem:

$$\begin{aligned} \underset{x,\,u}{\text{minimize}} \quad & \int_0^T x(t)^2 + u(t)^2 \, dt \\ \text{subject to} \quad & \dot{x} = (1 + x)\, x + u, \\ & |u(t)| \leq 0.075, \\ & x(0) = \bar{x}_0, \\ & x(T) = 0, \end{aligned}$$

where $u \in \mathbb{R}$ is the control input and $x \in \mathbb{R}$ is the state of the system, $T = 3$ and $\bar{x}_0 = 0.05$. The above formulation can be discretized by integrating the dynamics of the system over a fixed grid with $N+1$ nodes

leading to the finite-dimensional discrete-time problem

$$\begin{aligned}
\underset{\substack{x_0,\cdots,x_N\\ u_0,\cdots,u_{N-1}}}{\text{minimize}} \quad & h\sum_{i=0}^{N-1}(x_i^2+u_i^2)+x_N^2 \\
\text{subject to} \quad & x_{i+1}=f(x_i,u_i), \quad i=0,\cdots,N-1, \\
& |u_i|\le 0.075, \qquad i=0,\cdots,N-1, \\
& x_0=\bar{x}_0, \\
& x_N=0
\end{aligned}$$

where f describes the discretized dynamics obtained using an integration scheme, $h := \frac{T}{N}$, x_i and u_i refer to the evaluation of state and control trajectories respectively. Furthermore, we can transform the above discrete OCP into the single shooting scheme as:

$$\begin{aligned}
\underset{u}{\text{minimize}} \quad & \Phi(u) \\
\text{subject to} \quad & |u_i|\le 0.075, \quad i=0,\cdots,N-1, \\
& x_N(u)=0
\end{aligned}$$

where $\Phi(u) := h(\bar{x}_0^2+u_0^2+f(\bar{x}_0,u_0)^2+u_1^2+\cdots)$.

(a) Implement a CasADi `Function` f that takes as argument the states x and input u and returns the ODE right-hand-side $\dot{x}$.

(b) Divide the time horizon into $N = 30$ equidistant control intervals, then use the RK4 scheme to define the discrete-time dynamics as a CasADi function. This function should take $x(t_i)$ and u_i as inputs and return $x(t_{i+1})$. The key lines of the integrator implementation could look like this:

```
out  = f({X,U});
k1   = out{1};
% ...
X = X + h/6*(k1   + 2*k2   + 2*k3   + k4);
```

(c) Formulate the direct single shooting NLP and solve it with IPOPT. Note that the NLP should have N degrees of freedom, so start by defining a variable $u \in \mathbb{R}^N$:

```
u = SX.sym('u',N);
```

The key lines of the NLP formulation could look like this:

```
X = X0;
for i = 1:N
  out = F({X,v(i)});
  X = out{1};
  J = J + X(1)^2 + u(i)^2;
end
```

(d) Modify the script to so that it implements the direct multiple shooting method. The control parametrization and the definition of the integrator can remain the same. Tip: Start by replacing the line:

```
nv = N
```

with

```
nv = 1*N + 2*(N+1)
```

Make sure that you get the same solution.

(e) Compare the IPOPT output for both scripts. How did the change from direct single shooting to direct multiple shooting influence:

- The number of iterations
- The number of nonzeros in the Jacobian of the constraints
- The number of nonzeros in the Hessian of the Lagrangian
- The total solution time

(f) Introduce the additional path constraints $x_i \geq 0.05$, $i = 15, \cdots, 17$. Change your scripts to solve the modified problem.

(g) Replace the dynamics in the NLP from the previous task with their linearization at the origin $x_0 = 0$. Compute the optimal solution and apply it to the original system. Are the path constraints satisfied? Is there a neighborhood of the origin where this linearized optimal control problem will provide a feasible solution?

13.5 **CasADi Exercise:** Consider the following simple OCP for controlling a Van-der-Pol oscillator:

$$\begin{aligned} \underset{x,u}{\text{minimize}} \quad & \int_0^T x_1(t)^2 + x_2(t)^2 + u(t)^2 \, dt \\ \text{subject to} \quad & \dot{x}_1 = (1 - x_2^2)\, x_1 - x_2 + u, \qquad x_1(0) = 0, \\ & \dot{x}_2 = x_1, \qquad\qquad\qquad\qquad x_2(0) = 1, \\ & -1 \leq u(t) \leq 1 \end{aligned}$$

where $T = 10$.

(a) Implement a CasADi `Function` $f : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2 \times \mathbb{R}$ that takes the states x and input u and returns the ODE right-hand-side $\dot{x}$ and the Lagrange objective term L.

(b) Divide the time horizon into $N = 20$ equidistant intervals, $[t_k, t_{k+1}]$, $k = 0, \ldots, N-1$ and assume a constant control u_k on each interval. Then take $M = 4$ steps with a RK4 scheme to define the discrete-time dynamics as a `Function` $F : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2 \times \mathbb{R}$. F should take $x(t_k)$ and u_k and return $x(t_{k+1})$ and $J_k = \int_{t_k}^{t_{k+1}} L(x, u_k)$, the contribution to the objective from interval k. Evaluate the integrator with $x(t_k) = [0.2, 0.3]$ and $u_k = 0.4$.

(c) Formulate the direct single shooting NLP and solve it with IPOPT. Construct the NLP variables step-by-step starting with empty list:

```
w = []
lbw = []
ubw = []
```

Plot the results.

(d) Modify the script to so that it implements the direct multiple shooting method. The control parametrization and the definition of the integrator should remain the same. Introduce NLP variables corresponding to the state for all discrete time points, including $k = 0$.

(e) Compare the IPOPT output for both scripts. How did the change from direct single shooting to direct multiple shooting influence:

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

(f) Introduce the additional constraint $x_1(t) \geq -0.25$. You only need to enforce this path constraint at the end of each control interval. Modify your scripts to solve the modified problem with direct multiple shooting (easy) and direct single shooting (more tricky).

13.6 **CasADi Exercise:** Collocation, in its most basic sense, refers to a way of solving initial-value problems by approximating the state trajectory with piecewise polynomials. For each step of the integrator, corresponding to an interval of time, we choose the coefficients of these polynomials to ensure that the ODE becomes exactly satisfied at a given set of time points. The time points, in turn, are chosen to get the highest possible acuracy and, possibly, to make sure that the dynamics be satisfied at the beginning and/or end of the time interval. In the following, we will

choose the *Legendre points* of order $d = 3$:

$$\tau = [0, 0.112702, 0.500000, 0.887298] \tag{13.16}$$

where we have assumed that the time interval is $[0, 1]$.

Using these time points, we define a Lagrangian polynomial basis for our polynomials:

$$L_j(\tau) = \prod_{r=0,\, r\neq j}^{d} \frac{\tau - \tau_r}{\tau_j - \tau_r} \tag{13.17}$$

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

$$\tilde{x}_k(t) = \sum_{r=0}^{d} L_r\left(\frac{t - t_k}{h}\right) x_{k,r} \tag{13.18}$$

By differentiation, we get an approximation of the time derivative at each collocation point:

$$\dot{\tilde{x}}_k(t_{k,j}) = \frac{1}{h} \sum_{r=0}^{d} \dot{L}_r(\tau_j)\, x_{k,r} := \frac{1}{h} \sum_{r=0}^{d} C_{r,j}\, x_{k,r} \tag{13.19}$$

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

$$\tilde{x}_{k+1,0} = \sum_{r=0}^{d} L_r(1)\, x_{k,r} := \sum_{r=0}^{d} D_r\, x_{k,r} \tag{13.20}$$

We can also integrate our approximation over the interval, giving a formula for *quadratures*:

$$\int_{t_k}^{t_{k+1}} \tilde{x}_k(t)\, dt = h \sum_{r=0}^{d} \int_0^1 L_r(t)\, dt\, x_{k,r} := h \sum_{r=1}^{d} B_r\, x_{k,r} \tag{13.21}$$

(a) Download `collocation.m` (MATLAB) or `collocation.py` (Python) from the course website containing an implementation of the above collocation scheme. Go through the code and make sure you understand it well. Use the code to to reproduce the result from the second task of Exercise 13.5.

(b) Replace the RK4 integrator in the direct multiple shooting implementation from Exercise 13.5 with the above collocation integrator. Make sure that you get the same results as before.

(c) Instead of letting the rootfinder solve the collocation equations, augment the NLP variable and constraint vectors with additional degrees of freedom corresponding to the state at the collocation points and let the NLP solver also solve the integration problem. For simplicity, only consider a single collocation finite element per control interval. Compare the solution time and number of nonzeros in the Jacobian and Hessian matrices with the direct multiple shooting method.

(d) Form the Jacobian of the constraints and inspect the sparsity pattern using MATLAB's or SciPy's `spy` command. Repeat the same for the Hessian of the Lagrangian function $L(x, \lambda) = J(x) + \lambda^{\mathrm{T}} g(x)$.

14

Optimal Control with Differential-Algebraic Equations

So far we have regarded optimal control problems based on model dynamics in their simplest explicit-ODE form:

$$\dot{x}(t) = f\left(x(t), u(t)\right).$$

This form of model for dynamic systems tend to arise naturally from the first-principle modelling approaches standardly taught and used by engineers. As a result, most continuous dynamic systems are described via explicit ODEs. It is a common but less widespread knowledge that for a large number of applications, building a dynamic model in the form of explicit ODEs can be significantly more involved and yield dramatically more complex model equations than via alternative model forms. Before laying down some theory, let us start with a simple illustrative example that we will use throughout this chapter.

Consider a mass m attached to a fixed point via a rigid link of length L for which one wants to develop a dynamic model. A classic modelling approach is to describe the mass via two angles (azimuth and elevation of the mass), which yields an explicit ODE. The alternative model construction we will consider here describes the system via the cartesian coordinates $p \in \mathbb{R}^3$ of the mass in a fixed, inertial reference frame E positioned at the attachment point of the mass, see Figure 14.1. The rod maintains the mass at a distance L of its attachment point by applying a force on the mass along its axis, i.e. having the support vector p. We will then describe the force of the rod as $F_{\text{rod}} = -zp$, where $z \in \mathbb{R}$ is a variable that adjusts the force magnitude to maintain the mass on a sphere of radius L, i.e. such that the condition $p^\top p - L^2 = 0$ holds at all time. The model of the system can then takes a very simple form:

$$m\ddot{p} = u - zp + mgE_3, \qquad \frac{1}{2}\left(p^\top p - L^2\right) = 0. \tag{14.1}$$

where $E_3^\top = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$. One can readily observe here that the model equation

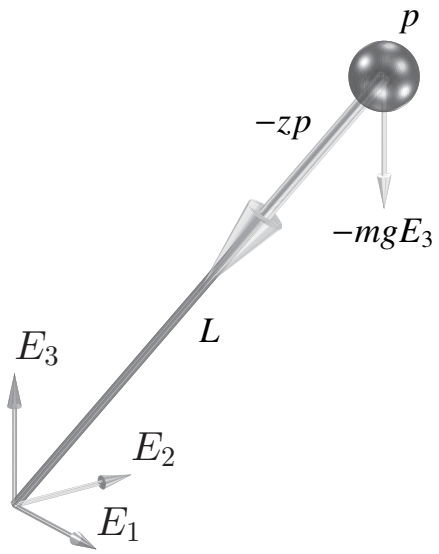


Figure 14.1 Illustration of the example considered in this chapter. The system is described via the cartesian position of the mass $p \in \mathbb{R}^3$ in the fixed frame E. The mass is subject to the gravity force $-mgE_3$ and to a force $-zp$ from the rod, which ensures that the mass remains at a distance L from its attachment point. Here the scalar z is a variable in the dynamics that scales this force adequately.

(14.1) is not a simple explicit ODE. Indeed, while the scalar variable z is intrinsically part of the model, its time derivative does not appear in the model equation. Hence, variable z is of a different nature than variable p. A variable that is intrinsic to the model equation (i.e. excluding possibly time-varying parameters and inputs) but that is not time differentiated in the model equation is labelled an *algebraic state*. A differential equation holding such variables is called a *Differential Algebraic Equation* (DAE).

Following up on this example, we will now provide a more formal view on the concept of Differential-Algebraic Equations.

14.1 What are DAEs ?

Let us consider a differential equation in a very generic form:

$$f(\dot{x}(t), x(t), u(t)) = 0. \tag{14.2}$$

Such a differential equation is labelled *implicit* as the state derivative $\dot{x}(t)$ is not provided via an explicit function of the state $x(t)$ and input $u(t)$, but implicitly as the solution of (14.2). The Implicit Function Theorem guarantees that $\dot{x}(t)$ can be seen as a locally unique and continuously differentiable function of $x(t)$ and $u(t)$ if the Jacobian of f with respect to $\dot{x}(t)$, i.e. $\frac{\partial f}{\partial \dot{x}}$, is full rank. Under this condition, one is guaranteed that for a given state $x(t)$ and input $u(t)$, the state time derivative $\dot{x}(t)$ can be computed, either explicitly or numerically e.g. via a Newton iteration. Then (14.2) is an *Ordinary Differential Equation*, since $\dot{x}(t)$ can be computed at every time instant, and the model can in principle be treated as a classic explicit ODE.

Formally, Differential-Algebraic Equations are equations in the form (14.2) for which the above rank condition fails. Let us formalise that in the following definition.

Definition 14.1. $f(\dot{x}(t), x(t), u(t)) = 0$ is a DAE if $\frac{\partial f}{\partial \dot{x}}$ is rank deficient.

It is admittedly not straightforward to relate Definition 14.1 to the earlier example (14.1). Before making this relationship clear, let us illustrate Definition 14.1 on a simple example.

Example 14.2. Let us consider the following implicit differential equation, having the form (14.2):

$$f(\dot{x}, x, u) = \begin{bmatrix} x_1 - \dot{x}_1 + 1 \\ \dot{x}_1 x_2 + 2u \end{bmatrix} = 0, \tag{14.3}$$

then the Jacobian of f with respect to the state derivatives $\dot{x}$ reads as:

$$\frac{\partial f}{\partial \dot{x}} = \begin{bmatrix} -1 & 0 \\ x_2 & 0 \end{bmatrix},$$

and is rank-deficient, entailing that (14.3) is, by Definition 14.1, a DAE.

Alternatively, one can also simply observe that $\dot{x}_2$ does not appear time-differentiated in (14.3), such that one can assess by simple inspection that it is a DAE. In order to gain some further intuition in this example, consider solving the first equation in (14.3) for $\dot{x}_1$, giving

$$\dot{x}_1 = x_1 + 1$$

Upon inserting the expression for $\dot{x}_1$ in the second equation, one can then write (14.3) as

$$\begin{aligned} \dot{x}_1 &= x_1 + 1, \\ 0 &= (x_1 + 1)\, x_2 + 2. \end{aligned}$$

We observe here that the second equation is in fact purely algebraic, such that the model can be written as a mixture of an explicit differential equation and of an algebraic equation. This form of DAE is actually the most commonly used in practice. It is referred to as a semi-explicit DAE.

The above example can mislead one to believe that DAEs are fairly simple objects. To dispel that impression, let us provide a simple example of a DAE that possess fairly exotic properties.

Example 14.3. Let us consider the following differential equation

$$\begin{aligned} \dot{x}_1 + x_1 - u &= 0, \\ (x_1 - x_2)\,\dot{x}_2 + x_1 - x_2 &= 0, \end{aligned}$$

having the Jacobian

$$\frac{\partial f}{\partial \dot{x}} = \begin{bmatrix} 1 & 0 \\ 0 & x_1 - x_2 \end{bmatrix}$$

which is rank-deficient for $x_1 = x_2$. Hence for the initial conditions:

$$x_1(0) = x_2(0)$$

our equation is a DAE and its solution obeys:

$$\begin{aligned} \dot{x}_1 &= u - x_1 \\ 0 &= x_2 - x_1, \end{aligned}$$

otherwise it is an ODE. The fact that some differential equation can switch between being DAEs and ODEs betrays the fact that DAEs are not necessarily simple to handle and analyse. However, in the context of numerical optimal control, simple DAEs are typically favoured.

As observed before, DAEs often simply arise from the fact that some states in the state vector x do not appear time-differentiated in the model equations, yielding a column of zeros in the Jacobian $\frac{\partial f}{\partial \dot{x}}$, as e.g. in example (14.2). In such a case, it is very useful to make an explicit distinction in the implicit differential equation (14.2) between the *differential variables*, i.e. the variables whose time derivative appear in f, typically labelled x, and the *algebraic variables*, i.e. the variables whose time derivative do not appear in f, typically labelled z. One can then rewrite (14.2) as:

$$f(\dot{x}, z, x, u) = 0. \tag{14.4}$$

A DAE in the form (14.4) is called a fully-implicit DAE. The application of

definition 14.1 to (14.4) must then be understood in the sense that

$$\det\begin{pmatrix} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial z} \end{pmatrix} = \det\begin{pmatrix} \frac{\partial f}{\partial \dot{x}} & 0 \end{pmatrix} = 0 \tag{14.5}$$

is always rank deficient. The differential equation (14.4) is therefore always a DAE.

As mentioned in example 14.2, a common form of DAE often used in practice is the so-called *semi-explicit* form. It consists in explicitly splitting the DAE between an explicit differential equation and an implicit algebraic one. It reads as:

$$\begin{aligned} \dot{x} &= f(x, z, u), \\ 0 &= g(x, z, u). \end{aligned}$$

The semi-explicit form is the most commonly used form of DAEs in optimal control. We turn next to a very important notion in the world of Differential-Algebraic Equations, both in theory and in practice.

14.2 Differential Index of DAEs

Before introducing the notion of differential index for DAE, it will be useful to take a brief and early tour into the problem of solving DAEs. Consider the semi-explicit DAE:

$$\dot{x} = f(x, z, u), \tag{14.6a}$$

$$0 = g(x, z, u), \tag{14.6b}$$

and suppose that one can construct (possibly via a numerical algorithm such as a Newton iteration) a function $\xi(x, u)$ such that:

$$g(x, \xi(x, u), u) = 0, \qquad \forall x,\ u.$$

That is, function $\xi(x, u)$ delivers the algebraic state z for any differential state x and input u. One can then proceed with eliminating the algebraic state z in (14.6), such that the DAE reads as:

$$\dot{x} = f(x, \xi(x, u), u), \tag{14.7a}$$

$$z = \xi(x, u). \tag{14.7b}$$

One can observe that (14.7a) is then an explicit ODE, and can therefore be handled via any classical numerical integration method. Moreover, (14.7b) provides the algebraic states explicitly. When such an elimination of the algebraic states is possible, one can consider the DAE (14.6) as "easy" to solve. It is then

natural to ask when such an elimination is possible. The Implicit Function Theorem (IFT) provides here a straightforward answer, namely the function $\xi(x,u)$ exists (locally) if the Jacobian

$$\frac{\partial}{\partial z} g(x,z,u) \tag{14.8}$$

is full rank along the trajectories x, u, z of the system. The full-rankness of the Jacobian (14.8) additionally guarantees that the Newton iteration:

$$z \leftarrow z - \frac{\partial g(x,z,u)}{\partial z}^{-1} g(x,z,u) \tag{14.9}$$

converges locally to the solution z of (14.6b). In that sense, (14.9) can be seen as a numerical procedure for constructing the implicit function $\xi(x,u)$.

These notions easily extend to fully-implicit DAEs in the distinct form (14.4). More specifically, suppose that there exists two functions $\xi_{\dot{x}}(x,u)$ and $\xi_z(x,u)$ that satisfy the fully implicit DAE (14.4), i.e.

$$f(\xi_{\dot{x}}(x,u), \xi_z(x,u), x, u) = 0, \quad \forall\, x,\ u.$$

Then one can rewrite (14.4) as:

$$\dot{x} = \xi_{\dot{x}}(x,u) \tag{14.10a}$$

$$z = \xi_z(x,u). \tag{14.10b}$$

Similarly to (14.7), one can treat (14.10a) as a simple ODE, while (14.10b) delivers the algebraic states z explicitly. The existence of functions $\xi_{\dot{x}}(x,u)$ and $\xi_z(x,u)$ can then again be guaranteed by invoking the IFT, namely if for (14.4) the Jacobian matrix

$$\begin{bmatrix} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial z} \end{bmatrix} \tag{14.11}$$

is full rank, then functions $\xi_{\dot{x}}(x,u)$ and $\xi_z(x,u)$ exist locally. The attentive reader will want to observe the important distinction between (14.5) which always hold for (14.4), and (14.11) whose full-rankness guarantees the local existence of the implicit functions $\xi_{\dot{x}}(x,u)$ and $\xi_z(x,u)$.

Let us consider two examples to illustrate these notions.

Example 14.4. Consider again the fully-implicit DAE of Example 14.2, i.e.:

$$f(\dot{x}, z, x, u) = \begin{bmatrix} x - \dot{x} + 1 \\ \dot{x} z + 2 \end{bmatrix} = 0.$$

We observe that

$$\begin{bmatrix} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial z} \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ z & \dot{x} \end{bmatrix}$$

is full rank whenever $\dot{x} \neq 0$, such that the implicit functions $\xi_{\dot{x}}(x,u)$ and $\xi_z(x,u)$ are guaranteed to exist when $\dot{x} \neq 0$. In this simple case, they can actually be computed explicitly. Indeed, we observe that:

$$\dot{x} = \xi_{\dot{x}}(x,u) = x + 1, \qquad z = \xi_z(x,u) = -\frac{2}{x+1}$$

solve $f(\dot{x}, z, x, u)$ whenever $\dot{x} = x + 1 \neq 0$.

This simple example needs to be pitted against a more problematic one.

Example 14.5. Consider the fully-implicit DAE:

$$f(\dot{x}, z, x, u) = \begin{bmatrix} \dot{x}_1 - z \\ \dot{x}_2 - x_1 \\ x_2 - u \end{bmatrix} = 0.$$

We observe that:

$$\begin{bmatrix} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial z} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

is always rank-deficient, such that the differential state $\dot{x}$ and algebraic state z cannot be uniquely obtained (even numerically) from solving $f(\dot{x}, z, x, u) = 0$ alone.

The topic of this section is the notion of differential index of DAEs. As we will see next, the loose idea of "easy" DAEs presented above is directly related to it. Let us introduce now the notion of differential index for DAEs.

Definition 14.6. The differential index of a DAE is the number of times it must be time-differentiated before an explicit ODE is obtained.

For the specific case of a semi-explicit DAE, the above definition also reads as follows.

Definition 14.7. The differential index of the semi-explicit DAE (14.6) is the number of times its algebraic part (14.6b) must be time-differentiated before an explicit ODE is obtained.

In order to clarify these definitions, let us make a simple example.

Example 14.8. Let us calculate the differential index of the DAE proposed in Example 14.2, i.e.:

$$f(\dot{x}, z, x) = \begin{bmatrix} x - \dot{x} + 1 \\ \dot{x} z + 2 \end{bmatrix} = 0.$$

We then consider the time derivative of f, i.e.:

$$\dot{f}(\ddot{x}, \dot{x}, x, \dot{z}, z) = \begin{bmatrix} \dot{x} - \ddot{x} \\ \ddot{x}z + \dot{x}\dot{z} \end{bmatrix} = 0. \tag{14.12}$$

For the sake of clarity, we label $v = \begin{bmatrix} x \\ z \\ \dot{x} \end{bmatrix}$ and rewrite (14.12) in the equivalent form:

$$\zeta(\dot{v}, v) = \begin{bmatrix} \dot{v}_1 - v_3 \\ v_3 - \dot{v}_3 \\ \dot{v}_3 v_2 + v_3 \dot{v}_2 \end{bmatrix} = 0.$$

The Jacobian

$$\frac{\partial \dot{\zeta}(\dot{v}, v)}{\partial \dot{v}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & v_3 & v_2 \end{bmatrix}$$

is then full rank, such that ζ is an ODE for v according to Definition 14.1. Since a single time-differentiation has converted the original DAE of this example into an ODE, we can conclude that the original DAE is of index 1.

Let us contrast this example with a DAE having a higher differential index.

Example 14.9. Let us calculate the differential index of our illustrative example (14.1). Using $v = \dot{p}$, and defining the differential state vector

$$x = \begin{bmatrix} p \\ v \end{bmatrix}$$

one can easily verify that the DAE (14.1) can be written as a semi-explicit DAE:

$$\dot{p} = v, \tag{14.13a}$$

$$\dot{v} = m^{-1}u - m^{-1}zp + gE_3, \tag{14.13b}$$

$$0 = \underbrace{\frac{1}{2}\left(p^\top p - L^2\right)}_{=g(x,z,u)}. \tag{14.13c}$$

We observe that for a given z, (14.13a)-(14.13b) are already ODEs in v and p. As per Definition 14.7, we need to differentiate the algebraic equation (14.13c) until (14.13) becomes an ODE. The two first time derivatives read as:

$$\dot{g}(x, z, u) = p^\top v = 0, \qquad \text{and} \qquad \ddot{g}(\dot{x}, x, z, u) = p^\top \dot{v} + v^\top v = 0$$

One can then use (14.13b) in $\ddot{g}(\dot{x}, x, z, u)$ to obtain

$$\ddot{g}(\dot{x}, x, z, u) = p^\top \left(u - m^{-1} z p + g E_3\right) + v^\top v = 0,$$

As z now appears explicitly in $\ddot{g}(\dot{x}, x, z, u)$, an extra time-differentiation yields a differential equation from which $\dot{z}$ can be computed if $p^\top p \neq 0$. We observe that 3 time-differentiations of (14.13c) were necessary to turn (14.13) into an ODE. It follows that (14.13) is an index-3 DAE.

Now we ought to relate the notion of "easy" DAEs to the notion of differential index. More specifically, we shall see next that index-1 DAEs are "easy" DAEs in the sense detailed previously. This observation can be formally described in the following Lemma.

Lemma 14.10. *For any fully-implicit index-1 DAE*

$$f(\dot{x}, z, x, u) = 0,$$

there exists implicit functions $\xi_{\dot{x}}(x, u)$ and $\xi_z(x, u)$ that satisfy:

$$f(\xi_{\dot{x}}(x, u), \xi_z(x, u), x, u) = 0, \quad \forall\, x,\, u.$$

Proof We observe that if f is of index 1, then a single time-differentiation:

$$\dot{f} = \frac{\partial f}{\partial \dot{x}} \ddot{x} + \frac{\partial f}{\partial z} \dot{z} + \frac{\partial f}{\partial x} \dot{x} + \frac{\partial f}{\partial u} \dot{u} = 0$$

yields a pure ODE. For the sake of clarity, we label $v = \begin{bmatrix} \dot{x} \\ z \end{bmatrix}$ and write:

$$\dot{f} = \begin{bmatrix} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial z} \end{bmatrix} \dot{v} + \frac{\partial f}{\partial x} \dot{x} + \frac{\partial f}{\partial u} \dot{u} = 0. \tag{14.14}$$

By assumption, (14.14) can be written as an explicit ODE, hence $\begin{bmatrix} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial z} \end{bmatrix}$ must be full rank, such that:

$$\dot{v} = -\begin{bmatrix} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial z} \end{bmatrix}^{-1} \left(\frac{\partial f}{\partial x} \dot{x} + \frac{\partial f}{\partial u} \dot{u} \right)$$

holds on the DAE trajectories. The IFT then guarantees the existence of the implicit functions $\xi_{\dot{x}}(x, u)$ and $\xi_z(x, u)$ in a neighborhood of the trajectories of the DAE. □

A similar result can clearly be established for any index-1 semi-explicit DAEs on the existence of an implicit function $\xi_z(x, u)$ that solves the algebraic equation, i.e. such that

$$g(x, \xi_z(x, u), u) = 0, \quad \forall\, x,\, u$$

The crucial practical consequence of these observations is that index-1 DAEs can be in principle solved numerically (or sometimes even explicitly) without difficulties, as for any state and input $x(t)$ and $u(t)$, the state derivative $\dot{x}(t)$ and algebraic state $z(t)$ can be computed, and the simulation of the dynamics performed. In practice, implicit integration methods are the most efficient approach to perform the simulations of index-1 DAEs (see Section 14.4 below for some details on this question), while DAEs of index higher than 1 require specially-tailored integrators.

A non-trivial but important point needs to be stressed here. DAEs of index higher than 1, often labelled *high-index DAEs*, present a pitfall to uninformed users of numerical methods. Indeed, one deploying a classical implicit integration method on a high-index DAE may observe that the implicit integration method converges reliably and be mislead into believing that simulations of the DAE model can be reliably computed. In order to clarify this issue, let us consider the following example, based on a linear, high-index DAE.

Example 14.11. In this example, we are interested to observe the result of "naively" deploying a classical implicit integration scheme on a high-index DAE. We consider again the fully-implicit DAE of Example 14.5, i.e.

$$f(\dot{x}, z, x, u) = \begin{bmatrix} \dot{x}_1 - z \\ \dot{x}_2 - x_1 \\ x_2 - u \end{bmatrix}. \tag{14.15}$$

The reader can easily verify that (14.15) is not an index-1 DAE. We observe that we can rewrite (14.15) in the linear form $E\dot{v} = Av + Bu$, where

$$v = \begin{bmatrix} x \\ z \end{bmatrix}, \quad E = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

We are interested now in naively deploying an implicit Euler scheme of step length h on this DAE, yielding the steps:

$$\frac{1}{h} E\left(v_+ - v(t)\right) = Av_+ + Bu(t+h)$$

where v_+ is an approximation of the state at time $v(t+h)$, i.e. $v_+ \approx v(t+h)$. It can be verified that the true trajectories $v(t+h)$ satisfy:

$$E\left[\frac{1}{h}\left(v(t+h) - v(t)\right) + \frac{h}{2}\ddot{x}(\tau)\right] = Av(t+h) + Bu(t+h)$$

for some $\tau \in [t, t+h]$. We can then consider the one-step integration error

$e_+ = v_+ - v(t+h)$ given by:

$$e_+ = \frac{h^2}{2}(E - Ah)^{-1} E\ddot{x}(\tau).$$

For DAE (14.15), matrix $\frac{h^2}{2}(E - Ah)^{-1} E$ reads as:

$$\frac{h^2}{2}(E - Ah)^{-1} E = \frac{1}{2}\begin{bmatrix} 0 & h & 0 \\ 0 & 0 & 0 \\ h & 1 & 0 \end{bmatrix},$$

such that the integration error is of order $\mathcal{O}(1)$, i.e. much worse than the integration error expected from the implicit Euler method, which is of order $\mathcal{O}\left(h^2\right)$.

This simple example reveals that, even though a classic implicit integration scheme deployed on high-index DAEs (14.15) can in some cases reliably deliver state trajectories, their lousy numerical accuracy typically makes them actually meaningless as simulations of the DAE model. The difficulty with DAE (14.15) stems from its index larger than 1. These observations must be taken as a warning that while one can sometimes deploy a classical implicit integration scheme on a high-index DAE without observing notable numerical difficulties, the resulting trajectories are typically nonetheless senseless. Hence, good practice in numerical optimization dictates that the index of a DAE ought to be systematically checked before tackling it via classical integration methods.

Because index-1 DAEs are significantly easier to treat than high-index DAEs, it is common in numerical optimal control to avoid DAEs of index larger than 1. Unfortunately, the index of a DAE stems from the nature of the physical system it models and cannot be decided. However, a treatment of high-index DAEs generally allows one to ultimately treat them as index-1 DAEs. We will cover this question next.

14.3 Index reduction

As detailed in the previous Section, index-1 DAEs are simpler to treat numerically than high-index DAEs, as index-1 DAEs can be approached using standard implicit integration methods. This observation motivates the deployment of procedures for reducing the index of an arbitrary high-index DAE into an index-1 DAE, a procedure labelled *index reduction*. Index-reduction proceeds very similarly to the procedure leading to assess the index of a DAE, i.e. via time-differentiation of the DAE (or of some parts of the DAE) until a DAE of index 1 is obtained. In order to explain this further, let us detail it on our illustrative example (14.1).

Example 14.12. We consider again the semi-explicit DAE (14.13), i.e.

$$\dot{p} = v, \tag{14.16a}$$
$$\dot{v} = u - m^{-1} z p + g E_3, \tag{14.16b}$$
$$0 = \underbrace{\frac{1}{2}\left(p^\top p - L^2\right)}_{=g(x,z,u)}, \tag{14.16c}$$

which is in a semi-explicit form. Similarly to the index evaluation presented in Example 14.9, i.e. we consider the time-derivatives of the algebraic equation (14.16c)

$$\dot{g}\,(x, z, u) = p^\top v = 0, \qquad \text{and} \qquad \ddot{g}\,(\dot{x}, x, z, u) = p^\top \dot{v} + v^\top v = 0$$

One can then easily verify that the new DAE:

$$\dot{p} = v, \tag{14.17a}$$
$$m\dot{v} = u - zp + mgE_3, \tag{14.17b}$$
$$0 = \underbrace{p^\top \dot{v} + v^\top v}_{=\ddot{g}(\dot{x},x,z,u)}, \tag{14.17c}$$

is of index 1. Alternatively, it is useful to put (14.17) in a more implicit form:

$$\dot{p} = v, \tag{14.18a}$$
$$\begin{bmatrix} m & p \\ p^\top & 0 \end{bmatrix} \begin{bmatrix} \dot{v} \\ z \end{bmatrix} = \begin{bmatrix} u + mgE_3 \\ -v^\top v \end{bmatrix}, \tag{14.18b}$$

which shows unambiguously that the state derivatives $\dot{v}$ and $\dot{p}$ as well as the algebraic state z can be computed for any state v, p and input u as long as $p \neq 0$. This observation tells us without further investigation that (14.18) is an "easy" DAE, i.e. of index 1.

Index-reduction procedures can be fairly intricate to deploy on very complex models. For the sake of completeness, let us report here a recipe proposed in [12] for performing the index-reduction on any semi-explicit DAE:

$$\dot{x} = f\,(x, z, u)$$
$$0 = g\,(x, z, u)$$

(i) Check if the DAE system is index 1 (i.e. $\frac{\partial g}{\partial z}$ full rank). If yes, stop.
(ii) Identify a subset of algebraic equations that can be solved for a subset of algebraic variables.
(iii) Perform a time-differentiation on the remaining algebraic equations that contain (some of) the differential variables x. Terms $\dot{x}$ will appear in these differentiated equations.

(iv) Substitute the $\dot{x}$ with their corresponding symbolic expressions $f(x, z, u)$. This generates new algebraic equations.
(v) With this new DAE system, go to step 1.

Our discussion on index reduction would not complete if we omit the question of consistency conditions. To understand this issue, consider the index-reduced DAE developed in Example 14.12, which takes the form:

$$\dot{x} = f(x, z, u) \tag{14.19a}$$
$$0 = \ddot{g}(\dot{x}, x, z, u) \tag{14.19b}$$

One needs to observe here that while a solution to the original DAE (14.16) in the form

$$\dot{x} = f(x, z, u) \tag{14.20a}$$
$$0 = g(x, z, u) \tag{14.20b}$$

is obviously also a solution for the index-reduced DAE (14.19), the converse is not necessarily true, i.e. a solution of the index-reduced DAE (14.19) is not necessarily a solution to the original DAE (14.20). To understand this statement, one simply ought to imagine a trajectory that is solution of (14.19a), and for which

$$g(x(t), z(t), u(t)) = g(x(0), z(0), u(0)) + t\dot{g}(x(0), z(0), u(0)) = 0 \tag{14.21}$$

holds. This trajectory clearly satisfies (14.19b) but not (14.20b). Equation (14.21) additionally reveals that the issue is not related the DAEs themselves, but rather to the initial conditions chosen for the simulation of the DAEs. Indeed, simply selecting the initial conditions $x(0)$ such that

$$g(x(0), z(0), u(0)) = 0, \quad \text{and} \quad \dot{g}(x(0), z(0), u(0)) = 0 \tag{14.22}$$

ensures that the trajectories of the index-reduced DAE are solution of the original one. More generally, enforcing

$$g(x(t_0), z(t_0), u(t_0)) = 0, \quad \text{and} \quad \dot{g}(x(t_0), z(t_0), u(t_0)) \tag{14.23}$$

at any time t_0 on the trajectory guarantees a simulation run with the index-reduced DAE is a valid simulation of the original DAE. Conditions that guarantees the validity of the simulation performed on the index-reduced DAE, such as (14.23), are labelled *consistency conditions*.

In the context of optimal control based on an index-reduced DAE, consistency conditions are crucial when the trajectories of the differential states of system do not have (fully) prescribed initial or terminal values. In such a case, the consistency conditions must be adequately enforced within the optimal

control problem. E.g. an optimal control problem involving our index-reduced DAE (14.18) having free initial or terminal states can e.g. be written as:

$$
\begin{aligned}
\underset{v,\,p,\,z,\,u}{\text{minimize}} \quad & \int_0^T L(v,p,z,u)\,\mathrm{d}t \\
\text{subject to} \quad & \dot{p} = v && \text{(Differential equ.)}, \\
& m\dot{v} = u - zp + mgE_3 && \text{(Differential equ.)}, \\
& 0 = p^\top \dot{v} + v^\top v && \text{(Algebraic equ.)}, \\
& 0 = p(t_0)^\top v(t_0) && \text{(Consistency cond.)}, \\
& 0 = p(t_0)^\top \dot{v}(t_0) + v(t_0)^\top v(t_0) && \text{(Consistency cond.)}
\end{aligned}
\tag{14.24}
$$

for any $t_0 \in [0,\, T]$.

It ought to be underlined here that imposing *some* constraints on the initial and/or terminal state trajectories in conjunction with imposing the consistency conditions must be done with great care in order to avoid generating a redundant set of constraints in the OCP. As a trivial example of this difficulty, imposing e.g. the initial states in (14.24) in addition to the consistency conditions with $t_0 = 0$ would clearly over-constrain the initial state values $p(0)$, $v(0)$. This issue can become significantly more involved in less obvious scenarios, such as e.g. in periodic OCPs, where the initial and terminal states are free but must satisfy a periodicity constraint of the form $x(0) = x(T)$. Handling the consistency conditions and the periodicity constraints together in the OCP without generating an over-constrained problem can then become fairly involved.

The consistency conditions can in principle be enforced at any time t_0 in the time span considered by the OCP. However, in some cases the selection of the time t_0 for imposing the consistency condition is not arbitrary. Indeed, one ought to observe that the combination of the index-reduced algebraic constraint and of the consistency conditions, i.e.

$$\ddot{g}\left(x(t), z(t), u(t)\right) = 0 \tag{14.25}$$

$$\dot{g}\left(x(t_0), z(t_0), u(t_0)\right) = 0 \tag{14.26}$$

$$g\left(x(t_0), z(t_0), u(t_0)\right) = 0 \tag{14.27}$$

$$\tag{14.28}$$

ensure mathematically that

$$g\left(x(t), z(t), u(t)\right) = g\left(x(t_0), z(t_0), u(t_0)\right) + (t - t_0)\dot{g}\left(x(t_0), z(t_0), u(t_0)\right) = 0$$

holds at any time t. However, when the DAE dynamics are handled via numerical integration, numerical errors tend to accumulate over time such that

$g(x(t), z(t), u(t)) = 0$ can be less accurately enforced at times that are distant from t_0. From this observation one ought to conclude that if the solution to an OCP is e.g. more important at the beginning of the time span the OCP covers, say $[0, T]$, then the consistency conditions ought to be enforced in the beginning of the time span, i.e. $t_0 = 0$. This situation occurs in Nonlinear Model Predictive Control (NMPC), where the first control input q_0 delivered by the OCP provides the control input to be deployed on the real system, such that the accuracy of the solution in the beginning of the time interval it covers is the more important than later in the horizon.

Conversely, if the OCP implements a Moving Horizon Estimation (MHE) scheme, then the differential state obtained at the very end of the time span covered by the OCP delivers a state estimation to e.g. an NMPC scheme. In such a case, the accuracy is most important at the very end of the time interval, such that the consistency conditions are best imposed at $t_0 = T$. These ideas are detailed in [].

14.4 Direct Methods with Differential-Algebraic Equations

We will now turn to discussing the deployment of direct optimal control methods on OCPs involving DAEs. For the reasons detailed previously, we will focus on OCPs involving index-1 DAEs, possibly arising from an index-reduction of a high-index DAE.

14.4.1 Numerical Solution of Differential-Algebraic Equations

In this Section, we will briefly discuss the numerical solution of DAEs. As hinted above, index-1 DAEs are significantly simpler to treat numerically than high-index ones, and are therefore often preferred in optimal control. In this Section, we will focus in the index-1 case.

Though low-order methods generally offer a poor ratio between accuracy and computational complexity and higher-order integrators should be preferred, let use nonetheless start here with a simple m-step implicit Euler scheme for the sake of illustration. For e.g. a semi-explicit DAE

$$\begin{aligned} \dot{x} &= f(x, z, u), \\ 0 &= g(x, z, u), \end{aligned}$$

the m-step implicit Euler scheme computes a numerical simulation $x(t_{k+1}, s_k, q_k)$ of the model dynamics over a time interval $[t_k,\ t_{k+1}]$ from the initial state s_k and the constant input q_k via the following algorithm.

Algorithm 14.13 (Implicit Euler integrator)**.**
Input: initial value s_k, input q_k and times $t_k,\ t_{k+1}$
Set $v = q_k$, and $h = (t_{k+1} - t_k)/m$
for $i = 0$ to $m - 1$ **do**
Solve

$$x_+ = v + hf\left(x_+, z_+, q_k\right)$$
$$0 = g\left(x_+, z_+, q_k\right)$$

for $x_+,\ z_+$ via a Newton iteration, set $v \leftarrow x_+$.
end for
Output: $x(t_{k+1}, s_k, q_k) = v$

A similar approach can be deployed using any implicit integration method, see Chapter 10, such as an IRK4 integrator.

A fairly efficient and useful type of implicit integrator already introduced in Chapter 10 and further detailed in Section 13.3 is the orthogonal collocation approach. Let us consider the building of the collocation equations for DAEs in a generic implicit form

$$f\left(\dot{x}, x, z, u\right) = 0 \tag{14.29}$$

on a time interval $[t_k,\ t_{k+1}]$, with initial value s_k and a constant input q_k. The differential states are, as in the ODE case described via polynomials $p\left(t, v_k\right)$, with $t \in [t_k,\ t_{k+1}]$ linearly parametrized in $v_k \in$ such that:

- the polynomial interpolation meets the initial value, i.e.:

$$\underbrace{p\left(t_k, v_k\right)}_{=v_{k,0}} = s_k \tag{14.30}$$

- the DAE is satisfied in the collocation times $t_{k,i}$ for $i = 1, \ldots, d$, i.e.:

$$f(\dot{p}_k\left(t_{k,i}, v_k\right), \underbrace{p_k\left(t_{k,i}, v_k\right)}_{=v_{k,i}}, z_{k,i}, q_k) = 0, \quad i = 1, \ldots, d \tag{14.31}$$

We can gather these requirements in the compact implicit equation:

$$c_k\left(v_k, z_k, q_k, s_k\right) = \begin{bmatrix} v_{k,0} - s_k \\ f(\dot{p}_k\left(t_{k,1}, v_k\right), v_{k,1}, z_{k,1}, q_k) \\ \vdots \\ f(\dot{p}_k\left(t_{k,d}, v_k\right), v_{k,d}, z_{k,d}, q_k) \end{bmatrix} = 0. \tag{14.32}$$

The same observations as for the semi-explicit case hold for the general case.

One ought to observe that the discretized algebraic states $z_{k,i}$ appear only for the indices $i = 1, \ldots, d$ in the collocation equations, while the discretized

differential states $v_{k,i}$ appear for the indices $i = 0, \dots, d$. I.e. the discrete algebraic states z_k have *one degree of freedom less* that the discrete differential states v_k. The extra degree of freedom granted to the differential state is actually required in order to be able to meet the initial value s_k of the differential state trajectories, while the initial value of algebraic state trajectories cannot be assigned as they are already defined implicitly by the DAE, subject to the imposed state initial value s_k and input q_k. This observation is most obvious in the semi-explicit case, where for a given state initial value s_k and input q_k, the initial value for the algebraic state is implicitly given by $g\left(s_k, z\left(t_k\right), q_k\right) = 0$.

For the sake of completeness, let us provide the algorithm for a collocation-based integrator for index-1 DAEs.

Algorithm 14.14 (Collocation-based integrator)**.**
Input: initial value s_k input q_k, initial guess $v_k,\ z_k$ and times $t_k,\ t_{k+1}$
Solve

$$c_k\left(v_k, z_k, q_k, s_k\right) = 0 \tag{14.33}$$

for $v_k,\ z_k$ via a Newton iteration
Output: $x(t_{k+1}, s_k, q_k) = p_k\left(t_{k+1}, v_k\right)$

It is interesting to observe here that while the algorithm ought to receive an initial guess for the discrete algebraic states z_k, it receives an initial value s_k *only* for the differential state. It is also important to notice that the algebraic states z_k can in principle be entirely hidden inside the integrator (even though they can be, of course, reported).

Sensitivities of the integrators The computation of the sensitivities of an implicit integrator such as (14.14) can be done as detailed in Section 10.4. More specifically, if we label $w_k = \begin{bmatrix} v_k \\ z_k \end{bmatrix}$, the collocation equation (14.36) in algorithm 14.14 is typically solved using a (often full-step) Newton iteration:

$$w_k = w_k - \frac{\partial c_k\left(w_k, q_k, s_k\right)}{\partial w_k}^{-1} c_k\left(w_k, q_k, s_k\right) \tag{14.34}$$

The sensitivities are then provided at the solution $c_k\left(w_k, q_k, s_k\right) = 0$ by:

$$\frac{\partial w_k}{\partial q_k} = -\frac{\partial c_k\left(w_k, q_k, s_k\right)}{\partial w_k}^{-1} \frac{\partial c_k\left(w_k, q_k, s_k\right)}{\partial q_k}, \tag{14.35a}$$

$$\frac{\partial w_k}{\partial s_k} = -\frac{\partial c_k\left(w_k, q_k, s_k\right)}{\partial w_k}^{-1} \frac{\partial c_k\left(w_k, q_k, s_k\right)}{\partial s_k}. \tag{14.35b}$$

It is important to note here that a factorization of the Jacobian matrix $\frac{\partial c_k(w_k,q_k,s_k)}{\partial w_k}$ is already computed for the Newton iterations (14.34) and the last factorization can be readily reused at the end of the iteration to form the sensitivities (14.35). The computational complexity of obtaining the sensitivities consists then only of the computation of the matrices $\frac{\partial c_k(w_k,q_k,s_k)}{\partial s_k}$ and $\frac{\partial c_k(w_k,q_k,s_k)}{\partial q_k}$ and the matrix products in (14.35). A collocation-based integrator with sensitivities then reads as:

Algorithm 14.15 (Collocation-based integrator with Sensitivities)**.**

Input: initial value s_k input q_k, initial guess $v_k,\ z_k$ and times $t_k,\ t_{k+1}$

Solve

$$c_k\left(v_k, z_k, q_k, s_k\right) = 0 \tag{14.36}$$

for $v_k,\ z_k$ via a Newton iteration

Compute (14.35)

Form:

$$\frac{\partial x(t_{k+1}, s_k, q_k)}{\partial s_k} = \frac{\partial p_k\left(t_{k+1}, v_k\right)}{\partial v_k}\frac{\partial v_k}{\partial w_k}\frac{\partial w_k}{\partial s_k} \tag{14.37}$$

$$\frac{\partial x(t_{k+1}, s_k, q_k)}{\partial q_k} = \frac{\partial p_k\left(t_{k+1}, v_k\right)}{\partial v_k}\frac{\partial v_k}{\partial w_k}\frac{\partial w_k}{\partial q_k} \tag{14.38}$$

Output: $x(t_{k+1}, s_k, q_k) = p_k\left(t_{k+1}, v_k\right)$, and $\frac{\partial x(t_{k+1},s_k,q_k)}{\partial s_k}$, $\frac{\partial x(t_{k+1},s_k,q_k)}{\partial q_k}$

where $\frac{\partial v_k}{\partial w_k} = \begin{bmatrix} I & 0 \end{bmatrix}$ is constant.

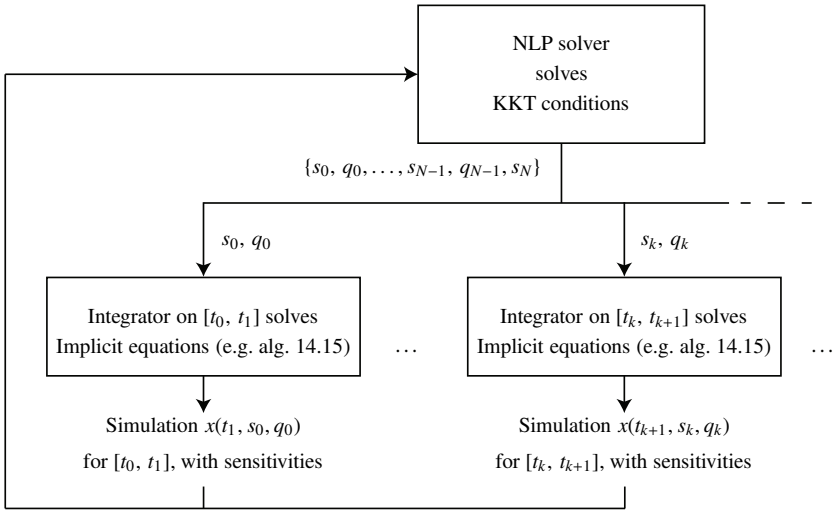
We can now turn to the deployment of Multiple-Shooting on DAE-based optimal control problems.

14.4.2 Direct Multiple-Shooting with Differential-Algebraic Equations

In the context of Multiple-Shooting for DAE-based optimal control problems, the implicit numerical integration schemes detailed above are interacting with the NLP solver tackling the NLP resulting from the Multiple-Shooting discretization. We illustrate this interaction in Figure 14.4.2. The NLP solver is then responsible for closing the shooting gaps, i.e. enforcing the continuity conditions:

$$x(t_{k+1}, s_k, q_k) - s_{k+1} = 0$$

for $k = 0, \ldots, N-1$, and solving the set of algebraic equations that capture the conditions of optimality. It is interesting to observe here that the overall process can be then construed as a "two-level Newton scheme", where the



upper level solve the KKT conditions (e.g. using the relaxed KKT obtained in the primal-dual interior-point approach, or using SQP iterations) and the lower level solves the equations underlying the numerical integration (e.g. (14.36)). The NLP solver passes the discrete states and inputs s_k, q_k, which become inputs to the numerical integration algorithms (e.g. 14.14), while the numerical integration algorithms report to the NLP solver the end states of the simulations $x(t_{k+1}, s_k, q_k)$ and their sensitivities.

One ought to observe here that the algebraic state dynamics can in principle be totally "hidden" inside the integrator scheme, and not reported at all to the NLP solver. In that sense, implicit integrators in general perform an elimination of the algebraic variables present in the dynamics, and hide their existence to the NLP solver.

Another crucial observation to make here is that no continuity condition nor initial condition is enforced on the algebraic state trajectories $z(t)$. Indeed, for a given differential state trajectory $x(t)$ and input profile $u(t)$, the algebraic state trajectories $z(t)$ are entirely defined via the DAE (e.g. by $g(x, z, u) = 0$ in the semi-explicit case), such that an extra continuity condition imposed on $z(t)$ would yield an over-constrained problem. As a matter of fact, if a discontinuous input parametrization is used (such as e.g. piecewise-constant), the algebraic state trajectories $z(t)$ can be discontinuous at the time instants t_k corresponding to the multiple-shooting time grid. E.g. in the semi-explicit case and if the algebraic equation $g(x, z, u) = 0$ depends on u, at the time instants t_k, a discontinuous input typically requires $z(t)$ to also be discontinuous.

As mentioned previously, the integrator can hide the algebraic variables from the NLP solvers and keep them as purely internal. However, one may want to use these variables in the cost function of the OCP, or impose some inequality constraints on them. In such a case, the algebraic states ought to be reported to the NLP solver, where they are regarded as functions of the decision variables s_k, q_k.

As illustrated in Figure 14.4.2, Multiple-Shooting with implicit integrators can be viewed as a two-level Newton scheme, where algebraic conditions are solved at two different levels. A natural alternative to this setup is then clearly to introduce the algebraic conditions underlying the numerical integrators into the NLP, and leave them to be solved by the NLP solver. Doing so leads us back to the Direct Collocation scheme, which we revisit next in the context of DAE-based optimal control problems.

14.4.3 Direct Collocation with Differential-Algebraic Equations

We will focus now on the deployment of the Direct Collocation method on such DAE-based optimal control problems. The principle here is extremely similar to those detailed in Section 13.3. However, there are a few important additional specific details arising from the presence of algebraic states and equations that need to be properly covered here. Let us briefly recall here the core principles of the direct collocation method. As detailed earlier in Section 13.3 and briefly recalled in Section 14.4.1 above, the differential state trajectories are approximated on each time intervals $[t_k,\ t_{k+1}]$ via the polynomials $p_k\,(t, v_k)$ linearly parametrized by the set of variables $v_k \in \mathbb{R}^{n(d+1)}$. For an explicit ODE $\dot{x} = f\,(x, u)$, the collocation equations then enforce:

- the continuity conditions of the differential states at the times t_k for $k = 0, \ldots, N-1$

$$p_k\,(t_{k+1}, v_k) - v_{k+1,0} = 0, \tag{14.39}$$

- the state dynamics at the times $t_{k,i}$ for $k = 0, \ldots, N-1$ and $i = 1, \ldots, d$

$$\dot{p}_k\,(t_{k,i}, v_k) = f(v_{k,i}, q_k).$$

Additional conditions are typically added as boundary conditions, e.g. $v_{0,0} - x_0 = 0$ to enforce the initial condition of the state trajectories.

The extension of the collocation equations for a semi-explicit DAE

$$\begin{aligned} \dot{x} &= f\,(x, z, u) \\ 0 &= g\,(x, z, u) \end{aligned}$$

follow the exact same philosophy, namely the collocation equations enforce:

- the continuity conditions of the differential states via (14.39) at the times $t_{0,\ldots,N-1}$.
- the state dynamics at the times $t_{k,i}$ for $k = 0, \ldots, N-1$ and $i = 1, \ldots, d$ via

$$\dot{p}_k\left(t_{k,i}, v_k\right) = f(v_{k,i}, z_{k,i}, q_k) \tag{14.40a}$$

$$0 = g\left(v_{k,i}, z_{k,i}, q_k\right) \tag{14.40b}$$

The collocation equations for a semi-implicit DAE therefore read as:

$$c_k\left(v_k, z_k, q_k, v_{k+1}\right) = \begin{bmatrix} \dot{p}_k\left(t_{k,1}, v_k\right) - f(v_{k,1}, q_k) \\ g\left(v_{k,1}, z_{k,1}, q_k\right) \\ \vdots \\ \dot{p}_k\left(t_{k,d}, v_k\right) - f(v_{k,d}, q_k) \\ g\left(v_{k,d}, z_{k,d}, q_k\right) \\ p_k\left(t_{k+1}, v_k\right) - v_{k+1,0} \end{bmatrix} = 0. \tag{14.41}$$

for $k = 0, \ldots, N-1$.

A few details ought to be properly stressed here. First, similarly to the observations made in Section 14.4.2, no continuity condition is enforced on the algebraic states, hence (14.39) applies to the differential state trajectories alone. Secondly, one ought to observe that the discretized algebraic states $z_{k,i}$ appear only for the indices $i = 1, \ldots, d$ in the collocation equations, i.e. the discrete algebraic states have *one degree of freedom less* that the discrete differential states $v_{k,i}$ which appear with the indices $i = 0, \ldots, d$ in the collocation equations. The extra degree of freedom granted to the differential state is actually required in order to be able to impose the continuity of the differential state trajectories, while the algebraic state trajectories are not required to be continuous. When building the NLP arising from a discretization of a DAE-based OCP using direct collocation, one ought to make sure that the adequate number of discrete algebraic states and discrete differential states are declared to the NLP solver. Indeed, e.g. introducing by mistake the unnecessary extra variables $z_{k,0}$ can create numerical difficulties in the solver, as these variables would be "free" in the NLP and their values not clearly fixed by the problem.

Building the collocation equations for DAEs in a generic implicit form

$$f\left(\dot{x}, x, z, u\right) = 0 \tag{14.42}$$

is a natural generalization of the constraints used in the case of a semi-explicit

DAE. In the general case, the collocation equations simply read as:

$$c_k\left(v_k, z_k, q_k, v_{k+1}\right) = \begin{bmatrix} f(\dot{p}_k\left(t_{k,1}, v_k\right), v_{k,1}, z_{k,1}, q_k) \\ \vdots \\ f(\dot{p}_k\left(t_{k,d}, v_k\right), v_{k,d}, z_{k,1}, q_k) \\ p_k\left(t_{k+1}, v_k\right) - v_{k+1,0} \end{bmatrix} = 0. \tag{14.43}$$

for $k = 0, \ldots, N-1$. The same observations as for the semi-explicit case hold for the general case.

Exercises

14.1 ...

15
Model Predictive Control

So far, we have regarded one single optimal control problem and focussed on ways to numerically solve this problem. Once we have computed such a solution, we might try to control the corresponding real process with the obtained control trajectory. This approach to use a precomputed control trajectory is called *open-loop control*. Unfortunately, the result will most probably be very dissatisfying, as the real process will typically not coincide completely with the model that we have used for optimization. If we wanted for example move a robot arm to a terminal point, the robot arm might end at a very different location than the model predicted. This is due to the difference of the model with the reality, sometimes called *model-plant-mismatch*. This mismatch might be due to modelling errors or external, unforeseen disturbances.

On the other hand, we might be able to observe the real process during its time development, and notice, for example, that the robot arm moves differently than predicted. This will allow us to correct the control inputs online in order to get a better performance; this procedure is called *feedback control* or *closed-loop control*. Feedback allows us to improve the practical performance of optimal control enormously. In its most basic form, we could use ad-hoc implementations of feedback that react to deviations from the planned state trajectory by basic control schemes such as a *proportional-integral (PI)* controller. On the other hand, we might use again optimal control techniques in order to react to disturbances of the state, by using *optimal feedback control*, which we had outlined in the Chapters 8 and 11 on dynamic programming (DP) and the HJB Equation. In the case of the moving robot arm this would result in the following behaviour: if during its motion the robot arm is strongly pushed by an external disturbance, it will not try to come back to its planned trajectory but instead adapt to the new situation and follow the new optimal trajectory. This is straightforward in the case of DP or HJB, where we have the optimal feedback control precomputed for all possible states. But as said,

these approaches are impossible to use for nontrivial state dimensions, i.e. systems with more than, say, 3-8 states. Thus, typically we cannot precompute the optimal feedback control in advance.

A possible remedy is to compute the optimal feedback control in *real-time*, or *online*, during the runtime of the process. In the case of the robot arm this means that after the disturbance, we would call our optimization solver again in order to quickly compute the new optimal trajectory. If we could solve this problem exactly and infinitely fast, we would get exactly the same feedback as in optimal feedback control. In reality, we have to work with approximations: first, we might simplify the optimal control problem in order to allow faster computation, e.g. by predicting only a limited amount of time into the future, and second, we might adapt our algorithms to the new task, namely that we have to solve optimization problems again and again. This task is called *real-time optimization* or *embedded optimization*, due to the fact that in many cases, the numerical optimization will be carried out on *embedded hardware*, i.e. processors that reside not in a desktop computer but e.g. in a feedback control system.

While this idea of *optimal feedback control via real-time optimization* sounds challenging or even impossible for the fast motion of robot arms, it is since decades industrial practice in the process control industry under the name of *Model Predictive Control (MPC)*. There, time scales are often in the range of minutes and allow ample time for each optimization. The main stream implementation of MPC can in discrete time roughly be formulated as follows: (1) observe the current state of the system $\bar{x}_0$, (2) predict and optimize the future behaviour of the process on a limited time window of N steps by solving an open-loop optimization problem starting at the state $\bar{x}_0$, (3) implement the first control action u_0^* at the real process, (4) move the optimization horizon one time step forward and repeat the procedure. MPC is sometimes also called *receding horizon control* due to this movement of the *prediction horizon*. The name *nonlinear MPC*, short *NMPC*, is reserved for the special case of MPC with underlying nonlinear dynamic systems, while linear MPC refers to MPC with linear system models. Note that NMPC leads typically to non-convex optimization problems while nearly all linear MPC formulations use convex cost and constraints.

Note that in the case of a time-invariant system and cost, the subsequent optimization problems differ only by the initial value $\bar{x}_0$ and nothing else, and thus, the MPC feedback is time-invariant as well. If we would be able to solve the problem with an infinite prediction horizon, we would obtain the stationary optimal feedback control. The limitation of the horizon to a finite length N

allows us to solve the problem numerically. If we choose N large enough, it will be a good approximation to the infinite horizon problem.

In this script, we do not focus on the different ways to formulate the MPC problem, but on its numerical solution by suitable real-time optimization methods. This and the next chapter follows the presentation given in [30] and [26] and focusses on the MPC optimal control problem.

15.1 NMPC Optimization Problem

Let us in this chapter regard the following simplified optimal control problem in discrete time augmented with algebraic equations.

$$\underset{x,\,z,\,u}{\text{minimize}} \quad \sum_{i=0}^{N-1} L(x_i, z_i, u_i) \quad + \quad E(x_N) \tag{15.1a}$$

$$\text{subject to} \qquad x_0 - \bar{x}_0 = 0, \tag{15.1b}$$

$$x_{i+1} - f(x_i, z_i, u_i) = 0, \quad i = 0, \ldots, N-1, \tag{15.1c}$$

$$g(x_i, z_i, u_i) = 0, \quad i = 0, \ldots, N-1, \tag{15.1d}$$

$$h(x_i, z_i, u_i) \leq 0, \quad i = 0, \ldots, N-1, \tag{15.1e}$$

$$r(x_N) \leq 0. \tag{15.1f}$$

Here, $x_i \in \mathbb{R}^{n_x}$ is the differential state, $z_i \in \mathbb{R}^{n_z}$ the algebraic state, and $u_i \in \mathbb{R}^{n_u}$ is the control. Functions f and g are assumed twice differentiable and map into $\mathbb{R}^{n_x}$ and $\mathbb{R}^{n_z}$, respectively. The algebraic state z_i is uniquely determined by (15.1d) when x_i and u_i are fixed, as we assume that $\frac{\partial g}{\partial z}$ is invertible everywhere.

We choose to regard this difference-algebraic system form because it covers several parametrization schemes for continuous time dynamic systems in differential algebraic equation (DAE) form, in particular direct multiple shooting with DAE relaxation [47] and direct collocation [66, 11]. Note that in the case of collocation, all collocation equations on a collocation interval would be collected within the function g and the collocation node values in the variables z_i, see the formulation in formula (**??**).

Here, the free variables are the differential state vector $x = (x_0, x_1 \ldots, x_{N-1}, x_N)$ at all considered time points and the algebraic and control vector on all but the last time points: $z = (z_0, z_1 \ldots, z_{N-1})$ and $u = (u_0, u_1 \ldots, u_{N-1})$.

The task in real-time optimization for NMPC is now the following: for a given value of $\bar{x}_0$, we need to approximately solve the above optimization problem as fast as possible, and of the obtained solution, it is the optimal value u_0

that we need fastest in order to provide the NMPC feedback. We might call the exact solution $u_0^*(\bar{x}_0)$ in order to express its dependence on the initial value $\bar{x}_0$. The only reason why we formulate and optimize the large optimization problem is because it delivers us this map $u_0^* : \mathbb{R}^{n_x} \to \mathbb{R}^{n_u}$, which is an approximation to the optimal feedback control.

Remark on fixed and free parameters: In most NMPC applications there are some *constant* parameters $\bar{p}$ that are assumed constant for the NMPC optimization, but that change for different problems, like $\bar{x}_0$. We do not regard them here for notational convenience, but note that they can be treated by state augmentation, i.e. regarded as constant system states with fixed initial value $\bar{p}$.

15.2 Nominal Stability of NMPC

Very often, one is interested in stabilizing the nonlinear dynamic system at a given set point for states and controls, which we might without loss of generality set to zero here. This steady state, that satisfies $f(0,0,0) = 0$, $g(0,0,0) = 0$ must be assumed to be feasible, i.e. $h(0,0,0) \leq 0$. One then often uses as stage cost the quadratic deviation from this set point, i.e., $L(x,u) = x^\top Q x + u^\top R u$ with positive definite matrices Q, R. It is important to note that this function is positive definite, i.e., $L(0,0) = 0$ and $L(x,u) > 0$ otherwise. In this case, one would ideally like to solve the infinite horizon problem with $N = \infty$ in order to obtain the true stationary optimal feedback control; this would automatically ensure stability, as the value function $J(x)$ can be shown to decrease along the trajectory of the nominal system in each time step by $-L(x_0, u^*(x_0))$ and can thus serve as a Lyapunov function. But as we have in practice to choose a finite N, the question arises how we can ensure nominal stability of NMPC nevertheless. One way due to [44, 52] is to impose a *zero terminal constraint* i.e. to require $x_N = 0$ as terminal boundary condition (15.1f) in the NMPC problem and to employ no terminal cost, i.e. $E(x_N) = 0$.

In this case of a zero terminal constraint, it can be shown that the value function J_0 of the finite horizon problem is a Lyapunov function that decreases by at least $-L(\bar{x}_0, u^*(\bar{x}_0))$ in each time step. To prove this, let us assume that $(x_0^*, z_0^*, u_0^*, x_1^*, z_1^*, u_1^*, \ldots, x_N^*)$ is the solution of the NMPC problem (15.1a)-(15.1f) starting with initial value $\bar{x}_0$. After application of this feedback to the nominal system, i.e. without model-plant-mismatch, the system will evolve exactly as predicted, and for the next NMPC problem the initial value $\bar{x}_0'$ will be given by $\bar{x}_0' = x_1^*$. For this problem, the *shifted* version of the previous

solution $(x_1^*, z_1^*, u_1^*, \ldots, x_N^*, 0, 0, 0)$ is a feasible point, and due to the zero values at the end, no additional cost arises at the end of the horizon. However, because the first stage cost term moved out of the horizon, we have that the cost of this feasible point of the next NMPC problem is reduced by exactly $-L(\bar{x}_0, u^*(\bar{x}_0))$. After further optimization, the cost can only be further reduced. Thus, we have proven that the value function J_0 is reduced along the trajectory, i.e. $J_0(\bar{x}_0') \leq J_0(\bar{x}_0) - L(\bar{x}_0, u^*(\bar{x}_0))$. More generally, one can relax the zero terminal constraint and construct combinations of terminal cost $E(x_N)$ and terminal inequalities $r(x_N) \leq 0$ that have the same property but are less restrictive, cf. e.g. [23, 25, 53].

15.3 Online Initialization via Shift

For exploiting the fact that NMPC requires the solution of a whole sequence of neighboring NLPs and not just a number of stand-alone problems, we have first the possibility to *initialize* subsequent problems efficiently based on previous information.

A first and obvious way to transfer solution information from one solved NMPC problem to the initialization of the next one is employing the shift that we used already in the proof of nominal stability above. It is motivated by the principle of optimality of subarcs, which, in our context, states the following: Let us assume we have computed an optimal solution $(x_0^*, z_0^*, u_0^*, x_1^*, z_1^*, u_1^*, \ldots, x_N^*)$ of the NMPC problem (15.1a)-(15.1f) starting with initial value $\bar{x}_0$. If we regard a shortened NMPC problem without the first interval, which starts with the initial value $\bar{x}_1$ chosen to be x_1^*, then for this shortened problem the vector $(x_1^*, z_1^*, u_1^*, \ldots, x_N^*)$ is the optimal solution.

Based on the expectation that the measured or observed true initial value for the shortened NMPC problem differs not much from x_1^* – i.e. we believe our prediction model and expect no disturbances – this "shrinking" horizon initialization is canonical, and it is used in MPC of batch or finite time processes, see e.g. [39, 28].

However, in the case of moving horizon problems, the horizon is not only shortened by removing the first interval, but also prolonged at the end by appending a new terminal interval – i.e. the horizon is moved forward in time. In the moving horizon case, the principle of optimality is thus not strictly applicable, and we have to think about how to initialize the appended new variables z_N, u_N, x_{N+1}. Often, they are obtained by setting $u_N := u_{N-1}$ or setting u_N as the steady state control. The states z_N and x_{N+1} are then obtained by forward simulation. In the case that zero is the steady state and we had a zero terminal

constraint, this would just result in zero values to be appended, as in the proof in the previous section. In any case, this transformation of the variables from one problem to the next is called "shift initialization". It is not as canonical as the "shrinking horizon" case, because the shifted solution is not optimal for the new undisturbed problem. However, in the case of long horizon lengths N we can expect the shifted solution to be a good initial guess for the new solution. Moreover, for most NMPC schemes with stability guarantee (for an overview see e.g. [53]) there exists a canonical choice of u_N that implies feasibility (but not optimality) of the shifted solution for the new, undisturbed problem. The shift initialization is very often used e.g. in [50, 13, 54, 32].

A comparison of shifted vs. non-shifted initializations was performed in [16] with the result that for autonomous NMPC problems that shall regulate a system to steady state, there is usually no advantage of a shift initialization compared to the "primitive" warm start initialization that leaves the variables at the previous solution. In the extreme case of short horizon lengths, it turns out to be even advantageous NOT to shift the previous solution, as subsequent solutions are less dominated by the initial values than by the terminal conditions. On the other hand, shift initialization are a crucial prerequisite in periodic tracking applications [32] and whenever the system or cost function are not autonomous.

15.4 Outline of Real-Time Optimization Strategies

In NMPC we would dream to have the solution to a new optimal control problem instantly, which is impossible due to computational delays. Several ideas help us to deal with this issue.

Offline precomputations: As consecutive NMPC problems are similar, some computations can be done once and for all before the controller starts. In the extreme case, this leads to an explict precomputation of the NMPC control law that has raised much interest in the linear MPC community [5], or a solution of the Hamilton-Jacobi-Bellman Equation, both of which are prohibitive for state and parameter dimensions above ten. But also when online optimization is used, code optimization for the model routines is often essential, and it is in some cases even possible to precompute and factorize Hessians or even Jacobians in Newton-type Optimization routines, in particular in the case of neighboring feedback control along reference trajectories [45, 22]. Also, preoptimized compilable computer code can be auto-generated that is specific to the family of optimization problems, which is e.g. in convex optimization pursued in [51].

Delay compensation by prediction: When we know how long our computa-

tions for solving an NMPC problem will take, it is a good idea *not* to address a problem starting at the current state but to simulate at which state the system will be when we will have solved the problem. This can be done using the NMPC system model and the open-loop control inputs that we will apply in the meantime [34]. This feature is used in many practical NMPC schemes with non-negligible computation time.

Division into preparation and feedback phase: A third ingredient of several NMPC algorithms is to divide the computations in each sampling time into a preparation phase and a feedback phase [29]. The more CPU intensive preparation phase (a) is performed with an old predicted state $\bar{x}_0$ before the new state estimate, say $\bar{x}_0'$, is available, while the feedback phase (b) then delivers quickly an *approximate* solution to the optimization problem for $\bar{x}_0'$. Often, this approximation is based on one of the tangential predictors discussed in the next chapter.

Iterating while the problem changes: A fourth important ingredient of some NMPC algorithms is the idea to work on the optimization problem while it changes, i.e., to never iterate the Newton-type procedure to convergence for an NMPC problem getting older and older during the iterations, but to rather work with the most current information in each new iteration. This idea is used in [50, 29, 56].

As a historical note, one of the first true online algorithms for nonlinear MPC was the *Newton-Type Controller of Li and Biegler* [49]. It is based on a sequential optimal control formulation, thus it iterates in the space of controls $u = (u_0, u_1, \ldots, u_{N-1})$ only. It uses an SQP type procedure with Gauss-Newton Hessian and line search, and in each sampling time, only one SQP iteration is performed. The transition from one problem to the next uses a shift of the controls $u^{\text{new}} = (u_1, \ldots, u_{N-1}, u_N^{\text{new}})$. The result of each SQP iterate is used to give an approximate feedback to the plant. As a sequential scheme without tangential predictor, it seems to have had sometimes problems with nonlinear convergence, though closed-loop stability was proven for open-loop stable processes [50].

In the next chapter, we will discuss several other real-time optimization algorithms in more detail that are all based on ideas from the field of parametric nonlinear optimization.

Exercises

15.1 In nonlinear model predictive control (NMPC), we repeatedly solve an optimal control problem (OCP) with changing data in order to derive an optimal feedback strategy for a controller. Since solving an NLP is an expensive operation, there is often a tradeoff between finding a better solution to the NLP or returning feedback to the system more frequently. In the most extreme case, we just do one iteration of the NLP solver for every feedback time. In the case of an SQP solver, this means solving a single QP.

Regard once again the simple OCP from Exercises 13.1, 13.2, 12.3 and 8.7.

$$\begin{aligned} \underset{x,\,u}{\text{minimize}} \quad & \int_0^\top x_1(t)^2 + x_2(t)^2 + u(t)^2 \, dt \\ \text{subject to} \quad & \dot{x}_1 = (1 - x_2^2)\, x_1 - x_2 + u, \qquad x_1(0) = 0, \\ & \dot{x}_2 = x_1, \qquad\qquad\qquad\qquad x_2(0) = 1, \\ & -1 \le\ u(t)\ \le 1, \end{aligned} \tag{15.2}$$

where $T = 10$ as earlier.

(a) In Exercise 13.2, you have implemented an SQP method to solve 15.2. Use this code as an inspiration for implementing a NMPC controller that uses a SQP solver. Remember that now the Gauss-Newton SQP only needs to make a single iteration, so in contrast to Exercise 13.2, you should allocate a QP solver instance just once and then call it multiple times.

(b) When just solving a single QP per NMPC iteration, it often make sense to divide the solution code into a *preparation phase* and a *feedback phase*. The preparation phase contains the part of the algorithm that can be calculated before we obtain measurements for the state of the system (i.e. the initial conditions of the ODE). This allows the controller to return feedback to the system faster. What part of the algorithm can be made part of preparation phase?

(c) Modify the solution to take more than one SQP iteration per NMPC iteration. Does it improve the controller?

References

[1] ACADO Toolkit. http://www.acadotoolkit.org, 2009–2016.

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

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

[4] R. Bellman. *Dynamic programming*. Princeton University Press, 1957.

[5] A. Bemporad, F. Borrelli, and M. Morari. Model Predictive Control Based on Linear Programming - The Explicit Solution. *IEEE Transactions on Automatic Control*, 47(12):1974–1985, 2002.

[6] A. Bemporad, F. Borrelli, and M. Morari. Min-max Control of Constrained Uncertain Discrete-Time Linear Systems. *IEEE Transactions on Automatic Control*, 2003. in press.

[7] 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.

[8] D. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, 3rd edition, 2005.

[9] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[10] J.T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. SIAM, 2nd edition, 2010.

[11] L. T. Biegler. Solution of dynamic optimization problems by successive quadratic programming and orthogonal collocation. *Computers and Chemical Engineering*, 8(3–4):243–248, 1984.

[12] Lorenz T. Biegler. *Nonlinear Programming*. MOS-SIAM Series on Optimization. SIAM, 2010.

[13] L.T. Biegler and J.B Rawlings. Optimization approaches to nonlinear model predictive control. In *Proc. 4th International Conference on Chemical Process Control - CPC IV*, pages 543–571. AIChE, 1991.

[14] J. Björnberg and M. Diehl. Approximate robust dynamic programming and robustly stable MPC. *Automatica*, 42(5):777–782, May 2006.

[15] J. Björnberg and M. Diehl. Approximate dynamic programming for generation of robustly stable feedback controllers. In H. G. Bock, editor, *Modeling, Simulation*

and Optimization of Complex Processes: Proceedings of the Third International Conference on High Performance Scientific Computing, pages 69–86. Springer Berlin Heidelberg, 2008.

[16] H. G. Bock, M. Diehl, D. B. Leineweber, and J.P. Schlöder. Efficient direct multiple shooting in nonlinear model predictive control. In F. Keil, W. Mackens, H. Voß, and J. Werther, editors, *Scientific Computing in Chemical Engineering II*, volume 2, pages 218–227. Springer, 1999.

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

[18] H.G. Bock. *Randwertproblemmethoden zur Parameteridentifizierung in Systemen nichtlinearer Differentialgleichungen*, volume 183 of *Bonner Mathematische Schriften*. Universität Bonn, Bonn, 1987.

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

[20] K.E. Brenan, S.L. Campbell, and L.R. Petzold. *The Numerical Solution of Initial Value Problems in Ordinary Differential-Algebraic Equations*. North Holland Publishing Co., Amsterdam, 1989.

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

[22] C. Büskens and H. Maurer. SQP-methods for solving optimal control problems with control and state constraints: adjoint variables, sensitivity analysis and real-time control. *Journal of Computational and Applied Mathematics*, 120(1–2):85–108, 2000.

[23] H. Chen and F. Allgöwer. A quasi-infinite horizon nonlinear model predictive control scheme with guaranteed stability. *Automatica*, 34(10):1205–1218, 1998.

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

[25] G. De Nicolao, L. Magni, and R. Scattolini. Stability and Robustness of Nonlinear Receding Horizon Control. In F. Allgöwer and A. Zheng, editors, *Nonlinear Predictive Control*, volume 26 of *Progress in Systems Theory*, pages 3–23, Basel Boston Berlin, 2000. Birkhäuser.

[26] M. Diehl. *Real-Time Optimization for Large Scale Nonlinear Processes*, volume 920 of *Fortschritt-Berichte VDI Reihe 8, Meß-, Steuerungs- und Regelungstechnik*. VDI Verlag, Düsseldorf, 2002. PhD Thesis.

[27] M. Diehl and J. Björnberg. Robust dynamic programming for min-max model predictive control of constrained uncertain systems. *IEEE Transactions on Automatic Control*, 49(12):2253–2257, December 2004.

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

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

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

[31] Moritz Diehl, Hans Georg Bock, Holger Diedam, and Pierre-Brice Wieber. *Fast Motions in Biomechanics and Robotics*, volume 340, chapter Fast Direct Multiple Shooting Algorithms for Optimal Robot Control, pages 65–93. Springer, 2006.

[32] Moritz Diehl, Lalo Magni, and Giuseppe De Nicolao. Efficient NMPC of unstable periodic systems using approximate infinite horizon closed loop costing. *Annual Reviews in Control*, 28(1):37–45, 2004.

[33] 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.

[34] R. Findeisen and F. Allgöwer. Computational Delay in Nonlinear Model Predictive Control. Proc. Int. Symp. Adv. Control of Chemical Processes, ADCHEM, 2003.

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

[36] 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.

[37] A. Griewank and Ph.L. Toint. Partitioned variable metric updates for large structured optimization problems. *Numerische Mathematik*, 39:119–137, 1982.

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

[39] A. Helbig, O. Abel, and W. Marquardt. Model predictive control for on-line optimization of semi-batch reactors. In *Proceedings of the American Control Conference (ACC)*, pages 1695–1699, Philadelphia, 1998.

[40] G.A. Hicks and W.H. Ray. Approximation methods for optimal control systems. *Can. J. Chem. Engng.*, 49:522–528, 1971.

[41] 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.

[42] C.N. Jones and M. Morari. Polytopic approximation of explicit model predictive controllers. *IEEE Transactions on Automatic Control*, 55(11):2542–2553, 2010.

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

[44] S.S. Keerthi and E.G. Gilbert. Optimal infinite-horizon feedback laws for a general class of constrained discrete-time systems: Stability and moving-horizon approximations. *Journal of Optimization Theory and Applications*, 57(2):265–293, 1988.

[45] P. Krämer-Eis and H.G. Bock. Numerical Treatment of State and Control Constraints in the Computation of Feedback Laws for Nonlinear Control Problems. In P. Deuflhard et al., editor, *Large Scale Scientific Computing*, pages 287–306. Birkhäuser, Basel Boston Berlin, 1987.

[46] 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.

[47] D. B. Leineweber, I. Bauer, A. A. S. Schäfer, H. G. Bock, and J. P. Schlöder. An efficient multiple shooting based reduced SQP strategy for large-scale dynamic process optimization. (Parts I and II). *Computers and Chemical Engineering*, 27:157–174, 2003.

[48] 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.

[49] W. C. Li and L. T. Biegler. Multistep, Newton-type control strategies for constrained nonlinear processes. *Chem. Eng. Res. Des.*, 67:562–577, 1989.

[50] W.C. Li and L.T. Biegler. Newton-Type Controllers for Constrained Nonlinear Processes with Uncertainty. *Industrial and Engineering Chemistry Research*, 29:1647–1657, 1990.

[51] J. Mattingley, Y. Wang, and Stephen Boyd. Code generation for receding horizon control. In *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design*, pages 985–992, Yokohama, Japan, 2010.

[52] D. Q. Mayne and H. Michalska. Receding horizon control of nonlinear systems. *IEEE Transactions on Automatic Control*, 35(7):814–824, 1990.

[53] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 26(6):789–814, 2000.

[54] A. M'hamdi, A. Helbig, O. Abel, and W. Marquardt. Newton-type Receding Horizon Control and State Estimation. In *Proc. 13rd IFAC World Congress*, pages 121–126, San Francisco, 1996.

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

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

[57] M.R. Osborne. On shooting methods for boundary value problems. *Journal of Mathematical Analysis and Applications*, 27:417–433, 1969.

[58] E. N. Pistikopoulos, V. Dua, N. A. Bozinis, A. Bemporad, and M. Morari. On-line optimization via off-line parametric optimization tools. *Computers and Chemical Engineering*, 24:183–188, 2000.

[59] L.S. Pontryagin, V.G. Boltyanski, R.V. Gamkrelidze, and E.F. Miscenko. *The Mathematical Theory of Optimal Processes*. Wiley, Chichester, 1962.

[60] S. Sager, H. G. Bock, and M. Diehl. The integer approximation error in mixed-integer optimal control. *Mathematical Programming (Series A)*, 133:1—23, 2012.

[61] R. W. H. Sargent and G. R. Sullivan. The development of an efficient optimal control package. In J. Stoer, editor, *Proceedings of the 8th IFIP Conference on Optimization Techniques (1977), Part 2*, pages 158–168, Heidelberg, 1978. Springer.

[62] A.A.S. Schäfer. *Efficient reduced Newton-type methods for solution of large-scale structured optimization problems with application to biological and chemical processes*. PhD thesis, University of Heidelberg, 2005.

[63] J.P. Schlöder. *Numerische Methoden zur Behandlung hochdimensionaler Aufgaben der Parameteridentifizierung*, volume 187 of *Bonner Mathematische Schriften*. Universität Bonn, Bonn, 1988.

[64] M. C. Steinbach. A structured interior point SQP method for nonlinear optimal control problems. In R. Bulirsch and D. Kraft, editors, *Computation Optimal Control*, pages 213–222, Basel Boston Berlin, 1994. Birkhäuser.

[65] M.J. Tenny, S.J. Wright, and J.B. Rawlings. Nonlinear model predictive control via feasibility-perturbed sequential quadratic programming. *Computational Optimization and Applications*, 28(1):87–121, April 2004.

[66] T.H. Tsang, D.M. Himmelblau, and T.F. Edgar. Optimal control via collocation and non-linear programming. *International Journal on Control*, 21:763–768, 1975.

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

[68] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.