Lecture Notes
Introduction to Optimization

Fall Semester 2022

Roland Herzog*

2023-03-30

*Interdisciplinary Center for Scientific Computing, Heidelberg University, 69120 Heidelberg, Germany (roland.herzog@iwr.uni-heidelberg.de, https://scoop.iwr.uni-heidelberg.de/team/rherzog/).

Material for 4 lectures of about 2 hours each.

Contents

§ 1 Introduction

Optimization is about solving problems of the form

$$\left.\begin{array}{rlll} \text{Minimize} & f(x) & \text{where } x \in \mathbb{R}^n & \text{(objective (function))} \\ \text{subject to} & g_i(x) \leq 0 & \text{with } i = 1, \ldots, n_{\text{ineq}} & \text{(inequality constraints)} \\ \text{and} & h_j(x) = 0 & \text{with } j = 1, \ldots, n_{\text{eq}}. & \text{(equality constraints)} \end{array}\right\} \tag{1.1}$$

x is termed the **optimization variable**, **decision variable**, or simply the **variable** of the problem. Moreover, we consider problems where

- the functions $f, g_i, h_j \colon \mathbb{R}^n \to \mathbb{R}$ are sufficiently smooth,
- the number n_{ineq} of inequality constraints and the number n_{eq} of equality constraints are finite (possibly zero).

More precisely, problems of type (1.1) fall into the realm of continuous **optimization**. In case some or all components of x are to be sought in a discrete set (such as the integers $\mathbb{Z}$), we have a mixed discrete-continuous problem or even a fully discrete problem, which we do not consider in this course.

Solving an optimization problem usually includes the following steps:

(1) Modeling the problem.
This step can be one of the most complex ones in the entire process. In particular, we need to answer the following questions:

   (a) What do we use as our optimization variable(s)? What is the significance of each component of x?
   (b) How do we formulate the objective function? What is its significance?
   (c) Do we need to formulate constraints? How do we model them? What is the significance of each of the constraints?
   (d) Are there any constants in the problem? What are their values and what is their significance?

(2) Understanding the problem's characteristics.
There are various dimensions along which you can classify optimization problems. Here are some examples:

   (a) What is the dimension n of the optimization variable $x \in \mathbb{R}^n$?
   (b) Is the problem constrained or unconstrained?
   (c) Is the objective linear or nonlinear? Are the constraints (if any) linear or nonlinear?
   (d) Is the problem convex or non-convex?

   The answer to these questions has an impact on the next step, the selection of a suitable solver. The more specifically the solver fits the problem, the more efficient the solution process usually

is. For instance, a linear optimization problem (where the objective and all constraints are affine functions of x) is best solved using a solver tailored to linear optimization problems (such as an implementation of the simplex method). It is "overkill" to solve a linear optimization problem using a solver capable of solving general nonlinear constrained problems and the more general optimization procedure will typically perform poorer than specialized routines (although it should usually succeed).

(3) Picking a numerical solver.
Pick your solver depending on the problem's characteristic as well as availability/license and your preferred programming language.

The following resources can help you identify suitable solvers:

- Hans Mittelmann's Decision Tree for Optimization Software
- the Optimization Tree of the NEOS project (a free internet-based service for solving optimization problems)
- the list of available solvers in the COIN-OR project
- the respective documentation of various optimization libraries, such as
  - MATLAB's Optimization Toolbox
  - the `scipy.optimize` PYTHON package
  - the CASADI package, which has MATLAB, PYTHON and C++ bindings

(4) Implementing the functions describing the problem for the solver.
You will need to implement the objective and constraint functions in a format suitable for the solver selected in the respective programming language (such as MATLAB, PYTHON, JULIA, or C++ for instance).

There are also specific modeling languages for optimization problems, e. g., AMPL, which are, in principle, independent of the solver. However, not all solvers will accept all input formats.

Finally, some solver packages such as CASADI (PYTHON, MATLAB) and JUMP (JULIA), include their own modeling paradigms which facilitate the formulation of optimization problems.

(5) Running the solver.
Launch the solver selected for the problem. In some cases, you may want to modify the default settings (such as subalgorithms, tolerances, maximum number of iterations etc.). Most solvers will accept an initial guess (often termed x_0) for the solution, which is often beneficial to set to a reasonable value.

(6) Checking and interpreting the result.
Carefully check the flag or message returned by the solver, i. e., whether it has converged or not. When successful, interpret and verify the result. Check that the solution "makes sense". Are all the constraints satisfied as expected? Are you happy with the result, or did you perhaps expect something different? Did you forget to include certain constraints? (A classical example is that you expect the solution to be non-negative but forget to model this constraint.)

The above steps are not necessarily run through sequentially. For instance, it might turn out later in the process that a different formulation of the problem than originally selected is more favorable for the solver, so one needs to go back. Or we might figure out from the solution that we forgot to model certain constraints, which we then include and run the solver again.

It is also important to understand that, given a problem in text form, the above steps do not have a predetermined outcome. It is up to us to **model the problem** so that we can identify (or design ourselves) a numerical solver that can efficiently solve the problem. Your modeling skills will get better with practice, and this process is partially based on trial-and-error.

Example 1.1 (Choice of optimization variables)**.** *This examples illustrates that there is no single correct answer to modeling a problem. For instance, what we choose to be our optimization variables is a modeling decision. Consider, for instance, an investor, who seeks to distribute a certain amount of money M between a bank account with fixed interest and a stock investment with uncertain future stock prices. The investor's objective may be to maximize the expected amount of their wealth at a certain time in the future.*

The investor may choose optimization variables

$$\begin{aligned} x_1 &\quad \text{amount of money invested in the bank account,} \\ x_2 &\quad \text{amount of money invested in the stock.} \end{aligned}$$

We are not specifying the problem any further here, but we mention that it is likely going to contain the constraints $x_1 \geq 0$, $x_2 \geq 0$ and $x_1 + x_2 = M$.

Alternatively, instead of determining how much money to put in either investment, the investor may describe their optimization variable using a single decision, namely what fraction ("percentage") of the money M is going to be invested in the bank account. That is, the investor may choose their optimization variable as

$$x_1 \quad \text{fraction of } M \text{ to be invested in the bank account,}$$

which is subject to the constraint $0 \leq x_1 \leq 1$. Naturally, the remainder fraction $1 - x_1$ of M is going to be invested in the stock.

We now discuss what it means to solve an optimization problem, i. e., to find and recognize minimizers.

Definition 1.2 (Basic concepts)**.**

(*i*) *The set of points $x \in \mathbb{R}^n$ satisfying all the constraints, i. e.,*

$$F := \left\{ x \in \mathbb{R}^n \,\middle|\, g_i(x) \leq 0 \text{ for all } i = 1, \dots, n_{\text{ineq}} \text{ and } h_j(x) = 0 \text{ for all } j = 1, \dots, n_{\text{eq}} \right\} \tag{1.2}$$

*is termed the **feasible set**. Every $x \in F$ is called a **feasible point**.*

(*ii*) *The inequality $g_i(x) \leq 0$ is said to be **active** at the point x in case of $g_i(x) = 0$. It is called **inactive** in case $g_i(x) < 0$. It is called **violated** in case $g_i(x) > 0$.*

(*iii*) *A point $x^* \in F$ is a* ***global minimizer*** *or* ***global solution*** *of* (1.1) *if*

$$f(x^*) \leq f(x) \text{ for all } x \in F.$$

(*iv*) *A point $x^* \in F$ is a* ***local minimizer*** *or* ***local solution*** *of* (1.1) *if there exists a (potentially small) neighborhood of x^**[1] *$U(x^*)$ such that*

$$f(x^*) \leq f(x) \text{ for all } x \in F \cap U(x^*).$$

While Definition 1.2 states what minimizers are, the definition is not very helpful when it comes to checking whether a given point x^* is indeed, say, a local minimizer. In order to verify this on the grounds of the definition would require us to compare the value of the objective $f(x^*)$ with the value of the objective $f(x)$ at an (uncountable!) number of nearby points x! This is clearly impractical.

Therefore, we need to resort to different ways of verifying that a certain point is a minimizer, and also of excluding certain points from the list of candidate minimizers. To this end, we can use derivative-based optimality conditions. Optimality conditions come in two flavors:

(1) A **necessary optimality condition** (NC) is a statement of the following form: "Suppose that x^* is a local minimizer of problem (1.1). Then (NC) holds at x^*."

    Necessary optimality conditions can be used in the following ways:

    (a) Finding points $x^* \in \mathbb{R}^n$ that satisfy (NC) means to compile a list of candidates for local minimizers. Those can then by further examined by other means.

    (b) A necessary optimality condition can also be read as:[2] "Suppose that (NC) does *not* hold at x^*, then x^* cannot be a local minimizer." Consequently, necessary optimality conditions can be used to exclude points from the search for local minimizers.

(2) A **sufficient optimality condition** (SC) is a statement of the following form: "Suppose that (SC) holds at x^*. Then x^* is a local minimizer."

    Sufficient optimality conditions thus allow us to verify candidates as local minimizers.

We will discuss appropriate forms of necessary and sufficient optimality conditions for the particular type of problem in each of the following sections.

[1] Note that the definition is equivalent if we were to replace "neighborhood of x^*" by "ball around x^*", i. e., $B_\varepsilon(x^*) = \{x \in \mathbb{R}^n \mid \|x - x^*\|_2 < \varepsilon\}$.

[2] This is called the contraposition of the original statement.

§ 2 Unconstrained Optimization and Parameter Estimation

As the name suggests, unconstrained optimization is about minimizing an objective in the absence of any equality or inequality constraints. In this case, our generic problem (1.1) reduces to

$$\text{Minimize} \quad f(x) \quad \text{where } x \in \mathbb{R}^n. \tag{2.1}$$

For this problem, "derivative equals zero" is a necessary optimality condition. Points which satisfy $f'(x) = 0$ are called **stationary points**.

Theorem 2.1 (Necessary optimality condition for unconstrained problems)**.**
Suppose that x^* is a local minimizer for (1.1). Then x^* is a stationary point, i. e., $f'(x^*) = 0$ holds.

Sufficient optimality conditions are usually based on second-order derivatives, and we do not state them here. Instead, we proceed to sketch some algorithms. Indeed, algorithms usually find stationary points, which may or may not be local minimizers. Nevertheless, we use the term "optimization algorithms" and continue to speak of "minimizing the objective".

§ 2.1 Algorithmic Sketches for Generic Unconstrained Problems

Optimization algorithms proceed in a sequential way and break down problem (1.1) into a sequence of simpler problems. In doing so, they generate a sequence $x^{(k)}$ of iterates, which (under some conditions) converges to a stationary point x^*.

The vast majority of algorithms for (1.1) proceeds as follows. At the current iterate $x^{(k)}$, they form a **quadratic model** of the objective. A quadratic model is a quadratic (i. e., second-order) polynomial that shares some properties with the true objective in the vicinity of the current iterate $x^{(k)}$. The quadratic model at $x^{(k)}$ is of the form (see Figure 2.1 for an illustration in 1D)

$$q^{(k)}(x) := \underbrace{f(x^{(k)})}_{\text{constant term}} + \overbrace{f'(x^{(k)})(x - x^{(k)})}^{\text{linear term}} + \underbrace{\frac{1}{2}(x - x^{(k)})^\mathsf{T} H^{(k)} (x - x^{(k)})}_{\text{quadratic term}}. \tag{2.2}$$

The matrix $H^{(k)}$ is called the **model Hessian** and it is designed to be symmetric.

At $x = x^{(k)}$, the model $q^{(k)}$ shares its function value and its derivative with the true objective f. (**Quiz 2.1:** Can you show this?)

Algorithms proceed by minimizing the current quadratic model $q^{(k)}$, or at least they look for a stationary point of the model:

$$[q^{(k)}]'(x) = f'(x^{(k)}) + (x - x^{(k)})^\mathsf{T} H^{(k)} \overset{!}{=} 0.$$

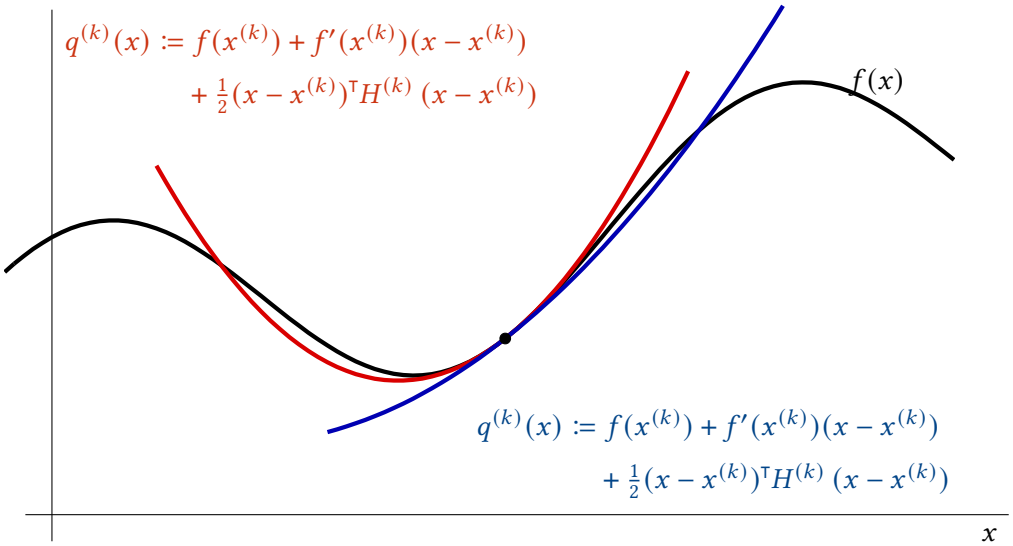


Figure 2.1: A function $f := \mathbb{R} \to \mathbb{R}$ and two possible quadratic models $q^{(k)}$ about the point $x^{(k)}$ (black dot). The red model is the second-order Taylor polynomial of f at $x^{(k)}$. In other words, the red model uses $H^{(k)} = f''(x^{(k)})$. The blue model instead uses an "incorrect" Hessian $H^{(k)} \neq f''(x^{(k)})$.

This amounts to solving the linear(!) system of equations

$$H^{(k)}(x - x^{(k)}) = -\nabla f(x^{(k)}).$$

Usually this is written in the form

$$H^{(k)} d = -\nabla f(x^{(k)}) \tag{2.3}$$

whose solution gives us an update direction d in the space of optimization variables.

In practical algorithms, this direction is then scaled with a suitable step length $\alpha^{(k)} > 0$, which accounts for the deviation of the quadratic model from the original objective and gives us the update

$$x^{(k+1)} := x^{(k)} + \alpha^{(k)} d.$$

The step length $\alpha^{(k)}$ has the purpose of making the step between consecutive iterates small when the model does not agree well with the true objective. This is done in an effort to obtain robust convergence properties regardless of the quality of the initial guess provided by the user. Since $\alpha^{(k)}$ is determined by a line search procedure, in which trial step sizes are proposed and evaluated until a satisfactory one is found, one speaks of a **line search algorithm**. An alternative concept is to limit the size of d in the first place by constraining the norm $\|d\|_2 \leq \Delta^{(k)}$ when minimizing the model (2.2). The solution of this problem is more complex than solving a linear system (2.3). This leads to **trust-region algorithms**, and $\Delta^{(k)}$ is the trust-region radius.

Depending on the choice of the model Hessians $H^{(k)}$ implemented in an algorithm, we refer to it by different names:[3]

[3]Combined with the choice above, one then obtains a *line-search quasi-Newton method*, or a *trust-region Newton algorithm*, for instance. The combination *trust-region gradient method* is, however, uncommon.

gradient method (steepest descent)	$H^{(k)} = \mathrm{Id}$ (identity matrix)	linear convergence
preconditioned gradient method	$H^{(k)} = M$	linear convergence
quasi-Newton method	$H^{(k)}$ varies	often superlinear convergence
Newton method	$H^{(k)} = f''(x^{(k)})$	superlinear convergence

Gradient methods are simple in the sense that the linear system (2.3) solved in each iteration has a constant coefficient matrix, which can be exploited. Gradient methods, however, typically exhibit slow convergence. On the other hand, Newton methods convergence faster once they are near a minimizer; however, the evaluation of the objective's second derivative matrix in every iteration is usually costly. In practice, quasi-Newton methods are a good compromise, in which $H^{(k)}$ tries to capture some of the properties of the objective's second derivative without actually computing it.

We close this section by mentioning what functions a user has to implement in order to run an algorithm for solving an unconstrained problem (2.1):

objective	$f(x)$	
derivative	$f'(x)$	optional (finite difference fallback)
2nd derivative	$f''(x)$ or $f''(x)\,d$	optional, only for Newton methods

The evaluation of the first-order derivative $f'(x)$, although required by all algorithms, is usually optional for the user. When it is not provided, the algorithm will fall back to an internal approximation of $f'(x)$ by finite differences. While this may sound convenient, it may be very time consuming and introduce inaccuracies. If possible, the user should provide first-order derivatives. To facilitate the task, they may use **algorithmic differentiation** (AD) for this purpose. This is beyond the scope of these notes.

Example 2.2 (Rosenbrock problem)**.** *The **Rosenbrock problem** is a classical test example in unconstrained optimization:*

$$\textit{Minimize} \quad f(x,y) := (a-x)^2 + b\,(y-x^2)^2 \quad \textit{where } (x,y) \in \mathbb{R}\times\mathbb{R}$$

It has a global minimizer at $(x,y) = (a,a^2)$, with an optimal function value of 0. A typical choice of the parameters is $(a,b) = (1,100)$. A plot of the function can be found in Figure 2.2.

§ 2.2 Algorithmic Sketches for Unconstrained Least-Squares Problems

Many unconstrained optimization problems are in fact of the form

$$\text{Minimize} \quad f(x) := \sum_{i=1}^{M} [r_i(x)]^2 = \|r(x)\|_2^2 \quad \text{where } x \in \mathbb{R}^n. \tag{2.4}$$

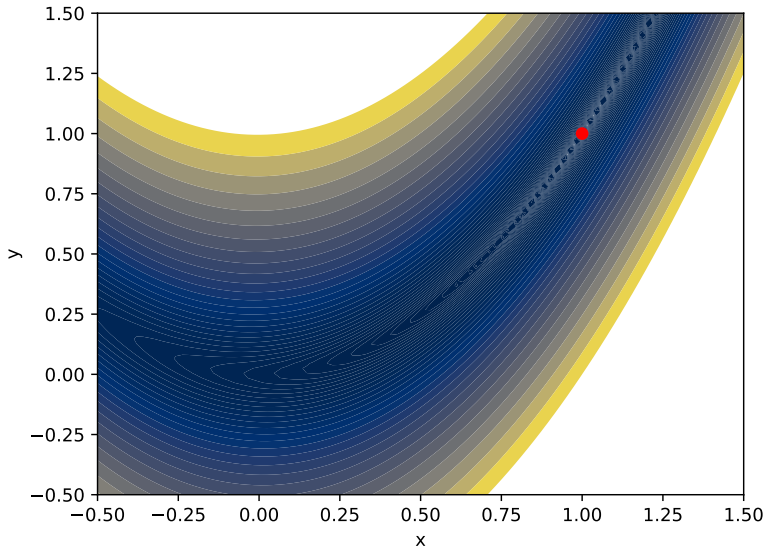


Figure 2.2: Contour plot of the Rosenbrock "banana" function (Example 2.2) with parameters $(a, b) = (1, 100)$, and its global minimizer.

These are called **least-squares problems**. The primary source of least-squares problems lies in **parameter estimation**, also known as **parameter identification**, **model calibration**, **curve fitting**, or **data assimilation**.

In parameter estimation one is given a model function m which describes a relation $\eta = m(\xi)$ between *independent variables* (model input) $\xi \in \mathbb{R}^r$ and *dependent variables* $\eta \in \mathbb{R}$ (model output). The model in fact contains yet unknown parameters $x \in \mathbb{R}^n$, i. e., we have a relation of the form

$$\eta = m(x; \xi).$$

Each measurement pair (ξ_i, η_i) contributes a **residual**

$$r_i(x) := \underbrace{m(x; \xi_i)}_{\text{model prediction}} - \overbrace{\eta_i}^{\text{measurement}}, \tag{2.5}$$

which describes the discrepancy between the value predicted by the model and the actual measurement η_i pertaining to the i-th input ξ_i.

The least-squares problem (2.4) describes our desire to choose the parameters $x \in \mathbb{R}^n$ in such a way that the model will fit the available measurement pairs $(\xi_i, \eta_i) \in \mathbb{R}^r \times \mathbb{R}$, $i = 1, \ldots, M$, in the best possible way, in the sense of least squares.

There are statistical reasons why — at least for measurements η_i perturbed by additive random measurement errors, which are uncorrelated and identically normally distributed — the solution of the least-squares problem (2.1) yields the best estimate of the parameters. When, more generally, the

measurement errors are still uncorrelated but may have non-identical variances σ_i^2, we should consider the **weighted least-squares problem**

$$\text{Minimize} \quad f(x) := \sum_{i=1}^{M} \frac{1}{\sigma_i^2} [r_i(x)]^2 = \|r(x)\|_{\Sigma^{-1}}^2 \quad \text{where } x \in \mathbb{R}^n, \tag{2.6}$$

where $\Sigma = \operatorname{diag}(\sigma_1^2, \dots, \sigma_M^2)$ is the measurement covariance matrix. However, we do not specifically consider the form (2.6) here. After all, (2.6) can be brought into the form (2.4) simply by scaling the residuals by $1/\sigma_i$.

We could use any of the algorithmic ideas from § 2.1 to solve (2.4). However, it is usually more efficient to use a different algorithm that exploits the particular structure of the objective in (2.4). In order to understand this, we determine the first- and second-order derivatives of the objective from (2.4) using the chain rule. We expect that the derivative of the residual function r will appear, i. e., its Jacobian, which we denote by r' or by J:

$$J(x) := \left[\frac{\partial}{\partial x_j} r_i(x)\right]_{ij} = \begin{bmatrix} | & & | \\ \frac{\partial}{\partial x_1} r(x) & \cdots & \frac{\partial}{\partial x_n} r(x) \\ | & & | \end{bmatrix} = \begin{bmatrix} — \nabla r_1(x)^\mathsf{T} — \\ \vdots \\ — \nabla r_m(x)^\mathsf{T} — \end{bmatrix} \in \mathbb{R}^{M \times n}. \tag{2.7}$$

For cosmetic reasons it is practical to divide the objective in (2.4) by 2. This does, of course, not influence where the minimizers lie.

$$f(x) = \frac{1}{2}\|r(x)\|_2^2 = \frac{1}{2} r(x)^\mathsf{T} r(x) \qquad \text{objective} \tag{2.8a}$$

$$\nabla f(x) = J(x)^\mathsf{T} r(x) = \sum_{i=1}^{M} r_i(x)\, \nabla r_i(x) \qquad \text{first-order derivative} \tag{2.8b}$$

$$f''(x) = J(x)^\mathsf{T} J(x) + \sum_{i=1}^{M} r_i(x)\, r_i''(x) \qquad \text{second-order derivative.} \tag{2.8c}$$

With the help of this data one *could* set up a quadratic model of the objective f at the current iterate $x^{(k)}$, determine a search direction $d^{(k)}$ by minimizing it, then find a step length $\alpha^{(k)}$, take the step etc. The additional knowledge, however, that we are dealing with a least-squares problem can be exploited even better algorithmically. We describe this using the popular **Levenberg-Marquardt method**[4] as an example.

The Levenberg-Marquardt method sets up a certain quadratic model about the point $x^{(k)}$:

$$\begin{aligned} q_{\mathrm{LM}}^{(k)}(d) :=& f(x^{(k)}) + \nabla f(x^{(k)})^\mathsf{T} d + \frac{1}{2} d^\mathsf{T} H_{\mathrm{LM}}^{(k)}\, d \\ =& \frac{1}{2} r(x^{(k)})^\mathsf{T} r(x^{(k)}) + r(x^{(k)})^\mathsf{T} J(x^{(k)})\, d + \frac{1}{2} d^\mathsf{T} H_{\mathrm{LM}}^{(k)}\, d \quad \text{by (2.8).} \end{aligned} \tag{2.9}$$

The Levenberg-Marquardt method uses

$$H_{\mathrm{LM}}^{(k)} := J(x^{(k)})^\mathsf{T} J(x^{(k)}) + \lambda^{(k)} \operatorname{Id} \tag{2.10}$$

as the model Hessian. Here $\lambda^{(k)} > 0$ is a positive number. This has the following advantages:

[4] proposed by Levenberg (1944), rediscovered by Marquardt (1963)

(i) One uses the "good" first part $J(x)^\mathsf{T} J(x)$ of the true Hessian (2.8c). This part is at least positive semidefinite and it can be computed without further effort since we need $J(x)$ anyway for the first-order derivative $\nabla f(x)$.

(ii) The "bad" part $\sum_{i=1}^{M} r_i(x)\, r_i''(x)$ of the Hessian (2.8c) is omitted. Since this part depends on the second-order derivatives $r_i''(x)$, it is expensive to evaluate, and it can potentially destroy the positive definiteness. Moreover we hope that the residuals $r_i(x^*)$ at the solution x^* will be small (near zero), i. e., we hope that we will have a good fit, and this part of the Hessian can be disregarded.

(iii) Instead one adds an auxiliary term $\lambda^{(k)}$ Id. In view of $\lambda^{(k)} > 0$, positive definiteness of the model Hessian $H_{\mathrm{LM}}^{(k)}$ is ensured, which implies that the linear problems that need to be solved in the algorithms substeps are uniquely solvable.

As with the methods of § 2.1, one is interested in the minimizer of the quadratic model (2.9) in the k-th iteration. This minimizer is unique and it can be calculated by solving the following linear system of equations, compare (2.3),

$$H_{\mathrm{LM}}^{(k)}\, d^{(k)} = -\nabla f(x^{(k)}). \tag{2.11}$$

In more explicit terms, see (2.10) and (2.8b), we can write

$$\left[J(x^{(k)})^\mathsf{T} J(x^{(k)}) + \lambda^{(k)}\,\mathrm{Id}\right] d^{(k)} = -J(x^{(k)})^\mathsf{T} r(x^{(k)}). \tag{2.12}$$

The size of the Levenberg-Marquardt parameter $\lambda^{(k)}$ implicitly determines the length of the step d. (**Quiz 2.2:** What happens in the extreme cases $\lambda^{(k)} \to 0$ and $\lambda^{(k)} \to \infty$, respectively?) $\lambda^{(k)}$ itself is controlled depending on how well the model (2.9) was able to predict the actual decrease in the values of the objective (2.4).

We mention what functions a user has to implement in order to run a dedicated least-squares algorithm for solving an problem of type (2.4):

residual	$r(x)$	
Jacobian	$J(x)$	optional (finite difference fallback)

Often times, the user has a choice (through different interfaces to the algorithm) to specify the model function m and the set of measurement pairs $(\xi_i, \eta_i) \in \mathbb{R}^r \times \mathbb{R}$, $i = 1, \ldots, M$, and have the method itself evaluate the residual function with components (2.5) and Jacobian

$$J(x) = r'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} m(x;\xi_1) & \cdots & \frac{\partial}{\partial x_n} m(x;\xi_1) \\ \vdots & & \vdots \\ \frac{\partial}{\partial x_1} m(x;\xi_M) & \cdots & \frac{\partial}{\partial x_n} m(x;\xi_M) \end{bmatrix} = \begin{bmatrix} — \nabla m(x;\xi_1)^\mathsf{T} — \\ \vdots \\ — \nabla m(x;\xi_M)^\mathsf{T} — \end{bmatrix}.$$

In this case, the user provides

model	$m(x;\xi)$	
model gradient	$\nabla m(x;\xi)$	optional (finite difference fallback)

and the set of measurement pairs (ξ_i, η_i). Again, when derivatives are not provided, most algorithms will fall back to finite differencing.

Example 2.3 (Parameter estimation)**.** *The position (x, y) of an airplane should be determined by radio bearing. To this end, the directions towards several radio beacons are measured from the airplane. The positions (X_i, Y_i) of the beacons are known and the directions are measured as angles α_i to the x-axis:*

beacon i	*beacon's position (X_i, Y_i)*	*angle α_i*
1	$(8, 6)$	$38°$
2	$(-3, -3)$	$220°$
3	$(1, 0)$	$222°$
4	$(8, -3)$	$300°$

The aim is to find the position (x, y) of the airplane for which the discrepancies between the predicted and measured angles are as small as possible in the least squares sense.

In this example, we have $M = 4$ individual measurements. We require a model which maps the independent variable $\xi = (X, Y) \in \mathbb{R}^2$ to the dependent variable ($\eta = \alpha \in \mathbb{R}$), as a function of the unknown airplane position (x, y). A model which achieves this is given by

$$m(\underbrace{(x, y)}_{\text{unknown parameter}}; \overbrace{(X, Y)}^{\text{independent variable}}) = atan2(X - x, Y - y) \cdot \frac{180°}{\pi},$$

which returns the angle of the vector $\begin{pmatrix} X-x \\ Y-y \end{pmatrix}$ against the x-axis in degrees. In fact, since $atan2$ returns angles in the interval $(-\pi, \pi]$, while our measurements are in $0°$ to $360°$, we have to normalize $atan2$'s output to $[0, 2\pi)$ beforehand; see the listing in § 7.3 for details.

The solution obtained with the code from § 7.3 is shown in Figure 2.4.

End of Class 1

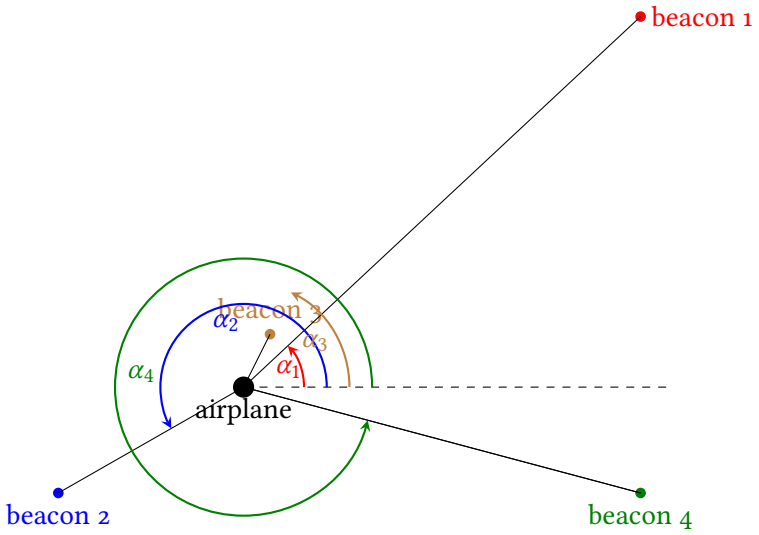


Figure 2.3: Illustration of the airplane position parameter estimation problem from Example 2.3.

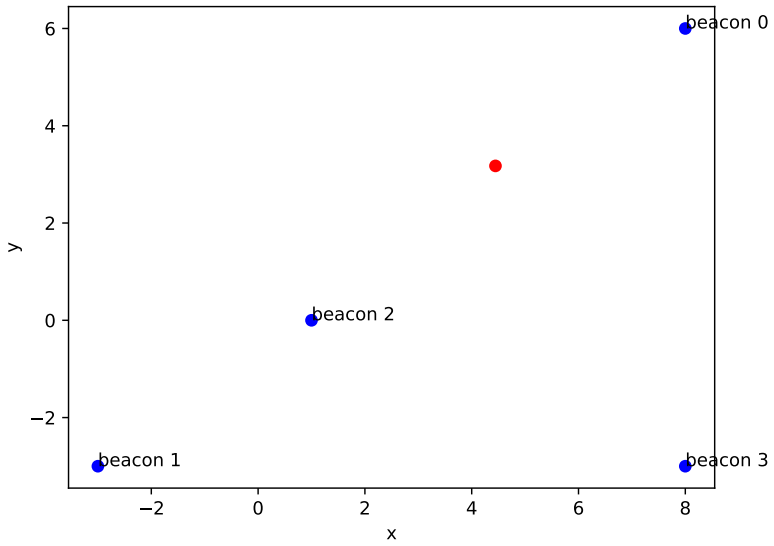


Figure 2.4: Solution of the airplane position parameter estimation problem from Example 2.3.

§ 3 Convex Optimization

In this section we consider another class of optimization problems:

$$\text{Minimize} \quad f(x) \quad \text{where } x \in \mathbb{R}^n. \tag{3.1}$$

These problems feature a *convex* objective $f\colon \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$. Due to the possibility of f taking the value ∞, we can implicitly model constraints, since points with a function value $f(x) = \infty$ are not considered to be minimizers, since we exclude the pathological case $f \equiv \infty$.

Definition 3.1 (Convex function)**.** *Suppose that $f\colon \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$.*

(*i*) *f is said to be **convex** if*

$$f(\alpha\, x + (1-\alpha)\, y) \le \alpha\, f(x) + (1-\alpha)\, f(y) \tag{3.2}$$

holds for all $x, y \in \mathbb{R}^n$ and all $\alpha \in [0,1]$.

(*ii*) *f is said to be **strictly convex** if*

$$f(\alpha\, x + (1-\alpha)\, y) < \alpha\, f(x) + (1-\alpha)\, f(y) \tag{3.3}$$

holds for all $x \neq y \in \mathbb{R}^n$ and all $\alpha \in (0,1)$.

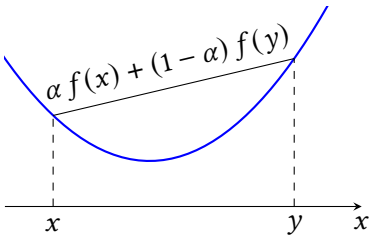


Figure 3.1: Example of a (strictly) convex function on $\mathbb{R}$.

The benefit of the objective being convex is clarified by the following theorem.

Theorem 3.2 (Fundamental theorem of convex optimization)**.**

(a) Every local minimizer of (3.1) is already a global minimizer.

(b) The set of global minimizers of (3.1) is a convex set.[5]

(c) When f is strictly convex, then (3.1) has at most one solution.

[5] A set $C \subseteq \mathbb{R}^n$ is said to be **convex** if the line joining any points in C remains in C, i. e.: for all $x, y \in C$ we have $\alpha\, x + (1-\alpha)\, y \in C$ for all $\alpha \in [0,1]$.

Consequently, in convex optimization one does not have to distinguish local from global minimizers.

We consider in this section optimization problems of a form more particular than the generic convex problem (3.1). Indeed, we consider composite problems of the form

$$\text{Minimize} \quad f(x) + g(Ax) \quad \text{where } x \in \mathbb{R}^n. \tag{3.4}$$

In (3.4), $f\colon \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ and $g\colon \mathbb{R}^m \to \mathbb{R} \cup \{\infty\}$ are both convex functions, and $A \in \mathbb{R}^{m,n}$ is a matrix. Problems of type (3.4) are encountered in many practical problems, and they allow specialized algorithms of convex optimization to be used, compared to the generic convex problem (3.1).

Example 3.3 (Image denoising)**.** *A classical problem in image denoising is as follows. Suppose we are given a noisy image (gray-scale for simplicity), which is represented as a matrix $Z \in \mathbb{R}^{n_1 \times n_2}$. The image height (number of pixels in vertical direction) is n_1 while n_2 is the width. We wish to remove the noise while retaining the image's content. In particular, we wish to preserve edges, i. e., sharp variations of the brightness values between neighboring pixels.*

A classical way to approach this task (Rudin, Osher, Fatemi, 1992) is to formulate it as an optimization problem:

$$\begin{aligned} \textit{Minimize} \quad & \frac{1}{2} \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \underbrace{\left(X_{i,j} - Z_{i,j}\right)^2}_{\textit{fidelity term}} + \beta \sum_{i=1}^{n_1} \sum_{j=1}^{n_2-1} \underbrace{|X_{i,j} - X_{i,j+1}|}_{\textit{horizontal jump}} + \beta \sum_{i=1}^{n_1-1} \sum_{j=1}^{n_2} \underbrace{|X_{i+1,j} - X_{i,j}|}_{\textit{vertical jump}} \\ \textit{where} \quad & X \in \mathbb{R}^{n_1 \times n_2}. \end{aligned} \tag{3.5}$$

The first term ensures that the resulting image X does not deviate too much from the noisy image Z observed. The remaining terms penalize differences in brightness between all pairs of neighboring pixels, both in horizontal and in vertical directions. These terms are also known as the total variation *of the image. The parameter $\beta > 0$ balances both objectives.*

Problem (3.5) can be written in the form (3.4) when we vectorize[6] the matrix-valued optimization variable to become $x = \operatorname{vec}(X) \in \mathbb{R}^{n_1 n_2}$. Likewise, we vectorize the observed image: $z = \operatorname{vec}(Z)$. The first term in (3.5) then simply becomes $\frac{1}{2}\|x - z\|_2^2$. To account for the remaining terms, we require a matrix A which converts the pixel values x to the vector of horizontal differences, followed by the vector of vertical differences. Such a matrix A is of dimension $[n_1\,(n_2-1) + n_2\,(n_1-1)] \times [n_1\,n_2]$. Each row contains precisely one entry $+1$ and one entry -1, all remaining entries are zero; see Figure 3.2. Finally, $g(y) = \beta\,\|y\|_1$ holds, i. e., $g(y)$ is β times the sum of the absolute values of all entries of the vector y.

The objective in Example 3.3 is convex since

$$f(x) := \frac{1}{2}\|x - z\|_2^2 \quad \text{and} \quad g(Ax) := \beta\|Ax\|_1 \tag{3.6}$$

are both convex, and since sums and non-negative multiples of convex functions are again convex.

The algorithm we are considering exploits the additive and composite structure of (3.5). In order to derive it, we require the notion of the Fenchel conjugate of a function.

[6] Vectorization of a matrix simply means to stack the columns of the matrix on top of each other to form a long vector with all entries of the matrix.

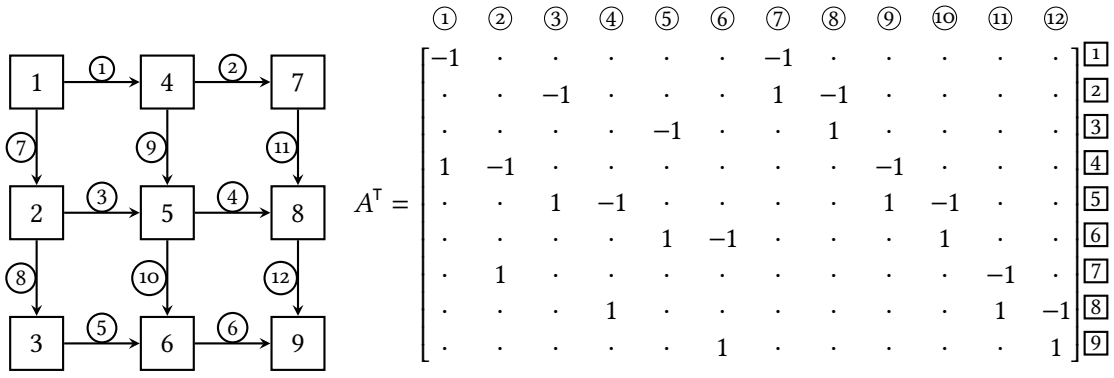


Figure 3.2: Illustration of the construction of the matrix A for an image denoising problem (Example 3.3). Its transpose A^T is the incidence matrix of the directed graph obtained from connecting neighboring pixels. For readibility, zeros are written as $\cdot$.

Definition 3.4 (Fenchel conjugate). *Suppose that $f\colon \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ is a function. Then the* ***Fenchel conjugate*** *of f is $f^*\colon \mathbb{R}^n \to \mathbb{R} \cup \{\pm\infty\}$, defined by*

$$f^*(\xi) := \sup_{x \in \mathbb{R}^n} \xi^\mathsf{T} x - f(x). \tag{3.7}$$

In view of

$$\begin{aligned} f^*(\xi) &= \inf\left\{\beta \in \mathbb{R} \,\middle|\, \xi^\mathsf{T} x - \beta \le f(x) \text{ for all } x \in \mathbb{R}^n\right\} \\ &= \inf\left\{\beta \in \mathbb{R} \,\middle|\, z \le f(x) \text{ for all } \left(\begin{smallmatrix} x \\ z \end{smallmatrix}\right) \in \mathbb{R}^n \times \mathbb{R} \text{ and } z = \xi^\mathsf{T} x - \beta\right\}, \end{aligned}$$

we have a geometric interpretation of $f^*(\xi)$. We consider a hyperplane

$$\begin{aligned} H_{n,\beta} &= \left\{\left(\begin{smallmatrix} x \\ z \end{smallmatrix}\right) \in \mathbb{R}^n \times \mathbb{R} \,\middle|\, n^\mathsf{T} \left(\begin{smallmatrix} x \\ z \end{smallmatrix}\right) = \beta\right\} \\ &= \left\{\left(\begin{smallmatrix} x \\ z \end{smallmatrix}\right) \in \mathbb{R}^n \times \mathbb{R} \,\middle|\, z = \xi^\mathsf{T} x - \beta\right\} \end{aligned}$$

in $\mathbb{R}^n \times \mathbb{R}$ with the fixed normal vector $n = \left(\begin{smallmatrix} \xi \\ -1 \end{smallmatrix}\right)$ and unknown offset β, and push it upwards against the graph of the function f by decreasing the offset β. Then we record the minimal offset β. We can find the (negative) offset β when evaluating $z = \xi^\mathsf{T} x - \beta$ at $x = 0$; see Figure 3.3.

The Fenchel conjugate f^* of any function f is always convex. Moreover, when f itself is convex and lower semicontinuous[7], then f^* contains enough information to recover f, in the sense that the biconjugate function f^{**} equals f.

Without going into any details here, one can show the following theorem; see for instance Clason, 2017, Section 7.4.

Theorem 3.5 (Characterization of solutions to (3.4))**.**

[7] A function $f\colon \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ is **lower semicontinuous** if $f(x_0) \le \liminf_{x \to x_0} f(x)$ holds for all $x_0 \in \mathbb{R}^n$.

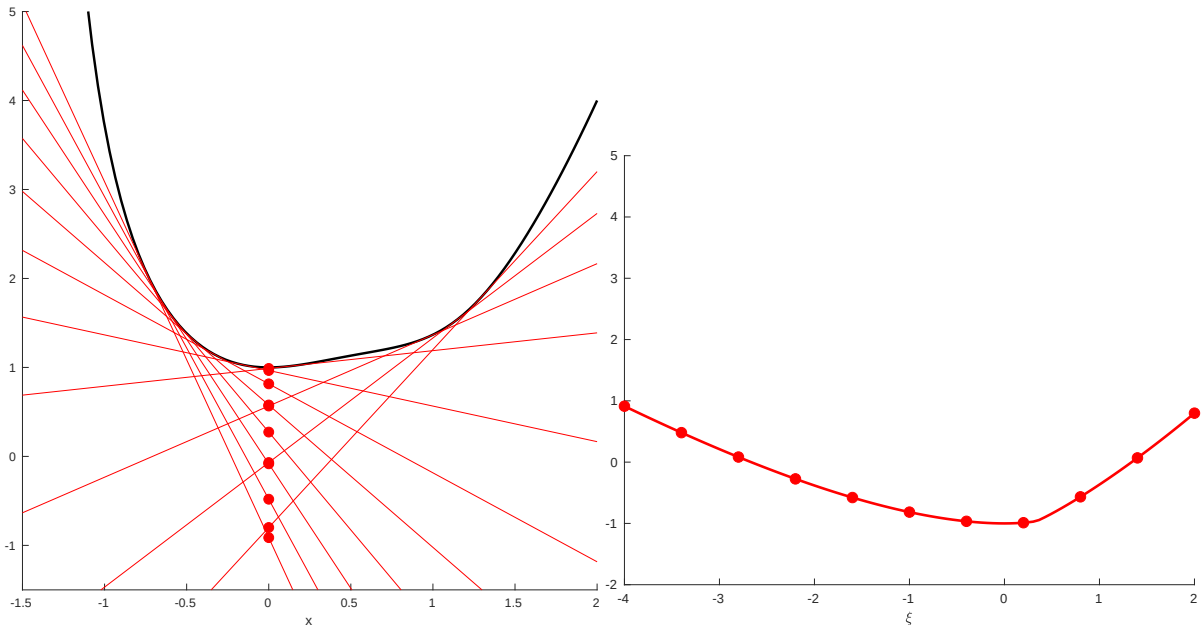


Figure 3.3: Illustration of the construction of the Fenchel conjugate function $f^*: \mathbb{R} \cup \{\pm\infty\}$ (right) of the (convex) function $f(x) = x^2 + \exp(-x^3)$ (left).

A point $x^* \in \mathbb{R}^n$ is a (global) minimizer for (3.4) if and only if there exists $p^* \in \mathbb{R}^m$ such that, for any $\sigma, \tau \neq 0$,

$$x^* \text{ solves } \quad \text{Minimize} \quad \frac{1}{2}\|y - (x^* - \tau A^\mathsf{T} p^*)\|_2^2 + \tau f(y) \quad \text{where } y \in \mathbb{R}^n, \tag{3.8a}$$

$$\text{and } \quad p^* \text{ solves } \quad \text{Minimize} \quad \frac{1}{2}\|q - (p^* + \sigma A x^*)\|_2^2 + \sigma g^*(q) \quad \text{where } q \in \mathbb{R}^m. \tag{3.8b}$$

The above theorem provides the idea for an algorithm. As the unknown solutions x^* and p^* appear in the respective objectives for both problems, we can convert (3.8) into a fixed-point algorithm.

Algorithm 3.6 (Primal-dual extragradient algorithm (Chambolle, Pock, 2011))**.**

1: *Set $x^{(k+1)}$ to the solution of the following problem:*

$$\text{Minimize} \quad \frac{1}{2}\|y - (x^{(k)} - \tau A^\mathsf{T} \overline{p}^{(k)})\|_2^2 + \tau f(y) \quad \text{where } y \in \mathbb{R}^n \tag{3.9}$$

2: *Set $p^{(k+1)}$ to the solution of the following problem:*

$$\text{Minimize} \quad \frac{1}{2}\|q - (p^{(k)} + \sigma A x^{(k+1)})\|_2^2 + \sigma g^*(q) \quad \text{where } q \in \mathbb{R}^m \tag{3.10}$$

3: *Set $\overline{p}^{(k+1)} := 2\,p^{(k+1)} - p^{(k)}$.*

The advantage of such an algorithm is that the solution to the original problem (3.4), involving both f and g, is broken down into the repeated solution to two different problems (3.9) and (3.10). Problem

(3.9) requires the minimization of τf plus a quadratic distance term. The solution to this problem is also known as **proximity (prox) operator**. More precisely, one defines

$$\operatorname{prox}_{\tau f}(x) \coloneqq \text{unique solution of "Minimize } \frac{1}{2}\|y-x\|_2^2 + \tau f(y) \text{ where } y \in \mathbb{R}^n\text{"}. \tag{3.11}$$

Similarly, problem (3.10) requires the minimization of σ times the conjugate g^* plus a quadratic distance term, i. e., the evaluation of $\operatorname{prox}_{\sigma g^*}$.

For a number of relevant problems, the solutions to problems (3.9) and (3.10) can be explicitly calculated.[8] We consider this now for f and g^* from the image denoising Example 3.3. We begin with $\operatorname{prox}_{\tau f}(x)$, i. e., the unique solution of the problem

$$\text{Minimize} \quad \frac{1}{2}\|y-x\|_2^2 + \tau f(y) = \frac{1}{2}\|y-x\|_2^2 + \frac{\tau}{2}\|y-z\|_2^2 \quad \text{where } y \in \mathbb{R}^{n_1 n_2}.$$

A short calculation shows (**Quiz 3.1:** Can you fill in the details?)

$$\operatorname{prox}_{\tau f}(x) = \frac{x+\tau z}{1+\tau}, \tag{3.12}$$

i. e., $\operatorname{prox}_{\tau f}(x)$ is simply a weighted average between x and z. On the other hand, we first need to evaluate

$$g^*(\xi) = \sup_{p\in\mathbb{R}^m} \xi^\mathsf{T} p - g(p) = \sup_{p\in\mathbb{R}^m} \xi^\mathsf{T} p - \beta\,\|p\|_1$$

where $m = n_1\,(n_2-1) + n_2\,(n_1-1)$ is the number of pairs of neighboring pixels (edges in Figure 3.2). We calculate (**Quiz 3.2:** Can you fill in the details?)[9]

$$g^*(\xi) = \begin{cases} 0 & \text{if } \|\xi\|_\infty \le \beta \\ \infty & \text{if } \|\xi\|_\infty > \beta. \end{cases}$$

Therefore we have that $\operatorname{prox}_{\sigma g^*}(p)$ is the unique solution of the problem

$$\text{Minimize} \quad \frac{1}{2}\|q-p\|_2^2 + \sigma\, g^*(q) \quad \text{where } q \in \mathbb{R}^m,$$

which is to say that $\operatorname{prox}_{\sigma g^*}(p)$ is the unique solution of

$$\begin{aligned} &\text{Minimize} && \frac{1}{2}\|q-p\|_2^2 \quad \text{where } q \in \mathbb{R}^m \\ &\text{subject to} && \|q\|_\infty \le \beta. \end{aligned}$$

A short calculation shows (**Quiz 3.3:** Can you fill in the details?)

$$[\operatorname{prox}_{\sigma g^*}(p)]_i = \frac{\beta\, p_i}{\max\{\beta,\ |p_i|\}} = \operatorname{proj}_{[-\beta,\beta]}(p_i), \tag{3.13}$$

which turns out to be independent of σ. This formula means that the values of p are simply clipped elementwise to lie inside $[-\beta, \beta]$.

[8] The web site http://proximity-operator.net/ collects computable prox operators.

[9] It can be shown in general that the convex conjugate of a norm is the indicator function of the unit ball w.r.t. the dual norm.

It can be shown (see Chambolle, Pock, 2011, Theorem 1 or Clason, 2017, Theorem 7.8) that Algorithm 3.6 converges to a solution of (3.4) as long as σ, τ are positive numbers satisfying $\sigma\,\tau\,\|A\|^2 < 1$, where $\|A\|$ is the largest singular value of A.

We close this section by mentioning what functions a user has to implement in order to run the Chambolle-Pock Algorithm 3.6 for solving (3.4):

proximity operator of $\tau\,f$	$\operatorname{prox}_{\tau f}(x)$	
proximity operator of $\sigma\,g^*$	$\operatorname{prox}_{\sigma g^*}(p)$	
matrix-vector product with A	Ax	
matrix-vector product with A^T	$A^\mathsf{T}p$	redundant if A is available as a matrix

It is remarkable that the user does not need to implement the objective (3.4) or even the functions f and g, but needs to implement the solution operators for certain problems involving f and the conjugate g^*.

Figure 3.4 shows the result of an image denoising optimization run.

Figure 3.4: Image denoising model problem (Example 3.3.) Left: original image (dimensions $n_1 = n_2 = 256$). Middle: image with noise. Right: denoising result obtained using 200 iterations of the Chambolle-Pock Algorithm 3.6 with $\sigma = 1$ and $\tau = 1/10$ and weighting parameter $\beta = 0.08$.

End of Class 2

§ 4 Constrained Optimization

We recall from § 1 that we are considering constrained optimization problems of the form

$$\left.\begin{array}{rll} \text{Minimize} & f(x) & \text{where } x \in \mathbb{R}^n \\ \text{subject to} & g_i(x) \leq 0 & \text{with } i = 1, \dots, n_{\text{ineq}} \\ \text{and} & h_j(x) = 0 & \text{with } j = 1, \dots, n_{\text{eq}}. \end{array}\right\} \tag{4.1}$$

These problems are known as **nonlinear optimization problems** or **nonlinear programming problems (NLP)**. We also recall the notion of the **feasible set**

$$F := \left\{ x \in \mathbb{R}^n \,\middle|\, g_i(x) \leq 0 \text{ for all } i = 1, \dots, n_{\text{ineq}} \text{ and } h_j(x) = 0 \text{ for all } j = 1, \dots, n_{\text{eq}} \right\} \tag{4.2}$$

associated with (4.1).

Example 4.1 (Production example; slightly modified from John E. Beasley's OR notes[10])**.**
A company makes two products (X and Y) using two machines (A and B). Each unit of X that is produced requires 50 minutes processing time on machine A and 30 minutes processing time on machine B. Each unit of Y that is produced requires 24 minutes processing time on machine A and 33 minutes processing time on machine B. At the start of the current week there are 30 units of X and 90 units of Y in stock. Available processing time on machine A is forecast to be 40 hours and on machine B is forecast to be 35 hours. The demand for X in the current week is forecast to be 70 units and for Y is forecast to be 95 units. Company policy is to maximize the combined sum of the units of X and the units of Y in stock at the end of the week.

Formulate this as an optimization problem. What kind of problem is it? What type of solver would you use to solve it?

Note: Spoiler alert: The solution of Example 4.1 is given in § 7.

Example 4.2 (Formulating non-smooth problems as smooth ones)**.** *Try to formulate the following optmization problems in the form* (1.1) *with smooth objective and smooth constraints.*

$$\left.\begin{array}{ll} \textit{Minimize} & -x_1^2 + x_2 \\ \textit{subject to} & \max\{x_1 + x_2,\ 2\,x_1 - x_2\} \leq 3 \end{array}\right\} \tag{4.3}$$

$$\left.\begin{array}{ll} \textit{Minimize} & -x_1^2 + x_2 \\ \textit{subject to} & \min\{x_1 + x_2,\ 2\,x_1 - x_2\} \leq 3 \end{array}\right\} \tag{4.4}$$

$$\left.\begin{array}{ll} \textit{Minimize} & \max\{-x_1^2 + x_2,\ x_1^2 + 2\,x_2\} \\ \textit{subject to} & x_1 + x_2 \leq 3 \end{array}\right\} \tag{4.5}$$

$$\left.\begin{array}{ll} \textit{Minimize} & |-x_1^2 + x_2| + |x_1 + x_2^2| \\ \textit{subject to} & x_1 + x_2 \leq 3 \end{array}\right\} \tag{4.6}$$

[10] https://people.brunel.ac.uk/~mastjjb/jeb/or/morelp.html

We will now look a little bit into the theory for problem (1.1) since it also tells us something about choices we may be facing while modeling the problem.

One can easily see by example (**Quiz 4.1:** Can you give an example?) that the well-known condition $\nabla f(x) = 0$ is no longer a necessary condition for local optimality in case of a constrained problem. It is one of the main conceptual difficulties in constrained optimization to come up with conditions that replace $\nabla f(x) = 0$.

The main idea is as follows. Consider a local minimizer x^* and a sequence $x^{(k)}$ of feasible points approaching it. The elements of the sequence will eventually (i. e., for k large enough) all remain in the neighborhood $U(x^*)$ of local optimality. Consequently,

$$f(x^*) \leq f(x^{(k)})$$

holds for all k sufficiently large. Therefore, we have

$$0 \leq f(x^{(k)}) - f(x^*). \tag{4.7}$$

By Taylor's theorem[11], for any $k \in \mathbb{N}$ there exist a number $\xi^{(k)} \in (0,1)$ such that

$$f(x^{(k)}) = f(x^*) + \nabla f(x^* + \xi^{(k)}\,(x^{(k)} - x^*))^\mathsf{T}(x^{(k)} - x^*)$$

holds. Plugging this into the inequality (4.7), we obtain

$$0 \leq \nabla f(x^* + \xi^{(k)}\,(x^{(k)} - x^*))^\mathsf{T}(x^{(k)} - x^*) \tag{4.8}$$

for sufficiently large $k \in \mathbb{N}$. Notice that since $x^{(k)} \to x^*$ and since $\xi^{(k)}$ is bounded, we must have

$$x^* + \xi^{(k)}\,(x^{(k)} - x^*) \to x^*$$

and therefore

$$\nabla f(x^* + \xi^{(k)}\,(x^{(k)} - x^*)) \to \nabla f(x^*).$$

However, passing to the limit in (4.8) only yields the meaningless $0 \leq 0$ so we must do something different.

Let us take another sequence $t^{(k)}$ which converges to zero from above: $t^{(k)} \searrow 0$, and let us restrict our attention to sequences $x^{(k)}$ such that the limit

$$d := \lim_{k\to\infty} \frac{x^{(k)} - x^*}{t^{(k)}}$$

exists. Dividing (4.8) by $t^{(k)}$ gives

$$0 \leq \nabla f(x^* + \xi^{(k)}\,(x^{(k)} - x^*))^\mathsf{T} \left(\frac{x^{(k)} - x^*}{t^{(k)}}\right)$$

and passing to the limit now yields the informative statement

$$0 \leq \nabla f(x^*)^\mathsf{T} d$$

for all directions $d \in \mathbb{R}^n$ constructed as above. We refer to those as **tangent directions** or **tangent vectors**.

[11] This requires that the objective f is a continuously differentiable (C^1) function.

Definition 4.3 (Tangent vectors, tangent cone). *A vector $d \in \mathbb{R}^n$ is said to be a* ***tangent direction*** *or* ***tangent vector*** *to the (feasible) set F at the point $x^* \in F$ if there exist sequences $x^{(k)}$ and $t^{(k)}$ such that $x^{(k)} \to x^*$ and $t^{(k)} \searrow 0$ such that*

$$d = \lim_{k \to \infty} \frac{x^{(k)} - x^*}{t^{(k)}}$$

holds. The set of all *tangent vectors to F at x^* is said to be the* ***tangent cone***[12] *to F at x^* and we denote it by $\mathcal{T}_F(x^*)$.*

Above we have proved the following

Theorem 4.4 (First-order necessary optimality conditions).
Suppose that x^* is a local minimizer of an optimization problem with feasible set F. Then

$$\nabla f(x^*)^\mathsf{T} d \geq 0 \quad \text{for all } d \in \mathcal{T}_F(x^*). \tag{4.9}$$

The difficulty with this theorem is that the tangent cone $\mathcal{T}_F(x^*)$ is generally hard to characterize and thus impossible to use in an optimization algorithm. Notice also that $\mathcal{T}_F(x^*)$ doesn't even use the description of the feasible set F (4.2) in terms of the inequality constraints g_i or the equality constraints h_j.

We therefore introduce another cone as a candidate to replace $\mathcal{T}_F(x^*)$.

Definition 4.5 (Linearized feasible vector, linearizing cone). *A vector $d \in \mathbb{R}^n$ is said to be a* ***linearized feasible direction*** *to the (feasible) set F described by (1.2) at the point $x^* \in F$ if it satisfies the following conditions:*

$$\nabla g_i(x^*)^\mathsf{T} d \leq 0 \quad \text{for all active indices } i = 1, \dots, m, \tag{4.10a}$$
$$\nabla h_j(x^*)^\mathsf{T} d = 0 \quad \text{for all } j = 1, \dots, p. \tag{4.10b}$$

The set of all *linearized feasible directions to F at x^* is said to be the* ***linearizing cone*** *to F at x^* and we denote it by $\mathcal{T}_F^{\text{lin}}(x^*)$.*

Recall that the active indices at x^* are those satisfying $g_i(x^*) = 0$. Therefore, inequalities that are inactive at x^* do not play a role in the definition.

Remark 4.6 (Tangent and linearizing cones). *The tangent cone $\mathcal{T}_F(x^*)$ can be viewed as a geometric approximation to the feasible set near the point x^*. By contrast, the linearizing cone $\mathcal{T}_F^{\text{lin}}(x^*)$ is an attempt to locally describe the feasible set in algebraic terms, by means of equalities and inequalities.*

[12] In mathematics, a **cone** in $\mathbb{R}^n$ is any set $K \subseteq \mathbb{R}^n$ such that $x \in K$ implies $\alpha x \in K$ for all $\alpha > 0$. In other words, a point x in the cone entails that the entire ray from the origin through x belongs to the cone as well.

One can show (although we do not do it here) that $\mathcal{T}_F(x^*) \subseteq \mathcal{T}_F^{\text{lin}}(x^*)$ holds at any feasible point x^*. Therefore, we cannot expect that Theorem 4.4 continues to hold if $d \in \mathcal{T}_F(x^*)$ were replaced by $d \in \mathcal{T}_F^{\text{lin}}(x^*)$. In order for that to be possible, we require an extra condition that ensures that $\mathcal{T}_F(x^*) = \mathcal{T}_F^{\text{lin}}(x^*)$. Such a condition guarantees the agreement of the geometric and algebraic descriptions and it is called a **constraint qualification**.

There are various types of constraint qualifications, and we consider only the most simple one.

Definition 4.7 (LICQ). *Suppose that the (feasible) set F is given by (1.2) and $x^* \in F$. We say that x^* satisfies the **linear independence constraint qualification** (in short: **LICQ**) if the following set of vectors is linearly independent:*

$$\{\nabla g_i(x)\}_{i \in \mathcal{A}(x^*)} \cup \{\nabla h_j(x)\}_{j=1}^{p}.$$

Here $\mathcal{A}(x^*)$ denotes the set of active indices, i. e. $\mathcal{A}(x^*) := \{i = 1, \dots, m \mid g_i(x^*) = 0\}$.

One can show that the LICQ at x^* implies $\mathcal{T}_F(x^*) = \mathcal{T}_F^{\text{lin}}(x^*)$. Provided that x^* is also a local minimizer, we thus get from Theorem 4.4 that

$$\nabla f(x^*)^\mathsf{T} d \geq 0 \quad \text{for all } d \in \mathcal{T}_F^{\text{lin}}(x^*). \tag{4.11}$$

What could be the benefit of this, compared to (4.9)?

By a technique called the Farkas lemma we can show that condition (4.11) can be equivalently case as a set (4.12) of equalities and inequalities, which is amenable to numerical algorithms. We state this result in the following lemma, without proof.

Lemma 4.8 ((4.11) as KKT conditions).
Suppose that the (feasible) set F is given by (4.2) and $x^* \in F$. The following two statements are equivalent.

(a) (4.11) holds.

(b) There exist vectors $\mu \in \mathbb{R}^m$ and $\lambda \in \mathbb{R}^p$ such that the following set of conditions hold:

$$\nabla f(x^*) + \sum_{i=1}^{m} \mu_i \nabla g_i(x^*) + \sum_{j=1}^{p} \lambda_j \nabla h_j(x^*) = 0, \tag{4.12a}$$

$$h(x^*) = 0, \tag{4.12b}$$

$$\mu \geq 0, \quad g(x^*) \leq 0, \quad \mu^\mathsf{T} g(x^*) = 0. \tag{4.12c}$$

The system (4.12) is known as the **Karush-Kuhn-Tucker conditions** (in short: **KKT conditions**) of problem (4.1). The vectors μ and λ are called **Lagrange multipliers** associated with the inequality and equality constraints, respectively. A point x^* which admits associated Lagrange multipliers μ and λ so that (4.12) holds is called a **KKT point**.

Condition (4.12a) is often written in terms of the **Lagrangian** associated with problem (4.1), which is defined as

$$\mathcal{L}(x, \mu, \lambda) := f(x) + \mu^\mathsf{T} g(x) + \lambda^\mathsf{T} h(x). \tag{4.13}$$

We can easily see that (4.12a) is nothing but $\nabla_x \mathcal{L}(x^*, \lambda, \mu) = 0$.

We summarize our findings in the following

Theorem 4.9 (First-order necessary optimality conditions under LICQ, compare Theorem 4.4)**.** *Suppose that x^* is a local minimizer of an optimization problem with feasible set F. Moreover, suppose that the LICQ holds at x^*. Then there exist Lagrange multipliers $\mu \in \mathbb{R}^{n_{\text{ineq}}}$ and $\lambda \in \mathbb{R}^{n_{\text{eq}}}$ such that the KKT conditions (4.12) hold. Moreover, μ and λ are unique.*

The uniqueness of the multipliers follows from viewing (4.12a) as a linear system of equations for μ and λ and observing that the LICQ implies that the coefficient matrix has linearly independent columns.

A number of comments concerning (4.12) are in order.

(a) In the absence of inequality constraints ($n_{\text{ineq}} = 0$), (4.12) is a (generally nonlinear) system of equations only.

(b) Condition (4.12c) — which comes in through inequality constraints — is called a **complementarity system**. Due to the signs of μ and $g(x^*)$, we can equivalently write $\mu^\mathsf{T} g(x^*) = 0$ componentwise as $\mu_i g_i(x^*) = 0$ for all $i = 1, n_{\text{ineq}}$. Consequently, Lagrange multipliers pertaining to inactive inequality constraints must be zero. On the other hand, a strictly positive multiplier $\mu_i > 0$ indicates that the associated constraint must be active, i. e., $g_i(x^*) = 0$.

We can ask the question how "likely" it is for the LICQ to hold for an optimization problem at hand. First of all, the very Definition 4.7 implies that for the LICQ to hold there cannot be too many active inequality and equality constraints in the problem. As soon as the combined number of active inequality and equality constraints at a point x^* exceeds n (the dimension of the optimization variable), the LICQ cannot hold at x^*.

What can we learn from this? When we are modeling an optimization problem, the observation above tells us not to unconsciously throw in additional constraints (which may not be needed) at will. The following example gives us additional hints about things to avoid in modeling constraints:

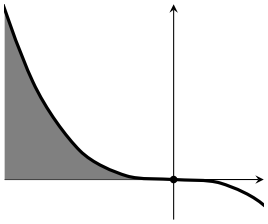
Example 4.10 (LICQ holding or not holding?)**.** *Consider the following formulations of a (trivial) optimization problem with variable $x \in \mathbb{R}$, and decide whether or not the LICQ holds at the (obvious) solution:*

$$\begin{Bmatrix} \text{Minimize} & x^2 \\ \text{subject to} & x = 0 \end{Bmatrix} \quad \begin{Bmatrix} \text{Minimize} & x^2 \\ \text{subject to} & x^2 = 0 \end{Bmatrix} \quad \begin{Bmatrix} \text{Minimize} & x^2 \\ \text{subject to} & x^2 \leq 0 \end{Bmatrix} \quad \begin{Bmatrix} \text{Minimize} & x^2 \\ \text{subject to} & x \leq 0 \\ \text{and} & x \geq 0 \end{Bmatrix}$$

The following example shows that the failure of constraint qualifications to hold can indeed be problematic in practice:

Example 4.11 (Minimizer where the KKT conditions fail to hold)**.** *Consider the problem*

$$\begin{aligned} \textit{Minimize} \quad & -x_1 \quad \textit{where } x \in \mathbb{R}^2 \\ \textit{subject to} \quad & x_2 + x_1^3 \le 0 \\ \textit{and} \quad & -x_2 \le 0 \end{aligned}$$



Obviously, $x^* = (0,0)^\mathsf{T}$ is the unique global minimizer, and there are no further local minimizers. Since both constraints are active at x^*, the KKT conditions (4.12) boil down to

$$\begin{pmatrix} -1 \\ 0 \end{pmatrix} + \mu_1 \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \mu_2 \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Clearly, it is impossible to satisfy the KKT conditions with any $\mu_1, \mu_2 \ge 0$. Hence, Lagrange multipliers do not exist at this minimizer. In other words, the minimizer x^* fails to be a KKT point.

The observation in the previous example is noteworthy since essentially all numerical algorithms for the solution of smooth, constrained problems (4.1) are indeed looking for KKT points.

Example 4.12 (Rosenbrock's post office problem)**.** *A simple test example for nonlinear programming is Rosenbrock's post office problem:*

$$\begin{aligned} \textit{Minimize} \quad & -x_1 x_2 x_3 \quad \textit{where } x \in \mathbb{R}^3 \\ \textit{subject to} \quad & x_1 + 2x_2 + 2x_3 - 72 \le 0 \\ \textit{and} \quad & x_1 + 2x_2 + 2x_3 \ge 0 \\ \textit{and} \quad & 0 \le x_i \le 42, \quad i = 1,2,3 \end{aligned}$$

This problem has only linear constraints[13] *but a nonlinear objective. The global minimizer is $x^* = (24,12,12)^\mathsf{T}$.*

As we discussed for unconstrained problems in § 2, most optimization algorithms proceed in a sequential way and break down problem (4.1) into a sequence of simpler problems. We will consider here only the class of **sequential quadratic programming (SQP) methods**, where these simpler problems are **quadratic programming (QP) problems**. A QP is a problem of type (4.2) whose objective is a quadratic polynomial and all of whose constraints are linear. In an SQP method, the QP formed at

[13] Problems featuring only linear constraints satisfy a constraint qualification known as **Abadie constraint qualification** (ACQ) at every feasible point. Hence for problems which have only linear constraints, every local minimizer is a KKT point. However, in contrast to LICQ, the Lagrange multipliers need not be unique.

iterate $x^{(k)}$ is of the form

$$\left.\begin{aligned} \text{Minimize} \quad q^{(k)}(x) &:= \underbrace{\mathcal{L}(x^{(k)}, \mu^{(k)}, \lambda^{(k)})}_{\text{constant term}} + \overbrace{\mathcal{L}_x(x^{(k)}, \mu^{(k)}, \lambda^{(k)})(x - x^{(k)})}^{\text{linear term}} \\ &\quad + \underbrace{\frac{1}{2}(x - x^{(k)})^\intercal H^{(k)}(x - x^{(k)})}_{\text{quadratic term}} \quad \text{where } x \in \mathbb{R}^n \\ \text{subject to} \quad g_i(x^{(k)}) + g_i'(x^{(k)})(x - x^{(k)}) &\leq 0 \qquad \text{with } i = 1, \dots, n_{\text{ineq}} \\ \text{and} \quad h_j(x^{(k)}) + h_j'(x^{(k)})(x - x^{(k)}) &= 0 \qquad \text{with } j = 1, \dots, n_{\text{eq}}. \end{aligned}\right\} \tag{4.14}$$

In comparison to (2.2), the objective is now a quadratic model of the Lagrangian, rather than a quadratic model of the objective. Depending on how the Hessians $H^{(k)}$ of the model are formed, one can distinguish between quasi-Newton and Newton-type SQP methods. As was the case in § 2, precautions need to be taken in order to achieve good convergence properties regardless of the quality of the initial guess provided by the user. As before, there are line-search as well as trust-region but also filter SQP methods around for this purpose.

Many other details need to be taken into account in order to obtain a robust SQP solver, including potentially infeasible QP subproblems, update of the Lagrange multipliers, stopping criteria, deterioration of superlinear convergence due to nonlinear constraints, failure of constraint qualifications etc.

We close this section by mentioning what functions a user has to implement in order to run an SQP for solving a constrained problem (4.1):

objective	$f(x)$	
derivative	$f'(x)$	optional (finite difference fallback)
2nd derivative	$f''(x)$ or $f''(x)\,d$	optional, only for Newton SQP methods
constraints	$g(x)$, $h(x)$	
Jacobian	$g'(x)$, $h'(x)$	optional (finite difference fallback)
2nd derivative	$g_i''(x)$, $h_j''(x)$ or $g_i''(x)\,d$, $h_j''(x)\,d$	optional, only for Newton SQP methods

End of Class 3

§ 5 Infinite-Dimensional Optimization

In this section we consider problems which are infinite-dimensional, i. e., which feature infinite-dimensional optimization variables. Most problems in this category involve a differential equation of some sort, and the optimization variables are functions. Although for the purpose of numerical optimization, these functions clearly need to be discretized to become finite-dimensional objects, it is still useful to recognize the properties of the underlying undiscretized (infinite-dimensional) problem.

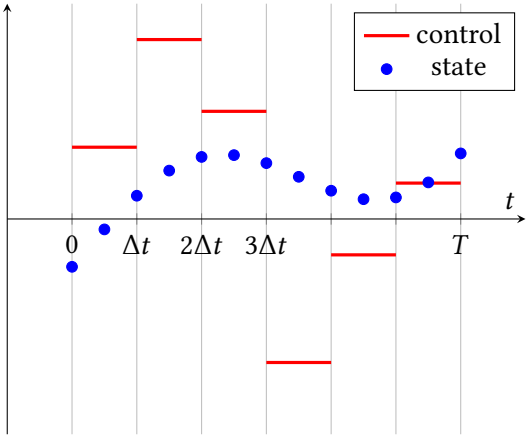
The following example is adapted from section 8 in the CasADi documentation.

Example 5.1 (Optimal control of the van der Pol oscillator)**.** *Consider the following optimal control problem:*

$$\begin{aligned}
&\textit{Minimize} && \int_0^T (x_1^2 + x_2^2)\,\mathrm{d}t + \gamma \int_0^T u^2\,\mathrm{d}t \quad \textit{where } x\colon [0,T] \to \mathbb{R}^2, u\colon [0,T] \to \mathbb{R} \\
&\textit{subject to} && \begin{cases} \dot{x}_1 = \mu\,(1 - x_2^2)\,x_1 - x_2 + u \\ \dot{x}_2 = x_1 \end{cases} \\
&\textit{initial conditions} && \begin{pmatrix} x_1(0) \\ x_2(0) \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
&\textit{and control constraints} && -1 \le u \le 1.
\end{aligned} \tag{5.1}$$

The ordinary differential equation (ODE) above is known as the van der Pol oscillator. The unknown function $x\colon [0,T] \to \mathbb{R}^2$ is the ***state variable*** *of the optimal control problem (5.1) and $u\colon [0,T] \to \mathbb{R}$ is the* ***control variable****. All constraints are meant in a pointwise sense on the interval $[0,T]$ with final time $T > 0$ given. The constant $\mu \ge 0$ is a given physical parameter. The objective expresses the goal to steer the system to its equilibrium point at the origin, while the second term in the objective penalizes the control effort. The parameter $\gamma \ge 0$ is often called the control-cost parameter and it is used to balance both terms in the objective.*

We are going to discretize problem (5.1) using a so-called *direct single shooting approach.* This means that we are going to use a numerical solver for the differential equation, which returns an approximate solution for $x(t)$ on a discrete grid of t-values. Therefore, the only optimization variable remaining is the control function u. This function is discretized on a grid that is possibly coarser than the grid for the state and we take u to be piecewise constant.



A possible solution approach to this problem is shown in § 7.10. Experiments show that off-the-shelf solvers have difficulties solving the problem even for moderate discretization sizes. For instance, we typically observe an increase in the iteration numbers as we refine the control discretization. This is the case in particular for CasADi's default solver Ipopt with default settings.

The reason for this behavior is that most solvers for nonlinear programming problems (see § 4) are designed for genuinely finite-dimensional problems. In particular, they are unaware of the geometry of the problem. In a function-space-aware method, the user would need to choose an inner product, which ideally should be inherited from the infinite-dimensional limit problem. This has an impact, for instance, how gradients are calculated, the geometry of trust regions, on stopping criteria, and many other algorithmic components. Of course one may argue that, for the discretized problem in $\mathbb{R}^n$, all norms are equivalent, but the equivalence constants typically deteriorate with increasing dimension n.

A simple strategy can sometimes be successful to mitigate the difficulties of off-the-shelf solvers: one first solves the problem on a coarse grid, then interpolates the solution to a finer grid and uses this as an initial guess for the next round, until the desired discretization level is reached. This strategy is known as **nested iteration**; see § 7.11 for a demonstration.

However, true efficiency requires an algorithm which "makes sense" also for the infinite-dimensional limit problem. Only then can we hope for convergence results which are independent of the fineness of discretization. These algorithms, however, are not generally available as software. As a researcher working with infinite-dimensional problems (infinite-dimensional optimization variables), one therefore often finds oneself designing and implementing the algorithm of choice oneself.

In addition, there are some pitfalls that only become apparent when analyzing the undiscretized problem. As a rule of thumb, there should be a term in the problem which bounds the control function in an appropriate norm. This can be either achieved by control bounds, or some form of control penalty. In (5.1), we find both, control bounds and the control cost term $\gamma \int_0^T u^2 \, \mathrm{d}t$. If such terms are not present, it must be expected that the undiscretized problem does not have a solution. In this case, it is natural that this ill-posedness of the problem becomes more and more apparent as we refine the control discretization.

§ 6 Summary and Practical Advice

We conclude these notes with a short summary and some practical advice.

(1) Modeling an optimization problem is a skill that requires training. The choices made may have an impact on the characteristics of a problem.

(2) For example, we saw that squaring a constraint $h_j(x) = 0$ to become $[h_j(x)]^2 = 0$ is not a good idea.

(3) There is not a single best solver for all kinds of optimization problems.

(4) It is useful to have an overview over different problem types in optimization in order to be able to make informed modeling decisions and solver choices.

(5) Most algorithms rely on and exploit the smoothness of the objective and constraint functions. They may not stall on the occasional kink, but generally we should strive to formulate our problems in a smooth way.

(6) Depending on the type of algorithm, the user may have to implement different routines than expected. For instance, for algorithms dedicated to least-squares problems, the user implements the residual function (not the objective). The Chambolle-Pock method for composite convex problems requires the user to implement prox operators for τf and σg^*.

(7) Problems with an infinite-dimensional optimization variable (such as the control function in an optimal control problem) often present difficulties to off-the-shelf solvers. A strategy that is sometimes viable is to solve a problem on a sequence of successively refined discretizations, starting from an interpolation of the previous solution as the initial guess.

Please feel free to reach out in the future in case you wish to discuss an optimization problem in your domain:

https://scoop.iwr.uni-heidelberg.de/

End of Class 4

§ 7 Solutions

§ 7.1 Solution of Example 2.2 (Rosenbrock) using CasADi (Problem-Based)

CasADi provides algorithmic differentiation for user-defined functions. We demonstrate below the description of the problem in a suitable form for CasADi's `nlpsol` solver interface.

```
# This code solves an unconstrained optimization problem with the Rosenbrock
# function using CasADi's python bindings. For lack of a solver specializing
# in unconstrained optimization solver, the solver of choice is nlpsol with
# ipopt, see https://web.casadi.org/docs/#nonlinear-programming.

# Resolve the dependencies.
from casadi import *
import matplotlib.pyplot as plt
import numpy as np

# Create a 1-by-1 optimization variable (termed 'x') and another 1-by-1
# optimization variable (termed 'y').
# SX is the class in CasADi representing matrices composed of symbolic expressions.
# There is also MX for more complex expressions.
x = SX.sym('x', 1, 1)
y = SX.sym('y', 1, 1)

# Set the fixed problem parameters.
a = DM(1)
b = DM(100)

# Setup the objective.
f = (a - x)**2 + b * (y - x**2)**2

# Setup the solver.
problem = {'x': vertcat(x,y), 'f': f}
solver = nlpsol('rosenbrock', 'ipopt', problem)

# Call the solver.
result = solver()

# Show the solution.
print()
print(result['x'])
```

§ 7.2 Solution of Example 2.2 (Rosenbrock) using CasADi (Opti Interface-Based)

CasADi offers the `Opti` interface, which allows a simplified description and solution of optimization problems.

```
# This code solves an unconstrained optimization problem with the Rosenbrock
# function using CasADi's python bindings, and CasADi's Opti interface for
# modeling of (un)constrained optimization problems.

# Resolve the dependencies.
from casadi import *

# Create an instance of the Opti interface.
opti = Opti()

# Create a 1-by-1 optimization variable (termed 'x') and another 1-by-1
# optimization variable (termed 'y').
x = opti.variable()
y = opti.variable()

# Set the fixed problem parameters.
a = opti.parameter()
b = opti.parameter()
opti.set_value(a, 1)
opti.set_value(b, 100)

# Setup the objective.
f = (a - x)**2 + b * (y - x**2)**2
opti.minimize(f)

# Setup the solver.
opti.solver('ipopt')

# Call the solver.
result = opti.solve()

# Show the solution.
print()
print(result.value(x), result.value(y))
```

§ 7.3 Solution of Example 2.3 (Airplane) using CasADi (Problem-Based)

Apparently CasADi has no interface for solving least-squares problems. Therefore, we resort to treating the problem as a generic unconstrained optimization problem. Consequently — in contrast to using a dedicated least-squares solver — we have to implement the objective $f(x) = \frac{1}{2}\sum_{i=1}^{M}[r_i(x)]^2$ instead of the residual function $r(x)$.

```
# This code solves a parameter estimation problem for a given model, based on a
# set of measurement pairs, using CasADi's python bindings for modeling of
# (un)constrained optimization problems.

# Resolve the dependencies.
from casadi import *
import matplotlib.pyplot as plt

# Create the unknown parameters as optimization variables.
xy = MX.sym('xy', 2, 1)

# Specify the pairs of measurements.
measurements = [
        ([ 8,  6],  38),
        ([-3, -3], 220),
        ([ 1,  0], 222),
        ([ 8, -3], 300),
        ]

# Declare the beacon position as a variable.
beacon = MX.sym('beacon position', 2, 1)

# Specify the model function.
angle = fmod(atan2(beacon[1] - xy[1], beacon[0] - xy[0]) + 2*pi, 2*pi) * 180 / pi
model = Function('airplane', [xy, beacon], [angle], ['xy', 'beacon'], ['angle'])

# Assemble the least-squares objective function.
f = 0
for i in range(len(measurements)):
    residual = model(xy = xy, beacon = measurements[i][0])['angle'] - measurements[i][1]
    f = f + residual**2
f = 0.5 * f

# Setup the initial guess.
x0 = [0, 0]

# Setup the solver.
problem = {'x': xy, 'f': f}
solver = nlpsol('airplane', 'ipopt', problem)

# Call the solver.
result = solver(x0 = x0)

# Show the solution.
print()
print(result['x'])
```

§ 7.4 Solution of Example 4.1 (Production) using Matlab's Optimization Toolbox (Solver-Based)

The classical use of Matlab's optimization toolbox required the user to model their optimization problem in a format suitable for the respective solver to be used. Since Example 4.1 is a linear optimization problem, `linprog` is the solver of choice.

```
% This code solves the production example problem using Matlab's optimization
% toolbox. The problem is a linear optization problem. The solver of choice is
% linprog from Matlab's optimization toolbox, see
% https://de.mathworks.com/help/optim/ug/linprog.html.

% Setup the cost vector describing the objective c'*x.
c = [-1; -1];

% Setup the matrix A and right hand side b for the inequality constraint
% A * x <= b.
A = [50, 24; 30, 33];
b = [2400; 2100];

% Setup the matrix Aeq and right hand side beq for the (void) equality constraint
% Aeq * x = beq.
Aeq = [];
beq = [];

% Setup the lower and upper bounds.
lb = [36; 5];
ub = [];

% Call the solver.
[x, fval] = linprog(c, A, b, Aeq, beq, lb, ub);

% Show the solution.
disp(x)
```

§ 7.5 Solution of Example 4.1 (Production) using Matlab's Optimization Toolbox (Problem-Based)

With more recent versions of Matlab's optimization toolbox, the user may formulate their problem in a more natural way, and independently of the solver. The solver will be picked automatically depending on the problem characteristics.

```
% This code solves the production example problem using Matlab's optimization
% toolbox. The problem is modeled using the problem-based approach, which
% allows for a more natural problem formulation and automatic selection of a
% suitable solver, see
% https://de.mathworks.com/help/optim/problem-based-basics.html.

% Create a maximization-type problem.
problem = optimproblem('ObjectiveSense', 'maximize', 'Description', 'production');

% Create a 2-by-1 optimization variable (termed 'x').
x = optimvar('x', 2, 1, 'LowerBound', [36; 5]);

% Setup the objective.
problem.Objective = x(1) + x(2);

% Setup the (named) inequality constraints.
problem.Constraints.timeConstraintA = 50 * x(1) + 24 * x(2) <= 2400;
problem.Constraints.timeConstraintB = 30 * x(1) + 33 * x(2) <= 2100;

% Review the problem.
show(problem)

% Call the solver.
result = solve(problem);

% Show the solution.
disp(result.x)
```

§ 7.6 Solution of Example 4.1 (Production) using `scipy.optimize`

Here we demonstrate the solution of Example 4.1 with `scipy.optimize.linprog`. We need to present the problem in a suitable form to the solver.

```
# This code solves the production example problem using the scipy.optimize
# package. The problem is a linear optization problem. The solver of choice is
# scipy.optimize.linprog, see
# https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html.

# Resolve the dependencies.
from scipy.optimize import linprog

#  Setup the cost vector describing the objective c @ x.
c = [-1, -1]

#  Setup the matrix A and right hand side b for the inequality constraint
#  A @ x <= b.
A = [[50, 24], [30, 33]]
b = [2400, 2100]

# Setup the lower and upper bounds.
bounds = [(36, None), (5, None)]

# Call the solver.
result = linprog(c, A_ub = A, b_ub = b, bounds = bounds)

# Show the solution.
print(result['x'])
```

§ 7.7 Solution of Example 4.1 (Production) using CasADi (Problem-Based)

CasADi provides algorithmic differentiation for user-defined functions (although this is not needed for linear optimization problems). We demonstrate below the description of the problem in a suitable form for CasADi's `nlpsol` solver interface.

```
# This code solves the production example problem using CasADi's python
# bindings. The problem is a linear optization problem. For lack of a linear
# optimization solver, the solver of choice is nlpsol with ipopt, see
# https://web.casadi.org/docs/#nonlinear-programming.

# Resolve the dependencies.
from casadi import *

# Create a 2-by-1 optimization variable (termed 'x').
# SX is the class in CasADi representing matrices composed of symbolic expressions.
# There is also MX for more complex expressions.
x = SX.sym('x', 2, 1)

# Setup the objective.
f = - x[0] - x[1]

#  Setup the constraint function g representing the inequality constraint
#  A @ x <= b.
# DM is the class in CasADi representing matrices with numerical values.
A = DM([[50, 24], [30, 33]])
g = A @ x

# Setup the upper bound for the constraint function g.
b = DM([2400, 2100])

# Setup the lower bound for the variable x.
l = DM([36, 5])

# Setup the solver.
problem = {'x': x, 'f': f, 'g': g}
solver = nlpsol('production', 'ipopt', problem)

# Call the solver, providing the upper bounds for the inequality constraint
# function g and the lower bound for the optimization variable.
result = solver(ubg = b, lbx = l)

# Show the solution.
print()
print(result['x'])
```

§ 7.8 Solution of Example 4.1 (Production) using CasADi (Opti Interface-Based)

CasADi offers the `Opti` interface, which allows a simplified description and solution of optimization problems.

```
# This code solves the production example problem using CasADi's python
# bindings, and CasADi's Opti interface for modeling of (un)constrained
# optimization problems.

# Resolve the dependencies.
from casadi import *

# Create an instance of the Opti interface.
opti = Opti()

# Create a 2-by-1 optimization variable (termed 'x').
x = opti.variable(2)

# Setup the objective.
opti.minimize(- x[0] - x[1])

#  Setup the constraint function representing the inequality constraint
#  A @ x <= b.
# DM is the class in CasADi representing matrices with numerical values.
A = DM([[50, 24], [30, 33]])
b = DM([2400, 2100])
opti.subject_to(A@x <= b)

# Setup the lower bound for the variable x.
l = DM([36, 5])
opti.subject_to(x >= l)

# Setup the solver.
opti.solver('ipopt')

# Call the solver.
result = opti.solve()

# Show the solution.
print()
print(result.value(x))
```

§ 7.9 Solution of Example 4.12 (Post-Office) using CasADi (Opti Interface-Based)

CasADi offers the `Opti` interface, which allows a simplified description and solution of optimization problems.

```
# This code solves Rosenbrock's post-oddice example problem using CasADi's
# python bindings, and CasADi's Opti interface for modeling of (un)constrained
# optimization problems.

# Resolve the dependencies.
from casadi import *

# Create an instance of the Opti interface.
opti = Opti()

# Create a 3-by-1 optimization variable (termed 'x').
x = opti.variable(3)

# Setup the objective.
opti.minimize(- x[0] * x[1] * x[2])

#  Setup the constraint functions.
opti.subject_to(x[0] + 2 * x[1] + 2 * x[2] - 72 <= 0)
opti.subject_to(x[0] + 2 * x[1] + 2 * x[2] >= 0)
opti.subject_to(opti.bounded(0, x, 42))

# Setup the solver.
opti.solver('ipopt')

# Call the solver.
result = opti.solve()

# Show the solution.
print()
print(result.value(x))
```

§ 7.10 Solution of Example 5.1 (van der Pol) using CasADi (Problem-Based)

```
# This code solves a discretized optimal control problem for the van der Pol
# oscillator.using CasADi's python bindings. The discretization is achieved
# using direct single shooting.

# Resolve the dependencies.
import numpy as np
from casadi import *
from matplotlib import pyplot as plt

# Define the control bounds.
lbu = -10
ubu = +10

# Define the final time and number of control intervals.
T = 10
nu = 10
control_grid = np.linspace(0, T, num = nu + 1)
dtu = T / (nu + 1)

# Define the step size for the state integrator and determine how many state
# points are associated with each control interval.
dtx_target = 0.1
states_per_control_interval = max([round(dtu / dtx_target), 1])

# Declare the control variables as our optimization variables.
U = MX.sym("u", 1, nu)

# Set the initial value for the state.
x0 = [0, 1]

# Define the parameter mu in the equation.
mu = 2

# Define the control cost parameter gamma.
gamma = 1

# Define the right-hand side function.
rhs = lambda x, u: vertcat(mu*(1 - x[1]**2) * x[0] - x[1] + u, x[0])

# Define a function which integrates the ODE one step.
# We use here an explicit Euler method for simplicity.
def take_step(x, u, dt):
    return x + dt * rhs(x, u)

# Set the initial state.
x = x0

# Prepare the integration loop and objective evaluation.
objective = 0
for step in range(nu):

    # Get the step size for the state integrator.
    dtu = control_grid[step+1] - control_grid[step]
    dtx = dtu / states_per_control_interval
```

```
    # Advance the state until the end of the current control interval and
    # update the objective.
    for substep in range(states_per_control_interval):
        x = take_step(x, U[step], dtx)
        objective = objective + dtx * (x[0]**2 + x[1]**2)

    # Add the control part to the objective.
    objective = objective + gamma * dtu * U[step]**2

# Setup the solver.
problem = {'x': U, 'f': objective}
solver = nlpsol('vanderpol', 'ipopt', problem)

# Call the solver.
result = solver(
        lbx = lbu,
        ubx = ubu,
)

# Show the solution.
U = np.array(result['x'])[0]
print()
print(U)
```

§ 7.11 Solution of Example 5.1 (van der Pol) using CasADi (Problem-Based) and Nested Iteration

```python
# This code solves a discretized optimal control problem for the van der Pol
# oscillator.using CasADi's python bindings. The discretization is achieved
# using direct single shooting. We use a nested iteration by solving the problem
# on a sequence of succcessively refined grids.

# Resolve the dependencies.
import numpy as np
from casadi import *
from matplotlib import pyplot as plt

# Define a solution function for the optimal control problem.
# T = final time
# nu = number of control intervals
# lbu = control lower bound
# ubu = control upper bound
# gamma = control cost parameter
# u0 = initial guess
def solve_ocp(T, nu, lbu, ubu, gamma, u0 = None):

    # Define the control grid and step size.
    control_grid = np.linspace(0, T, num = nu + 1)
    dtu = T / (nu + 1)

    # Define the step size for the state integrator and determine how many state
    # points are associated with each control interval.
    dtx_target = 0.1
    states_per_control_interval = max([round(dtu / dtx_target), 1])

    # Declare the control variables as our optimization variables.
    U = MX.sym("u", 1, nu)

    # Set the initial value for the state.
    x0 = [0, 1]

    # Define the parameter mu in the equation.
    mu = 2

    # Define the right-hand side function.
    rhs = lambda x, u: vertcat(mu*(1 - x[1]**2) * x[0] - x[1] + u, x[0])

    # Define a function which integrates the ODE one step.
    # We use here an explicit Euler method for simplicity.
    def take_step(x, u, dt):
        return x + dt * rhs(x, u)

    # Set the initial state.
    x = x0

    # Prepare the integration loop and objective evaluation.
    objective = 0
    for step in range(nu):
```

```
        # Get the step size for the state integrator.
        dtu = control_grid[step+1] - control_grid[step]
        dtx = dtu / states_per_control_interval

        # Advance the state until the end of the current control interval and
        # update the objective.
        for substep in range(states_per_control_interval):
            x = take_step(x, U[step], dtx)
            objective = objective + dtx * (x[0]**2 + x[1]**2)

        # Add the control part to the objective.
        objective = objective + gamma * dtu * U[step]**2

    # Setup the solver.
    problem = {'x': U, 'f': objective}
    solver = nlpsol('vanderpol', 'ipopt', problem)

    # Call the solver.
    if u0 is not None:
        result = solver(
                lbx = lbu,
                ubx = ubu,
                x0 = u0,
                )
    else:
        result = solver(
                lbx = lbu,
                ubx = ubu,
                )

    # Store the solution.
    U = np.array(result['x'])[0]

    # Integrate the trajectory again.
    t = [0]
    nx = nu * states_per_control_interval + 1
    X = np.zeros((len(x0), nx))
    X[:, 0] = x0
    for step in range(nu):
        # Get the step size for the state integrator.
        dtu = control_grid[step+1] - control_grid[step]
        dtx = dtu / states_per_control_interval
        # Advance the state until the end of the current control interval.
        for substep in range(states_per_control_interval):
            t.append(t[-1] + dtx)
            index = step * states_per_control_interval + substep
            X[:,index+1] = take_step(X[:,index], U[step], dtx).toarray()[:,0]

    return U, X, control_grid, t
```

```
# Define the control bounds.
lbu = -10
ubu = +10

# Define the final time and the initial number of control intervals.
T = 10
nu = 10

# Define the control cost parameter gamma.
gamma = 1

# Set the initial guess.
u0 = None

# Enter a refinement loop.
levels = 2
for level in range(levels):

    # Solve the problem.
    print('Solving with nu = {0:d} control variables.'.format(nu))
    U, X, control_grid, t = solve_ocp(T, nu, lbu, ubu, gamma, u0 = u0)

    # Prepare for the next round (if any).
    if level < levels-1:
        # Refine the control grid.
        nu = 2 * nu

        # Interpolate the control to be used as initial guess.
        u0 = np.ndarray.flatten(np.vstack([U, U]), order = 'F')

# Show the solution.
print()
print(U)
```

Bibliography

Andersson, J. A. E.; J. Gillis; G. Horn; J. B. Rawlings; M. Diehl (2018). "CasADi: a software framework for nonlinear optimization and optimal control". *Mathematical Programming Computation* 11.1, pp. 1–36. DOI: 10.1007/s12532-018-0139-4.

Chambolle, A.; T. Pock (2011). "A first-order primal-dual algorithm for convex problems with applications to imaging". *Journal of Mathematical Imaging and Vision* 40.1, pp. 120–145. DOI: 10.1007/s10851-010-0251-1.

Clason, C. (2017). *Nonsmooth analysis and optimization*. arXiv: 1708.04180.

Rudin, L. I.; S. Osher; E. Fatemi (1992). "Nonlinear total variation based noise removal algorithms". *Physica D* 60.1–4, pp. 259–268. DOI: 10.1016/0167-2789(92)90242-F.