An Introduction to Algorithmic Differentiation

Thomas Slawig
Christian-Albrechts-University zu Kiel
Cluster *Future Ocean*
24098 Kiel, Germany
ts@informatik.uni.kiel.de
www.informatik.uni-kiel.de/co2

Contents

CHAPTER 1

Introduction

What is Algorithmic Differentiation (AD)?. Algorithmic or Automatic Differentiation (AD) is a software technolgy. Assume there is a computer program given that evaluates a function F, i.e. for given input x it computes an output $y = F(x)$. Then the AD technology genaerates another computer programme (either in the form of source code or as binary) that evaluates both the function and its derivatiev, i.e. $F(x)$ and $F'(x)$.

Where can AD be useful? AD software can be useful everywhere where derivative information is useful, namely for

- the solution of nonlinear equations or systems of equations, e.g. by Newton's method,
- (Hermite) interpolation and Taylor series approximations, i.e. to derive simplified models
- sensitivity analysis
- uncertainty analysis
- optimization by local, gradient-based methods,
- optimal control, i.e. optimization with constraints that are given as ordinary or partial differential equations.

AD is specifically used when

- no or little analytical information of the underlying mathematical model is available
- this analytical information is hard and costly (with respect to time) to obtain
- only a computer program is available
- the program and/or the underlying model is rather complex
- the program undergoes a continuous development and thus changes rapidly

Why do applied mathematics study AD?. AD provides exact derivatives with respect to the coded function F (may be important in optimization). In contrary to finite difference derivatives no approximations are made, numerical instabilities – which are characteristic for finite difference derivatives – are avoided. Thus AD is a good alternative to deriving an analytical representation of the derivative and then discretizing it. AD uses the efficiency of the computer to symbolically differentiate exactly.

How does AD work? Every computer program basically is a (long) concatenation of elementary operations (i.e. operators and functions) of the used programming language. The derivatives of these elementary operations can be obtained easily. then the derivative of the whole function realized in the program can be computed by applying the chain rule of differentiation. AD software

- either generates new source code by inserting the derivative stements and chain rule operations directly in the code

- or uses operator and function overloading for a new datatype – consisting of both the value and the derivative – which is used for all needed quantities. The overloaded operators and functions are supplied in software libraries that are linked against the original code.

CHAPTER 2

The notion of differentiability

1. History

- Newton in "Methodus fluxonium et serierum infinitarum" 1671 (published 1736):
  - introduced the notion of "fluxion" for the relation between the " generated quantity (fluent quantity)" with respect to the change in time or an equivalent quantity, characterized by its continuous flow or growth"
  - used the notation $\dot{F}$ for his fluxions, which is still common in physics if the independent variable is time.
- Leibniz in "Calculus differentialis et integralis" 1675/76 (published in 1684)
  - introduced the notation $\frac{dF}{dt}$,
  - presented for example the chain rule.
- Cauchy: "Cours d'analyse de l'École Royale Polytechnique" (1823-29)
  - derivative as limit of difference quotient as now known from calculus:

$$F'(x) = \lim_{h \to 0} \frac{F(x+h) - F(x)}{h}$$

2. Definitions of differentiability

We present here the most important notions of differentiability for functions

$$F \quad : \quad X \supset D \to Y$$

with D being an open, non-empty subset of a linear space X, and Y another space. This setting is the most general. We will distinguish between whether X is finite or infinite-dimensional. In the first case we will discuss $X = \mathbb{R}^n$ – the extension for a finite-dimensional $X \neq \mathbb{R}^n$ is straightforward. For Y this difference is not that important, we assume that this space has the properties that are required for the definitions below, i.e. that it is a normed space if the definition incorporate norms of $F(x)$, and so on.

Note that we will define our function F such that X is the space which consists only of those parameters, with respect to whom the differentiation shall be performed.

Example 1. *A typical example in optimization is the* Rosenbrock function

$$\begin{aligned} F: \qquad & X = D := \mathbb{R}^2 \to \mathbb{R} =: Y \\ F(x) \quad = \quad & 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \end{aligned}$$

where both X and Y are finite-dimensional.

Infinite-dimensional examples typically involve differential equations, when the differentiation is studied before discretizing the equation.

EXAMPLE 2. *Consider the initial value problem (IVP)*

$$\begin{aligned} \dot{y}(t) &= f(t, y(t)), \quad t \in (0, T) \\ y(0) &= \alpha \end{aligned} \tag{2.1}$$

with $T > 0$, some linear or nonlinear function $f : (0, T) \times D \to \mathbb{R}^s, D \subset \mathbb{R}^n$, and $\alpha \in \mathbb{R}^n$. If we assume that the IVP is uniquely solvable, and that the solution is differentiable with respect to the initial data α, then we may be interested in the derivative of the function

$$\begin{aligned} F : & \quad \mathbb{R}^n \to Y \\ F(\alpha) &= y, \text{ the solution of } (\mathbf{??}). \end{aligned}$$

Here Y is an appropriate function space.

Similar cases can be considered with partial differential equations.

EXAMPLE 3. *Consider the weak solution of the PDE*

$$\begin{aligned} -\alpha \triangle y &= f \quad \text{in} \quad \Omega \\ y &= 0 \quad \text{on} \quad \partial\Omega \end{aligned} \tag{2.2}$$

with $f \in L^2(\Omega)$ and a parameter $\alpha \in \mathbb{R}^+$. It is known that $y \in H_0^1(\Omega)$. If we are interested in the dependency of y with respect to α, we may set $x = \alpha$ and study the differentiabilty of the function

$$\begin{aligned} F \ : \ \mathbb{R}^+ &\to H_0^1(\Omega) \\ \alpha &\mapsto y, \text{ the solution to } (2.2). \end{aligned}$$

Thus here X is finite-dimensional, and Y is infinite-dimensional.

EXAMPLE 4. *If we are interested in the dependency of Y with respect to f in (2.2), we set $x := f$ and*

$$\begin{aligned} F \ : \ L^2(\Omega) &\to H_0^1(\Omega) \\ f &\mapsto y, \text{ the solution to } (2.2). \end{aligned}$$

Thus here both X and Y are infinite-dimensional.

If the differentiabiilty is discussed after the discretization of (2.2), one ends up with a finite-dimensional setting again:

EXAMPLE 5. *A discretization of (2.2) with a Galerkin method results in the problem to compute an approximative solution $y = \sum_{i=1}^n y_i \phi_i$ of (2.2) in a finite-dimensional space $Y = \operatorname{span}\{\phi_i, i = 1, \ldots, n\}$. If the inhomogeneity in (2.2) is discretized as $f = \sum_{j=1}^n f_j \psi_j$ we get*

$$\begin{aligned} F \ : \ X &\to Y \\ f &\mapsto y, \text{ the approximate weak solution to } (2.2) \text{ in } Y. \end{aligned}$$

EXAMPLE 6. *To solve the Galerkin approximation one has to compute the solution of a linear system*

$$Ay = f, \tag{2.3}$$

where now $A \in \mathbb{R}^{n \times n}$ and $y, f \in \mathbb{R}^n$. This is also what one obtains when applying another discretization method, e.g. finite difference schemes. Thus one may set

$$\begin{aligned} F \ : \ \mathbb{R}^n &\to \mathbb{R}^n \\ f &\mapsto y, \text{ the solution to } (2.3). \end{aligned}$$

Here the discretization level determines the dimension of the (finite-dimensional) spaces X and Y.

The following definitions characterize differentiability in a single point $x \in D$. We say that F is partial/directional/Gâteaux/Fréchet differentiable in D, if it is partial/directional/Gâteaux/Fréchet differentiable, respectively, for all $x \in D$. Derivatives of a vector-valued function F can be defined component-wise.

2.1. Partial Differentiability. The generalization of the definition of the "usual" derivative for a one-dimensional function is the partial derivative, where the limit is taken with respect to one coordinate. Naturally this definition only make sense in a finite-dimensional spaces X.

DEFINITION 1 (Partial derivative). *Let $F : \mathbb{R}^n \supset D \to \mathbb{R}^m$. If for $x \in D$ and $i = 1, \ldots, n$ the limit*

$$\begin{aligned} \frac{\partial F}{\partial x_i}(x) &:= \lim_{t \to 0} \frac{F(x + te_i) - F(x)}{t} \\ &= \lim_{t \to 0+} \frac{F(x_1, \ldots, x_{i-1}, x_i + t, x_{i+1}, \ldots, x_n) - F(x_1, \ldots, x_n)}{t} \end{aligned} \tag{2.4}$$

exists, then it is called the i-th partial derivative of F in x. If it exists for all $i = 1, \ldots, n$, then F is called partially differentiable in x.

A function that is partially differentiable has not to be continuous.

EXAMPLE 7. *We consider the function*

$$F : \mathbb{R}^2 \to \mathbb{R}, \qquad F(x_1, x_2) := \begin{cases} \dfrac{x_1 x_2}{(x_1^2 + x_2^2)^2} & x \neq 0 \\ 0 & x = 0 \end{cases}$$

At the point $(0, 0)$ both partial derivatives vanish since

$$\frac{\partial F}{\partial x_1}(0,0) = \lim_{t \to 0+} \frac{F(t,0) - F(0,0)}{t} = 0,$$

and the same for the $\frac{\partial F}{\partial x_2}$. Taking the sequence $(t, t), t \to 0$ gives

$$\lim_{t \to 0} F(t,t) = \lim_{t \to 0} \frac{t^2}{4t^4} = \infty.$$

Thus F is not continuous in $(0, 0)$.

2.2. Directional Differentiability. A function may not be (partial) differentiable, just because the limit in (2) has different values, depending on the sign that is taken for t. Think of the modulus function for example. Then the weaker concept of directional differentiability is useful.

DEFINITION 2 (Directional derivative). *Let $F : X \supset D \to Y$. If the limit*

$$F'(x)v := \lim_{t \to 0+} \frac{F(x + tv) - F(x)}{t}, \quad v \in X, \tag{2.5}$$

exists for $x \in D$, then it is called the (directional) derivative of F in x in direction v. If it exists for all $v \in X$, then F is called directionally differentiable in x.

This concept can be even used in topological spaces X, Y.

For $X = \mathbb{R}^n, Y = \mathbb{R}^m$ the function F is partial differentiable in the i-th coordinate if it is differentiable in the directions $v = \pm e_i$, and

$$F'(x)e_i = -F'(x)(-e_i). \tag{2.6}$$

The directional derivative neither has to be linear nor continuous (which is the same for finite-dimensional X) with respect to the direction.

EXAMPLE 8. *We consider the modulus function*

$$F : \mathbb{R} \to \mathbb{R}, \qquad F(x) := |x|.$$

In $x = 0$ it has the directional derivatives

$$\begin{aligned} v = 1: \quad F'(0)v &= \lim_{t\to 0+} \frac{F(t) - F(0)}{t} = \lim_{t\to 0+} \frac{t}{t} = 1 \\ v = -1: \quad F'(0)v &= \lim_{t\to 0+} \frac{F(-t) - F(0)}{t} = \lim_{t\to 0+} \frac{t}{t} = 1. \end{aligned}$$

Thus F is directional differentiable in $x = 0$ in all directions, but not differentiable, compare (2.6). Moreover the mapping

$$v \quad \mapsto \quad F'(0)v$$

is not linear.

2.3. Gâteaux Differentiability. If the directional derivative is a linear and continuous mapping, and it exists for all directions $v \in X$, it is called a Gâteaux derivative.

DEFINITION 3 (Gâteaux derivative). *If for $F : X \supset D \to Y$ the limit (2.5) exists for all $v \in X$, and the mapping*

$$F'(x) : X \to Y, \quad v \mapsto F'(x)v, \tag{2.7}$$

is linear and continuous, then it is called the Gâteaux derivative of F in x, and F is called Gâteaux differentiable in x.

Still a function that is Gâteaux differentiable does not have to be continuous.

EXAMPLE 9. *The function*

$$F : \mathbb{R}^2 \to \mathbb{R}, \qquad F(x_1, x_2) := \begin{cases} 1, & x_1 \neq 0, x_2 = x_1^2 \\ 0, & \text{elsewhere} \end{cases}$$

has the directional derivative

$$F'(0,0)(v_1, v_2) \quad = \quad 0$$

at the point $(0, 0)$, for all directions (v_1, v_2). The mapping

$$(v_1, v_2) \quad \mapsto \quad F'(0,0)(v_1, v_2) = 0$$

clearly is linear and continuous with respect to (v_1, v_2). Thus it is a Gâteaux derivative. But F itself is clearly not continuous in $(0, 0)$. Crucial here is the fact that the set of discontinuity is not a straight line, but a curve. Since for directional derivatives the limit (2) is always taken along straight lines the discontinuity does not affect the limit, and thus the differentiability.

2.4. Fréchet Differentiability. The highest level of differentiability in $\mathbb{R}^n$ or a general normed space is the Fréchet differentiability. Here additionally the remainder term has the decreasing property known from differentiabilty in $\mathbb{R}$.

DEFINITION 4 (Fréchet derivative). *Let $F : X \supset D \to Y$ and $x \in D$. If there exists a linear and continuous mapping $F'(x) : X \to Y$ with*

$$\lim_{\|v\|_X \to 0} \frac{\|F(x+v) - F(x) - F'(x)v\|_Y}{\|v\|_X} \quad = \quad 0, \tag{2.8}$$

or equivalently

$$F(x+v) = F(x) + F'(x)v + R(x,v) \quad \text{with} \quad \lim_{\|v\|_X \to 0} \frac{\|R(x,v)\|_Y}{\|v\|_X} = 0, \tag{2.9}$$

then $F'(x)$ is called the Fréchet derivative of F in x, and F is called Fréchet differentiable in x.

For the Fréchet differentiability in $\mathbb{R}^n$ also the notion *total differentiability* or just *differentiability* is used.

For finite-dimensional X and Y the derivative, as a linear mapping, can be identified with the *Jacobian* or *functional matrix*

$$F'(x) = \begin{pmatrix} \frac{\partial F_1}{\partial x_1}(x) & \cdots & \frac{\partial F_1}{\partial x_n}(x) \\ \vdots & & \vdots \\ \frac{\partial F_m}{\partial x_1}(x) & \cdots & \frac{\partial F_m}{\partial x_n}(x) \end{pmatrix} \in \mathbb{R}^{m\times n}.$$

For $F : X \to \mathbb{R}$ also the term gradient and the notation $\nabla F(x)$ is used. In the case of a finite-dimensional space X it can be identified with the Jacobian. We will use the gradient as a *column* vector, i.e. we set

$$F'(x) \quad = \quad \left(\frac{\partial F}{\partial x_1}(x), \cdots, \frac{\partial F}{\partial x_n}(x)\right) \quad =: \quad \nabla F(x)^\top \in \mathbb{R}^n.$$

If F is Fréchet differentiable in $x \in \mathbb{R}^n$ with derivative $F'(x)$, the directional derivative is given as the matrix-vector product $F'(x)v$.

The following lemma summarizes some results about Gâteaux and Fréchet derivatives.

LEMMA 1. *Let $F : X \supset D \to Y$.*

(a) *The Fréchet derivative is unique.*
(b) *If $F : D \to Y$ is Fréchet differentiable in x, then it is also Gâteaux differentiable and both derivatives coincides.*
(c) *If $F : D \to Y$ is Fréchet differentiable in x, then it is continuous in x.*
(d) *If $F : D \to Y$ is Gâteaux differentiable in D, and the mapping*

$$F' : D \to \mathcal{L}(X, Y), \quad x \mapsto F'(x)$$

is continuous, then F is Fréchet differentiable in D.

PROOF. (a) [**Jah94**, Corollary 3.14], (b) [**Jah94**, Theorem 3.13], (c) [**Jah94**, Theorem 3.15], (d). □

The gap between Gâteaux and Fréchet differentiability is not empty, as the following example shows.

EXAMPLE 10. *The function from Example 9 is not Fréchet differentiable. This is clear from the above lemma, since F is not continuous in $(0,0)$. In the limit (2.8) we take the sequence $v = (t, t^2), t \to 0+$. Since F is Gâteaux differentiable in $(0,0)$ with $F'(0,0)v = 0$ for all $v \in \mathbb{R}^2$, this also would have to be the Fréchet derivative. We obtain*

$$\begin{aligned} \lim_{\|v\|x\to 0} \frac{\|F(x+v) - F(x) - F'(x)v\|}{\|v\|} &= \lim_{t\to 0+} \frac{\|F(t,t^2)\| - F(0,0) - 0}{\|(t,t^2)\|} \\ &= \lim_{t\to 0+} \frac{1}{t\sqrt{1+t^2}}, \end{aligned}$$

which does not exist. Changing the definition of F slightly, namely in $(0,0)$, to

$$F : \mathbb{R}^2 \to \mathbb{R}, \qquad F(x_1, x_2) := \begin{cases} 1, & x_2 = x_1^2 \\ 0, & \textit{elsewhere} \end{cases}$$

gives 0 for the above limit. But now the directional derivative

$$F'(0,0)(v_1, v_2) \quad = \quad \lim_{t\to 0+} \frac{F(tv_1, tv_2) - F(0,0)}{t} = \lim_{t\to 0+} \frac{-1}{t}$$

does not exist. This corresponds to a choice of the sequence $v = (t, t), t \to 0+$, *in the limit (2.8), which then is infinity, too. Thus with this modification* F *is not even directional differentiable.*

If $X = \mathbb{R}$ both Gâteaux and Fréchet differentiability coincide to the "usual" notion of differentiability.

If $X = X_1 \times \ldots \times X_n$ is finite-dimensional with the X_i being topological or normed spaces, the notion of partial Gâteaux and Fréchet differentiability in $x \in X$ is used to characterize the derivative with respect to one variable $x_i \in X_i$.

2.5. Higher Derivatives. Higher derivatives are defined by applying the same concepts to the first derivative, and so on. The second derivative of $F : X \supset D \to Y$ in x is defined by

$$F''(x) \quad : \quad X \times X \to Y, \quad (v, w) \mapsto F''(x)[v, w] \in Y.$$

For $X = \mathbb{R}^n, Y = \mathbb{R}$ it is given by the transposed Hessian matrix

$$F''(x) \quad = \quad \nabla^2 F(x)^\top$$

with

$$\nabla^2 F(x) \quad := \quad \begin{pmatrix} \dfrac{\partial^2 F}{\partial x_1^2}(x) & \cdots & \dfrac{\partial^2 F}{\partial x_n \partial x_1}(x) \\ \vdots & & \vdots \\ \dfrac{\partial^2 F}{\partial x_1 \partial x_n}(x) & \cdots & \dfrac{\partial^2 F}{\partial x_n^2}(x) \end{pmatrix},$$

sometimes also denoted by $H(x)$.

Further notions of differentiability, for example Clark's derivative and the subdifferential can be found in [**Jah94**, Sections 3.3-3.5].

CHAPTER 3

Use of derivative information in applied mathematics

1. Solution of nonlinear equations

Consider a non-linear mapping $F : X \supset D \to Y$. A classical way to solve the equation

$$F(x) = 0 \quad \text{in } Y \tag{3.1}$$

is Newton's method:

ALGORITHM 1 (Newton's method).

(1) *Choose* $x^{(0)} \in X$ *and set* $k = 0$.
(2) *For* $k = 0, 1, \ldots$:
   (a) *Compute* d_k *from* $F'(x^{(k)})d_k = -F(x^{(k)})$
   (b) *Set* $x^{(k+1)} = x^{(k)} + d_k$
   until a stopping criterion $K(x^{(k+1)}, x^{(k)}) \leq \varepsilon$ *is satisfied.*

Newton's method has a locally quadratic convergence rate.

THEOREM 1. *If* X *is a Banach space,* $D \subset X$ *open and convex, and* F *in* D *continuously Fréchet differentiable. Let* $F'(x^{(0)})$ *be invertible for some* $x^{(0)} \in D$ *and*

$$\begin{aligned} \|F'(x^{(0)})^{-1}(F'(\bar{x}) - F'(x))\| &\leq \beta\|\bar{x} - x\|_X \quad \textit{f.a. } \bar{x}, x \in D, \\ \|F'(x^{(0)})^{-1}F(x^{(0)})\| &\leq \alpha, \\ \textit{with} \quad \alpha\beta \leq \frac{1}{2}, \quad \bar{B}(x^{(0)}, \theta) &\subset D, \quad \theta := \frac{1 - \sqrt{1 - 2\alpha\beta}}{\beta} \end{aligned}$$

Then the sequence $\{x^{(k)}\}$ *defined in Algorithm 1 is well-defined in* $\bar{B}(x^{(0)}, \theta)$ *and converges quadratically to a root of* F.

PROOF. See [**Deu04**, Theorem 2.1] or – slightly modified – [**Lue69**, §10.3 Theorem 1] and the remarks below. □

To reduce the effort of computing the full Jacobian $F'(x^{(k)})$ in every step, it can be updated only after a certain number of iterations. A convergence result for this simplified variant of Newton's method in the case $X = \mathbb{R}^n$ can be found in [**Deu04**, Theorem 2.5].

The problem of local convergence of Newton's method can be treated by introducing a variable step-size ρ_k, i.e to replace the second step in Algorithm 1 by

2.(b) Choose $\rho_k > 0$ *and set* $x^{(k+1)} = x^{(k)} + \rho_k d_k$.

This class of methods is studied in [**Deu04**, Sections 3.3 and 3.4], a global convergence result can be found in [**Deu04**, Theorem 3.14].

Another way to solve (3.1) is to re-write it as a least-squares optimization problem

$$\min_{x \in X} \frac{1}{2} \|F(x)\|_Y^2,$$

and treat it with methods for unconstrained optimization, see below.

2. Characterization of fix points

In the case $Y \subset D \subset X$ problem (3.1) can be transfomed to the equivalent fix point problem

$$\bar{F}(x) \quad = \quad x \quad \text{in } X \tag{3.2}$$

using $\bar{F}(x) := F(x) + x$. A solution procedure and existence result gives the famous Banach fix point theorem.

THEOREM 2 (Banach fix point theorem 1922). *Let D be a closed subset of the Banach space X and $F : D \to D$ a contraction, i.e. there exists $K < 1$ such that*

$$\|F(x_1) - F(x_2)\|_X \quad \leq \quad K\|x_1 - x_2\|_X \quad \textit{for all } x_1, x_2 \in D. \tag{3.3}$$

Then F has exactly one fix point $x^* \in D$, and the iteration

$$x^{(i+1)} \quad = \quad F(x^{(i)}), \quad i = 0, 1, \ldots,$$

converges to x^* for arbitrary $x^{(0)} \in D$. Moreover the following error estimates hold.

$$\|x^* - x^{(i)}\| \quad \leq \quad \frac{K^i}{1-K}\|x^{(1)} - x^{(0)}\| \qquad \textit{(a-priori)}$$

$$\|x^* - x^{(i)}\| \quad \leq \quad \frac{K}{1-K}\|x^{(i)} - x^{(i-1)}\| \quad \textit{(a-posteriori).}$$

PROOF. See [**Ruz04**, Satz 1.5]. The theorem remains valid in a complete metric space. □

One way to show (3.3) is to use the derivative of F.

THEOREM 3. *If $F : D \to D$ is Fréchet differentiable $x \in D$ with*

$$\|F'(x)\|_X \quad \leq \quad K,$$

then (3.3) is satisfied in a neighborhood of x .

PROOF. This is a consequence of the mean value theorem. □

Thus the derivative gives information about the convergence of a fix point iteration. This motivates the following characterization of fix points.

DEFINITION 5. *Let X be a normed space, $D \subset X$ und $F : D \to D$. A fix point $x \in D$ of F is called*

- *attractive if* $\|F'(x)\|_X < 1$,
- *repulsive if* $\|F'(x)\|_X > 1$,
- *and indifferent if* $\|F'(x)\|_X = 1$.

3. Interpolation and reduced models

Taylor expansion

$$F(x) \approx \sum_{i=0}^{k} \frac{1}{i!} F^{(i)}(\bar{x})(x-\bar{x})^i \tag{3.4}$$

Hermite interpolation:

$$F(x) \approx \sum_{i=0}^{k} \left(F(x_i)H_{1i}(x) + F'(x_i)H_{2i}(x)\right), \tag{3.5}$$

$H_{ji} \in \Pi_{2n}$ basis.

4. Uncertainty analysis

Taylor expansion (3.4) of scalar funktion F and vector of uncertain input parameters $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$. Expansion of F about the mean $x^0 := E[x]$. Define $y^0 = F(x^0)$. Mean of the result $y = F(x)$ of y^0, using linearity of the mean:

$$\begin{aligned} E[y-y^0] &\approx E\left[\sum_{j=1}^{n} \frac{\partial F}{\partial x_j}(x^0)(x_j - x_j^0) + \sum_{j,k=1}^{n} \frac{\partial^2 F}{\partial x_j \partial x_k}(x^0)(x_j - x_j^0)(x_k - x_k^0)\right] \\ &= \sum_{j=1}^{n} \frac{\partial F}{\partial x_j}(x^0) E[x_j - x_j^0] + \sum_{j,k=1}^{n} \frac{\partial^2 F}{\partial x_j \partial x_k}(x^0) E[(x_j - x_j^0)(x_k - x_k^0)]. \end{aligned}$$

With $E[x_j - x_j^0] = 0$ and definition of *covariance* of x,

$$\mathrm{Covar}[x_j, x_k] := E[(x_j - x_j^0)(x_k - x_k^0)],$$

we get for the mean of the difference between y and y^0:

$$E[y-y^0] \approx \sum_{j,k=1}^{n} \frac{\partial^2 F}{\partial x_j \partial x_k}(x^0)\mathrm{Covar}[x_j, x_k].$$

To compute the variance $V[y]$ of y we obtain with Taylor expansion $E[y] \approx y^0$ and thus

$$\begin{aligned} V[y] &:= E[(y-y^0)^2] \\ &\approx E\left[\left(\sum_{j=1}^{n} \frac{\partial F}{\partial x_j}(x^0)(x_j - x_j^0)\right)^2\right] \\ &= E\left[\sum_{j,k=1}^{n} \frac{\partial F}{\partial x_j}(x^0)\frac{\partial F}{\partial x_k}(x^0)(x_j - x_j^0)(x_k - x_k^0)\right] \\ &= \sum_{j,k=1}^{n} \frac{\partial F}{\partial x_j}(x^0)\frac{\partial F}{\partial x_k}(x^0) E[(x_j - x_j^0)(x_k - x_k^0)] \\ &= \sum_{j=1}^{n} \left(\frac{\partial F}{\partial x_j}(x^0)\right)^2 V[x_j] + 2\sum_{j,k=i+1}^{n} \frac{\partial^2 F}{\partial x_j \partial x_k}(x^0)\mathrm{Covar}[x_j, x_k]. \end{aligned}$$

Here the definitions of variance and covariance were used. If the x_i are independent, the last term vanishes and we get

$$V[y] \approx \sum_{j=1}^{n} \left(\frac{\partial F}{\partial x_j}(x^0)\right)^2 V[x_j].$$

See [**MH90**]. Also Hermite interpolation (3.5) is used, compare [**?**].

5. Optimality conditions

Derivative information can be used to characterize minima of differentiable functions and to approximate them in an iterative algorithm. We study the problem

$$\min_{x \in X_{ad}} F(x) \tag{3.6}$$

for the set $X_{ad} \subset X$ of admissible parameters. In the case $X_{ad} = X$ we call (3.6) an unconstrained and in the case $X_{ad} \neq X$ an constrained one. We look for a solution in the following sense.

DEFINITION 6. *An element $x^* \in X_{ad}$ is called*

- *global solution of (3.6) or global minimum of F, if $F(x^*) \leq F(x)$ for all $x \in X_{ad}$ and*
- *local solution of (3.6) or local minimum of F, if $F(x^*) \leq F(x)$ for all x in a neighborhood $V(x^*) \subset X_{ad}$ of x^*.*

We study results for existence of solutions and optimality conditions in the general constrained case.

5.1. A Global Existence result. The following general result does not contain any derivative information.

THEOREM 4. *Let X_{ad} be weakly sequentially compact and F weakly lower-continuous, i.e. for $x_n \rightharpoonup x$ in X it follows that $F(x) \leq \liminf_{n\to\infty} F(x_n)$. Then F has a global minimum in X_{ad}.*

PROOF. S. [**Jah94**, Theorem 2.3]. □

5.2. First and second order conditions. For the characterization of local minima derivative plays a crucial role. To define a directional derivative for constrained problems, the notion of admissible directions is used.

DEFINITION 7. *Let $\bar{x} \in X_{ad}$. The direction $x - \bar{x} \in X$ is called admissible, if there exists $(a_n)_n \subset \mathbb{R}, a_n \to 0$, such that $\bar{x} + a_n(x - \bar{x}) \in X_{ad}$ for all $n \in \mathbb{N}$.*

We obtain the following first order optimality condition.

THEOREM 5 (Variational inequality). *If $x^* \in X_{ad}$ is a local minimum of F, and F has a directional derivative in x^* in the admissible direction $x - x^*$, then*

$$F'(x^*)(x - x^*) \quad \geq \quad 0. \tag{3.7}$$

PROOF. S. [**Jah94**, Theorem 3.8(a)]. □

For convex X_{ad} closedness is equivalent to weak closedness. If additionally F is convex, every local minimum is also a global one, and (3.7) is also sufficient for a minimum, see [**Jah94**, Theorem 3.8(b)].

Is there information about the second derivative of F, then positiv definiteness of F'' implies convexity of F. This gives the following sufficient condition.

THEOREM 6. *Let X be reflexive, $X_{ad} \subset X$ be convex and closed, and F Gâteaux differentiable and twice directionally differentiable in U in all directions $(v_1, v_2) \in X \times X$. If*

$$F''(x)[v, v] \quad \geq \quad r(\|v\|_X)\|v\|_X \quad \text{f.a. } x \in X_{ad}, v \in X,$$

with a function $r : [0, \infty) \to [0, \infty), \lim_{s\to\infty} r(s) = \infty$. then (3.6) has a solution in U. If $r(s) > 0$ for $s > 0$, then the solution is unique.

PROOF. S. [**Kun**, Chapter 1 Theorem 2.10] □

If $x^* \in U$ thus satisfies (3.7) and additionally for example a condition like

$$F''(x^*)[v,v] \quad \geq \quad \alpha \|v\|_X^2 \tag{3.8}$$

with $\alpha > 0$ in a neighborhood of x^*, then x^* is a local minimum.

5.3. Lagrange multiplier rule. In the following theorem the admissible set is characterized by equality constraints, i.e.

$$X_{ad} \quad = \quad \{x \in X : e(x) = 0\} \tag{3.9}$$

with $e : X \to Z$ and Banach spaces X, Z. For the existence of Lagrange multipliers the following definition is used.

DEFINITION 8. *Let $e : X \supset V \to Z$ be continuously Fréchet differentiable. A point $x \in V$ is called regular w.r.t. e, if the mapping $e'(x) : X \to Z$ is surjective.*

The Lagrange functional or Lagrangian is defined as

$$L : X \times Z^* \to \mathbb{R}, \qquad L(x,\lambda) := F(x) + \langle \lambda, e(x) \rangle_{Z^*,Z}. \tag{3.10}$$

The following theorem holds. Here L_x denotes the partial derivative of L w.r.t. x.

THEOREM 7. *Let $F : X \to \mathbb{R}$ be continuously Fréchet differentiable. If the regular point x^* is a local minimum of F in X_{ad}, then there exists $\lambda \in Z^*$ with*

$$L_x(x^*,\lambda) \quad = \quad F'(x^*) + \langle \lambda, e'(x^*) \rangle_{Z^*,Z} \; = \; 0. \tag{3.11}$$

PROOF. See [**Lue69**, §9.3 Theorem 1] or [**IT79**, Abschnitt 1.1.1 Satz 1], also for the general case where x^* is not regular, but $e'(x^*)$ has closed range. □

Using the second derivative, a necessary and a sufficient optimality condition can be stated.

THEOREM 8. *Let F and e twice Fréchet differentiable in the regular point $x^* \in X$. If x^* is a minimum of F, then*

$$L_{xx}(x^*,\lambda)[v,v] \quad \geq \quad 0 \qquad \text{f.a. } v \in \ker e'(x^*).$$

PROOF. See [**IT79**, Folgerung in Abschnitt 7.2.1]. □

THEOREM 9. *Let F, e und x^* as in Theorem 8. If there is $\lambda \in Z^*$ with (3.11) and*

$$L_{xx}(x^*,\lambda)[v,v] \quad \geq \quad \alpha \|v\|_X^2 \qquad \text{f.a. } v \in \ker e'(x^*)$$

for $\alpha > 0$, then x^* is a minimum of F.

PROOF. See [**IT79**, Hilfssatz 1, Abschnitt 7.2.2]. □

For inequality constraints, i.e. $X_{ad} = \{x \in X : g(x) \leq 0\}$ we assume that X is a linear space and Z a normed linear space with a positive cone P with nonempty interior (for the Definition see [**Lue69**, §8.2]). The definition of a regular point is modified as follows.

DEFINITION 9. *Let $e : X \to Z$ be Gâteaux differentiable. Then $x \in X$ is called regular w.r.t. $e(x) \leq 0$, if there exists $h \in X$ with*

$$e(x) + e'(x)h \quad < \quad 0.$$

The result equivalent to Theorem 7 is called Kuhn-Tucker-Theorem.

THEOREM 10 (General Kuhn-Tucker-Theorem). *let F, e Gâteaux differentiable. Let x^* be a regular point w.r.t $e(x) \leq 0$. If X^* is a local minimum of F in X_{ad}, then there is $\lambda \in Z^*$ satisfying (3.11), $\lambda \geq 0$ and $\langle \lambda, g'(x^*) \rangle_{Z^*,Z} = 0$.*

PROOF. See [**Lue69**, §9.4 Theorem 1]. □

5.4. Unrestricted Problems. In this case Theorem 5 takes the following simpler form.

COROLLARY 1. *Let $X_{ad} = X$.*

(a) *If X is reflexive, F weakly lower semi-continuous and radial unbounded, then F has a global minimum.*
(b) *If $x^* \in X$ is a local minimum of F and F is Gâteaux differentiable in x^*, then*

$$F'(x^*) \quad = \quad 0 \quad \text{in } X^*. \tag{3.12}$$

PROOF. For (b) see also [**Jah94**, Theorem 3.17]. □

6. Optimization algorithms

6.1. General descent method for unconstrained problems. In the case $X_{ad} = X$ a descent method has the following general form.

ALGORITHM 2 (General descent method).

(1) *Choose $x^{(0)} \in X$.*
(2) *For k=0,1,...:*
  (a) *determine descent direction $d_k \in X$*
  (b) *determine step size $\rho_k > 0$*
  (c) *set $x^{(k+1)} = x^{(k)} + \rho_k d_k$*

  until a stopping criterion $K(x^{(k+1)}, x^{(k)}) \leq \varepsilon$ is satisfied.

6.2. Gradient and related methods. In the gradient method we set

2.(a) $d_k = -F'(x^{(k)})$.

More generally:

DEFINITION 10. *A direction d_k is called gradient-related if there exists $\mu > 0$ such that*

$$\left.\begin{array}{rcl} -F'(x^{(k)})d_k & \geq & \mu \|F'(x^{(k)})\| \|d_k\| \\ d_k \neq 0 & \text{for} & F'(x^{(k)}) \neq 0 \end{array}\right\} \tag{3.13}$$

Convergence result:

THEOREM 11. *Let X be a Hilbert space, $x^{(0)} \in X$, and let*

- *F be bounded from below*
- *F be twice Fréchet differentiable in X with*

$$m\|v\|^2 \quad \leq \quad (F''(x)v, v)_X \quad \leq \quad M\|v\|^2 \quad \text{for all } x \in S, v \in X$$

  for some $m, M > 0$, S the closed convex hull of $\{x \in X : F(x) < F(x^{(0)})\}$,
- *the directions d_k satisfy (3.13),*
- *the step sizes ρ_k satisfy*

$$F'(x^{(k)} + \rho_k d_k) \;=\; F(x^{(k+1)}) \quad \leq \quad F(x^{(k)}) - \frac{m^2}{2M}\|F'(x^{(k)})\|^2. \tag{3.14}$$

Then the sequence $\{x^{(k)}\}$ generated by Algorithm 2 satisfies

- *$x^{(k)} \to x^* \in X$, a solution of (3.6),*
- *$F'(x^{(k)}) \to 0$.*

PROOF. Generalization of [**Lue69**, §10.5 Theorem 2], s. [**Kun**, Chapter 3 Theorem 1.2]. □

For an exact line search, i.e.

$$\rho_k \quad = \quad \operatorname*{argmin}_{\rho>0} F(x^{(k)} + \rho d_k),$$

the directions d_k are orthogonal to the gradients, i.e.

$$(F'(x^{(k+1)}), d_k)_X \quad = \quad 0 \quad \text{for all } k. \tag{3.15}$$

In the quadratic case $F(x) = \frac{1}{2}(Qx, x) - (b, x)$ with self-adjoint $Q, b \in X$ the convergence speed is determined by the realtion of m, M in in Theorem 13, which then are bounds for the spectrum of Q. These relation can be improved by an appropriate coordinate transformation. In the non-linear case this idea results in thh choice of a preconditioner C_k and

$$d_k = -C_k^{-1} F'(x^{(k)})$$

such that

$$\left.\begin{array}{rcl} \|C_k^{-1}x\| & \leq & k_1\|x\| \\ (C_k^{-1}x, x)_X & \geq & k_2\|x\|^2 \end{array}\right\} \qquad \text{for all } k = 0, 1, \ldots, x \in X.$$

Then (3.13) is satisfied, and Theorem 13 is also valid for this preconditioned gradient method.

6.3. Conjugate Gradient method. Choose Q-orthogonal directions, i.e.

$$(d_l, Qd_k)_X \quad = \quad 0 \quad \text{for all} l > k = 0, 1, \ldots.$$

Two variants:

Algorithm 3 (Conjugate Gradient method).

(1) *Choose $x^{(0)} \in X$ and set $g_0 = F'(x^{(0)}), d_0 = -g_0$.*
(2) *For $k = 0, 1, \ldots$:*
   (a) *choose step size $\rho_k > 0$,*
   (b) *Set $x^{(k+1)} = x^{(k)} + \rho_k d_k$,*
   (c) *determine search direction: Set*

$$\begin{array}{rcl} g_{k+1} & = & -F'(x^{(k+1)}), \\ \beta_{k+1} & = & \begin{cases} \dfrac{\|g_{k+1}\|^2}{\|g_k\|^2} & \textbf{(Fletcher-Reeves)} \\ \dfrac{(g_{k+1}, g_{k+1} - g_k)_X}{\|g_k\|^2} & \textbf{(Polak-Ribière)} \end{cases} \\ d_{k+1} & = & -g_{k+1} + \beta_{k+1} d_k. \end{array}$$

until a stopping criterion $K(x^{(k+1)}, x^{(k)}) \leq \varepsilon$ is satisfied.

Convergence result:

Theorem 12. *Under the assumptions of Theorem 13 the directions d_k defined in Algorithm 3 satisfy (3.13). The results of Theorem 13 are valid for both variants of the conjugate gradient method.*

Proof. See [**Kun**, Chapter 3 Theorem 3.5]. □

6.4. Newton's method. Newton's method (Alg. 1) can be used to solve (3.12), and results in a descent method with $d_k = -F''(x^{(k)})^{-1} F'(x^{(k)})$ in Alg. 2:

Algorithm 4 (Newton's method).

(1) *Choose $x^{(0)} \in X$ and set $k = 0$.*
(2) *For $k = 0, 1, \ldots$:*
   (a) *Compute d_k from $F''(x^{(k)})d_k = -F'(x^{(k)})$*
   (b) *Set $x^{(k+1)} = x^{(k)} + d_k$*

until a stopping criterion $K(x^{(k+1)}, x^{(k)}) \leq \varepsilon$ *is satisfied.*

Convergence can be deduced from Theorem 1.

Positiv-Definiteness of $F''(x^{(k)})$ during the iteration can be enforced by taking

2.(a) *For* $\mu > 0$ *determine* δ_k *such that* $F''(x) + \delta_k I \geq \mu I$.
Solve $(F''(x^{(k)}) + \delta_k I)d_k = -F'(x^{(k)})$.

Convergence result:

THEOREM 13. *Let* X *be a Hilbert space,* $x^{(0)} \in X$*, and let*

- F *be bounded from below*
- F *be twice Fréchet differentiable in* X *with*

$$(F''(x)v, v)_X \quad \leq \quad M\|v\|^2 \quad \text{for all } x \in S, v \in X$$

for some $M > 0$*,* S *the closed convex hull of* $\{x \in X : F(x) < F(x^{(0)})\}$*,*
- *the directions* d_k *satisfy (3.13).*

Then the sequence $\{x^{(k)}\}$ *generated by Algorithm 2 is well-defined, decreasing and satisfies* $F'(x^{(k)}) \to 0$.

PROOF. See [**Kun**, Chapter 3 Theorem 5.2]. □

Inexact Newton: in [**Deu04**, Abschnitt 8.3].

6.5. Quasi-Newton method. Approximation of the Hessian by

$$\begin{aligned} H_0 &\approx F''(x^{(0)}) \\ H_{k+1} &= H_k + U_k \end{aligned}$$

with U_k rank 1 or 2-operator, consisting of one or two terms of the form

$$\begin{aligned} u \otimes w &: X \to X, \quad u, w \in X \\ (u \otimes w)s &:= (u, s)w, \quad s \in X. \end{aligned}$$

Parametera $u, w \in X$ and weights are chosen such that the secant or quasi-Newton equation

$$H_{k+1}s_k = y_k, \qquad s_k := x^{(k+1)} - x^{(k)}, y_k := F'(x^{(k+1)}) - F(x^{(k)})$$

is valid. Two most popular choices:

$$\textbf{(BFGS)} \quad U_k = \frac{y_k \otimes y_k}{(y_k, s_k)} - \frac{H_k s_k \otimes H_k s_k}{(s_k, H_k s_k)}$$

$$\textbf{(DFP)} \quad U_k = \frac{y_k \otimes (y_k - H_k s_k) + (y_k - H_k s_k) \otimes y_k}{(y_k, s_k)} - \frac{(y_k - H_k s_k, s_k)}{(s_k, H_k s_k)} y_k \otimes y_k.$$

BFGS = Broyden, Fletcher, Goldfarb, Shanno (1970), DFP = Davidon, Fletcher, Powell (1959-63).

ALGORITHM 5 (Quasi-Newton method).

(1) *Chose* $x^{(0)} \in X$ *and* $H_0 \approx F''(x^{(0)})$.
(2) *For* $k = 0, 1, \ldots$*:*
  (a) *Solve* $H_k d_k = -F'(x^{(k)})$.
  (b) *Choose* $\rho_k > 0$.
  (c) *Set* $x^{(k+1)} = x^{(k)} + \rho_k d_k$.
  (d) *Update* $H_{k+1} = H_k + U_k$

until a stopping criterion $K(x^{(k+1)}, x^{(k)}) \leq \varepsilon$ *is satisfied.*

Also Update formulas for H_k^{-1} or Cholesky factor of H_k. It can be shown that the iterates H_k remain positive definite.

6.6. Variants of the gradient method for constrained problems. Choosing *penalty* function $P : X \to \mathbb{R}$ with $P \geq 0$ and $P = 0$ in U and $(p_k)_k \subset \mathbb{R}^+$ a constrained problem can be transformed to an unconstrained one:

$$F_p(x) \quad = \quad F(x) + p_k P(x).$$

Three variants of gradient method in Alg. 2 based on projection $P_{X_{ad}}$ in X onto X_{ad} for X_{ad} convex, bounded, and closed.

- **Point-wise projected Gradient method**:
  2.(a) $d_k = -F'(x^{(k)})$
  (b) *choose* ρ_k
  (c) $x^{(k+1)} = P_{X_{ad}}(x^{(k)} + \rho_k d_k)$.
- **Projected Gradient method**
  2.(a) $d_k = -F'(x^{(k)})$.
  (b) $\rho_k = \arg\min\limits_{\rho \in (0,1]} F(P_{X_{ad}}(x^{(k)} + \rho d_k))$.
  (c) $x^{(k+1)} = x^{(k)} + \rho_k d_k$.
- (Bedingtes) **Gradient method** is based on (3.7).
  2.(a) *Compute* $\bar{x} = \arg\min\limits_{d \in X_{ad}} F'(x^{(k)}) d_k$,
  set $d_k = \bar{x} - x^{(k)}$,
  (b) $\rho_k = \arg\min\limits_{\rho \in (0,1]} F(x^{(k)} + \rho d_k)$
  (c) $x^{(k+1)} = P_{X_{ad}}(x^{(k)} + \rho_k d_k)$.

6.7. SQP **method.** Application of Newton's method for Lagrangian optimality system (3.11) und (**??**), gives SQP. To solve $L'(x, \lambda) = 0$ in every Newton step the system

$$L''(x^{(k)}, \lambda^{(k)}) d_k \quad = \quad -L'(x^{(k)}, \lambda^{(k)}) \tag{3.16}$$

is solved for $d_k = (\tilde{x}, \tilde{\lambda}) \in X \times Z^*$. We have $L'(x^{(k)}, \lambda^{(k)}) : X \times Z^* \to \mathbb{R}$ and (3.16) is an equation in $(X \times Z^*)^* \cong X^* \times Z$:

$$\begin{pmatrix} L_{xx}(x,\lambda) & L_{\lambda x}(x,\lambda) \\ L_{x\lambda}(x,\lambda) & L_{\lambda\lambda}(x,\lambda) \end{pmatrix} \begin{pmatrix} \tilde{x} \\ \tilde{\lambda} \end{pmatrix} \quad = \quad - \begin{pmatrix} L_x(x,\lambda) \\ L_\lambda(x,\lambda) \end{pmatrix} \tag{3.17}$$

with $(x, \lambda) = (x^{(k)}, \lambda^{(k)})$. Second derivatives of L:

$$\begin{aligned} L_{xx}(x,\lambda)[v_1, v_2] &= F''(x)[v_1, v_2] + \langle \lambda, e''(x)[v_1, v_2] \rangle_{Z^*, Z} \\ &= F''(x)[v_1, v_2] + \langle e''(x)^* \lambda, [v_1, v_2] \rangle_{(X \times X)^*, X \times X} \\ L_{x\lambda}(x,\lambda)[v, \mu] = L_{\lambda x}(x,\lambda)[\mu, v] &= \langle \mu, e'(x) v \rangle_{Z^*, Z} = \langle e'(x)^* \mu, v \rangle_{X^*, X} \\ L_{\lambda\lambda}(x,\lambda) &= 0 \end{aligned}$$

for $v, v_1, v_2 \in X, \mu \in Z^*$. Thus (3.17) reads

$$\begin{pmatrix} F''(x) + e''(x)^* \lambda & e'(x)^* \\ e'(x) & 0 \end{pmatrix} \begin{pmatrix} \tilde{x} \\ \tilde{\lambda} \end{pmatrix} \quad = \quad - \begin{pmatrix} F'(x) + e'(x)^* \lambda \\ e(x) \end{pmatrix} \text{ in } X^* \times Z \tag{3.18}$$

The method thus can be written as

ALGORITHM 6 (Newton-SQP method).

(1) *choose* $(x^{(0)}, \lambda^{(0)}) \in X \times Z^*$.
(2) *For* $k = 0, 1, \ldots$*:*
  (a) *Solve (3.18).*
  (b) *Update* $(x^{(k+1)}, \lambda^{(k+1)}) = (x^{(k)}, \lambda^{(k)}) + (\tilde{x}, \tilde{\lambda})$.
  until a stopping criterion $K(x^{(k+1)}, x^{(k)}) \leq \varepsilon$ *is satisfied.*

Solution of (3.18) can be splittet:

2. (a) *Solve the linearized equation* $e'(x^{(k)})\tilde{x} = e(x^{(k)})$.

(b) *Update* $x^{(k+1)} = x^{(k)} + \tilde{x}$.

(c) *Solve adjoint equation*

$$e'(x^{(k+1)})^*\tilde{\lambda} \quad = \quad -\left(F'(x) + e'(x)^*\lambda^{(k)} + F''(x^{(k+1)}) + e''(x^{(k+1)})^*\lambda^{(k)}\right)$$

(d) *Update* $\lambda^{(k+1)} = \lambda^{(k)} + \tilde{\lambda}$.

Notion SQP: (3.18) is optimalty system of the linear-quadratic problem

$$\begin{aligned} \min_{x\in X} Q_k(x) \quad &:= \quad F'(x^{(k)})(x - x^{(k)}) + \frac{1}{2}\left(F''(x^{(k)}) + e''(x^{(k)})^*\lambda^{(k)}\right)[x - x^{(k)}, x - x^{(k)}] \\ \text{bei} \qquad & e'(x^{(k)})(x - x^{(k)}) + e(x^{(k)}) = 0 \end{aligned}$$

with Lagrangian

$$L^{(k)}(x, \lambda) \quad = \quad Q_k(x) + \langle \lambda, e'(x^{(k)})(x - x^{(k)}) + e(x^{(k)})\rangle_{Z^*,Z}$$

with

$$\begin{aligned} L_x^{(k)}(x, \lambda)v \quad &= \quad F'(x^{(k)})v + \left(F''(x^{(k)}) + e''(x^{(k)})^*\lambda^{(k)}\right)[x - x^{(k)}, v] + \langle \lambda, e'(x^{(k)})v\rangle_{Z^*,Z} \\ L_\lambda^{(k)}(x, \lambda)\mu \quad &= \quad \langle \mu, e'(x^{(k)})(x - x^{(k)}) + e(x^{(k)})\rangle_{Z^*,Z}. \end{aligned}$$

Now $L^{(k)'}(x, \lambda) = 0$ gives (3.18). Convergence finite-dimensional case in [**Alt02**, Kap. 8].

6.8. Line search. To given iterate $x = x^{(k)} \in X_{ad}$ and direction $d = d_k \in X$ the solution ρ^* of the one-dimensional problem

$$\min_{\rho>0} F(x + \rho d)$$

is determined.

Quadratic case: exact solution.

Approximations:

- Sequence $\{\rho^{(j)}\}_{j=0,1,\ldots}$ is generated and $F(x+\rho^{(j)}d)$ evaluated. Stop if F has sufficiently decreased.
  - *Armijo*: $\rho^{(j)} = \beta^j$ with $\beta \in (0, 1)$. Theoretical results: [**Alt02**, Abschnitt 4.5.2-3].
  - $\rho^{(j)} = 2^{-j}$
  - *Golden mean*, s. [**Lue08**, Chapter 7].
- Interpolation of $\Phi(\rho) = F(x + \rho d)$ and computation of the excat minimum of teh interpolant. In the case of Hermite-Interpolation derivative information can be used. Example in MATLAB with quadratic or cubic interpolation, s. [**Mat00**, S. 2-12ff].

CHAPTER 4

Ways to compute derivatives

We recall two examples from the Section section:notions to demonstrate the ways to actually compute derivatives. At first we consider the Rosenbrock function from Example 1, namely

$$\begin{aligned} F: \quad & \mathbb{R}^2 \to \mathbb{R} \\ F(x) \quad &= \quad 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \end{aligned}$$

The second involves the solution of a ordinary differential equation as in Example 2.1, where the function $F : \mathbb{R}^n \to Y$ was defined as

$$F(\alpha) \quad = \quad y,$$

with y being the solution of the IVP

$$\begin{aligned} \dot{y} \quad &= \quad f(t, y), \quad t \in (0, T) \\ y(0) \quad &= \quad \alpha. \end{aligned} \tag{4.1}$$

The following options are available when we actually want to compute the derivative of a function F (not only show differentiability), i.e. we want to evaluate the derivative value at arbitrarily chosen points on a computer.

1. Analytical differentiation

The best way of course is to differentiate analytically, on a piece of paper so to speak. For a function that is explictly given as in Example 1 this is straight forward. The resulting derivative, i.e. the gradient, is given by

$$F'(x) \quad = \quad \begin{bmatrix} -400(x_2 - x_1^2)x_1 - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{bmatrix}. \tag{4.2}$$

It can be implemented in a small program and evaluated for any given x.

For the second example, which is formulated in an infinite-dimensional setting (Y is a function space!), finding an analytical expression is not that simple. Since we want to evaluate the derivative on a computer we have to discretize the problem anyway. Our choice is if we

- *first discretize* and *then differentiate*
- or if we vice versa *first differentiate* and *then discretize*.

1.1. First discretize then differentiate. The IVP (4.1) can be approximately solved by any ODE solver, giving a solution $\tilde{y}$. For simplicity we write down the forward Euler method, which gives

Algorithm 7 (Forward Euler method for ODE).

(1) *Set* $t_0 = 0, \tilde{y}_0 = \alpha$.
(2) *For* $i = 0, \ldots$:
   (a) *Choose stepsize* $h_i > 0$,
   (b) *compute* $\tilde{y}_{i+1} = \tilde{y}_i + h_i f(t_i, \tilde{y}_i)$,
   (c) *set* $t_{i+1} = t_i + h_i$,
   until $t_i \geq T$.

Let us assume that the last step is taken such that we exactly reach $t_K = T$ for some $K \in \mathbb{N}$. This algorithm realizes a function

$$\begin{aligned} \tilde{F}: \quad & \mathbb{R}^n \to \mathbb{R}^{K\times n}, \\ \tilde{F}(\alpha) \;&=\; \tilde{y}. \end{aligned} \tag{4.3}$$

The above algorithm can be differentiated with respect to α. Only for reasons of simplicity of the representation we assume here that $\alpha \in \mathbb{R}$, i.e. that $n = 1$. The general case can be treated in a similar way with each $\tilde{y}_i$ being a vector. We obtain

Algorithm 8 (Differentiated forward Euler method).

(1) *Set* $t_0 = 0, \tilde{y}'_0 = 1$.
(2) *For* $i = 0, \ldots$*:*
  (a) *Choose stepsize* $h_i > 0$,
  (b) *compute* $\tilde{y}'_{i+1} = y'_i + h_i \dfrac{\partial f}{\partial y}(t_i, \tilde{y}_i)\tilde{y}'_i$,
  (c) *set* $t_{i+1} = t_i + h_i$,
  until $t_i \geq T$.

Note that this is only correct if the step sizes h_i are chosen independently of the $\tilde{y}'_i$. An adaptive step size control for example would introduce additional derivatives h'_i, t'_i. Thus also the differentiation of step 2(b) would include additional terms.

In Alg. 8 an analytical and thus exact differentiation of Alg. 7 was performed (in fact only a differentation of step 2(b) was necessary, besides the initialization). Thus we have obtained an analytical representation of the derivative $\tilde{F}'(\alpha)$ of the function (4.3).

1.2. First differentiate then discretize - Sensitivity equation approach. Since all quantities in Alg. 7 are finite-dimensional the method presented in the last section boils down to differentiation in $\mathbb{R}^n$. Changing the order of the differentiation and discretization processes this significantly changes. Now differentiation has to be performed for $F : \mathbb{R}^n \to Y$ with the altter being a infinite-dimensional function space. In the even more complex examples where the independent variable x is infinite-dimensional as well, the differentiation is performed for $F : X \to Y$ with both spaces being infinite-dimensional.

In our example (4.1) we define

$$y' \;=\; y'(t) \quad := \quad \frac{dy}{d\alpha}(t),$$

i.e. the prime denotes the derivative w.r.t α. Formally differentiating (4.1) w.r.t α we get another IVP for the derivative y':

$$\begin{aligned} (\dot{y'}) \;=\; \frac{d(y')}{dt} \;&=\; \frac{\partial f}{\partial y}(t,y)y', \quad t \in (0,T) \\ y'(0) \;&=\; I_Y \quad (=\text{ the identity in } Y). \end{aligned} \tag{4.4}$$

To justify this formal computation we have to study this IVP, check the properties of its solution (if there is one), and show that the two limit processes of differentiating with respect to α and to t commute.

If we assume this has been done, we may then solve this IVP with the forward Euler method, ending up again with Alg. 8.

In this case both approaches are equivalent. Note that any adjustment of the time integrator for the IVP (4.4) with its changed (now linear) righthand side $\dfrac{\partial f}{\partial y}(t,y)y'$, including step-size control, will affect and most likely destroy this equivalence.

Since we derived an equation (here an IVP) for the derivative (or sensitivity y' directly, this approach is also called sensitivity approach. It can be analogously performed for partial differential equations.

We study furthermore the case of an optimal control problem. Here we have a constrained optimization problem

$$\min_{x\in X} F(x) := f(w,x) \quad \text{subject to} \quad e(w,x) = 0 \tag{4.5}$$

with

$$\begin{aligned} f: &\quad W \times X \to \mathbb{R}, \\ e: &\quad W \times X \to Z \end{aligned}$$

and function spaces W, X, Z. The derivative of F w.r.t x can be computed by the chain rule as

$$F'(x) = \frac{\partial f}{\partial w}(w(x),x)w'(x) + \frac{\partial f}{\partial x}(w(x),x).$$

The unknown derivative $w'(x)$ can be characterized using the sensitivity approach from the equation

$$\frac{d}{dx}e(w(x),x) = \frac{\partial e}{\partial w}(w(x),x)w'(x) + \frac{\partial e}{\partial x}(w(x),x) = 0,$$

i.e. (omitting the argument x of w, w'):

$$\frac{\partial e}{\partial w}(w,x)w' = -\frac{\partial e}{\partial x}(w,x).$$

Again the differentaibility and solvability of these equations have to be proved to justify these formal computations.

1.3. First differentiate then discretize - Adjoint equation approach. When studying an optimal control problem as (4.5) another method is widely used, avoiding the calculation of the derivative of the state w w.r.t. to x. Readers familiar with optimal control problems may notice (and apologize for this) that we do not use the – in this community – common notations y, u, λ for state, control, and adjoint, but w, x, and z, respectively.

We introduce the Lagrangian

$$L : W \times X \times Z^* \to \mathbb{R}, \qquad L(w,x,z) := f(w,x) + \langle z, e(w,x)\rangle_{Z^*,Z}$$

where $\langle\cdot,\cdot\rangle_{Z^*,Z}$ denotes the pairing between Z and its dual Z^*. If – for example in the case of a PDE-constrained problem with boundary conditions – the function e is vector-valued, we have $Z = Z_1 \times \ldots \times Z_s$. Thus Z^* is isometrically isomorph to the Cartesian product of the duals Z_i^*, i.e. $Z^* \cong Z_1^* \times \ldots \times Z_s^*$, and the dual pairing is given as

$$\langle z, e(w,x)\rangle_{Z^*,Z} = \sum_{i=1}^{s} \langle z_i, e_i(w,x)\rangle_{Z_i^*,Z_i}.$$

The necessary optimality condition for saddle point of L and a minimum of (**??**) are then computed by setting the directional derivatives of L with respect to (w,x,z) equal to zero in all admissible directions. We get

$$\left\{\begin{array}{lllll} \frac{\partial L}{\partial w}(w,x,z)\bar{w} &=& \frac{\partial f}{\partial w}(w,x)\bar{w} + \langle z, \frac{\partial e}{\partial w}(w,x)\bar{w}\rangle_{Z^*,Z} &=& 0 \quad \forall \bar{w} \in Y \\ \frac{\partial L}{\partial x}(w,x,z)\bar{x} &=& \frac{\partial f}{\partial x}(w,x)\bar{x} + \langle z, \frac{\partial e}{\partial x}(w,x)\bar{x}\rangle_{Z^*,Z} &=& 0 \quad \forall \bar{x} \in X \\ \frac{\partial L}{\partial z}(w,x,z)\bar{z} &=& \langle \bar{z}, e(w,x)\rangle_{Z^*,Z} &=& 0 \quad \forall \bar{z} \in Z^*. \end{array}\right. \tag{4.6}$$

The first equation is called the adjoint equation, the second gives a relation between Lagrange multiplier z and the control u, and the third one is just the state equation (**??**). The adjoint may also be written as

$$\frac{\partial e}{\partial w}(w,x)^* z \quad = \quad -\frac{\partial f}{\partial w}(w,x) \quad \text{in } Y^*,$$

where A^* denotes the adjoint operator of A. The optimality system (4.6) may be solved directly in the so-called *one-shot* approach. Another alternative is to use a gradient-based iterative algorithm to solve (4.5). Then the directional derivative of f is needed. By the chain rule it is given as

$$f'(x)\bar{x} \quad = \quad \frac{\partial f}{\partial w}(w,x)w'(x)\bar{x} + \frac{\partial f}{\partial x}(w,x)\bar{x}.$$

In a similar way the total derivative of the constraint $e(w,x) = 0$ with respect to x is

$$\frac{\partial e}{\partial w}(w,x)w'(x)\bar{x} + \frac{\partial e}{\partial x}(w,x)\bar{x} \quad = \quad 0 \quad \forall \bar{x} \in X.$$

Adding the duality pairing of this expression with the Lagrange multiplier z gives

$$\begin{aligned} f'(x)\bar{x} \quad = \quad & \frac{\partial f}{\partial w}(w,x)w'(x)\bar{x} + \langle z, \frac{\partial e}{\partial w}(w,x)w'(x)\bar{x}\rangle_{Z^*,Z} \\ & + \frac{\partial f}{\partial x}(w,x)\bar{x} + \langle z, \frac{\partial e}{\partial x}(w,x)\bar{x}\rangle_{Z^*,Z}. \end{aligned}$$

The first two terms equal zero due to the adjoint equation (with $\bar{y} = y'(u)\bar{u}$). Thus f' can be characterized without explicitly knowing $y'(u)$. After solution of both state and adjoint equation

$$f'(x)\bar{x} \quad = \quad \frac{\partial f}{\partial x}(w,x)\bar{x} + \langle z, \frac{\partial e}{\partial x}(w,x)\bar{x}\rangle_{Z^*,Z} \tag{4.7}$$

can be evaluated and used in an iterative optimization algorithm.

2. Numerical differentation

Numerical differentiation is a way to approximate the derivatives using finite differences, i.e. to compute the difference quotient

$$\frac{F(x+h) - F(x)}{h},$$

which in the limit $h \to 0$ tends to $F'(x)$, with positive h. The motivation comes directly from the definition of $F'(x)$, and the technique can be used without any knowledge about F, from the programming point of view even without having nothing more than the executable code evaluating $F(x)$. No analytic knowledge and no source code is necessary - at the first glimpse. For these reasons finite difference derivatives are widely used in applied maths. Their main deficiencies are their intrinsic approximation error and the upcoming numerical instabilities when subtracting two numbers with nearly the same value, the phenomenon known in numerical mathematics as cancellation. FD approximations are nevertheless useful to test analytically or algorithmically generated derivatives.

Thus in this section we briefly summarize the most important FD approximations for the first and second derivatives, their accuracy with respect to h. Moreover we give some hints on the numerical instabilities, and we finally propose a strategy to use FD derivatives as test for derivatives obtained by other methods.

3. Symbolic differentiation

4. Algorithmic differentation – the basic idea

We consider a function

$$\begin{aligned} F : \mathbb{R}^n &\to \mathbb{R}^m \\ F : x &\mapsto y. \end{aligned}$$

It can be written as a concatenation of k functions

$$F = F_k \circ \cdots \circ F_1 \tag{4.8}$$

with

$$F_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}, \quad n_0 = n, n_k = m. \tag{4.9}$$

We define the intermediate variables

$$\begin{aligned} x_0 &:= x \\ x_i &:= F_i(x_{i-1}), \qquad i = 1, \ldots, k \\ y = F(x) &= x_k. \end{aligned} \tag{4.10}$$

Denoting the components of the intermediates by $x_{ij}, j = 1, \ldots, n_i$, and those of F_i by F_{ij} we obtain

$$x_i = \begin{bmatrix} x_{i1} \\ \vdots \\ x_{in_i} \end{bmatrix} = F_i(x_{i-1}) = \begin{bmatrix} F_{i1}(x_{i-1}) \\ \vdots \\ F_{in_i}(x_{i-1}) \end{bmatrix}.$$

If some component $x_{i-1,j}$ of an intermediate value is not used in the following F_i, but later on, e.g. in F_{i+1}, it shall be nevertheless put as an input for F_i and it just passed trough. This assumption guarantees that (4.10) remains valid for arbitrary functions.

Example 11. *Consider the Rosenbrock function from Example 1:*

$$\begin{aligned} F : &\quad \mathbb{R}^2 \to \mathbb{R} \\ F(\xi) &= 100(\xi_2 - \xi_1^2)^2 + (1 - \xi_1)^2, \end{aligned}$$

Setting $x_0 = [\xi_1, \xi_2]^T$ we may split the evaluation of $F(x)$ into the following steps.

$$\begin{aligned}
x_1 &= F_1(x_0) = \begin{bmatrix} x_{01} \\ x_{02} \\ x_{01}^2 \end{bmatrix} = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_1^2 \end{bmatrix} =: \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \end{bmatrix} \\
x_2 &= F_2(x_1) = \begin{bmatrix} x_{11} \\ x_{12} - x_{13} \end{bmatrix} = \begin{bmatrix} \xi_1 \\ \xi_2 - \xi_1^2 \end{bmatrix} =: \begin{bmatrix} x_{21} \\ x_{22} \end{bmatrix} \\
x_3 &= F_3(x_2) = \begin{bmatrix} x_{21} \\ x_{22}^2 \end{bmatrix} = \begin{bmatrix} \xi_1 \\ (\xi_2 - \xi_1^2)^2 \end{bmatrix} =: \begin{bmatrix} x_{31} \\ x_{32} \end{bmatrix} \\
x_4 &= F_4(x_3) = \begin{bmatrix} x_{31} \\ 100 x_{32} \end{bmatrix} = \begin{bmatrix} \xi_1 \\ 100(\xi_2 - \xi_1^2)^2 \end{bmatrix} =: \begin{bmatrix} x_{41} \\ x_{42} \end{bmatrix} \\
x_5 &= F_5(x_4) = \begin{bmatrix} 1 - x_{41} \\ x_{42} \end{bmatrix} = \begin{bmatrix} 1 - \xi_1 \\ 100(\xi_2 - \xi_1^2)^2 \end{bmatrix} =: \begin{bmatrix} x_{51} \\ x_{52} \end{bmatrix} \\
x_6 &= F_6(x_5) = \begin{bmatrix} x_{51}^2 \\ x_{52} \end{bmatrix} = \begin{bmatrix} (1 - \xi_1)^2 \\ 100(\xi_2 - \xi_1^2)^2 \end{bmatrix} =: \begin{bmatrix} x_{61} \\ x_{62} \end{bmatrix} \\
x_7 &= F_7(x_6) = x_{61} + x_{62} = F(x).
\end{aligned}$$

Computation of the derivative of F in x can be now performed in a straightforward way using the chain rule:

$$F'(x) \quad = \quad F'_k(F_{k-1}(\cdots(F_1(x)))\ \cdots\ F'_2(F_1(x)) \cdot F'_1(x),$$

or (using the x_i) as:

$$F'(x) \quad = \quad F'_k(x_{k-1}) \cdots F'_2(x_1) \cdot F'_1(x_0) \ = \ \prod_{i=1}^{k} F'_i(x_{i-1}). \tag{4.11}$$

Computation of the F'_i is easy. Defining the intermediate values for the derivatives,

$$x'_i := F'_i(x_{i-1}) \cdots F'_1(x_0) = \prod_{j=1}^{i} F'_j(x_{j-1}) = \frac{d(F_i \circ \cdots \circ F_1)}{dx_0}(x) = \frac{dx_i}{dx_0}, \tag{4.12}$$

lead to the recursion formula

$$x'_i \quad = \quad F'_i(x_{i-1})x'_{i-1}, \quad i = 1, \ldots, k. \tag{4.13}$$

Due to (4.9) the intermediates x'_i now are matrices in $\mathbb{R}^{n_i \times n_i}$ with components x'_{ijk}. By (4.11) the final result is

$$x'_k \quad = \quad F'(x)x'_0 \ = \ F'(x) \tag{4.14}$$

since the starting point for the recursion is the identity matrix

$$x'_0 = \frac{dx_0}{dx_0}.$$

Consider again the example above.

Example 12. *For the Rosenbrock function from Example 11 we obtain, setting $x'_0 = I_2$, that*

$$x'_1 \quad = \quad F'_1(x_0)x'_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 2x_{01} & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 2\xi_1 & 0 \end{bmatrix}$$

with $x_{01} = \xi_1$,

$$x'_2 \quad = \quad F'_2(x_1)x'_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 2\xi_1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -2\xi_1 & 1 \end{bmatrix}$$

$$x'_3 \quad = \quad F'_3(x_2)x'_2 = \begin{bmatrix} 1 & 0 \\ 0 & 2x_{22} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -2\xi_1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -4\xi_1(\xi_2 - \xi_1^2) & 2(\xi_2 - \xi_1^2) \end{bmatrix}$$

with $x_{22} = \xi_2 - \xi_1^2$,

$$x'_4 \quad = \quad F'_4(x_3)x'_3 = \begin{bmatrix} 1 & 0 \\ 0 & 100 \end{bmatrix} x'_3 = \begin{bmatrix} 1 & 0 \\ -400\xi_1(\xi_2 - \xi_1^2) & 200(\xi_2 - \xi_1^2) \end{bmatrix}$$

$$x'_5 \quad = \quad F'_5(x_4)x'_5 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} x'_4 = \begin{bmatrix} -1 & 0 \\ -400\xi_1(\xi_2 - \xi_1^2) & 200(\xi_2 - \xi_1^2) \end{bmatrix}$$

$$x'_6 \quad = \quad F'_6(x_5)x'_5 = \begin{bmatrix} 2x_{51} & 0 \\ 0 & 1 \end{bmatrix} x'_5 = \begin{bmatrix} -2(1 - \xi_1) & 0 \\ -400\xi_1(\xi_2 - \xi_1^2) & 200(\xi_2 - \xi_1^2) \end{bmatrix}$$

with $x_{51} = 1 - \xi_1$, and finally

$$x'_7 = F'_7(x_6)x'_6 = \begin{bmatrix} 1, & 1 \end{bmatrix} x'_3 = \begin{bmatrix} -2(1 - \xi_1) - 400\xi_1(\xi_2 - \xi_1^2), & 200(\xi_2 - \xi_1^2) \end{bmatrix}$$

This the gradient of F, compare (4.2).

Allowing for the use of a more arbitrarily defined starting point x_0' of the iteration we can compute also partial or directional derivatives in the same recursion (4.13). Then we get from (4.14) for example

$$\begin{aligned} x_0' := e_j \in \mathbb{R}^n, \text{ the } j\text{-th unit vector} \quad &\Longrightarrow \quad x_k' = F'(x)e_j = \frac{\partial F}{\partial x_j}(x), \\ &\qquad \text{i.e. the } j\text{-th partial derivative,} \\ x_0' := v \in \mathbb{R}^n \quad &\Longrightarrow \quad x_k' = F'(x)v, \\ &\qquad \text{i.e. a directional derivative,} \end{aligned}$$

and so on. Thus the choice of x_0' determines the meaning of x_k' and the derivative that is computed by (4.13). Since x_0' initiates the recursion it is called *Seed Matrix*.

If the function is split up as in (4.10) and the derivative statements F_i' can be generated algorithmically from the F_i statements, then we may combine (4.10) with (4.13) and obtain:

$$\begin{array}{rcrcll} & & x_0 & = & x & \\ & & x_0' & = & I_n & \\ (4.15) & & x_i' & = & F_i'(x_{i-1})x_{i-1}' & \left.\begin{array}{l}\\\end{array}\right\} \quad i = 1, \ldots, k \\ & & x_i & = & F_i(x_{i-1}) & \\ y & = & F(x) & = & x_k & \\ y' & = & F'(x) & = & x_k'. & \end{array}$$

CHAPTER 5

Forward mode

The *forward mode* may be called the straightforward way to apply AD. The function F is separated into elementary functions again, and the "way from x to $y = F(x)$" is written efficiently and more compact than above, namely as a graph. Then every elementary function or operation – each line of code in the programme to evaluate $F(x)$ – is augmented by a derivative statement. This can be done algorithmically. The forward mode is thus the first step to see how AD really works.

To describe the procedure, we start by recalling the basic notions of *directed graphs* (or *digraphs, oriented graphs*):

DEFINITION 11.

(1) *A directed graph/ digraph/ oriented graph G is a pair $G = (V, E)$ where*
   - *$V \neq \emptyset$ is the set of vertices*
   - *$E \subset V \times V$ is the set of edges.*

   A pair $(u, v) \in E$ is called an edge/ arc from u to v. We will also use the notation $u \prec v$ in this case.

(2) *A sequence $(v_1, \ldots, v_k), v_i \in V$ is called path from v_1 to v_k (in G) if*
$$(v_i, v_{i+1}) \in E \quad \forall i = 1, \ldots, k-1.$$

(3) *$u \in V$ is called predecessor of $v \in V$, if there exists a path $(u, \ldots, v)$ from u to v. Vice versa, v is called successor of u in this case.*

(4) *If $(u, v) \in E$, then u is called direct predecessor of v, and v is called direct successor of u.*

(5) *The edges and/ or vertices of a graph may have additional attributes.*

1. Representation of a function as directed graph

Instead of (4.10), we now use the notion of digraphs to describe the computational evaluation of a function

$$F : \mathbb{R}^n \to \mathbb{R}^m$$

that is coded in a computer programme. Just to make it simple at this point, we will assume $m = 1$, i.e.

$$\begin{aligned} F : \mathbb{R}^n &\to \mathbb{R} \\ x &\mapsto F(x). \end{aligned}$$

The extension for $m > 1$ is straightforward. We use the notation

$$\begin{aligned} [x_1, \ldots, x_n] &= x \in \mathbb{R}^n \\ x_i &= F_i(x_1, \ldots, x_{i-1}), \qquad i = n+1, \ldots, n+k \\ y = F(x) &= x_{n+k}. \end{aligned} \tag{5.1}$$

Here again F is split up into elementary functions or operations. We define the intermediates x_i as scalars in this case. The number k of functions used is the same as in (4.10), if in the latter only one operation is performed in every F_i.

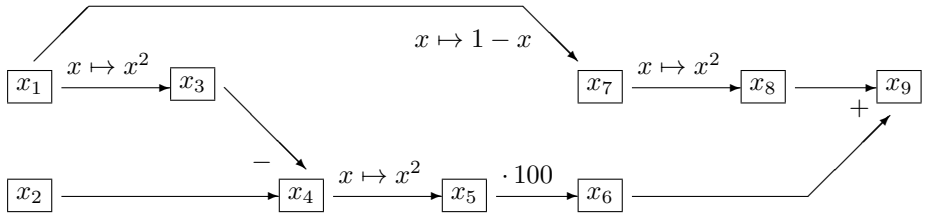

FIGURE 1. Computational graph of the Rosenbrock function.

Note that not all of the arguments $x_1, \ldots, x_{i-1}$ of F_i actually have to be used. Usually the function F_i does only depend on a subset of all input and intermediate values x_i. We now define the set of vertices as

$$V := \{x_i : i = 1, \ldots, k\} \tag{5.2}$$

and the edges by the relation

$$(x_j, x_i) \in E \iff x_j \prec x_i \quad :\iff \quad x_i \text{ depends directly on } x_j. \tag{5.3}$$

We will also use the notation

$$j \prec i \quad :\iff \quad x_j \prec x_i$$

for the indices of the vertices, such that we may write

$$x_i = F_i((x_j)_{j \prec i}), \qquad i = n+1, \ldots, n+k \tag{5.4}$$

in (5.1). If we add (5.4) as attributes to all vertices with indices $i = n+1, \ldots, n+k$, we we obtain the computational graph of F:

DEFINITION 12. *The directed graph $G = (V, E)$ with V, E defined in* (5.2) *and* (5.3)*, augmented by the attributes* (5.4)*, is called the computational graph of F.*

EXAMPLE 13. *For the Rosenbrock function from Examples 11 and 12 we have*

$$\begin{array}{lllllll}
x_3 &=& F_3(x_1) &=& x_1^2 & & \\
x_4 &=& F_4(x_2, x_3) &=& x_2 - x_3 &=& x_2 - x_1^2 \\
x_5 &=& F_5(x_4) &=& x_4^2 &=& (x_2 - x_1^2)^2 \\
x_6 &=& F_6(x_5) &=& 100 x_5 &=& 100(x_2 - x_1^2)^2 \\
x_7 &=& F_7(x_1) &=& 1 - x_1 & & \\
x_8 &=& F_8(x_7) &=& x_7^2 &=& (1 - x_1)^2 \\
x_9 &=& F_9(x_6, x_8) &=& x_6 + x_8 &=& F(x).
\end{array}$$

The computational graph of F is shown in Figure 1.

Computing the derivative with the chain rule now gives for $x_i' := \dfrac{dx_i}{dx}$:

$$x_i' = \begin{cases} e_i \text{ (the } i\text{-th unit vector)}, & i = 1, \ldots, n, \\ \dfrac{dF_i(x_1, \ldots, x_{i-1})}{d(x_1, \ldots, x_n)} = \displaystyle\sum_{j \prec i} \frac{\partial F_i((x_l)_{l \prec i})}{\partial x_j} x_j', & i = n+1, \ldots, n+k \end{cases} \tag{5.5}$$

and $F'(x) = x_k'$. Here we may regard the x_i' as row vectors, but this makes no difference. This means

$$\frac{d[x_1, \ldots, x_n]}{dx} = [x_1', \ldots, x_n'] =: x' = I_n, \tag{5.6}$$

which shall denote the identity matrix in $\mathbb{R}^{n\times n}$.

EXAMPLE 14. *Using (5.5) the derivative of Rosenbrock's function can be computed as follows.*

$$\begin{aligned}
x_1' &= [1,0]\\
x_2' &= [0,1]\\
x_3' &= F_3'(x_1)x_1' \;=\; 2x_1x_1' \;=\; [2x_1,0]\\
x_4' &= \frac{\partial F_4(x_2,x_3)}{\partial x_2}x_2' + \frac{\partial F_4(x_2,x_3)}{\partial x_3}x_3' \;=\; x_2' - x_3' \;=\; [-2x_1,1]\\
x_5' &= F_5'(x_4)x_4' \;=\; 2x_4x_4' \;=\; 2(x_2-x_1^2)[-2x_1,1]\\
x_6' &= F_6'(x_5)x_5' \;=\; 100x_5' \;=\; 200(x_2-x_1^2)[-2x_1,1]\\
x_7' &= F_7'(x_1)x_1' \;=\; -x_1' \;=\; [-1,0]\\
x_8' &= F_8'(x_7)x_7' \;=\; 2x_7x_7' \;=\; [-2(1-x_1),0]\\
x_9' &= \frac{\partial F_9(x_6,x_8)}{\partial x_6}x_6' + \frac{\partial F_9(x_6,x_8)}{\partial x_8}x_8'\\
&= x_6' + x_8' \;=\; [-400(x_2-x_1^2)x_1 - 2(1-x_1), 200(x_2-x_1^2)].
\end{aligned}$$

Conditional statements. Conditional statements as *if-else* constructs are a frequently used element in all programming languages, and they appear in nearly every computer programme. They can be still be written as an elementary function and then include this function as above in (5.4). Let us consider as instructive example the following *if-else*-statement:

$$\begin{array}{ll} if \quad x_{i-1}\geq 0: & x_i = F_{i1}(x_{i-1}),\\ else: & x_i = F_{i0}(x_{i-1}). \end{array}$$

As an easy example one could think of $F_{i1}(x_{i-1}) = x_{i-1}, F_{i0}(x_{i-1}) = -x_{i-1}$, and then the above construct would represent an implementation of the modulus function $F_i(x_{i-1}) = |x_{i-1}|$. More general we consider

$$\begin{array}{ll} if\ cond_i: & x_i = F_{i1}(x_{i-1}),\\ else: & x_i = F_{i0}(x_{i-1}). \end{array}$$

where now $cond_i$ is a boolean (logical) statement with values in $\{true, false\} \cong \{1,0\}$. In our case we have $cond_i = (x_{i-1} \geq 0)$. Nested *if-else* statements or *switch/case* statements can always be written as a sequence of simple *if-else* statements. Thus we only consider the latter here.

A conditional statement occurring in F_i can be easily written as

$$F_i(x_{i-1}) \;:=\; \left\{ \begin{array}{ll} F_{i1}(x_{i-1}), & cond_i\\ F_{i0}(x_{i-1}), & \neg\, cond_i. \end{array} \right. \tag{5.7}$$

This function can now be put in (5.4) as any other function. In the computational graph the function still can be represented by a single vertex. But this procedure hides what a conditional statement in fact does with the computational graph: It introduces a branch in the graph, and sometimes it is more instructive to explicitly show this. This is specifically true when studying the reverse mode later. So we would rather depict the graph as shown in the second picture in Fig. 2.

As a consequence, all following intermediates $x_j, j > i$ will depend on the value of $cond_i$, i.e., which branch of the graph is followed in the concrete case. Thus we may formally distinguish the values

$$x_j \;=\; \left\{ \begin{array}{ll} x_{j1}, & cond_i\\ x_{j0}, & \neg\, cond_i \end{array} \right\}, \quad j > i,$$

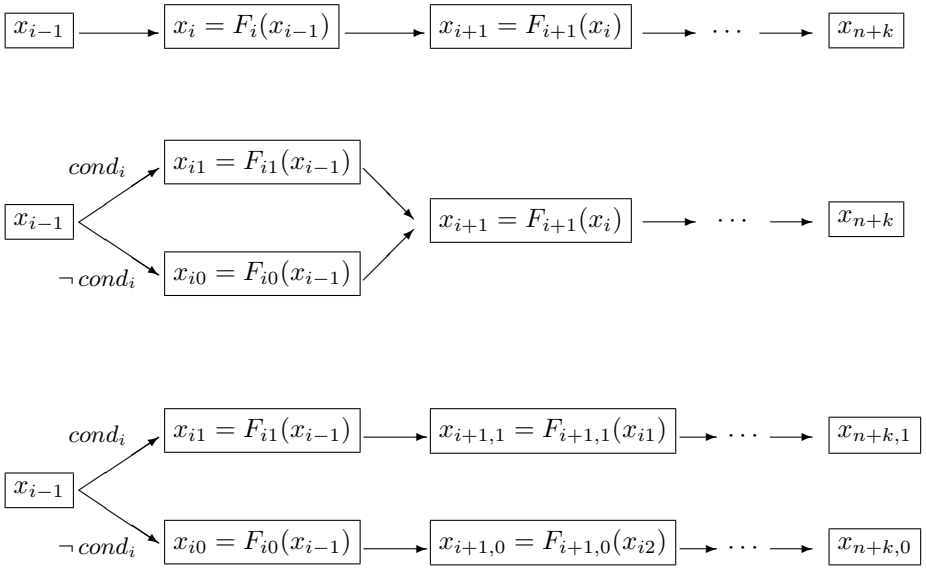

FIGURE 2. Computational graph hiding (top) and displaying (middle) a branch caused, e.g., by an *if-else* statement in F_i. The graph at the bottom additionally shows that all values $x_j, j > i$, depend on the value of the condition $cond_i$.

introducing additional indices 1 or 0 depending on the value of $cond_i$. If there are more conditional statements in the programme, the graph will include the according number of branches, and the notation becomes accordingly complex, with additional indices for every condition statement. Thus we can alos depcit the graph showing these intermediate values, as done in the bottom picture of Fig. 2.

Conditional loops, loops with dynamic termination criteria. An extension of the above case of a rather simple branch caused by an *if-else* statement is the conditional loop caused for example by a *while*, *do-while* or *repeat-until* construct with a dynamic running (or termination) criterion. When occurring in the i-th step of the programme, this can generally be written as

$$cond_i \quad = \quad cond_i((x_l)_{l<i}).$$

Again $cond_i$ is a boolean expression. Since all kind of conditional loops after considerable modification and maybe additional simple and/ or *if* statements can be written as a *while* loop we consider her the following case:

$$\begin{aligned}
&j = 0 \\
&x_{i0} = x_{i-1} \\
&while\ cond_i: \quad j = j + 1 \\
&\qquad\qquad\quad x_{ij} = F_{ij}(x_{i,j-1}) \\
&j_{max} = j \\
&x_i = x_{i,j_{max}}.
\end{aligned}$$

The loop counter j used as a second index is introduced here artificially. We use it and its final value j_{max} in the formal notation for the intermediates generated while the loop is running. As a consequence, we now still have

$$x_i \quad = \quad F_i(x_{i-1})$$

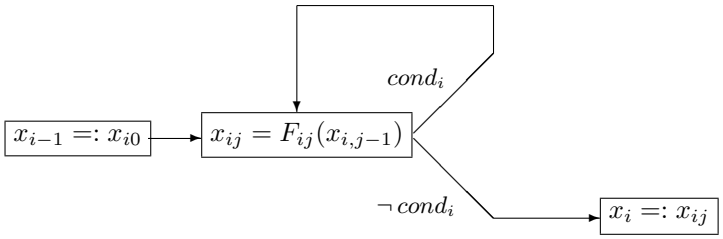

FIGURE 3. Computational graph of a *while* loop with dynamic running criterion $cond_i$. The increment of the inner loop counter j is not shown.

at the end of the loop. We now have

$$F_i \;=\; \bigcirc_{j=1}^{j_{max}} F_{ij} \;=\; F_{i,j_{max}} \circ \ldots \circ F_{i0}.$$

In the computational graph this situation can be represented by a cycle, see Fig. 3. Every loop can of course be represented by a cycle in the graph, but if the termination criterion is static this is not necessary and we will not use cycles since the notation becomes more complicated.

2. Forward mode algorithm

If the function is split up as in (5.1) and the derivative statements F_i' can be generated algorithmically from the statements of the F_i , then we may combine (5.1-5.4) with (5.5) and obtain:

$$(5.8)\quad \left\{ \begin{array}{rcl} [x_1,\ldots,x_n] & = & x \in \mathbb{R}^n \\ [x_1',\ldots,x_n'] & = & I_n \in \mathbb{R}^{n\times n} \\ x_i' & = & \displaystyle\sum_{j\prec i} \frac{\partial F_i((x_l)_{l\prec i})}{\partial x_j} x_j' \\ x_i & = & F_i((x_l)_{l\prec i}) \\ y \;=\; F(x) & = & x_{n+k} \\ y' \;=\; F'(x) & = & x_{n+k}'. \end{array} \right\} \qquad i = n+1,\ldots,n+k$$

For a vector-valued function $F = [F_l]_{l=1,\ldots,m} : \mathbb{R}^n \to \mathbb{R}^m$ the last two lines change to

$$\left. \begin{array}{rcccl} y_l & = & F_l(x) & = & x_{n+k-m+l} \\ y_l' & = & F_l'(x) & = & x_{n+k-m+l}' \end{array} \right\} \qquad l = 1,\ldots,m.$$

EXAMPLE 15. *For the Rosenbrock function we obtain:*

$$
\begin{array}{lcl|lcl|lcl}
x_1' & = & [1,0] & & & & & & \\
x_2' & = & [0,1] & & & & & & \\
x_3' & = & F_3'(x_1)x_1' & & = & 2x_1x_1' & = & [2x_1,0] \\
x_3 & = & F_3(x_1) & & = & x_1^2 & & \\
x_4' & = & \dfrac{\partial F_4(x_2,x_3)}{\partial x_2}x_2' + \dfrac{\partial F_4(x_2,x_3)}{\partial x_3}x_3' & = & x_2' - x_3' & = & [-2x_1,1] \\
x_4 & = & F_4(x_2,x_3) & & = & x_2 - x_3 & = & x_2 - x_1^2 \\
x_5' & = & F_5'(x_4)x_4' & & = & 2x_4x_4' & = & 2(x_2 - x_1^2)[-2x_1,1] \\
x_5 & = & F_5(x_4) & & = & x_4^2 & = & (x_2 - x_1^2)^2 \\
x_6' & = & F_6'(x_5)x_5' & & = & 100x_5' & = & 200(x_2 - x_1^2)[-2x_1,1] \\
x_6 & = & F_6(x_5) & & = & 100x_5 & = & 100(x_2 - x_1^2)^2 \\
x_7' & = & F_7'(x_1)x_1' & & = & -x_1' & = & [-1,0] \\
x_7 & = & F_7(x_1) & & = & 1 - x_1 & & \\
x_8' & = & F_8'(x_7)x_7' & & = & 2x_7x_7' & = & [-2(1-x_1),0] \\
x_8 & = & F_8(x_7) & & = & x_7^2 & = & (1-x_1)^2 \\
x_9' & = & \dfrac{\partial F_9(x_6,x_8)}{\partial x_6}x_6' + \dfrac{\partial F_9(x_6,x_8)}{\partial x_8}x_8' & = & x_6' + x_8' & = & [-400(x_2 - x_1^2)x_1 & & \\
 & & & & & & -2(1-x_1), 200(x_2 - x_1^2)] & & \\
 & & & & & & = F'(x) & & \\
x_9 & = & F_9(x_6,x_8) & & = & x_6 + x_8 & = & F(x) &
\end{array}
$$

3. The seed matrix

Using a differently defined starting point $x' := [x_1', \ldots, x_n']$ of the iteration we can compute also partial or directional derivatives in the same recursion (4.13) and get from (4.14) for example

$$
\begin{aligned}
x' := e_j \in \mathbb{R}^n, \text{ the } j\text{-th unit vector} \quad &\Longrightarrow \quad y' \;=\; x'_{n+k} = F'(x)e_j = \frac{\partial F}{\partial x_j}(x), \\
&\quad \text{i.e. the } j\text{-th partial derivative,} \\
x' := v \in \mathbb{R}^n \quad &\Longrightarrow \quad y' \;=\; x'_{n+k} = F'(x)v, \\
&\quad \text{i.e. a directional derivative,}
\end{aligned}
$$

and so on. Thus the choice of x' determines the meaning of x'_{n+k} and the derivative that is computed by (4.13). Since x' initiates the recursion it is called *Seed Matrix* and alos denoted by S. The consequences of different choices for the seed matrix for functions with different dimensions of domain and range can be seen in Table 1.

4. Active, passive and needed variables

To analyze the computational graph and to minimize the computational effort in AD the notion of *active, passive*, and *needed variables* is crucial. To illustrate the concept, we study a slight modification of the Rosenbrock function, namely the function

$$
F(x_1,x_2) \;:=\; (x_1 - x_1^2)^2 + \frac{100}{x_2^2}. \tag{5.9}
$$

Its computational graph is shown in Figure 4. We leave the numbering of the unknowns as in the Rosenbrock example for demonstration purposes only. Let us compute the first partial derivative by the forward mode: We proceed analogously to Example 15, but now with

$$
S = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}
$$

$F : x \mapsto y$	$F'(x) \in$	$x' \in$	$\Rightarrow \in$	$S = x' =$	$\Rightarrow y' =$	meaning of y'
$\mathbb{R} \to \mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	1 $c \in \mathbb{R}$	$F'(x)$ $cF'(x)$	derivative (scalar)
$\mathbb{R}^n \to \mathbb{R}$	$\mathbb{R}^n$	$\mathbb{R}^n$	$\mathbb{R}$	e_j	$\partial_j F(x)$	partial derivative
				v	$F'(x)v$	directional derivative
		$\mathbb{R}^{n\times s}$	$\mathbb{R}^s$	$(v_j)_{j=1}^s$	$(F'(x)v_j)_{j=1}^s$	several (s) directional derivatives
		$\mathbb{R}^{n\times n}$	$\mathbb{R}^n$	$(e_j)_{j=1}^n = I_n$	$F'(x)$	gradient
$\mathbb{R}^n \to \mathbb{R}^m$ $F =$ $(F^{(i)})_{i=1}^m$	$\mathbb{R}^{m\times n}$	$\mathbb{R}^n$	$\mathbb{R}^m$	e_j	$\partial_j F(x) =$ $(\partial_j F^{(i)}(x))_{i=1}^m$	partial derivative (column of Jacobian matrix)
				v	$F'(x)v$	directional derivative
		$\mathbb{R}^{n\times s}$	$\mathbb{R}^{m\times s}$	$(v_j)_{j=1}^s$	$(F'(x)v_j)_{j=1}^s$	several (s) directional derivatives
		$\mathbb{R}^{n\times n}$	$\mathbb{R}^{m\times n}$	$(e_j)_{j=1}^n$	$F'(x)$	full Jacobian matrix
$\mathbb{R}^{n\times p} \to \mathbb{R}$	$\mathbb{R}^{n\times p}$	$\mathbb{R}^{n\times p}$	$\mathbb{R}$	E_{ij}	$\partial_{ij} F(x)$	partial derivative
				V	$F'(x)V$	directional derivative
		$\mathbb{R}^{(n\times p)\times s}$	$\mathbb{R}^s$	$(V_l)_{l=1}^s$	$(F'(x)V_l)_{l=1}^s$	several (s) directional derivatives
		$\mathbb{R}^{(n\times p)^2}$	$\mathbb{R}^{(n\times p)^2}$	$(E_{ij})_{i,j=1}^{n,p}$	$F'(x)$	full Jacobian (tensor)

TABLE 1. Initialization of the seed matrix x' and resulting derivative objects x'_k. Here e_j denotes the j-th unit vector, and E_{ij} a matrix with all elements zero except the one in row i, column j, which equals 1.

as seed matrix and obtain

$$
\begin{array}{lcllcll}
x'_3 &=& F'_3(x_1)x'_1 & = & 2x_1x'_1 & = & 2x_1 \\
x_3 &=& F_3(x_1) & = & x_1^2 & & \\
x'_4 &=& F'_4(x_2)x'_2 & = & 0 & & \\
x_4 &=& F_4(x_2) & = & \dfrac{1}{x_2} & & \\
x'_5 &=& F'_5(x_4)x'_4 & = & 0 & & \\
x_5 &=& F_5(x_4) & = & x_4^2 & = & \dfrac{1}{x_2^2} \\
x'_6 &=& F'_6(x_5)x'_5 & = & 0 & & \\
x_6 &=& F_6(x_5) & = & 100x_5 & = & \dfrac{100}{x_2^2} \\
x'_7 &=& \dfrac{\partial F_7(x_1,x_3)}{\partial x_1}x'_1 + \dfrac{\partial F_7(x_1,x_3)}{\partial x_3}x'_3 & = & x'_1 - x'_3 & = & 1 - 2x_1 \\
x_7 &=& F_7(x_1,x_3) & = & x_1 - x_3 & = & x_1 - x_1^2 \\
x'_8 &=& F'_8(x_7)x'_7 & = & 2x_7x'_7 & = & 2(x_1 - x_1^2) \\
x_8 &=& F_8(x_7) & = & x_7^2 & = & (x_1 - x_1^2)^2 \\
x'_9 &=& \dfrac{\partial F_9(x_6,x_8)}{\partial x_6}x'_6 + \dfrac{\partial F_9(x_6,x_8)}{\partial x_8}x'_8 & = & x'_8 & = & 2(x_1 - x_1^2) \;=\; \dfrac{\partial F}{\partial x_1}(x) \\
x_9 &=& F_9(x_6,x_8) & = & x_6 + x_8 & = & F(x)
\end{array}
$$

It can be seen – and much easier from the computational graph – that the intermediate derivative values x'_4, x'_5, x'_6 are zero since x'_2 was set to zero and x_4, x_5, x_6 do only depend on x_2, not on x_1. The variables x_4, x_5, x_6 are called passive variables when computing the derivative of F with respect to x_1. The variables x_3, x_7, x_8, x_9,

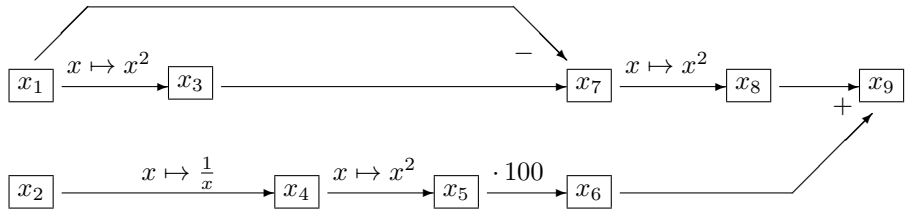

FIGURE 4. Computational graph of the function defined in (5.9).

in contrast, are called active. The input x_1 is also called active in this case. For passive variables no intermediate derivatives have to be computed. The exact definition is:

DEFINITION 13. *If we compute the derivative of a variable x_i with respect to a variable x_j, all variables on the path from x_i to x_j are called active variables. All variables not on this path are called passive variables.*

In the example above, variable x_2 is not used or needed at all to compute the partial derivative with respect to x_1.

This can be different, e.g., if the last operation F_9 is a multiplication and not an addition, i.e. if we consider

$$F(x_1, x_2) \quad := \quad 100\frac{(x_1 - x_1^2)^2}{x_2^2}.$$

Then we obtain at the end

$$\begin{array}{rclcclcl} x_9' & = & \dfrac{\partial F_9(x_6, x_8)}{\partial x_6} x_6' + \dfrac{\partial F_9(x_6, x_8)}{\partial x_8} x_8' & = & x_6 x_8' & = & 100 x_5 x_8' & = & 100 x_4^2 x_8' \\ & & & = & \dfrac{100}{x_2^2} x_8' & = & \dfrac{200(x_1 - x_1^2)}{x_2^2} & = & \dfrac{\partial F}{\partial x_1}(x) \\ x_9 & = & F_9(x_6, x_8) & = & x_6 x_8 & = & F(x). & & \end{array}$$

The variables x_4, x_5, x_6 still are not active, but they are *needed* to compute the partial derivative with respect to x_1.

DEFINITION 14. *Variables that are used to compute the derivative of a variable x_i with respect to a variable x_j, are called needed variables.*

Note that these variables become needed now because we have a function F that can no longer be split up in an additive way into

$$F(x_1, x_2) = G(x_1) + H(x_2),$$

but only as a product

$$F(x_1, x_2) = G(x_1) \cdot H(x_2).$$

By the product rule of differentiation we know that the derivative of a product $x_1 x_2$ leads to $x_1' x_2 + x_1 x_2'$, i.e., to compute one partial derivative one always needs the value of the other variable. The graphs of both functions look the same, only the attribute of the last vertex has changed.

5. Computational Effort

We now study the computational effort of the forward mode algorithm (5.8). Here we want to compare the needed effort for the evaluation of both F and F', denoted by $E(F, F')$, with the effort needed just to evaluate F, denoted by $E(F)$. The following estimates can be found in [**Gri89**]. Our aim is to obtain an estimate for the effort of the type

$$q_{F,fwd} \quad := \quad \frac{E_{fwd}(F, F')}{E(F)} \;\approx\; C(n),$$

where C is a function which may depend on the number of input variables $x_1, \ldots, x_n$. We consider here a scalar function $F : \mathbb{R}^n \to \mathbb{R}$ only, for a vector valued function $F : \mathbb{R}^n \to \mathbb{R}^m$ a rough estimate can be obtained by multiplication by m.

To estimate the effort, we denote the set of direct predecessors of a vertex x_i in the computational graph by

$$I_i := \{j : j \prec i\}$$

and its cardinality by $n_i := |I_i|$. For most elementary operations in a programming language one has $n_i \leq 2$ or $n_i \leq 3$. In any case we have $n_i < i$ since there are no more possible input values for F_i than $x_1, \ldots, x_{i-1}$.

For our estimate we assume that

(1) the computational effort is additive and
(2) the effort of each F_i is proportional to the number of inputs n_i, where the bound is independent of the F_i. This means we assume

$$E(F_i) \quad \leq \quad c\, n_i \quad \forall i \tag{5.10}$$

with $c > 0$ independent of i.

Studying the algorithm (5.1) for the evaluation of the function, we can now estimate

$$E(F) \quad = \quad \sum_{i=n+1}^{n+k} E(F_i) \quad \leq \quad c \sum_{i=n+1}^{n+k} n_i. \tag{5.11}$$

In algorithm (5.8), we have to evaluate the two lines

$$x_i' \quad = \quad \sum_{j \prec i} \frac{\partial F_i((x_l)_{l \prec i})}{\partial x_j} x_j' \tag{5.12}$$

$$x_i \quad = \quad F_i((x_l)_{l \prec i}) \tag{5.13}$$

for $i = n+1, \ldots, n+k$. In order to evaluate these two lines, we have to compute both F_i and

$$F_i'((x_l)_{l \prec i}) = \left(\frac{\partial F_i((x_l)_{l \prec i})}{\partial x_j} \right)_{j \prec i} \in \mathbb{R}^{n_i},$$

which together has an effort $E(F_i, F_i')$. To compute one term in the sum in (5.12), a component-wise multiplication of $x_j' \in \mathbb{R}^n$ with one partial derivative

$$\frac{\partial F_i((x_l)_{l \prec i})}{\partial x_j}$$

has to be computed, and this needs n multiplications. Since n_i terms are summed up, this has to be multiplied by n_i, and moreover $(n_i - 1)n$ additions are needed. From now on, we count each multiplication and addition as one operation. Thus for each i, computing (5.12) and (5.13) together needs the effort

$$\begin{aligned} E(F_i, F_i') + n(n_i - 1)E(+) + nn_iE(*) \quad &= \quad E(F_i, F_i') + n((n_i - 1) + n_i) \\ &\geq \quad E(F_i, F_i') + nn_i \end{aligned}$$

since $n_i \geq 1$. Summing up and using the estimate $E(F_i, F_i') \geq E(F_i)$, we obtain

$$\begin{aligned} E_{fwd}(F, F') \;\geq\; \sum_{i=n+1}^{n+k} \Big(E(F_i, F_i') + n n_i \Big) \;&\geq\; \sum_{i=n+1}^{n+k} \Big(E(F_i) + n n_i \Big) \\ &=\; \sum_{i=n+1}^{n+k} E(F_i) + n \sum_{i=n+1}^{n+k} n_i \\ &=\; E(F) + n \sum_{i=n+1}^{n+k} n_i \end{aligned}$$

For $q_{F,fwd}$ we compute

$$q_{F,fwd} \;=\; \frac{E_{fwd}(F, F')}{E(F)} \;\geq\; \frac{E(F) + n \sum_{i=n+1}^{n+k} n_i}{E(F)} \;=\; 1 + \frac{n \sum_{i=n+1}^{n+k} n_i}{E(F)}.$$

Using estimate (5.11) for the denominator, we get

$$q_{F,fwd} \;\geq\; 1 + \frac{n \sum_{i=n+1}^{n+k} n_i}{c \sum_{i=n+1}^{n+k} n_i} \;=\; 1 + \frac{n}{c} \;=\; \Theta(n).$$

Here we used the Landau symbol

$$f(n) = \Theta(g(n)) :\Longleftrightarrow \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} :\; c_1 g(n) \leq |f(n)| \leq c_2 g(n) \; \forall n \geq n_0.$$

Thus, the computational effort for computing both F and F' by the forward mode is at least by a factor of magnitude $\Theta(n)$ higher than the effort for just evaluating F. This means, the forward mode is – generally speaking – not faster than a finite difference computation. Nevertheless the computed derivative is exact and includes no approximation errors.

CHAPTER 6

Source transformation

If the function is split up as in (5.1) and the derivative statements F_i' can be generated algorithmically from the F_i statements, then algorithm (5.8) is rather easy to realize, compare example (16).

EXAMPLE 16. *For the Rosenbrock function from Examples 11 and 12 we have (after initialization of the seed matrix $S := [x_1', x_2']$):*

$$\begin{aligned}
x_3' &= 2x_1x_1' \\
x_3 &= x_1^2 \\
x_4' &= x_2' - x_3' \\
x_4 &= x_2 - x_3 \\
x_5' &= 2x_4x_4' \\
x_5 &= x_4^2 \\
x_6' &= 100x_5' \\
x_6 &= 100x_5 \\
x_7' &= -x_1' \\
x_7 &= 1 - x_1 \\
x_8' &= 2x_7x_7' \\
x_8 &= x_7^2 \\
x_9' &= x_6' + x_8' \\
x_9 &= x_6 + x_8.
\end{aligned}$$

The statements including the x_i' can be generated algorithmically from the original ones that include only the x_i.

Thus new source code can be written. If the original function F was given in the code as a function

```
y = F(x)
```

then a source transformation tool will generate a new function `g_F` with signature

```
[y,g_y] = g_F(x,g_x)
```

where `g_x` and `g_y` are names for the seed matrix x' and the derivative y', respectively.

Steps for a source transformation tool based on the representation (4.10):

- Identify all active variables x_i on the path from x to $y = F(x)$.
- Split up F according to (4.10).
- Generate the derivative statements according to (4.13),
- and insert them in the correct positions as in (4.15).

CHAPTER 7

Operator overloading

Operatorüberladen (engl.: *Operator Overloading*) ist die zweite Möglichkeit, die Ableitungen der elementaren Rechenoperationen der verwendeten Programmiersprache zu realisieren. Diese Technik ist ein wesentliches Merkmal objektorientierter Programmiersprachen und wurde, angeregt durch die verstärkte Verbreitung von C++, für das Algorithmische Differenzieren mit dem Tool ADOL-C zuerst verwendet. Das zweite wichtige Tool mit dieser Technik ist ADMAT für die mathematisch-technische Skriptsprache MATLAB. Durch die schlechtere Compileroptimierung für selbstdefinierte Datentypen eignet sich Operatorüberladen eher für nicht zu zeit- und speicherintensive Codes, die Realisierung von höheren Ableitungen ist jedoch ohne mehrfache Quelltransformation möglich. Insgesamt wird der Code kaum verändert. In der ADOL-C-Realisierung entsteht bei der mehrfachen Auswertung verschiedener Ableitungen (z.B. erster und zweiter) einer Funktion ein Vorteil.

Beim Operatorüberladen macht man sich die Möglichkeit moderner Programmiersprachen zunutze, die mathematischen Operatoren und Funktionen der Sprache zu *überladen*. Das bedeutet, dass man ihnen für verschiedene Datentypen auch verschiedene Bedeutungen gibt. Ein einfaches mathematisches Beispiel ist die Multiplikation, die für reelle Zahlen, komplexe Zahlen, Vektoren und Matrizen ganz unterschiedliche Operationen darstellt, vgl. Tabelle 1.

Definitionsbereich (Inputtyp)	Bedeutung der Multiplikation	Ergebnistyp
$\mathbb{R} \times \mathbb{R}$	$x \cdot y := xy$	$\mathbb{R}$
$\mathbb{C} \times \mathbb{C}$	$w \cdot z := (\mathrm{Re}(w)\mathrm{Re}(z) - \mathrm{Im}(w)\mathrm{Im}(z),$ $\mathrm{Re}(w)\mathrm{Im}(z) + \mathrm{Im}(w)\mathrm{Re}(z))$	$\mathbb{C}$
$\mathbb{R}^n \times \mathbb{R}^n$	$x \cdot y := \sum_i x_i y_i$	$\mathbb{R}$
...	...	...

TABLE 1. Verschiedene Bedeutung der Multiplikation in Abhängigkeit vom Definitionsbereich (Inputdatentyp).

$$
\begin{array}{rcll}
X+Y & = & (x,x')+(y,y') & := & (x+y, x'+y') \\
XY & = & (x,x')\cdot(y,y') & := & (xy, xy'+x'y) \\
\exp X & = & \exp(x,x') & := & (\exp x, x' \exp x) \\
\cos X & = & \cos(x,x') & := & (\cos x, -x' \sin x) \\
\dfrac{X}{Y} & = & \dfrac{(x,x')}{(y,y')} & = & \left(\dfrac{x}{y}, \dfrac{x'y - xy'}{y^2}\right)
\end{array}
$$

FIGURE 1. Überladene Operatoren und Funktionen für den Datentyp (7.1).

$$
\begin{array}{lcllcl}
X_1 & = & (x_1, x_1') & & & = & (a, [1,0,0]) \\
X_2 & = & (x_2, x_2') & & & = & (b, [0,0,0]) \\
X_3 & = & (x_3, x_3') & & & = & (c, [0,0,0]) \\
X_4 & = & F_4(X_1, X_1) & = & X_1 + X_2 & = & (a+b, [1,0,0]) \\
X_5 & = & F_5(X_3) & = & \cos X_3 & = & (\cos c, [0,0,0]) \\
X_6 & = & F_6(X_4) & = & \exp(X_4) & = & (e^{a+b}, [e^{a+b}, 0, 0]) \\
X_7 & = & F_7(X_4, X_5) & = & X_4 X_5 & = & ((a+b)\cos c, [\cos c, 0, 0]) \\
X_8 & = & F_8(X_6, X_7) & = & X_6 + X_7 & = & (e^{a+b} + (a+b)\cos c, \\
 & & & & & & \quad [e^{a+b} + \cos c, 0, 0] \quad) \\
X_9 & = & F_9(X_3, X_8) & = & X_8 / X_3 & = & (e^{a+b} + (a+b)\cos c)/c, \\
 & & & & & & \quad [(e^{a+b} + \cos c)/c, 0, 0] \quad)
\end{array}
$$

FIGURE 2. Benutzung des Ableitungscodes mit Operatorüberladen (links) zur Berechnung der partiellen Ableitung F_a. Die fünf mittleren Zeilen sind völlig unverändert, Grossbuchstaben deuten hier auf den veränderten Datentyp aus (7.1) hin.

```
function s=times(s1,s2)
s.val=s1.val.*s2.val;
for i=1:globp
    s.der(:,i)=s2.val(:).*s1.der(:,i)+s1.val(:).*s2.der(:,i);
end
```

FIGURE 3. Überladener Multiplikationsoperator des AD-Tools ADMAT (stark vereinfacht). Der ADMAT-Datentyp besteht aus den Komponenten `val`(ue) und `der`(ivative).

In Programmiersprachen werden die unterschiedlichen Definitionsbereiche durch unterschiedliche Datentypen (wie z.B. `real`, `complex` in FORTRAN) repräsentiert. In der objektorientierten Denkweise und damit in objektorientierten Programmiersprachen werden Daten und die damit möglichen Operationen als Einheit betrachtet und auch gemeinsam als sog. *Klasse* implementiert.

In analoger Weise kann man einen Datentyp definieren, der aus einer Variable und ihrer Ableitung (bezogen auf die Grösse, nach der abgeleitet werden soll) besteht, also

$$X \quad := \quad (x, x'). \tag{7.1}$$

Nun gibt man den in der jeweiligen Programmiersprache vorhandenen Funktionen und Operatoren eine Bedeutung, indem man die Regeln der Differentiation für die zweite Komponente x' von X implementiert. Bei der Multiplikation werden so z.B. die Werte der Operanden multipliziert und ihre Ableitungen nach der Produktregel der Differentiation weiterverarbeitet, vgl. Abb. 1.

Die Auswertung der Funktion $y = F(x)$ selbst und ihrer Ableitung mit der Kettenregel (4.11) erfolgen damit simultan durch Ausführen des ursprünglichen, in seinen Anweisungen unveränderten Programms, wenn alle relevanten (aktiven) Variablen vom Datentyp (7.1) sind. Am Anfang wird x' mit der Seed Matrix initialisiert, und am Ende enthält die zweite Komponente von $Y = (y, y')$ die entsprechende Ableitung, vgl. Tabelle 1 und das schematische Beispiel in Abb. 2.

Bei einer "typenlosen" Sprache (besser gesagt: einer Sprache, die automatisch alle generierten Variablen typisiert) wie MATLAB erhalten automatisch alle aktiven

Zwischenwerte den entsprechenden Datentyp (7.1). In einer typisierten Sprache wie C++ müssen alle relevanten *aktiven* Variablen mit dem entsprechenden Typ deklariert werden.

Die Definition des Datentyps (7.1) und die überladenen Operatoren und Funktionen werden in Form einer Bibliothek bereitgestellt. Diese wird mit dem ursprünglichen Programm zusammengebunden. Ein konkretes Beispiel für den überladenen Multiplikationsoperator bei dem AD-Tool ADMAT für MATLAB zeigt Abb. 3.

Analog zur Quelltransformation beschreiben wir die bei einem AD-Tool, das mit Operatorüberladen arbeitet, notwendigen Schritte von der diskretisierten Funktion bis zur Benutzung des generierten Codes zur Auswertung der Ableitung:

(1) Festlegung von In- und Outputvariablen (x und y).
(2) Identifizieren des relevanten (aktiven) Teils des Programms.
(3) Modifikation des ursprünglichen Programms:
   (a) *Vor* dem unveränderten aktiven Programmteil: Deklarieren der unabhängigen (Input-) Variable als X in dem entsprechenden Datentyp (7.1), der vom AD-Tool in einer Bibliothek bereitgestellt wird, und Initialisieren der Seed Matrix x' (entsprechende Komponente von X).
   (b) *Nach* dem unveränderten aktiven Programmteil: Ausgabe des Ergebnisses in y' (entsprechende Komponente von Y).
   (c) Gegebenenfalls (bei typisierenden Sprachen wie C++) müssen im aktiven Programmteil alle aktiven Variablen in dem entsprechenden Datentyp (7.1) deklariert werden.

Wir beschreiben die Vorgehensweise an zwei Beispielen: Zuerst betrachten wir ein Anwendungsproblem aus [**SZ04**], [**?**], in dem ein in MATLAB vorliegendes Programm algorithmisch mit ADMAT differenziert wurde.

Das zweite Beispiel in Abb. 4 zeigt, welche Änderungen bei einer typisierenden Sprache wie C++ notwendig sind. Hier wurde die erweiterte Rosenbrock-Funktion

$$\begin{aligned} F &: \mathbb{R}^4 \to \mathbb{R}, \\ F(x) &= 100(x_2 - x_1^2)^2 + (1 - x_1)^2 + 100(x_4 - x_3^2)^2 + (1 - x_3)^2, \end{aligned}$$

ein typisches Testbeispiel der nichtlinearen Optimierung, mit ADOL-C algorithmisch differenziert.

```
→ #include ''adolc/adouble''
→ #include ''adolc/drivers/drivers.h''

int main()
{
   double x[4], f;
   x = {...};
→ adouble ax[4], af;
   double F(double*);
→ adouble F(adouble*);
→ double adf[4];
→ trace_on(0);
→ for (i=0;i<4;i++) ax[i] <<= x[i];
// f = F(x);
→ af = F(ax);
→ af >>= f;
→ trace_off();
→ gradient(0,4,x,adf);
}

// double F(double* x)
→ adouble F(adouble* x)
{
   return 100.0*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0])+(1.0-x[0])*(1.0-x[0])
         +100.0*(x[3]-x[2]*x[2])*(x[3]-x[2]*x[2])+(1.0-x[2])*(1.0-x[2]);
}
```

FIGURE 4. Berechnung der Ableitung der erweiterten Rosenbrock-Funktion, realisiert in C++ mit ADOL-C (vereinfacht, →: eingefügte Zeilen). Entscheidend sind hier die Markierung der aktiven Codezeilen (zwischen den Befehlen `trace_on` und `trace_off`), die Deklarierung der aktiven Variablen und der Funktion mit dem ADOL-C-Typ `adouble`, und die Shiftoperationen, mit der die Konvertierung zwischen den urpsrünglichen `double`-Variablen `x`,`f` und ihren `adouble`-Entsprechungen `ax`,`af` erfolgt. Die Routine `gradient` erlaubt die direkte Auswertung des Gradienten. Analog kann man hier zusätzlich z.B. die Hesse-Matrix auswerten, ohne die differenzierte Funktion `F` ein zweites Mal aufzurufen. Der mit `//` auskommentierte Aufruf der Funktion `double F(double*)` wird nicht mehr benötigt. Wegen des in C++ möglichen Polymorphismus kann diese Funktion jedoch weiter, z.B. zum Vergleich mit einem Finite-Differenzen-Gradienten, benutzt werden, sofern sie definiert ist.

CHAPTER 8

Reverse mode

1. Motivation

The motivation for "something different" from the forward mode is the effort ($\geq \Theta(n)$ as seen in Section 5.5) which is not better than the one of a numerical differentation. The fact that there might be something better can be seen by going back to the first representation of a function

$$\begin{aligned} F : \mathbb{R}^n &\to \mathbb{R}^m \\ F : x &\mapsto y \end{aligned}$$

as a concatenation of k functions

$$F = F_k \circ \cdots \circ F_1$$

with

$$F_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}, \quad n_0 = n, n_k = m,$$

and again the intermediate variables

$$\begin{aligned} x_0 &:= x \\ x_i &:= F_i(x_{i-1}), \qquad i = 1, \ldots, k \\ y = F(x) &= x_k. \end{aligned}$$

The chain rule for the derivative of F, evaluated from the beginning to the end ($i = 1, \ldots, k$), gives:

$$\underbrace{F'(x)}_{(m\times n)} = \underbrace{F_k'(x_{k-1})}_{(m\times\ldots)} \cdots \underbrace{F_3'(x_2)}_{(\ldots\times\ldots)} \cdot \underbrace{\underbrace{F_2'(x_1)}_{(\ldots\times\ldots)} \cdot \underbrace{F_1'(x_0)}_{(\ldots\times n).}}_{(\ldots\times n)}$$

It follows that

$$x_i' = F_i(x_{i-1}) \in \mathbb{R}^{n_i \times n}.$$

Note that the x_i' can be (and usually are) matrices, dependent on the intermediate sizes $n_i, i > 0$. Anyhow, the sizes of the x_i' are bounded from above by the number n of independent variables times the maximum size ($\max_i n_i$) of the x_i'. For example, if $m = n_0 = 1$ and thus $F'(x)$ is the gradient (vector!), only in the last step of the chain rule evaluation the intermediate object is a vector.

If, in contrary, the matrix product in the chain rule is evaluated beginning from the end (i.e. for $i = k, \ldots, 1$), we obtain

$$\underbrace{F'(x)}_{(m\times n)} = \underbrace{\underbrace{F_k'(x_{k-1})}_{(m\times\ldots)} \cdot \underbrace{F_{k-1}'(x_{k-2})}_{(\ldots\times\ldots)}}_{(m\times\ldots)} \cdot \underbrace{F_{k-2}'(x_{k-3})}_{(\ldots\times\ldots)} \cdots \underbrace{F_1'(x_0)}_{(\ldots\times n)}$$

The intermediates $\bar{x}_j$ defined by

$$F'_k(x_k)\cdots F'_{i+1}(x_i) = \prod_{j=i+1}^{k} F'_j(x_{j-1}) \tag{8.1}$$
$$= \frac{d(F_k \circ \cdots \circ F_{i+1})}{dx_i}(x) = \frac{dy}{dx_i} =: \bar{x}_i^\top$$

satisfy

$$\bar{x}_i \in \mathbb{R}^{n_i \times m}.$$

2. Adjoint variables

The intermediate values in the reverse mode (8.1) are called *adjoint variables*. Clearly they satisfy the formula:

$$\bar{x}_j^\top = \bar{x}_{j+1}^\top F'_{j+1}(x_j),$$

or equivalently

$$\bar{x}_{j-1} = F'_j(x_{j-1})^\top \bar{x}_j, \quad j = k, \ldots, 1. \tag{8.2}$$

Their size is now bounded from above by the number m of dependent variables times the maximum size ($\max_i n_i$) of the $\bar{x}_i$. In the case $m = 1$ all $\bar{x}_i$ are vectors. This makes clear that the evaluation

- in forward direction is preferable if $n \ll m$,
- and in reverse direction if $m \ll n$.

The initialization is performed by the analogue to the seed matrix,

$$\left(\frac{dx_k}{dx_k}\right)^\top = \bar{x}_k =: \bar{y},$$

It now can be used to compute only some components or linear combinations of them of the derivative $\bar{x}_0^\top = y' = F'(x)$. A computation of partial or directional derivatives of F is performed easily afterwards by selecting or combining the corresponding componets of $\bar{x}_0^\top$.

The following example treats the case $m = 1$, namely again the Rosenbrock function

$$\begin{aligned} F: \quad \mathbb{R}^2 &\to \mathbb{R} \\ F(\xi_1, \xi_2) &= 100(\xi_2 - \xi_1^2)^2 + (1 - \xi_1)^2. \end{aligned}$$

EXAMPLE 17. *We have already written F as concatenation of $F_i, i = 1, \ldots, 7$ in Example 11 on page 27:*

$$
\begin{array}{rcl}
x_0 &=& \begin{bmatrix} x_{01} \\ x_{02} \end{bmatrix} = \begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} \\
x_1 &=& F_1(x_0) = \begin{bmatrix} x_{01} \\ x_{02} \\ x_{01}^2 \end{bmatrix} = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_1^2 \end{bmatrix} =: \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \end{bmatrix} \\
x_2 &=& F_2(x_1) = \begin{bmatrix} x_{11} \\ x_{12} - x_{13} \end{bmatrix} = \begin{bmatrix} \xi_1 \\ \xi_2 - \xi_1^2 \end{bmatrix} =: \begin{bmatrix} x_{21} \\ x_{22} \end{bmatrix} \\
x_3 &=& F_3(x_2) = \begin{bmatrix} x_{21} \\ x_{22}^2 \end{bmatrix} = \begin{bmatrix} \xi_1 \\ (\xi_2 - \xi_1^2)^2 \end{bmatrix} =: \begin{bmatrix} x_{31} \\ x_{32} \end{bmatrix} \\
x_4 &=& F_4(x_3) = \begin{bmatrix} x_{31} \\ 100 x_{32} \end{bmatrix} = \begin{bmatrix} \xi_1 \\ 100(\xi_2 - \xi_1^2)^2 \end{bmatrix} =: \begin{bmatrix} x_{41} \\ x_{42} \end{bmatrix} \\
x_5 &=& F_5(x_4) = \begin{bmatrix} 1 - x_{41} \\ x_{42} \end{bmatrix} = \begin{bmatrix} 1 - \xi_1 \\ 100(\xi_2 - \xi_1^2)^2 \end{bmatrix} =: \begin{bmatrix} x_{51} \\ x_{52} \end{bmatrix} \\
x_6 &=& F_6(x_5) = \begin{bmatrix} x_{51}^2 \\ x_{52} \end{bmatrix} = \begin{bmatrix} (1 - \xi_1)^2 \\ 100(\xi_2 - \xi_1^2)^2 \end{bmatrix} =: \begin{bmatrix} x_{61} \\ x_{62} \end{bmatrix} \\
x_7 &=& F_7(x_6) = x_{61} + x_{62} = F(x).
\end{array}
$$

In Example 12 on page 28 we already computed the matrices:

$$
\begin{array}{rcl}
F_1'(x_0) &=& \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 2x_{01} & 0 \end{bmatrix}, \quad F_2'(x_1) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}, \\
F_3'(x_2) &=& \begin{bmatrix} 1 & 0 \\ 0 & 2x_{22} \end{bmatrix}, \quad F_4'(x_3) = \begin{bmatrix} 1 & 0 \\ 0 & 100 \end{bmatrix}, \\
F_5'(x_4) &=& \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}, \quad F_6'(x_5) = \begin{bmatrix} 2x_{51} & 0 \\ 0 & 1 \end{bmatrix}, \quad F_7'(x_6) = [1, 1].
\end{array}
$$

To compute the derivative (i.e, the gradient) using the reverse mode and this representation we start with

$$\bar{y} = \bar{x}_7 = 1$$

and obtain according to (8.2)*:*

$$
\begin{array}{rcl}
\bar{x}_6 &=& F_7'(x_6)^\top \bar{x}_7 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \\
\bar{x}_5 &=& F_6'(x_5)^\top \bar{x}_6 = \begin{bmatrix} 2x_{51} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2x_{51} \\ 1 \end{bmatrix}, \\
\bar{x}_4 &=& F_5'(x_4)^\top \bar{x}_5 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2x_{51} \\ 1 \end{bmatrix} = \begin{bmatrix} -2x_{51} \\ 1 \end{bmatrix}, \\
\bar{x}_3 &=& F_4'(x_3)^\top \bar{x}_4 = \begin{bmatrix} 1 & 0 \\ 0 & 100 \end{bmatrix} \begin{bmatrix} -2x_{51} \\ 1 \end{bmatrix} = \begin{bmatrix} -2x_{51} \\ 100 \end{bmatrix}, \\
\bar{x}_2 &=& F_3'(x_2)^\top \bar{x}_3 = \begin{bmatrix} 1 & 0 \\ 0 & 2x_{22} \end{bmatrix} \begin{bmatrix} -2x_{51} \\ 100 \end{bmatrix} = \begin{bmatrix} -2x_{51} \\ 200x_{22} \end{bmatrix}, \\
\bar{x}_1 &=& F_2'(x_1)^\top \bar{x}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} -2x_{51} \\ 200x_{22} \end{bmatrix} = \begin{bmatrix} -2x_{51} \\ 200x_{22} \\ -200x_{22} \end{bmatrix}
\end{array}
$$

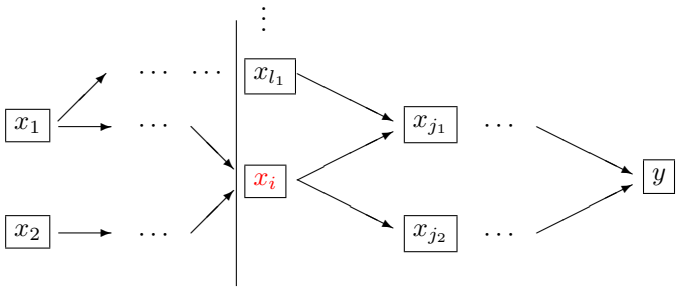

FIGURE 1. Motivation for the computation of the adjoint $\bar{x}_i$ by the chain rule in the reverse mode, see (8.4). The meaning of the adjoint variable $\bar{x}_i = \frac{\partial y}{\partial x_i}$ is visualized by the vertical line: The output y may be regarded as function of the intermediates x_i, x_l in this example, i.e. only the right part of the graph is considered. The adjoint $\bar{x}_i$ then is the derivative of the output w.r.t. x_i.

and finally

$$\begin{aligned}\bar{x}_0 \;=\; F_1'(x_0)^\top \bar{x}_1 \quad &= \quad \begin{bmatrix} 1 & 0 & 2x_{01} \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} -2x_{51} \\ 200x_{22} \\ -200x_{22} \end{bmatrix} \\ &= \quad \begin{bmatrix} -2x_{51} - 400x_{01}x_{22} \\ 200x_{22} \end{bmatrix} \\ &= \quad \begin{bmatrix} -2(1-\xi_1) - 400\xi_1(\xi_2 - \xi_1^2) \\ 200(\xi_2 - \xi_1^2) \end{bmatrix}.\end{aligned}$$

This the transposed gradient of F, compare (4.2) on page 23.

3. Graph representation

In the graph representation already used for the forward mode, we have set, see (5.1) on page 31 and (5.4) on page 32:

$$\begin{aligned}[x_1, \ldots, x_n] \quad &= \quad x \in \mathbb{R}^n \\ x_i \quad &= \quad F_i((x_j)_{j \prec i}), \qquad i = n+1, \ldots, n+k \\ y \;=\; F(x) \quad &= \quad x_{n+k}.\end{aligned} \tag{8.3}$$

Remember that the intermediate values x_i and also y are scalar quantities here. In the definition of the adjoint variables from (8.1), i.e.,

$$\bar{x}_i \quad := \quad \frac{dy}{dx_i},$$

we then may omit the transposition since the adjoint variables are now scalar values, too. Using the chain rule, we may write for the adjoint in the i-th step:

$$\bar{x}_i \quad = \quad \frac{dy}{dx_i} \;=\; \sum_{j:\, i \prec j} \underbrace{\frac{\partial y}{\partial x_j}}_{=\bar{x}_j} \frac{\partial x_j}{\partial x_i} \;=\; \sum_{j:\, i \prec j} \bar{x}_j \frac{\partial x_j}{\partial x_i} \;=\; \sum_{j:\, i \prec j} \bar{x}_j \frac{\partial F_j}{\partial x_i}((x_l)_{l \prec j}). \tag{8.4}$$

This computation is depicted in an example in Figure 1.

To write down a realization of this formula in an algorithm we change the orde rof the two loops that are involved, namely the outer one (over i) to compute the

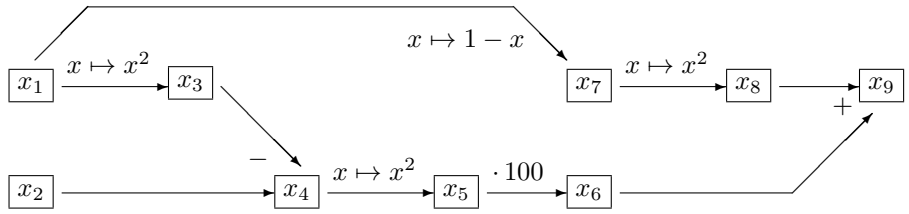

FIGURE 2. Computational graph of the Rosenbrock function.

$\bar{x}_i$ and the inner one (over j), necessary for the summation. This is done in the following algorithm of the reverse mode:

(8.5)
$$
\left\{
\begin{array}{l}
\text{Initialize } [x_1, \ldots, x_n] = x \in \mathbb{R}^n \\
\text{For } i = n+1, \ldots, n+k: \\
\qquad \text{Compute } x_i = F_i((x_l)_{l \prec i}) \\
\text{Return function value: } y \;=\; F(x) = x_{n+k} \\
\text{For } i = 1, \ldots, n-k+1: \\
\qquad \text{Initialize } \bar{x}_i = 0 \\
\text{Initialize } \bar{y} = \bar{x}_{n+k} = 1 \\
\text{For } j = n+k, \ldots, n+1: \\
\qquad \text{For } i \prec j: \\
\qquad\qquad \text{Compute } \bar{x}_i = \bar{x}_i + \bar{x}_j \dfrac{\partial F_j}{\partial x_i}((x_l)_{l \prec j}) \\
\text{Return derivative/ gradient: } F'(x) = [\bar{x}_1, \ldots, \bar{x}_n].
\end{array}
\right.
$$

The first four lines are a complete function evaluation also called *forward sweep*. In this – ideal – formulation all intermediate values x_i are stored, since they are used in the double for-loop as inputs for the F_j.

To see that this works we consider again the Rosenbrock example, whose computational graph is again shown in Figure 2.

EXAMPLE 18. *Forward sweep (function evaluation) for the Rosenbrock function:*

$$
\begin{array}{lllll}
x_3 &=& F_3(x_1) &=& x_1^2 \\
x_4 &=& F_4(x_2, x_3) &=& x_2 - x_3 \qquad\qquad = \quad x_2 - x_1^2 \\
x_5 &=& F_5(x_4) &=& x_4^2 \\
x_6 &=& F_6(x_5) &=& 100 x_5 \\
x_7 &=& F_7(x_1) &=& 1 - x_1 \\
x_8 &=& F_8(x_7) &=& x_7^2 \\
x_9 &=& F_9(x_6, x_8) &=& x_6 + x_8 \;=\; F(x) \;=\; y.
\end{array}
$$

Initialization of adjoint variables:

$$\bar{x}_i = 0, i = 1, \ldots, 8, \qquad \bar{y} = \bar{x}_9 = 1.$$

Derivative computation by reverse mode ($n = 2, n + k = 9$):

$$
\begin{array}{llllllll}
j=9: & i=8: & \bar{x}_8 & = & \bar{x}_8 + \bar{x}_9 \dfrac{\partial F_9}{\partial x_8} & = & 0 + 1 \cdot 1 & = & 1 \\
 & i=6: & \bar{x}_6 & = & \bar{x}_6 + \bar{x}_9 \dfrac{\partial F_9}{\partial x_6} & = & 0 + 1 \cdot 1 & = & 1 \\
j=8: & i=7: & \bar{x}_7 & = & \bar{x}_7 + \bar{x}_8 \dfrac{\partial F_8}{\partial x_7} & = & 0 + 1 \cdot 2x_7 & = & 2x_7 \\
j=7: & i=1: & \bar{x}_1 & = & \bar{x}_1 + \bar{x}_7 \dfrac{\partial F_7}{\partial x_1} & = & 0 + 2x_7 \cdot (-1) & = & -2x_7 \\
j=6: & i=5: & \bar{x}_5 & = & \bar{x}_5 + \bar{x}_6 \dfrac{\partial F_6}{\partial x_5} & = & 0 + 1 \cdot 100 & = & 100 \\
j=5: & i=4: & \bar{x}_4 & = & \bar{x}_4 + \bar{x}_5 \dfrac{\partial F_5}{\partial x_4} & = & 0 + 100 \cdot 2x_4 & = & 200x_4 \\
j=4: & i=3: & \bar{x}_3 & = & \bar{x}_3 + \bar{x}_4 \dfrac{\partial F_4}{\partial x_3} & = & 0 + 200x_4 \cdot (-1) & = & -200x_4 \\
j=3: & i=2: & \bar{x}_2 & = & \bar{x}_2 + \bar{x}_4 \dfrac{\partial F_4}{\partial x_2} & = & 0 + 200x_4 \cdot 1 & = & 200x_4 \\
 & & & & & & & = & 200(x_2 - x_1^2) \\
 & i=1: & \bar{x}_1 & = & \bar{x}_1 + \bar{x}_3 \dfrac{\partial F_3}{\partial x_1} & = & -2x_7 - 200x_4 \cdot 2x_1 & = & -2x_7 - 400x_1x_4 \\
 & & & & & & & = & -2(1 - x_1) \\
 & & & & & & & & -400x_1(x_2 - x_1^2).
\end{array}
$$

Now $[\bar{x}_1, \bar{x}_2]$ is the gradient. It can be seen that the two intermediate variables x_4, x_7 have to be stored or re-computed since they are needed to evaluate the derivative.

The above algorithm of the reverse mode can be generalized in the following way:

(8.6)
$$
\left\{
\begin{array}{ll}
 & \text{Function evaluation as above.} \\
 & \text{For } i = 1, \ldots, n: \\
\rightarrow & \qquad \text{Initialize } \bar{x}_i = g_i \in \mathbb{R} \\
 & \text{For } i = n+1, \ldots, n-k+1: \\
 & \qquad \text{Initialize } \bar{x}_i = 0 \\
\rightarrow & \text{Initialize } \bar{y} = \bar{x}_{n+k} = w \in \mathbb{R} \\
 & \text{Reverse derivative computation as above.}
\end{array}
\right.
$$

Differences occur only in the initialization of the adjoints $\bar{y}, \bar{x}_i, i = 1, \ldots, n$. Looking closely at the algorithm, one observes that it then computes

$$[\bar{x}_1, \ldots, \bar{x}_n] \quad = \quad g + wF'(x),$$

where $g = [g_1, \ldots, g_n]$. The g_i and w are arbitrary weights. This form of the algorithm is useful when studying the effort of the reverse mode in the next section.

4. Computational effort

We now compare the effort of an evaluation of the above algorithm (8.6) with the one necessary to evaluate just F. Similarly to the estimate in Section 5.5 we want to obtain a bound for

$$Q_{F,rev} \quad := \quad \frac{E_{rev}(F, g + wF')}{E(F)}$$

for arbitrary weights $g \in \mathbb{R}^n, w \in \mathbb{R}$. The reason for choosing this quantity (and not

$$q_{F,rev} = \frac{E_{rev}(F, F')}{E(F)}$$

as in Section 5.5 for the forward mode) lies in the technique of the following estimates. Clearly

$$E_{rev}(F, g + wF') \quad \geq \quad E_{rev}(F, F')$$

and thus

$$Q_{F,rev} \quad \geq \quad q_{F,rev}.$$

We obtain

$$Q_{F,rev} \quad = \quad \frac{E_{rev}(F, g + wF')}{E(F)} \quad = \quad \frac{\sum_{i=n+1}^{n+k} E(F_j, \bar{g}_j + wF'_j)}{\sum_{i=n+1}^{n+k} E(F_j)} \tag{8.7}$$

where F'_j denotes the vector of partial derivatives

$$F'_j \quad := \quad \left[\frac{\partial F_j}{\partial x_i}\right]_{i \prec j} \in \mathbb{R}^{n_j}, \quad n_j := |I_j|,$$

and

$$\bar{g}_j \quad := \quad [g_j]_{i \prec j} \in \mathbb{R}^{n_j}$$

the corresponding components of g. We define the ratio Q also for every function F_j and its derivative separately as

$$Q_{F_j} \quad := \quad \frac{E(F_j, \bar{g}_j + wF'_j)}{E(F_j)}.$$

This ratio is independent of forward or reverse mode since only one function is differentiated. Now we get from (8.7) that

$$Q_{F,rev} \quad = \quad \frac{\sum_{i=n+1}^{n+k} Q_{F_j} E(F_j)}{\sum_{i=n+1}^{n+k} E(F_j)} \quad \leq \quad \max_{j=n+1,\ldots,n+k} Q_{F_j}$$

As a consequence we obtain that $Q_{F,rev}$ is bounded form above – independently of n (!!!) – if Q_{F_j} is bounded for all j. For the elementary operators and mathematical functions it is shown in [**Gri89**] that a reasonable upper bound is

$$Q_{F_j} \leq 5 \quad \forall j$$

and thus also

$$Q_{F,rev} \leq 5,$$

which does not depend on n. Computation of the derivative by the reverse mode for a function $F : \mathbb{R}^n \to \mathbb{R}$ thus can be performed efficiently for high values of n. But: What is not taken into account is any storage or re-computation effort necessary for the needed intermediate values (from the forward sweep) that are used in the reverse derivative computation. In the Rosenbrock example, these were x_4, x_7. In other cases, it can be more. Thus the treatment of these variables and specifically the decision whether to re-compute or to store – or to combine both strategies efficiently – is of high importance for the overall efficiency of the reverse mode.

CHAPTER 9

Non-differentiability and floating point exceptions

A common question is what AD does (or should do) with points where the function F to be differentiated is not differentiable. In most programming languages there are intrinsic functions as modulus, maximum, minimum functions available. All of them are non-differentiable at certain points. Moreover most programs contain `if` or `while`-statements that also may introduce non-differentiability.

In our representation of the function as a computational graph, see (5.1) on page 31, we introduced F_i as a single-valued (not vector-valued) function. In the forward and reverse mode algorithms (compare (5.8) on page 35 and (8.5) on page 51), we only have to compute derivatives with respect to *one* input variable at a time. Thus we may as well consider the case where F_i only depends on a single input x_{i-1}.

In the forward mode, we then compute

$$F_i'(x_{i-1})x_{i-1}'. \tag{9.1}$$

This expression can be interpreted as the directional derivative

$$F_i'(x_{i-1})d \tag{9.2}$$

of F_i at the point x_{i-1} in direction $d = x_{i-1}'$. As long as this directional derivative exists, the above statement in the forward mode algorithm makes sense. If – according to an appropriate setting of the seed matrix – the intermediate variable x_{i-1}' is a vector, then (9.1) is a vector of directional derivatives.

In the reverse mode we compute

$$\bar{x}_j \frac{\partial F_j}{\partial x_i} \quad \text{for } i \prec j,$$

which in our situation (where F_i depends on x_{i-1} only) translates to

$$\bar{x}_{i-1}F_i'(x_{i-1}).$$

Since in the reverse mode algorithm the adjoints (and thus $\bar{x}_{i-1}$) are scalar variables, we may as well write

$$F_i'(x_{i-1})\bar{x}_{i-1},$$

which again is a directional derivative (9.2), now with $d = \bar{x}_{i-1}$.

Thus, from the point of differentiability, only the question of directional or partial differentiability is relevant for both forward and reverse mode.

In (5.7) on page 33 we have written conditional statements like *if* statements as

$$F_i(x_{i-1}) \quad := \quad \begin{cases} F_{i1}(x_{i-1}), & cond_i \\ F_{i0}(x_{i-1}), & \neg\, cond_i, \end{cases} \tag{9.3}$$

i.e., as a function F_i representing a branch in the computational graph. Here $cond_i$ is the corresponding condition (i.e., a boolean expression).

1. Conditional statements depending on passive variables

With the conditional statement written like above, it is important whether the condition $cond_i$ depends on an active variable or not.

If *not*, then $cond_i$ depends only on one or several passive variable(s) which we summarize here by λ. We may then write

$$F_i(x_{i-1}) = F_i(x_{i-1}, \lambda) \quad = \quad F_{ij}(x), \quad \lambda \in \Lambda_j, j \in \{0, 1\}.$$

where $\Lambda_0 \cup \Lambda_1$ is the set of all possible values the passive variable λ may take. Note that we only consider here a simple branch (corresponding to a simple *if-else*), and assume that more complicated or nested constructs are separated into such simple ones. Each index j then represents a branch in the computational graph as in (9.3). Here we have differentiability since F_i is differentiable with respect to x_{i-1}.

EXAMPLE 19. *Consider the function*

$$F_i(x_{i-1}) \quad = \quad F_i(x_{i-1}, \lambda) \; = \; \begin{cases} x_{i-1} & =: F_{i1}(x_{i-1}), \quad \text{if } \lambda \geq 0 \\ -x_{i-1} & =: F_{i0}(x_{i-1}), \quad \text{if } \lambda < 0, \end{cases}$$

where λ is some (passive) variable or parameter. Note that F_i is not *the modulus/ absolute value function. Every F_{ij} can be differentiated separately, and that is what a source transformation tool and also an operator overloading implementation will do.*

2. Nonsmooth functions – Conditional statements depending on active variables

If in the other case the condition $cond_i$ depends on an active variable (here, for the simplest case, on the only one, namely x_{i-1}) then we may write it as

$$cond_i(x_{i-1}) \quad = \quad (x_{i-1} \in S_i) \; := \; \begin{cases} true, & \text{if } x_{i-1} \in S_i, \\ false, & \text{if } x_{i-1} \notin S_i, \end{cases}$$

where here S_i is a certain subset of $\mathbb{R}$ and the term in the middle is regarded as a boolean expression. We then again have the situation as in (9.3).

EXAMPLE 20. *A simple example is the modulus/ absolute value function*

$$F_i(x_{i-1}) \quad = \quad \begin{cases} x_{i-1}, & \text{if } x_{i-1} \geq 0 \iff x_{i-1} \in S_i \iff cond_i, \\ -x_{i-1}, & \text{if } x_{i-1} < 0 \iff x_{i-1} \notin S_i \iff \neg\, cond_i, \end{cases}$$

with $S_i = \{x \in \mathbb{R} : x \geq 0\} =: \mathbb{R}_0^+$.

Now the differentiability properties of the function F_i depend on those of the two functions $F_{ij}, j = 0, 1$, at the point(s) where the condition changes its (boolean) value. These points belong to the boundary of the set S_i, denoted by ∂S_i.

There are three cases concerning the differentiability properties of F_i at a critical point $x_{i-1} = x^* \in \partial S_i$ where the condition changes its value:

(1) The function F_i is differentiable. This is the case if the transition between F_{i0} and F_{i1} in the critical point $x^* \in \partial S_i$ is differentiable, i.e., if

$$F'_{i1}(x^*)d \quad = \quad F'_{i0}(x^*)(-d), \tag{9.4}$$

where d is the direction pointing from the interior of S_i to the boundary point $x^* \in \partial S_i$.

(2) The function is not differentiable, but directional differentiable in both directions at the considered point $x^* \in \partial S_i$, i.e., both directional derivatives in (9.4) exist but satisfy

$$F'_{i1}(x^*)d \quad \neq \quad F'_{i0}(x^*)(-d).$$

(3) At least one of the two directional derivatives in (9.4) does not exist. Then F_i is not differentiable, and the corresponding branch should generate an error or exception.

We give some examples for these different cases.

EXAMPLE 21. *Differentiable function:*

$$F_i(x_{i-1}) = \begin{cases} F_{i1}(x_{i-1}) = x_{i-1}^2, & x_{i-1} \geq 0 \\ F_{i0}(x_{i-1}) = 0, & x_{i-1} < 0. \end{cases}$$

Here we have $S_i = \mathbb{R}_0^+$ and $\partial S_i = \{0\}$, i.e., the critical point is $x^* = 0$. Since for $d > 0$

$$F'(0)d = \lim_{h \to 0+} \frac{F_i(\overbrace{0 + hd}^{>0}) - F_i(0)}{h} = \lim_{h \to 0+} \frac{(hd)^2 - 0}{h} = \lim_{h \to 0+} hd^2 = 0$$

and for $d < 0$

$$F'(0)d = \lim_{h \to 0+} \frac{F_i(\overbrace{0 + hd}^{<0}) - F_i(0)}{h} = \lim_{h \to 0+} \frac{0 - 0}{h} = 0$$

the transition is differentiable, and the whole function is differentiable.

EXAMPLE 22. *Not differentiable, but directional differential function:*

$$F_i(x_{i-1}) = \begin{cases} F_{i1}(x_{i-1}) = x_{i-1}, & x_{i-1} \geq 0 \\ F_{i0}(x_{i-1}) = 0, & x_{i-1} < 0. \end{cases}$$

Again $S_i = \mathbb{R}_0^+, \partial S_i = \{0\}$ with the critical point $x^* = 0$. Now for $d > 0$

$$F'(0)d = \lim_{h \to 0+} \frac{F_i(\overbrace{0 + hd}^{>0}) - F_i(0)}{h} = \lim_{h \to 0+} \frac{hd - 0}{h} = \lim_{h \to 0+} d = d$$

and for $d < 0$

$$F'(0)d = \lim_{h \to 0+} \frac{F_i(\overbrace{0 + hd}^{<0}) - F_i(0)}{h} = \lim_{h \to 0+} \frac{0 - 0}{h} = 0.$$

The transition is not differentiable, but the two directional derivatives exist.

Treatment in the forward mode. The forward mode may treat every branch of the conditional statement separately. The result will then reflect the differentiabilty properties of the function correctly in all points $x_{i-1} \notin \partial S_i$, i.e. all non-critical points.

Additionally, if both branches are directional differentiable, the forward mode algorithm can be augmented in such a way that it provides the correct directional derivative in critical points $x^* \in \partial S_i$ as well, depending on the provided direction x'_{i-1}. This results in the following setting:

$$F_i'(x_{i-1})x'_{i-1} := \begin{cases} F'_{i1}(x_{i-1})x'_{i-1}, & x_{i-1} \in S_i \setminus \partial S_i \\ F'_{i0}(x_{i-1})x'_{i-1}, & x_{i-1} \notin S_i \cup \partial S_i, \\ F'_{i1}(x_{i-1})x'_{i-1}, & x_{i-1} \in \partial S_i, x_{i-1} + x'_{i-1} \in S_i \\ F'_{i0}(x_{i-1})x'_{i-1}, & x_{i-1} \in \partial S_i, x_{i-1} + x'_{i-1} \notin S_i. \end{cases}$$

The first two cases are the usual setting, but only in the interior of the set S_i and its complement $\mathbb{R} \setminus S_i$, the others treat the critical points in ∂S_i. See Figure 1 for an example.

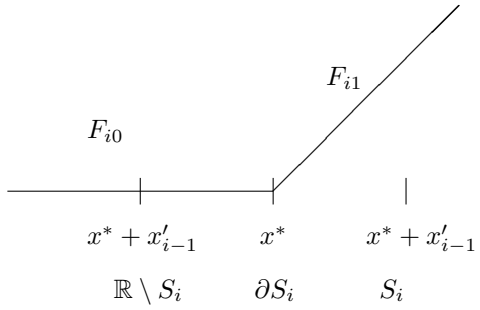

FIGURE 1. Chosing the correct directional derivative for a critical point $x_{i-1} = x^* \in \partial S_i$ in a conditional statement, depending on the direction x'_{i-1}. In this picture a positive value of x'_{i-1} points to the interior of S_i (here to the right), whereas a negative value points to the outside of S_i (here to the left).

EXAMPLE 23. *We consider the modulus (absolute value) function (omitting the indices $i-1, i$):*

$$abs(x) \quad := \quad \begin{cases} -x, & x < 0 \\ x, & x \geq 0. \end{cases}$$

Its derivative in forward mode can be implemented as

$$abs'(x)x' \quad := \quad \begin{cases} -x', & x < 0 \text{ or } (x = 0 \text{ and } x' < 0), \\ x', & x > 0 \text{ or } (x = 0 \text{ and } x' > 0). \end{cases}$$

3. Non-differentiable functions with infinite slope

Another case is given by non-differentiable intrinsic functions where not even a directional derivative exists at one or more critical point(s). This case occurs for example when the slope of the tangent is infinity. One may distinguish between functions where the infinite slope occurs at a point where the function is not defined, i.e., at the boundary of the open (!) domain of the function (e.g., $F(x) = \frac{1}{x}$ at $x = 0$) or those where the function is defined, as, e.g., the square root function $F(x) = \sqrt{x}$. Its derivative $F'(x) = \frac{1}{2\sqrt{x}}$ does not exist at the point $x = 0$, which is a boundary point of the domain, but now the domain is closed.

In both cases there is no way to define a meaningful directional derivative in the critical point $x = 0$. One can either evaluate $F'(x)$ at $x = 0$ obtaining a floating point exception, stop the execution of the programme or set the derivative to any given (user-defined) value, e.g., zero.

4. Floating point exceptions generated by AD

Even when not evaluated at a critical point where the derivative is infinity, the derivative computed at points close to the critical one may cause a floating point exception. In the above example $F(x) = \frac{1}{x}$ a value of $|x| < \frac{1}{x_{max}}$, where x_{max} denotes the biggest floating point number on the machine, the value $F'(x) = \frac{1}{x^2} > x_{max}$ exceeds the range of the machine numbers and thus causes a floating point exception, whereas evaluation of $F(x) = \frac{1}{x}$ does not.

CHAPTER 10

Differentiation of iterative algorithms

Many computations are done by iterative algorithms, e.g., the computations of the square root or more complex simulation where temporally steady states are looked for. Usually the iteration is stopped when some condition on the iterates (or derived quantities as differences or norms) are satisfied. When using AD, it is crucial to know how such an iteration is transformed by the AD software. Moreover, in reverse mode, it is important to deal with the usually large overhead due to extenisve computations that occur when applying the reverse mode naively.

In this chapter, we give examples for typical iterative algorithms, show what "naive" AD produces, and finally present several strategies to deal with the disadvantages.

1. One-step iterations

We start here by giving some examples for iterative algorithms. We start by iterations we call *one-step* iterations, as they have the characteristic feature that one iterate can be computed by knowing just the single preceeding one.

Let a function $F : \mathbb{R}^n \to \mathbb{R}^m$ describe an iterative process with iteration function $g : \mathbb{R}^{n\times m} \to \mathbb{R}^m$, i.e.,

$$y \;=\; F(x) \quad :\Longleftrightarrow \quad y = \lim_{i\to\infty} y_i \;\text{ for }\; y_{i+1} = g(x, y_i) \quad i = 0, 1, \ldots \tag{10.1}$$

where the initial value $y_0 \in \mathbb{R}^m$ for the iteration is given. The limit $y \in \mathbb{R}^n$ (if it exists, a fact that we will assume here) is called a fixed-point of g. Here $x \in \mathbb{R}^n$, the quantity is not changed during the iteration, it can be seen as a parameter. We want to differentiate the limit y of the iteration with respect to this quantity. A special case would be that the initial value y_0 itself is the parameter x, i.e., we want to compute the sensitivity of the limit with respect to the initial value.

In a computer programme, the limit y of course can only be approximated, i.e. (for given accuracy ε), we compute

$$\begin{aligned} y \;=\; F(x) \quad :\Longleftrightarrow \quad & y = y_k \;\text{ where }\; \|y_k - y_{k-1}\| < \varepsilon \text{ for some } k, \\ & y_{i+1} = g(x, y_i) \quad i = 0, \ldots, k-1. \end{aligned}$$

Note that here also the stopping accuracy ε becomes a parameter of F.

In the AD context, we are still after the derivative of $y = y_k$, i.e., the last iterate, with respect to some parameter (the input) x. This parameter x might be

- the initial value of the iteration y_0, or y_0 depends on the parameter x. Then usually the iteration function g itself will not depend on it, i.e., we have $g = g(y)$ instead of $g(x, y)$. Then g_x which shall denote the partial derivative $\frac{\partial g}{\partial x}$ vanishes. We will then have to initialize the derivative object $y'_0 = x'$ accordingly.
- a parameter of the iteration function, i.e., we really have $g = g(x, y)$ and $g_x \neq 0$.

Of course a combination of both is possible, since we allow vector-valued parameters $x \in \mathbb{R}^n$.

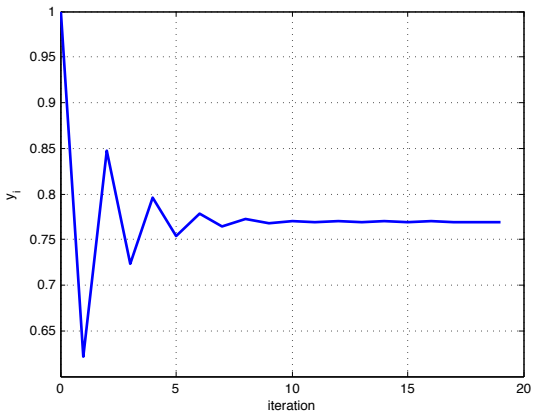

FIGURE 1. Iteration $y_{i+1} = \cos(y_i)$ with $y_0 = 1$.

The notation here is slightly different from the one used for simple functions like the Rosenbrock example above. Here we use y_i for the iterates which are of course intermediate values in the evaluation of F. The y_i can be vectors themselves. If they are scalars, the above notation of fits in the one used in the graph representation (5.1) on page 31 with

$$x_{n+i} \quad = \quad y_i, \quad i = 1, \ldots, k.$$

We now give some examples where such kind of iterations occur.

EXAMPLE 24. *Fixed-point problem*

$$y \quad = \quad f(x, y).$$

Under certain assumptions stated in Banach's fixed-point theorem (see Theorem 2 on page 14, with F there is $f(x, \cdot)$ here), a unique fixed-point y exists and the iteration (10.1) *with $g = f$ converges to y. If especially*

$$\|F'\| \text{ (from Theorem 2)} = \|f_y\| = \|g_y\| =: K < 1,$$

this assumption is satisfied, compare Theorem 3 on page 14. Here we used the notation $\frac{\partial g}{\partial y} =: g_y$ and the same for f.

A fixed-point iteration may also converge if the contraction condition ($K < 1$) cannot be shown for all y, but is satisfied for all iterates:

EXAMPLE 25. *Consider the iteration (no x here)*

$$y_{i+1} \quad = \quad g(y_i) \; := \; \cos(y_i)$$

which for the initial value $y_0 = 1$ is shown in Figure 1. It obviously converges though the derivative of the iteration function g can only be estimated as

$$|g_y(y)| \; = \; |g'(y)| \quad = \quad |-\sin(y)| \; = \; |\sin(y)| \; \leq \; K \; = \; 1 \quad \forall y \in \mathbb{R},$$

and not by $K < 1$ (which would be necessary to apply Banach's Theorem). But since $|\sin(y)| = 1$ only for $y = \pm(2n-1)\frac{\pi}{2}, n \in \mathbb{Z}$ and these values do not occur during the iteration, we have

$$|g_y(y_i)| \quad \leq \quad K \quad \forall i = 0, \ldots, k$$

with $K < 1$ and the iteration converges.

EXAMPLE 26. *We may introduce a parameter x in the last example by defining*

$$y_{i+1} \;=\; g(x,y_i) \;:=\; \cos(xy_i)$$

If $x \in]-1,1[$, we now obtain

$$|g_y(x,y)| \;=\; |-x\sin(xy)| = |x|\,|\sin(xy)| \;\le\; |x| < 1 \quad \forall y \in \mathbb{R},$$

and the iteration converges.

EXAMPLE 27. *Newton's method to solve $f(x,y)=0$:*

$$g(x,y) \;=\; y - f_y(x,y)^{-1} f(x,y).$$

In this case there are several convergence results stating under which assumptions the iterations converge against a root y of $f(x,y)=0$.

EXAMPLE 28. *Time stepping with* explicit *one-step procedure (here: forward Euler), step size τ:*

$$y_{i+1} \;=\; y_k + \tau\phi(x,y_i) \;=:\; g(x,y_i).$$

EXAMPLE 29. *Time stepping with* implicit *one-step procedure (here: backward Euler) and* linear *$\phi(x,y) = A(x)y$:*

$$\begin{aligned} y_{i+1} \;=\; y_i + \tau\phi(x,y_{i+1}) \quad &\Longleftrightarrow \quad y_{i+1} - \tau\phi(x,y_{i+1}) \;=\; y_i \\ &\Longleftrightarrow \quad y_{i+1} \;=\; (I-\tau A(x))^{-1} y_i \;=:\; g(x,y_i). \end{aligned}$$

EXAMPLE 30. *Time stepping with* implicit *one-step procedure (here: backward Euler) and* non-linear *ϕ:*

$$y_{i+1} \;=\; y_i + \tau\phi(x,y_{i+1}) \quad \Longleftrightarrow \quad 0 \;=\; y_{i+1} - \tau\phi(x,y_{i+1}) - y_i \;=:\; f(x,y_{i+1})$$

which can be solved for y_{i+1} by Newton's method, i.e. the whole time stepping procedure consists of two nested iterations (inner: Newton, outer: time).

From now on we consider the following general form of iterative method to solve (10.1):

ALGORITHM 9 (General iterative method).

(1) *Choose initial value y_0.*
(2) *For $i = 0, 1, \ldots$:*
    Compute $y_{i+1} = g(x,y_i)$
    until a stopping criterion $C((y_i)_{i\le k})$ is satisfied (where $k = i+1$).

We here formulated the stopping criterion in a very general way, incorporating the whole trajectory $(y_i)_{j\le k} = (y_0, y_1, \ldots, y_k)$ of iterates. The stopping criterion is a boolean expression which usually includes a given accuracy ε. In the simplest case one only considers the difference between the last two iterates, i.e., $\|y_k - y_{k-1}\|$ in some appropriate norm.

2. Transformed iterations using AD

We now study what the iteration looks like when the forward reverse mode are applied on it. Furthermore, we present some more or less straightforward modifications.

Forward mode. Applying the forward mode directly on the iteration in Algorithm 9, we obtain the following augmented iteration.

ALGORITHM 10 (Iterative method – AD forward mode).

(1) *Choose* y_0, y_0'.
(2) *For* $k = 0, 1, \ldots$:
  (a) *compute* $y_{i+1}' = g_y(x, y_i)y_i' + g_x(x, y_i)x'$,
  (b) *compute* $y_{i+1} = g(x, y_i)$,
  until a stopping criterion $C((y_i)_{i\le k})$ *is satisfied (where* $k = i + 1$*).*

Here the stopping criterion only ensures (numerical) convergence of the values y_k, not of the derivatives y_k'. In the case where the contraction property $\|g_y(x, y)\| = K < 1$ from Banach's theorem is satisfied, this property directly also is valid for the derivative iteration: Banach's Theorem requires the contraction property (3.3), compare page 14, which here for the derivative iteration means

$$\left\|g_y(x, y_i)y_j' + g_x(x, y_i)x' - (g_y(x, y_i)y_l' + g_x(x, y_i)x')\right\| \quad \le \quad K\|y_j' - y_l'\|$$

for arbitrary iterates y_j', y_l'. Since the term with g_x is the same for y_j', y_l', it disappears by the subtraction, and we have only to show

$$\left\|g_y(x, y_i)y_j' - g_y(x, y_i)y_l'\right\| \quad \le \quad K\|y_j' - y_l'\|$$

The two terms on the left in the norm are matrix-vector multiplications with the same matrix g_y. We thus obtain, using submultiplicativity of the norm

$$\left\|g_y(x, y_i)y_j' - g_y(x, y_i)y_l'\right\| \quad \le \quad \|g_y(x, y_i)\|\,\|y_j' - y_l'\|.$$

Thus the derivative iteration is contractive if $\|g_y(x, y)\| = K < 1$ which is the same condition as for the original iteration for y_i. In Banach's theorem, K also determines an upper bound on the convergence speed, but only an upper bound. Thus the convergence speed of both the original iteration (for the y_i) and the one for the derivatives y_i' can have different speeds, and it may be the case that the iteration for the y_i has converged (up to the given accuracy) whereas the one for y_i' has not (yet) – and vice versa. The following example shows that this is necessary:

EXAMPLE 31. *The direct forward mode iteration of Example 26 looks like this:*

(1) *Choose* y_0,
(2) *Set* $x' = 1$ *(we want to compute the derivative w.r.t* x*).*
(3) *Set* $y_0' = 0$ *(initial value does not depend on* x*).*
(4) *For* $i = 0, 1, \ldots$:
  (a) *compute* $y_{i+1}' = -x\sin(xy_i)y_i' - y_i\sin(xy_i)x'$,
  (b) *compute* $y_{i+1} = \cos(xy_i)$,
  until a stopping criterion is satisfied (where $k = i + 1$*).*

We see in Figure 2 that the derivative iteration is a little behind the original one.

Piggyback Iteration. Here the modification of the stopping criterion is applied, the rest remains unchanged.

ALGORITHM 11 (Iterative method – AD piggyback).
As Alg. 10, but with modified stopping criterion $C((y_i)_{i\le k}, (y_i')_{i\le k})$.

Two-Phase Iteration. In the forward mode, one could as well separate both iterations and first let the values y_i converge. Then in the derivative iteration, the (numerically) converged value y_k is used. This leads to the following variant:

ALGORITHM 12 (Iterative method – Two-Phase).

(1) *Use Alg. 9 to compute* $y_k =: y$.
(2) *Choose* y_0'.

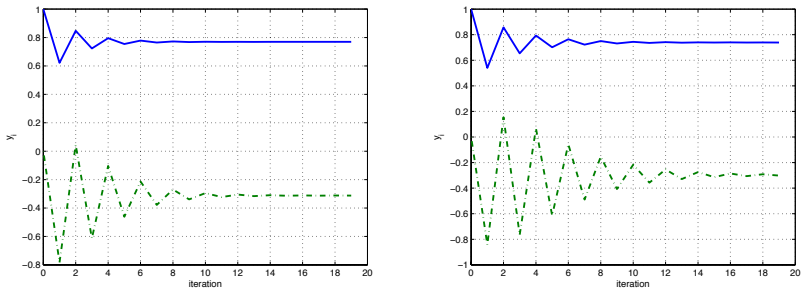

FIGURE 2. Iteration $y_{i+1} = \cos(xy_i)$ with $y_0 = 1, x = 0.9$ (left), $x = 0.9999$ (right) and corresponding derivative iteration (dashed).

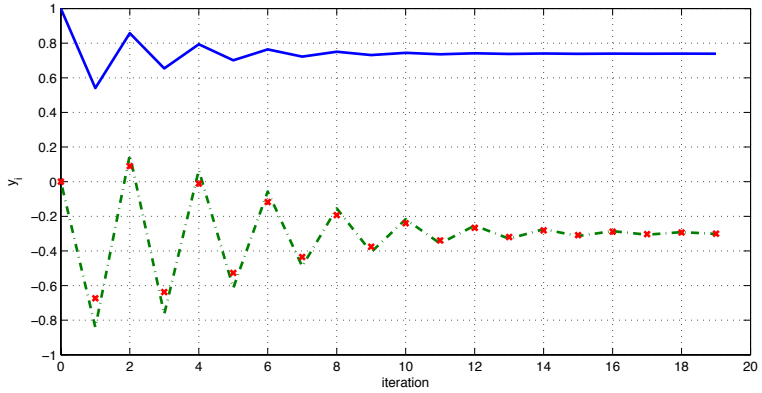

FIGURE 3. Two-phase iteration (Alg. 13, dotted) for $y_{i+1} = \cos(xy_i)$ with $y_0 = 1, x = 0.9999$ and corresponding original and direct forward derivative iteration (Alg. 10, dashed).

(3) *For* $i = 0, 1, \ldots$*:*
  compute $y'_{i+1} = g_y(x, y)y'_i + g_x(x, y)x'$,
until a stopping criterion $C((y'_i)_{i \le k'})$ *is satisfied (where* $k' = i + 1$*).*

The k' of course can be different form the k obtained in the iteration for y_i. The result for Examples 26 and 31 can be seen in Figure 3.

Delayed iteration. Sometimes it may be useful to let the values y_i iterate for some time before starting the derivative iteration. This *delayed iteration* is a mixture of the last two algorithms.

ALGORITHM 13 (Iterative method – delayed iteration).

(1) *Use Alg. 9 to compute* y_k.
(2) *Choose* y'_k.
(3) *For* $i = k + 1, \ldots$*:*
  (a) *compute* $y_{i+1} = g(x, y_i)$,
  (b) *compute* $y'_{i+1} = g_y(x, y_i)y'_i + g_x(x, y_i)x'$,
until a stopping criterion $C((y_i)_{i \le k'}, (y'_i)_{i \le k'})$ *is satisfied (where* $k' = i + 1$*).*

Reverse mode. We now show how a direct reverse mode application looks like for a converging iteration. As discussed above in the chapter about the reverse

mode, this mode of AD is preferable if the dimension of input variable $x \in \mathbb{R}^n$ is much higher than the number of outputs, in this case $y = y_k$, the numerically converged iterate. If y itself is a vector (and thus this condition is not satisfied), the reverse mode is often used in cases where the iteration for y is just a part of a more complex computation, which at the end gives a single value. This case is given in optimization problems and is studied in the next subsection. It requires an appropriate initialization of the adjoint variables. At first, we want to show the reverse mode looks like if only the iteration for y is differentiated.

The following two algorithms show the direct implementation of the reverse mode for the iteration, once with re-computation and the other time with storing of the needed intermediate values y_i. There are two crucial points when writing down this algorithm:

The first one is the correct reverse-mode-differentiation of the step of the original iteration, where we evaluate the function

$$(x, y_i) \quad \mapsto \quad y_{i+1} := g(x, y_i). \tag{10.2}$$

This is a function with two inputs and one output.

Example 32. *The iteration step* (10.2) *is comparable to the intermediate Function F_9 in the Rosenbrock Example 18 on page 51, where the step was*

$$(x_6, x_8) \quad \mapsto \quad x_9 \ := \ F_9(x_6, x_8) \ = \ x_6 + x_8$$

and the corresponding reverse mode step according to the algorithm (8.5) *on page 51 was*

$$\begin{aligned} \bar{x}_8 &= \bar{x}_8 + \bar{x}_9 \frac{\partial F_9}{\partial x_8}(x_6, x_8) \\ \bar{x}_6 &= \bar{x}_6 + \bar{x}_9 \frac{\partial F_9}{\partial x_6}(x_6, x_8). \end{aligned}$$

Applying the reverse mode algorithm (8.5) similarly to (10.2), the corresponding step is

$$\begin{aligned} \bar{y}_i &= \bar{y}_i + \bar{y}_{i+1} g_y(x, y_i) \\ \bar{x} &= \bar{x} + \bar{y}_{i+1} g_x(x, y_i). \end{aligned}$$

These are steps (4b-c) in the following algorithm.

The initialization of the adjoint variables is the second delicate point in this algorithm: Since the iteration computes $y = y_k = F(x)$, we have to initialize $\bar{y}_k = 1$ or – if more generally $y \in \mathbb{R}^m$ – by the identity matrix. As a result, we obtain the following

Algorithm 14 (Iterative method – AD reverse mode, recomputation).

(1) *Use Alg. 9 to compute y_k.*
(2) *Initialize the adjoint variable $\bar{y}_k$ accordingly.*
(3) *Initialize the adjoint variables $\bar{y}_i = 0, i = k-1, \ldots, 0,$ and $\bar{x} = 0$.*
(4) *For $j = k-1, \ldots 0$:*
  (a) *For $i = 0, \ldots, j-1$:*
    compute $y_{i+1} = g(x, y_i)$ *(Recomputation)*
  (b) *Update $\bar{y}_j = \bar{y}_j + \bar{y}_{j+1} g_y(x, y_j)$*
  (c) *Update $\bar{x} = \bar{x} + \bar{y}_{j+1} g_x(x, y_j)$.*

Steps (4b,c) remain valid if $y_j \in \mathbb{R}^m$ is a vector – if it is regarded as a row vector. If y_j is regarded as a column vector, one has to write

(4) (b) Update $\bar{y}_j = \bar{y}_j + g_y(x, y_j)^\top \bar{y}_{j+1}$
  (c) Update $\bar{x} = \bar{x} + g_x(x, y_j)^\top \bar{y}_{j+1}$.

The extensive recomputations can be replaced by storing the whole trajectory. This results in the following changes:

ALGORITHM 15 (Iterative method – AD reverse mode, storing).

(1) *Use Alg. 9 to compute* $y_1, \ldots, y_k$ *and store them.*
(2–3) *as above.*
(4) *For* $j = k-1, \ldots 0$*:*
  (a) *retrieve* y_j
  (b–c) *as above.*

If just the converged iterate is differentiated w.r.t. x, then $\bar{y}_k$ has to be initialized by 1 (or if y_k is a vector by the identity matrix). If – as in the example at the beginning of this subsection – the converged iterate $y = y_k$ is used in a cost function F, then the correct initialization of $\bar{y}_k$ depends on the dependency of F on y_k.

It becomes clear that there may be a huge effort of storage or re-compuations since in both algorithms the whole trajectory $(y_i)_{i=0,\ldots,k}$ is needed. The following modification by [**Chr98**] avoids this. It uses the converged iterate $y0y_k$ throughout the reverse mode computations:

ALGORITHM 16 (Iterative method – AD reverse mode, Christianson).

(1) *Use Alg. 9 to compute* $y = y_k$*.*
(2–3) *as above.*
(4) *For* $j = k-1, \ldots 0$*:*
  (a) *Update* $\bar{y}_j = \bar{y}_j + \bar{y}_{j+1} g_y(x, y)$
  (b) *Update* $\bar{x} = \bar{x} + \bar{y}_{j+1} g_x(x, y)$.

Again the last two lines refer to a row vector and have to be re-written as above for a column vector.

We compute the direct reverse mode (with storing, Alg. 15) implementation with Chirstianson's version for the iteration from Examples 26 and 31. Of course here the reverse mode makes not much sense since we have only a single input x.

EXAMPLE 33. *The direct reverse mode iteration for Example 26 looks like this:*

(1) (a) *Choose* y_0*.*
  (b) *For* $i = 0, 1, \ldots$*:*
    compute $y_{i+1} = \cos(xy_i)$*,*
    until a stopping criterion is satisfied.
  (c) *Set* $k = i + 1$*.*
(2) *Initialize the adjoint variable* $\bar{y}_k = 1$*.*
(3) *Initialize the adjoint variables* $\bar{y}_i = 0, i = k-1, \ldots, 0$*, and* $\bar{x} = 0$*.*
(4) *For* $j = k-1, \ldots 0$*:*
  (b) *Update* $\bar{y}_j = \bar{y}_j - \bar{y}_{j+1} x \sin(xy_j)$
  (c) *Update* $\bar{x} = \bar{x} - \bar{y}_{j+1} y_j \sin(xy_j)$

We see in Figure 4 that the derivative iteration is a little behind the original one.

Iteration as part of a function evaluation. Differentiating an iteration using the reverse mode is usually important if at the end a single real-valued function – depending on the converged output $y = y_k$ of the iteration – is computed. Before we study this case, we This is for example the case in optimization problems, where x are the parameters or variables to be optimized and a single-valued cost function (here denoted by F) that depends on y has to be minimized or maximized. If the iterates y_i and thus the converged solution $y = y_k$ depend on x, then the cost function F depends on x via the converged iterate $y = y_k$, i.e. $F = F(y) = F(y(x))$. A derivative computation applied on the cost function using the reverse mode of AD will then apply the reverse mode also on the iteration for the y_i. Then the overall code to which the reverse mode is applied looks like this:

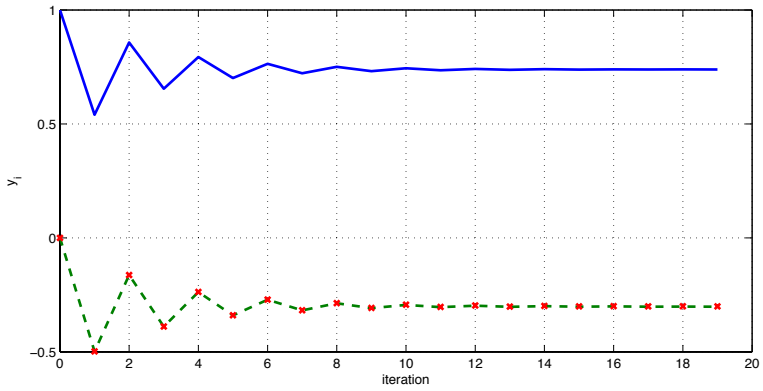

FIGURE 4. Iteration $y_{i+1} = \cos(xy_i)$ with $y_0 = 1, x = 0.9999$ (solid) and corresponding values of x' in the derivative iterations by direct reverse mode (dashed) and Christianson approach ($\times$).

ALGORITHM 17 (Sketch for iterative method used in optimization algorithm).

(1) *Chose parameter vector* x.
(2) *Compute converged iterate* $y = y_k$ *by applying Alg. 9.*
(3) *Evaluate* $w = F(y)$.

A typical example is that the aim of the optimization is to adjust the parameters x in such a way that the output $y = y_k$ of the iteration is as close as possible to some given *target* denoted by y_{target}. This can be achieve by using the cost functional

$$F(y) \quad = \quad \frac{1}{2}\|y - y_{target}\|_2^2 \quad = \quad \frac{1}{2}(y - y_{target})^\top (y - y_{target}),$$

where $\|\cdot\|_2$ denotes the Euclidean norm

$$\|w\|_2 \quad := \quad \sqrt{\sum_{i=1}^{m} w_i^2} \quad = \quad \sqrt{w^\top w}, \quad w = (v_i)_{i=1,\ldots,m} \in \mathbb{R}^m,$$

Minimizing F now results in moving the output $y = y_k$ of the iteration towards the target, i.e., $y = y_k \approx y_{target}$. Then we have for the directional derivative w.r.t. y in direction v, compare Definition 2 on page 9:

$$\begin{aligned}
F'(y)v \quad &:= \quad \lim_{t \to 0+} \frac{F(y + tv) - F(y)}{t} \\
&= \quad \lim_{t \to 0+} \frac{(y + tv - y_{target})^\top (y + tv - y_{target}) - (y - y_{target})^\top (y - y_{target})}{2t} \\
&= \quad \lim_{t \to 0+} \frac{2t(y - y_{target})^\top v + t^2 v^\top v}{2t} \\
&= \quad (y - y_{target})^\top v.
\end{aligned}$$

(This explains the factor $\frac{1}{2}$ in our choice of F.) Now the reverse mode for the code line

$$w \; := \; F(y_k) \quad = \quad \frac{1}{2}(y_k - y_{target})^\top (y_k - y_{target})$$

gives

$$\bar{y}_k \quad = \quad \bar{y}_k + \bar{w}^\top (y_k - y_{target}).$$

Here $\bar{w}$ is the adjoint of the final output, i.e. the cost function. As seen above in the reverse mode this variable should be initialized by 1, compare (8.5) on page 51. Since the adjoint of the intermediate y_k is initialized by zero when applying the reverse mode on the whole Alg. 17 (compare again the reverse mode algorithm in (8.5)), we obtain

$$\bar{y}_k \quad = \quad y_k - y_{target}.$$

This then has to be used in the above algorithms.

Full Jacobian method. This approach is another alternative. Here we directly differentiate the fixed-point equation

$$y \quad = \quad g(x, y(x)) \tag{10.3}$$

which is satisfied by the converged value $y = y_k$, with respect to x. Since the converged value $y = y_k$ depends on x, we write $y = y(x)$, and are looking for an equation for the derivative $y'(x)$, which with the above dimensions $y \in \mathbb{R}^m$ and $x \in \mathbb{R}^n$ is a matrix in $\mathbb{R}^{m\times n}$. Differentiating (10.3), obtain the following equation

$$g_x(x, y) + \big[g_y(x, y) - I\big]y'(x) \quad = \quad 0$$

(where $I \in \mathbb{R}^{m\times m}$ is the identity matrix) for the desired derivative $y'(x)$. Thus $y'(x)$ can be computed by solving the linear system

$$\big[g_y(x, y) - I\big]y'(x) \quad = \quad -g_x(x, y). \tag{10.4}$$

Note that the unknown here is a matrix, namely $y'(x) \in \mathbb{R}^{m\times n}$. Thus, solving (10.4) is equivalent to solving m linear systems with the same matrix $\big[g_y(x, y)-I\big] \in \mathbb{R}^{m\times m}$ and n right-hand sides, summarized in the matrix $(-g_x) \in \mathbb{R}^{m\times n}$.

Since g_y is the Jacobian of the mapping $y \mapsto g(x, y)$, this method is called the *full Jacobian approach*. We now obtain the following method. Here we assume that if x, y are known, the partial derivative $g_x(x, y)$ can be computed explicitly.

Algorithm 18 (Full Jacobian method).

(1) *Use Alg. 9 to compute* $y = y_k$.
(2) *Compute* $A := g_y(x, y) - I, b := -g_x(x, y)$.
(3) *Initialize* y'_0.
(4) *Solve* $\big[g_y(x, y) - I\big]y'(x) = -g_x(x, y)$, *which means:*

For i=1,. . . ,n:

$$\textit{Solve} \quad \big[g_y(x, y) - I\big]\frac{\partial y}{\partial x_i}(x) = -\frac{\partial g}{\partial x_i}(x, y)$$

The last steps makes clear that for every partial derivative $\dfrac{\partial y}{x_i}(x)$ one linear system (of size $m \times m$) has to be solved. This leads to an total effort of $\mathcal{O}(m^3)$ for a factorization of the system matrix factorization and $\mathcal{O}(nm^2)$ for the solution. The linear system (10.4) can be quite big, and thus it may be useful to solve it iteratively thus avoiding the storing of the full Jacobi matrix.

Example 34. *For the simple iteration from Examples 26, 31, and 33, the full Jacobian approach is very simple since* $m = n = 1$, *i.e., there are no matrices, but just numbers. One easily computes*

$$\begin{aligned} g_x(x, y) &= -y\sin(xy) \\ g_y(x, y) &= -x\sin(xy) \end{aligned}$$

and thus

$$y'(x) \quad = \quad -\frac{g_x(x, y)}{g_y(x, y) - 1} \quad = \quad -\frac{y\sin(xy)}{x\sin(xy) + 1}.$$

CHAPTER 11

Sparsity Patterns

In the general case $F = (F^i)_{i=1,\ldots,m} : \mathbb{R}^n \to \mathbb{R}^m$ the derivative of F with respect to $x = (x_j)_{j=1,\ldots,n}$ is the Jacobian matrix

$$F'(x) \quad = \quad \left(\frac{\partial F^i}{\partial x_j}(x)\right)_{i=1,\ldots,m,j=1,\ldots,n} \in \mathbb{R}^{m\times n}.$$

Here F^i denotes the i-th component of the vector-valued function F and should not to be mixed with F_i which denotes the i-th intermediate function in the computational graph of F.

Each row of the Jacobian of F is the gradient of the component function F^i, i.e. we can also write

$$F'(x) \quad = \quad \begin{bmatrix} \frac{\partial F^1}{\partial x_1}(x) & \cdots & \frac{\partial F^1}{\partial x_n}(x) \\ \vdots & & \vdots \\ \frac{\partial F^m}{\partial x_1}(x) & \cdots & \frac{\partial F^m}{\partial x_n}(x) \end{bmatrix} = \begin{bmatrix} \cdots & \nabla F^1(x) & \cdots \\ & \vdots & \\ \cdots & \nabla F^m(x) & \cdots \end{bmatrix}.$$

In our notation in the Algorithm of the forward mode (compare (5.8) on page 35 and the remark below concerning vector-valued functions) the output of F in the vector-valued case was denoted by

$$y = (y_1, \ldots, y_m) = (x_{n+k+1}, \ldots, x_{n+k+m}).$$

The Jacobian $F'(x)$ can then be written – using the intermediates x'_i for the derivatives – as

$$F'(x) \quad = \quad \begin{bmatrix} \nabla F^1(x) \\ \vdots \\ \nabla F^m(x) \end{bmatrix} = \begin{bmatrix} x'_{n+k+1} \\ \vdots \\ x'_{n+k+m} \end{bmatrix}.$$

If $n \approx m$ and the derivative (Jacobian) is computed by the forward mode of AD, then the computational effort is at least $\mathcal{O}(m \cdot n)$. The same is true when finite difference approximations are used. In many cases the Jacobian is not *dense*, i.e., not all elements are non-zeros. In this case and especially if m, n are big, it is very important to reduce the number of necessary operations exploiting the *sparsity pattern* of the Jacobian matrix.

DEFINITION 15. *The sparsity pattern of a matrix is the set of index pairs where the elements of a matrix are non-zero.*

A sparsity pattern can be described as a matrix with entries in $\{0, 1\}$ only, or visually as a matrix with symbols as stars only for the non-zero entries, seethe next example below.

The idea in exploiting the sparsity pattern is to chose an appropriate seed matrix $x' = S \in \mathbb{R}^{n\times p}$ with $p < n$ that still allows to compute the whole Jacobian when applying the forward mode. Recall that in order to compute the full Jacobian $F'(x)$, the forward mode algorithm is initialized with a seed matrix $S = x' = I \in$

$\mathbb{R}^{n\times n}$, I being the identity matrix. Taking $S = s \in \mathbb{R}^{n\times 1} \cong \mathbb{R}^n$, the forward mode will give at the end $y' = F'(x)s \in \mathbb{R}^n$, i.e. the directional derivative in direction of the vector s. Thus the idea is to chose a number p of vectors s_i, combine them as columns in the seed matrix $S \in \mathbb{R}^{n\times p}$, such that the full Jacobian can be extracted form the end product $y' \in \mathbb{R}^{n\times p}$ of the forward mode. Two questions arise:

- How to get information about the sparsity pattern?
- How to determine an optimal choice of directions s_i that can be used in the seed matrix S?

1. Getting information about the sparsity pattern

In many problems there is a-priori information about the sparsity pattern of the Jacobian since it is determined by the underlying problem that is to be solved. This for example is the case the Jacobian is used in Newton's method and the underlying problem is a partial differential equation:

EXAMPLE 35. *A one-dimensional model for fluid flow is the non-linear so-called* Burgers equation *which reads*

$$-u'' + uu' \quad = \quad f$$

where $u = u(x)$ is the unknown velocity of the fluid, x the spatial coordinate, and f an given external force. The problem is studied for $x \in [0, 1]$, and at both boundaries ($x = 0, x = 1$) the value $u = 0$ is given. To solve the problem, a discretization scheme is applied, which now leads to a non-linear equation

$$(11.5)\ F_i(u) \ := \ -\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + u_i \frac{u_{i+1} - u_{i-1}}{2h} - f_i \quad = \quad 0, \ \ i = 1, \ldots, N.$$

Here $u = (u_i)_{i=1,\ldots,N}$ with $u_i \approx u(x_i)$ represents the vector with the approximation of the solution at the grid points $x_i = ih, i = 1, \ldots, N$, where h is a spatial step-size.

To compute the solution of the non-linear system $F(u) = 0$ where the i-th equation of the system is given by (11.5), Newton's method can be used, which requires the Jacobian $F'(u)$. But since the discretization scheme used in (11.5) is standard, the structure or sparsity pattern of the Jacobian can be easily obtained as

$$F'(u) \quad = \quad \begin{bmatrix} * & * & & \\ * & * & * & \\ & * & * & * \\ & & * & * \end{bmatrix}$$

where the stars denote non-zero entries and the zero entries are omitted. This can be seen form the fact that in (11.5) only unknowns with indices $i, i-1$ and $i, i+1$ are used. Thus the Jacobian only contains entries on the diagonal and the first sub- an super diagonal.

Similar a-priori information is available anytime when a problem comes from the discretization of a differential equation and the used discretization scheme is known.

Automatic sparsity pattern detection. If there is no a-priori information about the sparsity pattern is available or it shall not be used, also AD tools can be used to detect the sparsity pattern. Basically are sparsity pattern detection is similar to a derivative computation, with the difference that *the values* of the intermediates x_i' (here for the forward mode) are not important, it is just important if they are zero or not. In other words: if there is a dependency of the intermediate value $x_i, i \geq n$, on the inputs $x = (x_i)_{i=1,\ldots,n}$.

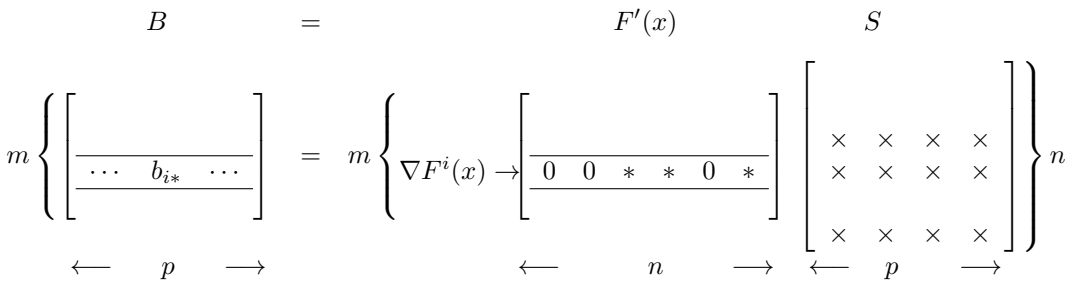

FIGURE 1. Jacobian $F'(x)$ times seed matrix S for a sparse Jacobian. Only the elements of S marked $\times$ are needed to compute the row b_{i*} of the result $B = F'(x)S$.

The real-valued intermediates x'_i storing the derivatives in the forward mode can be replaced by boolean variables $\hat{x}_i$. The statement in Algorithm (5.8) on page 35 that updates the derivatives,

$$x'_i \;=\; \sum_{j \prec i} \frac{\partial F_i((x_l)_{l \prec i})}{\partial x_j} x'_j, \quad i = n+1, \ldots, n+k,$$

can now be simplified since no actual derivatives $\frac{\partial F_i}{\partial x_j}$ have to be computed. Instead the boolean variables $\hat{x}_i$ are updated according to some simple rules:

- If F_i is a binary operation $F_i : x_i = x_{j_1} \circ x_{j_2}$ with $\circ \in \{+, -, *, /, **\}$ (here $**$ denoting the exponential function: $x * *y = x^y$) or a function of two arguments:
$$\hat{x}_i := \hat{x}_{j_1} \vee \hat{x}_{j_2}.$$
- If F_i is a unary operation $F_i : x_i = \circ x_j$ with $\circ \in \{+, -\}$ or a function of one argument as $F_i(x_j) = \sin(x_j)$:
$$\hat{x}_i := \hat{x}_j.$$

Similar rule can be derived for the reverse mode.

2. Exploiting a sparsity pattern for Jacobian computation

If for $F : \mathbb{R}^{m \times n}$ the forward mode is applied with a seed matrix

$$x' = S \in \mathbb{R}^{n \times p}, \quad p < n,$$

then the resulting output matrix satisfies

$$y' = \underbrace{F'(x)}_{\in \mathbb{R}^{m \times n}} \cdot \underbrace{S}_{\in \mathbb{R}^{n \times p}} =: \underbrace{B}_{\in \mathbb{R}^{m \times p}}.$$

We recall that each row of the Jacobian of F is the gradient of the component function F_i (see beginning of the chapter). Then the i-th row of B, denoted here by

$$b_{i*} := (b_{ij})_{j=1,\ldots,p} = e_i^\top B,$$

satisfies

$$b_{i*} \;=\; e_i^\top B \;=\; \underbrace{e_i^\top F'(x)}_{=\nabla F^i(x) = x'_{n+k+i}} \quad S \;=\; \nabla F^i(x) S \tag{11.6}$$

and is determined by the gradient of F^i only. Here e_i is the i-the unit vector as a column vector, thus $e_i^\top$ is a row vector. The above relation can be seen in Figure 1.

Necessary dimension of S. The length p of the row of B can be chosen, and our aim is to keep it as small as possible but still be able to extract all necessary information about the Jacobian $F'(x)$ out of the matrix B. In the picture, the columns of S (on the right) have to be long enough such that all non-zeros in every row of $F'(x)$ (i.e., all gradients $x'_{n+k+i} = \nabla F^i(x), i = 1, \ldots, m$) are multiplied by some entry (depicted here in one column as stars) of the seed matrix S.

From these computations it can be seen that we need at least as much entries in b_{i*}, i.e. the i-th row of B, as we have non-zeros in the gradient $x'_{n+k+i} = \nabla F^i(x)$ of the i-th component function of F. Since all rows of B have the same length p, we end up with

$$p \geq nnz(\nabla F^i(x)) \quad \forall i = 1, \ldots, m,$$

where nnz denotes the numbers of non-zeros (here in the gradient vector $\nabla F^i(x)$).

This means that we need at least a length p of the rows (i.e., number of columns) in B of

$$p := \max\{nnz(\nabla F^i(x)) : i = 1, \ldots, m\}.$$

Computing the non-zeros of the Jacobian. To compute all non-zeros of $F'(x)$, we consider again (11.6):

$$b_{i*} \;=\; e_i^\top F'(x) S \;=\; x'_{n+k+i} S.$$

The vector x'_{n+k+i} obtained by the vector-matrix product

$$x'_{n+k+i} \;=\; e_i^\top F'(x) \tag{11.7}$$

has non-zeros only in the entries $j \in \{1, \ldots, n\}$ (these correspond to the independent/ input variables $x_1, \ldots x_n$), but only if these are predecessors of x_{n+k+i}, i.e., if x_{n+k+i} depends on them and thus the partial derivative

$$\frac{\partial F^i}{\partial x_j}(x), \quad j \in \{1, \ldots, n\},$$

is not zero. We denote the set of the indices of these nonzero entries by

$$\chi_i := \left\{ j \in \{1, \ldots, n\} : \frac{\partial F^i}{\partial x_j}(x) \neq 0 \right\}, p_i := |\chi_i|.$$

To compute (11.7), we thus only need the columns with indices $j \in \chi_i$ of the Jacobian. These can be written as the Jacobian-vector product $F'(x)e_j$. This means

$$x'_{n+k+i} \;=\; e_i^\top F'(x) \;=\; \sum_{j \in \chi_i} e_i^\top F'(x) e_j \;=\; \sum_{j \in \chi_i} \frac{\partial F^i}{\partial x_j}(x)$$

and, see (11.6):

$$b_{i*} \;=\; e_i^\top F'(x) S \;=\; \sum_{j \in \chi_i} e_i^\top F'(x) e_j S \;=\; a_{i*} S_i \tag{11.8}$$

where

$$a_{i*} \;:=\; \left(\frac{\partial F^i}{\partial x_j}(x) \right)_{j \in \chi_i} \in \mathbb{R}^{p_i} \quad \text{(the non-zeros of } x'_{n+k+i} = \nabla F^i(x) \text{ as row vector)}$$

and

$$S_i \;:=\; \left(e_j^\top S \right)_{j \in \chi_i} \in \mathbb{R}^{p_i \times p} \quad \text{(only the needed rows of the seed matrix } S, \text{ i.e., the submatrix marked by } \times \text{ in Fig. 1).}$$

To determine $a_{i*} \in \mathbb{R}^{p_i}$, i.e., the non-zeros of the i-th row of the Jacobian, we have to solve the overdetermined system

$$b_{i*} = a_{i*} S_i \iff S_i^\top a_{i*}^\top = b_{i*}^\top,$$

see (11.8). To obtain all non-zeros of the Jacobian, m such systems have to be solved.

3. Choice of the appropriate seed matrix

The main task is to find an appropriate seed matrix with minimal number of columns p.

Curtis-Powell-Reed-Coloring.

Bibliography

[Alt02] Walter Alt. *Nichtlineare Optimierung. Eine Einführung in Theorie, Verfahren und Anwendungen.* Vieweg, August 2002.

[Chr98] B. Christianson. Reverse accumulation and implicit functions. *Optimization Methods and Software*, 9(4):307–322, 1998.

[Deu04] P. Deuflhard. *Newton Methods for Nonlinear Problems.* Springer, 2004.

[Gri89] A. Griewank. On Automatic Differentiation. In Masao Iri and Kunio Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[IT79] A.D. Ioffe and V.M. Tichomirov. *Theorie der Extremalaufgaben.* VEB Deutscher Verlag der Wissenschaften, 1979.

[Jah94] J. Jahn. *Introduction to the Theory of Nonlinear Optimization.* Springer, Berlin, 1994.

[Kun] K. Kunisch. *Vorlesungsskript: Nonlinear Optimization.* Karl-Franzens-Universit"at Graz, "Osterreich.

[Lue69] D.G. Luenberger. *Optimization by Vector Space Methods.* Wiley, 1969.

[Lue08] D. G. Luenberger. *Linear and Nonlinear Programming.* Springer, New York, 3rd edition, 2008.

[Mat00] The Mathworks, Inc., Natick, MA, USA. *Optimization Toolbox For Use with* MATLAB *User's Guide Version 2*, 2000.

[MH90] M. G. Morgan and M. Henrion. *Uncertainty – A Guide to Dealing with Uncertainty in Quantitative Risk and Policy Analysis.* Cambridge University Press, 1990.

[Ruz04] M. Ruzicka. *Nichtlineare Funktionalanalysis.* Springer, 2004.

[SZ04] T. Slawig and K. Zickfeld. Parameter optimization using algorithmic differentiation in a reduced-form model of the Atlantic thermohaline circulation. *Nonlinear Analysis: Real World Applications*, 5/3:501–518, 2004.