ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Literature Review of High Order ODE Solvers

Semester thesis

Niklas Pfister

October 4, 2015

Advisor: Prof. Dr. Arnulf Jentzen

Department of Mathematics, ETH Zürich

Contents

Chapter 1

Introduction

The purpose of this thesis is to take a look at several available options for constructing high order solvers for ordinary differential equations and study their behaviour for small numbers of time steps (between 1 and 10 time steps). We focus on the practical aspect of implementing these methods in a way that allows the construction of methods with arbitrarily high orders. Of course this is only theoretically possible due to computational complexity. The implementations are in Matlab and all codes are directly included into the thesis.

The paper has been divided into four main parts. The first three consider three different types of methods: Taylor methods, Runge-Kutta methods and extrapolation methods. These are not intended as self-contained introductions to these methods, but rather as short overviews giving the necessary background required to be able to understand the implementations outlined. The last chapter applies the methods to a few examples and intends to illustrate the behaviour of increasing orders of the different methods for small numbers of time steps.

The following section outlines the mathematical setting that is used throughout the rest of the thesis.

1.1 Setting

Throughout this paper let $d \in \mathbb{N}$, $t_0 \in [0, \infty)$, $T \in (t_0, \infty)$, $y_0 \in \mathbb{R}^d$ and $f \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$ and concentrate on ordinary differential equations (ODEs) of the form

$$\begin{cases} y'(t) = f(y(t)), & t \in [t_0, T] \\ y(t_0) = y_0. \end{cases} \tag{1.1}$$

Using the fundamental theorem of calculus any solution of the ODE (1.1) also has to satisfy the following integral equation

$$y(t) = y_0 + \int_{t_0}^{t} f(y(s))ds, \quad t \in [t_0, T]. \tag{1.2}$$

Existence and uniqueness of solutions is guaranteed by the famous Picard–Lindelöf theorem which can be found in any book on ordinary differential equations (see, e.g., [8]).

The assumption that the right hand side f is infinitely often differentiable is a lot more restrictive than necessary. However, the purpose of this paper is only to illustrate high order numerical approximations and using this rather basic setting allows us to circumvent many analytical details and therefore allows us to focus on the fundamental ideas behind the methods.

Also observe that since f has no time dependence we often assume $t_0 = 0$ as this has no effect on the theory.

1.2 Numerical approximations

In this section we introduce some standard notation related to numerical approximations of ODEs. We follow [6] and also refer to this book for more details.

Assume we are in the setting described in Section 1.1. We call $\triangle$ a grid on $[t_0, T]$ if it is a set of discrete times $\triangle := \{t_0, t_1, \dots, t_n\}$ satisfying

$$t_0 < t_1 < \cdots < t_n = T. \tag{1.3}$$

The idea behind numerical approximations of ODEs is to find a function $y_\triangle : \triangle \to \mathbb{R}^d$ such that

$$y_\triangle(t) \approx y(t), \quad \text{for all } t \in \triangle \tag{1.4}$$

where y is the exact solution of the ODE (1.1). A method that calculates such an approximation $y_\triangle$ for all grids $\triangle$ on $[t_0, T]$ is called a discretization method. If a discretization method additionally fulfills that for all grids $\triangle$ on $[t_0, T]$ the approximation $y_\triangle$ satisfies the two term recursion

(i) $y_\triangle(t_0) = y_0$,

(ii) $y_\triangle(t_{j+1}) = \Psi\left(t_{j+1} - t_j, y_\triangle(t_j)\right)$ for all $j = 0, 1, \dots, n-1$

for some function $\Psi : [0, T - t_0] \times \mathbb{R}^d \to \mathbb{R}^d$ which is independent of $\triangle$, then we call it a one-step method. The function Ψ is called discrete evolution. In

view of this and using that in the setting of Section 1.1 the function f is independent of the time variable t, a one-step method is uniquely determined by specifying $\Psi(t, y)$ for all $y \in \mathbb{R}^d$ and for all $t \in [t_0, T]$. Therefore in order to ease the notation we often write y_h to represent $\Psi(h, y_0)$.

Chapter 2

Taylor methods

Taylor methods are an intuitive collection of one-step methods for solving ordinary differential equations. They can be constructed for arbitrary orders of convergence and play a crucial role in deriving the famous Runge-Kutta methods. However, they have two severe drawbacks when it comes to practical applications.

(i) A Taylor method of order m requires all derivatives of the right hand side up to order $m-1$.

(ii) Taylor methods become complicated for higher orders, in particular for multi-dimensional right hand sides.

Despite these issues and due to their intuitive construction, Taylor methods are an important class of methods also in practise.

In the following sections we derive the general form of the m-th order Taylor method. For more details see [6].

2.1 Deriving Taylor methods

The main idea behind Taylor methods is to iteratively apply the fundamental theorem of calculus and the chain rule to step by step achieve better approximations of the solution $y(t)$ around the value t_0.

Starting with the integral form (1.2) and applying the fundamental theorem

of calculus to $f(y(s))$ inside the integral we get

$$\begin{aligned} y(t) &= y_0 + \int_{t_0}^{t} f(y(s))ds \\ &= y_0 + \int_{t_0}^{t} \left(f(y(t_0)) + \int_{t_0}^{s_0} f'(y(s_1))[y'(s_1)]ds_1 \right) ds_0 \\ &= y_0 + (t-t_0)f(y_0) + \int_{t_0}^{t}\int_{t_0}^{s_0} f'(y(s_1))[f(y(s_1))]ds_1 ds_0 \end{aligned} \tag{2.1}$$

Set $h := t - t_0$ and observe that due to the continuity of f' the integral term is of order h^2. We take the first part, which only depends on the initial value, as the first order Taylor method (which corresponds to the famous explicit Euler method)

$$y_h = y_0 + hf(y_0) \tag{2.2}$$

To keep the notation as simple as possible we make the following definitions.

Definition 2.1 (L-operator) *Let $d \in \mathbb{N}$ and $f \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$. Then we denote by $L_f : C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right) \to C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$ the function with the property that for all $g \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$, $x \in \mathbb{R}^d$, it holds that*

$$L_f g(x) = g'(x)[f(x)] \tag{2.3}$$

and we call L_f the L-operator associated to the ODE (1.1)*.*

Definition 2.2 (m-th L-operator) *Let $d, m \in \mathbb{N}$ and $f \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$. Then we denote by $L_f^{(m)} : C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right) \to C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$ the function with the property that for all $g \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$ it holds that*

$$L_f^{(m)} g = L_f^m g. \tag{2.4}$$

and we call $L_f^{(m)}$ the m-th L-operator associated to the ODE (1.1)*.*

The first three L-operators (x argument is left out) are given by

$$\begin{aligned} L_f^{(1)} f &= f^{(1)}[f] \\ L_f^{(2)} f &= f^{(2)}[f,f] + f^{(1)}\left[f^{(1)}[f]\right] \\ L_f^{(3)} f &= f^{(3)}[f,f,f] + 3f^{(2)}\left[f, f^{(1)}[f]\right] + f^{(1)}\left[f^{(2)}[f,f]\right] \\ &\quad + f^{(1)}\left[f^{(1)}\left[f^{(1)}[f]\right]\right] \end{aligned}$$

We can now express (2.1) using the L-operator and apply the fundamental theorem of calculus once more to the term inside the integral.

$$
\begin{aligned}
y(t) &= y_0 + hf(y_0) + \int_{t_0}^{t}\int_{t_0}^{s_0} f'(y(s_1))[f(y(s_1))]ds_1 ds_0 \\
&= y_0 + hf(y_0) + \int_{t_0}^{t}\int_{t_0}^{s_0} (L_f f)(y(s_1))ds_1 ds_0 \\
&= y_0 + hf(y_0) + \int_{t_0}^{t}\int_{t_0}^{s_0} \left(L_f f(y(t_0)) + \int_{t_0}^{s_1} L_f\big((L_f f)(y(s_2))\big)ds_2 \right) ds_1 ds_0 \\
&= y_0 + hf(y_0) + \frac{h^2}{2}\big(L_f^{(1)}\big)f(y_0) + \int_{t_0}^{t}\int_{t_0}^{s_0}\int_{t_0}^{s_1} \big(L_f^{(2)} f\big)(y(s_2))ds_2 ds_1 ds_0
\end{aligned}
$$

Now the integral term is of order h^3 and we take the second order Taylor method as

$$y_h = y_0 + hf(y_0) + \frac{h^2}{2} L_f^{(1)} f(y_0)$$

Continuing this process iteratively we end up with the general Taylor expansion of the solution.

Theorem 2.3 (general Taylor expansion) *Assume the setting in Section 1.1. Then it holds for all $h \in [0, T - t_0]$ that*

$$
\begin{aligned}
y(t_0 + h) = &\underbrace{y_0 + hf(y_0) + \cdots + \frac{h^m}{m!} L_f^{(m-1)} f(y_0)}_{\textit{Taylor approximation}} \\
&+ \underbrace{\int_{t_0}^{t_0+h} \cdots \int_{t_0}^{s_{m-1}} L_f^{(m)} f(y(s_m))ds_m \cdots ds_0}_{\textit{remainder term}}
\end{aligned}
$$

Proof Follows immediately by induction from the considerations above. □

The remainder term can be estimated to be of order h^{m+1}. This is done in the following theorem.

Theorem 2.4 (remainder estimate) *Assume the setting in Section 1.1. Then it holds for all $h \in [0, T - t_0]$ that*

$$
\begin{aligned}
&\left\| \int_{t_0}^{T} \cdots \int_{t_0}^{s_{m-1}} L_f^{(m)} f(y(s_m))ds_m \cdots ds_0 \right\|_{\mathbb{R}^n} \\
&\quad \leq \frac{h^{m+1}}{(m+1)!} \sup_{s\in[t_0,T]} \left\| L_f^{(m)} f(y(s)) \right\|_{\mathbb{R}^n} < \infty
\end{aligned}
$$

Proof Observe that $L_f^{(m)} f(y(s_m))$ is continuous and hence attains its supremum on $[t_0, T]$. □

We can therefore define the m-th order Taylor methods.

Definition 2.5 (m-th order Taylor method) *Assume the setting in Section 1.1 and let $h \in (0, \infty)$. Then we define the Taylor method of order m by*

$$y_h = y_0 + hf(y_0) + \frac{h^2}{2} L_f^{(1)} f(y_0) + \cdots + \frac{h^m}{m!} L_f^{(m-1)} f(y_0)$$

Remark 2.6 *Observe that Theorem 2.4 ensures that the m-th order Taylor method indeed has order of convergence m.*

If the right hand side is a one dimensional polynomial, it is possible to implement the Taylor methods very efficiently using the internal polynomial functions in Matlab. Such an implementation is given in Listing 2.1.

Listing 2.1: Matlab implemention of the Taylor methods for polynomial rhs

```
1
function [ y_T ] = poly_taylor( poly_rhs, T, y_0, steps, order )
3 % poly_taylor: Taylor method ODE-solver with arbirary order for
    polynomial rhs
% INPUT
5 % poly_rhs: polynomial rhs given as vector
% T:        final time
7 % y_0:      initial value
% steps:    number of time steps
9 % order:    order of ODE-solver
% OUTPUT
11 % y_T:      approximated solution at time T

13 %% Calculate required derivates
n=length(poly_rhs)-1;
15 mpl=2^(order-1)*(n-1)+2;
polymat=zeros(order,mpl);
17 polymat(1,(mpl-n):mpl)=poly_rhs;
for k=2:order
19     cpd=2^(k-1)*(n-1)+1;
    temp=conv(polyder(polymat(k-1,:)),polymat(1,:));
21     polymat(k,(mpl-cpd):mpl)=temp(length(temp)-cpd:end);
end
23 %% Apply Taylor method
h=T/steps;
25 y_T=y_0;
increment_poly=sum(polymat.*repmat(cumprod(h*ones(order,1))./cumprod
    ((1:(order))'),1,mpl),1);
27 for k=1:steps
    y_T=y_T+polyval(increment_poly,y_T);
29 end
end
```

Implementing the Taylor methods for general right hand sides in Matlab is more difficult and we need to

(i) find an appropriate formalism that allows us to construct and generate the L-operators of arbitrary order, and

(ii) find a way of dealing with high order derivatives of the right hand side f.

The next two sections (Section 2.2 and Section 2.3) will deal with (i), while we will discuss (ii) in Section 2.4.

2.2 Differential statements and structure vectors

As we have seen above, the L-operators become complicated very fast. In order to be able to implement them in Matlab, we will make use of their recursive structure and iteratively generate the next higher L-operator from the previous one. In this section we introduce the notion of differential structure vectors which capture the recursive structure of the individual differential terms in the L-operators.

2.2.1 Differential structure vectors

One possibility to abstractly capture the structure of these differential terms is to use rooted trees. This is also the standard approach in literature. We will however use vectors (with special properties) as this allow a more straightforward implementation in Matlab. The trade-off is that we loose the intuitiveness of the rooted trees and mathematical rigour will lead to somewhat tedious notation.

Definition 2.7 (sub vector) *Let $A \in \{\mathbb{N}, \mathbb{N}_0, \mathbb{Z}\}$, $k \in \mathbb{N}$, $\mathbf{v} \in A^k$, $l_0 \in \{0, 1, 2, \dots, k\}$, then a vector $\mathbf{w} \in A^{l_0}$ (where A^0 consists of the empty vector) is called sub vector of $\mathbf{v}$ if there exists $l_1, l_2 \in \{0, 1, 2, \dots, k\}$, $\mathbf{v}^{(1)} \in A^{l_1}$ and $\mathbf{v}^{(2)} \in A^{l_2}$ such that $\mathbf{v} = (\mathbf{v}^{(1)}, \mathbf{w}, \mathbf{v}^{(2)})$.*

Definition 2.8 (non-trivial sub vector) *Let $A \in \{\mathbb{N}, \mathbb{N}_0, \mathbb{Z}\}$, $k \in \mathbb{N}$, $\mathbf{v} \in A^k$, $l_0 \in \{0, 1, 2, \dots, k\}$, then a sub vector $\mathbf{w} \in A^{l_0}$ is called a non-trivial sub vector if $\mathbf{w}$ is not the empty vector and $\mathbf{w} \neq \mathbf{v}$.*

The following definition introduces the essential object of this section.

Definition 2.9 (differential structure vector (DSV)) *Let $m \in \mathbb{N}$ and $\mathbf{v} = (v_1, \dots, v_m) \in \mathbb{N}_0^m$, then if $m = 1$ we call $\mathbf{v}$ a differential structure vector of length 1 (sometimes trivial structure vector) if $\mathbf{v} = 0$. Else if $m \in \{2, 3, \dots\}$ we call $\mathbf{v}$ a differential structure vector of length m if there exists $m_1 \dots, m_{v_1} \in \mathbb{N}$ such that for all $i \in \{1, 2, \dots, v_1\}$ there exists $\mathbf{v}^{(i)} \in \mathbb{N}_0^{m_i}$ satisfying*

(i) $\mathbf{v} = \left(v_1, \mathbf{v}^{(1)}, \dots, \mathbf{v}^{(v_1)}\right)$

(ii) for all $i \in \{1, 2, \ldots, v_1\}$*:* $\mathbf{v}^{(i)}$ *is a differential structure vector of length* m_i *.*

The tupel of sub vectors $\left(\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)}\right)$ *is called a recursive decomposition of* $\mathbf{v}$*.*

The following example illustrates this definition.

Example 2.10 *Set* $\mathbf{v} = (3, 2, 0, 0, 0, 1, 0) \in \mathbb{N}_0^7$*, then* $\mathbf{v}$ *is a DSV. To see this we let* $m_1 = 3$*,* $m_2 = 1$*,* $m_3 = 2$*,* $\mathbf{v}^{(1)} = (2, 0, 0)$*,* $\mathbf{v}^{(2)} = 0$ *and* $\mathbf{v}^{(3)} = (1, 0)$*. Clearly* $\mathbf{v} = \left(v_1, \mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)}\right)$ *hence it only remains to prove that* $\mathbf{v}^{(1)}$*,* $\mathbf{v}^{(2)}$ *and* $\mathbf{v}^{(3)}$ *are DSVs:*

- $\mathbf{v}^{(2)}$ *is the trivial DSV.*
- *For* $\mathbf{v}^{(1)}$*, observe that one can use the recursive decomposition* $(0, 0)$ *to see that* $\mathbf{v}^{(1)}$ *is indeed a DSV.*
- *For* $\mathbf{v}^{(3)}$*, observe that one can use the recursive decomposition* (0) *to see that* $\mathbf{v}^{(3)}$ *is indeed a DSV.*

The recursive decomposition can be calculated in Matlab using the function `find_rec_decomp` given in Listing 2.2, which outputs a vector containing the indices of the first element of each sub vector of the recursive decomposition.

Listing 2.2: Matlab function for determining recursive decomposition of a DSV

```
function [ind] = find_rec_decomp( DSV )
% find_rec_decomp: calculates the recursive decomposition of a DSV
% INPUT
% DSV: differential structure vector
% OUTPUT
% ind: vector of length DSV(1) containing the location of the first
    element of each recursive decomposition as an index of DSV

ind=zeros(1,DSV(1));
ind(1)=2;
for i=2:DSV(1)
    level=1;
    step=ind(i-1);
    while level>0
        if DSV(step)==0
            level=level-1;
        elseif DSV(step)>1
            level=level+DSV(step)-1;
        end
        step=step+1;
    end
    ind(i)=step;
end
end
```

Definition 2.11 (set of DSVs) *We denote by*

$$\mathcal{F} := \{\mathbf{v} \in \cup_{m\in\mathbb{N}}\mathbb{N}_0^m : \mathbf{v} \text{ is a DSV}\}$$

the set of all DSVs.

2.2.2 Differential statements

We can now use DSVs to define differential statements.

Definition 2.12 (differential statement (DS)) *Let $d \in \mathbb{N}$ and let $f \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$. Then denote by $f^{(\mathbf{v})} \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^n\right)$, $\mathbf{v} \in \mathcal{F}$ the functions with the property that $f^{(0)} = f$ and with the property that for every $\mathbf{v} \in \mathcal{F}$ non-trivial DSV with recursive decomposition $\left(\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)}\right)$ it holds for all $x \in \mathbb{R}^d$ that*

$$f^{(\mathbf{v})}(x) = f^{(v_1)}(x)\left[f^{(\mathbf{v}^{(1)})}(x), \ldots, f^{(\mathbf{v}^{(v_1)})}(x)\right]. \tag{2.5}$$

We call $f^{(\mathbf{v})}$ the differential statement (DS) of f associated to $\mathbf{v}$.

In other words, we associate to any $f \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$ and $\mathbf{v} \in \mathcal{F}$ a differential statement ($f^{(\mathbf{v})}$). The following definition makes this correspondence explicit and will later ease the notation.

Definition 2.13 (correspondence map) *Let $d \in \mathbb{N}$. Then we denote by $\Phi_d : \mathcal{F} \times C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right) \to C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$ the function with the property that for all $\mathbf{v} \in \mathcal{F}$, $f \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$ it holds that*

$$\Phi_d(\mathbf{v}, f) = f^{(\mathbf{v})} \tag{2.6}$$

and Φ_d is called correspondence map.

The Matlab function `correspondence_map` (see Listing 2.3) implements the functionality of the correspondence map. It takes a DSV and outputs the differential statement associated to the DSV as a string.

Listing 2.3: Matlab implementation of the correspondence map

```
function [ DS ] = correspondence_map( DSV )
% correspondence_map: converts a DSV to a string containing the DS
% INPUT
% DSV:  differential structure vector
% OUTPUT
% DS:   string containing the differential statement

if length(DSV)>1
    % decompose into recursive decomposition
    ind_start=find_rec_decomp(DSV);
    ind_end=[ind_start(2:end)-1,length(DSV)];
```

```
    n=length(ind_start);
    % construct the differential statement using a recursion
    DS=['D',num2str(DSV(1)),'(x,[',correspondence_map(DSV(ind_start
    (1):ind_end(1)))];
    for i=2:n
        DS=[DS,',',correspondence_map(DSV(ind_start(i):ind_end(i)))
    ];
    end
    DS=[DS,'])'];
else
    DS='fval';
end
end
```

2.2.3 Differentiating DSVs

In this section we will see that given $\mathbf{v} \in \mathcal{F}$ applying the L-operator to the DS $f^{(\mathbf{v})}$ results in the sum of DSs associated to a set of new DSVs derived from $\mathbf{v}$ by simple vector transformations. In some sense this extends the notion of differentiation to DSVs.

Definition 2.14 ($\mathcal{L}^{(m)}$-operators) *Let $m \in \mathbb{N}$ and let $i \in \{1, \ldots, m\}$. Then we denote by $\mathcal{L}_i^{(m)} : \mathbb{N}_0^m \to \mathbb{N}_0^{m+1}$ the function with the property that for all $\mathbf{v} \in \mathbb{N}_0^m$ it holds*

$$\mathcal{L}_i^{(m)}(\mathbf{v}) = (v_1, \ldots, v_{i-1}, v_i + 1, 0, v_{i+1}, \ldots, v_m). \tag{2.7}$$

Lemma 2.15 *Let $m \in \mathbb{N}$, $i \in \{1, \ldots, m\}$. Then for all DSVs $\mathbf{v} = (v_1, \ldots, v_m \in \mathbb{N}_0^m$ the vector $\mathcal{L}_i^{(m)}(\mathbf{v})$ is a DSV.*

Proof We prove Lemma 2.15 by induction on the length m. In the base case $m = 1$ it holds that $i = 1$ and $\mathbf{v} = 0$. Hence $\mathcal{L}_1^{(1)}(0) = (1, 0)$ which is again a DSV. For the induction step we assume that the statement holds for all DSVs of length smaller or equal to m.f Let $\mathbf{v} \in \mathbb{N}_0^{m+1}$ DSV with recursive decomposition $\left(\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)}\right)$. Then for $i = 1$ observe that $\left(0, \mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)}\right)$ is a recursive decomposition of $\mathcal{L}_1^{(m)}(\mathbf{v}) = (v_1 + 1, 0, \mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)})$ and therefore $\mathcal{L}_1^{(m)}(\mathbf{v})$ is a DSV. For $i \in \{2, \ldots, m+1\}$, let $j, l \in \{2, \ldots, m+1\}$ be such that $v_j^{(l)}$ is the i-th component of $\mathbf{v}$ and let m_l be the length of $\mathbf{v}^{(l)}$. By the induction hypothesis $\mathcal{L}_j^{(m_l)}(\mathbf{v}^{(l)})$ is DSV and since

$$\left(\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(j-1)}, \mathcal{L}_j^{(m_l)}\left(\mathbf{v}^{(l)}\right), \mathbf{v}^{(j+1)}, \ldots, \mathbf{v}^{(v_1)}\right)$$

is a recursive decomposition of $\mathcal{L}_i^{(m)}(\mathbf{v})$ which completes the proof of Lemma 2.15. □

The following proposition gives the desired connection between the L-operator (Definition 2.1) and the $\mathcal{L}^{(m)}$-operators (Definition 2.14).

Proposition 2.16 *Let $d, m \in \mathbb{N}$, $f \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$ and $\mathbf{v} \in \mathbb{N}_0^m \cap \mathcal{F}$. Then it holds that*

$$L_f\left(\Phi_d(\mathbf{v}, f)\right) = \sum_{i=1}^{m} \Phi_d\left(\mathcal{L}_i^{(m)}(\mathbf{v}), f\right). \tag{2.8}$$

Proof We prove Proposition 2.16 by induction on the length m. In the base case $m = 1$ it holds that $v = 0$ and hence

$$L_f \Phi_d(\mathbf{v}, f) = L_f f = f^{(1)}[f] = \Phi_d\left((1,0), f\right) = \Phi_d\left(\mathcal{L}_1^{(1)}(\mathbf{v}), f\right).$$

For the induction step assume (2.8) holds for all DSVs of length smaller or equal to m. Let $\mathbf{v} \in \mathbb{N}_0^{m+1}$ be DSV with recursive decomposition $\left(\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)}\right)$ and for all $i \in \{1, \ldots, v_1\}$ denote by m_i the length of $\mathbf{v}^{(i)}$. Then using the product rule and the induction hypothesis we get

$$\begin{aligned}
L_f \Phi_d(\mathbf{v}, f) &= L_f f^{(v_1)}\left[\Phi_d\left(\mathbf{v}^{(1)}, f\right), \ldots, \Phi_d\left(\mathbf{v}^{(v_1)}, f\right)\right] \\
&= f^{(v_1+1)}\left[f, \Phi_d\left(\mathbf{v}^{(1)}, f\right), \ldots, \Phi_d\left(\mathbf{v}^{(v_1)}, f\right)\right] + \\
&\qquad \sum_{i=1}^{m} f^{(v_1)}\left[\ldots, L_f \Phi_d\left(\mathbf{v}^{(i)}, f\right), \ldots\right] \\
&= f^{(v_1+1)}\left[\Phi_d(0, f), \Phi_d\left(\mathbf{v}^{(1)}, f\right), \ldots, \Phi_d\left(\mathbf{v}^{(v_1)}, f\right)\right] + \\
&\qquad \sum_{i=1}^{m} f^{(v_1)}\left[\ldots, \sum_{j=1}^{m_i} \Phi_d\left(\mathcal{L}_j^{(m_i)}\left(\mathbf{v}^{(i)}\right), f\right), \ldots\right] \\
&= \Phi_d\left(\mathcal{L}_1^{(m)}(\mathbf{v}), f\right) + \sum_{i=1}^{m}\sum_{j=1}^{m_i} f^{(v_1)}\left[\ldots, \Phi_d\left(\mathcal{L}_j^{(m_i)}\left(\mathbf{v}^{(i)}\right), f\right), \ldots\right] \\
&= \Phi_d\left(\mathcal{L}_1^{(m)}(\mathbf{v}), f\right) + \sum_{i=1}^{m}\sum_{j=1}^{m_i} \Phi_d\left(\left(v_1, \mathbf{v}^{(1)}, \ldots, \mathcal{L}_j^{(m_i)}\left(\mathbf{v}^{(i)}\right), \ldots, \mathbf{v}^{(v_1)}\right), f\right) \\
&= \Phi_d\left(\mathcal{L}_1^{(m)}(\mathbf{v}), f\right) + \sum_{i=1}^{m}\sum_{j=1}^{m_i} \Phi_d\left(\mathcal{L}^m_{1+m_1+\cdots+m_{i-1}+j}(\mathbf{v}), f\right) \\
&= \Phi_d\left(\mathcal{L}_1^{(m)}(\mathbf{v}), f\right) + \sum_{i=2}^{m} \Phi_d\left(\mathcal{L}_i^m(\mathbf{v}), f\right) \\
&= \sum_{i=1}^{m} \Phi_d\left(\mathcal{L}_i^m(\mathbf{v}), f\right).
\end{aligned}$$

This completes the proof of Proposition 2.16. □

Hence given $\mathbf{v} \in \mathcal{F}$ and applying the L-operator to the differential statement $f^{(\mathbf{v})}$ results in the sum of the differential statements associated to the DSVs $\mathcal{L}_1^{(m)}(\mathbf{v}), \ldots, \mathcal{L}_m^{(m)}(\mathbf{v})$. This means we can interpret "differentiating" a DSV of length m as calculating the DSVs $\mathcal{L}_1^{(m)}(\mathbf{v}), \ldots, \mathcal{L}_m^{(m)}(\mathbf{v})$. The Matlab function `diff_DSV` given in Listing 2.4 implements this functionality.

Listing 2.4: Matlab function for "differentiating" DSVs

```
function [ DSVs ] = diff_DSV( DSV )
% diff_DSV: "differentiates" DSV
% INPUT
% DSV:  differential structure vector
% OUTPUT
% DSVs: matrix containing the differentiated DSVs as columns

m=length(DSV);
DSVs=zeros(m+1,m);
for i=1:m
    DSVs(:,i)=[DSV(1:i-1);DSV(i)+1;0;DSV(i+1:end)];
end
end
```

2.2.4 Normalized DSVs

Observe, for every $\mathbf{v} \in \mathcal{F}$ with recursive decomposition $\left(\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)}\right)$, every $d \in \mathbb{N}$ and every $f \in C^\infty\left(\mathbb{R}^d, \mathbb{R}^d\right)$, the DS $f^{(\mathbf{v})}$ is a v_1-linear form and hence does not depend on the ordering of terms $f^{(\mathbf{v}^{(1)})}, \ldots, f^{(\mathbf{v}^{(v_1)})}$. Therefore different DSVs can correspond to the same differential statement. In other words, the correspondence map Φ_d is not injective.

In this section we introduce a method to normalize DSVs that allows us to identify which DSVs correspond to the same DSs.

Definition 2.17 (dictionary order) *We denote by $\preccurlyeq$ the relation on $(\mathcal{F}, \mathcal{F})$ with the property that for all $\mathbf{v}, \mathbf{w} \in \mathcal{F}$ it holds that $\mathbf{v} \preccurlyeq \mathbf{w}$ if and only if $\mathbf{v} = \mathbf{w}$, or $\text{length}(\mathbf{v}) < \text{length}(\mathbf{w})$, or $\text{length}(\mathbf{v}) = \text{length}(\mathbf{w}) =: m$ and $\exists k \in \{0, 1, \ldots m-1\} : v_1 = w_1, \ldots, v_k = w_k, v_{k+1} < w_{k+1}$. We call $\preccurlyeq$ the dictionary order.*

Proposition 2.18 *The dictionary order $\preccurlyeq$ is a total order on $\mathcal{F}$.*

Proof The properties antisymmetry, transitivity and totality can be check directly. □

We can use the dictionary order to define normalized DSVs. This is done in the next definition.

Definition 2.19 (normalized DSV) *Let $m \in \mathbb{N}$ and $\mathbf{v} = (v_1, \ldots, v_m) \in \mathcal{F}$ be a DSV with recursive decomposition $\left(\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)}\right)$. Then $\mathbf{v}$ is called a normalized DSV if one of the following is fulfilled*

(i) $\mathbf{v}$ *has length* 1

(ii) $\mathbf{v}^{(1)} \preccurlyeq \mathbf{v}^{(2)} \preccurlyeq \cdots \preccurlyeq \mathbf{v}^{(v_1)}$ *and for all* $i \in \{1, \ldots, v_1\}$ *it holds that* $\mathbf{v}^{(i)}$ *is a normalized DSV.*

Definition 2.20 (set of normalized DSV) *We denote by*

$$\mathcal{G} := \{\mathbf{v} \in \mathcal{F} : \mathbf{v} \text{ is a normalized DSV}\}$$

the set of all normalized DSVs.

The definition of normalized DSVs allows us to recursively normalize a DSV.

Definition 2.21 *We set* $S_n := \{\pi : \{1, \ldots, n\} \to \{1, \ldots, n\} : \pi \text{ is a bijection}\}$ *and call this the set of permutations on the set* $\{1, \ldots, n\}$.

Definition 2.22 (normalizing function) *We denote by* $\Psi : \mathcal{F} \to \mathcal{G}$ *the function with the property that* $\Psi(0) = 0$ *and with the property that for every* $\mathbf{v} \in \mathcal{F}$ *non-trivial DSV with recursive decomposition* $\left(\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(v_1)}\right)$ *and for every* $\pi \in S_{v_1}$ *satisfying for all* $i \in \{1, \ldots, v_1 - 1\}$ *that* $\Psi(\mathbf{v}^{(\pi(i))}) \preccurlyeq \Psi(\mathbf{v}^{(\pi(i+1))})$ *it holds that* $\Psi(\mathbf{v}) = \left(v_1, \Psi(\mathbf{v}^{(\pi(1))}), \ldots, \Psi(\mathbf{v}^{(\pi(v_1))})\right)$.

Remark 2.23 *The required permutation always exists and is unique because by Proposition 2.18* $\preccurlyeq$ *is a total order on* $\mathcal{F}$.

In Matlab the function `norm_DSV` is an implementation of the normalizing function Ψ. The code is given in Listing 2.5.

Listing 2.5: Matlab implementation of the normalizing function

```
function [ DSV ] = norm_DSV( DSV )
% norm_DSV: normalizes the DSV
% INPUT
% DSV: differential structure vector
% OUTPUT
% DSV: normalized form of DSV

if length(DSV)>2
    % decompose into recursive decomposition
    ind_start=find_rec_decomp(DSV);
    ind_end=[ind_start(2:end)-1,length(DSV)];
    n=length(ind_start);
    % recursively normalize subvectors
    max_length=max(ind_end-ind_start)+1;
    temp_vec=zeros(n,max_length);
    pos_ind=zeros(1,n);
    for i=1:n
        pos_ind(i)=max_length-(ind_end(i)-ind_start(i));
        temp_vec(i,pos_ind(i):end)=norm_DSV(DSV(ind_start(i):ind_end
    (i)));
```

```
    end
    % sort normalized subvectors
    [temp_vec,sort_ind]=sortrows(temp_vec);
    pos_ind=pos_ind(sort_ind);
    % update vec and ordervec
    pos=2;
    for i=1:n
        currlength=max_length-pos_ind(i);
        DSV(pos:pos+currlength)=temp_vec(i,pos_ind(i):end);
        pos=pos+currlength+1;
    end
end
```

The function `norm_DSV` is used in the function `simplify` (see Listing 2.6) that takes a list of DSVs (given as a matrix where the top row indicates how often the DSV appears), normalizes all DSVs and then removes repeating DSVs. Observe that `simplify` also keeps track of how often a particular normalized DSV appeared and saves this amount in the first row of the matrix.

Listing 2.6: Matlab function that simplifies a list of DSVs

```
function[ DSVs ] = simplify( DSVs )
% simplify: normalizes list of DSV and removes duplicates
% INPUT
% DSVs: a matrix containing DSVs as columns with the first row
    representing the number of repetitions of the DSV in that column
% OUTPUT
% DSV:  a matrix containing the reduced list of DSVs as columns with
     the first row representing the number of repetitions of the DSV
     in that column

% Count and remove duplicate DSVs and save the number of repetitions
     in the first row (this is done once before normalizing in order
     to reduce the runtime)
i=1;
while i<=size(DSVs,2)
    count=0;
    j=i+1;
    while j<=size(DSVs,2)
        if DSVs(2:end,i)==DSVs(2:end,j)
            count=count+DSVs(1,j);
            DSVs(:,j)=[];
            j=j-1;
        end
        j=j+1;
    end
    DSVs(1,i)=DSVs(1,i)+count;
    i=i+1;
end
% Normalize DSVs
for j=1:size(DSVs,2)
```

```
    DSVs(2:end,j)=norm_DSV(DSVs(2:end,j));
end
% Count and remove duplicate DSVs and save the number of repetitions
    in the first row
i=1;
while i<=size(DSVs,2)
    count=0;
    j=i+1;
    while j<=size(DSVs,2)
        if DSVs(2:end,i)==DSVs(2:end,j)
            count=count+DSVs(1,j);
            DSVs(:,j)=[];
            j=j-1;
        end
        j=j+1;
    end
    DSVs(1,i)=DSVs(1,i)+count;
    i=i+1;
end
end
```

Remark 2.24 *There exists a one-to-one correspondence between normalized DSVs and rooted trees.*

2.3 Generating L-operators in Matlab

This section describes how the differential structure vectors can be used to construct the L-operators (see Section 2.1) as functions in Matlab. The main idea is to iteratively generate them, using the correspondence between DSVs and DSs.

2.3.1 Recursive construction

The first L-operator is given for all $x \in \mathbb{R}^d$ by

$$L_f^{(1)} f(x) = f^{(1)}(x)[f(x)] \tag{2.9}$$

which is the differential statement associated to $\mathbf{v} = (1, 0)$ and f (i.e. $\Phi_d(\mathbf{v}, f) = f^{(1)}[f] = L_f^{(1)} f$). The second L-operator is given by

$$L_f^{(2)} f(x) = L_f(L_f^{(1)} f) \tag{2.10}$$

which using Proposition 2.16 we can write as

$$\begin{aligned} L_f^{(2)} f(x) &= L_f\left(\Phi_d(\mathbf{v}, f)\right) \\ &= \sum_{i=1}^{2} \Phi_d(\mathcal{L}_i^2(\mathbf{v}), f) \\ &= \Phi_d((2,0,0), f) + \Phi_d((1,1,0), f) \\ &= f^{(2)}[f, f] + f^{(1)}[f^{(1)}[f]]. \end{aligned}$$

Continuing this argument iteratively it becomes clear that in order to calculate the operator $L_f^{(n)}$ we can take the set of all DSVs $\{\mathbf{v}_1, \ldots, \mathbf{v}_{n!}\}$ that generate the differential statements for the operator $L_f^{(n-1)}$ and then differentiate each of these DSVs to get the new set of DSVs given by

$$\begin{aligned} &\big\{\mathcal{L}_1^{n+1}(\mathbf{v}_1), \ldots, \mathcal{L}_{n+1}^{n+1}(\mathbf{v}_1), \mathcal{L}_1^{n+1}(\mathbf{v}_2), \ldots \\ &\quad \ldots, \mathcal{L}_{n+1}^{n+1}(\mathbf{v}_2), \ldots, \mathcal{L}_1^{n+1}(\mathbf{v}_{n!}), \ldots, \mathcal{L}_{n+1}^{n+1}(\mathbf{v}_{n!})\big\} \end{aligned}$$

which now generates the differential statements required for the operator $L_f^{(n)}$. Therefore we have a method to construct L-operators of any order.

2.3.2 Improving construction using normalized DSVs

Performing the basic procedure outlined in Subsection 2.3.1 becomes costly quite fast. In order to calculate the operator $L_f^{(n)}$, we need to calculate $n!$ DSVs.

As already outlined in Subsection 2.2.4, it turns out that many of these DSVs correspond to the same differential statements. Therefore we can use the methods we constructed in that section in order to normalize the DSVs and group them appropriately.

Using this simplification, we end up with the method that generates the L-operators iteratively as in Subsection 2.3.1, while in each step normalizing the DSVs, replacing duplicates and keeping track of how often a normalized DSV occurred.

This method is implemented in the function `genLops` which is given in Listing 2.7. It uses the functions `diff_DSV` (see Listing 2.4) to differentiate the DSVs, the function `simplify` (see Listing 2.6) to normalize and reduce the list of DSVs and finally the function `printLop` (see Listing 2.8) to print the L-operator to an m-file.

Listing 2.7: Matlab function for generating L-operator m-files

```
function [ ] = genLops( N, mypath )
```

```
% genLops: generates the Loperator functions L1,...,LN as m-files
% INPUT
% N:          number of maximal L operator
% mypath:     path of current folder ('mypath/Loperators' needs to
    exist)

%% File management
cd([mypath,'/Loperators'])
%% Main Function
% iteratively calculate the generating DSVs
DSV_mat=cell(1,N);
DSV_mat{1}=[1;1;0];
for k=2:N
    tic
    size_old=size(DSV_mat{k-1});
    size_old(1)=size_old(1)-1;
    DSV_mat{k}=zeros(size_old(1)+2,size_old(2)*size_old(1));
    % differentiate all DSVs of the previous L-operator
    for i=1:size_old(2)
        DSV_mat{k}(1,(i-1)*size_old(1)+1:i*size_old(1))=DSV_mat{k
    -1}(1,i);
        DSV_mat{k}(2:end,(i-1)*size_old(1)+1:i*size_old(1))=diff_DSV
    (DSV_mat{k-1}(2:end,i));
    end
    % normalize and reduce DSVs
    DSV_mat{k}=simplify(DSV_mat{k});
    size(DSV_mat{k})
    toc
end
% generate L-operator m-files based on the generating DSVs
for l=1:N
    printLop(DSV_mat{l})
end
%% File management
cd(mypath)
end
```

Listing 2.8: Example code for printing an L-operator to an m-file using the generating DSVs

```
function [] = printLop(DSVs)
% printLops:    constructs the L-operator corresponding to DSVs into
    an m-file
% INPUT
% DSVs:         a matrix containing DSVs as columns with the first
    row representing the number of repetitions of the DSV in that
    column

N=size(DSVs,1)-2;
% delete and reopen a new file with the name LN.m
if exist(['L',num2str(N),'.m'],'file')
    delete(['L',num2str(N),'.m'])
end
```

```
file_id = fopen(['L',num2str(N),'.m'],'w+');
% print first part of the the L-operator to file
fprintf(file_id,'%% This code was automatically generated\n');
fprintf(file_id,['function [ out ] = L',num2str(N),' (x)\n\n']);
fprintf(file_id,'fval=myfun(x);\n');
% print the L-operator to file using correspondance_map
DS=correspondence_map(DSVs(2:end,1));
fprintf(file_id,['out=',DS]);
for j=2:size(DSVs,2)
    DS=correspondence_map(DSVs(2:end,j));
    fprintf(file_id,['+',num2str(DSVs(1,j)),'*',DS]);
end
fprintf(file_id,';\n');
fprintf(file_id,'end');
end
```

2.4 Automatic differentiation

In order to apply the L-operators from the previous section, we need a method to calculate the elementary derivatives of the right hand side f in Matlab. The method we use is called automatic differentiation which can efficiently calculate the derivative of functions given in algorithmic form (e.g. as an m-file in Matlab). The only limitation is that the function only consists of the basic arithmetic operations (+,-,*,/), elementary functions (e.g. exp,sin,cos) and for-loops.

We do not go into any further detail, but only remark that automatic differentiation is superior to both numerical approximations of the derivatives and symbolic differentiation.

For Matlab there exists an open source project called ADiGator (see [5]) which provides this functionality.

The Matlab function Listing 2.9 calculates the required elementary derivatives from the functions of the ADiGator output.

Listing 2.9: Matlab function for generating D-operator m-files

```
function [ ] = genDops( l, dim, mypath )
% genDops: generates the functions D1,...,Dl corresponding to the
    elementary derivatives (based on adigator) as m-files
% INPUT:
% l:        order of highest derivative
% dim:      dimension of solution vektor, rhs: R^dim --> R^dim
% mypath:   path of current folder ('mypath/Doperators' needs to
    exist)

%% File management
```

```
cd([mypath,'/Doperators'])
%% Create adigator derivative functions
opts=adigatorOptions('overwrite',1);
x=adigatorCreateDerivInput([dim 1],'x');
adigator('myfun',{x},'f1',opts)
x=struct('f',x);
dxstr=[];
for i=2:l
    % update dxstr to dx...dx (i times)
    dxstr=[dxstr,'dx'];
    % update input variable x
    x.(dxstr)=ones(dim,1);
    % generate adigator derivative file
    adigator(['f',num2str(i-1)],{x},['f',num2str(i)],opts)
end
%% Create D1.m,...,Dl.m
dxstr=[];
for i=1:l
    % update dxstr to dx...dx (i times)
    dxstr=[dxstr,'dx'];
    % delete and reopen a new file with the name Dk.m
    if exist(['D',num2str(i),'.m'],'file')
        delete(['D',num2str(i),'.m'])
    end
    file_id = fopen(['D',num2str(i),'.m'],'w+');
    % check whether i'th derivative is zero
    var=struct('f',ones(dim,1),'dx',ones(dim,1));
    test=eval(['f',num2str(l),'(var)']);
    if not(isfield(test,dxstr))
        % write the file
        fprintf(file_id,'%% This code was automatically generated\n'
    );
        fprintf(file_id,['function [ out ] = D',num2str(i),'( x, v )
    \n\n']);
        fprintf(file_id,'out=zeros(size(v,1),1);\n');
        fprintf(file_id,'end\n\n');
    elseif dim==1
        % write the file
        fprintf(file_id,'%% This code was automatically generated\n'
    );
        fprintf(file_id,['function [ out ] = D',num2str(i),'( x, v )
    \n\n']);
        fprintf(file_id,['temp=f',num2str(l),'(struct(''f'',x,''dx''
    ,1));\n']);
        fprintf(file_id,['\t out=temp.',dxstr,'*prod(v);\n\n']);
        fprintf(file_id,'end\n');
    else
        % write the file
        fprintf(file_id,'%% This code was automatically generated\n'
    );
        fprintf(file_id,['function [ out ] = D',num2str(i),'( x, v )
```

```
    \n\n']);
        fprintf(file_id,'%% unit vektor\n');
        fprintf(file_id,'uv = @(n,k) [zeros(k-1,1); 1; zeros(n-k,1)
    ];\n\n');
        fprintf(file_id,'n=size(v,1);\n');
        fprintf(file_id,['temp=f',num2str(l),'(struct(''f'',x,''dx''
    ,ones(n,1)));\n']);
        fprintf(file_id,'out=zeros(n,1);\n');
        fprintf(file_id,['for i=1:length(temp.',dxstr,')\n']);
        fprintf(file_id,['\t ind=sub2ind(size(v),temp.',dxstr,'
    _location(i,2:end),1:',num2str(i),');\n']);
        fprintf(file_id,['\t out=out+temp.',dxstr,'(i)*prod(v(ind))*
    uv(n,temp.',dxstr,'_location(i,1));\n']);
        fprintf(file_id,'end\n\n');
        fprintf(file_id,'end\n');
    end
end
%% File management
cd(mypath)
end
```

2.5 Taylor methods in Matlab

Using the L-operators `L1`,...,`Lm` (see Section 2.3) and the derivative operators `D1`,...,`Dm` corresponding to the right hand side f (see Section 2.4), we can easily construct the Taylor methods up to order $m+1$. The Matlab function `genTaylorStep` (Listing 2.10) combines the L-operators in the correct form, so that they can be used by the function `taylor_method` (Listing 2.11).

Listing 2.10: Matlab function for generating the Taylorstep m-files

```
function [ ] = genTaylorStep( orders, mypath )
% genTaylorStep: generates the TaylorStep function for each order
    given in orders and saves them as m-files
% INPUT:
% orders: vector containing the orders for which the TaylorStep
    should be calculated
% mypath: path of current folder ('mypath/TaylorSteps' needs to
    exist)

%% File management
cd([mypath,'/TaylorSteps'])
%% Main function
for k=1:length(orders)
    % delete and reopen a new file with the name TaylorStep.m
    if exist(['TaylorStep',num2str(orders(k)),'.m'],'file')
        delete(['TaylorStep',num2str(orders(k)),'.m'])
    end
    file_id = fopen(['TaylorStep',num2str(orders(k)),'.m'],'w+');
```

```
    fprintf(file_id,'%% This code was automatically generated\n');
    fprintf(file_id,['function [ step ] = TaylorStep',num2str(orders
    (k)),' (h, x)\n']);
    fprintf(file_id,'step=h*myfun(x)');
    for i=2:orders(k)
        fprintf(file_id, ['+h^',num2str(i),'/factorial(',num2str(i),
    ')*L',num2str(i-1),'(x)']);
    end
    fprintf(file_id, ';\n');
    fprintf(file_id, 'end');
end
%% File management
cd(mypath)
end
```

Listing 2.11: Matlab implementation of the Taylor methods

```
function [ y_T ] = taylor_method( T, y_0, steps, order)
% taylor_method: Taylor method ODE-solver with arbirary order for
    myfun.m as rhs
% INPUT
% T:          final time
% y_0:        initial value
% steps:      number of time steps
% order:      order of ODE-solver
% OUTPUT
% y_T:        approximated solution at time T

% generate array containing all taylor steps up to order 15 (add
    more if required)
TaylorStep={@(h,x)TaylorStep1(h,x),@(h,x)TaylorStep2(h,x),@(h,x)
    TaylorStep3(h,x),@(h,x)TaylorStep4(h,x),@(h,x)TaylorStep5(h,x),@
    (h,x)TaylorStep6(h,x),@(h,x)TaylorStep7(h,x),@(h,x)TaylorStep8(h
    ,x),@(h,x)TaylorStep9(h,x),@(h,x)TaylorStep10(h,x),@(h,x)
    TaylorStep11(h,x),@(h,x)TaylorStep12(h,x),@(h,x)TaylorStep13(h,x
    ),@(h,x)TaylorStep14(h,x),@(h,x)TaylorStep15(h,x)};
% Apply Taylor Method
h=T/steps;
y_T=y_0;
for i=1:steps
    y_T=y_T+TaylorStep{order}(h,y_T);
end
end
```

Chapter 3

Runge-Kutta methods

In this chapter we present the famous Runge-Kutta methods and present a bootstrapping procedure that given a Runge-Kutta method of order m creates a new Runge-Kutta method of order $m+1$. The advantage of this method is that it allows to explicitly write down the higher order method. However, subsequent applications of the bootstrapping approach lead to a blow up in complexity of the new method.

We follow [6] and [1] and refer to these books for more details.

3.1 Runge-Kutta methods

Definition 3.1 (explicit Runge-Kutta method) *Assume the setting in Section 1.1. Let $s \in \mathbb{N}$, $\mathbf{b} = (b_1, \dots, b_s) \in \mathbb{R}^s$ and $\mathbf{A} = (a_{i,j})_{i,j\in\{1,\dots,s\}} \in \mathbb{R}^{s,s}$ be a strictly lower triangular matrix, and let $k_i : [0, T - t_0] \to \mathbb{R}^d$, $i \in \{1, \dots, s\}$ with the property that for all $i = 1, \dots, s$ and all $h \in [0, T - t_0]$ it holds that*

$$k_i(h) = f\left(y_0 + h\sum_{j=1}^{s} a_{i,j}k_j\right). \tag{3.1}$$

Then we call

$$y_h = y_0 + h\sum_{i=1}^{s} b_i k_i(h) \tag{3.2}$$

an explicit s-step $(\mathbf{A}, \mathbf{b})$-Runge-Kutta method.

Remark 3.2 *To shorten the notation we generally just neglect the h as it is clear from the context and simply write k_i instead of $k_i(h)$. If the right hand side f additionally has a time dependence (i.e. $f : [t_0, T] \times \mathbb{R}^d \to \mathbb{R}^d$), then an additional vector $\mathbf{c} \in \mathbb{R}^s$ is introduced and the steps k_i are defined by*

$$k_i = f\left(t + c_i h, y_0 + h\sum_{j=1}^{s} a_{i,j}k_j\right).$$

In this case the Runge-Kutta methods are often given in a so-called Butcher tableau

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array}$$

We will also use this convention and will always add the vector $\mathbf{c}$ *for the method although it will not be required in our considerations.*

Historically Runge-Kutta methods with order of convergence m were created by comparing appropriate Taylor expansions of the steps k_i with the Taylor method of order m (see Chapter 2) and ensuring that all terms of order smaller or equal to m in h cancel out by an appropriate choice of $\mathbf{A}$ and $\mathbf{b}$. The most famous such methods up to order 4 are collected below in Figure 3.1. A list of some further methods up to order 8 is given in Section 3.2 below.

explicit Euler method

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$$

explicit RK midpoint method

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$$

Kutta's third order method

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$$

classic fourth order method

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

Figure 3.1: classical RK-methods up to order 4

The Matlab function `RungeKutta` given in Listing 3.1 is an implementation of explicit Runge-Kutta methods.

Listing 3.1: Matlab implementation of Runge-Kutta methods

```
function [ y_T ] = RungeKutta( f, T, y_0, steps, b, A)
% RungeKutta: (A,b)-Runge-Kutta method ODE-solver
% INPUT
% f:        function handle of rhs
% T:        final time
```

```
% y_0:       initial value
% steps:     number of time steps
% b:         b-vector
% A:         A-matrix
% OUTPUT
% y_T:       approximated solution at time T

s=length(b);
h=T/steps;
y_T=y_0;
for n=1:steps
    k=zeros(s,length(y_0));
    for i=1:s
        k(i,:)=f(y_T+h*(A(i,:)*k)');
    end
    y_T=y_T+h*(b*k)';
end
end
```

3.2 High-order Runge-Kutta methods

The following methods were constructed by Fehlberg in [2] and are examples of explicit Runge-Kutta methods of order 5 to 8. All these methods are part of embedded Runge-Kutta methods which allow an efficient adaptive step size control. For more details see [2].

0	0	0	0	0	0	0
$\frac{1}{6}$	$\frac{1}{6}$	0	0	0	0	0
$\frac{4}{15}$	$\frac{4}{75}$	$\frac{16}{75}$	0	0	0	0
$\frac{2}{3}$	$\frac{5}{6}$	$\frac{-8}{3}$	$\frac{5}{2}$	0	0	0
$\frac{4}{5}$	$\frac{-8}{5}$	$\frac{144}{25}$	-4	$\frac{16}{25}$	0	0
1	$\frac{361}{320}$	$\frac{-18}{5}$	$\frac{407}{128}$	$\frac{-11}{80}$	$\frac{55}{128}$	0
	$\frac{31}{384}$	0	$\frac{1125}{2816}$	$\frac{9}{32}$	$\frac{125}{768}$	$\frac{5}{66}$

Figure 3.2: 5th order Runge-Kutta method

0	0	0	0	0	0	0	0	0
$\frac{2}{33}$	$\frac{2}{33}$	0	0	0	0	0	0	0
$\frac{4}{33}$	0	$\frac{4}{33}$	0	0	0	0	0	0
$\frac{2}{11}$	$\frac{1}{22}$	0	$\frac{3}{22}$	0	0	0	0	0
$\frac{1}{2}$	$\frac{43}{64}$	0	$\frac{-165}{64}$	$\frac{77}{32}$	0	0	0	0
$\frac{2}{3}$	$\frac{-2383}{486}$	0	$\frac{1067}{54}$	$\frac{-26312}{1701}$	$\frac{2176}{1701}$	0	0	0
$\frac{6}{7}$	$\frac{10077}{4802}$	0	$\frac{-5643}{686}$	$\frac{116259}{16807}$	$\frac{-6240}{16807}$	$\frac{1053}{2401}$	0	0
1	$\frac{-733}{176}$	0	$\frac{141}{8}$	$\frac{-335763}{23296}$	$\frac{216}{77}$	$\frac{-4617}{2816}$	$\frac{7203}{9152}$	0
	$\frac{77}{1440}$	0	0	$\frac{1771561}{6289920}$	$\frac{32}{105}$	$\frac{243}{2560}$	$\frac{16807}{74880}$	$\frac{11}{270}$

Figure 3.3: 6th order Runge-Kutta method

0	0	0	0	0	0	0	0	0	0	0
$\frac{2}{33}$	$\frac{2}{33}$	0	0	0	0	0	0	0	0	0
$\frac{4}{33}$	0	$\frac{4}{33}$	0	0	0	0	0	0	0	0
$\frac{2}{11}$	$\frac{1}{22}$	0	$\frac{3}{22}$	0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{43}{64}$	0	$\frac{-165}{64}$	$\frac{77}{32}$	0	0	0	0	0	0
$\frac{2}{3}$	$\frac{-2383}{486}$	0	$\frac{1067}{54}$	$\frac{-26312}{1701}$	$\frac{2176}{1701}$	0	0	0	0	0
$\frac{6}{7}$	$\frac{10077}{4802}$	0	$\frac{-5643}{686}$	$\frac{116259}{16807}$	$\frac{-6240}{16807}$	$\frac{1053}{2401}$	0	0	0	0
1	$\frac{-733}{176}$	0	$\frac{141}{8}$	$\frac{-335763}{23296}$	$\frac{216}{77}$	$\frac{-4617}{2816}$	$\frac{7203}{9152}$	0	0	0
0	$\frac{15}{352}$	0	0	$\frac{-5445}{46592}$	$\frac{18}{77}$	$\frac{-1215}{5632}$	$\frac{1029}{18304}$	0	0	0
1	$\frac{-1833}{352}$	0	$\frac{141}{8}$	$\frac{-51237}{3584}$	$\frac{18}{7}$	$\frac{-729}{512}$	$\frac{1029}{1408}$	0	1	0
	$\frac{11}{864}$	0	0	$\frac{1771561}{6289920}$	$\frac{32}{105}$	$\frac{243}{2560}$	$\frac{16807}{74880}$	0	$\frac{11}{270}$	$\frac{11}{270}$

Figure 3.4: 7th order Runge-Kutta method

0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\frac{2}{27}$	$\frac{2}{27}$	0	0	0	0	0	0	0	0	0	0	0	0
$\frac{1}{9}$	$\frac{1}{36}$	$\frac{1}{12}$	0	0	0	0	0	0	0	0	0	0	0
$\frac{1}{6}$	$\frac{1}{24}$	0	$\frac{1}{8}$	0	0	0	0	0	0	0	0	0	0
$\frac{5}{12}$	$\frac{5}{12}$	0	$\frac{-25}{16}$	$\frac{25}{16}$	0	0	0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{20}$	0	0	$\frac{1}{4}$	$\frac{1}{5}$	0	0	0	0	0	0	0	0
$\frac{5}{6}$	$\frac{-25}{108}$	0	0	$\frac{125}{108}$	$\frac{-65}{27}$	$\frac{125}{54}$	0	0	0	0	0	0	0
$\frac{1}{6}$	$\frac{31}{300}$	0	0	0	$\frac{61}{225}$	$\frac{-2}{9}$	$\frac{13}{900}$	0	0	0	0	0	0
$\frac{2}{3}$	2	0	0	$\frac{-53}{6}$	$\frac{704}{45}$	$\frac{-107}{9}$	$\frac{67}{90}$	3	0	0	0	0	0
$\frac{1}{3}$	$\frac{-91}{108}$	0	0	$\frac{23}{108}$	$\frac{-976}{135}$	$\frac{311}{54}$	$\frac{-19}{60}$	$\frac{17}{6}$	$\frac{-1}{12}$	0	0	0	0
1	$\frac{2383}{4100}$	0	0	$\frac{-341}{164}$	$\frac{4496}{1025}$	$\frac{-301}{82}$	$\frac{2133}{4100}$	$\frac{45}{82}$	$\frac{45}{164}$	$\frac{18}{41}$	0	0	0
0	$\frac{3}{205}$	0	0	0	0	$\frac{-6}{41}$	$\frac{-3}{205}$	$\frac{-3}{41}$	$\frac{3}{41}$	$\frac{6}{41}$	0	0	0
1	$\frac{-1777}{4100}$	0	0	$\frac{-341}{164}$	$\frac{4496}{1025}$	$\frac{-289}{82}$	$\frac{2193}{4100}$	$\frac{51}{82}$	$\frac{33}{164}$	$\frac{12}{41}$	0	1	0
0		0	0	0	0	$\frac{34}{105}$	$\frac{9}{35}$	$\frac{9}{35}$	$\frac{9}{280}$	$\frac{9}{280}$	0	$\frac{41}{840}$	$\frac{41}{840}$

Figure 3.5: 8th order Runge-Kutta method

3.3 Gauss-Legendre quadrature

In this section we give a short overview of the Gauss-Legendre quadrature, we follow [3] and also refer to this book for more details.

Let $d \in \mathbb{N}$, $a, b \in \mathbb{R}$ with $a < b$ and a sufficiently regular mapping $f : [a, b] \to \mathbb{R}^d$. Then for $n \in \mathbb{N}$, $\mathbf{x} \in [a, b]^n$ and $\mathbf{w} \in [0, 1]^n$ we call

$$I^n_{\mathbf{x},\mathbf{w}}(f) = \sum_{i=1}^{n} w_i f(x_i) \tag{3.3}$$

an n-step quadrature formula with nodes $\mathbf{x}$ and weights $\mathbf{w}$. The idea of numerical quadrature is to choose the weights and nodes in such a way that the quadrature formula $I^n_{\mathbf{x},\mathbf{w}}(f)$ is in some sense a good approximation of the the integral of f on the interval $[a, b]$, i.e.

$$I^n_{\mathbf{x},\mathbf{w}}(f) \approx \int_a^b f(x)dx. \tag{3.4}$$

The Gauss-Legendre quadrature assumes that f is approximately polynomial and then chooses $\mathbf{x}$ and $\mathbf{w}$ such that the associated quadrature formula $I(f)^n_{\mathbf{x},\mathbf{w}}$ is exact for all polynomials of degree at most $2n - 1$. To achieve this one introduces for $k = \{0, 1, 2, \dots\}$ the normalized Legendre polynomials $p_k : [-1, 1] \to [-1, 1]$ given by

$$p_k(x) = \frac{k!}{(2k)!} \frac{d^k}{dx^k} \left(x^2 - 1\right)^k. \tag{3.5}$$

Then for $a = -1$ and $b = 1$, the Gauss-Legendre nodes $\mathbf{x}$ and weights $\mathbf{w}$ are the unique points satisfying

- $p_n(x_k) = 0$ for all $k = 1, \dots, n$
- $\sum_{i=1}^{n} p_k(x_i) w_i = \mathbb{1}_{\{k=0\}}$ for all $k = 1, \dots, n$.

It can be shown that this choice of nodes and weights results in a quadrature formula on the interval $[-1, 1]$ which is exact for polynomials up to degree at most $2n - 1$ (see Theorem 3.6.12 in [3]). For a general interval $[a, b]$ we can simply shift the nodes and rescale the weights appropriately.

A good Matlab implementation that determines the Gauss-Legendre nodes and weights is the freely available open source function `lgwt` written by Greg von Winckel (see [7]).

3.4 Bootstrapping

This section mainly follows [6]. Bootstrapping is a method that allows us to take a one-step method and increase the order by 1. The underlying idea

is very similar to the idea used in the derivation of the Taylor methods (see Section 2.1). Again we begin with the integral form of the ODE (1.2)

$$y(h) = y_0 + \int_{t_0}^{h} f(y(s))ds. \tag{3.6}$$

Next assume that we are given an s-step $(\mathbf{A}, \mathbf{b})$-Runge-Kutta method y_h^{RK} with order of convergence m. We then replace $y(s)$ in (3.6) by y_s^{RK}

$$y(h) \approx y_0 + \int_{t_0}^{h} f\left(y_s^{RK}\right)ds. \tag{3.7}$$

In order to get rid of the integral, we use the r-step Gauss-Legendre quadrature on $[0, 1]$ with nodes $\mathbf{x}$ and weights $\mathbf{w}$ to get

$$\begin{aligned} y_h^{new} &= y_0 + h\sum_{i=1}^{r} w_i f\left(y_{hx_i}^{RK}\right) \\ &= y_0 + h\sum_{i=1}^{r} w_i f\left(y_0 + hx_i \sum_{i=1}^{s} b_i k_i^{hx_i}\right) \end{aligned}$$

where $k_i^{hx_i} = f\left(y_0 + hx_i \sum_{j=1}^{s} a_{i,j}k_j\right)$. Observe that this is the Runge-Kutta method corresponding to

$$\hat{\mathbf{A}} = \begin{pmatrix} x_1\mathbf{A} & \dots & \mathbf{0} & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \dots & x_r\mathbf{A} & \mathbf{0} \\ x_1\mathbf{b}^\top & \dots & \mathbf{0} & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \dots & x_r\mathbf{b}^\top & \mathbf{0} \end{pmatrix} \in \mathbb{R}^{r(s+1)\times r(s+1)} \tag{3.8}$$

$$\hat{\mathbf{b}}^\top = \begin{pmatrix} \mathbf{0} & \dots & \mathbf{0} & \mathbf{w}^\top \end{pmatrix} \in \mathbb{R}^{1\times r(s+1)} \tag{3.9}$$

Given that r is chosen such that the quadrature formula converges with order m, we can hence construct a new method with order $m + 1$ from the old method. A Matlab implementation is given in Listing 3.2.

Listing 3.2: Matlab implementation of the bootstrapping method

```
function [ b_new, A_new ] = bootstrapButcher( b, A, r )
% bootstrapButcher: constructs the b-vector and A-matrix by
    bootstrapping (A,b)-RK method with an r-step Gauss-Legendre
    quadrature
% INPUT
% b:      b-vector of old RK-method
% A:      A-matrix of old RK-method
```

```
% r:      number of quadrature nodes
% OUTPUT
% b_new: b-vector of bootstrapped RK-method
% A_new: A-matrix of bootstrapped RK-method

% use lgwt.m (by Greg von Winckel)
[x,w]=lgwt(r,0,1);
s=length(b);
% construct new s
s_new=r*(s+1);
% construct new A
A_new=zeros(s_new);
A_new(1:r*s,1:r*s)=kron(diag(x),A);
A_new(r*s+1:end,1:r*s)=kron(diag(x),b);
% construct new b
b_new=zeros(1,s_new);
b_new(r*s+1:end)=w';
% simplify butcher table
rows_to_remove=[];
for i=1:s_new
    if all(A_new(i,:)==0) && all(A_new(:,i)==0)
        rows_to_remove=[rows_to_remove, i];
    end
end
for i=1:length(rows_to_remove)
    A_new(rows_to_remove(i),:)=[];
    A_new(:,rows_to_remove(i))=[];
    rows_to_remove=rows_to_remove-1;
    s_new=s_new-1;
    b_new(1)=[];
end
end
```

Chapter 4

Extrapolation methods

In this chapter we introduce the explicit extrapolation methods. These turn out to have many desirable properties which make them quite efficient even for high orders. In fact, it turns out that the extrapolation methods we discuss here are explicit Runge-Kutta methods, however due to the way they are constructed, they are not explicitly represented using Butcher tableaus.

The following closely follows [6].

4.1 Motivation

Assume the setting in Section 1.1 with $t_0 = 0$. Fix $\tau \in [0, T]$, let Ψ be a one-step method of order m and denote the exact solution by y. Moreover, let $n_1, n_2, \cdots \in \mathbb{N}$ be an increasing sequence and set for all $i \in \mathbb{N}$, $h_i = \tau/n_i$. Then we define

$$y_{h_i}(\tau) := \underbrace{\Psi^{h_i}\left(\cdots\left(\Psi^{h_i}(y_0)\right)\right)}_{n_i \text{ compositions of } \Psi^{h_i}} \tag{4.1}$$

the approximated solution at τ using the one-step method Ψ with time-step size h_i. Observe that

$$y(\tau) = \lim_{i\to\infty} y_{h_i}(\tau). \tag{4.2}$$

The idea behind the extrapolation methods is to fix $k \in \mathbb{N}$ and find an interpolating polynomial p such that

$$p(h_i) = y_{h_i}(\tau) \quad \text{for all } i = 1, \ldots, k \tag{4.3}$$

and then use $p(0)$ as a new approximation of $y(\tau)$.

In order for this to be a sensible approximation, we need that $y_h(\tau)$ (as a function of h) can be approximated by a polynomial. This is made precise in the following definition.

Definition 4.1 ((m, k, ω)-asymptotic expansion) *Assume the setting in Section 1.1 with $t_0 = 0$, let $m, k \in \mathbb{N}$ and Ψ be a one-step method of order m and assume that there exist $\omega \in \mathbb{N}$, $\varepsilon \in (0, \infty)$, $e_0, e_1, \dots, e_{k-1} \in \mathbb{R}^d$ and a function $R_k : [0, T] \times (0, \varepsilon) \to \mathbb{R}^d$ such that for all $h \in (0, \varepsilon)$ with $\frac{\tau}{h} \in \mathbb{N}$ it holds that*

$$y_h(\tau) = y(\tau) + e_0 h^m + e_1 h^{m+\omega} + \cdots + e_{k-1} h^{m+(k-1)\omega} + R_k(\tau, h) h^{m+k\omega}.$$

Then we say Ψ has an (m, k, ω)-asymptotic expansion and we denote by $\bar{p}$ the polynomial satisfying for all $h \in \mathbb{R}$ that

$$\bar{p}(h) = y(\tau) + e_0 h^m + e_1 h^{m+\omega} + \cdots + e_{k-1} h^{m+(k-1)\omega}. \tag{4.4}$$

Remark 4.2 *For any $k \in \mathbb{N}$ it can be shown that a one-step method of order m has an $(m, k, 1)$-asymptotic expansion (see Satz 4.37 [6]). If the numerical approximation is additionally reversible, it also has an $(m, k, 2)$-asymptotic expansion (see Satz 4.42 [6]). In both cases it holds that the remainder term satisfies uniformly in $0 < h \leq \tau$*

$$R_k(\tau, h) = \mathcal{O}(\tau). \tag{4.5}$$

Given a one-step method for which there exists an (m, k, ω)-asymptotic expansion with polynomial part $\bar{p}$ we set

$$\begin{aligned} \mathcal{V}_{k+1}^{m,\omega} := \{ q \in C^0\left(\mathbb{R}, \mathbb{R}^d\right) : \exists \alpha_0, \dots, \alpha_k \text{ s.t.} \\ q(x) = \alpha_0 + \alpha_1 x^m + \cdots + \alpha_k x^{m+(k-1)\omega} \} \end{aligned} \tag{4.6}$$

and choose the interpolating polynomial p such that it satisfies

(i) $p \in \mathcal{V}_{k+1}^{m,\omega}$,

(ii) $p(h_i) = y_{h_i}(\tau)$ for all $i = 1, \dots, k$.

It can be shown that a unique such interpolating polynomial always exists given that the h_i in (ii) are distinct (see Lemma 4.33. [6]).

Using the (m, k, ω)-asymptotic expansion, it follows for all $i = 1, \dots, k+1$ that the polynomial $q := p - \bar{p}$ satisfies

$$q(h_i) = R_k(\tau, h_i) h_i^{m+k\omega}. \tag{4.7}$$

Therefore q is an interpolating polynomial of the function $R_k(\tau, \cdot)(\cdot)^{m+k\omega}$ and since $\bar{p}(0) = y(\tau)$ we get using the error estimate for interpolation (see Lemma 4.33 [6]) that

$$|q(0)| = |p(0) - y(\tau)| \leq C_{h_1, \dots, h_{k+1}} \max_{1 \leq i \leq k+1} |R_k(\tau, h_i) h_i^{m+k\omega}|. \tag{4.8}$$

Hence using (4.5) we get for $\omega \in \{1, 2\}$ that

$$|p(0) - y(\tau)| = \mathcal{O}(\tau^{m+k\omega+1}). \tag{4.9}$$

Now fix an increasing sequence $n_1, \dots, n_k \in \mathbb{N}$ and set $\mathcal{F} := \{n_1, \dots, n_k\}$. Then the previous discussion motivates the following definition.

Definition 4.3 ($\mathcal{F}$-extrapolation methods) *Assume the setting in Section 1.1 with $t_0 = 0$, let $m \in \mathbb{N}$, Ψ be a one-step method of order m, and let $p \in \mathcal{V}_{k+1}^{m,w}$ satisfying for all $n \in \mathcal{F}$ that $p(\frac{h}{n}) = y_{\frac{h}{n}}(h)$. Then we call the one-step method defined by*

$$y_h^{\mathcal{F}} = p(0) \tag{4.10}$$

an $\mathcal{F}$-extrapolation method associated to Ψ (an $\mathcal{F}$-extrapolation method with base method Ψ).

Observe that if we use a base method with order m and $m = \omega$ then the interpolation can be performed using classical interpolation techniques (e.g. Aitken-Neville see Section 4.2) with $\tilde{h}_i = (h_i)^\omega$. Therefore such methods are generally more efficient, compared to methods with $m \neq \omega$ for which a more sophisticated interpolation needs to be performed. One class of extrapolation methods where $m = \omega = 1$ is illustrated in Section 4.3. There the explicit Euler method is used as base method. A more advanced class of extrapolation methods where $m = \omega = 2$ is given in Section 4.4. There the explicit midpoint method together with an Euler starting step is used as base method.

Nevertheless, it is also possible to use high order methods as base methods. Take for example a one-step method y_h of order m. Then for $\mathcal{F} = \{1, 2\}$ we can construct the corresponding $\mathcal{F}$-extrapolation method explicitly. Let $p(h) = \alpha_0 + \alpha_1 h^m$ be the interpolating polynomial, then it has to satisfy

$$p(h) = y_h(h), \quad p(\tfrac{h}{2}) = y_{\frac{h}{2}}(h). \tag{4.11}$$

Solving this system of equations leads to

$$y_h^{\{1,2\}} = p(0) = \alpha_0 = \frac{2^m p(\frac{h}{2}) - p(h)}{2^m - 1}. \tag{4.12}$$

It can be shown that if y_h is a s-step Runge-Kutta method then $y_h^{\{1,2\}}$ is a $(3s-1)$-step Runge-Kutta method.

4.2 Aitken-Neville interpolation

The following is intended as a short overview of the Aitken-Neville interpolation scheme, which we use to perform the interpolation in the methods described in Section 4.3 and Section 4.4. For more details see [4].

For $n \in \mathbb{N}$, let $x_1, \dots, x_{n+1} \in \mathbb{R}$ be pairwise distinct points and let $z_1, \dots, z_{n+1} \in \mathbb{R}^d$. Then it is a well known fact that there exist unique polynomial $p(x) = \alpha_0 + \alpha_1 x + \cdots + \alpha_n x^n$ satisfying

$$p(x_i) = z_i \quad \text{for all } i = 1, \dots, n+1. \tag{4.13}$$

The Aitken-Neville interpolation scheme allows for fixed $x \in \mathbb{R}$ to recursively calculate the value of the interpolation polynomial at the point x (i.e. $p(x)$). Start by setting $p_{i,0} := y_i$ for all $i \in \{1, \dots, n+1\}$. Then for all $k \in \{1, \dots, n\}$ and all $i \in \{1, \dots, n+1-k\}$ set

$$p_{i,k+1}(x) = p_{i+1,k}(x) + \frac{x_{i+k+1} - x}{x_{i+k+1} - x_i}(p_{i,k}(x) - p_{i+1,k}(x)) \tag{4.14}$$

Then $p_{1,n+1}(x) = p(x)$.

A Matlab implementation of the Aitken-Neville Scheme is given in Listing 4.1.

Listing 4.1: Matlab implementation of Aitken-Neville Scheme

```
function [ pval ] = aitken_neville( x, y, val )
% aitken_neville: calculates p(val) where p is the unique polynomial
    of degree length(x) s.t. p(x(i))=y(i) for all i
% INPUT
% x:    interpolation nodes
% y:    interpolation values
% val:  evaluation point of interpolation polynomial
% OUTPUT
% pval: p(val) where p is interpolation polynomial

AN_step=y;
n=length(x);
for k=1:n
    for i=1:n-k
        AN_step(:,i)=AN_step(:,i+1)+(x(i+k)-val)/(x(i+k)-x(i))*(
    AN_step(:,i)-AN_step(:,i+1));
    end
end
pval=AN_step(:,1);
end
```

4.3 Euler extrapolation method

For $k \in \mathbb{N}$, set $\mathcal{F}_k := \{1, \dots, k+1\}$. Since the explicit Euler method has order 1 and in view of Remark 4.2, it holds that it has a $(1, k, 1)$-asymptotic expansion. Therefore, using the considerations in Section 4.1 (in particular (4.9)), the extrapolation methods $y_h^{\mathcal{F}_k}$ have order $k+1$. Furthermore, since $\mathcal{V}_{k+1}^{1,1}$ is

just the space of polynomial of degree k with d-dimensional coefficients, we can apply classical interpolation techniques to find the interpolating polynomial p.

The Matlab function `euler_extrapolation` in Listing 4.2 realizes one step of the extrapolation method associated to the explicit Euler method. The interpolation is performed using the Aikten-Neville Algorithm described in Section 4.2.

Listing 4.2: Matlab implementation of the class of extrapolation methods associated to explicit Euler method

```
function [ y_h ] = euler_extrapolation( f, y_0, h, k )
% euler_extrapolation: Euler-extrapolation one-step method with
    arbirary order
% INPUT
% f:     function handle of rhs
% y_0:   initial value
% h:     timestep size
% k:     number of interpolations (order=k+1)
% OUTPUT
% y_h:   approximation resulting from one step of k-Euler-
    extrapolation

F=1:(k+1);
sigma=h./F;
% apply explicit euler method as base method
interpolation_pts=repmat(y_0,1,k+1);
for i=1:k+1
    for j=1:(F(i)-1)
        interpolation_pts(:,i)=interpolation_pts(:,i)+sigma(i)*f(
    interpolation_pts(:,i));
    end
end
% find the value of the interpolation polynomial at 0 using Neville
    Aitken algorithm
y_h=aitken_neville(sigma,interpolation_pts,0);
end
```

4.4 Explicit midpoint extrapolation method

As noted in Remark 4.2, there exists an asymptotic expansion with $\omega = 2$ if the base method is reversible. Hence, if we use a method with order 2 which is additionally reversible, we can expect the associated extrapolation methods to increase in order by 2 for each new extrapolation step. One option is to use the explicit midpoint rule

$$y_h = y_0 + hf(y_{\frac{h}{2}}) \tag{4.15}$$

which is a two step method together with the explicit Euler method as a starting step. This results in

$$y_h = y_0 + hf(y_0 + \frac{h}{2}f(y_0)). \tag{4.16}$$

which is the explicit Runge-Kutta midpoint method listed in Figure 3.1. Although this method is not reversible, it can be shown that an extension of (4.16) to the ODE

$$x'(t) = f(y(t))$$
$$y'(t) = f(x(t))$$

is reversible and hence falls into the setting of Remark 4.2. The (2,k,2)-asymptotic expansion then carries over to the one-step method (4.16); for more details see Section 4.3.3 in [6].

For $k \in \mathbb{N}$, set $\mathcal{F}_k := \{2, 4, \ldots, 2(k+1)\}$. Then using the same considerations as in Section 4.1 (in particular (4.9)), the extrapolation methods $y_h^{\mathcal{F}_k}$ associated to the method in (4.16) have order $2k+2$. These methods are sometimes referred to Gragg-Bulirsch-Stoer methods (GBS-methods) in the literature.

In particular, these methods are Runge-Kutta methods with step sizes given by

$$s = (k+1)^2 + 1 = (m/2)^2 + 1 \tag{4.17}$$

where k is the number of extrapolation steps and $m := 2k+2$ is the order of the methods (see Satz 4.46 in [6]).

The Matlab function `midpoint_extrapolation` (see Listing 4.3) is an implementation of these extrapolation methods associated to the explicit RK midpoint method.

Listing 4.3: Matlab implementation of the class of extrapolation methods associated to explicit Runge-Kutta midpoint method

```
function [ y_h ] = midpoint_extrapolation( f, y_0, h, k )
% midpoint_extrapolation: midpoint-extrapolation one-step method
    with arbirary order
% INPUT
% f:     function handle of rhs
% y_0:   initial value
% h:     timestep size
% k:     number of interpolations (order=2*k+2)
% OUTPUT
% y_h:   approximation resulting from one step of k-midpoint-
    extrapolation

F=2*(1:(k+1));
```

```
sigma=h./F;
% apply explicit midpoint method as base method with an euler
    starting step
interpolation_pts=zeros(length(y_0),k+1);
for i=1:k+1
    step1=y_0;
    step2=y_0+sigma(i)*f(y_0);
    for j=1:(F(i)-1)
        temp=step1+2*sigma(i)*f(step2);
        step1=step2;
        step2=temp;
    end
    interpolation_pts(:,i)=step2;
end
% find the value of the interpolation polynomial at 0 using Neville
    Aitken algorithm
y_h=aitken_neville(sigma.^2,interpolation_pts,0);
end
```

Chapter 5

Applications

In this chapter we apply the Taylor methods (Chapter 2), the Runge-Kutta methods (with bootstrapping steps) (Chapter 3) and the extrapolation methods (Chapter 4) to five different examples and illustrate their convergence rates in plots.

The convergence plots are constructed in the following way. First we approximate the exact solution $y(T)$ using the Matlab integrator `ode45` with an absolute tolerance of $10e$-13 and a relative tolerance of $10e$-16. Due to the very low tolerance it is sufficient to take this as the "exact" solution. Then for a fixed one step method we calculate approximations of $y(T)$ using 1 to 10 time steps and denote these by $\bar{y}(T, n)$, where n is the number of time steps. Finally, we plot $\|\bar{y}(T, n) - y(T)\|_1$ against the number of time steps n to get the convergence plot.

5.1 Example 1

The first example we consider, is the ODE given by

$$\begin{cases} y'(t) = -y(t)^5, & t \in [0, 0.1] \\ y(0) = 1. \end{cases} \tag{5.1}$$

5.1.1 Taylor methods

Due to the one-dimensional polynomial form of the right hand side we can use the extremely efficient polynomial implementation of the Taylor methods (see Listing 2.1). This allows the use of very high order Taylor methods (here order 1 to 20), see Figure 5.1.

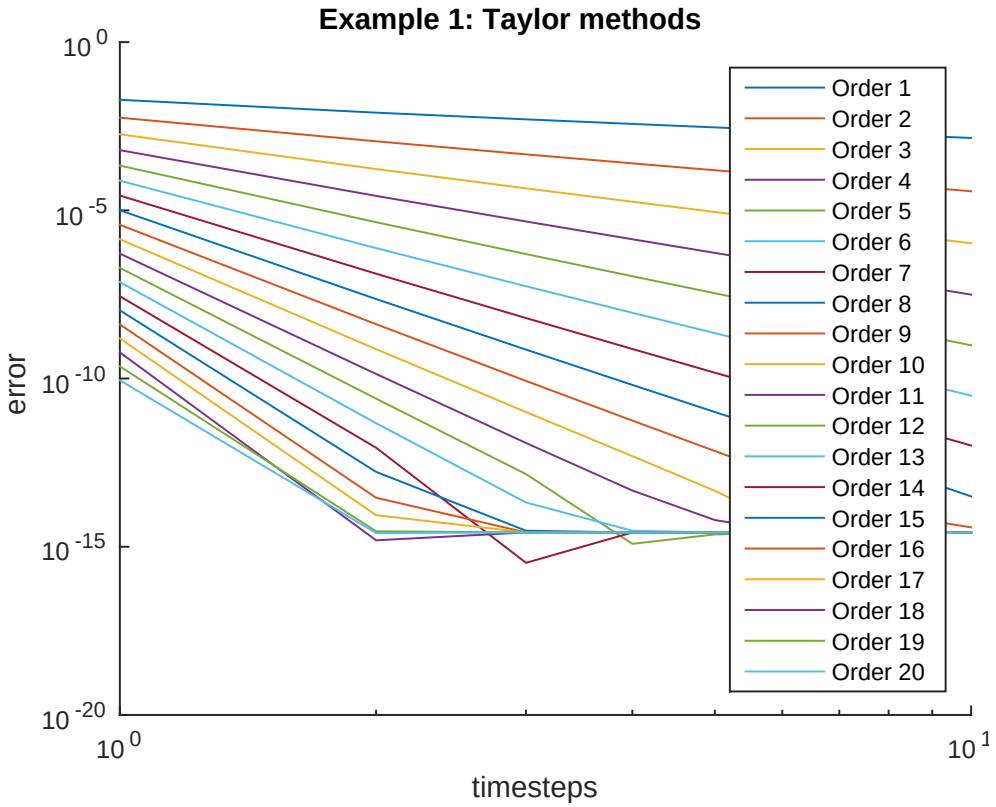

Figure 5.1: Taylor methods of order 1 to 20 applied to Example 1

5.1.2 Runge-Kutta methods

Here we use the Runge-Kutta methods given in Section 3.1 and Section 3.2 of order 1 to 8 and then apply two bootstrapping steps (with Gauss-Legendre quadrature) to the 8^{th} order method to get methods of order 9 and 10. The resulting convergence plot is given in Figure 5.2.

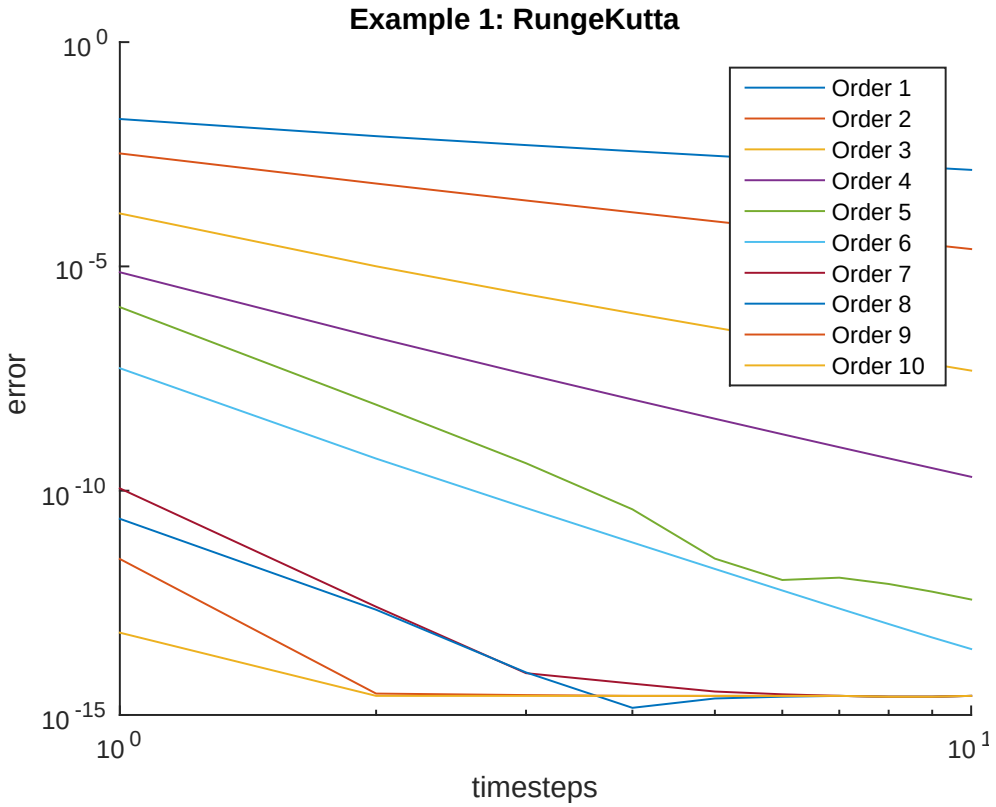

Figure 5.2: Runge-Kutta methods of order 1 to 10 applied to Example 1

5.1.3 Extrapolation methods

To illustrate the extrapolation methods, we use the explicit Euler extrapolation methods (see Section 4.3) with extrapolation steps $\mathcal{F}_k = \{1, 2, .., k+1\}$ for $k = 1, \ldots, 9$ which corresponds to methods of order $2, 3, \ldots, 10$, see Figure 5.3.

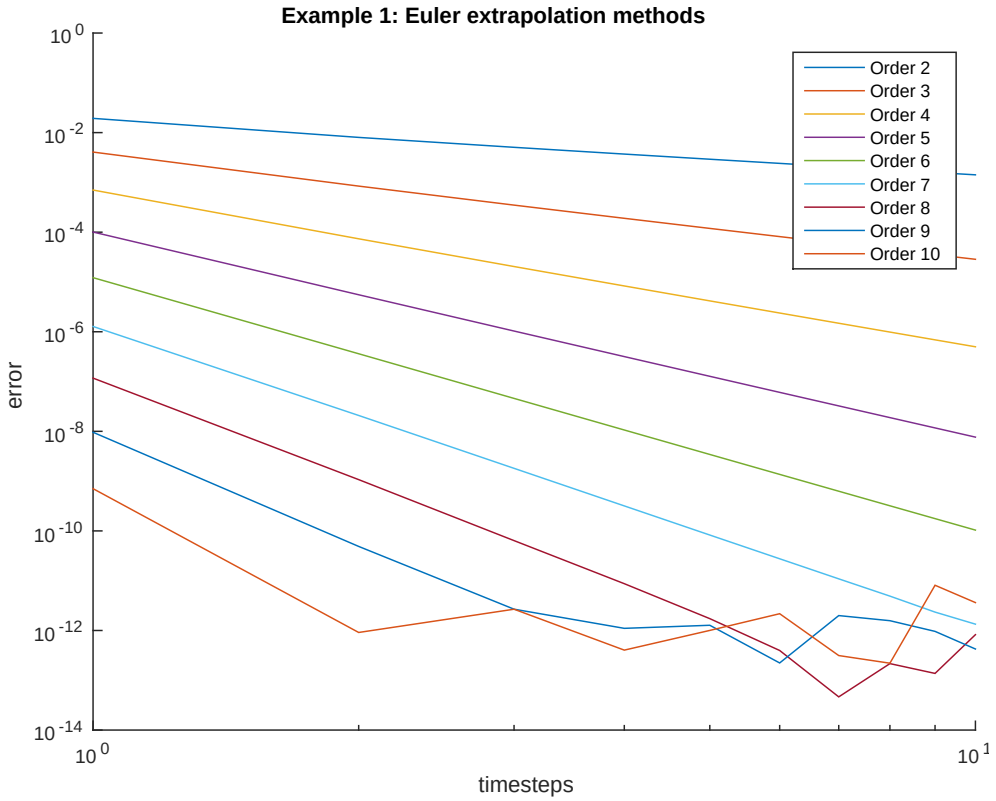

Figure 5.3: Euler extrapolation methods of orders 2,3,...,10 applied to Example 1

5.2 Example 2

Next we take a look at a high-dimensional example. For this let $d \in \mathbb{N}$ and set

$$f(\mathbf{x}) := \begin{pmatrix} x_1^2 - \frac{1}{1+x_2^2} \\ x_2^2 - \frac{1}{1+x_3^2} \\ \vdots \\ x_{d\text{-}1}^2 - \frac{1}{1+x_d^2} \\ x_d^2 - \frac{1}{1+x_1^2} \end{pmatrix} \quad \text{for all } \mathbf{x} \in \mathbb{R}^d. \tag{5.2}$$

In this example we take $d = 20$ and consider the ODE given by

$$\begin{cases} \mathbf{y}'(t) = f(\mathbf{y}(t)), \quad t \in [0, 0.1] \\ y_i(0) = 1, \quad i \in \{1, \dots, d\}. \end{cases} \tag{5.3}$$

5.2.1 Taylor methods

To apply the Taylor methods to this ODE we need the general version using the L-operators, which we derived in Chapter 2. Since these methods rely

on auto-differentiation, they require more f-evaluations and are quite a bit slower. The convergence plot in Figure 5.4 shows the Taylor methods of order 1 to 8.

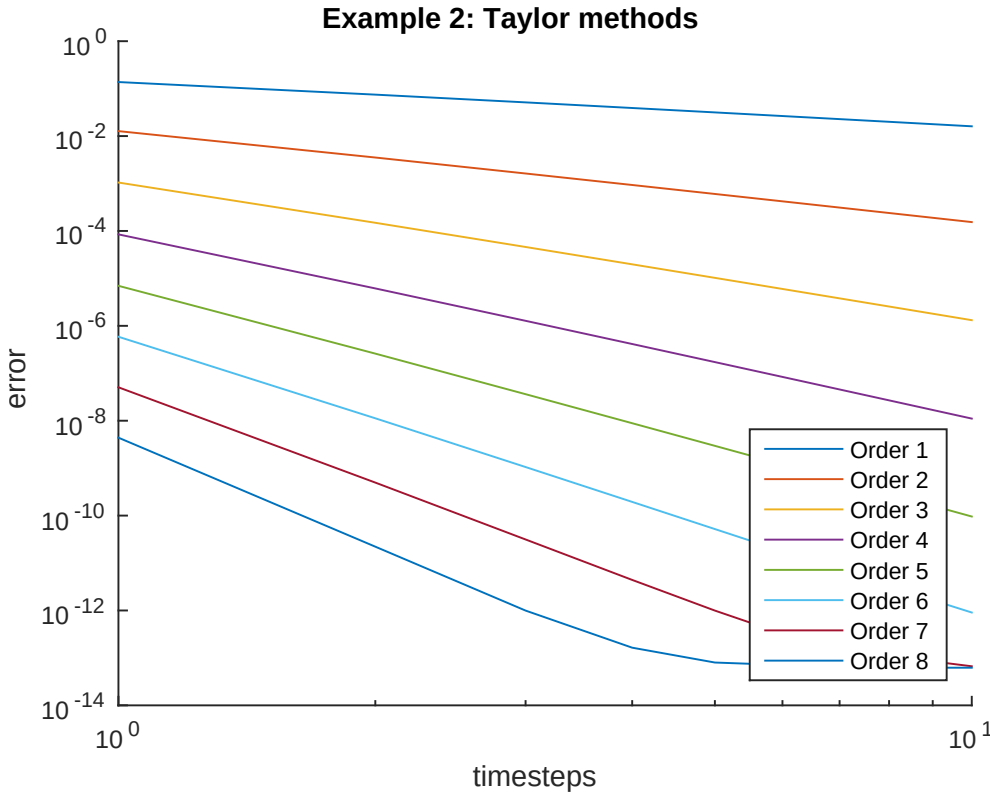

Figure 5.4: Taylor methods of orders 1 to 8 applied to Example 2 with $d = 20$

5.2.2 Runge-Kutta methods

Again we use the Runge-Kutta methods given in Section 3.1 and Section 3.2 of order 1 to 8 and then apply two bootstrapping steps (with Gauss-Legendre quadrature) to the 8^{th} order method to get methods of order 9 and 10. The resulting error plot is given in Figure 5.5.

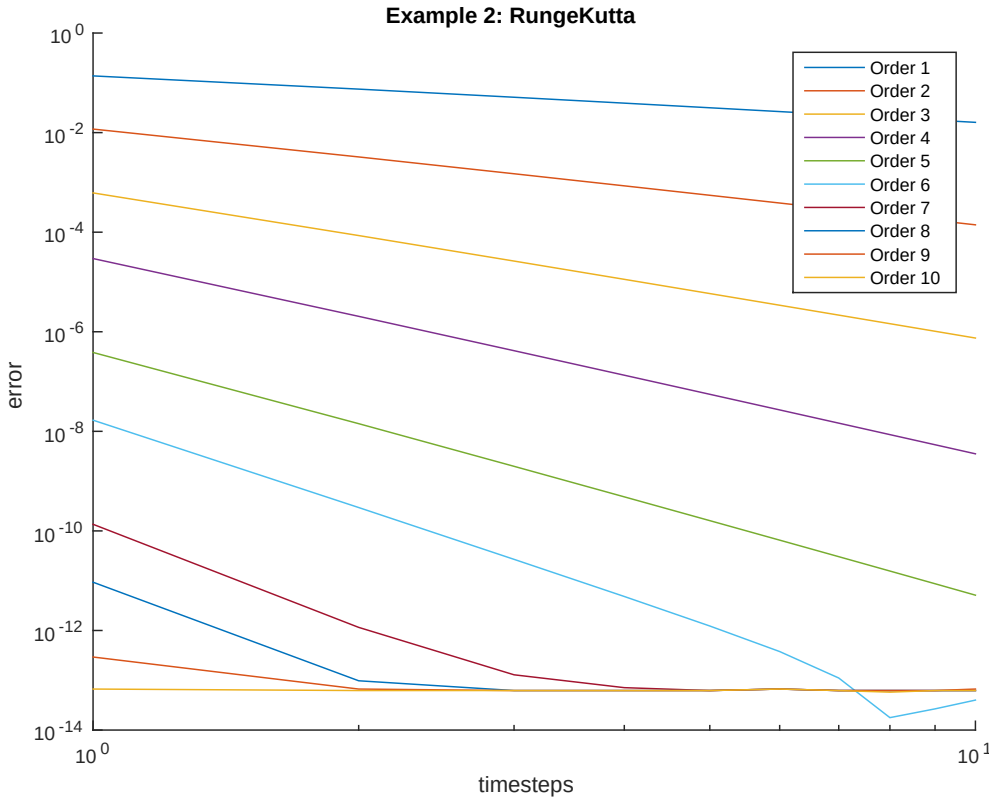

Figure 5.5: Runge-Kutta methods of order 1 to 10 applied to Example 2 with $d = 20$

5.2.3 Extrapolation methods

As in Example 1, we use the explicit Euler extrapolation methods (see Section 4.3) with extrapolation steps $\mathcal{F}_k = \{1, 2, .., k+1\}$ for $k = 1, \ldots, 9$. This corresponds to methods of order $2, 3, \ldots, 10$, see Figure 5.6.

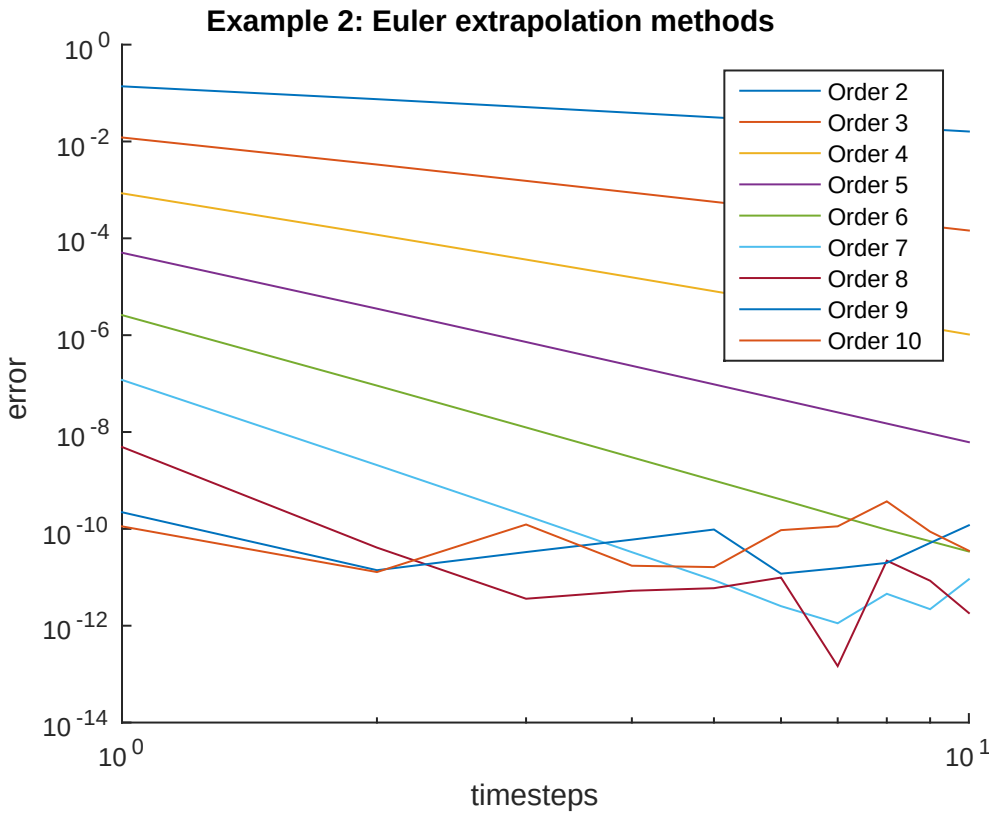

Figure 5.6: Euler extrapolation methods of orders 2,3,...,10 applied to Example 2 with $d = 20$

5.3 Example 3

The third ODE we consider is

$$\begin{cases} y'(t) = \dfrac{1}{1+y(t)^2}, & t \in [0,1] \\ y(0) = 1. \end{cases} \tag{5.4}$$

5.3.1 Taylor methods

Although this right hand side is C^∞ the radius of convergence of the Taylor series is only 1. Therefore it can happen that increasing the order of the Taylor method leads to worse approximations of the solutions if the time interval is too long. The Taylor methods of order 1 to 11 are illustrated in the convergence plot in Figure 5.7.

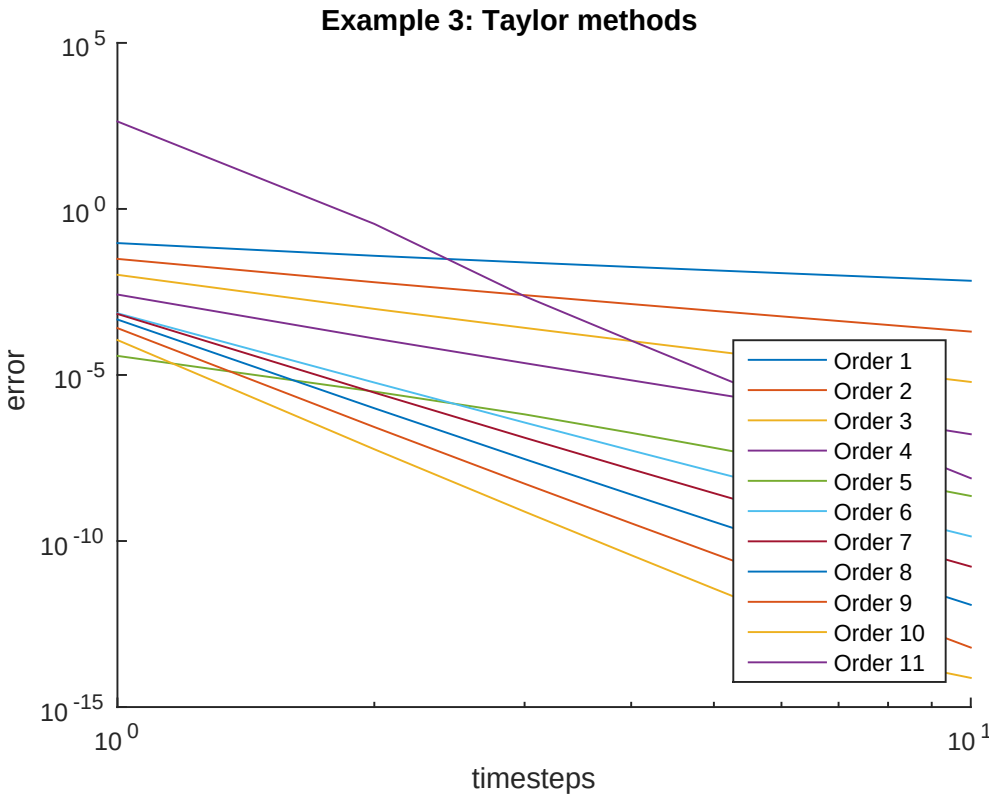

Figure 5.7: Taylor methods of orders 1 to 11 applied to Example 3

5.3.2 Runge-Kutta methods

Here we again use the Runge-Kutta methods given in Section 3.1 and Section 3.2 of order 1 to 8 and then apply two bootstrapping steps (with Gauss-Legendre quadrature) to the 8^{th} order method to get methods of order 9 and 10. The resulting convergence plot is given in Figure 5.8.

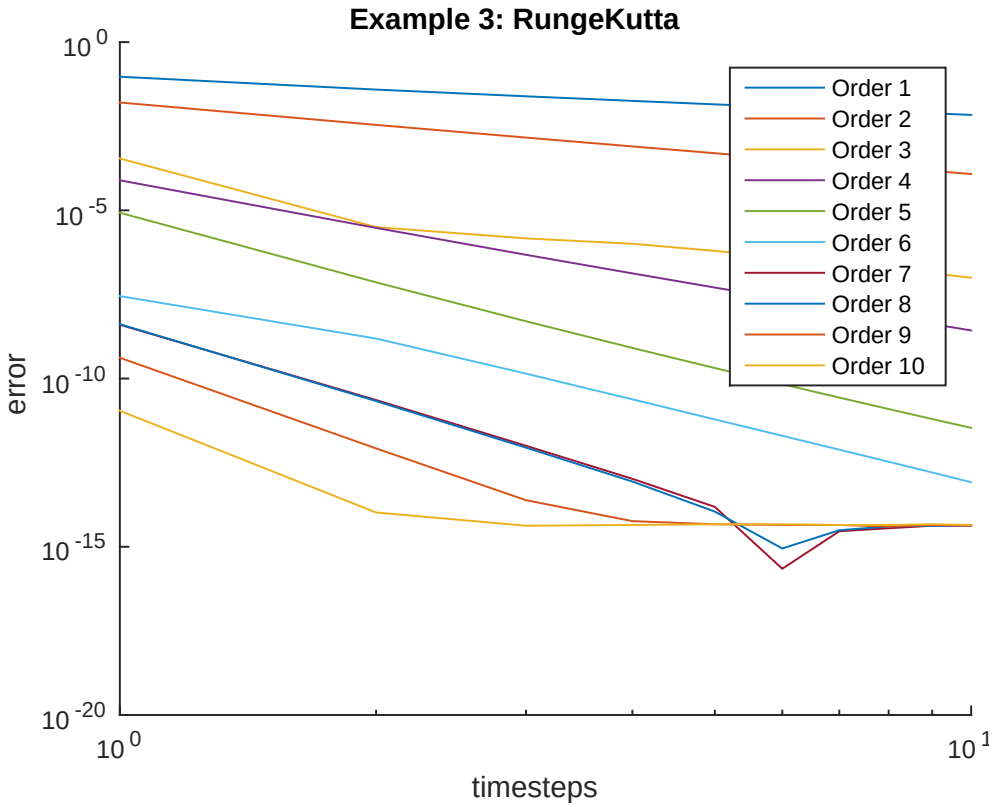

Figure 5.8: Runge-Kutta methods of order 1 to 10 applied to Example 3

5.3.3 Extrapolation methods

This time we use the explicit midpoint extrapolation methods (see Section 4.4) with extrapolation steps $\mathcal{F}_k = \{2, 4, .., 2k+2\}$ for $k = 1, \ldots, 7$ to illustrate the extrapolation methods. These correspond to methods of order $4, 6, \ldots, 16$, since the order increases by 2 for each additional extrapolation step. The convergence plot is shown in Figure 5.9. Note that the extrapolation methods appear to be more stable with respect to an increase in order compared to the Taylor methods.

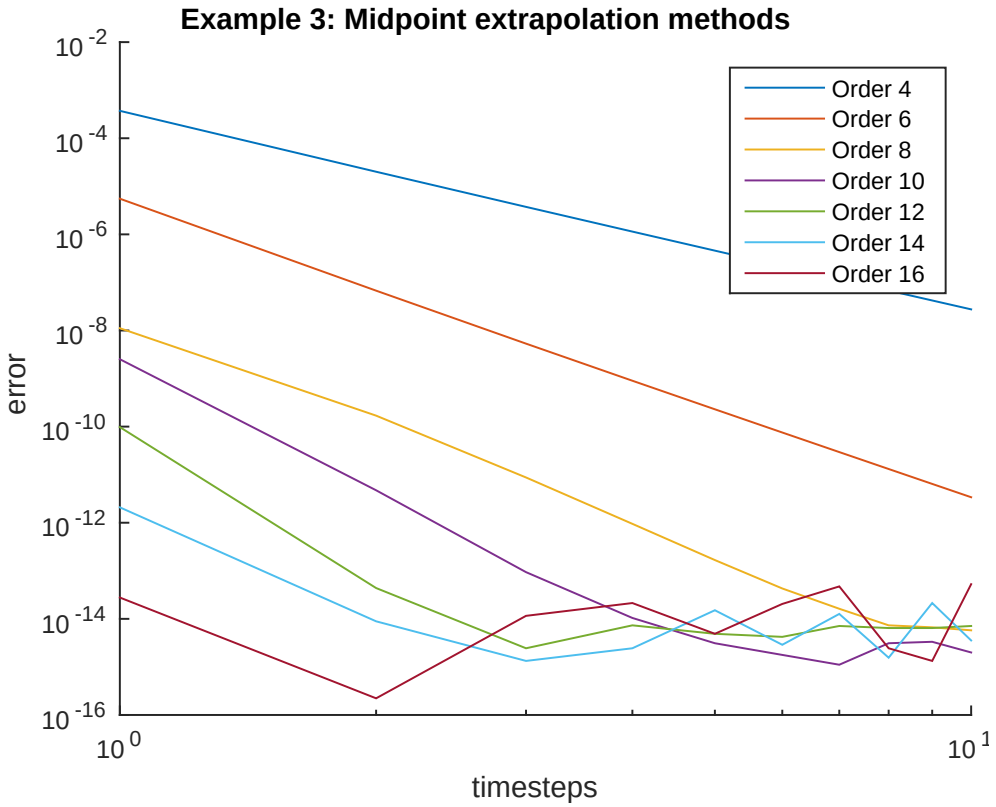

Figure 5.9: Midpoint extrapolation methods of orders 4,6,...,16 applied to Example 3

5.4 Example 4

In this example let $d \in \mathbb{N}$ and consider a d^{th}-order ODE of the form

$$\begin{cases} y^{(d)}(t) = y(t) - y(t)^3, \quad t \in [t_0, T] \\ y^{(i)}(t_0) = 1, \quad i \in \{0, 1, \dots, d-1\}. \end{cases} \tag{5.5}$$

Although this is not an ODE of the form (1.1) it can be converted into our framework by setting

$$\mathbf{z} := \begin{pmatrix} y \\ y' \\ y^{(2)} \\ \vdots \\ y^{(d\text{-}1)} \end{pmatrix} \tag{5.6}$$

and

$$f(\mathbf{x}) := \begin{pmatrix} x_2 \\ x_3 \\ \vdots \\ x_d \\ x_1 - x_1^3 \end{pmatrix} \quad \text{for all } \mathbf{x} \in \mathbb{R}^d. \tag{5.7}$$

Then the last coordinate of the solution of the ODE

$$\begin{cases} \mathbf{z}'(t) = f(\mathbf{z}(t)), & t \in [0,1] \\ z_i(0) = 1, & i \in \{1, \dots, d\}. \end{cases} \tag{5.8}$$

is the solution of the original ODE (5.5), i.e. $y(t) = z_i(t)$. In this example we set $d = 20$.

5.4.1 Taylor methods

Here we use Taylor methods of order 1 to 8. The convergence plot is shown in Figure 5.10.

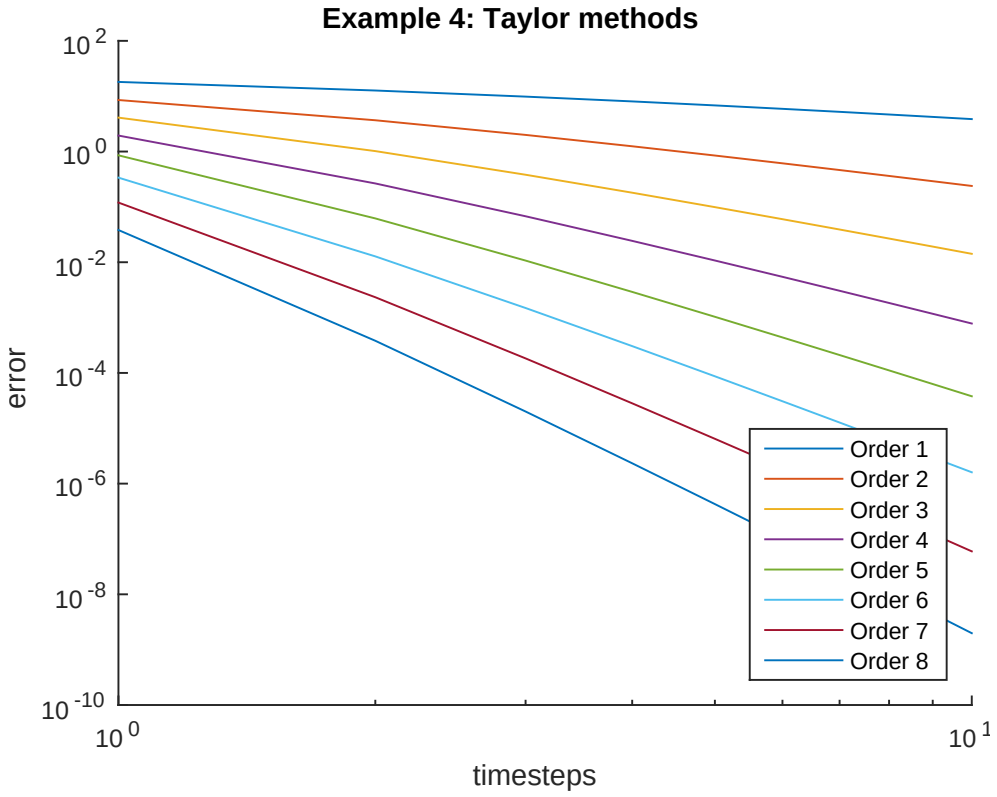

Figure 5.10: Taylor methods of orders 1 to 8 applied to Example 4 with $d = 20$

5.4.2 Runge-Kutta methods

As above, we again use the Runge-Kutta methods given in Section 3.1 and Section 3.2 of order 1 to 8 and then apply two bootstrapping steps (with Gauss-Legendre quadrature) to the 8^{th} order method to get methods of order 9 and 10, see Figure 5.11.

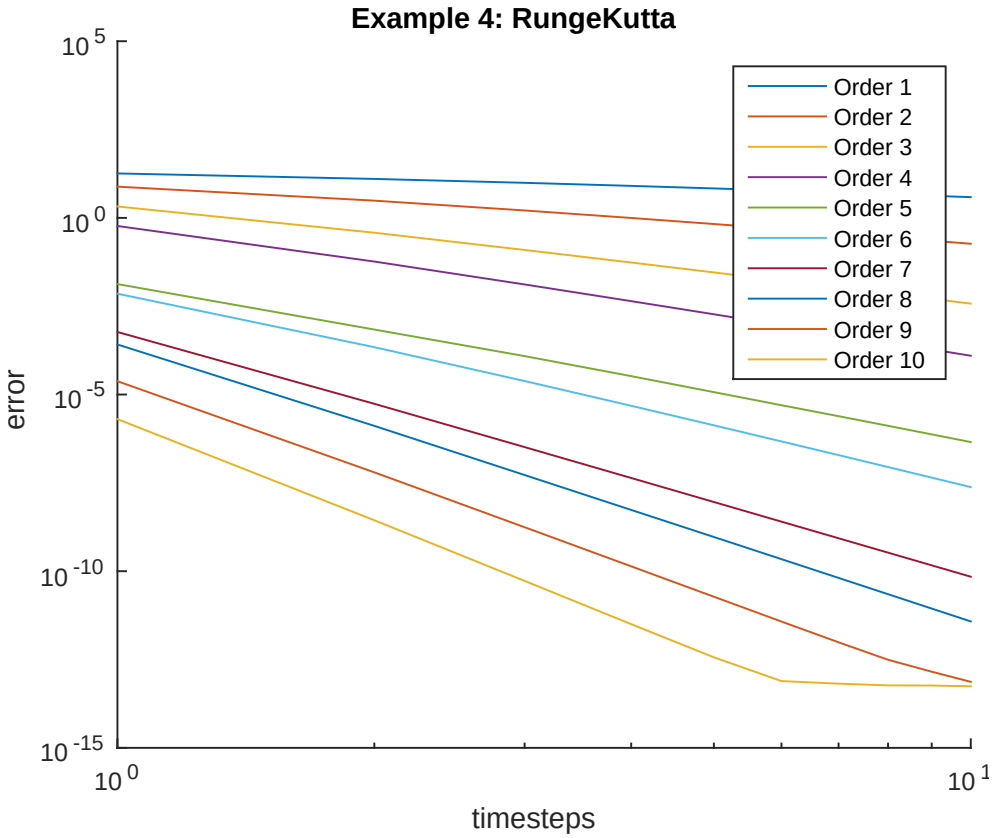

Figure 5.11: Runge-Kutta methods of order 1 to 10 applied to Example 4 with $d = 20$

5.4.3 Extrapolation methods

In this example, we again use the explicit midpoint extrapolation methods with Section 4.4) with extrapolation steps $\mathcal{F}_k = \{2, 4, .., 2k + 2\}$ for $k = 1, \ldots, 9$. The resulting convergence plot is given in Figure 5.12.

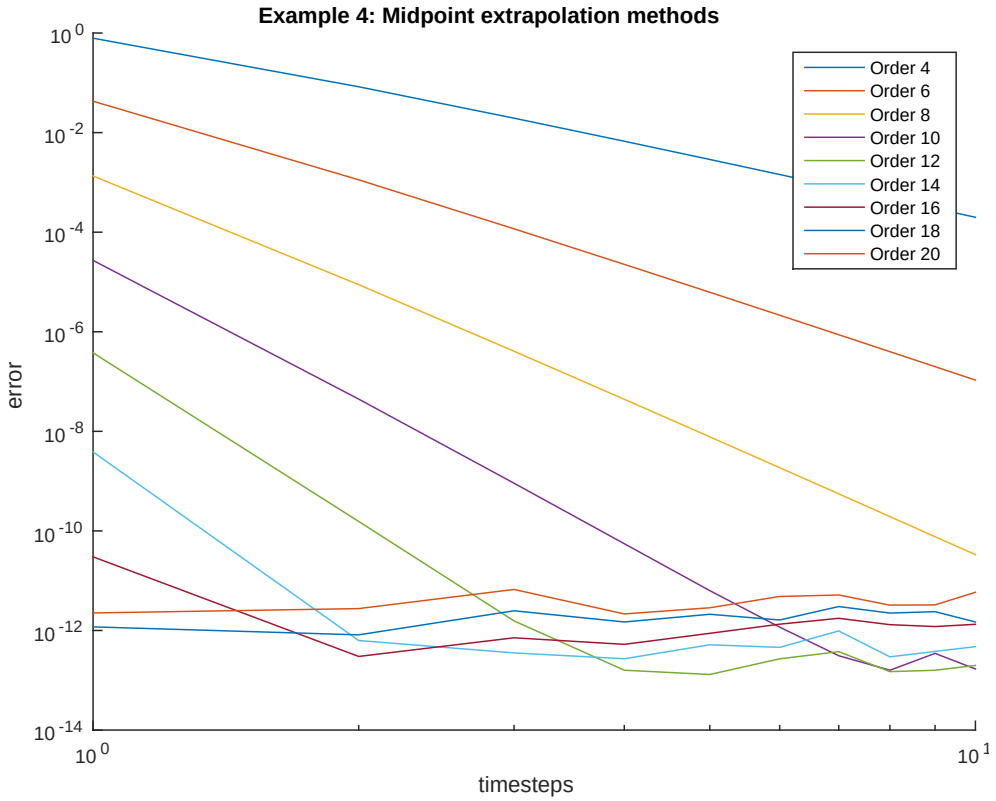

Figure 5.12: Midpoint extrapolation methods of orders 4,6,...,20 applied to Example 4 with $d = 20$

5.5 Example 5

Finally, we examine another high-dimensional ODE. Again let $d \in \mathbb{N}$ and set

$$f(\mathbf{x}) := A\mathbf{x} - \begin{pmatrix} x_1^3 \\ \vdots \\ x_d^3 \end{pmatrix} \quad \text{for all } \mathbf{x} \in \mathbb{R}^d. \tag{5.9}$$

where $A \in \mathbb{R}^{d \times d}$ is the matrix given by

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}. \tag{5.10}$$

In this example we take $d = 50$ and consider the ODE

$$\begin{cases} \mathbf{y}'(t) = f(\mathbf{y}(t)), & t \in [0,1] \\ y_i(0) = 1, & i \in \{1,\dots,d\}. \end{cases} \tag{5.11}$$

5.5.1 Taylor methods

Due to the high dimensionality of this ODE the Taylor methods become very inefficient and slow. Here we only use the Taylor methods of order 1 to 6, see Figure 5.13.

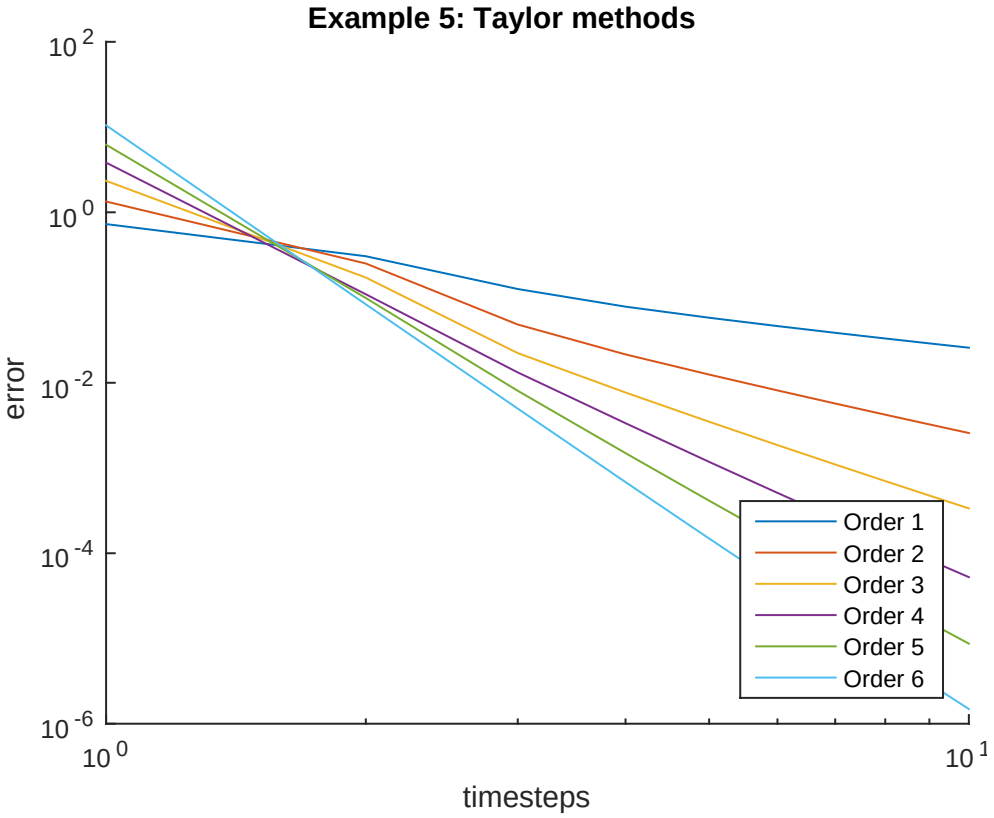

Figure 5.13: Taylor methods of orders 1 to 6 applied to Example 5

5.5.2 Runge-Kutta methods

The Runge-Kutta method are not effected by the dimension as strongly as the Taylor methods. Here we again use the Runge-Kutta methods given in Section 3.1 and Section 3.2 of order 1 to 8 and then apply two bootstrapping steps (with Gauss-Legendre quadrature) to the 8^{th} order method to get methods of order 9 and 10, see Figure 5.14.

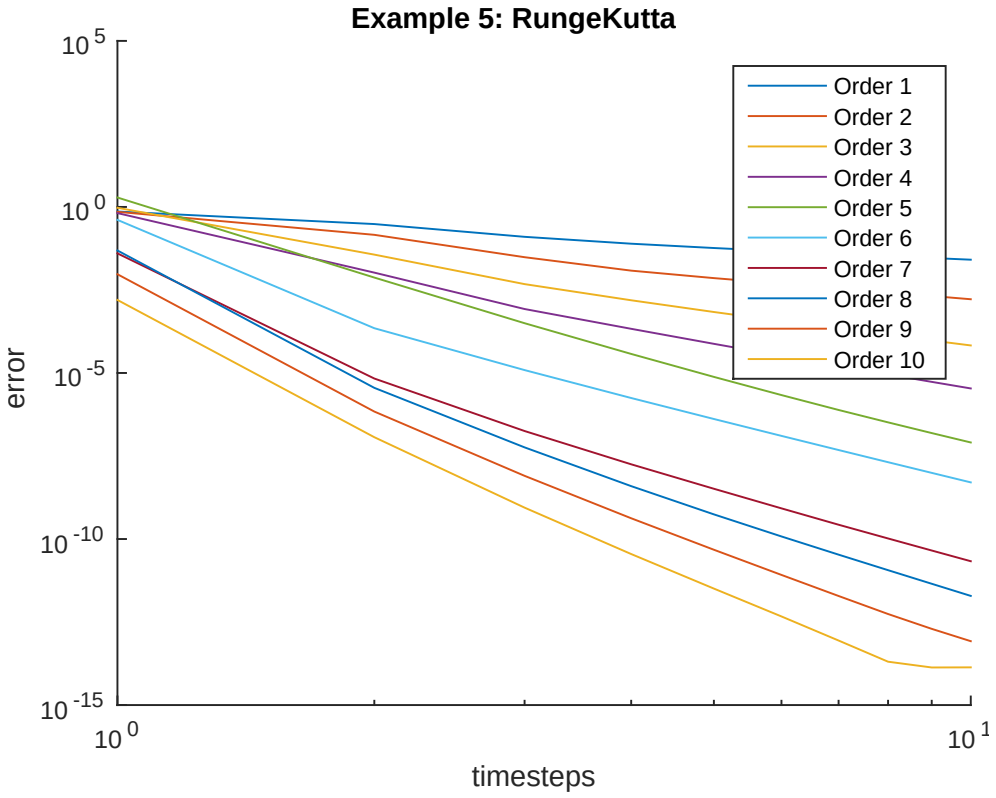

Figure 5.14: Runge-Kutta methods of order 1 to 10 applied to Example 5

5.5.3 Extrapolation methods

The extrapolation methods also remain efficient for high dimensions. For this example we use the explicit midpoint extrapolation methods (see Section 4.4) with extrapolation steps $\mathcal{F}_k = \{2, 4, .., 2k + 2\}$ for $k = 1, \ldots, 9$. The convergence plot is given in Figure 5.15.

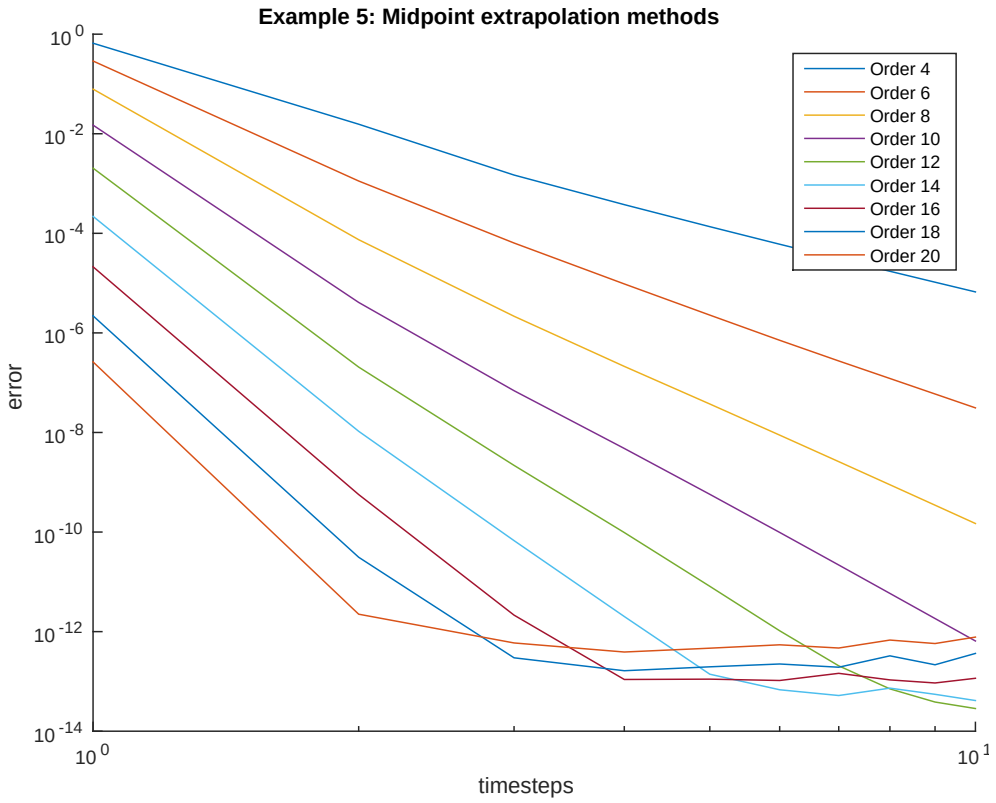

Figure 5.15: Midpoint extrapolation methods of orders 4,6,...,20 applied to Example 5

Bibliography

[1] Syvert Paul Norsett Ernst Hairer, Gerhard Wanner. *Solving Ordinary Differential Equations I*. Springer, 2000.

[2] Erwin Fehlberg. Classical fifth-, sixth-, seventh-, and eight-order runge-kutta formulas with stepsize control. Technical report, George C. Marshall Space Flight Center, NASA, October 1968.

[3] Roland Bulirsch Josef Stoer. *Introduction to Numerical Analysis*. Springer, 2002.

[4] Michael Knorrenschild. *Numerische Mathematik*. Carl Hanser, 2013.

[5] Anil V. Rao Matthew J. Weinstein. Adigator. http://sourceforge.net/projects/adigator/, December 2014.

[6] Folkmar Bornemann Peter Deuflhard. *Numerische Mathematik 2*. De Gruyter, 2013.

[7] Greg von Winckel. Legendre-gauss quadrature weights and nodes. http://www.mathworks.com/matlabcentral/fileexchange/4540-legendre-gauss-quadrature-weights-and-nodes, May 2004.

[8] Wolfgang Walter. *Ordinary Differential Equations*. Springer, 1998.