Automatic Validation of Numerical Solutions

Ole Stauning

LYNGBY 1997

IMM-PHD-1997-36

IMM

Contents

Preface

This thesis has been written in partial fulfilment of the requirements for obtaining the Ph.D. degree. The work was supported by the Technical University of Denmark (DTU) and has been completed at Department of Mathematical Modelling, DTU, from September 1, 1994 to August 31, 1997. In the period from September 1, 1995 to February 29, 1996 I visited Institut für Angewandte Mathematik, Universität Karlsruhe.

I thank my main supervisor Prof. Kaj Madsen at the Department of Mathematical Modelling. Kaj Madsen was the person who introduced me to interval analysis when I as an undergraduate student attended his lectures in “0202 Advanced Numerical Analysis”: In this course, I realized how powerful interval methods are. My thanks extend to my supervisor Dr. Carsten Knudsen at the Department of Applied Mathematical Studies, University of Leeds. He introduced me to problems occuring in discrete dynamical systems, which seem to be impossible to solve by analytical methods, but to some extend can be dealt with using interval methods, as described in my master’s thesis. I have benefited considerably from the many discussions I have had with Kaj Madsen and Carsten Knudsen with respect to interval analysis and mathematics in general.

I am also very grateful to Dr. Rudolf Lohner at the Institut für Angewandte Mathematik. First of all, for allowing me to visit the Institute and for his hospitality during my stay, but also for the many discussions we had about the method behind his famous AWA-program for solving ordinary differential equations obtaining interval bounds of the solution. Rudolf Lohner is also the person who introduced me to automatic differentiation. This introduction has had an enormous influence on my work; and is also reflected by the contents of this thesis. I thank the staff at the Institut für Angewandte Mathematik for their hospitality – my stay in Karlsruhe has been a very nice experience for me.

During the studies of automatic differentiation I have had the pleasure to work with Dr. Claus Bendtsen at Danish Computing Centre for Research and Education, UNI•C. He is the co-author of our packages: FADBAD/TADIFF for performing automatic differentiation, and two technical reports describing the packages. I thank Claus Bendtsen for the many discussions we have had about automatic differentiation and for the many hours we have spent together in developing our packages.

Thanks to Pia Stauning for reading the draft and pointing out a few errors. Last, but not least, I would like to thank my friends and my family for their support during my studies.

OLE STAUNING

Technical University of Denmark
August, 1997

Summary

This thesis is concerned with “Automatic Validation of Numerical Solutions”. The basic theory of interval analysis and self-validating methods is introduced. The mean value enclosure is applied to discrete mappings for obtaining narrow enclosures of the iterates when applying these mappings with intervals as initial values. A modification of the mean value enclosure of discrete mappings is considered, namely the extended mean value enclosure which in most cases leads to even better enclosures. These methods have previously been described in connection with discretizing solutions of ordinary differential equations, but in this thesis, we describe how to use the methods for enclosing iterates of discrete mappings, and then later use them for discretizing solutions of ordinary differential equations.

The theory of automatic differentiation is introduced, and three methods for obtaining derivatives are described: The forward, the backward, and the Taylor expansion methods. The three methods have been implemented in the C++ program packages FADBAD/TADIFF. Some examples showing how to use the three methods are presented. A feature of FADBAD/TADIFF not present in other automatic differentiation packages is the possiblility to combine the three methods in an extremely flexible way. We examine some applications where this flexibility is very useful.

A method for Taylor expanding solutions of ordinary differential equations is presented, and a method for obtaining interval enclosures of the truncation errors incurred, when truncating these Taylor series expansions is described. By combining the forward method and the Taylor expansion method, it is possible to implement the (extended) mean value enclosure of a truncated Taylor series expansion with enclosures of the truncation errors. A C++ program package: ADIODES, using this method has been developed[1].

ADIODES is used to prove existence and uniqueness of periodic solutions to specific ordinary differential equations occuring in dynamical systems theory. These proofs of existence and uniqueness are difficult or impossible to obtain using other known methods. Also, a method for solving boundary value problems is described.

Finally a method for enclosing solutions to a class of integral equations is described. This method is based on the mean value enclosure of an integral op-

[1] ADIODES is an abbreviation of “Automatic Differentiation Interval Ordinary Differential Equation Solver”.

erator and uses interval Bernstein polynomials for enclosing the solution. Two numerical examples are given, using two orders of approximation and using different numbers of discretization points.

Danish summary – Resumé

Denne afhandling omhandler "Automatisk Bevisførelse for Numeriske Løsninger". Den grundlæggende teori for interval analyse samt automatisk differentiation er introduceret. Middelværdiformen er anvendt på en diskret afbildning for at opnå en snæver interval indkapsling af afbildningens iterater. Også en forbedring af middelværdiformen, kaldet den udvidede middelværdiform, er introduceret. Metoderne har før været beskrevet i forbindelse med diskretisering af løsninger til sædvanlige differentialligninger. Her bliver det beskrevet hvordan man bruger metoden til at indkapsle iteraterne af en diskret afbildning, for senere at diskretisere løsninger til sædvanlige differentialligninger.

Teorien bag automatisk differentiation er introduceret og tre metoder: forlæns- og baglæns differentiation samt Taylorudvikling, er beskrevet; de tre metoder er blevet implementeret i C++ programpakkerne FADBAD/TADIFF. Eksempler på brugen af disse programpakker er givet. I programpakkerne FADBAD/TADIFF er det muligt også at bruge kombinationer af metoder. Ingen andre programpakker benytter denne mulighed. Vi vil se nogle applikationer hvor disse kombinationsmuligheder er meget brugbare.

En metode for at Taylorudvikle løsninger til sædvanlige differentialligninger samt beregning af en interval indkapsling af trunkerings fejlen, begået ved trunkering af denne række, er beskrevet. Ved at kombinere forlæns metoden med Taylorudvikling er det muligt at implementere den (udvidede) middelværdiform af en trunkeret Taylorudvikling med en indkapsling af trunkeringsfejlen og en C++ programpakke, ADIODES, der benytter denne metode, er udviklet[2].

ADIODES er brugt til at bevise eksistens samt entydighed af periodiske løsninger til nogle sædvanlige differentialligninger som er af speciel interesse indenfor dynamisk systemteori. Disse beviser er svære – måske umulige, at udføre med andre kendte metoder. Også en metode til løsning af randværdiproblemer er kort beskrevet.

En metode til løsning af en klasse af integralligninger er beskrevet. Denne metode er baseret på middelværdiformen af en integraloperator og bruger interval Bernstein polynomier til indkapsling af løsningen. To eksempler på integralligninger er givet og løsningerne til disse er indkapslet ved brug af to approksimations-ordener samt forskelligt antal af diskretiseringspunkter.

[2]ADIODES er en forkortelse af "Automatic Differentiation Interval Ordinary Differential Equation Solver".

1 Introduction

Computers seem to play a more and more important role in the scientific community. Even new fields of science have emerged because of the invention and development of the computer. However the computer is in many cases not a perfect tool for doing scientific calculations. When using floating point arithmetic, which is a discretization of real arithmetic, the results of the computations performed will usually be affected by rounding errors and in the worst cases leads to completely wrong results. This problem is getting even worse since computers are becoming faster, and it is possible to execute more and more computations within a fixed time, without the standard floating point arithmetic becoming more reliable. Since it is impossible to verify the accuracy of the results generated by some complicated programs, 'by hand', we have to trust the validity of the computations we perform.

A branch of numerical analysis, called interval analysis, is dedicated to this problem [2, 22, 38, 39]. In interval analysis we use intervals of real numbers as the fundamental elements of computation rather than real numbers themselves. We will in Section 2 see how to define operations on intervals so that the result of an interval operation encloses the true results of the corresponding real operation with any combination of real arguments in the corresponding interval arguments. Interval vectors and matrices and operations on these can be defined in the usual way, i.e., the elements of interval matrices and vectors are intervals. The rounding is controlled in every single arithmetic operation when implementing interval arithmetic on a computer. The infimum of the result should be computed while rounding down, and the supremum should be computed while rounding up. Using this outward rounding method, we will always get correct enclosures of the result of an algorithm, when running a computer-based implementation. On most modern computers this rounding control is hardwired into the CPU, and it is just a matter of controlling a register [24]. A public domain C++ program package PROFIL/BIAS for doing interval arithmetic with outwards rounding is available [28, 29, 30]. This package is used to perform all interval computations presented in this report.

Another useful application of interval arithmetic is when some parameters or initial values in an algorithm are not exactly known, but known to lie within some intervals. By implementing algorithms using interval arithmetic and including the uncertain values as intervals, we obtain results which are valid for every combination of real parameters and/or initial values in their respective interval enclosures. This property is important when modelling real systems, where values of parameters are based on measurements performed on a real system and therefore are uncertain due to the uncertaincy of the measurements.

A general problem in interval analysis is to obtain interval results with narrow bounds. It is usually no problem to implement existing algorithms using intervals instead of real numbers, but obtaining narrow interval enclosures when using the algorithms is often difficult, and interval results can be totally useless if they are too wide. The reason why a specific interval algorithm yields intervals which are wide can either be because of the nature of the problem we are trying to solve, in which case nothing can be done, or it can be because the algorithm is not suited for intervals, in which case the interval enclosures used in the algorithm might be improved by modifying the algorithm.

One way to improve enclosures of a differentiable function evaluated with interval arguments is by using the mean value enclosure [38, 8, 47]. This method is described in Section 3. In the mean value enclosure, we use derivatives of a function to obtain narrow interval enclosures when evaluating it. We will use the mean value enclosure extensively in this report.

One of the most exciting properties of interval methods is their ability to provide information about existence of solutions to some implicit problems, e.g. to non-linear equations [1, 31, 39, 42, 53]. These self-validating methods are possible since we are capable of performing computations on sets of real numbers and to obtain interval bounds of the results. In Section 3 we describe two of the most common self-validating methods for obtaining solutions of non-linear equations. The two methods use a set theoretic fixed point theorem and are capable of proving existence of solutions, which are impossible to prove by known analytical methods.

By using automatic differentiation, described in Section 5, it is possible to obtain derivatives fast and without any symbolic manipulations [5, 16, 19, 20, 45, 51]. The derivatives obtained by automatic differentiation are just as accurate as evaluating the expressions for the true derivatives. By using interval arithmetic for evaluating the derivatives, we obtain correct enclosures. Two C++ program packages FADBAD/TADIFF for doing automatic differentiation have been developed [5, 6]. These packages are capable of differentiating functions implemented in C++ functions. Three methods have been implemented in FADBAD/TADIFF: The forward, the backward, and the Taylor expansion methods. Since FADBAD/TADIFF are capable of differentiating a C++ function which itself uses automatic differentiation, it is possible to combine the methods and this way generate derivatives in a very flexible way.

Automatic differentiation can also be used to obtain derivatives of a function given implicitly as a solution to an ordinary differential equation [6, 17, 18, 15]. By using these derivatives, it is possible to form a truncated Taylor series expansion and this way discretize the solution by an approximation of any order. Using interval analysis, it is also possible to enclose the remainder term of a truncated Taylor series expansions [34, 35, 33, 41, 50, 49, 55]. These remainder term enclosures will be used in Section 6 to develop a program package ADIODES for enclosing solutions of ordinary differential equations. This package is used in Section 7 to generate computer-assisted proofs.

A method for enclosing solutions of integral equations using the mean value form of an integral operator [7, 11, 43, 44, 48, 54] is described in Section 8, and two applications are presented.

2 Interval arithmetic

2.1 Intervals

The fundamentals of interval analysis described in this section are also described in the books of R.E. Moore [38, 39] and later in a book by G. Alefeld and J. Herzberger [2]. An interval $[x]$ is a nonempty set of real numbers[3]

$$[x] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}, \tag{2.1}$$

where $\underline{x} \leq \overline{x}$ are real numbers. Here $\underline{x}$ is called the infimum of $[x]$, and $\overline{x}$ is called the supremum. We also use the notation $[\underline{x}, \overline{x}]$ to denote the interval $[x]$. The set of all intervals is denoted by $\mathbb{IR}$.

$$\mathbb{IR} = \{[\underline{x}, \overline{x}] \mid \underline{x}, \overline{x} \in \mathbb{R}, \underline{x} \leq \overline{x}\} \tag{2.2}$$

The real number

$$\mathrm{m}([x]) = \left(\frac{\underline{x} + \overline{x}}{2}\right) \tag{2.3}$$

is the midpoint of $[x]$, and the non-negative number

$$\mathrm{w}([x]) = \overline{x} - \underline{x} \tag{2.4}$$

denotes the width of $[x]$. If the width of $[x]$ is zero, then the interval is called degenerate and consists of only one real number, i.e., $[x] \in \mathbb{R}$. The magnitude[4] of an interval $[x]$ is defined by

$$|[x]| = \max\{|\overline{x}|, |\underline{x}|\}. \tag{2.5}$$

The intersection of two intervals $[x], [y] \in \mathbb{IR}$ is

$$[x] \cap [y] = \begin{cases} \emptyset, & \text{if } \overline{x} < \underline{y} \text{ or } \overline{y} < \underline{x}, \\ [\max\{\underline{x}, \underline{y}\}, \min\{\overline{x}, \overline{y}\}], & \text{otherwise.} \end{cases} \tag{2.6}$$

Since the empty set is not an interval, we have that $\mathbb{IR}$ is not closed with respect to intersection, and special care has to be taken when $[x] \cap [y] = \emptyset$. We extend the ordering relation $<$ to intervals by

$$[x] < [y] \Leftrightarrow \overline{x} < \underline{y} \tag{2.7}$$

[3]We use the notation $[\cdot]$ in this report to denote intervals.

[4]Sometimes also called the absolute value.

and $>$ in a similar manner. Both relations are transitive. If $[x]$ and $[y]$ overlap, then $<$ and $>$ do not exclude each other. Therefore the ordering is only partial. Another transitive and partial ordering relation, inclusion $\subseteq$ in $\mathbb{IR}$, is defined by

$$[x] \subseteq [y] \quad \text{if and only if} \quad \underline{y} \leq \underline{x} \text{ and } \overline{y} \geq \overline{x}, \tag{2.8}$$

and the proper inclusion $\subset$ by

$$[x] \subset [y] \quad \text{if and only if} \quad \underline{y} < \underline{x} \text{ and } \overline{y} < \overline{x}. \tag{2.9}$$

Interval arithmetic operations are defined on $\mathbb{IR}$ so that the interval result of an operation encloses the corresponding real operations. Let $[x], [y] \in \mathbb{IR}$ be intervals. We define the usual binary operations with

$$[x] \star [y] = \{x \star y \mid x \in [x], y \in [y]\}, \tag{2.10}$$

for $\star \in \{+, -, \cdot, /\}$, and $[y] \not\ni 0$ if $\star$ is the division operator. As indicated above we use the same symbols for the interval operations as for the real operations. This is natural since interval arithmetic is a superset of real arithmetic. This can be seen from the fact that if $[x]$ and $[y]$ are degenerate intervals, and hence real numbers, then any of the defined interval operations produces a degenerate interval which by definition is the result of the corresponding real operation.

The elementary operations on $\mathbb{IR}$ given by Eq. (2.10) can be implemented with

$$\begin{aligned}
[x] + [y] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}], && \text{(2.11a)}\\
[x] - [y] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}], && \text{(2.11b)}\\
1/[x] &= [1/\overline{x}, 1/\underline{x}] \quad \text{if } 0 \notin [x], && \text{(2.11c)}\\
[x] \cdot [y] &= [\min(\underline{x}\,\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\,\overline{y}), && \text{(2.11d)}\\
&\quad\ \max(\underline{x}\,\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\,\overline{y})].
\end{aligned}$$

The formula for the product can also be split into nine cases

$$
\begin{aligned}
0 \le x,\ 0 \le y \quad &: \quad \underline{x \cdot y} = \underline{x}\ \underline{y},\ \overline{x \cdot y} = \overline{x}\ \overline{y}, && (2.12a)\\
\underline{x} < 0 < \overline{x},\ 0 \le y \quad &: \quad \underline{x \cdot y} = \underline{x}\overline{y},\ \overline{x \cdot y} = \overline{x}\ \overline{y}, && (2.12b)\\
x \le 0,\ 0 \le y \quad &: \quad \underline{x \cdot y} = \underline{x}\overline{y},\ \overline{x \cdot y} = \overline{x}\underline{y}, && (2.12c)\\
0 \le x,\ \underline{y} < 0 < \overline{y} \quad &: \quad \underline{x \cdot y} = \overline{x}\underline{y},\ \overline{x \cdot y} = \overline{x}\ \overline{y}, && (2.12d)\\
x \le 0,\ \underline{y} < 0 < \overline{y} \quad &: \quad \underline{x \cdot y} = \underline{x}\overline{y},\ \overline{x \cdot y} = \underline{x}\ \underline{y}, && (2.12e)\\
0 \le x,\ y \le 0 \quad &: \quad \underline{x \cdot y} = \overline{x}\underline{y},\ \overline{x \cdot y} = \underline{x}\overline{y}, && (2.12f)\\
\underline{x} < 0 < \overline{x},\ y \le 0 \quad &: \quad \underline{x \cdot y} = \overline{x}\underline{y},\ \overline{x \cdot y} = \underline{x}\ \underline{y}, && (2.12g)\\
x \le 0,\ y \le 0 \quad &: \quad \underline{x \cdot y} = \overline{x}\ \overline{y},\ \overline{x \cdot y} = \underline{x}\ \underline{y}, && (2.12h)\\
x \le 0,\ y \le 0 \quad &: \quad \underline{x \cdot y} = \min(\underline{x}\overline{y},\ \overline{x}\underline{y}), && \\
&\quad\ \ \overline{x \cdot y} = \max(\underline{x}\overline{y},\ \overline{x}\underline{y}). && (2.12i)
\end{aligned}
$$

The elementary functions such as $\cos, \sin, \exp$, etc. can also be defined to operate on elements in $\mathbb{IR}$ so that the interval result encloses the corresponding real operation.

For addition and multiplication we have the associative and commutative laws

$$
\begin{aligned}
[x] + ([y] + [z]) &= ([x] + [y]) + [z], && (2.13a)\\
[x]([y][z]) &= ([x][y])[z], && (2.13b)\\
[x] + [y] &= [y] + [x], && (2.13c)\\
[x][y] &= [y][x]. && (2.13d)
\end{aligned}
$$

The distributive law "$x(x+y) = xy + xz$" is not always valid for interval values. Instead we have the sub-distributive law [39]

$$[x]([y] + [z]) \subseteq [x][y] + [x][z]. \qquad (2.13e)$$

In some special cases, the distributive law is valid:

$$
\begin{aligned}
x([y] + [z]) &= x[y] + x[z] \text{ for } x \in \mathbb{R} \text{ and } [y], [z] \in \mathbb{IR}, && (2.14a)\\
[x]([y] + [z]) &= [x][y] + [x][z] \text{ if } [y][z] \ge 0. && (2.14b)
\end{aligned}
$$

We have the following properties regarding the absolute values and the widths of the result of arithmetic operations [2]:

$$\begin{aligned}
|[x]+[y]| &\leq |[x]|+|[y]|, && (2.15a)\\
|[x][y]| &= |[x]||[y]|, && (2.15b)\\
\mathrm{w}([x]\pm[y]) &= \mathrm{w}([x])+\mathrm{w}([y]), && (2.15c)\\
\mathrm{w}([x][y]) &\geq \max\{|[x]|\mathrm{w}([y]),|[y]|\mathrm{w}([x])\}, && (2.15d)\\
\mathrm{w}([x][y]) &\leq |[x]|\mathrm{w}([y])+|[y]|\mathrm{w}([x]), && (2.15e)\\
\mathrm{w}(1/[y]) &\leq |1/[y]|^2\mathrm{w}([y]). && (2.15f)
\end{aligned}$$

When implementing interval arithmetic on a computer, we control the rounding performed in every elementary interval operation so that the infimum of an interval computation is rounded downwards, and the supremum is rounded upwards. Computations using rounded interval arithmetic always encloses the result of the exact interval arithmetic calculations.

2.2 Interval vectors and matrices

We define interval vectors and interval matrices in the natural way, i.e., having intervals instead of real numbers as elements. The space of all n dimensional interval vectors is denoted by $\mathbb{IR}^n$, and the space of all $m\times n$ interval matrices is denoted $\mathbb{IR}^{m\times n}$. Let $D\subset\mathbb{R}^n$. We denote the set of all interval vectors in D by $\mathbb{I}D$

$$\mathbb{I}D=\{[x]\in\mathbb{IR}^n \mid [x]\subseteq D\}. \tag{2.16}$$

All arithmetic operations on interval matrices and vectors arise from interval operations in the same way real matrix and vector operations arise from real operations. The midpoint, width, magnitude, and intersection are defined on $\mathbb{IR}^{m\times n}$ by component-wise definitions. Let $[X]$ and $[Y]\in\mathbb{IR}^{m\times n}$ be interval matrices or vectors with interval components $[x_{ij}],[y_{ij}]$. Then we have

$$\mathrm{m}([X])=\left\{\mathrm{m}([x_{ij}])\right\} \tag{2.17}$$

$$\mathrm{w}([X])=\left\{\mathrm{w}([x_{ij}])\right\} \tag{2.18}$$

$$|[X]|=\left\{|[x_{ij}|]\right\}. \tag{2.19}$$

$$[X] \cap [Y] = \{[x_{ij}] \cap [y_{ij}]\}. \tag{2.20}$$

The ordering relations are defined component-wise,

$$[X] < [Y] \Leftrightarrow [x_{ij}] < [y_{ij}] \text{ for } i = 1\ldots m, j = 1\ldots n, \tag{2.21}$$

and $>$ in a similar manner. Inclusion $\subseteq$ is defined by

$$[X] \subseteq [Y] \Leftrightarrow [x_{ij}] \subseteq [y_{ij}] \text{ for } i = 1\ldots m, j = 1\ldots n, \tag{2.22}$$

and proper inclusion

$$[X] \subset [Y] \Leftrightarrow [x_{ij}] \subset [y_{ij}] \text{ for } i = 1\ldots m, j = 1\ldots n. \tag{2.23}$$

For interval matrix and vector additions, we have the associative and commutative laws

$$[X] + ([Y] + [Z]) = ([X] + [Y]) + [Z], \tag{2.24a}$$
$$[X] + [Y] = [Y] + [X], \tag{2.24b}$$

for $[X], [Y], [Z] \in \mathbb{IR}^{m\times n}$. Clearly we do not in general have the associative and commutative laws for interval matrix and vector multiplications. We however do still have the sub-distributive law

$$[X]([Y] + [Z]) \subseteq [X][Y] + [X][Z], \tag{2.24c}$$
$$([Y] + [Z])[X] \subseteq [Y][X] + [Z][X], \tag{2.24d}$$

for suitable dimensions of the interval matrices or vectors. If X is a real matrix, or vector, of the proper size, we have the distributive laws

$$X([Y] + [Z]) = X[Y] + X[Z], \tag{2.24e}$$
$$([Y] + [Z])X = [Y]X + [Z]X. \tag{2.24f}$$

Further details on the properties of interval matrix operations can be found in Alefeld and Herzberger [2, pp. 120-130].

2.3 Interval functions

An interval function is a function $F : \mathbb{IR}^m \to \mathbb{IR}^n$. F is said to be an interval extension of the real function f if

$$f(x) \in F([x]) \text{ for } x \in [x]. \tag{2.25}$$

Interval functions are usually designed to be interval extensions of some real function. If an interval function F has the property

$$[x_1] \subseteq [x_2] \Rightarrow F([x_1]) \subseteq F([x_2]), \tag{2.26}$$

then it is said to be inclusion monotonic.

If a real function $f : \mathbb{R}^m \to \mathbb{R}^n$ is given by an expression, then an interval extension $F : \mathbb{IR}^m \to \mathbb{IR}^n$ of f can be obtained by replacing all real arguments by interval arguments, and all real operations by the corresponding interval operations. It should be evident that by this transformation we obtain an interval extension F of f. Interval extensions obtained in this way are called natural interval extensions. Simple interval extensions are in theory inclusion monotonic, but in practice, when performing rounded interval arithmetic on a computer, it will depend on the actual rounding performed. We do not use inclusion monotonicity in this report because of this machine dependency.

Using interval arithmetic we can calculate enclosures of the range $\mathrm{R}(f,[x])$ of f over interval vectors $[x] \in \mathbb{IR}^n$

$$\mathrm{R}(f,[x]) = \{f(x) \mid x \in [x]\}. \tag{2.27}$$

However, we will generally not obtain the exact range $\mathrm{R}(f,[x])$ when evaluating the interval extension $F([x])$. For example, the function

$$f(x) = x(1-x) \tag{2.28}$$

has the range $[0, 0.25]$ over $[x] = [0, 1]$, while the interval extension

$$F([x]) = [x](1-[x]) \tag{2.29}$$

yields the interval $[0, 1]$. This overestimation occurs because the two occurrences of x in the expression are regarded as independent in the interval extension. Since every occurrence of a variable in an expression is considered as independent when forming the interval extension, it is important to minimize these occurrences. The occurrence of $[x]$ in the above example cannot be minimized. However, if for $g(x) = x/(1-x)$, we should use the interval extension $G([x]) = 1/(1/[x]-1)$ instead of the obvious $G([x]) = [x]/(1-[x])$.

Overestimation can also cause an interval extension of a real function to be undefined for some interval arguments [2, p.22]. The real function

$$h(x) = \frac{1}{x^2 + \frac{1}{2}} \tag{2.30}$$

is defined for all $x \in \mathbb{R}$ and has an interval extension

$$H([x]) = \frac{1}{[x][x] + \frac{1}{2}}. \tag{2.31}$$

However, evaluating $H([-1,1])$, we obtain

$$H([-1,1]) = \frac{1}{[-1,1][-1,1] + \frac{1}{2}} = \frac{1}{[-0.5, 1.5]}, \tag{2.32}$$

so the interval extension H is not defined for all $[x] \in \mathbb{IR}$. In the following we will not regard this as a problem since we automatically will discover if an interval function is undefined when evaluating it.

Integrals of interval functions can be defined [10, 38]. We will only consider integration of interval functions of type $F : [x] \to \mathbb{IR}$, where $[x] \in \mathbb{IR}$, which have integrable endpoint functions $\underline{F}, \overline{F}$, so that $F(t) = [\underline{F}(t), \overline{F}(t)]$, for $t \in [x]$. We define the integral of F in the interval $[a,b] \subseteq [x]$ by

$$\int_{[a,b]} F(t)dt = \left[\int_{[a,b]} \underline{F}(t)dt, \int_{[a,b]} \overline{F}(t)dt \right], \tag{2.33}$$

which has the following properties [10, 38]:

$$f(t) \in F(t), \text{ for } t \in [a,b] \Rightarrow \int_{[a,b]} f(t)dt \in \int_{[a,b]} F(t)dt, \tag{2.34a}$$

$$\int_{[a,b]} F(t)dt = \int_{[a,x]} F(t)dt + \int_{[x,b]} F(t)dt, \text{ for } x \in [a,b]. \tag{2.34b}$$

Furthermore, if F is either non-negative or non-positive in the interval $[a,b]$, then for $[c] \in \mathbb{IR}$ we have [54]

$$\int_a^b [c]F(t)dt = [c]\int_a^b F(t)dt. \tag{2.34c}$$

Proof
Assume that F is non-negative, i.e., that

$$0 \le \underline{F}(t) \le \overline{F}(t) \text{ for } t \in [a,b]. \tag{2.35}$$

Now we have

$$0 \le \int_a^b F(t)dt = [\int_a^b \underline{F}(t)dt, \int_a^b \overline{F}(t)dt]. \tag{2.36}$$

By splitting into three cases and using the relations Eqs. (2.12a-2.12i), we obtain

a) If $0 \le \underline{c} \le \overline{c}$, then using Eq. (2.12a) we obtain the following

$$\begin{aligned}\int_a^b [c]F(t)dt &= \int_a^b [\underline{c}\underline{F}(t), \overline{c}\overline{F}(t)]dt = [\underline{c}\int_a^b \underline{F}(t)dt, \overline{c}\int_a^b \overline{F}(t)dt] \\ &= [\underline{c}, \overline{c}] \int_a^b [\underline{F}(t), \overline{F}(t)]dt = [c]\int_a^b F(t)dt.\end{aligned}$$

b) If $\underline{c} < 0 < \overline{c}$, then using Eq. (2.12b) we have

$$\begin{aligned}\int_a^b [c]F(t)dt &= \int_a^b [\underline{c}\overline{F}(t), \overline{c}\overline{F}(t)]dt = [\underline{c}\int_a^b \overline{F}(t)dt, \overline{c}\int_a^b \overline{F}(t)dt] \\ &= [\underline{c}, \overline{c}] \int_a^b \overline{F}(t)dt = [c]\int_a^b F(t)dt.\end{aligned}$$

c) If $\underline{c} \le \overline{c} \le 0$, then using (2.12c) we have

$$\begin{aligned}\int_a^b [c]F(t)dt &= \int_a^b [\underline{c}\overline{F}(t), \overline{c}\underline{F}(t)]dt = [\underline{c}\int_a^b \overline{F}(t)dt, \overline{c}\int_a^b \underline{F}(t)dt] \\ &= [\underline{c}, \overline{c}] \int_a^b [\underline{F}(t), \overline{F}(t)]dt = [c]\int_a^b F(t)dt.\end{aligned}$$

Since all three cases leads to the same result we have proven Eq. (2.34c) for non-negative $F(t)$. The same can be done for an F which is non-positive.

□

It is important to note that Eq. (2.34c) cannot be used if F changes sign over the interval we integrate. As an example, let $[c] = [-1, 1]$ and $F(t) = [t, t]$:

$$\int_{-1}^{1} [-1, 1][t, t] dt = \int_{-1}^{1} [-|t|, |t|] dt = [-1, 1],$$

but

$$[-1, 1] \int_{-1}^{1} [t, t] dt = [-1, 1]0 = 0.$$

3 Taylor's Theorem and the mean value enclosure

Taylor's Theorem plays a very important role in this report.

Theorem 1 (Taylor's Theorem) *Suppose that $p(x)$ is differentiable n times in a convex set D containing the point $x_0 \in D$, and that $p^{(n)}(x)$ is integrable from x_0 to any point $x_1 \in D$. Now we have*

$$p(x_1) = p(x_0) + \sum_{k=1}^{n-1} \frac{1}{k!} p^{(k)}(x_0)(x_1 - x_0)^k + R_n(x_0, x_1), \tag{3.1a}$$

where

$$R_n(x_0, x_1) = (x_1 - x_0)^n \int_0^1 p^{(n)}(\theta x_1 + (1-\theta)x_0) \frac{(1-\theta)^{n-1}}{(n-1)!} d\theta. \tag{3.1b}$$

For the proof, see [44].

Using $n = 1$ we obtain the well known formula

$$p(x_1) = p(x_0) + (x_1 - x_0) \int_0^1 p'(\theta x_1 + (1-\theta)x_0) d\theta. \tag{3.2}$$

This relation is used in the following theorem.

Theorem 2 *Let $f \in C^1(D, \mathbb{R}^n)$, where $D \subseteq \mathbb{R}^m$ is an open set. For $[x] \in \mathbb{I}D$ we have*

$$f(x) - f(y) \in F'([x])(x - y) \text{ for } x, y \in [x], \tag{3.3}$$

Let F' be an interval extension of f'.

Proof

Using that $[x]$ is a convex set, Eq. (3.2) and Eq. (2.34c) yields

$$\begin{aligned} f(x) - f(y) &= (x - y) \int_0^1 f'(\theta x + (1-\theta)y) d\theta \\ &\in (x - y) \int_0^1 F'([x]) d\theta \\ &= (x - y) F'([x]) \int_0^1 1 d\theta \\ &= (x - y) F'([x]) \end{aligned}$$

□

A way to reduce overestimation in the evaluation of a differentiable function $f \in C^1(\mathbb{R}^m, \mathbb{R}^n)$ in a set of real numbers $x \in [x]$ is by using the mean value enclosure: Since $f(x) \in f(y) + f'([x])([x] - y)$, where $x, y \in [x]$, we have $f(x) \in F_m([x], y)$,

$$F_m([x], y) = f(y) + F'([x])([x] - y), \text{ for } y \in [x]. \tag{3.4}$$

The interval function F_m is called the mean value enclosure of f. For small widths of $[x]$, this interval function often provides tighter enclosures than the natural interval extension of f. Normally y is chosen to be the midpoint of $[x]$.

It has been shown that F_m is inclusion monotonic if the interval Jacobian matrix F' is inclusion monotonic [8, 47]. However this property is not always inherited when using rounded interval arithmetic. As an example, consider the function $f(x) = x^2$, with the mean value interval extension $F_m([x]) = \mathrm{m}([x])^2 + 2[x]([x] - \mathrm{m}([x]))$. If we evaluate F_m using rounded interval arithmetic where only integers are allowed as interval bounds, we can obtain different results when evaluating in $[x] = [0, 1]$,

$$F_m([0,1]) = \mathrm{m}([0,1])^2 + 2[0,1]([0,1] - \mathrm{m}([0,1])).$$

Depending on which way we round $\mathrm{m}([0,1])$, we have the two cases

$$\begin{aligned} \tilde{F}_m([0,1]) &= 0^2 + 2[0,1]([0,1] - 0) = [0,2], \\ \tilde{F}_m([0,1]) &= 1^2 + 2[0,1]([0,1] - 1) = [-1,1]. \end{aligned}$$

Both cases leads to an enclosure of the true result $[0, 1]$, but the example shows that $\tilde{F}_m$ does not have the inclusion monotonicity property.

3.1 The interval Newton and Krawczyk methods

Among the most important tools in interval analysis are fixed point theorems. One of these is the Brouwer fixed point Theorem [21].

Theorem 3 (Brouwer's fixed point Theorem) *Every continuous mapping of a closed bounded convex set in $\mathbb{R}^n$ into itself has a fixed point.*

Assume that $f : \mathbb{R}^n \to \mathbb{R}^n$ is a continuous function, and F is an interval extension of f. Since an interval vector $[x] \in \mathbb{IR}^n$ is a closed and bounded convex set in $\mathbb{R}^n$, we have that if $\mathrm{R}(f, [x]) \subseteq [x]$ then it follows from the fixed point theorem that

f has a fixed point in $[x]$. Since $\mathrm{R}(f,[x]) \subseteq F([x])$, it follows that the condition $F([x]) \subseteq [x]$, which can be checked automatically in a computer program, also implies existence of a fixed point of f in $[x]$. Algorithms which use fixed point theorems in this way to prove existence are called “self-validating algorithms”. In the example, Eq. (2.28), where $f(x) = x(1-x)$ we found that $F([0,1]) = [0,1]$. Hence, f has a fixed point in $[0,1]$. In the following we will see some very important applications of fixed point methods.

Consider the problem of solving the non-linear equation

$$f(x) = 0, \tag{3.5}$$

where $f : \mathbb{R}^n \to \mathbb{R}^n$ is a continuous function. A well known method for solving this equation is finding fixed points of the map $g(x,Y) = x - Yf(x)$, where $Y \in \mathbb{R}^{n\times n}$ is a non-singular matrix. We have the relation

$$f(x) = 0 \Leftrightarrow g(x,Y) = x. \tag{3.6}$$

Assume that f is differentiable. Using $Y = (f'(x))^{-1}$ in the fixed point operator g yields the method of Newton

$$n(x) = x - (f'(x))^{-1} f(x), \tag{3.7}$$

if $f'(x)$ is a non-singular matrix. Because of the property Eq. (2.15c) the simple interval extension of Newton's method $[x] - (F'([x]))^{-1}F([x])$ is useless since its width generally is larger than $[x]$, unless $F([x]) = 0$. Instead, we define the interval Newton operator by

$$N([x],x) = x - (F'([x]))^{-1} f(x) \text{ for } x \in [x], \tag{3.8}$$

where F' is a interval extension of f', and $F'([x])$ should not contain any singular matrices. The interval Newton operator has the following properties [1]:

1) If $x^\star \in [x]$ is a solution of Eq. (3.5), then $x^\star \in N([x],x)$.

2) If $[x] \cap N([x],x) = \emptyset$, then no solution in $[x]$ exists.

3) If $N([x],x) \subseteq [x]$ then a unique solution exists in $[x]$.

It is normal to choose $x = \mathrm{m}([x])$ when implementing the method. Furthermore the interval linear system $F'([x])[r] = f(x)$ can be solved using an interval Gaussian algorithm instead of explicitly computing the inverse of $F'([x])$, (see [1]).

If we have a case where $N([x],x) \cap [x] \neq \emptyset$, but neither 2) og 3) apply it is usual to continue the search for solutions of Eq. (3.5) in $[x'] = N([x],x) \cap [x]$ or, if $[x'] = [x]$, the two halves in a bisection of $[x']$.

Another fixed point operator is the Krawczyk operator, which occurs when forming the mean value enclosure of $g(x,Y)$, (see [31]):

$$K([x],x,Y) = x - Yf(x) + (I - YF'([x])([x]-x) \text{ for } x \in [x]. \tag{3.9}$$

Since $g(x,Y) = x \Leftrightarrow f(x) = 0$ and $K([x],x,Y)$ is an interval extension (mean value enclosure) of $g(x,Y)$, we have by Brouwer's fixed point theorem that the condition $K([x],x,Y) \subseteq [x]$ implies existence of a solution $x^\star \in [x]$ to Eq. (3.5). We have the same properties for the Krawczyk operator as for the Newton operator, except for uniqueness, but if one of the following criteria is fulfilled along with the condition $K([x],x,Y) \subseteq [x]$, then we have uniqueness:

1) $F'([x])$ does not contain any singular matrices, (see [1]).

2) We have the proper inclusion $K([x],x,Y) \subset [x]$, (see [42]).

It is normal again to choose $x = \mathrm{m}([x])$ and the real matrix $Y = (f'(x))^{-1}$ when implementing the Krawczyk operator. The advantage of using the Krawczyk operator instead of the Newton operator is that we do not have to solve a linear interval system, but instead we invert $(f'(x))^{-1}$ using normal machine arithmetic.

Since the mentioned fixed point methods are capable of proving existence and uniqueness or non-existence of solutions to Eq. (3.5) in some interval vector it is possible to create algorithms that find all solutions of Eq. (3.5) in a given interval vector. These algorithms work by subdividing the interval vectors until existence/uniqueness or non-existence of solutions can be proved [9, 40]. Using these methods it is possible to find all solutions of non-linear equations of very high complexity [53, 26].

4 Enclosure methods for discrete mappings

An n dimensional discrete map is a function φ which maps the n dimensional real space onto itself, $\varphi : \mathbb{R}^n \to \mathbb{R}^n$. These classes of functions are often used to describe the development of the state of some system, e.g. when discretizing ordinary differential equations, which we will see later. Given an initial value $y_0 \in \mathbb{R}^n$, we can generate a sequence of points $\{y_i\}_{i=0,\ldots}$ defined by

$$y_{i+1} = \varphi(y_i), \text{ for } i = 0, \ldots. \tag{4.1}$$

The problem of enclosing these points by using interval arithmetic is very well known, and in many cases it is difficult to obtain narrow interval vector enclosures. The difficulties arise since the set $\mathrm{R}(\varphi, [y_i])$ in general is not an interval vector, but since the evaluation of the simple interval extension of φ is an interval vector which encloses[5] $\mathrm{R}(\varphi, [y_i])$, it inevitably also includes points which are not in the set $\mathrm{R}(\varphi, [y_i])$. If we use the simple interval iteration

$$[y_{i+1}] = \Phi([y_i]), \text{ for } i = 0, \ldots, \tag{4.2}$$

where Φ is an interval extension of φ, the problem gets even worse since false values introduce other false values throughout the iterative process. This effect has been named “The wrapping effect” and is mainly discussed in connection with solving ordinary differential equations [37, 38, 41].

Moore demonstrates the wrapping effect by using the following map [38]

$$\varphi_{rot}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} \tag{4.3}$$

This map rotates points in the plane clockwise in an angle of Δt about the origin. If we apply the map, using $\Delta t = \frac{\pi}{5}$, on an interval vector $([x_0], [y_0]) = ([0.95, 0.05], [-0.05, 0.05])$, we get a set $\mathrm{R}(\varphi_{rot}, ([x_0], [y_0]))$, which is a rotated rectangle. Enclosing $\mathrm{R}(\varphi_{rot}, ([x_0], [y_0]))$ in the interval vector $([x_1], [y_1])$ introduces some overestimation. In the next step, $([x_2], [y_2])$ is an enclosure of $\mathrm{R}(\varphi_{rot}, ([x_1], [y_1]))$, and so forth; see Figure 4.1.

From Figure 4.1 we see that the simple interval iteration Eq. (4.2) performs very bad on Eq. (4.3); the size of $([x_5], [y_5])$ is many times bigger than $([x_0], [y_0])$, even though $\mathrm{R}(\varphi_{rot}^5, ([x_0], [y_0]))$ in theory has the same size as $([x_0], [y_0])$.

[5] This is called the wrapping of $\mathrm{R}(\varphi, [y_i])$.

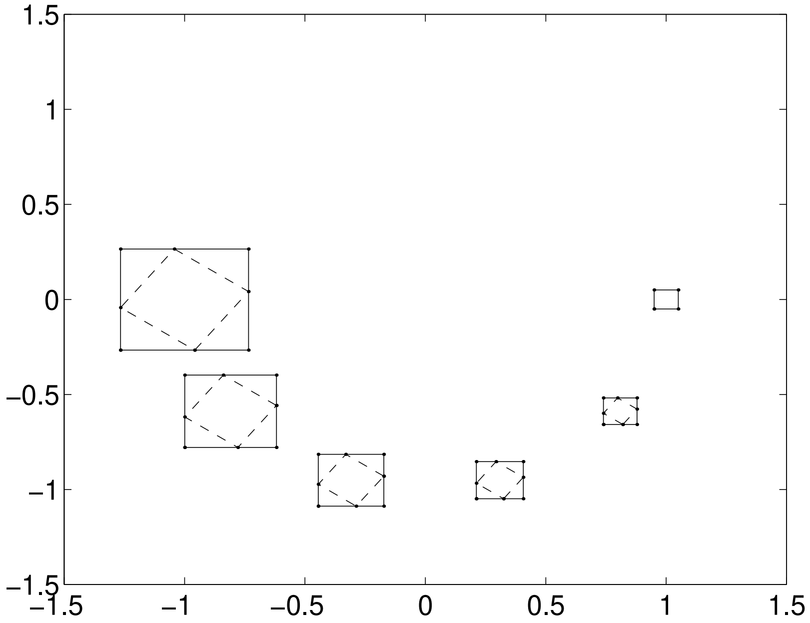


Figure 4.1: Five simple interval iterations, using the map in Eq. (4.3) with $\Delta t = \frac{\pi}{5}$ and the initial interval vector $([x_0],[y_0]) = ([0.95, 1.05], [-0.05, 0.05])$. The solid rectangles are the interval vector enclosures $\{([x_j],[y_j])\}_{j=0,\dots,5}$ while the dashed rectangles are their images $R(\varphi_{rot}, ([x_j],[y_j]))$, for $j = 0,\dots,4$.

4.1 Lohner's method used on discrete maps

Lohner's method is probably the best known method for solving ordinary differential equations with automatic verification of existence of the solution in interval bounds. The method is described in various places [34, 35, 41, 50, 49, 55] and Lohner has developed a program, called "AWA"[6], for solving initial value problems, using this method. We will here see how to use the method on discrete mappings, and then later use it for discretizing ordinary differential equations.

Consider discrete maps of the type

$$\varphi(y) = \tilde{\varphi}(y) + \varepsilon(y), \tag{4.4}$$

where $\tilde{\varphi}$ is a known and differentiable function which approximates φ, and ε is an error which can be bounded by some interval function $\Sigma([y])$, i.e., we have $\varepsilon(y) \in$

[6] AWA is an abbreviation for "anfangswertaufgaben" which is german and means initial value problem.

$\Sigma([y])$, for $y \in [y]$. Given an initial point $y_0 \in \mathbb{R}^n$, we generate the sequence $\{y_j\}_{j=0,\dots}$ by

$$y_{j+1} = \varphi(y_j) = \tilde{\varphi}(y_j) + \varepsilon(y_j), \quad j \geq 0. \tag{4.5}$$

Let $\{[z_j]\}_{j=0,\dots}$ be a sequence of interval vectors in $\mathbb{IR}^n$ which encloses the error term, i.e., $[z_{j+1}] = \Sigma(y_j)$ so that $\varepsilon(y_j) \in [z_{j+1}]$, for $j \geq 0$. Note that $[z_0]$ is chosen arbitrarily. Furthermore, let $\{s_j\}_{j=0,\dots}$ be a sequence of vectors so $s_j \in [z_j]$, for $j \geq 0$. Define the vector sequence $\{\hat{y}_j\}_{j=0,\dots}$ by

$$\hat{y}_0 = y_0 + s_0 \quad \text{and} \quad \hat{y}_{j+1} = \tilde{\varphi}(\hat{y}_j) + s_{j+1}, \text{ for } j \geq 0, \tag{4.6}$$

and an enclosure sequence $\{[\hat{y}_j]\}_{j=0,\dots}$ by

$$[\hat{y}_0] = y_0 + [z_0] \quad \text{and} \quad [\hat{y}_{j+1}] = \tilde{\varphi}(\hat{y}_j) + [z_{j+1}], \text{ for } j \geq 0, \tag{4.7}$$

We will now have $\hat{y}_j \in [\hat{y}_j]$, for $j \geq 0$ and

$$0 \in [z_j] - s_j = [\hat{y}_j] - \hat{y}_j, \text{ for } j \geq 0. \tag{4.8}$$

Using Eq. (3.2) to expand $\tilde{\varphi}(y_j)$ in Eq. (4.5) we obtain

$$y_{j+1} = \tilde{\varphi}(\hat{y}_j) + \left\{ \int_0^1 \frac{\partial \tilde{\varphi}}{\partial y}(\theta y_j + (1-\theta)\hat{y}_j) d\theta \right\} (y_j - \hat{y}_j) + \varepsilon(y_j). \tag{4.9}$$

Defining $r_j = y_j - \hat{y}_j$ and

$$R(y_j, \hat{y}_j) = \int_0^1 \frac{\partial \tilde{\varphi}}{\partial y}(\theta y_j + (1-\theta)\hat{y}_j) d\theta, \tag{4.10}$$

we can rewrite Eq. (4.9) as

$$\begin{aligned} y_{j+1} &= \tilde{\varphi}(\hat{y}_j) + s_{j+1} + R(y_j, \hat{y}_j) r_j + \varepsilon(y_j) - s_{j+1} \\ &= \hat{y}_{j+1} + r_{j+1}, \end{aligned} \tag{4.11}$$

where

$$r_{j+1} = R(y_j, \hat{y}_j) r_j + \varepsilon(y_j) - s_{j+1}. \tag{4.12}$$

Since $\hat{y}_{j+1}$ in Eq. (4.11) is a real vector which can be found in practice, the problem of finding a narrow enclosure of r_{j+1} remains. In the following we will use an interval extension of $\frac{\partial \tilde{\varphi}}{\partial y}$, denoted by $\tilde{\Phi}'$.

4.1.1 The mean value enclosure

Assume that a) $[r_j]$ is an enclosure of r_j, b) $0 \in [r_j]$, and c) $[y_j]$ is an enclosure of both y_j and $\hat{y}_j$. Define $[S_j] = \tilde{\Phi}'([y_j])$ so that $R(y_j, \hat{y}_j) \in [S_j]$. We can now enclose r_{j+1} in Eq. (4.12) by

$$r_{j+1} \in [S_j][r_j] + [z_{j+1}] - s_{j+1}. \tag{4.13}$$

Using the relation in Eq. (4.8) we have $r_{j+1} \in [r_{j+1}]$, where

$$[r_{j+1}] = [S_j][r_j] + [\hat{y}_{j+1}] - \hat{y}_{j+1}. \tag{4.14}$$

From Eq. (4.11) we have $y_{j+1} \in \hat{y}_{j+1} + [r_{j+1}]$. Hence we define

$$[y_{j+1}] = \hat{y}_{j+1} + [r_{j+1}]. \tag{4.15}$$

We have now that

a) $r_{j+1} \in [r_{j+1}]$ by definition.

b) Since $\hat{y}_{j+1} \in [\hat{y}_{j+1}]$ and $0 \in [r_j]$, we have from Eq. (4.14) that $0 \in [r_{j+1}]$.

c) $y_{j+1} \in [y_{j+1}]$ by definition. Since $0 \in [r_{j+1}]$ we have from Eq. (4.15) that $\hat{y}_{j+1} \in [y_{j+1}]$.

Since our assumptions a)-c) are invariant throughout one iteration, they will be true throughout all iterations as long as they are true from the beginning.

This leads us to Algorithm 4.1.

If also the error term $\varepsilon(y_j)$ in Eq. (4.5) is known to be a differentiable function with the interval extension $\Sigma'([y])$ of the Jacobian matrix, then we are also capable of calculating an enclosure of the matrix

$$D_j = \frac{\partial \varphi^j}{\partial y}(y_0). \tag{4.16}$$

Using the chain rule on Eq. (4.5), we get

$$D_{j+1} = \left(\frac{\partial \tilde{\varphi}}{\partial y}(y_j) + \frac{\partial \varepsilon}{\partial y}(y_j) \right) D_j. \tag{4.17}$$

Hence we have $D_{j+1} \in [D_{j+1}]$, where

$$[D_0] \quad = \quad I \quad \text{and} \tag{4.18a}$$
$$[D_{j+1}] \quad = \quad \big([S_j] + \Sigma'([y_j])\big)\,[D_j], \text{ for } j \geq 0. \tag{4.18b}$$

It is easy to modify Algorithm 4.1 to also include calculation of $[D_{j+1}]$.

Initialize:
$[y_0], \hat{y}_0 = \mathrm{m}([y_0]),\ [r_0] = [y_0] - \hat{y}_0.$

Input:
$[y_j], \hat{y}_j,\ [r_j].$

Iteration:
$[z_{j+1}] = \Sigma([y_j]),$
$[\hat{y}_{j+1}] = \tilde{\varphi}(\hat{y}_j) + [z_{j+1}],$
$[S_j] = \tilde{\Phi}'([y_j]),$
$\hat{y}_{j+1} = \mathrm{m}([\hat{y}_{j+1}]),$
$[y_{j+1}] = [S_j][r_j] + [y_{j+1}],$
$[r_{j+1}] = [y_{j+1}] - \hat{y}_{j+1}.$

Output:
$[y_{j+1}], \hat{y}_{j+1},\ [r_{j+1}].$

Algorithm 4.1: The mean value enclosure of the discrete map Eq. (4.1).

4.1.2 The extended mean value enclosure

Let $\{A_j\}_{j=0,\ldots}$ be a sequence of regular real matrices[7] and define $\hat{r}_j = A_j^{-1} r_j$. Assume that a) $[\hat{r}_j]$ is an enclosure of $\hat{r}_j$, b) $0 \in [\hat{r}_j]$, and c) $[y_j]$ is an enclosure of both y_j and $\hat{y}_j$. Define $[S_j] = \tilde{\Phi}'([y_j])$, so that $R(y_j, \hat{y}_j) \in [S_j]$. We have from Eq. (4.12) that

$$\begin{aligned} r_{j+1} &= A_{j+1} A_{j+1}^{-1} R(y_j, \hat{y}_j) A_j A_j^{-1} r_j + A_{j+1} A_{j+1}^{-1} (\varepsilon(y_j) - s_{j+1}) \\ &= A_{j+1} \left\{ A_{j+1}^{-1} R(y_j, \hat{y}_j) A_j \hat{r}_j + A_{j+1}^{-1} (\varepsilon(y_j) - s_{j+1}) \right\}. \end{aligned} \tag{4.19}$$

And using $\hat{r}_{j+1} = A_{j+1}^{-1} r_{j+1}$, we find

$$\hat{r}_{j+1} = A_{j+1}^{-1} R(y_j, \hat{y}_j) A_j \hat{r}_j + A_{j+1}^{-1} (\varepsilon(y_j) - s_{j+1}), \tag{4.20}$$

which can be enclosed by

$$[\hat{r}_{j+1}] = (A_{j+1}^{-1}([S_j] A_j))[\hat{r}_j] + A_{j+1}^{-1}([\hat{y}_{j+1}] - \hat{y}_{j+1}). \tag{4.21}$$

[7] We will later discuss how to choose $\{A_j\}_{j=0,\ldots}$.

An enclosure of y_{j+1} in Eq. (4.11) can be found by [50]

$$[y_{j+1}] = \hat{y}_{j+1} + ([S_j]A_j)[\hat{r}_j] + [\hat{y}_{j+1}] - \hat{y}_{j+1}. \tag{4.22}$$

We now have

a) $\hat{r}_{j+1} \in [\hat{r}_{j+1}]$ by definition.

b) Since $\hat{y}_{j+1} \in [\hat{y}_{j+1}]$ and $0 \in [\hat{r}_j]$, we have from Eq. (4.21) that $0 \in [\hat{r}_{j+1}]$.

c) $y_{j+1} \in [y_{j+1}]$ by definition. Since $0 \in [\hat{r}_{j+1}]$ we have from Eq. (4.22) that $\hat{y}_{j+1} \in [y_{j+1}]$.

Also here our assumptions a)-c) are invariant throughout one iteration, and they will be true throughout all iterations as long as they are true from the beginning.

This leads us to Algorithm 4.2.

Initialize:
$[y_0]$, $\hat{y}_0 = \mathrm{m}([y_0])$, $[\hat{r}_0] = [y_0] - \hat{y}_0$, $A_0 = I$.

Input:
$[y_j]$, $\hat{y}_j$, $[\hat{r}_j]$, A_j.

Iteration:
$[z_{j+1}] = \Sigma([y_j])$,
$[\hat{y}_{j+1}] = \tilde{\varphi}(\hat{y}_j) + [z_{j+1}]$,
$[S_j] = \tilde{\Phi}'([y_j])$,
$\hat{y}_{j+1} = \mathrm{m}([\hat{y}_{j+1}])$,
Choose a regular real matrix A_{j+1},
$[y_{j+1}] = ([S_j]A_j)[\hat{r}_j] + [\hat{y}_{j+1}]$,
$[\hat{r}_{j+1}] = (A_{j+1}^{-1}([S_j]A_j))[\hat{r}_j] + A_{j+1}^{-1}([\hat{y}_{j+1}] - \hat{y}_{j+1})$.

Output:
$[y_{j+1}]$, $\hat{y}_{j+1}$, $[\hat{r}_{j+1}]$.

Algorithm 4.2: The extended mean value enclosure of the discrete map Eq. (4.1).

As in the previous method, we are able to calculate an enclosure of the matrix D_j given in Eq. (4.16) if the error term $\varepsilon(y_j)$ in Eq. (4.5) is known to be a differentiable function with the interval extension $\Sigma'([y])$ of its Jacobian matrix. Let $\hat{D}_j = A_j^{-1} D_j$, we have from Eq. (4.5) that

$$\hat{D}_{j+1} = A_{j+1}^{-1} \left(\frac{\partial \tilde{\varphi}}{\partial y}(y_j) + \frac{\partial \varepsilon}{\partial y}(y_j) \right) A_j \hat{D}_j, \tag{4.23}$$

Hence we have $\hat{D}_{j+1} \in [\hat{D}_{j+1}]$, where

$$[\hat{D}_0] = A_0^{-1} \quad \text{and} \tag{4.24a}$$

$$[\hat{D}_{j+1}] = \left(A_{j+1}^{-1}(([S_j] + \Sigma'([y_j]))A_j) \right) [\hat{D}_j], \text{ for } j \geq 0. \tag{4.24b}$$

We now have an enclosure of D_{j+1} by

$$[D_{j+1}] = A_{j+1}[\hat{D}_{j+1}]. \tag{4.25}$$

This enclosure can easily be included in Algorithm 4.2.

If we choose $A_j = I$ for $j \geq 0$ in Algorithm 4.2, then we will obtain the normal mean value method described in Algorithm 4.1, which encloses the uncertainty of the solution set in an interval vector

$$r_{j+1} \in [r_{j+1}]. \tag{4.26}$$

The reason for introducing the sequence $\{A_j\}_{j=0,\ldots}$ in the latter method is to be able to represent the uncertainty of the solution set as

$$r_{j+1} \in \{A_{j+1}\hat{r} \mid \hat{r} \in [\hat{r}_{j+1}]\}, \tag{4.27}$$

which is a more flexible set than the interval vector enclosure. The matrix A_{j+1} should be chosen in such a way that the set

$$\hat{r}_{j+1} \in \{(A_{j+1}^{-1}(SA_j))\hat{r} + A_{j+1}^{-1}(y - \hat{y}_{j+1}) \mid S \in [S_j], \hat{r} \in [\hat{r}_j], y \in [\hat{y}_{j+1}]\} \tag{4.28}$$

"looks" like an interval vector so that the interval vector enclosure $[\hat{r}_{j+1}]$ does not introduce too much overestimation. Since the error term in general is small, i.e., $[\hat{y}_{j+1}] - \hat{y}_{j+1}$ is a narrow interval vector, we will only consider the problem of choosing A_{j+1} in a way that the set $\{(A_{j+1}^{-1}(SA_j))\hat{r} \mid S \in [S_j], \hat{r} \in [\hat{r}_j]\}$ "looks" like an interval vector.

An obvious choice is $A_{j+1} = \mathrm{m}([S_j]A_j)$ so that $A_{j+1}^{-1}([S_j]A_j)$ encloses the identity matrix. Lohner [34] calls the enclosure using this choice of A_{j+1}, the "parallelepiped enclosure". He argues that this enclosure method is only practical if the matrices A_{j+1} are very well conditioned. This leaves out stiff systems. The matrix A_{j+1} should also be regular, which is not assured by this choice of A_{j+1}.

In general, the best known method is the "QR-factorization method" which was proposed by Lohner [34]. In this method, the matrix A_{j+1} is the Q matrix obtained in a QR-factorization of the matrix $\hat{A}_{j+1} = \tilde{A}_{j+1}P_{j+1}$, where $\tilde{A}_{j+1} \in [S_j]A_j$ and P_{j+1} is a permutation matrix. The Q matrix in a QR-factorization is orthogonal, and the matrix R is upper triangular. Since the matrix A_{j+1} is orthogonal, the enclosure obtained by using this matrix is a rectangular enclosure, which can rotate freely.

By the relation $A_{j+1}R = \hat{A}_{j+1}$, where R is an upper triangular matrix and $\|A_{j+1}\|_2 = 1$, we see that the first column in A_{j+1} is a normalization of the first column of $\hat{A}_{j+1}$ and that the ith column of A_{j+1}, for $i = 2, \ldots, n$ is a normalization of the projection of the ith column of $\hat{A}_{j+1}$ to the orthogonal complement of the previous $i-1$ columns of A_{j+1} .

By choosing a permutation matrix P_{j+1} so that the first column in $\tilde{A}_{j+1}P_{j+1}$ contains the vector in which the parallelepiped $\{\tilde{A}_{j+1}\hat{r} \mid \hat{r} \in [\hat{r}_j]\}$ has the largest span and the second column of $\tilde{A}_{j+1}P_{j+1}$ contains the vector in which the set has its second largest span, and so forth, we insure that the directions in which the parallelepiped has the largest span are enclosed best by the rectangular enclosure.

Consider the vector l, where the ith component l_i is the norm of the ith column of $\tilde{A}_{j+1}$ multiplied with the width of the ith component of $[\hat{r}_j]$:

$$l_i = \|\tilde{A}_{j+1}(:,i)\|_2 \mathrm{w}([\hat{r}_j(i)]). \tag{4.29}$$

Now l_i is the length of the edge in the parallelepiped induced by the ith component of $[\hat{r}_j]$. By choosing P_{j+1} so that the vector $l^T P_{j+1}$ contains the elements of l, sorted in decreasing order, we insure that the vectors in $\hat{A}_{j+1}$ are sorted by importance.

Consider the following example: Let

$$\tilde{A}_{j+1} = \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix} \quad \text{and} \quad [\hat{r}_j] = \begin{pmatrix} [-1,1] \\ [-3,2] \end{pmatrix}. \tag{4.30}$$

The parallelepiped $\{\tilde{A}_{j+1}\hat{r} \mid \hat{r} \in [\hat{r}_j]\}$, which we want to enclose, is shown in Figure 4.2. We find $l = (2\sqrt{10}, 5\sqrt{5})^T$, and see that the second component of $[\hat{r}_j]$ induces the direction in which the parallelepiped has the largest span. Hence

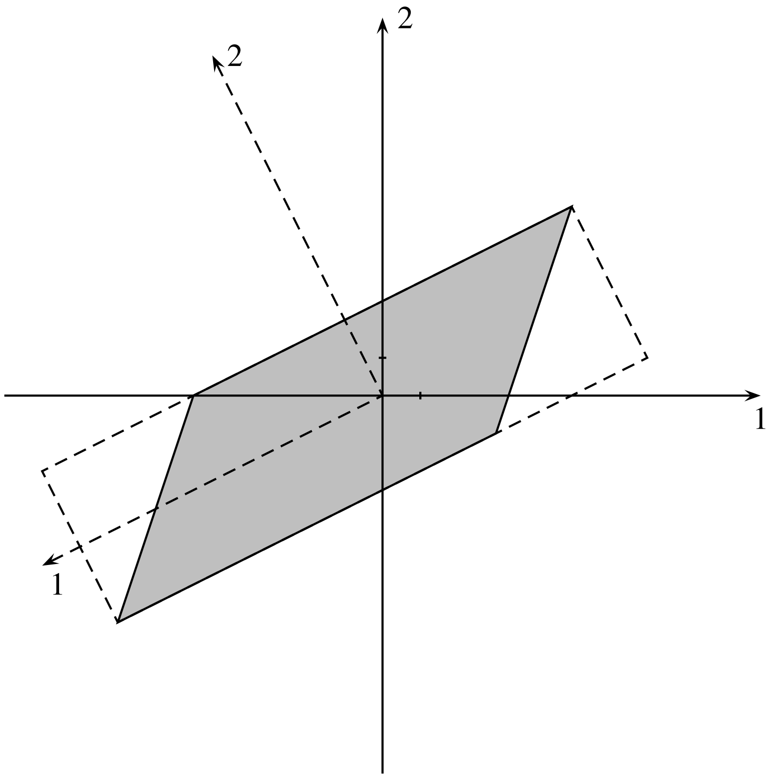


Figure 4.2: The parallelepiped $\{\tilde{A}_{j+1}\hat{r} \mid \hat{r} \in [\hat{r}_j]\}$ (grey area), with $\tilde{A}_{j+1}$ and $[\hat{r}_j]$ given in Eq. (4.30), the coordinate system induced by A_{j+1}, and the rectangular enclosure of the parallelepiped.

the matrix $\hat{A}_{j+1}$ is chosen so that it contains the columns from $\tilde{A}_{j+1}$ in reverse order.

$$\hat{A}_{j+1} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}. \tag{4.31}$$

A QR-factorization of $\hat{A}_{j+1}$ yields

$$A_{j+1} = \frac{1}{\sqrt{5}} \begin{pmatrix} -2 & -1 \\ -1 & 2 \end{pmatrix}. \tag{4.32}$$

The coordinate system induced by A_{j+1} is shown in Figure 4.2. The parallelepiped can then be enclosed by the rectangle $\{A_{j+1}\hat{r} \mid \hat{r} \in [\hat{r}_{j+1}]\}$,

$$[\hat{r}_{j+1}] = (A_{j+1}^T \tilde{A}_{j+1})[\hat{r}_j] = \frac{1}{\sqrt{5}} \begin{pmatrix} [-10,7] \\ [-5,5] \end{pmatrix}, \tag{4.33}$$

where we have used that A_{j+1} is orthogonal, hence $A_{j+1}^{-1} = A_{j+1}^T$. The rectangular enclosure is shown in Figure 4.2.

In practice, on a computer where we compute A_{j+1} using a *QR*-factorization based on floating point calculations, we cannot be sure that A_{j+1} is completely orthogonal. To enclose the true inverse, consider the following: We have that

$$\begin{aligned} A^{-1} &= A^T(AA^T)^{-1} \\ &= A^T + A^T(AA^T)^{-1} - A^TAA^T(AA^T)^{-1} \\ &= A^T + A^T(I - AA^T)(AA^T)^{-1} \\ &= A^T + A^T(I - AA^T)((I - (I - AA^T))^{-1} \end{aligned}$$

Consider a norm where $\|I\| = 1$, e.g. the max-norm $\|\cdot\|_\infty$, and let $q = \|I - AA^T\|$. Assume that $q < 1$, now we have

$$\|A^{-1} - A^T\|_\infty \le \frac{\|A^T\|_\infty \|I - AA^T\|_\infty}{\|I - (I - AA^T)\|_\infty} \le \frac{q}{1-q}\|A^T\|_\infty, \tag{4.34}$$

which means that we can enclose the correct inverse by

$$A^{-1} \in A^T + \{[-d,d]\}, \quad d = \frac{q}{1-q}\|A^T\|_\infty, \tag{4.35}$$

where $\{[-d,d]\}$ is an $n \times n$ interval matrix with all elements equal to $[-d,d]$.

Consider again the map in Eq. (4.3), which rotates points in $\mathbb{R}^2$ clockwise in an angle of Δt about the origin. This time we form the extended mean value enclosure using the "*QR*-factorization method". Since the map Eq. (4.3) is completely known, we have that $\varepsilon(y) = 0$ in Eq. (4.4). This means that $[z_j] = 0$ for $j \ge 1$ in Algorithm 4.2. Furthermore the Jacobian

$$[S_j] = \tilde{\Phi}'([y_j]) = \begin{pmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{pmatrix} \tag{4.36}$$

is an real orthogonal matrix so, $A_{j+1} = [S_j]$, assuming that no permutation is done before the *QR*-factorization. With these considerations in mind, we can simplify Algorithm 4.2 considerably.

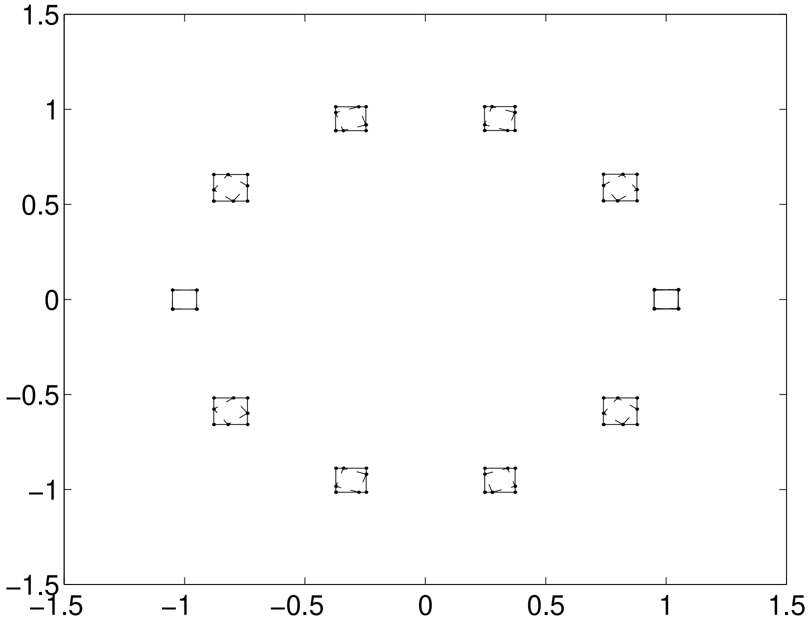


Figure 4.3: Ten interval iterations of the extended mean value enclosure of the map in Eq. (4.3) with $\Delta t = \frac{\pi}{5}$ and the initial interval vector $([x_0],[y_0]) = ([0.95,1.05],[-0.05,0.05])$. The solid rectangles are the interval vector enclosures $\{([x_j],[y_j])\}_{j=0,\dots,10}$ while the dashed rectangles are the rotating rectangle enclosures $\{A_{j+1}\hat{r} \mid \hat{r} \in [\hat{r}_{j+1}]\}$, for $j = 1,\dots,10$.

Using the same Δt and initial values as before and applying the extended mean value enclosure, we obtain the enclosures shown in Figure 4.3. From the figure we see that the extended mean value enclosure performs better on this example, compared to the natural interval extension that we used in Figure 4.1. Since the set $\mathrm{R}(\varphi_{rot},([x_j],[y_j]))$ itself is a rotating rectangle, we can enclose it perfectly, without any global overestimation. The only overestimation present in this example is when forming the local enclosure the rotating rectangle.

4.2 Enclosing iterates of the Cos-Sin map

Consider the Cos-Sin map, given by [26, 53]:

$$\varphi_{cs}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos(x+ay) \\ \sin(bx+y) \end{pmatrix}. \tag{4.37}$$

This map is a non-invertible map which maps the area $[-1,1] \times [-1,1]$ into itself. The map is completely known, and we can use $\varepsilon(y) = 0$ in Eq. (4.4). From an initial point (x_0, y_0), we generate a sequence of points in $\mathbb{R}^2$, $\{(x_j, y_j)\}_{j=0,\ldots}$ by $(x_{j+1}, y_{j+1}) = \varphi_{cs}(x_j, y_j)$. By forming the simple, the mean value, and the extended mean value interval enclosures of the iteration starting with the initial value $([x_0],[y_0]) = 10^{-6}([-1,1],[-1,1])$, and the parameters $a = 2$ and $b = 2$, we obtain three sequences of interval vectors. In Figure 4.4 the maximal widths of iterates $\max(\mathrm{w}([x_j]), \mathrm{w}([y_j]))$ for the three sequences are plotted versus the iteration number j.

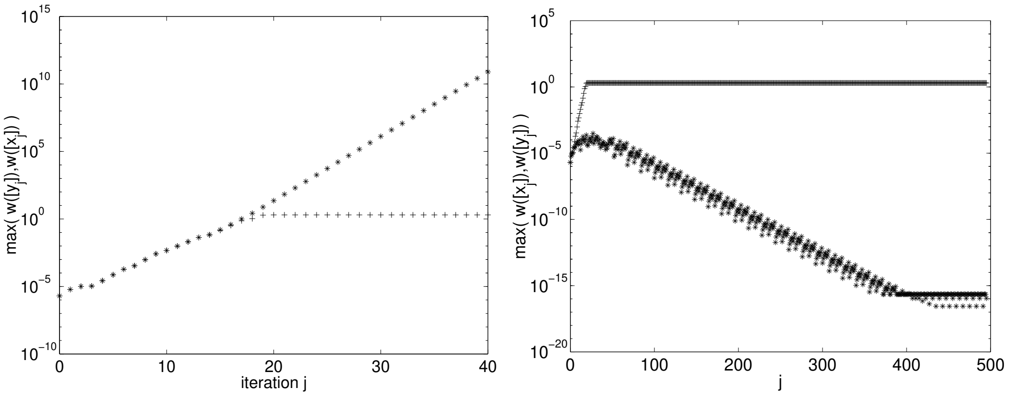


Figure 4.4: The width of three interval sequences enclosing iterates of the Cos-Sin map. (both figures) + : natural interval enclosure, (left figure) * : mean value enclosure for $j = 0, \ldots, 40$, (right figure) * : extended mean value enclosure for $j = 0, \ldots, 250$. The parameters are $a = 2$ and $b = 2$.

From Figure 4.4 we see that when using the natural interval enclosure, the width of the iterates grows very fast until they have reached the width 2 after approx. 20 iterations, then it stalls since the interval implementation of cos and sin returns intervals which lies in the interval $[-1,1]$. The width of the mean value enclosure also grows very fast, but does not stop at the width 2, but continues to grow because of overestimation. However the width of the extended mean value enclosure grows in the beginning, but then, after the a few iterations, it decays until a steady state is reached, after approx. 400 iterations. The width of the enclosure when it has reached the steady state is of the order of the machine accuracy and we cannot expect better enclosures. When examining the sequence $\{([x_j],[y_j])\}_{j=0,\ldots}$ from the extended mean value enclosure (not shown here), it appears that the sequence converges towards a period 8 fixed point, i.e., a sequence of points where $(x_j, y_j) = \varphi_{cs}^8(x_j, y_j)$. From the example, we see that wrapping can cause the width of the interval solution to grow unacceptably – causing the enclosures to be completely useless. But by using a method which fights the wrapping effect, the example shows that even if we start with some uncertainty in the initial value, this uncertainty will be damped throughout the iterative process. We can in principle continue the iteration of the extended mean value enclosure and enclose any iterate (x_j, y_j).

Now consider a modified Cos-Sin map,

$$\varphi_{mcs}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos(x+ay)+\varepsilon_x(x,y) \\ \sin(bx+y)+\varepsilon_y(x,y) \end{pmatrix}. \tag{4.38}$$

where the functions ε_x and ε_y are not exactly known, but known to belong to some interval $\varepsilon_x, \varepsilon_y \in [-e, e]$. Two interval sequences were generated using the extended mean value enclosure with the error bounds $e = 10^{-8}$ and $e = 10^{-7}$. The same parameters and initial values as before were used. The widths of the enclosures are plotted in Figure 4.5.

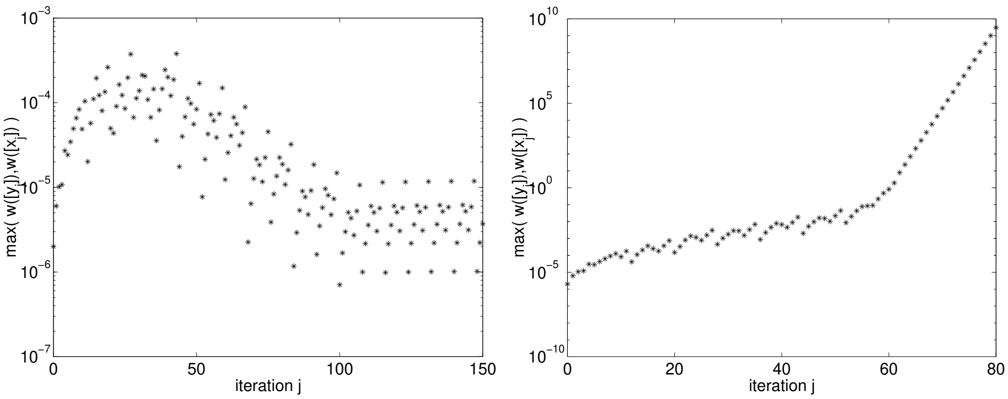


Figure 4.5: Enclosing iterates of the not completely known function φ_{mcs} using the extended mean value enclosure. Left figure, using $e = 10^{-8}$. Right figure, using $e = 10^{-7}$. The parameters are $a = 2$ and $b = 2$.

From Figure 4.5 we see that even if each step in the iterative process is not completely known, we are still able to form a good enclosure of the sequence ($e = 10^{-8}$), however, if the uncertainty gets too big, the enclosure will after some iterations get too wide and not be very informative ($e = 10^{-7}$).

We can even also allow the parameters a and b to be intervals; using $[a] = 2+[-d,d]$, $[b] = 2+[-d,d]$ for $d = 10^{-8}, 10^{-7}$ and $e = 10^{-8}$, we generated two interval sequences using the same initial values as before. The width of the enclosures are plotted in Figure 4.6.

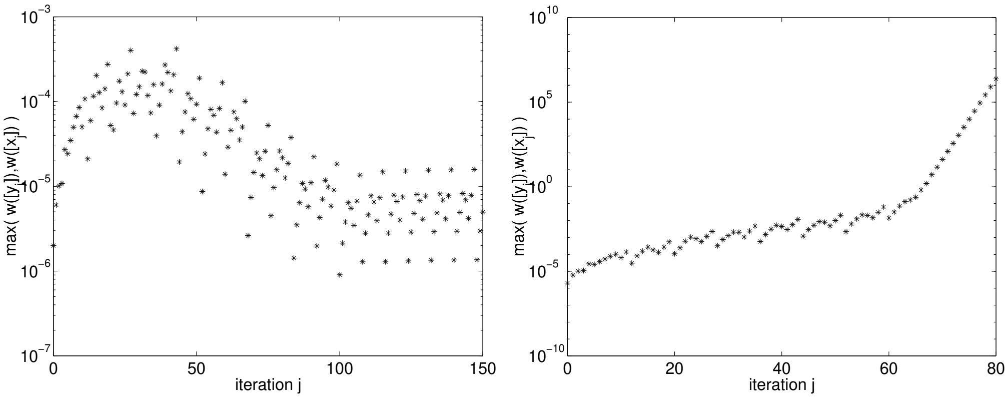


Figure 4.6: Enclosing iterates of the not completely known function φ_{mcs} using the extended mean value enclosure. Left figure, using $a,b = 2+[-1,1]10^{-8}$ and $e = 10^{-8}$. Right figure, using $a,b = 2+[-1,1]10^{-7}$ and $e = 10^{-8}$.

As in the previous example, where we only had uncertainty on ε_x and ε_y, we see that we are able to enclose all iterations of the map with all combinations of the uncertaincies introduced, and as long as these uncertaincies are reasonably small, still obtain narrow bounds.

5 Automatic differentiation

In the previous sections, we have seen that derivatives are quite important in connection with interval analysis. We have used derivatives in Section 3 for obtaining narrow bounds when evaluating functions and for proving existence of solutions to non-linear equations using the Newton or the Krawczyk operators. In Section 4 derivatives were used in connection with obtaining narrow interval vector enclosures when evaluating iterative mappings.

Derivatives are important in other areas of science as well, but the use of exact derivatives have been quite limited due to the misconception that they are hard to obtain. Many people still think that the only alternative to the symbolic way of obtaining derivatives is to use divided differences in which the difficulties in finding an expression for the derivatives are avoided. By using divided differences, truncation errors are introduced, which usually have a negative effect on further computations – in fact they can lead to very inaccurate results.

The use of a symbolic differentiation package such as Maple or Mathematica can solve the problem of obtaining expressions for the derivatives. This method obviously avoids truncation errors, but these packages usually have problems in handling large expressions and the time/space usage for computing derivatives can be enormous. In worst case it can even cause a program to crash. Furthermore, common subexpressions are usually not identified in the expressions and this leads to unnecessary computations during the evaluation of the derivatives.

Automatic differentiation (AD) is an alternative to the above methods. Here derivatives are computed by using the chain rule for composite functions. In automatic differentiation the evaluation of a function and its derivatives are calculated using the same code and common temporary values. If the code for the evaluation is optimized, then the computation of the derivatives will be optimized as well. The resulting differentiation is accurate up to roundoff errors. If we calculate the derivatives using interval arithmetic we obtain enclosures of the true derivatives. Automatic differentiation is easy to implement in languages with operator overloading such as C++, Fortran 90, Java, Ada, and PASCAL–XSC: See e.g. [25] for a survey of available AD tools.

5.1 Rational functions, code-lists and computational graphs

Assume that $f : \mathbb{R}^m \to \mathbb{R}^n$ is a rational function, given by an expression in which only rational operations occurs, e.g. the *elementary operations*: $+, -, *, /$, sin, exp, etc. We can decompose the expression for $f(x)$, $x=(x_1,\dots,x_m)$, into a list of equations representing the function.

$$\begin{aligned}\tau_i(x) &= g_i(x)=x_i \text{ for } i=1,\dots m, &(5.1a)\\ \tau_i(x) &= g_i(\tau_1(x),\dots,\tau_{i-1}(x)) \text{ for } i=m+1,\dots,l, &(5.1b)\end{aligned}$$

where all the functions τ_i and g_i are scalar functions and only one elementary operation occurs in each of the functions g_i. Such a list of equations are called a *code-list*. Let a_i be the arity (number of dependencies) of the ith function g_i in the code-list. Elementary functions usually have an arity of 0, 1, or 2, corresponding to a constant, an unary, or a binary operation. Define the map

$$\kappa_i : \{1,\dots,a_i\} \mapsto I_i \subset \{1,\dots,i-1\}, \tag{5.2}$$

so that Eqs. (5.1a-5.1b) can be written

$$\tau_i = g_i(\tau_{\kappa_i 1},\dots,\tau_{\kappa_i a_i}) \text{ for } i=1,\dots l. \tag{5.3}$$

As an example, consider the function $f:\mathbb{R}^3\to\mathbb{R}^2$ given by

$$f(x,y,t)=\begin{pmatrix} A+x(xy-B-1)+a\cos(\omega t)\\ x(B-xy)\end{pmatrix}, \tag{5.4}$$

where A,B,a and ω are given constants. Introducing the scalar functions $\tau_i = \tau_i(x,y,t)$, this function can be decomposed into the following code-list

$$\begin{aligned}
\tau_1 &= x, & \tau_9 &= \tau_5-\tau_8,\\
\tau_2 &= y, & \tau_{10} &= \tau_1\cdot\tau_9,\\
\tau_3 &= t, & \tau_{11} &= \tau_4-\tau_1,\\
\tau_4 &= A, & \tau_{12} &= \tau_{11}-\tau_{10},\\
\tau_5 &= B, & \tau_{13} &= \tau_7\cdot\tau_3,\\
\tau_6 &= a, & \tau_{14} &= \cos(\tau_{13}),\\
\tau_7 &= \omega, & \tau_{15} &= \tau_6\cdot\tau_{14},\\
\tau_8 &= \tau_1\cdot\tau_2, & \tau_{16} &= \tau_{12}+\tau_{15},
\end{aligned} \tag{5.5}$$

where $f=(\tau_{16},\tau_{10})$. The number of elementary operations used in Eq. (5.4) is 12 while it is 9 in Eq. (5.5). The reason for this difference is that we have used $\tau_{10}(x,y,t)=x(B-x\cdot y)$ as a common subexpression.

The code-list can also be represented as a *directed acyclic graph (DAG)*, which is a graph where the functions τ_i are represented by nodes, and the dependencies are represented by vertices. The graphs are directed to indicate which

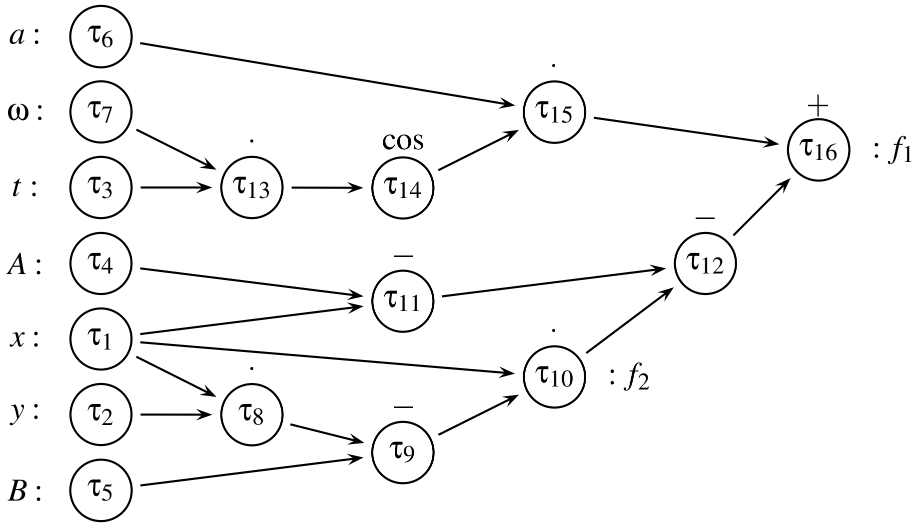


Figure 5.7: A directed acyclic graph (DAG) representing the code-list in Eq. (5.5).

way the dependencies goes, and they are acyclic since τ_i cannot depend on τ_j, for $j > i$.

Figure 5.7 shows the DAG representation of the code-list in Eq. (5.5).

When a computer is used to evaluate a program implementing the function f, the actual operations performed correspond to the operations in the code-list. That is, the computer will interpret the expression into a list of simple operations similar to those in the code-list and the values obtained when evaluating the functions τ_i, at runtime correspond to temporary variables. Hence, we call this the computational graph.

5.2 Theory of the Forward and Backward modes

Assume that $f : \mathbb{R}^m \to \mathbb{R}^n$ is a rational function, given by an expression which is decomposable into a code-list given by the functions g_i. Furthermore assume that all the functions g_i are differentiable, and that we can obtain their derivatives. Using the chain rule for composite functions on Eqs. (5.1a-5.1b), we obtain

$$\frac{\partial \tau_i}{\partial \tau_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial g_i}{\partial \tau_k} \frac{\partial \tau_k}{\partial \tau_j}, \quad \text{where} \quad \delta_{ij} = \begin{cases} 1 & i = j, \\ 0 & \text{otherwise}, \end{cases} \tag{5.6}$$

for $j \le i \le l$. Using $\tau = \{\tau_i\}_{i=1,\dots,l}$ and $g = \{g_i\}_{i=1,\dots,l}$, the code-list Eqs. (5.1a-5.1b) can be written as $\tau = g(\tau)$. Introducing the matrices

$$Dg = \left\{ \frac{\partial g_i}{\partial \tau_j} \right\}_{i,j=1,\dots,l} = \begin{pmatrix} 0 & \dots & & \\ \frac{\partial g_2}{\partial \tau_1} & 0 & \dots & \\ \frac{\partial g_3}{\partial \tau_1} & \frac{\partial g_3}{\partial \tau_2} & 0 & \dots \\ \dots & \ddots & \ddots & \ddots \end{pmatrix}, \tag{5.7}$$

and

$$D\tau = \left\{ \frac{\partial \tau_i}{\partial \tau_j} \right\}_{i,j=1,\dots,l} = \begin{pmatrix} 1 & 0 & \dots & \\ \frac{\partial \tau_2}{\partial \tau_1} & 1 & \ddots & \\ \frac{\partial \tau_3}{\partial \tau_1} & \frac{\partial \tau_3}{\partial \tau_2} & 1 & \ddots \\ \dots & \ddots & \ddots & \ddots \end{pmatrix}, \tag{5.8}$$

we can formulate Eq. (5.6) as the matrix equation

$$D\tau = I + Dg\,D\tau. \tag{5.9}$$

Since $I - Dg$ is a regular matrix, we have

$$\begin{aligned} D\tau = I + Dg\,D\tau & \quad \Leftrightarrow \\ (I - Dg)D\tau = I & \quad \Leftrightarrow \qquad (5.10) \\ D\tau = (I - Dg)^{-1} & \quad \Leftrightarrow \\ D\tau(I - Dg) = I & \quad \Leftrightarrow \\ (I - Dg)^T D\tau^T = I & \qquad\qquad (5.11) \end{aligned}$$

The matrix Dg is usually very sparse since only the columns $\{\kappa_i k\}_{k=1,\dots,a_i}$ in the ith row can contain non-zero elements. This sparsity will be exploited later when solving the above matrix equation w.r.t. $D\tau$. We can solve the equation either by using Eq. (5.10) or by using the transposed equation Eq. (5.11). Investigating these matrix equations more closely, we see some important differences.

In Eq. (5.10) we have

$$I = \begin{pmatrix} 1 & 0 & \dots & & 0 \\ -\frac{\partial g_2}{\partial \tau_1} & 1 & \ddots & & 0 \\ -\frac{\partial g_3}{\partial \tau_1} & -\frac{\partial g_3}{\partial \tau_2} & 1 & \ddots & 0 \\ \dots & \ddots & \ddots & \ddots & \vdots \\ -\frac{\partial g_n}{\partial \tau_1} & -\frac{\partial g_n}{\partial \tau_2} & \dots & -\frac{\partial g_n}{\partial \tau_{n-1}} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & \dots & & 0 \\ \frac{\partial \tau_2}{\partial \tau_1} & 1 & \ddots & & 0 \\ \frac{\partial \tau_3}{\partial \tau_1} & \frac{\partial \tau_3}{\partial \tau_2} & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \frac{\partial \tau_n}{\partial \tau_1} & \frac{\partial \tau_n}{\partial \tau_2} & \dots & \frac{\partial \tau_n}{\partial \tau_{n-1}} & 1 \end{pmatrix}. \tag{5.12}$$

This equation can be solved for any of the columns in $D\tau$ by forward substitution. By solving for the ith column, we obtain all derivatives with respect to τ_i. If we want to find derivatives w.r.t. all arguments of f, we solve for the first m columns of $D\tau$. This method is called forward mode automatic differentiation (FAD).

Algorithm 5.3 is an algorithm, using forward substitution, exploiting the sparsity of $D\tau$. The algorithm evaluates the function f and all its partial derivatives. It is very simple to alter the algorithm to find derivatives with respect to a subset of the arguments of f.

<u>Initialize the function evaluation and differentiation:</u>

$\tau_i = x_i,\ \hat{\tau}_{ij} = \delta_{ij},$ for $i, j = 1, \dots m.$

<u>Function evaluation and differentiation:</u>

for $i = m+1$ *to* l,

$\quad \tau_i = g_i(\tau_{\kappa_i 1}, \dots, \tau_{\kappa_i a_i}),$

$$\hat{\tau}_{ij} = \sum_{k=1}^{a_i} \frac{\partial g_i}{\partial \tau_{\kappa_i k}} \hat{\tau}_{\kappa_i k, j} \text{ for } j = 1, \dots m.$$

<u>Output:</u>

$$\{\hat{\tau}_{ij}\}_{i=1,\dots,l,\ j=1,\dots,m} \quad (= \frac{\partial \tau_i}{\partial \tau_j})$$

Algorithm 5.3: The forward mode automatic differentiation method (FAD).

From Eq. (5.11) we have

$$I = \begin{pmatrix} 1 & -\frac{\partial g_2}{\partial \tau_1} & \dots & -\frac{\partial g_{n-1}}{\partial \tau_1} & -\frac{\partial g_n}{\partial \tau_1} \\ \vdots & \ddots & \ddots & \ddots & \dots \\ 0 & \ddots & 1 & -\frac{\partial g_{n-1}}{\partial \tau_{n-2}} & -\frac{\partial g_n}{\partial \tau_{n-2}} \\ 0 & & \ddots & 1 & -\frac{\partial g_n}{\partial \tau_{n-1}} \\ 0 & & \dots & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & \frac{\partial \tau_2}{\partial \tau_1} & \dots & \frac{\partial \tau_{n-1}}{\partial \tau_1} & \frac{\partial \tau_n}{\partial \tau_1} \\ \vdots & \ddots & \ddots & \ddots & \dots \\ 0 & \ddots & 1 & \frac{\partial \tau_{n-1}}{\partial \tau_{n-2}} & \frac{\partial \tau_n}{\partial \tau_{n-2}} \\ 0 & & \ddots & 1 & \frac{\partial \tau_n}{\partial \tau_{n-1}} \\ 0 & & \dots & 0 & 1 \end{pmatrix}. \tag{5.13}$$

Equation Eq. (5.13) can be solved using backward substitution. If we solve the equation w.r.t. the ith column in $D\tau^T$, we obtain derivatives of τ_i w.r.t. all arguments of f (τ_j for $1 \le j \le m$). This method is called backward mode automatic differentiation (BAD).

Let $\mathcal{D}$ be the set of indices of the functions τ_i which are function values of f. Algorithm 5.4 is an algorithm which evaluates the function f and all its partial derivatives using backward substitution and exploiting the sparsity of $D\tau$.

The surprising property of the backward method, and the reason why it has become very popular, is that we are capable of computing all partial derivatives of a scalar function $f : C^1(\mathbb{R}^m, \mathbb{R})$ just by solving w.r.t. one column of $D\tau^T$ in Eq. (5.13), whereas in Eq. (5.12) we must solve w.r.t. m columns of $D\tau$ to obtain all partial derivatives. The rule of thumb when choosing which method to use for obtaining the Jacobian of $f : C^1(\mathbb{R}^m, \mathbb{R}^n)$ is to use the forward mode if $m \le n$ and the backward mode if $m > n$. Another main difference between the two methods is that, in the forward mode we obtain derivatives along with the function evaluation while. In the backward mode we compute the function values first while saving all intermediate results, and then compute the derivatives using the code-list in the reverse order. Because of this reverse order evaluation of the derivatives, we have to obtain the dependencies and the values of the intermediate variables used when evaluating the function. This corresponds to "recording" a representation of the DAG for the function.

In both the forward and the backward modes we use partial derivatives of the functions g_i. In the following tables, we summarize the most commonly used operations and standard functions and their derivatives.

<u>Initialize the function evaluation:</u>

$\tau_i = x_i,$ for $i = 1, \ldots m.$

<u>Function evaluation:</u>

for $i = m+1$ to $l,$

$\quad \tau_i = g_i(\tau_{\kappa_i 1}, \ldots, \tau_{\kappa_i a_i}),$

<u>Initialize the backward differentiation:</u>

$\hat{\tau}_{ij} = \delta_{ij}$ for $i \in \mathcal{D},\ j = 1, \ldots, l.$

<u>Backward differentiation:</u>

for $j = l$ *downto* $m+1,$

$$\hat{\tau}_{i,\kappa_j k} = \hat{\tau}_{i,\kappa_j k} + \frac{\partial g_j}{\partial \tau_{\kappa_j k}} \hat{\tau}_{ij} \text{ for } i \in \mathcal{D},\ k = 1, \ldots, a_j.$$

<u>Output:</u>

$$\{\hat{\tau}_{ij}\}_{i \in \mathcal{D},\ j=1,\ldots,l} \quad (= \frac{\partial \tau_i}{\partial \tau_j})$$

Algorithm 5.4: The backward mode automatic differentiation method (BAD).

Binary operations:		
$g_i(u,v)$	$\frac{\partial g_i}{\partial u}(u,v)$	$\frac{\partial g_i}{\partial v}(u,v)$
$u+v$	1	1
$u-v$	1	-1
$u\cdot v$	v	u
$\frac{u}{v}$	$\frac{1}{v}$	$\frac{-g_i(u,v)}{v}$
u^v	vu^{v-1}	$g_i(u,v)\ln(u)$

Unary operations:	
$g_i(u)$	$\frac{\partial g_i}{\partial u}(u)$
$+u$	1
$-u$	-1
$\exp(u)$	$\exp(u)$
$\log(u)$	$\frac{1}{u}$
$\sqrt{u}$	$\frac{1}{2g_i(u)}$
$\sin(u)$	$\cos(u)$
$\cos(u)$	$-\sin(u)$
$\tan(u)$	$1+g_i^2(u)$
$\mathrm{asin}(u)$	$\frac{1}{\sqrt{1+g_i^2(u)}}$
$\mathrm{acos}(u)$	$\frac{-1}{\sqrt{1-g_i^2(u)}}$
$\mathrm{atan}(u)$	$\frac{1}{1+g_i^2(u)}$

5.3 Theory of the Taylor expansion method

The Taylor expansion method is a generalization of the forward method, where instead of computing only the first derivatives, we obtain higher order derivatives using recursive rules. These rules, also called Taylor arithmetic, is applied on the code-list for the function in order to obtain derivatives of all intermediate values in the same way that we did in the forward method. In this report we only consider Taylor expansions of functions in one variable.

The kth Taylor coefficient function of $f \in C^k(\mathbb{R},\mathbb{R}^n)$ is denoted by $f^{[k]}$,

$$f^{[k]} = \frac{f^{(k)}}{k!} = \frac{1}{k!}\frac{d^k f}{dt^k}, \tag{5.14}$$

where $f = f(t)$. The kth Taylor coefficient of f in the point of expansion $t_0 \in \mathbb{R}$ is denoted by $(f)_k$,

$$(f)_k = f^{[k]}(t_0) = \frac{f^{(k)}}{k!}(t_0). \tag{5.15}$$

Note that the zero order Taylor coefficient of f by definition is the function value in t_0, i.e., $(f)_0 = f(t_0)$. We have an important relationship between the Taylor coefficients of f and the Taylor coefficients of f',

$$(f)_{k+1} = \frac{1}{k+1}\left(\frac{1}{k!}\frac{d^k}{dt^k}\right)\frac{df}{dt}(t_0) = \frac{1}{(k+1)}\left(f'\right)_k. \tag{5.16}$$

This relationship will be used extensively.

Let $u(t)$ and $v(t)$ be k times differentiable functions. The elementary operations of the Taylor series arithmetic are [39]:

$$\begin{aligned}
(u+v)_k &= (u)_k + (v)_k, && (5.17a)\\
(u-v)_k &= (u)_k - (v)_k, && (5.17b)\\
(u \cdot v)_k &= \sum_{i=0}^{k}(u)_i(v)_{k-i} = \sum_{i=0}^{k}(u)_{k-i}(v)_i, && (5.17c)\\
(u/v)_k &= \frac{1}{(v)_0}\left((u)_k - \sum_{j=1}^{k}(v)_j(u/v)_{k-j}\right) \text{ for } (v)_0 \neq 0. && (5.17d)
\end{aligned}$$

The rule for division is formed by a simple rewriting of Eq. (5.17c). Let $w = u/v$, where $(v)_0 \neq 0$. Now we have

$$\begin{aligned}
(u)_k &= \sum_{j=0}^{k}(v)_j(w)_{k-j} = (v)_0(w)_k + \sum_{j=1}^{k}(v)_j(w)_{k-j} \Rightarrow\\
(w)_k &= \frac{1}{(v)_0}\left((u)_k - \sum_{j=1}^{k}(v)_j(w)_{k-j}\right).
\end{aligned}$$

If one of the functions u or v in the above binary operations is a constant, then all of the Taylor expansion formulas shown above can be simplified considerably. This is possible since all but the zero order Taylor coefficient of a constant is zero, i.e., $u(t) \equiv C \Rightarrow (u)_0 = C$ and $(u)_j = 0$ for $j = 1, \ldots$.

Because of symmetry in Eq. (5.17c) when $u = v$, we can make a special formula for the Taylor coefficients of the square function. For $k \geq 1$ we have

$$
\begin{aligned}
(u^2)_k &= \sum_{i=0}^{k} (u)_i (u)_{k-i} \\
&= \begin{cases} 2 \displaystyle\sum_{i=0}^{(k-1)/2} (u)_i (u)_{k-i}, & k \text{ is odd,} \\ 2 \displaystyle\sum_{i=0}^{(k-2)/2} (u)_i (u)_{k-i} + (u)_{k/2}^2, & k \text{ is even.} \end{cases}
\end{aligned}
\tag{5.18}
$$

For the square root, let $w = \sqrt{u}$ so that $w^2 = u$, by using Eq. (5.18) we obtain

$$
(u)_k = \begin{cases} 2(w)_0(w)_k + 2 \displaystyle\sum_{i=1}^{(k-1)/2} (w)_i (w)_{k-i}, & k \text{ is odd,} \\ 2(w)_0(w)_k + 2 \displaystyle\sum_{i=1}^{(k-2)/2} (w)_i (w)_{k-i} + (w)_{k/2}^2, & k \text{ is even.} \end{cases}
$$

Isolating $(w)_k = (\sqrt{u})_k$, we obtain

$$
(\sqrt{u})_k = \begin{cases} \frac{1}{2(\sqrt{u})_0} \left((u)_k - 2 \displaystyle\sum_{i=1}^{(k-1)/2} (\sqrt{u})_i (\sqrt{u})_{k-i} \right), & k \text{ is odd,} \\ \frac{1}{2(\sqrt{u})_0} \left((u)_k - 2 \displaystyle\sum_{i=1}^{(k-2)/2} (\sqrt{u})_i (\sqrt{u})_{k-i} - (\sqrt{u})_{k/2}^2 \right), & k \text{ is even.} \end{cases}
\tag{5.19}
$$

The formula for $w = u^a$, where a is a constant, can be derived using Eq. (5.16) and Eq. (5.17c). Assume that $u(t_0) \neq 0$, since $w' = au^{a-1}u' \Leftrightarrow w'u = au^a u' = awu'$ we have

$$
\sum_{j=0}^{k-1} (w')_j (u)_{k-1-j} = a \sum_{j=0}^{k-1} (w)_j (u')_{k-1-j} \text{ for } k \geq 1.
$$

From Eq. (5.16) we have $(w')_j = (j+1)(w)_{j+1}$. Using this relation we get

$$
k(w)_k(u)_0 + \sum_{j=0}^{k-1} j(w)_j(u)_{k-j} = a \sum_{j=0}^{k-1} (k-j)(w)_j(u)_{k-j} \text{ for } k \geq 1.
$$

Isolating $(w)_k = (u^a)_k$ gives the formula

$$(u^a)_k = \frac{1}{k(u)_0} \sum_{j=0}^{k-1} (a(k-j)-j)\,(u^a)_j (u)_{k-j} \text{ for } k \geq 1. \tag{5.20}$$

This formula can not be used when $u(t_0) = 0$. In this case we have to use another method.

The formula for $w = \exp u$ can be found in a similar way. Since $w' = wu'$ we have the relation

$$(\exp u)_k = \frac{1}{k} \sum_{j=0}^{k-1} (k-j)(w)_j (u)_{k-j} \text{ for } k \geq 1. \tag{5.21}$$

Formulas for cos and sin can be obtained from the relations $\cos' u = -\sin u \cdot u'$ and $\sin' u = \cos u \cdot u'$

$$(\cos u)_k = -\frac{1}{k} \sum_{j=0}^{k-1} (k-j)(\sin u)_j (u)_{k-j} \text{ for } k \geq 1, \tag{5.22}$$

$$(\sin u)_k = \frac{1}{k} \sum_{j=0}^{k-1} (k-j)(\cos u)_j (u)_{k-j} \text{ for } k \geq 1. \tag{5.23}$$

These relations have to be used in pairs.

Some other elementary functions can be expanded after the following considerations. Let $w = f(u)$ be a composite function where $w' = \frac{1}{g}u'$ and $\frac{1}{g} = \frac{df}{du}$. Assume that the Taylor coefficients of g can be obtained. Now we have $w'g = u'$, so

$$k(u)_k = \sum_{j=0}^{k-1} (w')_j (g)_{k-1-j} = \sum_{j=0}^{k-1} (j+1)(w)_{j+1} (g)_{k-(j+1)} = \sum_{j=1}^{k} j(w)_j (g)_{k-j}.$$

After isolating $(w)_k$, we obtain

$$(w)_k = \frac{1}{(g)_0} \left((u)_k - \frac{1}{k} \sum_{j=1}^{k-1} j(w)_j (g)_{k-j} \right) \text{ for } k \geq 1. \tag{5.24}$$

The following list of functions has been expanded using Eq. (5.24); see [51] for a more complete list.

$$(\log u)_k = \frac{1}{(u)_0}\left((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\log u)_j (u)_{k-j}\right), \tag{5.25}$$

$$(\tan u)_k = \frac{1}{\cos^2(u)_0}\left((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\tan u)_j (\cos^2 u)_{k-j}\right), \tag{5.26}$$

$$(\arcsin u)_k = \frac{1}{\sqrt{1-(u)_0^2}}\left((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\arcsin u)_j (\sqrt{1-u^2})_{k-j}\right), \tag{5.27}$$

$$(\arccos u)_k = \frac{-1}{\sqrt{1-(u)_0^2}}\left((u)_k + \frac{1}{k}\sum_{j=1}^{k-1} j(\arccos u)_j (\sqrt{1-u^2})_{k-j}\right), \tag{5.28}$$

$$(\arctan u)_k = \frac{1}{1+(u)_0^2}\cdot\left((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\arctan u)_j (1+u^2)_{k-j}\right), \tag{5.29}$$

all for $k \geq 1$.

Assume that $f : \mathbb{R}^m \to \mathbb{R}^n$ is a rational function given by an expression decomposed into a code-list given by the functions g_i as in Eqs. (5.1a-5.1b). Furthermore assume that all the functions g_i are k times differentiable and that we can obtain recursive formulas for their derivatives. Algorithm 5.5 computes Taylor coefficients of $f(x)$ to order k, using the code-list and the recursive rules, with respect to the pth component in $x = (x_1, \dots, x_m)$, where $1 \leq p \leq m$.

Since we have the relation Eq. (5.16) we can compute Taylor coefficients of a function $u(t)$ given implicitly by an ordinary differential equation

$$u' = f(u). \tag{5.30}$$

Using the recursive relation

$$(u)_{k+1} = \frac{(f)_k}{k+1} \tag{5.31}$$

and some initial value $(u)_0 = u(t_0) \in \mathbb{R}^n$, we use Algorithm 5.5 to compute $(u)_1 = (f)_0$, then $(u)_2 = \frac{(f)_1}{2}$ and so forth. In practice, one saves the Taylor coefficients of all the intermediate functions $\{\tau_{ij}\}_{i=1,\dots,l,\ j=0,\dots,k}$, when obtaining the kth order coefficients, since they are unchanged when computing the $k+1$th Taylor coefficients[16].

Initialize the Taylor coefficients of the arguments:

$\tau_{i0} = x_i,\ \tau_{i1} = \delta_{ip},\ \tau_{ij} = 0,\ \text{for } i = 1, \dots m,\ j = 2, \dots k$

Function evaluation in Taylor arithmetic:

for $i = m+1$ *to* l,

for $j = 0$ *to* k,

$$\tau_{ij} = (g_i)_j, \quad (= g_i^{[j]}(\tau_{\kappa_i 1}, \dots, \tau_{\kappa_i a_i}))$$

Output:

$$\{\tau_{ij}\}_{i=1,\dots,l,\ j=0,\dots,k} \quad \left(= \frac{\partial^j \tau_i}{\partial \tau_p^j}\right)$$

Algorithm 5.5: The Taylor expansion method.

5.4 The FADBAD/TADIFF packages

Two program packages FADBAD and TADIFF have been developed for doing automatic differentiation of functions implemented as C++ programs [5, 6]. Both packages work by overloading every arithmetic operation used in the program implementing the function. This overloading does not affect the functionality of the program, but adds computations of the derivatives as described previously.

The arithmetic used for performing the computations, called the *basic type* in this connection, can be chosen freely when using FADBAD/TADIFF. This way, programs based on double precision arithmetic, interval arithmetic, etc. can be differentiated using the same library of overloaded operators, and the usage of the packages is the same no matter what basic type one chooses.

- FADBAD is a C++ program package which implements the forward and backward automatic differentiation modes. As already discussed, the forward method is mainly used for differentiating programs with few input variables but many function values, i.e., of the type $f : C^1(\mathbb{R}^m, \mathbb{R}^n)$, $m \leq n$, whereas the backward mode is superior on programs which compute few function values but have many input variables, i.e., $m > n$. Usually the forward method is preferable on functions of the type $n = m$. These rules are just rules of thumb, as the optimal method depends on the actual structure of the computations performed [20].

- TADIFF is also a C++ program package, but this package implements a method which specializes in computing Taylor expansions, i.e., higher order derivatives with respect to one variable. This is also possible using the FADBAD package, but since FADBAD is designed to compute one order of derivatives at a time, it is not optimal to use it for performing Taylor expansions.

To invoke the overloaded operators, so that derivatives are computed when running the program, one changes the names of the arithmetic types used in the program. If the arithmetic type of the computations performed in a program is, e.g. `INTERVAL`, then automatic differentiation is enabled by changing occurrences of `INTERVAL` in the program to one of the types `FINTERVAL`, `BINTERVAL`, or `TINTERVAL`, depending on which method to use. Since the computation of the derivatives using one of the three implemented methods in FADBAD/TADIFF again can be differentiated, it is possible to generate higher order derivatives in an extremely flexible way by combining the methods. Types which implement combinations of the three methods are named: FF, FB, FT, BF, BB, BT, TF, TB, TT, FFF, FFB,... and so forth, depending on the methods and order of differentiation. The automatic differentiation libraries in FADBAD/TADIFF are generic, so that any combination of the methods is possible. This flexibility opens up for a whole new range of applications.

5.4.1 Types and states of arithmetic variables

When declaring variables to be one of the three automatic differentiation types, some extra functionality is added. In order to describe how to use this functionality when a program using automatic differentiation is being executed, we have to define some types and states of variables.

- *Temporary variables* are variables that have been used during a computation to store intermediate values and then later discarded. This includes variables introduced by the compiler to contain temporary results in evaluation of expressions. These variables are previously used variables which are not accessible by the user in the active scope of the program.

- *Active variables* are variables that are declared in the currently active scope of the program. If a new value is assigned to an active variable, the old value of the active variable is remembered using a temporary variable.

These two types of variables exclude each other. Furthermore the variables can have one of two states:

- *Dependent variables* are variables whose values are results of expressions in which variables occured. Also an assignment to another variable makes the assigned variable dependent.

- *Independent variables* are variables which are not dependent. I.e., variables whose values has been assigned to a constant or an expression in which only constants occured. Uninitialized variables are independent.

The type and state of a variable is dependent on its place in the program and the state of the execution. Consider the function `brussel` in Program 5.1. This function is a C++ implementation of Eq. (5.4) using the arithmetic type `type` for performing the evaluation.

We can also present the function as the computational graph shown in Figure 5.7. Each node in the graph represents a variable which has been used during the function execution. A vertex corresponds to a dependency: If the arrow on the vertex is pointing to the node, then this node is dependent of the node in the other end of the vertex. A node is independent if no arrow is pointing to it. E.g. node number 1 is independent, corresponding to the independent variable `x`, node number 16 is dependent, corresponding to the variable `xp`. Node number 10 is also dependent, corresponding to the temporary variable `tmp`, which was used internally in `brussel`, and the dependent variable `yp`. From the graph, we see that also temporary variables used internally in expression evaluations are nodes in the graph, e.g. node number 8 corresponds to the subexpression `x*y` in the expression for the variable `tmp`.

Normally when implementing a function f as a C++ function, the independent variables in the C++ program correspond to the arguments of f, while the dependent variables would correspond to function values of f. In terms of automatic differentiation of a C++ function, we differentiate the dependent variables with respect to the independent variables.

5.4.2 Using the forward mode (FAD)

The forward mode of AD is probably the most easy to use since this method does not need any recording of dependencies. Assume that the variables `v` and `w` have been declared to be of `Ftype`, where `type` is some arithmetic type (e.g. another automatic differentiation type). Furthermore assume that `w` during the

Program 5.1 A simple C++ program, based on the arithmetic type `type`.

```
#include <math.h>
#include <type.h>

// Declare some independent variables:
type A(2.0/5),B(6.0/5),o(M_PI/4),a(.03);

void brussel(type &xp,type &yp,type x,type y,type t)
{
  type tmp(x*(B-x*y));    // Declare dependent variable tmp

  xp=A-x-tmp+a*cos(o*t);    // xp is now a dependent variable.
  yp=tmp;                   // yp is now a dependent variable.
}                           // tmp runs out of scope, it is
                            // now a temporary variable.


void main()
{
  // Declare independent variables:
  type x(0.5),y(1.5),t(0),xp,yp;

  brussel(xp,yp,x,y,t);     // Function evaluation.

  // The variables xp and yp are here dependent variables.
  // They are dependent on x,y and t.
}
```

function evaluation becomes dependent of the independent variable `v`. We have the following member functions:

- `void v.diff(int i, int m)` is called *before* the function which makes `w` dependent of `v`. It indicates that `v` is the `i`'th variable out of `m` that we want to differentiate with respect to. This actually corresponds to the initialization in Algorithm 5.3.
- `type w.d(int i)` is called after the function which makes `w` dependent of `v`. It is used to obtain the derivatives of `w` w.r.t. the `i`'th variable as indicated by `v.diff(i,m)`. This corresponds to reading the output values in Algorithm 5.3. The value of the derivative is returned in the underlying type; `type`.
- `type v.x(int i)` or `type w.x(int i)` can be called at any time. It returns the value of the variable using the underlying type; `type`.

If we want to differentiate the dependent variables `xp` and `yp` returned from the C++ function `brussel` in Program 5.1 w.r.t. the independent variables `x` and `y`, we change the types of the variables in `brussel` and `main` to `Ftype` and insert the following piece of code in main instead of `brussel(xp,yp,x,y,t);`.

```
x.diff(0,2);              // Indicate that we want to
y.diff(1,2);              // differentiate wrt. x and y.
brussel(xp,yp,x,y,t);     // Evaluation and differentiation.
```

After the call to `brussel` in `main`, we have the values `xp.x()`, `yp.x()`, and their derivatives `xp.d(0)`, `xp.d(1)`, `yp.d(0)`, `yp.d(1)`.

5.4.3 Using the backward mode (BAD)

Assume that the variables `v` and `w` have been declared to be of `Btype`, where `type` is some arithmetic type (e.g. another automatic differentiation type). Furthermore assume that `w` during the function evaluation becomes dependent of the independent variable `v`. We have the following member functions:

- `void w.diff(int i, int n)` is called *after* the function which makes `w` dependent of `v`. It indicates that `w` is the `i`'th function value out of `n` that we want to differentiate. This actually correspond to the initialization of the backward differentiation in Algorithm 5.4. Using this function we will also trigger the backward differentiation, so that derivatives are propagated backwards in the computational graph which was obtained when the function was evaluated. This backward propagation has been programmed so that each node in the graph awaits results from all other nodes dependent on it before propagating derivatives to the nodes which the node itself is dependent on. Since all dependencies in the graph should be triggered for the method to work properly it is very important to trigger the backward differentiation on *all* active and dependent variables obtained from the evaluation.

- `type v.d(int i)` is called after the backward differentiation. It is used to obtain the derivatives of the `i`'th variable as indicated by `w.diff(i,n)` w.r.t. `v`. This corresponds to reading the output values in Algorithm 5.4. The value of the derivative is returned in the underlying type; `type`.

- `type v.x(int i)` or `type w.x(int i)` can be called at any time, it will return the value of the variable using the underlying type; `type`.

If we want to differentiate the dependent variable `yp`, returned from the C++ function `brussel` in Program 5.1 w.r.t. the independent variables `x`, `y`, and `t`, we change the types of the variables in `brussel` and `main` to `Btype` and insert the following piece of code in main instead of `brussel(xp,yp,x,y,t)`.

```
brussel(xp,yp,x,y,t);     // Evaluation and differentiation.
xp=xp.x();                // Make xp an independent variable.
yp.diff(0,1);             // Differentiate yp.
```

After the call to `brussel` in `main`, we will have a computational graph equivalent to Figure 5.7 stored internally in the computer. With the assignment `xp=xp.x();` we assign a variable of type `type` to a variable of type `Btype`, the value of `xp` is unchanged, but `xp` is now considered as independent. This way when we trigger the backward differentiation using `yp.diff(0,1);` all active and dependent variables has been triggered. Also an assignment `xp=117;` would make `xp` independent, but this would change the value of `xp`. After the backward differentiation, `xp` and `yp` contains the function values, and the partial derivatives of `yp` can be found in `x.d(0)`, `y.d(0)`, and `t.d(0)`. When performing the backward differentiation, the allocated graph will automatically be deallocated.

5.4.4 Using the Taylor expansion method (TADIFF)

The Taylor expansion method implemented in TADIFF can be used in two different ways; it can be used for Taylor expanding functions given explicitly by a C++ function, or we can Taylor expand a function which is given implicitly as the solution of an ordinary differential equation $u' = f(u)$, where the right hand side is given explicitly by a C++ function.

Assume that the variables `v` and `w` have been declared to be of `Ttype`, where `type` is some arithmetic type (e.g. another automatic differentiation type). Furthermore assume that `w` during the function evaluation becomes dependent of the independent variable `v`. We have the following member functions:

- The index operator `[]` is used to access the Taylor coefficients. E.g. `v[i]=a` assigns the `i`'th Taylor coefficient of `v` to the value `a`. Since the values of the Taylor coefficients of dependent variables are dependent themselves, the user should only assign other values to Taylor coefficients of variables which are independent. After computing the Taylor coefficients of the dependent variables – see `w.eval(j)` later – these coefficients are accessible using the index operator, e.g. `a=w[i]`.

- `w.eval(k)` computes up to order `k`'th Taylor coefficients of `w`. This operation will also compute Taylor coefficients of all the intermediate values which was used to compute `w`. Note that, since `w` is dependent on `v`, the operation will use up to order `k` Taylor coefficients of `v`. These coefficients have to be initialized by the user, either by assigning a value to `v` before the recording of the tree, or explicitly by using the index operator `[]` as shown above. Using the index operator on a variable `v` will not disturb dependencies on `v`, while assigning a constant to `v` will decouple all dependencies of `v`.

- `w.reset()` resets the Taylor coefficients of the dependent variable `w` and Taylor coefficients of all dependent variables of which `w` is dependent, including temporary variables. The independent variables will not be affected. This operation is necessary if one wishes to reuse the computational graph to perform several Taylor expansions.

If we want to Taylor expand the function `brussel` in Program 5.1 with respect to the variable `t`, we first change the types of the variables in `brussel` and `main` to `Ttype` and insert the following piece of code in main instead of `brussel(xp,yp,x,y,t);`.

```
brussel(xp,yp,x,y,t);      // Record the computational graph.

// The variables xp and yp are here dependent variables.
// They are dependent on x,y and t.

t[1]=1;                    // Taylor expansion wrt. t
xp.eval(10);               // Compute xp[1],...,xp[10].
yp.eval(10);               // Compute yp[1],...,yp[10].
```

After the call to `brussel` in `main`, we have a computational graph equivalent to Figure 5.7 stored internally.

The line `t[1]=1;` sets the first order Taylor coefficient of `t` to the value 1. This indicates that the Taylor coefficients of `xp` and `xp` are to be calculated with respect to `t`. If the line were omitted, all Taylor coefficients, higher that order zero, will simply become zero. The line `xp.eval(10);` computes Taylor coefficients to order 10 of `xp`. The Taylor coefficients to order `i` of `xp` are available after the evaluation as `xp[0]`,...,`xp[i]`. All temporary results used in the computations are saved along the computational graph, so when we evaluate the next line `yp.eval(10);` the temporary variable corresponding to `tmp` in Program 5.1 does not have to be Taylor expanded once again.

Since we do not deallocate the computational graph when we perform the Taylor expansions it is possible to reuse the graph for Taylor expanding in other points as well. If we also want to expand in e.g. the point (`x,y,t`)=(`2,-0.5,1.8`) we could insert the following piece of code after the previously inserted code.

```
xp.reset();         // Resets dependent variables of which
yp.reset();         // either xp or yp is dependent.

x[0]=2;y[0]=-.5;    // New point of expansion is inserted
t[0]=1.8;           // in the zero order coefficients.

t[1]=1;             // Taylor expansion wrt. t
xp.eval(10);        // Compute xp[0],...,xp[10].
yp.eval(10);        // Compute yp[0],...,yp[10].
```

If the two lines resetting the dependent variables `xp` and `yp` were omitted, the statements `xp.eval(10);` and `yp.eval(10);` would do nothing since the

Taylor coefficients of `xp` and `yp` and all intermediate variables in the dependency graph already have been computed to order 10. It is also important to see how the values of the independent variables are changed, i.e., we use `x[0]=2;` and NOT `x=2;` as the latter would decouple the variable `x` from the computational graph.

As mentioned in Section 5.3, Taylor expansion can also be used for expanding a function given implicitly as the solution to an ordinary differential equation (ODE) $u' = f(u)$. Here we have the relation from Eq. (5.31) between the $k+1$th coefficient of the solution u and the kth coefficient of $f(u)$. Since this kind of dependency forms a kind of "feedback" in the variables which cannot be represented by a directed acyclic graph, we have to make additional code for performing this kind of Taylor expansion.

The function `expand` in Program 5.2 is an example which shows how the solution of the ODE $(x', y') = f(x, y, t)$, with f given in Eq. (5.4), can be expanded. The `main` program starts by "recording" the computational graph for `brussel` with independent variables: `x` and `y`, `t` and dependent variables: `xp`, `yp`. These variables are then used in `expand` to access the computational graph of `brussel` for Taylor expanding the solution of the ODE, here to order 10 at the point $(x, y, t) = (0.5, 1.5, 0)$, using the arithmetic type `double`.

5.4.5 Using combinations of methods

The real strength of FADBAD/TADIFF lies in the possibility to differentiate functions implemented as algorithms which themselves uses automatic differentiation. Unfortunately the structures of the variables, which are combinations of automatic differentiation types, can become *very* complicated, so to use this possibility it is important to implement the algorithm in a modular way and apply one automatic differentiation type to the program at a time [6]. Also the problem of keeping track of the types and the states of the variables in a program can be avoided by using scopes, e.g. function scopes, and it is a good rule to avoid global variables of automatic differentiation types.

Assume that a function `void f(type& o1,...,type& on,type& i1,..., type& im)` is an implementation of the function $f : \mathbb{R}^m \to \mathbb{R}^n$, with the input variables `i1`...`im` and the output variables `o1`...`on`. The function `f` is allowed to call/use other functions with automatic differentiation types, derived from `type`, i.e., with names ending with `type`, as input and output variables. To differentiate f using one of the three methods, all occurrences of the word `type` are replaced

Program 5.2 Taylor expanding the solution of an ODE.

```
#include <math.h>
#include "Tdouble.h"

double A(2.0/5),B(6.0/5),o(M_PI/4),a(.03);

void brussel(Tdouble &xp,Tdouble &yp,
             Tdouble x,Tdouble y,Tdouble t)
{
  Tdouble tmp(x*(B-x*y));   // Declare dependent variable tmp

  xp=A-x-tmp+a*cos(o*t);    // xp is now a dependent variable
  yp=tmp;                   // yp is now a dependent variable
}
void expand(Tdouble &xp,Tdouble &yp,
                Tdouble &x,Tdouble &y,Tdouble t,int order)
{
  xp.reset();              // Reset the computational
  yp.reset();              // graph of brussel.
  t[1]=1;                  // Taylor expand wrt. t.
  for(int i=0;i<order;i++) // One coefficient at a time
  {
    xp.eval(i);            // Evaluate the i'th order
    yp.eval(i);            // coefficients of xp and yp
    x[i+1]=xp[i]/(i+1);    // Use the relation:
    y[i+1]=yp[i]/(i+1);    //   (x',y')=f(x,y,t)
  }
}
void main()
{
  Tdouble x,y,t,xp,yp;     // Declare variables.

  x=0.5;y=1.5;t=0;          // Specify the point of expansion.
  brussel(xp,yp,x,y,t);    // Get the computational graph of
                           // the function brussel.
  expand(xp,yp,x,y,t,10);  // Compute the Taylor Expansion of
                           // brussel in the point (.5,1.5,0)
}
```

with `Ftype`, `Btype`, or `Ttype`, depending on the method to use, in the function `f` and all functions upon which `f` is dependent. Now we are capable of evaluating `f` with input and output variables which are of automatic differentiation type.

5.5 Examples using the FADBAD/TADIFF packages

In the Section 5.4.4 we saw how to use the TADIFF package for Taylor expanding a function given by an expression, and how to expand a function which is the solution of an ordinary differential equation. In this section we will see some more advanced applications using TADIFF, and we will also see how to use FADBAD to differentiate the computations we perform with TADIFF, using the strategy introduced in Section 5.4.5.

5.5.1 Numerical integration

Consider the following quadrature rule [32]

$$\begin{aligned} \int_a^b f(x)\,dx &= \frac{h}{70}\{\frac{1}{3}(41((f_a)_0+(f_b)_0)+128(f_m)_0)+h\{(f_a)_1-(f_b)_1 \\ &\quad +\frac{h}{18}\{(f_a)_2+(f_b)_2+16(f_m)_2\}\}\}+R, \end{aligned} \tag{5.32}$$

where $m=(a+b)/2$, $h=b-a$, and $(f_a)_i$ is a shorthand notation for the ith Taylor coefficient of f at the point a, etc. The remainder has the form

$$R = Ch^{10}f^{(10)}(\xi), \quad \xi\in[a,b], \tag{5.33}$$

$$C = \frac{1}{130977000}. \tag{5.34}$$

We will use this approximation piecewise on N subintervals of the interval $[0,\pi]$ to compute the value of the integral

$$I(c,d)=\int_0^\pi \ln(c+d\cos x)\,dx, \tag{5.35}$$

which has the true solution

$$I(c,d)=\pi\ln\left(\frac{c+\sqrt{c^2-d^2}}{2}\right). \tag{5.36}$$

The following C++ function is an implementation of the integrand

```
double f(double x, double c, double d)
{
  return log(c+d*cos(x));
}
```

After replacing all occurrences of `double` by `Tdouble` in the function `f` so that Taylor expansion of f is possible, we can implement the numerical integral as shown in Program 5.3. Note that in this program the actual recording of the computational graph of `f` takes place in the first line of `I`. This line also shows a neat trick which is possible using the TADIFF package: It is possible to record the computational graph without first initializing the independent variables. This has been made possible since the computational graph can be used to produce several Taylor expansions, using different values of the independent variables. One restriction to this method is that the actual values in the graph may not be used during the recording [6]. Since the Taylor expansion `fx.eval(2);` uses Taylor coefficients of `x` to the order 2, these coefficients has to be initialized before the expansion. This is done using the index operator `[]` on `x`. It is important to specify the Taylor coefficients of `x` to the same order as we are going to expand the function, in this case to order 2. Otherwise we get an error message from TADIFF.

Since the integral $I(c,d) : \mathbb{R}^2 \to \mathbb{R}$ in Eq. (5.35) is a differentiable function with the partial derivatives

$$\frac{\partial I}{\partial c}(c,d) = \frac{\pi}{\sqrt{c^2-d^2}}, \tag{5.37}$$

$$\frac{\partial I}{\partial d}(c,d) = -\frac{\pi \cdot d}{(c+\sqrt{c^2-d^2})\sqrt{c^2-d^2}}, \tag{5.38}$$

it is also possible to differentiate the program which computes the numerical approximation. To apply the backward automatic differentiation method from the FADBAD package, we replace all occurrences of `double` to `Bdouble` in the functions `f` and `I`. This way `Tdouble` is modified to `TBdouble`. The function `dI` below computes numerical approximations of the integral $I(c,d)$ and its partial derivatives with respect to c and d.

Program 5.3 Numerical integration using Taylor Expansion.

```
double I(double c, double d, double N)
{
  Tdouble x,fx(f(x,c,d));      // Record the graph of f.
  double fa[3],fm[3],fb[3],
         h(M_PI/N),sum(0);
  int i,j;
  x[1]=1;x[2]=0;               // Specify order 1 and 2 of x.

  x[0]=0;                      // Expand f in the left point
  fx.eval(2);                  // to order 2.
  for(j=0;j<=2;j++)            // Save the 0.,1. and 2. order
    fa[j]=fx[j];               // coefficients of f.

  for(i=0;i<N;i++)
  {
    fx.reset();                // Reset before using the graph.
    x[0]=(M_PI*(2*i+1))/(2*N);// Expand f in the midpoint
    fx.eval(2);                // to order 2.
    for(j=0;j<=2;j++)          // Save the 0.,1. and 2. order
      fm[j]=fx[j];             // coefficients of f.

    fx.reset();                // Reset before using the graph.
    x[0]=(M_PI*(i+1))/N;       // Expand f in the right point.
    fx.eval(2);                // to order 2.
    for(j=0;j<=2;j++)          // Save the 0.,1. and 2. order
      fb[j]=fx[j];             // coefficients of f.

    // Compute the integral, using the Taylor Coefficients:
    sum+=(41*(fa[0]+fb[0])+128*fm[0])/3+h*(fa[1]-fb[1]+
         h*(fa[2]+fb[2]+16*fm[2])/18);

    for(j=0;j<=2;j++)      // The right endpoint is the next
      fa[j]=fb[j];         // left endpoint.
  }
  return h*sum/70;
}
```

N	$I(c,d,N)$	$\frac{\partial I}{\partial c}(5,3,N)$	$\frac{\partial I}{\partial d}(5,3,N)$
2	4.725206749584612	0.7853779605604702	−0.2617657164041858
4	4.725198468158792	0.7853982250374074	−0.2617994905324145
8	4.725198500140277	0.7853981634075534	−0.2617993878159910
16	4.725198500142803	0.7853981633974484	−0.2617993877991494
	4.725198500142803	0.7853981633974483	−0.2617993877991494

Table 5.1: The result of differentiating a numerical integration for an increasing number of subintervals N using the values $c = 5$ and $d = 3$. The last line shows the values obtained when evaluating the expression of $I(c,d)$ and its partial derivatives using double precision.

```
void dI(double &Ival, double &dIdc, double &dIdd,
        double c, double d, int N)
{
  Bdouble Bc(c),Bd(d),     // Initialize the input variables
          BI(I(Bc,Bd,N)); // Compute the integral.

  BI.diff(0,1);           // Compute the partial derivatives
                          // of the integral wrt. Bc and Bd.

  Ival=BI.x();            // Store the value of the integral
  dIdc=Bc.d(0);           // and its partial derivatives in
  dIdd=Bd.d(0);           // the variables Ival,dIdc and dIdd.
}
```

When calling `dI` for $c = 5$ and $d = 3$, using different values of N, we obtain the numbers shown in Table 5.1. When we use 16 subintervals the result of the numerical integration and its partial derivatives are just as accurate as evaluating their true expressions in double precision.

5.5.2 Solving an initial value problem (IVP)

We have already seen in Section 5.4.4 how to Taylor expand the solution of an ordinary differential equation (ODE) of the form $u' = f(u)$ in some point $u(t_0) = u_0$. Since we only are capable of calculating a finite number of Taylor coefficients on a computer, our Taylor expansions are only local approximations to the true solution. If we wish to solve the ODE for some $t_n \gg t_0$, we have to discretize the interval $[t_0,t]$ in some points $t_0 < t_1 < \cdots < t_n$ and find an approximation to the solution pointwise in $t = t_i$ for increasing values of i. By computing the Taylor polynomial of order p in the point (u_i, t_i), and evaluating this polynomial at $t = t_{i+1}$, we obtain an approximation of the solution u_{i+1} in t_{i+1},

$$u_{i+1} = \sum_{k=0}^{p} (u_i)_k (t_{i+1} - t_i)^k, \tag{5.39}$$

where $(u_i)_k$ denotes the kth Taylor coefficient of the solution of $u' = g(u,t)$ in the point (u_i, t_i).

Consider the ODE $(x', y') = f(x, y, t)$ with f given in Eq. (5.4). The function `solve` in Program 5.4 uses the C++ implementation `brussel` of f given in Program 5.2 to compute an approximation of the solution for some $t > t_0$ based on a given initial value (x_0, y_0, t_0). The order of the Taylor expansions and the number of discretization points used in the interval $[t_0, t]$ are specified by the user. In the program, we use the relation that if $w(\tau) = u(h\tau)$ we have $\frac{dw}{d\tau}(\tau) = hf(w(\tau), \tau)$. If we choose $h = t_{i+1} - t_i$, the sum in Eq. (5.39) can be calculated as the sum of the Taylor coefficients for w. Hence we compute the Taylor expansion for w and not u.

Since the right hand side of the ODE is a periodic function with the period $T = 2\pi/\omega$, we can consider $u(p \cdot T)$, where p is a positive integer, as a discrete map of the initial value $u(0) = u_0$. This map is differentiable, and we can use Newton's method for locating periodic solution, i.e., find solutions of the equation

$$u_0 - u(p \cdot T) = 0, \quad \text{and } u' = f(u,t),\ \ u(0) = u_0 \tag{5.40}$$

for a given positive integer value of p.

For Newton's method, we need derivatives of $u_0 - u(p \cdot T)$ with respect to the initial value u_0. Since $u(p \cdot T)$ is a function of u_0 given approximately by the C++ function `solve`, we differentiate `solve`, by replacing all occurrences of `double` with `Fdouble` given by the FADBAD package. This way we can automatically obtain derivatives of `solve` with respect to the initial values. See Program 5.5 for an implementation of Newton's method. The program will after a few iterations find the periodic $p = 2$ solution with the initial values $(x(0), y(0)) \approx (0.385047, 3.25168)$.

It is worth noting that the initial value solver, when using the type `TFdouble` for evaluation, not only computes an approximation to the solution of $u' = f(u,t)$ but also computes an approximation to the solution of the variational equation

Program 5.4 Solving an initial value problem using Taylor expansion.

```
void solve(double &x,double &y, double &t, double tto,
           int N, int order)
{
  Tdouble Tx,Ty,Tt(0),xp,yp; // Declare variables.
  double h((tto-t)/N);
  int i,j;

  brussel(xp,yp,Tx,Ty,Tt);   // Get the computational graph of
                             // the function brussel.

  Tt[1]=h;                   // Taylor expand wrt. t.

  for(i=0;i<N;i++)
  {
    xp.reset();              // Reset the computational
    yp.reset();              // graph of brussel.

    Tx[0]=x;                 // Initialize the point of
    Ty[0]=y;                 // expansion by initializing
    Tt[0]=t;                 // the zero order coefficients.

    for(j=0;j<order;j++)     // One coefficient at a time
    {
      xp.eval(j);            // Evaluate the i'th order
      yp.eval(j);            // coefficients of xp and yp
      Tx[j+1]=xp[j]*h/(j+1); // Use the relation:
      Ty[j+1]=yp[j]*h/(j+1); //   (x',y')=f(x,y,t)

      x+=Tx[j+1];            // Evaluate the Taylor
      y+=Ty[j+1];            // polynomials.
    }
    t+=h;                    // We have a new solution point.
  }
}
```

Program 5.5 Taylor expanding the solution of an ODE, and searching for periodic solutions, using Newton's method.

```
void Newton(double &x,double &y)
{
  Fdouble Fx,Fy,Ft,IFx,IFy;
  double det,dx,dy;
  int i;

  do
  {
    Fx=x;Fy=y;Ft=0;           // Initial value of integration.

    Fx.diff(0,2);             // We want derivatives of the
    Fy.diff(1,2);             // integration wrt. x and y.

    IFx=Fx;IFy=Fy;            // Save the initial point.

    solve(IFx,IFy,Ft,16,40,7);// Solve the IVP in 2 periods.

    Fx-=IFx;Fy-=IFy;          // (Fx,Fy)=0 => periodic sol.

    det=Fx.d(0)*Fy.d(1)-Fy.d(0)*Fx.d(1);    // Compute
    dx=(Fy.d(1)*Fx.x()-Fx.d(1)*Fy.x())/det; // the Newton
    dy=(Fx.d(0)*Fy.x()-Fy.d(0)*Fx.x())/det; // correction.

    x-=dx;y-=dy;              // Make the Newton iteration.

  }while(dx*dx+dy*dy>1e-6);   // Repeat until convergence.
}

void main()
{
  double x(0.38),y(3.3);      // Initial guess.

  Newton(x,y);                // Find periodic solution.

}
```

$v' = D_u f(u,t)v$ where $D_u f$ denotes the Jacobian matrix of f with respect to u. All this, just by replacing `double` with `Fdouble`.

6 Enclosing solutions of ordinary differential equations

There are many algorithms for obtaining approximate solutions of ordinary differential equations. Unfortunately most algorithms are unable to give a realistic bound of the global error accumulated during the integration process. In many cases, the numerical solution found by some algorithm is not even close to the exact solution. Many people do not see this as a problem since only the overall behaviour of the system may be of interest, but even in this case some numerical algorithms might produce spurious behaviour which is not seen in the original system, e.g. spurious periodic orbits. The problem of keeping track of when spurious behaviour can occur using some numerical method is often quite complicated [56].

Using the method described in Section 4, we can obtain enclosures of the iterates found when discretizing ordinary differential equations using the Taylor expansion method described in Section 5.3 and a method described later to prove existence of the solution within some bounds.

We consider ordinary differential equations (ODE's) of the form

$$y' = f(y), \tag{6.1}$$

where $f \in C^k(D, \mathbb{R}^n)$, $D \subseteq \mathbb{R}^n$ is an open set, and y is a function of the independent variable t, i.e., $y = y(t)$.

6.1 Proving existence and uniqueness of the solution

For more details about the basics of proving existence of solutions to ordinary differential equations see [23, 14, 3].

Recall the well known Theorem [21]:

Theorem 4 (Contraction Mapping Theorem) *A mapping $T : S \to S$ of a closed subset S of a Banach space has exactly one fixed point $u^\star \in S$ so that $Tu^\star = u^\star$ if there exists a positive number $a < 1$, so that*

$$\|Tu - Tv\| \le a\|u - v\|, \text{ for all } u, v \in S. \tag{6.2}$$

T is said to be a contraction mapping on S.

Consider solutions to Eq. (6.1) which satisfy the initial condition $y(t_j) = y_j \in D$. With this initial condition of the solution, we have an integral equation equivalent to Eq. (6.1)

$$y(t) = y_j + \int_{t_j}^{t} f(y(s))\, ds, \text{ for } t \in [t_j, t_{j+1}]. \tag{6.3}$$

This equation has the same solution as Eq. (6.1) with the initial value $y(t_j) = y_j$ for $t \in [t_j, t_{j+1}]$. We define the Picard-Lindelöf operator, by

$$(Ty)(t) = y_j + \int_{t_j}^{t} f(y(s))\, ds, \text{ for } t \in [t_j, t_{j+1}]. \tag{6.4}$$

Consider the space of continuous functions $C^0([t_j, t_{j+1}], \mathbb{R}^m)$, with the norm

$$\|u\|_\alpha = \max_{t \in [t_j, t_{j+1}]} \left(e^{-\alpha(t-t_j)} \|u(t)\| \right), \tag{6.5}$$

for some fixed $\alpha > 0$. It can be shown that $C^0([t_j, t_{j+1}], \mathbb{R}^m)$ with this norm is a Banach space.

Let $S \subseteq C^0([t_j, t_{j+1}], \mathbb{R}^m)$ be a closed set and assume that S is mapped into itself by the Picard-Lindelöf operator. Furthermore assume that f satisfies the Lipschitz condition

$$\|f(u(t)) - f(v(t))\| \leq L\|u(t) - v(t)\|, \text{ for } u, v \in S, t \in [t_j, t_{j+1}]. \tag{6.6}$$

Let $\alpha > L$ be fixed. Now for $t \in [t_j, t_{j+1}]$ we have

$$\begin{aligned} \|(Tu)(t) - (Tv)(t)\| &\leq \int_{t_j}^{t} \|f(u(s)) - f(v(s))\|\, ds \\ &\leq L \int_{t_j}^{t} \|u(s) - v(s)\|\, ds. \end{aligned}$$

Hence

$$\begin{aligned} e^{-\alpha(t-t_j)}\|(Tu)(t) - (Tv)(t)\| &\leq L \int_{t_j}^{t} e^{-\alpha(t-s)} e^{-\alpha(s-t_j)} \|u(s) - v(s)\|\, ds \\ &\leq L\|u - v\|_\alpha \int_{t_j}^{t} e^{-\alpha(t-s)}\, ds \\ &\leq \alpha^{-1} L \|u - v\|_\alpha. \end{aligned}$$

Therefore

$$\|(Tu)(t) - (Tv)(t)\|_\alpha \leq \alpha^{-1} L \|u - v\|_\alpha.$$

By the Contraction Mapping Theorem, we have proven the existence of a unique solution to Eq. (6.3) in S provided that S is mapped into itself by T and f satisfies the Lipschitz condition Eq. (6.6).

Notice that by the Mean Value Theorem, we have that if the function f is differentiable, it automatically satisfies the Lipschitz condition Eq. (6.6).

6.2 Obtaining an interval vector enclosure of the solution

A rough enclosure of the solution to Eq. (6.1) with a fixed $y(t_j) = y_j \in [y_j]$, can be obtained by applying the Picard-Lindelöf operator [34, 35, 33, 41, 50, 49, 55]. Assume that $[\tilde{y}_j^0]$ is a superset of $[y_j]$ so we have that $y_j \in [y_j] \subseteq [\tilde{y}_j^0]$. Let S be the closed set of continuous functions in the interval $[t_j, t_{j+1}]$ bounded by the interval $[\tilde{y}_j^0]$,

$$S = \{u \mid u \in C^0([t_j, t_{j+1}], [\tilde{y}_j^0])\}. \tag{6.7}$$

Using the Picard-Lindelöf operator on a function $u \in S$, we obtain a continuous function $(Tu)(t)$, where

$$\begin{aligned} (Tu)(t) &\in y_j + \int_{t_j}^{t} F([\tilde{y}_j^0])\, ds \\ &= y_j + F([\tilde{y}_j^0]) \int_{t_j}^{t} 1\, ds \\ &\subseteq [y_j] + F([\tilde{y}_j^0])[0, h_j] = [\tilde{y}_j^1], \end{aligned} \tag{6.8}$$

for $t \in [t_j, t_{j+1}]$ and $h_j = t_{j+1} - t_j$. If the condition

$$[\tilde{y}_j^1] \subseteq [\tilde{y}_j^0] \tag{6.9}$$

is true, then S is mapped into itself. If f is differentiable in $[\tilde{y}_j^0]$, we have proven existence and uniqueness of the solution in $[\tilde{y}_j^0]$ (and in $[\tilde{y}_j^1]$) for $t \in [t_j, t_{j+1}]$. Notice that we can always find a step size h_j small enough in Eq. (6.8), so that Eq. (6.9) is satisfied.

Consider Algorithm 6.6, an algorithm for obtaining a rough enclosure of the solutions of Eq. (6.1) with initial values in the interval $[y_j]$. The input variables are a guess of the step size h, the desired number of Picard iterations before proving the existence and uniqueness of the solution IT_{normal}, the maximum number of Picard iterations before reducing the step size IT_{max}, and a parameter e for the epsilon-inflation $[\tilde{y}_j^0] = (1+e)[\tilde{y}_j^1] - e[\tilde{y}_j^1]$ which generates an interval vector $[\tilde{y}_j^0] \supset [\tilde{y}_j^1]$ around $[\tilde{y}_j^1]$. The output of the algorithm is a rough enclosure $[\tilde{y}_j^1]$, the actual step size performed h, and a guess of the next step size h_{next}. The algorithm will not stop unless an enclosure of the solution has been obtained.

The strategy of the step size control is to raise the step size if less than IT_{normal} Picard iterations are used before succeeding the proof and to lower the step size if more iterations were used. This way the step size is changed according to how easy we obtain the condition Eq. (6.9). It is normal to choose such a step size strategy since it is the the Picard operator which proves the existence of the solution in each integration step.

6.3 Enclosing solutions of initial value problems

Consider Eq. (6.1) with the initial value $y(t_0) = y_0 \in D$. Assume that a unique solution $y(t)$ exists for $t \in [t_0, t_N]$ where $t_N > t_0$ [8]. Consider a discretization of the solution by $y_j = y(t_j)$, $j = 0, \ldots, N$, for the discrete values of the independent variable $t_0 < t_1 < \cdots < t_N$. Since $y \in C^{k+1}([t_0, t_N], D)$, we can use Thm. 1 to obtain a discrete map of the same form as the map in Eq. (4.4),

$$y_{j+1} = \varphi_j(y_j) = \tilde{\varphi}_j(y_j) + \varepsilon_j(y_j), \text{ for } 0 \le j < N, \tag{6.10}$$

where

$$\tilde{\varphi}_j(y_j) = y_j + \sum_{i=1}^{k_j} (y_j)_i h_j^i, \tag{6.11a}$$

$$\varepsilon_j(y_j) = h_j^{k_j+1}(k_j+1) \int_0^1 y^{[k_j+1]}(\theta t_{j+1} + (1-\theta)t_j)(1-\theta)^{k_j}\, d\theta, \tag{6.11b}$$

where $h_j = t_{j+1} - t_j$ is the step size in the jth step, and k_j is the degree of the Taylor expansion in the jth step, where $k_j \le k$. The function $\tilde{\varphi}_j(y_j)$ is differentiable with respect to y_j. We need an enclosure of the remainder term $\varepsilon_j(y_j)$ in order to apply the mean value enclosures as described in Section 4. Since

$$y^{[k_j+1]}(t) = \frac{f^{[k_j]}(y(t))}{k_j+1}, \tag{6.12}$$

[8] Algorithm 6.6 will stall or break down if this is not the case.

<u>Input:</u>

$h, [y_j], IT_{normal}, IT_{max}, e.$

<u>Validation:</u>

$it = 0,$

proved $= false,$

$[\tilde{y}_j^1] = [y_j],$

do

$\quad [\tilde{y}_j^0] = (1+e)[\tilde{y}_j^1] - e[\tilde{y}_j^1],$

$\quad [\tilde{y}_j^1] = [y_j] + F([\tilde{y}_j^0])[0,h],$

$\quad if\ ([\tilde{y}_j^1] \subseteq [\tilde{y}_j^0])\ \ then$ proved $= true,$

$\quad it = it + 1,$

$\quad if\ (it = IT_{normal})\ \ then\ h_{next} = \max\limits_h : [y_j] + F([\tilde{y}_j^0])[0,h] \subseteq [\tilde{y}_j^0],$

$while\ (\ NOT(\text{proved})\ AND\ it < IT_{max}),$

$if\ \ (it < IT_{normal})\ \ then\ h_{next} = \max\limits_h : [y_j] + F([\tilde{y}_j^0])[0,h] \subseteq [\tilde{y}_j^0],$

$if\ \ NOT(\text{proved})\ \ then\ h = h_{next},\ goto$ <u>Validation</u>,

<u>Output:</u>

$[\tilde{y}_j^1], h, h_{next}.$

Algorithm 6.6: Algorithm for proving existence and uniqueness and for obtaining a rough enclosure of the solution of the ODE.

we have that

$$\varepsilon_j(y_j) = h_j^{k_j+1} \int_0^1 f^{[k_j]}(y(\theta t_{j+1} + (1-\theta)t_j))(1-\theta)^{k_j}\, d\theta. \tag{6.13}$$

In order to obtain a value for $\varepsilon_j(y_j)$, we have to know the solution $y(t)$ in the interval $[t_j, t_{j+1}]$. Assume that an interval vector $[\tilde{y}_j]$, enclosing the solution in this interval is known. Now we have

$$\begin{aligned}
\varepsilon_j(y_j) &\in h_j^{k_j+1} \int_0^1 F^{[k_j]}([\tilde{y}_j])(1-\theta)^{k_j}\, d\theta \\
&= h_j^{k_j+1} F^{[k_j]}([\tilde{y}_j]) \int_0^1 (1-\theta)^{k_j}\, d\theta \\
&= h_j^{k_j+1} \frac{F^{[k_j]}([\tilde{y}_j])}{k_j+1} \\
&= h_j^{k_j+1} ([\tilde{y}_j])_{k_j+1},
\end{aligned} \tag{6.14}$$

where $F^{[k_j]}$ is an interval extension of $f^{[k_j]}$, and $([\tilde{y}_j])_{k_j+1}$ is an enclosure the k_j+1th Taylor coefficient of the solution in the interval $[t_j, t_{j+1}]$. An enclosure of the remainder term can be obtained by expanding the k_j+1th Taylor coefficient of the solution in the interval vector $[\tilde{y}_j] = [\tilde{y}_j^1]$ obtained as the rough enclosure from Algorithm 6.6.

We are now capable of applying the mean value enclosures to enclose the solution of the initial value problem Eq. (6.1) with $y(t_0) \in [y_0]$ by using the following strategy for obtaining enclosures of $[\hat{y}_{j+1}]$ and $[S_j] = \tilde{\Phi}'([y_j])$, which is needed in Algorithms 4.1 and 4.2.

- Use Algorithm 6.6 to find a $h_j > 0$ and $[\tilde{y}_j^1]$ so that $y(t) \in [\tilde{y}_j^1]$, for $t \in t_j + [0, h_j]$.
- Find the degree of expansion k_j so that $\mathrm{w}([\hat{y}_{j+1,k_j}]) \leq \mathrm{w}([\hat{y}_{j+1,k_j+1}])$, where

$$[\hat{y}_{j+1,k_j}] = \tilde{\varphi}_j(\hat{y}_j) + [z_{j+1,k_j}], \tag{6.15}$$

and

$$\tilde{\varphi}_j(\hat{y}_j) = \hat{y}_j + \sum_{i=1}^{k_j} (\hat{y}_j)_i h_j^i, \tag{6.16a}$$

$$[z_{j+1,k_j}] = h_j^{k_j+1}([\tilde{y}_j^1])_{k_j+1}. \tag{6.16b}$$

The degree of the Taylor expansion k_j is by this choice the smallest integer for which the width of $[\hat{y}_{j+1,k_j}]$, i.e., the local error (truncation+rounding), has a minimum (hopefully global). In practice we will also have the restriction $k_j \leq k_{max}$ for some maximum allowed degree of expansion k_{max}. By differentiating $\tilde{\varphi}_j(y_j)$ in Eq. (6.11a) with respect to y_j and forming the interval extension, we obtain the interval matrix

$$[S_j] = \tilde{\Phi}'([y_j]) = I + \sum_{i=1}^{k_j} \frac{\partial y^{[i]}}{\partial y_j}([y_j])h_j^i, \tag{6.17}$$

where $\frac{\partial y^{[i]}}{\partial y_j}([y_j])$ is the ith Taylor coefficient differentiated with respect to the point of expansion y_j, evaluated using interval arithmetic in $[y_j]$.

- We can enclose the partial derivatives D_j in Eq. (4.16), where $\varphi^j = \varphi_j \circ \varphi_{j-1} \circ \cdots \circ \varphi_0$, by providing an enclosure of the partial derivatives of the error term $\frac{\partial \varepsilon_j}{\partial y_j}$,

$$\frac{\partial \varepsilon_j}{\partial y}(y_j) \in \frac{\partial y^{[k_j+1]}}{\partial y_j}([\tilde{y}_j])h_j^{k_j+1}. \tag{6.18}$$

We have the relation $D_j = D(t_j)$, where the function D is the solution to the variational equation

$$D'(t) = \frac{\partial f}{\partial y}(y(t))D(t), \quad D(t_0) = I, \tag{6.19}$$

6.3.1 An automatic differentiation interval ordinary differential equation solver (ADIODES)

A package for solving ordinary differential equations has been developed. The package uses the interval packages BIAS/PROFIL, defining the `INTERVAL` type, and the packages FADBAD/TADIFF for performing the differentiation, using the types `TINTERVAL` and `TFINTERVAL`. To use ADIODES for solving Eq. (6.1), the function f is specified as a C++ function, an enclosure of the initial value $y_0 \in [y_0]$ is specified as an interval vector, and also the interval $[t_0, t_N]$ in which we want to solve the equation is specified. Some additional parameters, such as the initial step size h, the maximum allowed degree of expansion k_{max}, and the parameters IT_{normal}, IT_{max}, and e used in Algorithm 6.6 can also be specified.

For enclosing the solution, the two enclosure methods described in Section 4, with or without an enclosure of the solution to the variational equation, can be used. The output of the algorithm is an enclosure of

$$y(t_N; y_0) = \varphi^N(y_0), \quad \text{where} \quad (t_N - t_0) = \sum_{i=0}^{N-1} h_i \tag{6.20}$$

and (if desired) its partial derivatives

$$\frac{\partial y(t_N; y)}{\partial y}(y_0) = \frac{\partial \varphi^N}{\partial y}(y_0). \tag{6.21}$$

6.4 Integration example (the Brusselator)

Consider the ordinary differential equation, called the Brusselator [27]

$$\begin{aligned} x' &= A + x(xy - B - 1), \\ y' &= x(B - xy), \end{aligned} \tag{6.22}$$

which is a model of a chemical reaction with variables x and y representing two chemical intermediates. For the values $A = \frac{2}{5}$ and $B = \frac{6}{5}$ and using the initial values $x(0) \in [.3074, .3081]$ and $y(0) = 3$, the solution has been enclosed using the extended mean value enclosure for discrete values of $t \in [0, 50]$. Figure 6.8 shows the enclosures of $x(t_j)$ and $y(t_j)$, the width of the enclosures $\max(\mathrm{w}([x(t_j)]), \mathrm{w}([y(t_j)]))$, the step sizes used h_j, and the orders of the Taylor expansions used k_j, all as functions of the discrete values of the independent variable t_j.

In Figure 6.8 it can be seen that the width of the enclosure of the solution is well behaved for $t \in [0, 40]$, but for $t \in [40, 50]$ the width begins to grow faster. The step sizes used lies in the interval $[0.1, 0.3]$, and the order of the Taylor expansions lies in the interval $[11, 21]$. We will later show that the initial values used in the integration overlaps a periodic solution of Eq. (6.22), i.e., a solution which repeats itself infinitely. The overlap in the initial value will cause all enclosures of the solution to overlap when integrating the system. This property of the solution is more obvious in Figure 6.9, where the last revolution of the encapsulated solution ($t \in [30, 50]$) is shown in phase space. The boxes in the figure shows the "rotating rectangle" enclosure used in Algorithm 4.2. From the figure it seems like the uncertainty is smeared out along the periodic solution. If the integration process was continued for $t > 50$, the width of the enclosure would get too large, causing Algorithm 6.6 to stall, i.e., $h_j \to 0$.

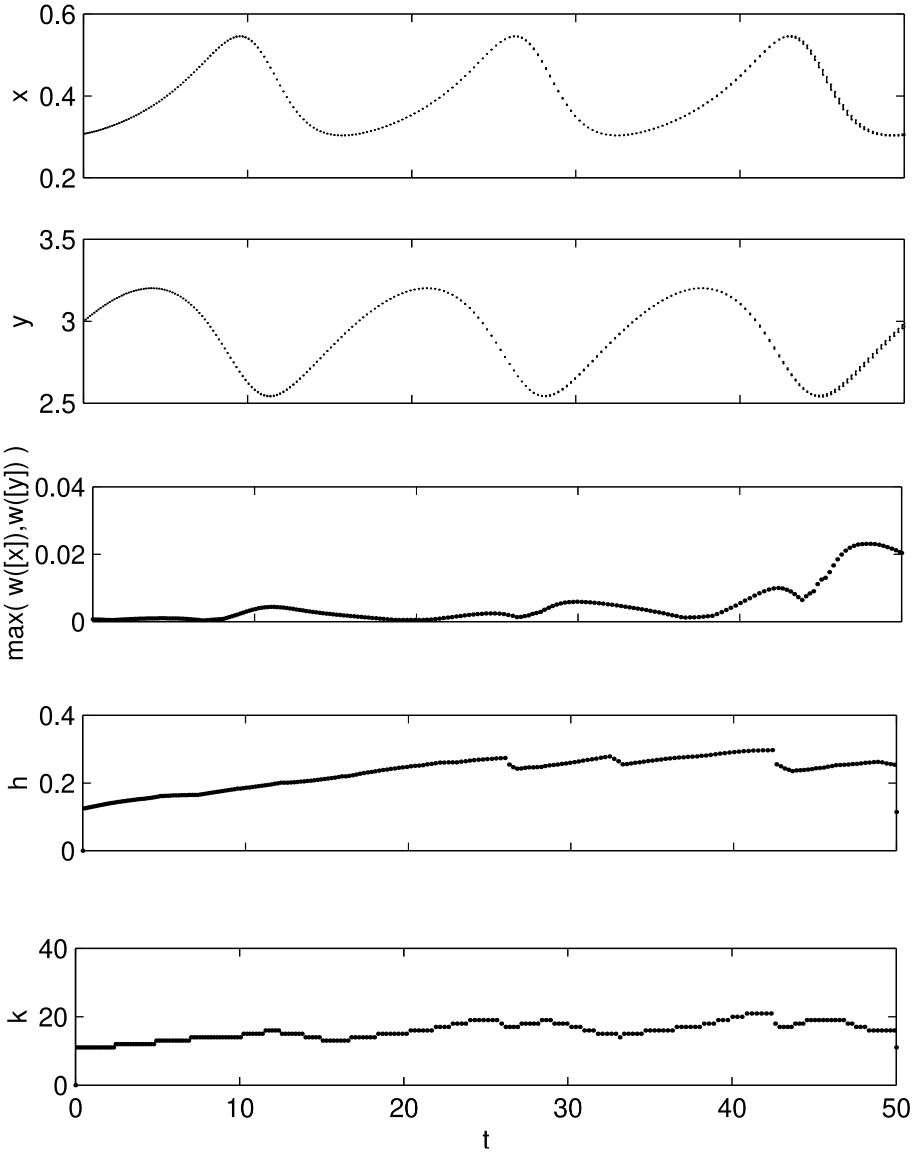


Figure 6.8: Solving the unforced Brusselator: The graphs shows the enclosure of the solution, the width of the enclosure, the step sizes used and the orders of the Taylor expansions used.

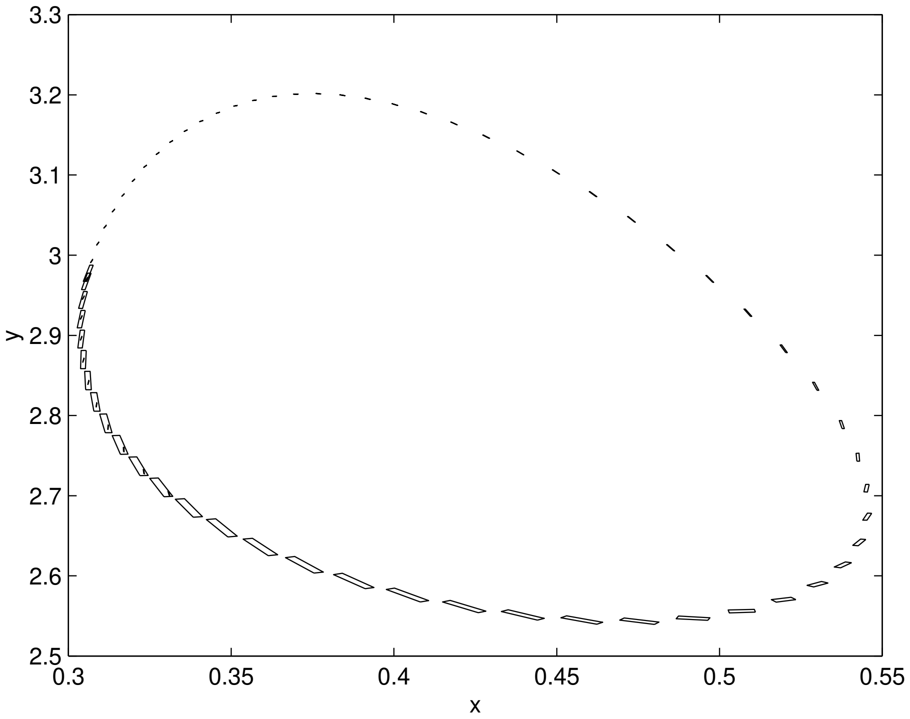


Figure 6.9: Solution of the unforced Brusselator: The graph shows the enclosure of the solution for discrete values of $t \in [30, 50]$ in phase space.

7 Computer-assisted proofs in dynamical systems

The study of ordinary differential equations is an important field of dynamical systems. The behaviour of real-life systems in physics is often described by differential equations, but since these equations often have solutions which are impossible to represent in closed form, people use numerical methods to obtain approximations of the solutions. Since the use of approximate solutions can only lead to a qualitative investigation of systems, it is difficult to prove whether some property of some approximate solution also is a property of the exact solution or if it is an artifact introduced by the approximation method. Since it is possible using the interval methods described in this report to obtain verified enclosures of the solutions of ordinary differential equations, it is also possible to prove properties of the exact solutions. We will in this section investigate some systems, which are considered to be non-trivial, and prove existence and uniqueness of some specific solutions of the systems.

7.1 A note on the representation of intervals used in this report

A general problem when publishing results obtained using a computer is that the computer's representation of floating point numbers is not the same as we humans use. Humans represent numbers in base 10, while most computers represent numbers in base 2. This difference in representation is normally not considered as a problem since conversion programs, convert numbers between the two bases. Unfortunately these programs often commit roundoff errors so that our interpretation of a computer-assisted proof may be wrong. In most cases we are satisfied knowing that the computer has proven a result for some intervals of which we do not know the exact values, but if we want to publish these intervals, then we have to publish the exact values down to the last digit.

Fortunately the integer numbers are normally exact representable on computers, if we stick to small values. In most cases the numbers used as parameters and initial values in some programs are not exactly representable by the computer, and intervals are used instead to bound the correct values. To generate the correct bounds, we use integers and interval arithmetics. For example, to enclose the number $\frac{1}{3}$ we perform the division in rounded interval arithmetic to obtain a lower and an upper bound of the value; for a correct enclosure of the interval $[0.30^{81}_{74}]$ we use the outward rounded result $[30^{81}_{74}]/10^4$.

In some cases, using outward rounding is not enough. If we want to prove existence and uniqueness of some solution in an interval vector $[x]$ which is not exact representable by the computer, the computer-assisted proof will then be valid for the rounded value of the interval vector instead of the interval vector itself: If outward rounding were used to enclose the interval vector, the solution could lie outside $[x]$. If inward rounding were used, another solution could exist between $[x]$ and the inward rounded interval vector. To deal with this problem, both roundings have to be applied, so we have $[\hat{x}] \subseteq [x] \subseteq [\check{x}]$, and the proof has to be applied twice, once for the outward rounded interval vector $[\hat{x}]$ and once for the inwards rounded interval vector $[\check{x}]$. Obviously we only have to prove uniqueness for the outwards rounded interval vector and existence for the inwards rounded interval vector to obtain existence and uniqueness of the solution in the original interval vector $[x]$.

7.2 Periodic solutions of autonomous systems

Consider the equation

$$y' = f(y), \quad y(0) = y(T), \quad \text{for } T > 0. \tag{7.1}$$

where $f \in C^k(D, \mathbb{R}^n)$, $D \subseteq \mathbb{R}^n$ is an open set. Solutions of this equation are called periodic solutions of period T. Furthermore, if $y(t) \neq y(0)$ for $t \in (0, T)$, then T is called the prime period. Obviously if T is a period of the periodic solution, then $2T, 3T, 4T, \ldots$ are also periods of the same solution, and we have $y(0) = y(T) = y(2T) = \cdots$.

Since we also want to find an enclosure of the period T when solving Eq. (7.1), we transform it into an equation

$$\left\{ \begin{array}{lcl} \tilde{y}' & = & pf(\tilde{y}) \\ p' & = & 0 \end{array} \right\}, \ \tilde{y}(0) = \tilde{y}(1), \tag{7.2}$$

introducing a state variable p. This equation is just a rescaling w.r.t. the independent variable t compared to Eq. (7.1), and solutions of this equation are scaled solutions of Eq. (7.1), so that $y(Tt) = \tilde{y}(t)$ where $T = p$.

Let $\tilde{y}(t; y_0, p_0)$ be the solution of the ordinary differential equation in Eq. (7.2) with the initial value (y_0, p_0). If y_0 is a point on the periodic solution of Eq. (7.1) with period $T = p_0$, then we have

$$\tilde{y}(1; y_0, p_0) - y_0 = 0. \tag{7.3}$$

A solution to this equation is not isolated since other points on the periodic orbit exist arbitrarily close to y_0, but by letting one of the components in y_0 be fixed in Eq. (7.3), it is possible to make the solution isolated. Let us w.l.o.g. fix the nth component $y_{0,n}$ of y_0. Defining $x = (y_{0,1}, y_{0,2}, \dots, y_{0,n-1}, p_0)$, we obtain the system of equations

$$\left\{\begin{array}{ccc} \tilde{y}_1(1;x,y_{0,n}) & - & x_1 \\ \tilde{y}_2(1;x,y_{0,n}) & - & x_2 \\ & \cdots & \\ \tilde{y}_{n-1}(1;x,y_{0,n}) & - & x_{n-1} \\ \tilde{y}_n(1;x,y_{0,n}) & - & y_{0,n} \end{array}\right\} = 0. \tag{7.4}$$

which can be solved for x using an interval Newton or interval Krawczyk method. In the following sections, the interval Newton method will be used to solve Eq. (7.4), proving existence and uniqueness of periodic solutions. Enclosures of $\tilde{y}(1;\cdot)$ and its partial derivatives are obtained by using the extended mean value enclosure implemented in ADIODES. In the following computer-assisted proofs, we wish to prove uniqueness of solutions in intervals which are as wide as possible.

7.2.1 The Brusselator

Using the interval Newton method, a periodic solution (x,y) of period $T \in [16.75_0^7]$ of the Brusselator Eq. (6.22), with the parameters $A = \frac{2}{5}$ and $B = \frac{6}{5}$, has been proven to exist and be unique for $x(0) \in [0.30_{74}^{81}]$ and a fixed $y(0) = 3$. By continuing the Newton iterations, it was possible to determine $x(0)$ and T to an uncertainty of order 10^{-10}; see Figure 7.10.

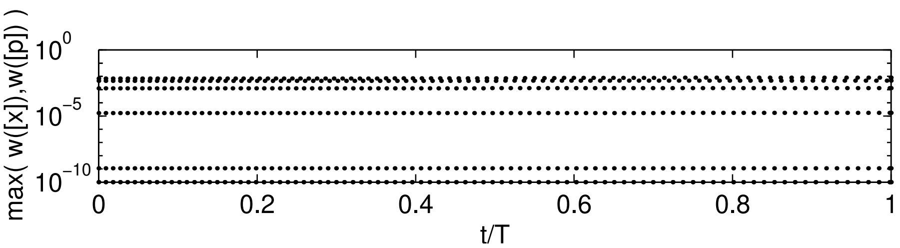


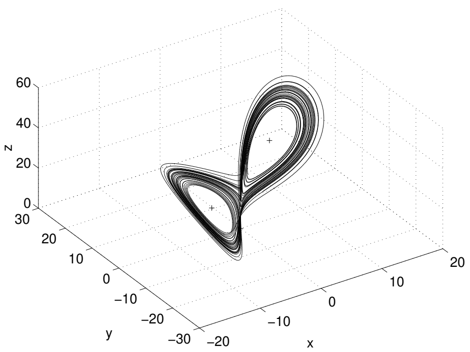
Figure 7.10: The widths of the enclosures of the periodic solution of the Brusselator during the Newton iterations.

7.2.2 The Lorenz system

The Lorenz equations are given by [36]

$$\begin{aligned} x' &= \sigma(y-x), \\ y' &= rx-y-xz, \\ z' &= xy-bz, \end{aligned} \tag{7.5}$$

where b, r, and σ are positive parameters. For a detailed analysis of these equations see [52]. The equations have a stationary point at the origin. For $r > 1$ the points $(\xi, \xi, r-1)$ and $(-\xi, -\xi, r-1)$, where $\xi = \sqrt{b(r-1)}$ are stationary points. Using the parameters $b = \frac{8}{3}, r = 28$, and $\sigma = 6$, solutions of Eq. (7.5) seem to behave chaoticly. In Figure 7.11 the result of a floating-point ODE solver, solving the system with the initial values $x(0) = 4.1879, y(0) = 6.7601$ and $z(0) = 16.1091$, is shown. From the figure, it is seen that the solution lies on some object in $\mathbb{R}^3$. This object is called a strange attractor since it is a strange looking set and because solutions of the system converges to it.



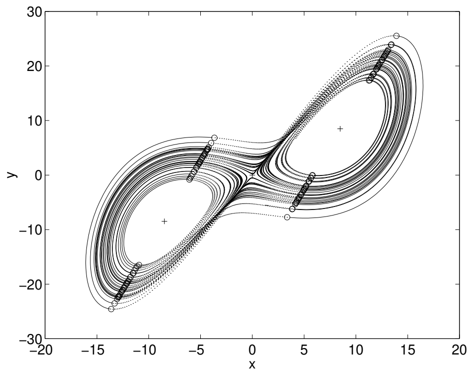


Figure 7.11: Floating-point solution of Eq. (7.5) using $b = \frac{8}{3}, r = 28$ and $\sigma = 6$. The left graph shows the result of the computations in a three-dimensional phase plot. In the right graph, a projection of the solution to the (x, y) plane is seen and the intersection of the solution with the plane $z = 27$ has been marked with small circles. The stationary points have been marked with '+' on both figures.

In the right graph of Figure 7.11, the intersection of the floating point based solution and the plane $z = r - 1 = 27$ is marked with small circles. The figure indicates that using a fixed $z(0) = 27$ when looking for periodic solutions is a good choice, and the locations of the circles mark the areas where to find them.

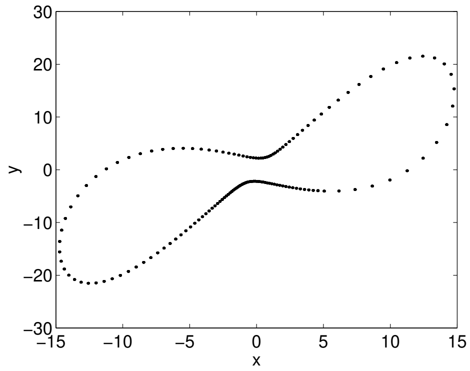
Six different periodic solutions of Eq. (7.5) were proven, by the interval Newton method, to exist and to be unique for interval values of $T, x(0), y(0)$, and a fixed value $z(0) = 27$. Figures 7.12-7.13 show the periodic solutions found, and the corresponding intervals in which existence and uniqueness was obtained. Since the Lorenz equations have a natural symmetry $(x, y, z) \to (-x, -y, z)$, four other solutions, different from the previously found, exist with initial values $T, -x(0), -y(0)$, and $z(0) = 27$ for values of $T, x(0)$ and $y(0)$ corresponding to the asymmetric solutions listed in Figures 7.12-7.13, i.e., the solutions 2:1, 3:1, 4:1, 3:2. By continuing the interval Newton iterations, it was possible to compute the values of $T, x(0)$ and $y(0)$ to an accuracy of order 10^{-10} for solution 1:1, which was the best accuracy obtained, and to order $3 \cdot 10^{-9}$ for solution 3:2, which was the lowest accuracy obtained. The interval Newton method was unable to prove solutions with higher periodicity than solution 3:2, since the widths of the interval Newton operations becomes larger than the widths of the initial values when attempting to prove existence.

In an article by Brian A. Coomes, Hüseyin Koçak and Kenneth J. Palmer [13], two periodic solutions of the Lorenz equations have been proven to exist by using a method called periodic shadowing. In their first proof they use parameters $b = \frac{8}{3}, r = 100.5$, and $\sigma = 10$ and prove that a stable periodic solution exist for

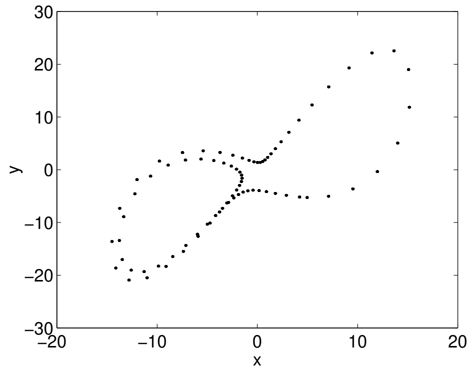
$$\begin{array}{rcl} x(0) & \in & 1.7589044527748 27471 + [-\varepsilon, \varepsilon], \\ y(0) & \in & -4.4809108734587 81704 + [-\varepsilon, \varepsilon], \\ z(0) & \in & 80.992671614836 50640 + [-\varepsilon, \varepsilon], \end{array} \tag{7.6}$$

with a shadowing distance $\varepsilon \leq 3.2992205444891 39846 \cdot 10^{-12}$ and with a period $T \approx 1.0962388136$. To verify this result, the interval Newton method was applied. We proved existence and uniqueness of a periodic solution of period $T \in [1.09623^{90}_{87}]$ with $x(0) \in [1.758904^{5}_{3}]$, $y(0) \in [-4.48091^{07}_{10}]$ and a fixed interval value $z(0) \in [80.992671161^{5}_{4}]$. By continuing the Newton iterations with a fixed value $z(0) \approx 80.99267161483650640$ the accuracy of T, $x(0)$, and $y(0)$ could be determined to the order $4 \cdot 10^{-11}$, which is less accurate than the accuracy obtained by the periodic shadowing method. Since the enclosures obtained by the interval Newton method overlaps with the intervals obtained by periodic shadowing it was not possible to prove or disprove the result obtained from the periodic shadowing method.

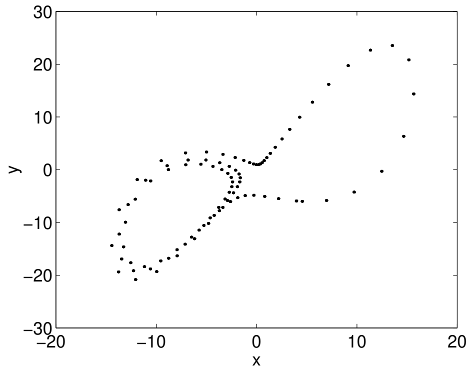
In the second proof of the article [13], an unstable periodic solution of the Lorenz equations, using the parameters $b = \frac{8}{3}, r = 28$ and $\sigma = 10$, has been



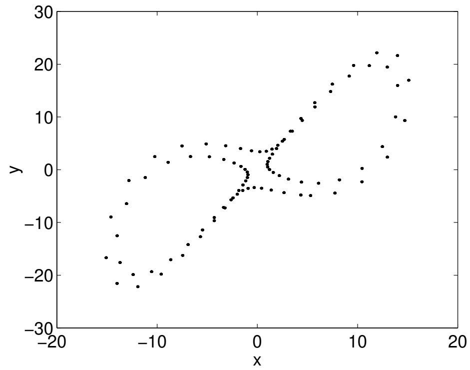
(a) Solution 1:1, $T \in [1.7516^{9}_{8}]$, $x(0) \in [4.5737^{9}_{7}]$, $y(0) \in [-3.978^{19}_{21}]$, and $z(0) = 27$.



(b) Solution 2:1, $T \in [2.594277^{7}_{6}]$, $x(0) \in [4.194260^{4}_{3}]$, $y(0) \in [-5.173485^{7}_{8}]$, and $z(0) = 27$.

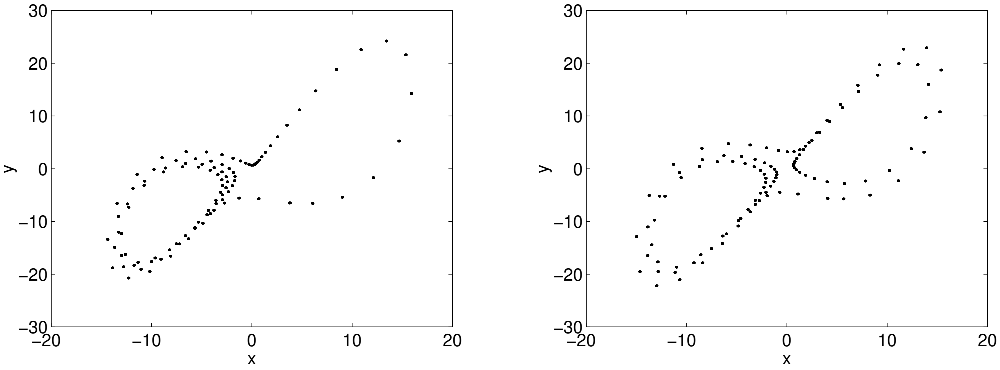


(c) Solution 3:1, $T \in [3.405937^{8}_{7}]$, $x(0) \in [3.952332^{3}_{2}]$, $y(0) \in [-5.928351^{4}_{5}]$, and $z(0) = 27$.



(d) Solution 2:2, $T \in [3.469322^{2}_{0}]$, $x(0) \in [4.312692^{6}_{0}]$, $y(0) \in [-4.80296^{77}_{83}]$, and $z(0) = 27$.

Figure 7.12: Periodic solutions of the Lorenz equations Eq. (7.5) using parameters $b = 8/3, r = 28$ and $\sigma = 6$. All solutions were proven to exist and be unique for the specified intervals by using the interval Newton method.



(a) Solution 4:1, $T \in [4.2012773^{90}_{89}]$, $x(0) \in [3.7720541^{62}_{59}]$, $y(0) \in [-6.48667584^{3}_{7}]$, and $z(0) = 27$.

(b) Solution 3:2, $T \in [4.300104^{60}_{59}]$, $x(0) \in [4.0501795^{8}_{6}]$, $y(0) \in [-5.6242059^{0}_{3}]$, and $z(0) = 27$.

Figure 7.13: Periodic solutions of the Lorenz equations Eq. (7.5) using parameters $b = \frac{8}{3}, r = 28$ and $\sigma = 6$. All solutions were proven to exist and be unique for the specified intervals by using the interval Newton method.

proven to exist for

$$\begin{aligned} x(0) &\in -12.78619065852397651 + [-\varepsilon, \varepsilon], \\ y(0) &\in -19.36418793711800464 + [-\varepsilon, \varepsilon], \\ z(0) &\in 24 + [-\varepsilon, \varepsilon], \end{aligned} \tag{7.7}$$

with a shadowing distance $\varepsilon \leq 1.799087099871078045 \cdot 10^{-12}$ and with a period $T \approx 1.558652210$. Using the interval Newton method a periodic solution has been proved to exist and be unique for $T \in [1.55865^{3}_{2}]$ and $x(0) \in [-12.786^{18}_{20}]$, $y(0) \in [-19.364^{18}_{20}]$ with a fixed interval value $z(0) \in 24 + 10^{-6}[-1, 1]$. By continuing the Newton iterations with a fixed value $z(0) = 24$ the accuracy of T, $x(0)$ and $y(0)$ could be determined to the order $4 \cdot 10^{-11}$. Also in this case the result obtained by the interval Newton method overlaps with the result obtained by using the periodic shadowing method and it was not possible to prove or disprove the result obtained from the periodic shadowing method.

7.2.3 The Van der Pol system

Consider the Van der Pol equation

$$u'' + \varepsilon(u^2 - 1)u' + u = 0. \tag{7.8}$$

This equation becomes stiff in a region of phase space for increasing values of ε, and ordinary differential equation solvers intended for non-stiff problems tend to choose small integration step sizes in this region. Furthermore the type of the stability changes from stiff to non-stiff and back when integrating the system along the periodic orbit for which we will prove the existence.

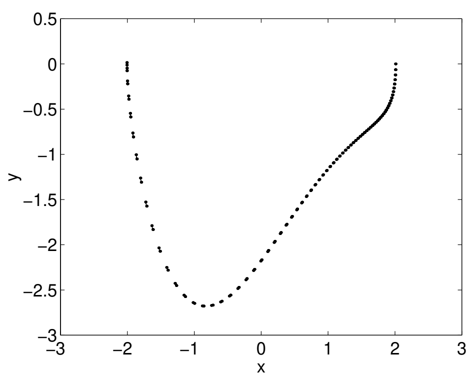
The change of variables $x = u$ and $y = x'$ yields the system

$$\begin{array}{rcl} x' &=& y, \\ y' &=& \varepsilon(1 - x^2)y - x, \end{array} \tag{7.9}$$

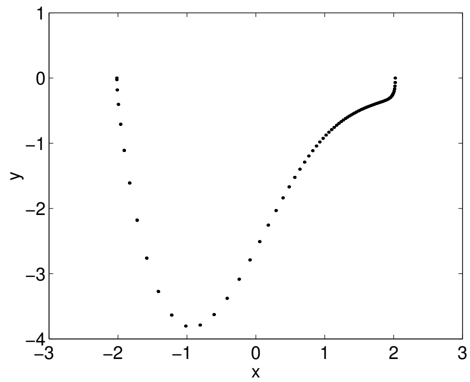
which is the form needed by ADIODES. Since this equation has the natural symmetry $(x, y) \to (-x, -y)$ we have that a solution of Eq. (7.9) for which $x(0) = x(T/2)$ and $y(0) = 0, y(T/2) = 0$ for some $T > 0$ is a periodic solution with period T. Using this property, it is possible to modify Eq. (7.4) so that only the piece of the periodic solution which lies in the area $y \leq 0$ has to be encapsulated for proving the existence.

In Figure 7.14, some periodic solutions for different values of the ε parameter, are shown. Existence and uniqueness has been proven for all solutions by using the interval Newton method.

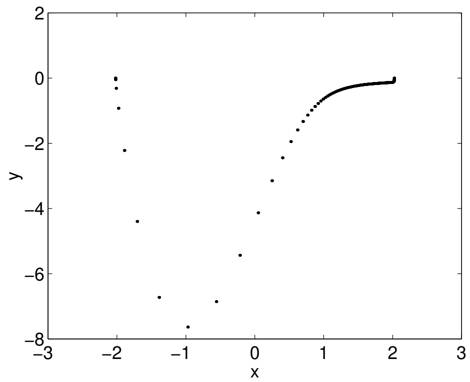
When increasing the stiffness parameter ε, the number of discretization points increased as expected. The accuracy of the enclosure is almost constant in the stiff area, but when crossing over into the non-stiff area (when the distance in between the points increases) some accuracy is lost. In the last case, when $\varepsilon = 10$, too much accuracy was lost, and the interval Newton operator became unable to prove existence. By choosing a smaller epsilon-inflation parameter $e = 0.001$ instead of the default value 0.01 in Algorithm 6.6, it was possible to force ADIODES to choose smaller step sizes in the non-stiff area and the accuracy of the enclosure became better. By doing this, it became possible to prove the existence and uniqueness of the periodic solution when $\varepsilon = 10$. In Figure 7.15 the width of the enclosure, the step sizes, and the orders of the Taylor expansions are shown, when encapsulating the periodic solution for $\varepsilon = 10$ using the initial values listed in Figure 14(d). From this figure, it is seen that the width of the enclosure blows up for a while but then contracts again just before the end of the integration.



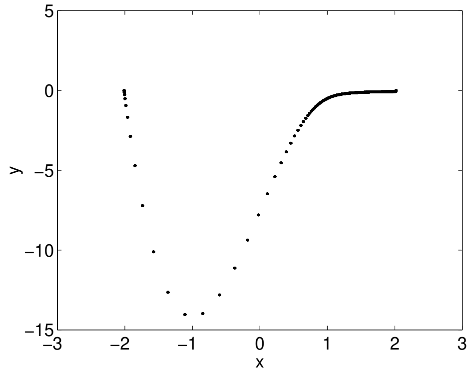
(a) $\varepsilon = 1$, $x(0) \in [2.0^{10}_{07}]$, $y(0) = 0$, $\frac{T}{2} \in [3.33^{4}_{0}]$.



(b) $\varepsilon = 2$, $x(0) \in [2.0^{200}_{198}]$, $y(0) = 0$, $\frac{T}{2} \in [3.81^{50}_{49}]$.



(c) $\varepsilon = 5$, $x(0) \in [2.02150^{9}_{7}]$, $y(0) = 0$, $\frac{T}{2} \in [5.80611^{6}_{5}]$.



(d) $\varepsilon = 10$, $x(0) \in [2.01428^{6}_{5}]$, $y(0) = 0$, $\frac{T}{2} \in [9.53918^{7}_{3}]$.

Figure 7.14: Periodic solutions of the Van der Pol equation Eq. (7.9). All solutions were proven to exist and be unique for the specified intervals by using the interval Newton method.

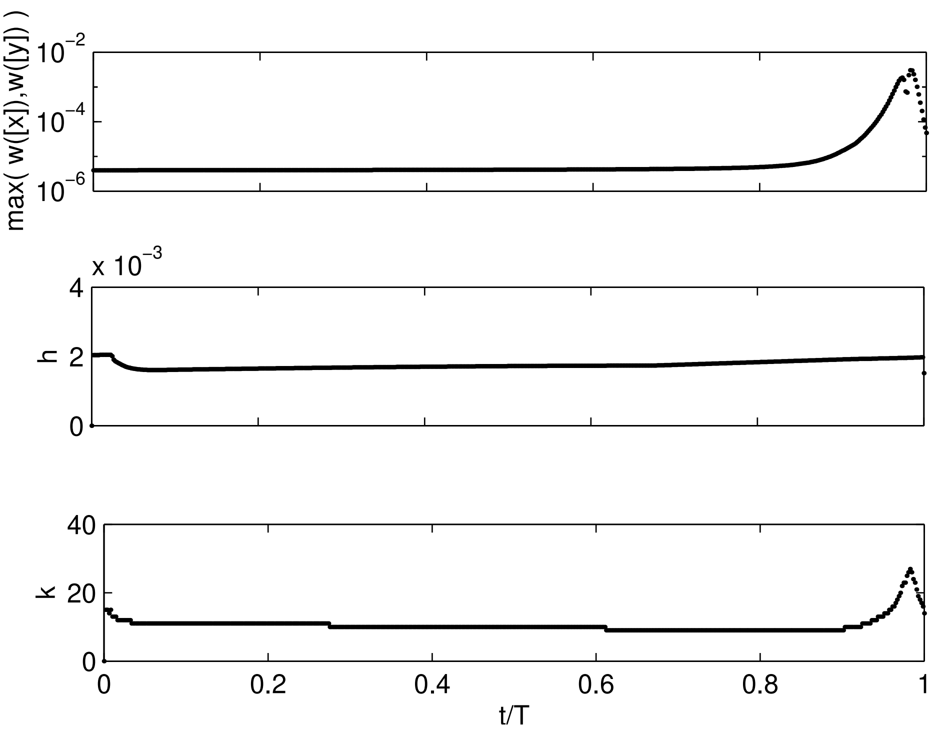


Figure 7.15: Solving the Van der Pol equation with $\varepsilon = 10$: The graphs shows the width of the enclosure, the step sizes used, and the orders of the Taylor expansions used.

7.3 Periodic solutions of non-autonomous systems

Consider the equation

$$y' = f(y,t), \;\; y(0) = y(pT), \tag{7.10}$$

where the function $f \in C^k(D \times \mathbb{R}, \mathbb{R}^n)$ is periodic in t with the period $T > 0$, $D \subseteq \mathbb{R}$ is an open set, and p is a positive integer. Solutions of this equations are called periodic solutions of period p. Furthermore, if $y(jT) \neq y(0)$ for $j = 1, \ldots, p-1$ then p is called the prime period. Obviously, if p is a period of the periodic solution, then also $2p, 3p, 4p, \ldots$ are periods on the same solution, and we have $y(0) = y(pT) = y(2pT) = \cdots$.

We can rewrite Eq. (7.10) into an equivalent autonomous form

$$\left\{ \begin{array}{rcl} \tilde{y}' & = & pTf(\tilde{y}, pTt) \\ t' & = & 1 \end{array} \right\}, \ \tilde{y}(0) = \tilde{y}(1), t(0) = 0. \tag{7.11}$$

The solutions of this equation are scaled solutions of Eq. (7.10) so that $y(pTt) = \tilde{y}(t)$.

Let $\tilde{y}(t; y_0)$ be the solution of the ordinary differential equation in Eq. (7.11) with the initial value y_0 when using a fixed positive integer p. If y_0 is a point on a periodic solution of Eq. (7.10) with the period p, then we have

$$\tilde{y}(1; y_0) - y_0 = 0, \tag{7.12}$$

which can be solved for y_0 by using the interval Newton method or the interval Krawczyk method.

7.3.1 The forced Brusselator

The forced Brusselator is given by the equations [27]

$$\begin{array}{rcl} x' & = & A + x(xy - B - 1) + a\cos(\omega t), \\ y' & = & x(B - xy), \end{array} \tag{7.13}$$

where A, B, a, and ω are parameters. The function given by the right hand side is periodic in t with the period $T = \frac{2\pi}{\omega}$. Using the parameters $A = \frac{2}{5}$, $B = \frac{6}{5}$, $a = 0.03$, and $\omega = \frac{\pi}{4}$ is was possible to use the interval Newton method to prove existence and uniqueness of an unstable periodic solution of period $p = 1$, for the initial values $x(0) \in [0.38^{9}_{7}]$ and $y(0) \in [3.05^{2}_{1}]$ and a stable periodic solution with period $p = 2$, for the initial values $x(0) \in [0.385^{1}_{0}]$ and $y(0) \in [3.25^{20}_{14}]$. In Figure 7.16, the computed solutions of Eq. (7.13) is shown, when using the previously obtained initial values. The initial values of both solutions have been determined to an accuracy of the order 10^{-13} by continuing the interval Newton iterations.

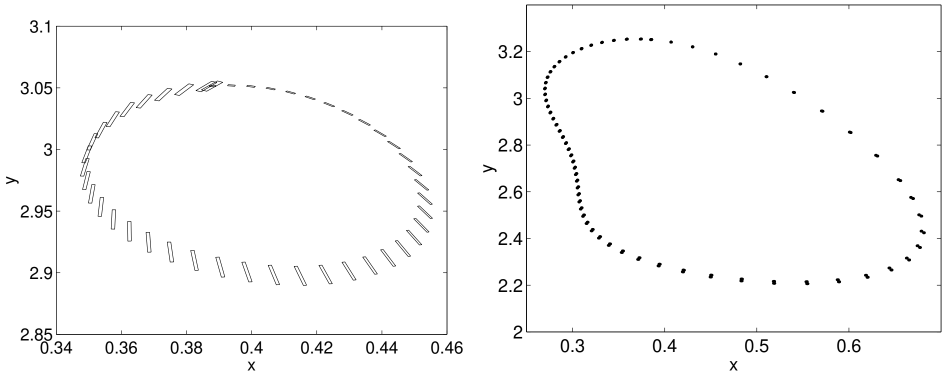


Figure 7.16: Periodic solution of the forced Brusselator, using the parameters $A = \frac{2}{5}$, $B = \frac{6}{5}$, $a = .03$, and $\omega = \frac{\pi}{4}$. The left figure shows the period $p = 1$ solution, and the right figure shows the period $p = 2$ solution.

7.4 Solutions of boundary value problems

Also solutions of boundary value problems can be proved to exist with the aid of interval methods [35]. Consider the problem

$$u''(t) + a^2 \sin(u(t)) - \sin(t\pi) = 0, \quad u(0) = u(1) = 0, \tag{7.14}$$

where $a \in \mathbb{R}$. Using the change of variables $x = u$ and $y = x'$ and adding t as a state variable, we obtain the equivalent problem

$$\left\{ \begin{array}{lcl} x' & = & y \\ y' & = & \sin(t\pi) - a^2 \sin(x(t)) \\ t' & = & 1 \end{array} \right\}, x(0) = x(1) = 0, t(0) = 0. \tag{7.15}$$

Since the values of $x(0)$ and $t(0)$ are known, we can solve the equation by finding an initial value y_0, so that the ordinary differential equation in Eq. (7.15) with the initial value $x(0) = 0, y(0) = y_0$ and $t(0) = 0$ yields a solution for which $x(1) = 0$. Such an initial value can be found by solving

$$x(1; y_0) = 0, \tag{7.16}$$

where $x(t; y_0)$ is the $x-$component of the solution of the ordinary differential equation in Eq. (7.15) with the initial value $x(0) = 0, y(0) = y_0$ and $t(0) = 0$. Equation Eq. (7.16) can be solved by using the interval Newton method or the interval Krawczyk method. We used the interval Newton method to prove existence and uniqueness of a solution to Eq. (7.15), using $a = 1$, for $y_0 \in [-0.3_{6}^{5}]$. See Figure 7.17.

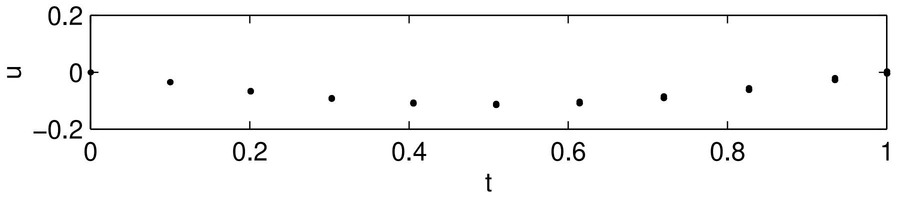


Figure 7.17: Enclosure of the solution u to the boundary value problem Eq. (7.14) for discrete values of t.

8 Solving integral equations

We will consider the problem of solving integral equations of the following type

$$y(x) - \int_a^b g(x,t,y(x),y(t))\,dt = f(x), \quad x \in [a,b], \tag{8.1}$$

for an unknown function y, where f and K are given functions.

If $g(x,t,y(x),y(t)) = K(x,t)y(t)$, then Eq. (8.1) is a Fredholm equation of the second kind. This form of integral equation is probably the most interesting in real applications [57], but also the non-linear Fredholm equation of the second kind are covered by using $g(x,t,y(x),y(t)) = K(x,t,y(t))$ while the linear and non-linear Volterra type integral equation of the second kind are attained by using a function g for which $g(x,t,y(x),y(t)) = 0$ for $y > x$.

8.1 Theory

We shall look for solutions of Eq. (8.1) in the class of square-integrable functions $L_2[a,b]$, using the usual norm

$$\|y\| = \left\{ \int_a^b |y(x)|^2\,dx \right\}^{\frac{1}{2}}. \tag{8.2}$$

Assume that

(i) $f \in L_2[a,b]$.

(ii) g satisfies the Lipschitz condition

$$|g(x,t,z_1,y_1) - g(x,t,z_2,y_2)| \leq N(x,t)(|z_1 - z_2| + |y_1 - y_2|), \tag{8.3}$$

where N is square-integrable, with

$$p^2 = \int_a^b \int_a^b |N(x,t)|^2\,dt\,dx. \tag{8.4}$$

(iii) $g(x,t,0,0)$ is continuous for $x, y \in [a,b]$.

Under these conditions we will show that the integral operator T defined by

$$(Ty)(x) = f(x) + \int_a^b g(x,t,y(x),y(t))\,dt\ ,\ \ x \in [a,b], \tag{8.5}$$

maps $L_2[a,b]$ into itself. Since $f \in L_2[a,b]$ and $L_2[a,b]$ is a vector space, it remains to show that $\int_a^b g(x,t,y(x),y(t))\,dt \in L_2[a,b]$.
Proof
Using condition (ii), we have

$$\begin{aligned}
&|g(x,t,u,v) - g(x,t,0,0)| \le N(x,t)(|u|+|v|) \quad \Rightarrow \\
&|g(x,t,u,v)| \le |g(x,t,0,0)| + N(x,t)|u| + N(x,t)|v| \quad \Rightarrow \\
&|\int_a^b g(x,t,u(x),v(t))\,dt| \le \\
&\qquad \int_a^b |g(x,t,0,0)|\,dt + |u(x)| \int_a^b N(x,t)\,dt + \int_a^b N(x,t)|v(t)|\,dt.
\end{aligned}$$

Each term in the right hand side of this inequality is square-integrable, the first because $g(x,t,0,0)$ is continuous. For last two terms since for a function $w \in L_2[a,b]$, using the Cauchy-Schwartz inequality for integrals yields

$$\begin{aligned}
\int_a^b \left| \int_a^b N(x,t)|w(t)|\,dt \right|^2 dx \ &\le \ \int_a^b \left\{ \int_a^b |N(x,t)|^2 dt \int_a^b |w(t)|^2 dt \right\} dx \\
&= \ \|w\|^2 \int_a^b \int_a^b |N(x,t)|^2\,dt\,dx = \|w\|^2 p^2.
\end{aligned}$$

Hence $\int_a^b g(x,t,y(x),y(t))dt$ is square-integrable, and we have shown that T maps $L_2[a,b]$ into itself.

□

8.2 The mean value enclosure of an integral operator

We wish to solve the equation

$$Ty = y, \tag{8.6}$$

which is equivalent to solving Eq. (8.1). Consider the set of functions in $L_2[a,b]$ bounded by an upper and a lower endpoint function $\underline{y}, \overline{y} \in L_2[a,b]$

$$Y = [\underline{y}, \overline{y}] = \{y \in L_2[a,b] \mid \underline{y}(t) \le y(t) \le \overline{y}(t),\ \text{for } t \in [a,b]\}. \tag{8.7}$$

Such a set of functions is called a function interval. Assume that we can calculate the set

$$TY = \{Ty \mid y \in Y\} \tag{8.8}$$

for a function interval $Y = Y^{(0)}$, where $y^{\star} \in Y^{(0)}$ and $y^{\star} = Ty^{\star}$ is a solution to Eq. (8.1). The iteration

$$Y^{(k+1)} = Y^{(k)} \cap TY^{(k)}, \qquad k = 0, 1, \ldots \tag{8.9}$$

will then generate a nested sequence of non-empty function intervals $\{Y^{(k)}\}_{k=0,\ldots}$ for which $y^{\star} \in Y^{(k)}$. The condition $Y^{(k)} \cap TY^{(k)} = \emptyset$ would result in a contradiction since $y^{\star} \in Y^{(k)} \Rightarrow Y^{(k+1)}$, and the assumption that $y^{\star} = Ty^{\star}$ cannot be true if this happens. It is usually not possible to compute TY exactly, unless T has some special properties, e.g. if T has a monotonicity property $TY = [T\underline{y}, T\overline{y}]$ or $TY = [T\overline{y}, T\underline{y}]$. Here we will consider a general computational scheme which does not assume any special properties of T. Consider a subdivision $a = \xi_0 \leq \xi_1 \leq \cdots \leq \xi_n = b$, and define

$$[x_k] = [\xi_k, \xi_{k+1}], \text{ for } k = 0, \ldots, n-1. \tag{8.10}$$

Let the function interval Y be defined piecewise by the functions $\{Y_k\}_{k=0,\ldots,n-1}$ so that $Y(x) = Y_k(x)$, for $x \in [x_k]$, where

$$Y_k = [\underline{y_k}, \overline{y_k}] = \{y \in L_2[x_k] \mid \underline{y_k}(t) \leq y(t) \leq \overline{y_k}(t), \text{ for } t \in [x_k]\}. \tag{8.11}$$

See Figure 8.18 for an example.

TY can now be enclosed by

$$TY(x) \in F([x_i]) + \sum_{j=0}^{n-1} G([x_i], [x_j], [y_i], [y_j])\mathrm{w}([x_j]), \text{ for } x \in [x_i], \tag{8.12}$$

where F and G are interval extensions of f and g, and $[y_k]$ is an interval which contains the range of Y_k

$$[y_k] = \{y(t) \mid y \in Y_k,\ t \in [x_k]\}. \tag{8.13}$$

However the approximation Eq. (8.12) is very rough and does not necessarily maintain a contraction property of T [11, 7]. Instead we will consider a mean value enclosure T_m of T so that $TY(x) \subseteq T_mY(x)$, [8, 53]

$$T_mY = Ty_m + (T'Y)(Y - y_m) \tag{8.14}$$

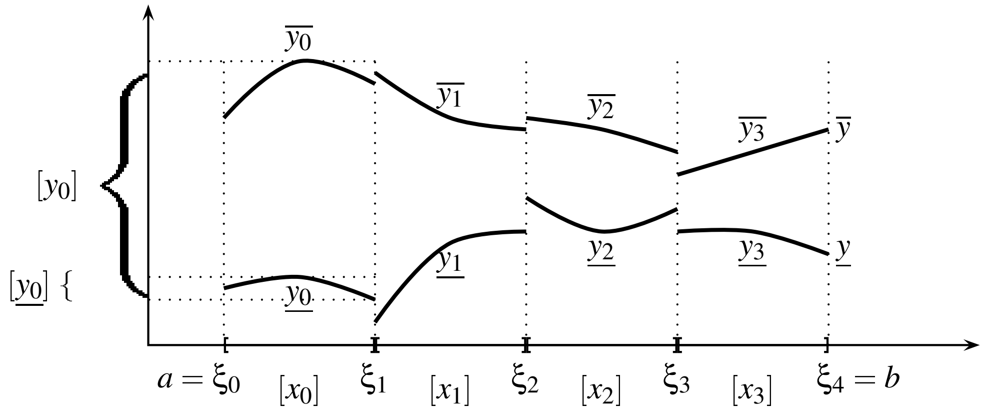


Figure 8.18: A function interval $Y = [\underline{y}, \overline{y}]$ defined piecewise on the grid $\{\xi_k\}_{k=0,\dots,n}$. In this example, we have $n = 4$.

for $y_m \in Y$. Or more detailed

$$\begin{aligned} T_mY(x) =& Ty_m(x) + (Y(x) - y_m(x)) \int_a^b G_3'(x,t,Y(x),Y(t))\,dt \\ &+ \int_a^b G_4'(x,t,Y(x),Y(t))(Y(t) - y_m(t))\,dt, \end{aligned} \tag{8.15}$$

where

$$g_3'(x,t,u,v) = \frac{\partial g(x,t,u,v)}{\partial u}, \tag{8.16a}$$

$$g_4'(x,t,u,v) = \frac{\partial g(x,t,u,v)}{\partial v}, \tag{8.16b}$$

and G_3' and G_4' are interval extensions of g_3' and g_4'. Using $y_m = \underline{y}$ in Eq. (8.15) yields

$$
\begin{aligned}
T_m Y(x) =& T\underline{y}(x) + (Y(x) - \underline{y}(x)) \sum_{j=0}^{n-1} \int_{[x_j]} G_3'(x,t,Y(x),Y(t))\,dt \\
&+ \sum_{j=0}^{n-1} \int_{[x_j]} G_4'(x,t,Y(x),Y(t))(Y(t) - \underline{y}(t))\,dt \\
\subseteq& T\underline{y}(x) + (Y(x) - \underline{y}(x)) \sum_{j=0}^{n-1} G_3'([x_i],[x_j],[y_i],[y_j])\mathrm{w}([x_j]) \\
&+ \sum_{j=0}^{n-1} G_4'([x_i],[x_j],[y_i],[y_j]) \int_{[x_j]} (Y(t) - \underline{y}(t))\,dt
\end{aligned}
\tag{8.17}
$$

using Eq. (2.34c) and that $Y(t) - \underline{y}(t)$ is non-negative. The last two terms in this formula are exactly computable in practice. Under the assumption that $T\underline{y}$ can be found exactly and certain other assumptions, it can be shown that T_m retains a contraction property of T [11].

To apply an enclosure of Eq. (8.17), we need to compute an enclosure of $T\underline{y}$. We will describe three different formulas for calculating such an enclosure, based on the zero, the first, and the second order Taylor approximations of f and g.

8.2.1 Zero order approximation

We have the obvious enclosure

$$
T\underline{y}(x) \in F([x_i]) + \sum_{j=0}^{n-1} G([x_i],[x_j],[\underline{y_i}],[\underline{y_j}])\mathrm{w}([x_j]),
\tag{8.18}
$$

for $x \in [x_i]$, where

$$
[\underline{y_k}] = \{\underline{y}(x) \mid x \in [x_k]\}.
\tag{8.19}
$$

Using this enclosure, we define the zero order interval integral operator by

$$\begin{aligned} I_0TY(x) =& F([x_i]) + \sum_{j=0}^{n-1} G([x_i],[x_j],[\underline{y_i}],[\underline{y_j}])\mathrm{w}([x_j]) \\ &+ (Y(x) - \underline{y}(x)) \sum_{j=0}^{n-1} G_3'([x_i],[x_j],[y_i],[y_j])\mathrm{w}([x_j]) \\ &+ \sum_{j=0}^{n-1} G_4'([x_i],[x_j],[y_i],[y_j]) \int_{[x_j]} (Y(t) - \underline{y}(t))\,dt, \end{aligned} \tag{8.20}$$

for $x \in [x_i]$.

8.2.2 First order approximation

Defining the real function h by

$$h(x,t) = g(x,t,\underline{y}(x),\underline{y}(t)), \tag{8.21}$$

we have

$$h_1'(x,t) = g_1'(x,t,\underline{y}(x),\underline{y}(t)) + g(x,t,\underline{y}(x),\underline{y}(t))\underline{y}'(x), \tag{8.22a}$$
$$h_2'(x,t) = g_2'(x,t,\underline{y}(x),\underline{y}(t)) + g(x,t,\underline{y}(x),\underline{y}(t))\underline{y}'(t). \tag{8.22b}$$

Let H_1' and H_2' be interval extensions of h_1' and h_2'. From the mean value enclosure, Eq. (3.3), we have for any $\hat{x}_i, x \in [x_i]$ and any $\hat{x}_j, t \in [x_j]$ that

$$h(x,t) \in h(\hat{x}_i,\hat{x}_j) + H_1'([x_i],[x_j])(x - \hat{x}_i) + H_2'([x_i],[x_j])(t - \hat{x}_j), \tag{8.23}$$

and

$$f(x) \in f(\hat{x}_i) + F'([x_i])(x - \hat{x}_i). \tag{8.24}$$

Using these enclosures with $\hat{x}_k = \xi_k$, and for $x \in [x_i]$ we obtain the first order enclosure

$$\begin{aligned}
T\underline{y}(x) \in & f(\xi_i) + F'([x_i])(x-\xi_i) + \sum_{j=0}^{n-1}\Big\{(h(\xi_i,\xi_j) \\
& + H_1'([x_i],[x_j])(x-\xi_i))\mathrm{w}([x_j]) + \int_{[x_j]} H_2'([x_i],[x_j])(t-\xi_j)dt\Big\} \\
= & f(\xi_i) + F'([x_i])(x-\xi_i) + \sum_{j=0}^{n-1}\Big\{(h(\xi_i,\xi_j) \\
& + H_1'([x_i],[x_j])(x-\xi_i))\mathrm{w}([x_j]) + H_2'([x_i],[x_j])\frac{1}{2}\mathrm{w}([x_j])^2\Big\}.
\end{aligned} \tag{8.25}$$

The first order interval integral operator can be defined by

$$\begin{aligned}
I_1TY(x) = & f(\xi_i) + F'([x_i])(x-\xi_i) + \sum_{j=0}^{n-1}\Big\{(h(\xi_i,\xi_j) \\
& + H_1'([x_i],[x_j])(x-\xi_i))\mathrm{w}([x_j]) + H_2'([x_i],[x_j])\frac{1}{2}\mathrm{w}([x_j])^2\Big\} \\
& + (Y(x)-\underline{y}(x))\sum_{j=0}^{n-1} G_3'([x_i],[x_j],[y_i],[y_j])\mathrm{w}([x_j]) \\
& + \sum_{j=0}^{n-1} G_4'([x_i],[x_j],[y_i],[y_j])\int_{[x_j]}(Y(t)-\underline{y}(t))\,dt.
\end{aligned} \tag{8.26}$$

8.2.3 Second order approximation

Differentiating Eqs. (8.22a-8.22b) once more yields

$$h''_{11}(x,t) = g''_{11} + (g''_{13} + g''_{31} + g''_{33}\underline{y}'(x))\underline{y}'(x) + g'_3\underline{y}''(x), \tag{8.27a}$$
$$h''_{12}(x,t) = g''_{12} + g''_{14}\underline{y}'(t) + (g''_{32} + g''_{34}\underline{y}'(t))\underline{y}'(x), \tag{8.27b}$$
$$h''_{21}(x,t) = g''_{21} + g''_{23}\underline{y}'(x) + (g''_{41} + g''_{43}\underline{y}'(x))\underline{y}'(t), \tag{8.27c}$$
$$h''_{22}(x,t) = g''_{22} + (g''_{24} + g''_{42} + g''_{44}\underline{y}'(t))\underline{y}'(t) + g'_4\underline{y}''(t), \tag{8.27d}$$

where all derivatives of g are evaluated in the point $(x,t,\underline{y}'(x),\underline{y}'(t))$, i.e., $g''_{12} = g''_{12}(x,t,\underline{y}'(x),\underline{y}'(t))$, etc. . Let $H''_{11}, H''_{12}, H''_{21}$ and H''_{22} be interval extensions of the second order derivatives of h. For any $\hat{x}_i, x \in [x_i]$ and any $\hat{x}_j, t \in [x_j]$, we have

$$\begin{aligned} h(x,t) \in & h(\hat{x}_i,\hat{x}_j) + h_1'(\hat{x}_i,\hat{x}_j)(x-\hat{x}_i) + h_2'(\hat{x}_i,\hat{x}_j)(t-\hat{x}_j) \\ & + \frac{1}{2}\{H_{11}''([x_i],[x_j])(x-\hat{x}_i)^2 + H_{22}''([x_i],[x_j])(t-\hat{x}_j)^2 \\ & + (H_{12}''([x_i],[x_j]) + H_{21}''([x_i],[x_j]))(x-\hat{x}_i)(t-\hat{x}_j)\}, \end{aligned} \tag{8.28}$$

and

$$f(x) \in f(\hat{x}_i) + f'(\hat{x}_i)(x-\hat{x}_i) + \frac{1}{2}F''([x_i])(x-\hat{x}_i). \tag{8.29}$$

Again choosing $\hat{x}_k = \xi_k$, and for $x \in [x_i]$ we obtain the second order enclosure

$$\begin{aligned} T\underline{y}(x) \in & f(\xi_i) + (f'(\xi_i) + \frac{1}{2}F''([x_i])(x-\xi_i))(x-\xi_i) \\ & + \sum_{j=0}^{n-1}\Big\{(h(\xi_i,\xi_j) + (h_1'(\xi_i,\xi_j) + \frac{1}{2}H_{11}''([x_i],[x_j])(x-\xi_i))(x-\xi_i))\mathrm{w}([x_j]) \\ & + (h_2'(\xi_i,\xi_j) + \frac{1}{2}(H_{12}''([x_i],[x_j]) + H_{21}''([x_i],[x_j]))(x-\xi_i))\frac{1}{2}\mathrm{w}([x_j])^2 \\ & + \frac{1}{6}H_{22}''([x_i],[x_j])\mathrm{w}([x_j])^3\Big\}. \end{aligned} \tag{8.30}$$

The second order interval integral operator can be defined by

$$\begin{aligned} I_2TY(x) = & f(\xi_i) + (f'(\xi_i) + \frac{1}{2}F''([x_i])(x-\xi_i))(x-\xi_i) \\ & + \sum_{j=0}^{n-1}\Big\{(h(\xi_i,\xi_j) + (h_1'(\xi_i,\xi_j) + \frac{1}{2}H_{11}''([x_i],[x_j])(x-\xi_i))(x-\xi_i))\mathrm{w}([x_j]) \\ & + (h_2'(\xi_i,\xi_j) + \frac{1}{2}(H_{12}''([x_i],[x_j]) + H_{21}''([x_i],[x_j]))(x-\xi_i))\frac{1}{2}\mathrm{w}([x_j])^2 \\ & + \frac{1}{6}H_{22}''([x_i],[x_j])\mathrm{w}([x_j])^3\Big\} \\ & + (Y(x) - \underline{y}(x))\sum_{j=0}^{n-1} G_3'([x_i],[x_j],[y_i],[y_j])\mathrm{w}([x_j]) \\ & + \sum_{j=0}^{n-1} G_4'([x_i],[x_j],[y_i],[y_j])\int_{[x_j]}(Y(t) - \underline{y}(t))\,dt. \end{aligned} \tag{8.31}$$

8.3 Using a Bernstein polynomial enclosure

For representing the function interval Y, we use interval Bernstein polynomials since they have some nice properties; see [54]. The Bernstein basis polynomials of degree k are given by

$$\phi_j^{(k)} = \binom{k}{j} z^j(1-z)^{(k-j)},\ j=0,\dots,k,\ z\in[0,1]. \tag{8.32}$$

A real function in the Bernstein basis of order k is given by the coefficients $a_j \in \mathbb{R},\ j=0,\dots,k,$

$$u(z) = \sum_{j=0}^{k} a_j \phi_j^{(k)}(z). \tag{8.33}$$

An interval function is given by the coefficients $[a_j] \in \mathbb{IR},\ j=0,\dots,k,$

$$U(z) = \sum_{j=0}^{k} [a_j]\phi_j^{(k)}(z). \tag{8.34}$$

The Bernstein polynomials have some nice properties:

1) The basis functions are non-negative for $z \in [0,1]$.
2) The sum of the k basis functions is 1,

$$\sum_{j=0}^{k} \phi_j^{(k)}(z) = 1. \tag{8.35}$$

3) The value of the polynomials in $z=0$ and $z=1$ are given by the 0th and the kth coefficients. i.e., $u(0)=a_0, u(1)=a_k$, and $U(0)=[a_0], U(1)=[a_k]$.

Because of property 1) we have that the endpoint functions $\underline{u}$ and $\overline{u}$ of the interval function $U(z)$ in the Bernstein basis are given by real Bernstein polynomials of the same degree with coefficients which are the endpoints of the corresponding interval coefficients in the interval Bernstein polynomial,

$$U(z) = [\underline{u}(z), \overline{u}(z)] = \left[\sum_{j=0}^{k} \underline{a_j}\phi_j^{(k)}(z),\ \sum_{j=0}^{k} \overline{a_j}\phi_j^{(k)}(z)\right]. \tag{8.36}$$

Assume that U and V are interval Bernstein polynomials of degree k given by the coefficients $\{[a_j]\}_{j=0,\dots,k}$ and $\{[b_j]\}_{j=0,\dots,k}$ and let $[c] \in \mathbb{IR}$ be an interval. Now we have the following elementary operations [55]

$$(U+V)(z) = \sum_{j=0}^{k} ([a_j]+[b_j])\phi_j^{(k)}(z), \tag{8.37a}$$

$$(U+[c])(z) = \sum_{j=0}^{k} ([a_j]+[c])\phi_j^{(k)}(z), \tag{8.37b}$$

$$(U-V)(z) = \sum_{j=0}^{k} ([a_j]-[b_j])\phi_j^{(k)}(z), \tag{8.37c}$$

$$(U-[c])(z) = \sum_{j=0}^{k} ([a_j]-[c])\phi_j^{(k)}(z), \tag{8.37d}$$

$$(U\cdot V)(z) = \sum_{i=0}^{2k}\sum_{j=0}^{k} [a_j][b_{i-j}] \binom{k}{j}\binom{k}{i-j} \phi_i^{(2k)}(z) / \binom{2k}{i}, \tag{8.37e}$$

$$(U\cdot [c])(z) = \sum_{j=0}^{k} [c][a_j]\phi_j^{(k)}(z). \tag{8.37f}$$

We have that

$$\int_0^1 \phi_j^{(k)}(z)\,dz = \frac{1}{k+1}, \text{ for } j = 0,\dots,k. \tag{8.38}$$

Since $\phi_j^{(k)}(z)$ is non-negative for $z \in [0,1]$, we have that

$$\begin{aligned} \int_0^1 U(z)\,dz &= \int_0^1 \sum_{j=0}^{k} [a_j]\phi_j^{(k)}(z)\,dz = \sum_{j=0}^{k} [a_j] \int_0^1 \phi_j^{(k)}(z)\,dz \\ &= \frac{1}{k+1} \sum_{j=0}^{k} [a_j], \end{aligned} \tag{8.39}$$

using Eq. (2.34c). We want the function intervals to be defined piecewise by Eq. (8.11), where each function $Y_k : [x_k] \to \mathbb{IR}$ is an interval Bernstein polynomial given by

$$Y_k(x) = U_k\left(\frac{x-\xi_k}{\mathrm{w}([x_k])}\right), \text{ for } k = 0,\dots,n-1. \tag{8.40}$$

where $U_k(z) : [0,1] \to \mathbb{IR}$ are interval Bernstein polynomials given in the same form as in Eq. (8.34). Because of this rescaling of the argument, we have

$$\begin{aligned}\int_{[x_k]} Y_k(x)\,dx &= \mathrm{w}([x_k]) \int_0^1 U_k(z)\,dz \\ &= \frac{\mathrm{w}([x_k])}{k+1} \sum_{j=0}^{k} [a_j] \end{aligned} \tag{8.41}$$

We also need to differentiate real Bernstein polynomials for implementing the first and second order enclosures. Since for $k \neq 0$ we have that

$$\left(\phi_i^{(k)}\right)' = \begin{cases} -k\phi_0^{(k-1)}, & \text{for } i = 0, \\ k\left(\phi_{i-1}^{(k-1)} - \phi_i^{(k-1)}\right), & \text{for } 0 < i < k, \\ k\phi_{k-1}^{(k-1)}, & \text{for } i = k, \end{cases} \tag{8.42}$$

the differentiation of a real Bernstein polynomial yields

$$u'(z) = \sum_{j=0}^{k-1} k(a_{j+1} - a_j)\phi_j^{(k-1)}(z), \tag{8.43}$$

If the real functions $y_k : [x_k] \to \mathbb{R}$ are given by

$$y_k(x) = u_k\left(\frac{x - \xi_k}{\mathrm{w}([x_k])}\right), \text{ for } k = 0, \dots, n-1, \tag{8.44}$$

where u_k are real Bernstein polynomials given in the same form as in Eq. (8.33), then we have

$$y_k'(x) = \frac{1}{\mathrm{w}([x_k])} \sum_{j=0}^{k-1} k(a_{j+1} - a_j)\phi_j^{(k-1)}\left(\frac{x - \xi_k}{\mathrm{w}([x_k])}\right). \tag{8.45}$$

8.4 Numerical examples

Using the iteration scheme Eq. (8.9) based on one of the interval integral operators I_0T, I_1T, or I_2T it is possible to implement Algorithm 8.7 for obtaining an enclosure of the solution to the integral equation Eq. (8.1).

The symbol $\tilde{\cap}$ in Algorithm 8.7 denotes a modified intersection operator: Since the result of the intersection $I_pTY_j \cap Y_j$ cannot necessarily be represented

Initialize:
$n, \{\xi_j\}_{j=0,\dots,n}, \{Y_k\}_{k=0,\dots,n}, p.$
Iteration:
for $j = 0$ to $n-1$,
$Y_j := I_p T Y_j \tilde{\cap} Y_j$,
if $Y_j = \emptyset$ *then* No solution exists,
Output:
$\{Y_k\}_{k=0,\dots,n}$

Algorithm 8.7: Function interval iteration scheme.

by an interval Bernstein polynomial, we use a modification: $U_j = I_p T Y_j \tilde{\cap} Y_j$ defined by

$$\underline{U_j} = \begin{cases} \underline{I_p T Y_j}, & \text{if } \underline{I_p T Y_j}(x) \leq \underline{Y_j}(x), \text{ for } x \in [x_j] \\ \underline{Y_j}, & \text{otherwise} \end{cases} \tag{8.46a}$$

$$\overline{U_j} = \begin{cases} \overline{I_p T Y_j}, & \text{if } \overline{I_p T Y_j}(x) \geq \overline{Y_j}(x), \text{ for } x \in [x_j] \\ \overline{Y_j}, & \text{otherwise} \end{cases} \tag{8.46b}$$

Using $\tilde{\cap}$ instead of $\cap$ in the iteration Eq. (8.9) retains the nesting property of the iterates. If we encounter a case where $Y_j = \emptyset$, i.e., $\underline{Y_j}(x) > \overline{Y_j}(x)$ for some $x \in [x_j]$, we stop the program with the message that no solution exists within the initial value.

The function interval Y in Algorithm 8.7 is updated each time a new piece of the function interval Y_j has been calculated. This Gauss-Seidel-like method helps to speeds up the convergence of the method [12].

We terminate the iteration process in Algorithm 8.7 when Y becomes invariant, i.e., when $Y = I_p T Y \tilde{\cap} Y$. It can be proven that this condition will be true after a finite number of iterations [46]. The result will be denoted $Y^\star$ since it is a fixed point of Algorithm 8.7.

8.4.1 The Chandrasekhar equation

Consider the Chandrasekhar equation (or the H-equation)

$$y(x) = 1 + xy(x)\int_0^1 \frac{\Psi(t)y(t)}{x+t}dt, \text{ for } x \in [0,1] \tag{8.47}$$

which arises from problems in radiative transfer [4, 48]. The function Ψ is a known function for $x \in [0,1]$. Here we use $\Psi(t) \equiv \lambda$, where λ is a constant. Using $f(x) \equiv 1$ and

$$g(x,t,y(x),y(t)) = \lambda \frac{x}{x+y} y(x)y(t), \tag{8.48}$$

we obtain the form of Eq. (8.1). The derivatives needed for implementing the zero order enclosure Eq. (8.20) are

$$\begin{aligned} g_3'(x,t,y(x),y(t)) &= \lambda \frac{x}{x+t} y(t), \\ g_4'(x,t,y(x),y(t)) &= \lambda \frac{x}{x+t} y(x). \end{aligned}$$

For $x,t > 0$ we can use the simple interval extension, but when $x,t \in [x_0]$, we use the fact that $x \leq x+t$ to derive the interval extensions

$$G([x_0],[x_0],[y_0],[y_0]) = \lambda[0,1][y_0][y_0], \tag{8.49}$$

and

$$G_3'([x_0],[x_0],[y_0],[y_0]) = G_4'([x_0],[x_0],[y_0],[y_0]) = \lambda[0,1][y_0].$$

When implementing the first order approximation Eq. (8.26), we also need the interval extensions of g_1' and g_2',

$$\begin{aligned} g_1'(x,t,y(x),y(t)) &= \lambda \frac{x}{(x+t)^2} y(x)y(t), \\ g_2'(x,t,y(x),y(t)) &= \lambda \frac{x}{(x+t)^2} y(x)y(t). \end{aligned}$$

For $x,t > 0$ the simple interval extensions of g_1' and g_2' are used, but g_1' and g_2' become unbounded when $x,t \in [x_0]$. In this case, we are forced to use the zero order approximation of h in Eq. (8.26).

n	I_0T	I_1T
15	$9.692423 \cdot 10^{-2}$	$4.693960 \cdot 10^{-2}$
30	$4.492288 \cdot 10^{-2}$	$1.134352 \cdot 10^{-2}$
60	$2.177521 \cdot 10^{-2}$	$2.817790 \cdot 10^{-3}$

Table 8.2: The width of the final function interval $\mathrm{w}(Y^\star)$ when using the zero and the first order enclosures and using different numbers of discretization points $n = 15, 30, 60$.

Algorithm 8.7 has been applied to Eq. (8.47) with the constant $\lambda = \frac{1}{4}$, using the non-equidistant discretization points $\xi_k = \frac{k^2}{n^2}$ for $k = 0, \ldots, n$ and an initial value $Y = [0.99, 1.4]$. The result of using the zero order and the modified first order method for different numbers of discretization points $n = 15, 30, 60$ is shown in Figure 8.19. In the graphs in Figure 8.19, all iterates are shown until the algorithm terminates. Table 8.2 lists the width of $Y^\star$

$$\mathrm{w}(Y^\star) = \max_{x \in [0,1]} \mathrm{w}(Y(x)). \tag{8.47}$$

From this example we see that the convergence of the zero order method is $\mathrm{O}(\frac{1}{n})$, while it is $\mathrm{O}(\frac{1}{n^2})$ for the first order method. This result of the convergence has been shown to be true in general [11].

8.4.2 Solving a boundary value problem

Consider again the boundary value problem Eq. (7.14). By using Green's function [44, 21], we obtain an equivalent integral equation, which has the same solutions as Eq. (7.14)

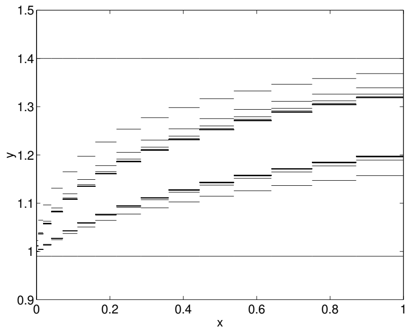
$$u(x) = \int_0^1 s(x,t)(a^2 \sin(u(t)) - \sin(t\pi))\, dt, \tag{8.48}$$

where s is Green's function

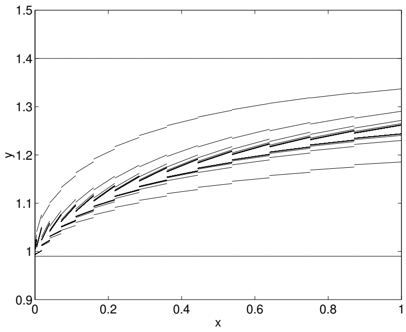
$$s(x,t) = \begin{cases} t(1-x), & 0 \le t \le x, \\ x(1-t), & x \le t \le 1. \end{cases} \tag{8.49}$$

Using $f(x) \equiv 0$ and

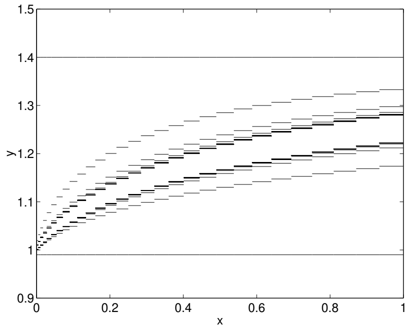
$$g(x,t,u(x),u(t)) = s(x,t)(a^2 \sin(u(t)) - \sin(t\pi)), \tag{8.50}$$



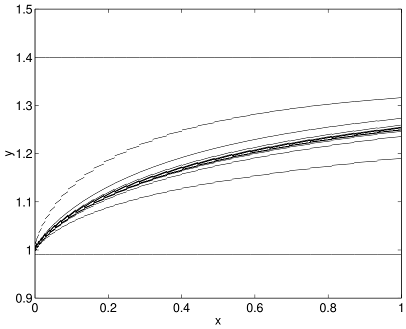
(a) $p = 0$, $n = 15$.



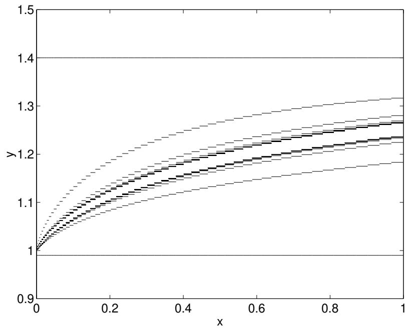
(b) $p = 1$, $n = 15$.



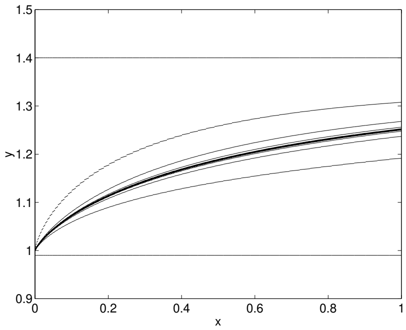
(c) $p = 0$, $n = 30$.



(d) $p = 1$, $n = 30$.



(e) $p = 0$, $n = 60$.



(f) $p = 1$, $n = 60$.

Figure 8.19: Solving the Chandrasekhar equation using the zero- and the first order enclosures and different numbers of discretization points.

we obtain the form of Eq. (8.1). The derivatives of g with respect to the third and fourth variable are

$$\begin{aligned} g_3'(x,t,u(x),u(t)) &= 0, \\ g_4'(x,t,u(x),u(t)) &= a^2 s(x,t)\cos(u(t)), \end{aligned}$$

The interval extensions of g and its derivatives are

$$\begin{aligned} G([x_i],[x_j],[u],[v]) &= S([x_i],[x_j])(a^2\sin([v]) - \sin([x_j]\pi)), \\ G_3'([x_i],[x_j],[u],[v]) &= 0, \\ G_4'([x_i],[x_j],[u],[v]) &= a^2 S([x_i],[x_j])\cos([v]), \end{aligned}$$

where for $x \in [x_i]$ and $t \in [x_j]$, we have the interval extension

$$S([x_i],[x_j]) = \begin{cases} [x_j](1-[x_i]), & 0 \le t \le x, \\ [x_i](1-[x_j]), & x \le t \le 1. \end{cases} \tag{8.51}$$

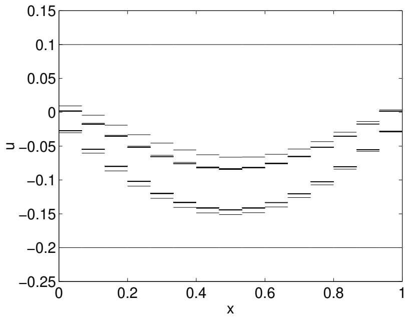
We have that $i < j \Rightarrow x \le t$ and $i > j \Rightarrow t \le x$, but when $i = j$ we do not know which of the cases in Eq. (8.51) to use. Fortunately, it does not matter here since the two cases become equal.

The partial derivatives of g with respect to the its first and second variable are

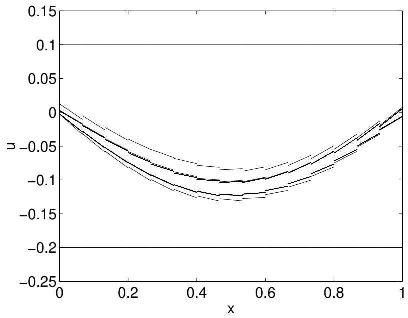
$$\begin{aligned} g_1'(x,t,u(x),u(t)) &= \begin{cases} t(\sin(t\pi) - a^2\sin(u(t))), & 0 \le t \le x \\ (1-t)(a^2\sin(u(t)) - \sin(t\pi)), & x \le t \le 1 \end{cases} \\ g_2'(x,t,u(x),u(t)) &= \begin{cases} (1-x)(a^2\sin(u(t)) - \sin(t\pi) - t\pi\cos(t\pi)), \\ \hfill 0 \le t \le x \\ x((1-t)\pi\cos(t\pi) - a^2\sin(u(t)) + \sin(t\pi)), \\ \hfill x \le t \le 1 \end{cases} \end{aligned}$$

When implementing the interval extensions G_1' and G_2', we have to take special care when $i = j$ since the two cases have different values. One way to fix this problem is to use the smallest interval containing the values of the two cases. However we use the zero order approximation of h instead as we did when solving the Chandrasekhar equation, since this method seems to produce tighter enclosures. The result of using the zero and the modified first order method with an equidistant discretization $\xi_k = \frac{k}{n}$ for $k = 0,\dots,n$, $n = 15, 30, 60$ and an initial value $Y = [-0.2, 0.1]$ is shown in Figure 8.48. In the graphs in Figure 8.20, all iterates are shown until the algorithm terminates. Table 8.3 lists the width of $Y^\star$.

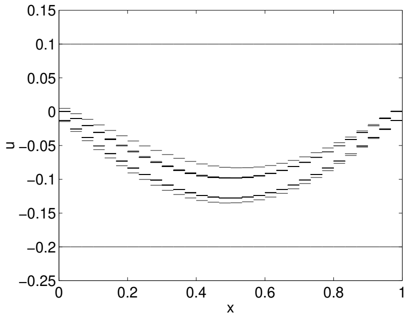
Also here the convergence of the zero order method is $O(1/n)$ and $O(1/n^2)$ for the first order method.



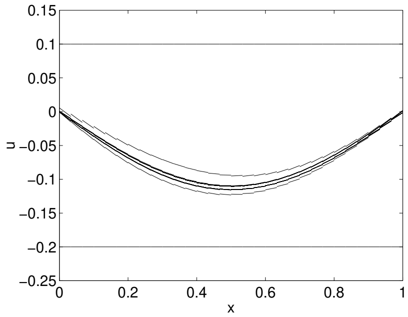
(a) $p = 0$, $n = 15$.



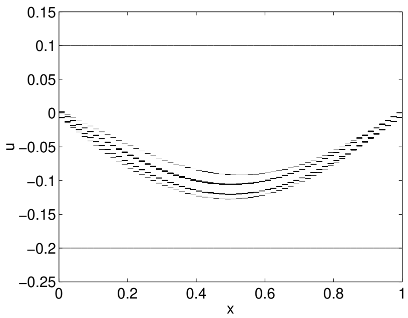
(b) $p = 1$, $n = 15$.



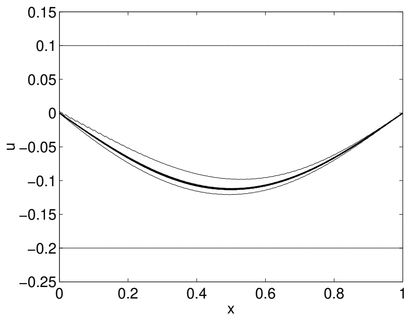
(c) $p = 0$, $n = 30$.



(d) $p = 1$, $n = 30$.



(e) $p = 0$, $n = 60$.



(f) $p = 1$, $n = 60$.

Figure 8.20: Solving the integral equation Eq. (8.48) using the zero- and the first order enclosures and different numbers of discretization points.

n	I_0T	I_1T
15	$1.212875 \cdot 10^{-1}$	$2.173564 \cdot 10^{-2}$
30	$5.808942 \cdot 10^{-2}$	$5.510507 \cdot 10^{-3}$
60	$2.868874 \cdot 10^{-2}$	$1.388055 \cdot 10^{-3}$

Table 8.3: The width of the final function interval $\mathrm{w}(Y^{\star})$ when using the zero and the first order enclosures and using different numbers of discretization points $n = 15, 30, 60$.

9 Conclusion

We have seen how to perform interval arithmetic in which intervals of real numbers are used instead of real numbers. These intervals are represented by two real numbers, the lower and the upper endpoints. When implementing interval arithmetic on a computer, the endpoints are taken from the limited set of real numbers which are representable by the actual computer used. When operating on intervals, we use outward rounding so that the interval result is guaranteed to contain all the true results of the corresponding real operation for all combination of real numbers within the interval arguments. Since we are capable of performing computations on sets of real numbers, we can apply fixed point theorems, which enable us to prove existence and in some cases uniqueness of fixed points of mappings. For example using the interval Newton or the interval Krawczyk methods, we can implement algorithms capable of proving existence and uniqueness of solutions to non-linear equations within some interval vector. Algorithms based on such methods are called "self-validating methods".

Using the natural interval extension of a discrete map often lead to unnecessarily wide interval enclosures of its iterates. However, by using the derivatives to form the mean value enclosure will in general lead to tighter enclosures since this extension reduces the overestimation caused by multiple instances of interval variables representing the same value. Using the extended mean value enclosure will in many cases improve the enclosures even more since this enclosure fights the so-called wrapping by using a rotating rectangular enclosure.

The actual expressions we need to differentiate in practice can be fairly complicated and it would be quite tedious to differentiate by hand. By using automatic differentiation, it is possible to generate derivatives automatically along with the computations themselves and expressions implemented as algorithms can easily be differentiated without writing additional code. Three methods for obtaining derivatives have been implemented in C++: the forward- and the backward methods have been implemented in the package FADBAD, and the Taylor expansion method has been implemented in the package TADIFF. We have also seen that the methods can be combined so that derivatives can be obtained in a very flexible way – opening up for several new applications which would be hard to implement without the aid of automatic differentiation.

One important application of the Taylor expansion method is the ability to obtain the Taylor coefficients of a function given implicitly by an ordinary differential equation. Since we are also capable of computing enclosures of the truncation errors of truncated Taylor series, it is possible to discretize solutions of ordinary differential equations and obtain interval mappings which enclose the true solutions of the ordinary differential equations. The enclosures which are obtained using these mappings can be improved considerably by using the mean value enclosure of the mapping instead of the mapping itself. By using the extended mean value enclosure, we can improve the enclosures even more, since this method also fights the wrapping effect. Using the FADBAD/TADIFF packages, an ordinary differential equation solver called "Automatic Differentiation Interval Ordinary Differential Equation Solver (ADIODES)" has been implemented in C++. This package is based on a combination of the forward and the Taylor expansion methods for obtaining the mean value enclosure of a Taylor expansion of the ordinary differential equation, which is given by the user as a C++ function. Only this function has to be altered when implementing other ordinary differential equations.

Using ADIODES and the interval Newton method, we have proven existence and uniqueness of periodic solutions to both autonomous and non-autonomous systems of ordinary differential equations. These equations are of special interest in dynamical systems theory, and the existence of these periodic solutions has not yet been proven by analytical methods.

We have also seen how to use ADIODES for proving existence and uniqueness of a solution to a boundary value problem.

Interval methods has also been applied to a class of integral equations, and the mean value enclosure of the corresponding integral operator is use to obtain a piecewise upper- and lower endpoint functions which bounds the true solutions. Two integral equations have been implemented using zero and first order polynomial bounds.

9.1 Directions for future research

The FADBAD/TADIFF packages can handle combinations of automatic differentiation methods using in principle any arithmetic type. This flexibility enables for new interesting applications, of which we have seen only a few. Other applications which might be worth examining are

- Applications involving multidimensional Taylor series expansions.
- Taylor expanding a function $x(\alpha)$ given implicitly by a fixed point problem $f(x(\alpha),\alpha) = x(\alpha)$.

The FADBAD/TADIFF packages need to be further developed

- Optimization w.r.t. sparsity. The packages has not yet been optimized for handling large and sparse systems.

We have seen how to combine interval analysis and automatic differentiation for developing a modular, user friendly and easy to use package for enclosing solutions of ordinary differential equations, but still much more work needs to be done to improve the method. Some problems include:

- Since the Picard iterations used for proving existence and obtaining the first hand enclosure of the solution are based on an interval vector enclosure, the integration steps are restricted to Euler steps. We need higher order methods which are capable of proving existence using longer integration steps, also when solving stiff problems.

- When solving stiff systems, the Taylor series expansion has a very slow convergence, the coefficients becomes very large, and they oscillate. In ADIODES, we handle this problem by taking very small integration steps, but if we want to take larger integration steps we need alternative methods instead of evaluating the Taylor series expansion, e.g. by using implicit methods.

- We need to find a better strategy for combined control of the integration step size and the order of the Taylor expansion.

The ADIODES program needs to be further developed

- A function for obtaining an enclosure of the solution at any value of the independent variable needs to be implemented.

- A function for obtaining an enclosure of the intersection of the solution and a hyper-plane in the phase space needs to be implemented.

Other applications in dynamical systems theory which need to be developed include:

- Global methods for finding and proving existence of all periodic solutions of an ordinary differential equation, e.g. by using a divide and conquer strategy, as used in [53] for discrete mappings.

- Methods for proving existence of chaos in a system. E.g. by proving intersection of unstable and stable manifolds of hyperbolic fixed points.
- Methods for encapsulating all possible solutions of an ordinary differential equation by enclosing its invariant sets in phase space, e.g. by using the method described in [53, 26] for discrete mappings.

References

[1] G. ALEFELD, *Inclusion Methods for Systems of Nonlinear Equations – The Interval Newton Method and Modifications*, in Topics in Validated Computations, J. Herzberger, ed., IMACS–GAMM, 1994, pp. 7–26.

[2] G. ALEFELD AND J. HERZBERGER, *Introduction to Interval Computations*, Academic Press, 1983.

[3] V. I. ARNOLD, *Ordinary Differential Equations*, The MIT Press, Cambridge, Massachusetts, and London, England, 1973.

[4] C. T. H. BAKER, *The Numerical Treatment of Integral Equations*, Clarendon press, Oxford, 1977.

[5] C. BENDTSEN AND O. STAUNING, *FADBAD, A Flexible C++ Package for Automatic Differentiation, Using the Forward and Backward Methods*, Tech. Rep. IMM-REP-1996-17, Department of Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, Aug. 1996.

[6] ——, *TADIFF, A Flexible C++ Package for Automatic Differentiation, Using Taylor Series Expansion*, Tech. Rep. IMM-REP-1997-07, Department of Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, Apr. 1997.

[7] O. CAPRANI AND K. MADSEN, *Interval Contractions for the Solution of Integral Equations*, Interval Mathematics, (1980), pp. 281–290.

[8] ——, *Mean Value Forms in Interval Analysis*, Computing, 25 (1980), pp. 147–154.

[9] ——, *Experiments with Interval Methods for Nonlinear Systems*, tech. rep., Institut für Angewandte Mathematik, Universität Freiburg i. Br., Hermann–Herder–Straße 10, D-7800 Freiburg i. Br., Germany, 1981.

[10] O. CAPRANI, K. MADSEN, AND L. B. RALL, *Integration of Interval Functions*, SIAM J. Numer. Anal., 12 (1981), pp. 321–341.

[11] O. CAPRANI, K. MADSEN, AND O. STAUNING, *Enclosing Solutions of Integral Equations*, Tech. Rep. IMM-REP-1996-19, Department of Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, Aug. 1996.

[12] ——, *Existence Test for Asynchronous Interval Iteration*, Reliable Computing, 3 (1997), pp. 269–275.

[13] B. A. COOMES, H. KOÇAK, AND K. J. PALMER, *Periodic shadowing*, Contemporary Mathematics, 172 (1994), pp. 115–130.

[14] W. A. COPPEL, *Stability and Asymptotic Behavior of Differential Equations*, D. C. Heath and Company, Boston, 1965.

[15] G. F. CORLISS, *Survey of Interval Algorithms for Ordinary Differential Equations*, Appl. Math. Comput., 31 (1989), pp. 112–120.

[16] ——, *Overloading Point and Interval Taylor Operators*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., Philadelphia, Penna., 1991, SIAM, pp. 139–146.

[17] G. F. CORLISS AND Y. CHANG, *Solving Ordinary Differential Equations using Taylor Series*, ACM Transactions on Mathematical Software, 8 (1982), pp. 114–144.

[18] G. F. CORLISS, G. S. KRENZ, AND P. H. DAVIS, *Bibliography on interval methods for the solution of ordinary differential equations*, Tech. Rep. 289, Department of Mathematics, Statistics and Computer Science, Marquette University, Milwaukee, WI 53233., Sept. 1988.

[19] A. GRIEWANK, *On Automatic Differentiation*, Kluwer Academic Publishers, P.O.Box 17, 2200 AA Dordrecht, The Netherlands, 1989, pp. 83–108. Also available as Tecnical Report number ANL/MCS-P10-1088 or CRPC-TR89003 from Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue Argonne, Illinois 60439.

[20] A. GRIEWANK AND S. REESE, *On the Calculation of Jacobian Matrices by the Markowitz Rule*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., Philadelphia, Penna., 1991, SIAM, pp. 126–135.

[21] D. GRIFFEL, *Applied Functional Analysis*, Ellis Horwood, Market Cross House, Cooper Street Chichester, West Sussex, PO19 1EB, 1981.

[22] E. HANSEN, *Topics in Interval Analysis*, Oxford University Press, Ely House, London W. 1, 1969.

[23] P. HARTMAN, *Ordinary Differential Equations*, John Wiley & Sons, Inc., New York, London, Sydney, 1964.

[24] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC., *IEEE Standart for Binary Floating-Point Arithmetic*, 345 East 47th Street, New York, NY 10017, USA., Aug. 1985.

[25] D. W. JUEDES, *A Taxonomy of Automatic Differentiation Tools*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., Philadelphia, Penna., 1991, SIAM, pp. 315–329.

[26] C. KNUDSEN AND O. STAUNING, *Exact Numerical Methods in Dynamical Systems Theory*. Submitted, 1997.

[27] C. KNUDSEN, J. STURIS, AND J. S. THOMSEN, *Generic bifurcation structures of Arnol'd tongues in forced oscillators*, Physical Review A, 44 (1991), pp. 3503–3510.

[28] O. KNÜPPEL, *BIAS – Basic Interval Arithmetic Subroutines*, Technische Universität Hamburg–Harburg, Technische Universität Hamburg–Harburg Technische Informatik III D-21071 Hamburg, Germany, 1993. Also available from <http://www.ti3.tu-harburg.de/indexEnglisch.html>.

[29] ——, *PROFIL – Programmer's Runtime Optimized Fast Interval Library*, Technische Universität Hamburg–Harburg, Technische Universität Hamburg–Harburg Technische Informatik III D-21071 Hamburg, Germany, 1993. Also available from <http://www.ti3.tu-harburg.de/indexEnglisch.html>.

[30] ——, *PROFIL/BIAS - A Fast Interval Library*, Computing, 53 (1994), pp. 277–287.

[31] R. KRAWCZYK, *Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken*, Computing, 4 (1969), pp. 187–201.

[32] J. LAMBERT AND A. R. MITCHELL, *On the Solution of $y' = f(x,y)$ by a Class of High Accuracy Difference Formulae of Low Order*, ZAMP, 13 (1962), pp. 223–232.

[33] R. LOHNER, *Interval Arithmetic in Staggered Correction Format*, Scientific Computing with Automatic Result Verification, (1993), pp. 301–321.

[34] R. J. LOHNER, *Enclosing the Soulutions of Ordinary Initial and Boundary Value Problems*, in Computer Arithmetic: Scientific Computation and Programming Languages, E. Kaucher, U. Kulisch, and C. Ullrich, eds., B. G. Teubner Stuttgart, 1987, Universität Karlsruhe, pp. 255–286.

[35] ——, *Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben*, PhD thesis, Fakultät für Mathematik der Universität Karlsruhe, June 1988. In German.

[36] E. N. LORENZ, *Deterministic Nonperiodic Flow*, J. Atmos. Sci, (1963), pp. 130–141.

[37] R. E. MOORE, *Automatic Local Coordinate Transformation to Reduce the Growth of Error Bounds in Interval Computation of Solutions of Ordinary Differential Equations*, in Error in Digital Computation, L. B. Rall, ed., vol. 2, John Wiley & Sons, 1965, pp. 103–140.

[38] ——, *Interval Analysis*, Prentice–Hall, Series in Automatic Computation, Englewood Cliffs, N.J., 1966.

[39] ——, *Methods and Applications of Interval Analysis*, SIAM Studies in Applied Mathematics, Philadelphia, 1978.

[40] ——, *Interval Methods for Nonlinear Systems*, in Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis, G. Alefeld, R. Grigorieff, R. Albrecht, U. Kulisch, and F. Stummel, eds., vol. 2, Wien New York, 1980, Springer-Verlag, pp. 113–120.

[41] N. S. NEDIALKOV, K. R. JACKSON, AND G. F. CORLISS, *Validated Solutions if Initial Value Problems for Ordinary Differential Equations*, tech. rep., Department of Computer Science of the University of Toronto, Feb. 1997.

[42] L. QI, *Interval Boxes of Solutions of Nonlinear Systems*, Computing, 27 (1981), pp. 137–144.

[43] L. B. RALL, *Numerical Integration and the Solution of Integral Equations by the use of Riemann Sums*, SIAM J. Numer. Anal., 7 (1965), pp. 55–64.

[44] ——, *Computational Solution of Nonlinear Operator Equations*, Prentice–Hall, Series in Automatic Computation, Englewood Cliff, NJ., 1979.

[45] ——, *Applications of Software for Automatic Differentiation in Numerical Computation*, in Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis), G. Alefeld, R. Grigorieff, R. Albrecht, U. Kulisch, and F. Stummel, eds., vol. 2, Wien, New York, 1980, Springer-Verlag, pp. 141–156.

[46] ——, *A Theory of Interval Iteration*, Proceedings of the American Mathematical Society, 86 (1982), pp. 625–631.

[47] ——, *Mean Value and Taylor Forms in Interval Analysis*, SIAM J. Numer. Anal., 14 (1983), pp. 223–238.

[48] ——, *Application of Interval Integration to the Solution of Integral Equations*, Journal of Integral Equations, 6 (1984), pp. 127–141.

[49] R. RIHM, *Interval Methods for Initial Value Problems in ODEs*, in Topics in Validated Computations, J. Herzberger, ed., IMACS–GAMM, 1994, pp. 173–207.

[50] ——, *On a Class of Enclosure Methods for Initial Value Problems*, Computing, 53 (1994), pp. 369–377.

[51] D. SHIRIAEV, *Fast Automatic Differentiation for Vector Processors and Reduction of the Spatial Complexity in a Source Translation Environment*, PhD thesis, Fakultät für Mathematik der Universität Karlsruhe, Dec. 1993.

[52] C. SPARROW, *The Lorenz equations*, Springer, New York, 1982.

[53] O. STAUNING, *Interval Analyse i Dynamisk Systemteori*, Master's thesis, Department of Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, 1994. In Danish.

[54] ——, *Solving Integral Equations Using Interval Analysis*, Tech. Rep. IMM-REP-1995-21, Department of Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, 1995.

[55] ——, *Enclosing Solutions of Ordinary Differential Equations*, Tech. Rep. IMM-REP-1996-18, Department of Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, 1996.

[56] A. STUART, *Numerical Analysis of Dynamical Systems*, Cambridge University Press, 1994.

[57] P. P. ZABREYKO, A. I. KOSHELEV, M. A. KRASNOSEL'SKII, S. G. MIKHLIN, L. S. RAKOVSHCHIK, AND V. Y. STET'SENKO, *Integral Equations - A Reference Text*, Noordhoff, The Netherlands, 1975.

Index