Case Studies in Exact Real Arithmetic - Implementations and empirical Evaluation

Fallstudien in exakter reeller Arithmetik: Implementation und empirische Evaluation
Master-Thesis von Holger Thies
Tag der Einreichung:

1. Gutachten: Prof. Dr. Martin Ziegler
2. Gutachten: Prof. Dr. Peter Hertling

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department of Mathematics
Mathematical Logic Group

Case Studies in Exact Real Arithmetic - Implementations and empirical Evaluation
Fallstudien in exakter reeller Arithmetik: Implementation und empirische Evaluation

Vorgelegte Master-Thesis von Holger Thies

1. Gutachten: Prof. Dr. Martin Ziegler
2. Gutachten: Prof. Dr. Peter Hertling

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den April 14, 2015

(H. Thies)

Abstract. The thesis gives examples for the possibilities and limitations of exact real arithmetic in form of two case studies. Theoretical results from real complexity theory are compared with time measurements obtained from empirical evaluation of algorithms implemented in the `iRRAM` C++ framework for exact real arithmetic. An overview on some results from computable analysis and an introduction to the `iRRAM` framework is included.

Acknowledgments. I first want to thank my supervisor, Martin Ziegler, for giving me the opportunity to work on this topic and the helpful advice he provided. I would also like to thank him for his kind and constant support during my graduate and undergraduate education.

I also want to thank Christoph Spandl and Peter Hertling from the Bundeswehr University Munich for providing the topic for the second case study.

Further, I would like to thank Akitoshi Kawamura, under whose supervision a large part of the second case study was conducted. I am also very grateful to him and Hiroshi Imai for allowing me to stay at the University of Tokyo as an exchange student and for their support during this stay.

Last but not least, I want to thank Florian Steinberg, for working together with me on the second case study. Many of the algorithms were implemented by him and he had lots of great ideas on how to improve the implementation.

Contents

1 Introduction

A large amount of problems that are solved with the help of computers are problems dealing with real numbers. Examples include computing zeros, maxima or minima of real functions, solving differential equations or determining eigenvalues.

The most common way to represent real numbers on a computer is the use of floating point arithmetic. The floating point representation of a real number is a finite word, consisting of a mantissa and an exponent of fixed length. Due to the finiteness of the representation, it is unavoidable that errors occur. While in many cases the error stays small, there are also several examples where computations with floating point numbers yield results that are completely wrong and far off from the correct value.

In any case, the error is not visible from the representation, thus without additional analysis it is impossible to know if the error is too big for a particular application or not. Floating point arithmetic is therefore not a reliable way to perform real number computations.

There are many cases where reliable results are necessary. Writing reliable algorithms on numerical problems requires, however, a sound theoretical foundation.

While every natural number is obviously computable, the same can not hold for real numbers for countability reasons.

Computable real numbers were already introduced by Alan Turing in his famous paper "On computable numbers, with an application to the Entscheidungsproblem" [Tur36]. However, while for discrete-valued problems there is a widely accepted theory for computability and complexity, there are several non-equivalent theories for computations on real numbers and none of them is universally accepted among researchers.

One such theory is the Type-2 Theory of Effectivity (TTE) [Wei00]. In TTE reals are represented as infinite strings over some finite alphabet Σ. Informally one can say, a real number is computable in TTE if it is possible to approximate it with any desired precision.

TTE is supposed to be a realistic model in the sense that exactly those real numbers and functions are computable in TTE that can be computed with a computer.

Similar to the discrete-valued case a vast theory on computability and complexity has been built upon TTE.

Since TTE strives to be a realistic model, an important task is to compare theoretical claims with practical implementations. Using computers to perform exact computations on real numbers is called exact real arithmetic. There are already quite a few implementations for exact real arithmetic. One that has been proven to be particularly fast and therefore well-suited for practical implementations, is Norbert Müller's C++ framework `iRRAM` [Mül00].

`iRRAM` extends C++ by classes and functions for error-free computations with real numbers. The details of the internal representation are taken care of by the framework. The user can write ordinary C++ extended by a data-type that behaves like real numbers, without thinking about rounding errors.

This thesis presents two case studies in exact real arithmetic, both using the `iRRAM` framework to implement the algorithms.

The first case-study shows how exact real arithmetic can be used to get simpler algorithms for classical problems in numerical analysis. The particular problem considered is the shadowing distance for chaotic dynamic systems, i.e. how close a numerically computed orbit of a map $f : \mathbb{R} \to \mathbb{R}$ is followed by an exact orbit. This problem was originally considered by Hammel, Yorke and Grebogi in 1987 [HYG87]. Since they could not compute real numbers exactly, they used a form of interval arithmetic to bound the exact orbit. However, `iRRAM` can be used to compute the exact orbit which thus yields a simplified version of their algorithm.

The second case-study deals with analytic functions. Analytic functions have been thoroughly studied in real complexity theory as a subset of real functions where many in general computationally hard problems become feasible. To show how well those complexity claims compare with practical implementations, data-types to represent analytic functions have been written. The implementation is meant as an extension to the `iRRAM` framework, that provides user-friendly classes for computations with analytic functions. Empirical evaluation was done on the running time of those classes and compared with the expected running times from the theoretical examination.

2 Theoretical Background

2.1 Real Computability Theory

2.1.1 Classical Computability Theory

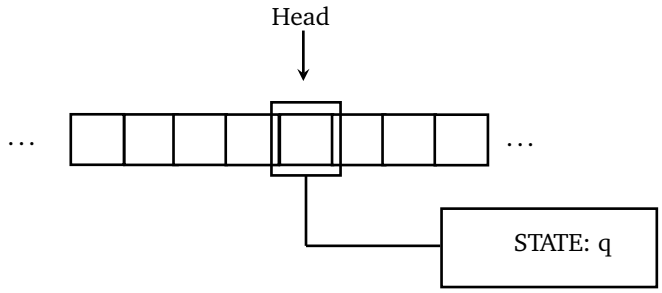


Figure 2.1: A Turing-machine consists of an infinite tape with symbols from Σ. The head is positioned on top of one tape cell.

Since in real computability theory many aspects of classical computability theory are extended, a very brief overview is given in the following section. A detailed introduction can for example be found in [Sch93].

To define computability, the Turing-Machine model is used. The Turing-Machine was invented by Alan Turing in 1936 [Tur36] and can be seen as a simplified mathematical model for a computer.

The machine consists of an infinite tape that is divided into cells. Each cell contains exactly one symbol from a predefined finite alphabet Σ. It further consists of a head that is positioned on top of one cell and can (in one step) read and write the content of the cell and then move one cell left or right on the tape.

The machine is always in one of finitely many predefined states and has a finite instruction table containing instructions of the form "when in state q and reading symbol s, write s' and move head to the left (or to the right)".

A formal definition of the Turing-machine can for example be found in [HMU13].

Definition 2.1.1: *A possibly partial function $f :\subseteq \Sigma^* \to \Sigma^*$ is called **computable** if there exists a Turing-Machine, such that for all $x \in dom(f)$ the machine terminates after finitely many steps with $f(x)$ on its tape and the head is positioned on the first symbol of $f(x)$. For $x \notin dom(f)$ the machine does not terminate.*

Even tough the Turing-Machine model is quite simple, it can be used to simulate every computer algorithm. The widely believed **Church-Turing thesis** even states that anything that is computable in an informal sense can be computed with a Turing-machine.

Many other definitions to formalize the computability notion have been thought of that could all be shown to be equivalent to the Turing-machine.

Sometimes it is more convenient to consider a slightly modified model instead of the standard Turing-machine. Some variations on the model that will be useful later are the following

1. A multi-tape Turing-machine has k independent tapes for some fixed $k \in \mathbb{N}$. Each tape has its own head that can move independently from the others.
2. A non-deterministic Turing-machine can have several possible resulting actions in the same situation. A valid computation can take any of these actions.

The above modifications do not alter the computability notion, i.e. the functions computable will be exactly the same, no matter which kind of Turing-machine is used.

The following definitions for sets also play a very important role in computability theory.

Definition 2.1.2: *A set $A \subseteq \Sigma^*$ is called **decidable**, if its characteristic function is computable.*

Definition 2.1.3: *A set $A \subseteq \Sigma^*$ is called **recursively enumerable** (r.e.) or **computably enumerable** (c.e.) if it is empty or if A is the domain of a computable function.*

2.1.2 Computability of real numbers

The previous section showed how to define computability for functions over finite alphabets $\Sigma^* \to \Sigma^*$. That is enough to define computability for finite structures like natural numbers or graphs, but does not suffice to make any statements on real numbers.

The following section extends the classical notion to uncountable objects such as real or complex numbers, functions or infinite sequences.

There are several non equivalent ways to define computability on such objects. In contrast to the classical case, where the definition using Turing-machines is widely accepted, there is no generally accepted model for real complexity theory.

The model used for this thesis is the so called **Type 2 Theory of Effectivity** (TTE). A detailed overview of this model can for example be found in [Wei00] or [BHW08]. TTE aims to realistically model what is possible to compute with a computer and gives a tool to analyze computability and complexity of real numbers and functions.

Since real numbers are infinite, it is not possible to read or write them in finite time. A first definition can therefore look as the following.

Definition 2.1.4: *A real number is computable if there is a Turing-Machine, that has no input and writes the binary expansion of x on its tape without ever terminating.*

To extend the definition to other uncountable structures, like real or complex functions, a slightly different approach is more convenient.

TTE extends the Turing-Machine model to so called Type-2 Turing-machines. In contrast to an ordinary Turing-machine a type-2 machine can work on infinite strings $s \in \Sigma^w$.

Definition 2.1.5: *A type-2 Turing-machine is a multi-tape Turing-machine with two special tapes, an **input tape** and an **output tape**. There also exists at least one working tape.*

The input tape is read-only, i.e. it is not possible to change a cell on the tape. Further, the head on both input and output tape can not be moved to the left. In particular it is not possible to change a symbol that has been written on the output tape once.

The term Turing-Machine will be used both for classical and type-2 machines, when it is obvious by context which model is meant.

Definition 2.1.6: *A function $F :\subseteq \Sigma^\omega \to \Sigma^\omega$ is called computable if there is a Turing-Machine that for each infinite string $\sigma \in dom(F)$ on its input tape, writes the infinite string $F(\sigma)$ on its output tape.*

To talk about computability over some arbitrary set, it has to be encoded to Σ^ω. Such an encoding can be formalized in the following way:

Definition 2.1.7: *A **representation** of a set X is a partial surjective mapping $\alpha : \Sigma^\omega \to X$.*

*$\bar{\sigma} \in \alpha^{-1}(\sigma)$ is called an **α-name** of σ.*

*$x \in X$ is **α-computable** if it has a decidable α-name.*

In contrast to the countable case, where a canonical encoding is obvious in most cases, finding a good representation is more challenging in the uncountable case.

Different representations can lead to a different computability notion. Thus, it is important to find a representation that leads to a useful and realistic notion of computability.

Some possible representations for real numbers are as follows

1. A ρ_{10}-name of x is the usual decimal expansion of x.
2. A ρ-name of $x \in \mathbb{R}$ is a sequence $a_n \in \mathbb{Z}$ s.t. $|x - a_n| \leq 2^{-n}$
3. A ρ_C-name of $x \in \mathbb{R}$ consists of two sequences rational $(q_n)_{n\in\mathbb{N}}$ and $(\varepsilon_n)_{n\in\mathbb{N}}$, so that $|x_n - q_n| < \varepsilon_n$ and $\lim_{n\to\infty} \varepsilon_n = 0$
4. A $\rho_<$-name enumerates all $q \in \mathbb{Q}$ with $q < x$, similar a $\rho_>$-name can be defined

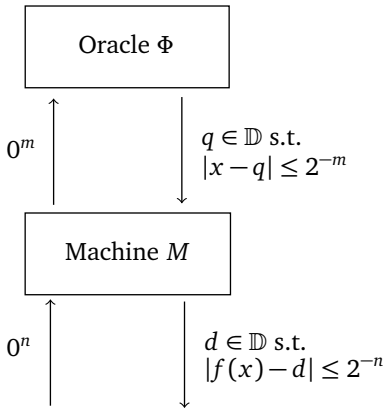


Figure 2.2: Computing a real function with an oracle Turing-machine. With input $n \in \mathbb{N}$ coded in unary, the machine has to return an approximation to $f(x)$ with error bounded by 2^{-n}. For that it can ask the oracle for approximations of x with error bounded by 2^{-m}.

Some of the above notions lead to an equivalent definition of a computable real numbers, but others do not.

In particular the following are equivalent

1. $x \in \mathbb{R}$ is computable in the sense of Definition 2.1.4
2. $x \in \mathbb{R}$ is ρ_{10} computable
3. $x \in \mathbb{R}$ is ρ-computable

A real number will be called **computable** if it is computable in the sense of one of those definitions.

It can easily be seen that there must be non-computable reals, since the number of reals is uncountable, while the number of Type-2 Turing-machines is countable.

The following gives an explicit construction for a non-computable real number

Example 2.1.8: Let $A \subseteq \mathbb{N}$ be any recursively enumerable, but not decidable subset of the natural numbers.

Define the real number x by

$$x := \sum_{n \in A} 2^{-n}.$$

Note, that this is well defined since the series of partial sums is strictly increasing and bounded by 2.

However, x can not be computable since otherwise it would yield a decision procedure for A.

A sequence as the one from the partial sums used to define x, i.e. a sequence that is computable, monotonic, bounded and consists only of rational numbers, but has a non-computable supremum is called **Specker-sequence**.

For practical applications more interesting than just computing single real numbers is computing real functions or even functionals.

For a real valued function f in general input x as well as output $f(x)$ are infinite. Thus, it is not possible to read the entire input or write the entire output in finite time.

Instead, for a function $f : \mathbb{R} \to \mathbb{R}$ to be called computable, it is demanded that the machine can approximate the output arbitrary well. To do that, it can ask for arbitrary well approximations of the input x. An easy way to formalize this concept is the use of oracle Turing-Machines.

Definition 2.1.9: *An **oracle Turing-machine** is a Turing-Machine M with one extra special tape, the **query tape**. Further, the machine is connected to an oracle function Φ. If M is connected to the function Φ it is written as M^Φ.*

*The machine behaves like a usual (type-1) Turing-machine, but has two special states, the **query state** and the **response state**. When the machine enters the query state, the following happens in one time step:*

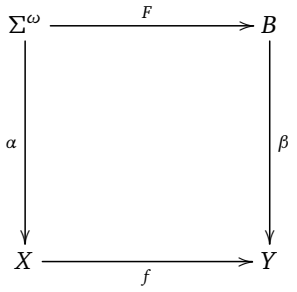


Figure 2.3: Computability with representations

- *The string x on the query tape is replaced by the string $\Phi(x)$*
- *The machine goes into the answer state*
- *The head of the query tape is moved on top of the first symbol of the oracle's answer*

A real function $f : \mathbb{R} \to \mathbb{R}$ is called computable if there is an oracle Turing-machine M, such that M^Φ computes with input $n \in \mathbb{N}$ a dyadic rational number $d \in \mathbb{D}$ such that $|f(x) - d| \leq 2^{-n}$ for each oracle function $\Phi : \mathbb{N} \to \mathbb{D}$ such that for all $m \in \mathbb{N}$ $|\Phi(m) - x| \leq 2^{-m}$.

A more general definition can be achieved by using representations and type-2 Turing Machines.

Definition 2.1.10: *A function $f :\subseteq X \to Y$ is called (α, β)-computable, if there exists a computable function $F :\subseteq \Sigma^\omega \to \Sigma^\omega$ such that $\beta(F(\sigma)) \in f(\alpha(\sigma))$ for all $\sigma \in dom(f \circ \alpha)$.*

The choice of representation is crucial when defining computability, as the following theorem shows.

Theorem 2.1.11: Multiplication is not (ρ_{10}, ρ_{10})-computable.

Thus, the decimal expansion does not seem to be useful to define computable real functions. The same holds for any other base b.

With respect to the Cauchy representation, however, all of the following become computable

1. Arithmetical operations $+, -, x, / :\subseteq \mathbb{R}^2 \to \mathbb{R}$
2. The absolute value function
3. The minimum and maximum functions
4. constant functions with computable constant
5. Projections $\mathbb{R}^k \to \mathbb{R}$
6. polynomials with computable coefficients
7. The functions $\exp, \sin, \cos$
8. The square-root function and the logarithm function

Thus, the Cauchy representation is used as the standard representation to define computability, and the term computable function will be used for functions computable with respect to the Cauchy representation.

Some properties of computable functions are

1. Computable functions map computable reals to computable reals
2. They map computable sequences of real numbers to computable sequences of real numbers
3. They are closed under composition

The following theorem shows, however, that many important functions are not computable. A proof can for example be found in [Wei00].

Theorem 2.1.12: Every computable function $f : \mathbb{R} \to \mathbb{R}$ is continuous.

In particular, tests of the form $\mathbb{R} \to \{0,1\}$ are non-computable if their value is not constant.

A relaxation can be achieved by allowing functions to be multi-valued.

Definition 2.1.13: *A multi-valued function $f :\subseteq X \rightrightarrows Y$ is another name for a relation $f \subseteq X \times Y$.*

A multi-valued function is (ρ_X, ρ_Y) computable, if there is a computable (single valued) function $F :\subseteq \Sigma^\omega \to \Sigma^\omega$ such that for all $\sigma \in dom(f \circ \rho_X)$, $\rho_Y(F(\sigma)) \in f(\rho_X(\sigma))$.

In other words, a multi-valued function can have several values for the same point and only one of these values has to be computed. This definition allows the computation to be non-deterministic in some sense.

In practical applications, a multi-valued functions often has the form of a non-computable single-valued function that is allowed to give slightly wrong results close to the points of non-computability. That is, the result may also depend on the given representation, and not on the underlying function alone.

Example 2.1.14: The floor function $x \mapsto \lfloor x \rfloor$ is not computable due to continuity reasons.

However, the multi-valued function that computes $\lfloor x \rfloor$ if the distance of x to an integer is bigger than 2^{-k} and computes $\lfloor x \rfloor$ or $\lfloor x - 1 \rfloor$ otherwise is computable.

2.1.3 Computability of real operators and functionals

A common task in numerical analysis is providing answers to questions of the following form: given a real function, what is its maximum on $[0,1]$. The input to such a question is a real function.

A real operator maps functions $\mathbb{R} \to \mathbb{R}$ to functions $\mathbb{R} \to \mathbb{R}$ and a functional maps functions $\mathbb{R} \to \mathbb{R}$ to real numbers $\mathbb{R}$. To talk about computability of operators and functionals, one first has to fix the space that should be represented.

Continuous functions on a compact subset $X \subseteq R^d$ can be uniformly approximated by polynomials arbitrarily close. A possible representation for real valued functions is thus given by the following definition

Definition 2.1.15: *A $[\rho^d \to \rho]$-name of a function $f \in C([0,1]^d, \mathbb{R})$ is given by a sequence $P_n \in \mathbb{D}[x_1, \ldots, x_d]$ of polynomials (i.e. degree and list of coefficients), such that $|f - P_n|_\infty < 2^{-n}$*

Computability of Operators and functionals operating on functions in $C([0,1])^d$ can be defined with respect to this representation analogously to the definitions in the last section.

With this notion the following holds

Theorem 2.1.16: The integration operator

$$I : C[0,1] \to C[0,1], f \to \left(x \to \int_0^x f(t)dt \right)$$

is computable.

Unfortunately, the same does not hold for the derivative

Theorem 2.1.17 (Myhill 1971): There is a computable function $f : [0,1] \to \mathbb{R}$ with continuous but uncomputable derivative.

However, if the function is two times continuously differentiable, the following holds

Theorem 2.1.18: Every computable function $f \in C^2([0,1])$ has a computable derivative.

This does not imply that the operator is computable, but only that a computable derivative exists. In fact, the derivative operator is still uncomputable when restricting the input to functions in $C^2([0,1])$ unless some additional discrete information is supplied.

This difference will be further explained in the next section.

2.1.4 Uniformity and Non-Uniformity

In computability theory, one has to distinguish two modes of computability, **uniform** and **non-uniform**.

For non-uniform computability it suffices that for every input, there is an algorithm that computes the output. The algorithm may however depend on the input in a non-computable way.

In contrast, a problem is uniformly computable only if there is **one** algorithm, that computes the output for every valid input.

An example is the effective version of the Intermediate Value Theorem, i.e. the task to find a zero of a computable function $f : [0,1] \to \mathbb{R}$ with $f(0) \cdot f(1) < 0$.

If the set of zeros does not contain an interval it is possible to compute a zero. On the other hand, an interval always contains a computable number, thus in this case there also is a computable zero. However, it can be shown that there is no Turing-machine that can decide which case holds and compute a zero from a given function alone.

In practical applications it is usually desired to have uniform algorithms. However, in many cases there does not exist a uniform algorithm in the standard representation of the input.

One way to deal with this problem is the use of multi-valued functions.

Another way is to give the algorithm some additional information on the input. In many cases it is possible to turn non-uniform algorithms into uniform ones by extending the input by some additional discrete information that can not be computed from the input only.

For example, the floor function can be computed uniformly when given one additional bit containing the information if the input is an integer or not.

2.2 Real Complexity Theory

2.2.1 Classical Complexity Theory

In contrast to computability theory, complexity theory deals with the question how many resources (in form of e.g. time or space) are needed to compute a computable function. A detailed introduction to discrete complexity theory can for example be found in [AB09].

Again, the Turing-Machine model is used to describe the computations

Definition 2.2.1: *For a given Turing Machine M and $w \in \Sigma^*$, $time_M(w)$ denotes the number of head movements the Turing Machine on input w executes before it terminates.*

For $n \in \mathbb{N}$ define $time_M(n) = \max\{time_M(w) \mid w \in \Sigma^n$ and M terminates on input $w\}$.

Space constraints can be defined similarly.

In most cases it is not important to compute the exact running time, but one rather wants to approximate how the algorithm will behave when the input is getting larger.

The standard way to compare the asymptotic running time of algorithms is the use of the O-notation.

Definition 2.2.2: *For functions $f, g : \mathbb{N} \to \mathbb{N}$ one writes $f \in O(g(n))$, if there are constants $M \in \mathbb{R}$, $n_0 \in \mathbb{N}$, such that $f(n) \leq M \cdot g(n)$ for all $n > n_0$.*

Thus, for a given algorithm a way to measure its complexity is giving an upper bound on its worst case running time for inputs of length n in terms of the O-notation.

However, complexity theory is not only about the complexity of specific algorithms, but also deals with the complexity of problems. That is, trying to classify the complexity of **any** algorithm to decide some subset of Σ^*.

The following gives a classifications of problems depending on the running time for algorithms solving those problems.

Definition 2.2.3: *For a function $t : \mathbb{N} \to \mathbb{N}$ let*

$$DTIME(t) = \{A \subseteq \Sigma^* \mid \text{There is a Turing-machine } M \text{ deciding } A \text{ with } time_M(n) \in O(t(n))\}$$

$NTIME(t)$ describes the same for the case of M being a non-deterministic Turing-machine.

Similarly, DSPACE can be defined for the space.

The above definitions suffice to define the most important complexity classes

Definition 2.2.4: *The complexity classes P, NP, PSPACE and EXPTIME are defined as follows:*

$$\begin{aligned} P &:= \bigcup_{k \in N} DTIME(n^k) \\ NP &:= \bigcup_{k \in N} NTIME(n^k) \\ PSPACE &:= \bigcup_{n \in N} DSPACE(n^k) \\ EXPTIME &:= \bigcup_{k \in N} DTIME(2^{n^k}) \end{aligned}$$

It holds $P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$. It is $P \neq EXPTIME$ while for all other inclusions it is not known if equality holds. Most researchers in complexity theory, however, believe that equality does not hold for any of the inclusions.

Another way to characterize the important class NP is the following

Theorem 2.2.5: $A \subseteq \Sigma^*$ is in NP if and only if there are polynomials p and q and a deterministic Turing-Machine M such that for all $x \in \Sigma^*$ with length n, there is an $y \in \Sigma^*$ with length bounded by $q(n)$ such that $x \in A$ iff $M(\langle x, y \rangle = 1$ and $x \notin A$ iff $M(\langle x, y \rangle = 0$. Further M's running time is bounded by $p(n)$ for all such inputs $\langle x, y \rangle$.

Whether $P = NP$ is one of the Millennium problems and one of the biggest unsolved problems in Computer Science.

One can also consider function problems instead of decision problems leading to a dual hierarchy of complexity classes.

Definition 2.2.6: *A function $F : \Sigma^* \rightrightarrows \Sigma^*$ is computed by a Turing-Machine if the machine writes on every $x \in \Sigma^*$ an output $z \in F(x)$ or decides that no such output exists.*

The definitions of time- and space complexity can easily be adopted to functional problems.

The classes FP and $FPSPACE$ are defined as the class of polynomial time resp. space computable functions.

The class $\#P$ is defined as the class of functions that give the number of solutions to a problem in NP.

It holds $FP \subseteq \#P \subseteq FPSPACE$ and if $FP = \#P$ would hold, it would follow that $P = NP$.

Problems that are known to be in P are often considered as the feasible ones, for which it is possible to find an efficient algorithm.

Since most of the complexity classes are not known to be distinct, they do not suffice to classify problems by their difficulty.

To compare the difficulty of problems, reductions can be used.

Definition 2.2.7: *$A \subset \Sigma^*$ is **polynomial time reducible** to $B \subseteq \Sigma^*$ ($A \leq_P B$), if there is a polynomial time computable function $f : \Sigma^* \to \Sigma^*$, such that*

$$w \in A \Leftrightarrow f(w) \in B.$$

Many other types of reductions exist, e.g. to compare complexity classes lower then P, but they are not needed in this thesis and therefore omitted.

The "hardest" problems in a complexity class are called complete for this class

Definition 2.2.8: *A set $A \subseteq \Sigma^*$ is called $\mathscr{C}$-**hard** (w.r.t. $\leq_P$) for a complexity class $\mathscr{C}$, if for all $B \in \mathscr{C}$ $B \leq_P A$.*

*A is called $\mathscr{C}$-**complete** (w.r.t. $\leq_P$), if A is $\mathscr{C} - hard$ and $A \in \mathscr{C}$.*

If one would show for a single $\mathscr{C}$-complete problem that it is in P, it would follow that all problems in $\mathscr{C}$ are in P. Consequently a way to show that there is most likely no efficient algorithm to a problem, is to show that the problem is NP-hard.

For many problems it could be shown that they are NP-complete. Some of the most important ones can for example be found in [GJ79].

2.2.2 Complexity Theory on real numbers

In classical complexity theory the time complexity is usually measured in terms of the input size. However, when considering continuous problems, the input is an infinite string and thus can not be used to measure the running time of an algorithm.

In numerical computations an important parameter is the desired output precision. Thus, it seems reasonable to analyze the running time as a function depending on the desired output precision.

This leads to the following definition for the complexity of a real number

Definition 2.2.9: *A real number $x \in \mathbb{R}$ is computable in time t if there is a Turing Machine M that with input $n \in \mathbb{N}$ in binary, outputs the binary expansion of a dyadic rational number d with $|x - d| \leq 2^{-n}$.*

Again this notion can be generalized by using representations and Type-2 Turing machines, i.e. by considering the time that is needed to write the first n symbols of the name of the output.

One has to be careful, however, since representations that are computationally equivalent, do not necessary lead to the same complexity bounds. For example, one can construct arbitrarily long Cauchy names for any rational number. Thus, it is not even possible to define the complexity of a real number with respect to its Cauchy name.

An often used representation for complexity considerations is the signed digit representation.

Definition 2.2.10: *The **signed digit representation** ρ_{sd} is defined as follows:*

A ρ_{sd}-name of a real number x is a sequence

$$a_n a_{n-1} \dots a_0 . a_{-1} a_{-2} \dots \text{ with } a_i \in \{-1, 0, 1\}, n \geq -1, a_n \neq 0, a_n + a_{n-1} \neq 0$$

and

$$x = \sum_{i=n}^{-\infty} a_i \cdot 2^i$$

As with the binary expansion, the number digits after the binary point correspond to the precision.

It can be shown that the signed digit representation is computably equivalent to the Cauchy representation.

When extending the complexity notion to functions, apart from the output precision, an additional parameter might be interesting: the precision of the input needed to compute the output up to the demanded precision.

This leads to the following definition

Definition 2.2.11: *A function $f :\subseteq \mathbb{R} \to \mathbb{R}$ is computed on a compact set $K \subseteq dom(f)$ in time t with lookahead l if for all ρ_{sd}-names of elements $x \in K$ by a Turing Machine M if M computes a ρ_{sd}-name of $f(x)$ and after t steps and with reading at most l symbols of the input string outputs the n-th symbol of the output.*

The above complexity notion really only makes sense on a compact set, since otherwise the machine could already spend arbitrarily much time on just reading the input left from the binary point.

An alternative approach to define complexity of functions is using Oracle Turing Machines which have polynomial time bounded running time.

With the above notion computable real numbers and functions can be classified into complexity classes analogously to the discrete case.

An important class for practical purposes is the class of polynomial time computable numbers and functions, since in many cases they correspond to the ones that can be computed efficiently with a computer.

An important property of polynomial time computable reals is given in the following theorem.

Theorem 2.2.12 (Ko and Friedman): The set of polynomial time computable real numbers forms a real algebraically closed field.

Example 2.2.13: The following real valued functions are polynomial time computable on a compact set K of real numbers.

1. Addition, Subtraction and Multiplication as functions $K \times K \to \mathbb{R}$.
2. The function $x \mapsto \frac{1}{x}$ if $0 \notin K$.
3. The functions $\exp, \sin, \cos$ as functions $K \to \mathbb{R}$.

2.2.3 Complexity of Operators

Since most interesting function spaces such as $C[0,1]$ are not locally compact, there is no straight forward way to generalize the above definitions to get a uniform notion of complexity for operators.

In fact, the following holds

Theorem 2.2.14: Even restricted to continuous functions $f : [0,1] \to [0,1]$, the evaluation operator $x \to f(x)$ is not computable within time uniformly bounded in terms of the output error only.

A different approach to study the complexity of operators and functionals is followed by Ko and Friedman [Ko91]. Instead of trying to define a complexity notion for a functional F, they study the time complexity of the real valued function $F(f)$ where f is a polynomial time computable function.

This method leads to many connections between discrete complexity classes and numerical problems.

One example of such a connection is given in the following theorem which is due to Friedman. A proof can be found in [Ko91].

Theorem 2.2.15: The following are equivalent

1. $FP = \#P$
2. For every polynomial time computable function $f : [0,1] \to \mathbb{R}$, the function $g : [0,1] \to \mathbb{R}$, defined by $g(x) = \int_0^x f(t)dt$ is polynomial time computable.

Thus, one can say that integration of polynomial time computable functions is at least as hard as solving a problem in $\#P$. In that sense, Integration can be called $\#P$-hard.

3 Computing with Reals

3.1 Floating Point Arithmetic

The traditional way to perform computations with real numbers on a computer is the use of **floating point arithmetic**.

The floating point representation of a real number is a bit-string with a fixed length.

Numbers are represented the following way:

Definition 3.1.1: *A floating point representation with base b and precision p is a string $\pm d_0.d_1 \dots d_{p-1} \times b^e$ with $0 \leq d_i < b$. It represents the number*

$$\pm b^e \cdot \sum_{i=0}^{p-1} d_i \beta^{-i}$$

$\pm d_0.d_1 \dots d_{p-1}$ is called the Mantissa and e the exponent.

The smallest and largest exponent allowed are e_min and e_max. A floating point number can be encoded in

$$\lceil \log_2(e_{max} - e_{min} + 1) \rceil + \lceil p \cdot \log_2(b) \rceil + 1$$

bits.

Information on floating point arithmetic can for example be found in [Gol91].

If a real number does not fit into the above representation, it has to be rounded appropriately.

Rounding also happens when an operation (e.g. multiplication) is applied and the result does not fit into the finite representation anymore.

With the above definition the floating point representation of a number is not unique. To guarantee uniqueness it is usually required that the representation is normalized, i.e. that $d_0 \neq 0$.

Of course, 0 can not be represented in this form, so often a special representation is chosen for 0.

Doing real number computations using floating point numbers will in most cases lead to rounding errors.

One can distinguish two cases of errors, the **absolute error**, the distance between the real result and the computed one, and the **relative error**, the absolute error divided by the real result.

Floating point arithmetic is not necessarily associative, e.g. in many programming languages for $x := 1e30$, $y := -1e30$ and $z := 1$ the expression $(x + y) + z$ evaluates to 1 while $x + (y + z)$ evaluates to 0.

Floating point types are often built into hardware, and thus computations can be performed extremely quickly.

3.1.1 The IEEE Standard

Floating point representations for binary base are standardized by IEEE 754 [iee08]. This standard is followed by almost all modern computers.

The standard defines four different precisions: single, single-extended, double and double-extended. The details can be seen in Table 3.1.

Type	p	e_{max}	e_{min}	exponent width (bits)	format width (bits)
Single	24	+127	-126	8	32
Single-Extended	32	+1023	≤ -1022	≤ 11	43
Double	53	+1023	-1022	11	64
Double-Extended	64	> 16383	≤ -16382	15	79

Table 3.1: Parameters for the different types in the IEEE 754 standard

Apart from the numbers definable in the way of Definition 3.1.1, the standard also introduces special quantities $-0, +0, \text{inf}, +\text{inf}$ and NAN (not a number).

It is required that the result of addition, subtraction, multiplication and division is exactly rounded, i.e. the operation has to be performed exactly and then rounded afterwards.

The rules for rounding are also defined in the standard. It is required to offer at least the following four rounding modes.

1. **Round to nearest** All results are rounded to the nearest representable value. If a result is in the middle of two representable values, the one where the lowest bit is zero is chosen.
2. **Round toward** ∞ All results are rounded to the smallest representable value larger than the result.
3. **Round toward** $-\infty$ All results are rounded to the largest representable value smaller than the result.
4. **Round toward** 0 If the result is negative, it is rounded up, otherwise rounded down.

The user should be able to select which of the rounding modes should be applied.

3.2 Arbitrary-Precision Arithmetic

The accuracy that can be obtained by using hardware floating point types sometimes does not suffice.

When precise results are more important than speed of computation, Arbitrary-precision arithmetic can be used.

Arbitrary precision means that the user is able to choose the precision with which computations are performed (usually only limited by the available main memory).

Usually an arbitrary-precision number is implemented as a list or array of built-in data-structures together with algorithms that perform arithmetic operations on this representation.

Many libraries for arbitrary-precision arithmetic exist for nearly all modern programming languages.

Examples are the GNU MPFR Library for `C/C++` [FHL+07], The Boost Multiprecision Library for `C++` [MK12] or `mpmath` for `Python` [J+10].

The floating point types defined by such libraries can have a user-defined length. Thus, the precision can be arbitrarily good, but will still be fixed during the program flow. Rounding errors still occur and in most implementations it is not possible to get bounds on the error from the computation result.

The user usually has to perform additional analysis on the problem, to choose a computation precision that yields exact enough results.

3.3 Interval Arithmetic

When using floating point arithmetics several real numbers are mapped to the same floating point representation. This leads to numbers not being represented exactly in almost all cases.

Further, a single floating point number does not contain any information about its accuracy, and so there is no way to find out if the result of a computation can be trusted or not.

Even if exact computation is not necessary, often one wants to have at least a bound on the error of the result to decide whether it is useful or not.

A rather easy way to get error bounds is using interval arithmetic [Kea96]. Interval arithmetic means that instead of making computations on isolated single real numbers whole intervals are used.

A real number $r \in \mathbb{R}$ is represented by an interval $x := [x_l, x_r]$, so that $r \in x$. When performing operations (addition, multiplication, etc.) a new interval, containing each possible result that can be obtained from performing the operation on the original intervals, has to be computed. The advantage to floating point arithmetics is that the length of the interval gives a bound on the rounding error, thus making it possible to guarantee the accuracy of computed results.

The fundamental property of interval arithmetic is the inclusion property:

Every $f : \mathbb{R} \to \mathbb{R}$ can be extended to a function $\bar{f}$ on intervals, such that for all $x \in [a, b]$, $f(x) \in \bar{f}[a, b]$.

The interval extension of a function is not unique, but usually one tries to find a mapping that is as tight as possible, i.e. the length of the resulting interval should be as small as possible.

Basic arithmetic operations can be easily defined on intervals:

Theorem 3.3.1: Let $x = [x_l, x_r]$ and $y = [y_l, y_r]$ then it holds

$$\begin{aligned} x + y &= [x_l + y_l, x_r + y_r] && (3.1)\\ x - y &= [x_l - y_r, x_r + y_l] && (3.2)\\ x \times y &= [\min(x_l y_l, x_l y_r, x_r y_l, x_r y_r), \max(x_l y_l, x_l y_r, x_r y_l, x_r y_r)] && (3.3)\\ \frac{1}{x} &= \left[\frac{1}{x_r}, \frac{1}{x_l}\right] \text{ if } x_l > 0 \text{ or } x_r < 0 && (3.4)\\ \frac{x}{y} &= x \times \frac{1}{y} && (3.5) \end{aligned}$$

It is easy to see, that addition and multiplication are associative and commutative. However, the following example shows that the distributive law does not necessarily hold

Example 3.3.2:

$$[-1,1] \times ([-1,0] + [3,4]) = [-1,1] \times [2,4] = [-4,4]$$
$$[-1,1] \times [-1,0] + [-1,1] \times [3,4] = [-1,1] + [-4,4] = [-5,5]$$

Thus, the length of the resulting interval is not independent of the way the computation is done.

The next theorem provides a simple way to find interval extensions of many functions.

Theorem 3.3.3: For every monotonic function $f : \mathbb{R} \to \mathbb{R}$ it holds

$$f(x) = [\min(f(x_l), f(x_r)), \max(f(x_l), f(x_r))]$$

In particular this can be used to find interval extensions of piecewise monotonic functions, such as

$$x^n = \begin{cases} [x_l^n, x_r^n] & \text{if } n \text{ is even or } x_l \geq 0 \\ [x_r^n, x_l^n] & \text{if n is odd and } x_r \leq 0 \\ [0, \max(x_r^n, x_l^n)] & \text{otherwise.} \end{cases} \quad (3.6)$$

In a similar way, interval version for $\exp, \log, \sin, \cos$ and other elementary functions can be found that can then be used to find the interval versions of more complicated functions.

The strength of interval arithmetic is that it can easily be implemented even when the above operations on the interval endpoints are not computed exactly but finite approximations are used.

This does, however, only work if outward rounding is used, i.e. after every operation the left point of the interval has to be rounded down and the right point rounded up.

Since IEEE 754 requires that the user is able to specify the rounding mode, interval arithmetic can be implemented using standard floating point numbers.

Note that the length of the interval can increase quickly.

There are many implementations of interval arithmetic in different programming languages, e.g. for `C++` the boost interval arithmetic library can be used.

When using floating point arithmetic, the minimal interval length is bounded by the machine epsilon. Interval arithmetic can, however, also be combined with arbitrary-precision arithmetic to get arbitrarily small intervals, leading to arbitrarily exact computations with the possibility to bound the rounding error after the computation.

3.4 Symbolic Computation

Symbolic computation means that mathematical expressions (i.e. formulas consisting of numbers, functions, variables and mathematical constants) are manipulated algebraically.

Software that performs symbolic computations is called a **Computer Algebra System** (CAS).

Some example of Computer Algebra Systems include Maple, Mathematica or Sage.

In a CAS, at each step of the computation the numbers are represented exactly. However, symbolic computation is very limited, because many problems are either very difficult to be solved symbolically or do not even have a symbolic solution at all.

Also, even if the problem can be solved, an exact symbolic solution might be very long and complex, thus making it less useful on its own.

Therefore, symbolic methods are often combined with numerical methods to get an approximate result in the end.

3.5 Exact Real Arithmetic

Using interval arithmetic or arbitrary precision arithmetic, the precision is set in the beginning and rounding and truncation errors still accumulate. Without previous careful analysis, there is no guarantee that the result's precision will suffice.

In contrast, the goal of exact real arithmetic is to perform computations on real numbers without accumulating errors.

In practice, that means that on demand a number can be printed up to any desired precision. More precisely, the user does not specify the precision the numbers are represented with in the beginning, but gives a desired precision **for the answer** of the computation.

The implementation has to take care of the steps necessary to achieve the precision. For the user, it should seem like real numbers are manipulated exactly, while of course internally the implementation still works on finite approximations.

There are several approaches to exact real arithmetic. For example

1. **Signed-digit Streams**: A real number is represented as an infinite string stream. This is very close to the definition with type-2 machines.
2. **Continued Fractions**: A real number is represented by its continued fraction representation. Rational numbers have a finite representation, while irrational numbers have an infinite representation.
3. **Directed Acyclic Graphs (DAGs)**: A real number is given by an algorithm that computes an approximation to the number with a desired error bound. Arithmetic expressions are expressed as graphs with real number leaves.

Since the approach followed in this thesis is the DAG approach, it is the only one that will be described in more detail.

3.5.1 DAGs

An arithmetic expression over real numbers can be expressed by a directed acyclic graph (DAG) where the leaf nodes are real numbers and the inner nodes are operations on real numbers.

The inner nodes contain the information how precise the inputs to the operation have to be to guarantee a given output precision. Thus, the needed precision can be propagated down to the leaf nodes, and the algorithms to approximate the real numbers can be invoked.

The term **evaluating** is used to describe the process of getting an approximation to the real number represented by a DAG with a desired output precision.

There are essentially two ways to evaluate a DAG:

1. **Top-down evaluation**: The desired precision is computed from the top node down to the leaves. That is, a node contains the information how precisely it needs the input given by its children to compute the output with the

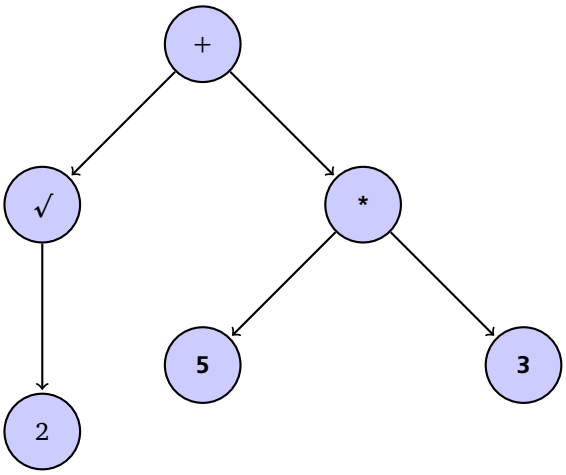


Figure 3.1: The DAG representation for the expression $\sqrt{2}+(2+5*3)$

necessary precision. The top node is given the desired output precision, and it then asks its children to provide this input.

2. **Bottom-up evaluation**: the computation is started with a fixed precision in the leaf nodes. It is kept track of the errors when going upward in the DAG (this can be done e.g. by using interval arithmetic). At the top the error bound is compared to the desired output precision. If the error is too large, the computation is restarted with higher precision.

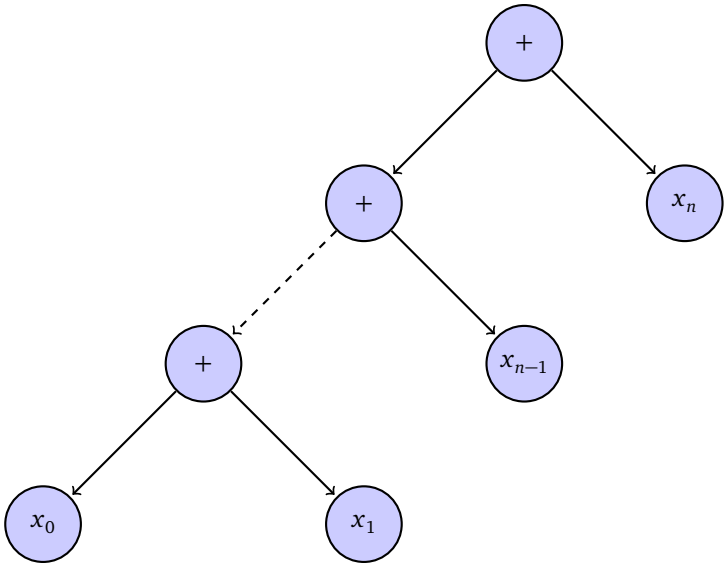


Figure 3.2: The DAG created when the expression $\sum_{i=0}^{n} x_i$ is computed iteratively

While the top-down approach seems to evade unnecessary recomputations, the precision needed is usually largely overestimated.

For example, to compute additions of two numbers up to precision 2^{-n}, it suffices to have the inputs with precision $2^{-(n+1)}$. Now, if the sum $x := \sum_{i=0}^{1000} x_i$ is computed iteratively (e.g. in a for-loop), a DAG as in Figure 3.2 will be created. Thus, x_0 has to be computed with precision $2^{-(n+1000)}$ while actually computing all input values with error less than $2^{-(n+10)}$ would suffice to guarantee error of the sum of less than 2^{-n}. This makes the bottom-up approach more feasible in many cases.

In general, the DAG approach has the problem that saving the DAG needs a huge amount of memory. Arithmetic operations in loops, that are executed many times, quickly grow the created DAG and thus may exceed the computer's main memory.

The pure DAG approach is therefore not applicable for many numerical algorithms.

3.6 iRRAM

iRRAM is a C++ framework for exact real arithmetic developed by Norbert Müller.

The case studies in the next chapters were all implemented using the iRRAM framework. This chapter gives therefore a short introduction to this framework.

iRRAM extends C++ by a data-type REAL for error free computations with real numbers. For the user, an object of type REAL behaves like a real number that can be manipulated without any rounding errors. The framework takes care of all details necessary to finitely represent real numbers internally, and in most cases this internal representation will be invisible for the user.

3.6.1 Real Number representation

iRRAM's approach is similar to the bottom-up DAG approach described in Section 3.5.

There is one important difference tough: The DAGs are stored only implicitly. Instead, there is only enough information saved, so that the whole computation can be repeated from the beginning. If the precision at some point of the program does not suffice, the whole program is restarted with higher precision. This drastically reduces the memory needed for the computation.

For making error analysis, iRRAM uses a simplified form of interval arithmetic.

At every time during the program execution, the internal representation of a real number is always a **single** interval. More precisely, a real number $x \in \mathbb{R}$ is represented by a pair (d, e) such that $x \in [d - e, d + e]$.

d is saved as a multiple precision number and e consists of two `longs` p, z such that $e = p \cdot 2^p$.

The internal representation of a REAL can change during the run of the program.

Whenever the precision of some REAL is not sufficient at some point, the whole computation is restarted. This is, when the representations of the reals change to admit higher precision.

This process of restarting the whole computation with higher precision is called **iteration** in iRRAM's terminology. Since a reiteration is only done when higher precision is needed, there are only few operations, such as comparisons or output, that can trigger such a reiteration.

An iRRAM program will usually do several iterations of the computation, until the desired output precision is reached.

3.6.2 Overview

The following gives an overview over the most important classes and functions provided by the iRRAM framework. For a more detailed overview see [Mül00].

iRRAM provides some classes for discrete valued structures, in particular the class INTEGER for infinite sized integers, the class RATIONAL for rational numbers and the class DYADIC for dyadic rational numbers.

The most important class is REAL, the class for error-free computations with real numbers.

An object of type REAL can be constructed from `int`, `char*`, `double`, DYADIC or another REAL.

Arithmetic operations can be applied on REALs by using the overloaded operators +, -, *, /.

Many functions on REALs are already included in the iRRAM package, e.g. `abs(x)`, `power(x, n)`, `exp(x)`, `log(x)`, trigonometric functions and their inverse and many more.

The operators <, <=, ==, >=, > are also overloaded. Note, however, that comparison operators have to be used carefully, as will be explained in the next section.

iRRAM further provides classes for complex numbers and real matrices.

An iRRAM program is written in usual C++ extended by the data-types provided by a framework. However, a few things have to be kept in mind when writing a program

$a \vert\vert b$	0	1	$\perp$
0	0	1	$\perp$
1	1	1	1
$\perp$	$\perp$	1	$\perp$

$a \&\& b$	0	1	$\perp$
0	0	0	0
1	0	1	$\perp$
$\perp$	0	$\perp$	$\perp$

$!a$	
0	1
1	0
$\perp$	$\perp$

Table 3.2: The semantics of the `LAZY_BOOLEAN` data-type

1. **compute function** The `iRRAM` framework overwrites the `C++` `main` function, so it can not be written by the user. Instead it is replaced by a function `void iRRAM :: compute()` . The main part of the program should be written inside this function.
2. **Input and Output** The input and output streams `cin` and `cout` are overloaded with `iRRAM`'s own versions, that can read and write real numbers. The precision of the output can be adjusted with the parameterized manipulator `setwidth` that sets the number of digits that are written. Alternatively the function `rwrite(REAL& r, int prec)` can be used to write a real number with a given number of decimals.
3. **Global variables** Since `iRRAM` only reiterates the code executed inside the compute function, global variables will not be part of the reiteration. Therefore, `REAL` variables should never be defined globally.

3.6.3 Multivalued Functions

TTE can be used as a theoretical model to reason about `iRRAM` programs.

Thus, the limitations seen in Chapter 2.1 also hold for `iRRAM` programs. In particular, every function computable with `iRRAM` necessarily has to be continuous.

That also implies, that comparisons and tests for equality of `REAL`s are not computable.

Indeed, for the simplified representation a comparison of two numbers can be decided if and only if the intervals of the two representations do not overlap.

As long as the two intervals overlap, `iRRAM` has to assume that the intervals are to large to make a decision yet, and has to reiterate to get smaller intervals. But if the numbers are equal, the intervals will always overlap, no matter how small they are, leading to an infinite loop of iterations. The reiteration is realized by the `LAZY_BOOLEAN` data-type. `LAZY_BOOLEAN` extends the `boolean` type to ternary logic with values $T, F, \perp$. The semantics of `LAZY_BOOLEAN` can be found in Table 3.2.

The informal meaning of $\perp$ is, that the precision is not yet sufficient to make a decision. When casting to `bool` $\perp$ leads to a reiteration.

Not being able to make comparisons might first seem like a huge disadvantage, but note that in numerical practice it is also strongly advised to never test floating point numbers on equality. The `gnu gcc` compiler even has a warning option on that [Fre05]:

```
-Wfloat-equal
  Warn if floating-point values are used in equality comparisons.

  The idea behind this is that sometimes it is convenient (for the programmer) to consider
  floating-point values as approximations to infinitely precise real numbers. [...]
  Instead of testing for equality, you should check to see whether the two values
  have ranges that overlap; and this is done with the relational operators, so equality
  comparisons are probably mistaken.
```

In contrast to that `iRRAM` has a semantically correct way to handle comparisons and other cases of non-continuity: multi-valued functions.

In `iRRAM` a multi-valued function can have multiple valid outputs for the same input and return one of the valid outputs. For the user the choice of output can seem indeterministic. Of course, the output will be deterministic in the underlying finite representation of the number, but the representation might differ even when the real number represented is the same.

iRRAM already has some multi-valued functions included, e.g. the function

$$bound(x,k) = \begin{cases} \texttt{true} & \text{if } |x| \le 2^{k-2} \\ \texttt{true or false} & \text{if } 2^k \ge |x| > 2^{k-2} \\ \texttt{false} & \text{if } |x| > 2^k. \end{cases}$$

can be used for multi-valued comparisons.

Another example is the function `approx(const REAL& x, const long p)` that returns a DYADIC approximation with error less than 2^{-k} to x.

One way to construct multi-valued functions without accessing the internal representation of the REALs is to use LAZY_BOOLEAN together with the choose-function. choose is a multi-valued function that takes up to 6 lazy booleans and returns the index of one that is evaluated to T or 0 if all evaluate to F (leading to reiteration as long as neither of the two cases hold).

The powerful feature to have multi-valued functions introduces, however, a new problem.

A multi-valued function can have different results for different representations of the same real number. Since the internal representation of a real number changes when the computation is reiterated, a reiteration could lead to a completely different result of the multi-valued function. Thus, the program flow could change from one iteration to another, leading to unexpected results.

To prevent that from happening, the result of a multi-valued function is always saved in the so called **multi-value cache** and read from there when reiterating. It is therefore guaranteed that the program flow stays the same in all iterations.

Saving everything in the cache will, however, increase the needed memory. Multi-valued functions should only be used rarely and one has to be especially careful when they are used inside of loops.

3.6.4 Limits

iRRAM also has the feature to generate new user-defined REALs by using a special limit operator. There are three types of limits

1. **Simple limits**:
   This limit has the form

   ```
   REAL limit(REAL a(long, const REAL&), const REAL& x).
   ```

   The limit operator takes a sequence of real functions converging to a single-valued function $f : \mathbb{R} \to \mathbb{R}$ rapidly, i.e. such that

   $$|a_p(x) - f(x)| \le 2^p \text{ for all } x \in dom(f)$$

   and returns the value of $f(x)$ as a REAL.

   There exist similar functions to compute limits of sequences of functions with 0 or 2 input variables.

2. **Lipschitz limits**
   The limit computation can be improved if a Lipschitz bound on the limit function f is known, i.e. one has an l such that

   $$|f(x) - f(y)| \le 2^l |x - y| \text{ for } x, y in dom(f).$$

   Then for $x \in [d-e, d+e]$

   $$|a_p(d) - f(x)| \le |a_p(d) - f(d)| + |f(d) - f(x)| \le 2^p + 2^l \cdot e$$

   Thus for large enough p (i.e. such that $2^p > 2^l \cdot e$)

   $$|a_p(d) - f(x)| \le 2^{p+1}$$

   To use this in iRRAM there is a function

   ```
   REAL limit_lip(REAL a(long, const REAL&), long lip, const REAL& x).
   ```

3. **Multi-valued limits** The multi-valued limit-operator has the form

```
REAL limit(REAL f(int prec, int* choice, const REAL&), const REAL&)
```

The parameter choice selects a subset of values that are possible for the limit of f. 0 means that all values are allowed, and changing the value of choice corresponds to a reduction of the possible values. The choice parameter will be saved in the multi-value cache.

3.6.5 Access to the underlying representation

Usually the user is expected to see the data-type REAL as a kind of black-box that makes error-free computations with real numbers possible. The framework should take care of all the underlying representations and the finite representation should be invisible to the user.

In rare cases it may however be helpful, to access and change the underlying representation of REALs. This is also possible in iRRAM.

The error of a REAL has the following type

```
struct sizetype { SIZETYPEMANTISSA mantissa; SIZETYPEEXPONENT exponent; }
```

where SIZETYPEMANTISSA is unsigned int and SIZETYPEEXPONENT is int.

An object of type REAL has the two member functions

```
void seterror (sizetype error);
void geterror (sizetype& error) const;
```

They can be used to get and set the error.

The information about the current iteration can also be accessed. The most notable variable is ACTUAL_STACK.ACTUAL_PREC that contains the precision bound for the current iteration.

a	behavior
$[0,1]$	Stabilizes at 0
$(1,3]$	Stabilizes at $\frac{a-1}{r}$
$(3,\approx 3.544]$	oscillates between 2, 4, 8, ... values.
$(\approx 3.544,4]$	For almost all initial points no oscillation with finite period. Small changes in the initial point yield large differences over the iterations.
$(4,\infty)$	The values eventually leave the interval [0,1] and diverge for almost all initial points

Table 4.1: behavior of iterating the logistic map for different values of the parameter a

4.1 Introduction

A dynamical system describes how the state of a system evolves with time. Formally

Definition 4.1.1: *A **discrete dynamical system** is a triple $(\mathbb{N}, X, \Phi)$ with a non empty set X (the state space) and an operation $\Phi : \mathbb{N} \times X \to X$, so that $\Phi(0,x) = x$ and $\Phi(n, \Phi(m,x)) = \Phi(n+m,x)$.*

Thus, a discrete dynamical system is the model of a system that evolves over discrete time steps.

The simplest case of a discrete dynamical system is when the state space is 1-dimensional (say $X \subseteq \mathbb{R}$) and the transition function only depends on the previous value, formally $\Phi(1,x) = f(x)$ for some $f : \mathbb{R} \to \mathbb{R}$. In this case it can be written by the recurrence relation $x_{n+1} = f(x_n)$ with initial condition $x_0 \in \mathbb{R}$. The set of the x_n is called **orbit** of the map f.

In this thesis only very simple examples of dynamical systems are used. A broader introduction can for example be found in [KH97].

A chaotic system is a dynamical system that is highly sensitive to initial conditions. As a consequence even relatively small numerical errors made during the computation grow exponentially fast.

A prototypical example of a chaotic system is the logistic map.

Definition 4.1.2: *The **logistic map** is given by the recurrence relation $x_{n+1} = a x_n (1 - x_n)$ with a, $x_0 \in \mathbb{R}$.*

The behavior of the logistic map depends on the parameter a.
Table 4.1 shows the behavior for different values for this parameter.

The interesting case for this thesis is for $a \in (3.544, 4)$, since this is where the iteration of the map leads to chaotic behavior.

The `iRRAM` framework can be used to compute iterations of the logistic map exactly. It is therefore very well suited to investigate this chaotic behavior.

Figure 4.1 shows how sensitive the map is to small errors that occur because of finite precision computations. When using 10 significant digits for the computation, it can be seen that after around 50 iterations, the finite precision version diverges from the `iRRAM` version and after that behaves completely differently.

`iRRAM` can also be used to approximate the complexity in terms of needed computation precision. `iRRAM` automatically increases the internal precision until it suffices to output the result with the demanded number of digits. Thus, the internal precision at the end of the computation gives a measure of this complexity.

As can be seen in Figure 4.2, the needed internal precision grows extremely quickly with the number of iterations. To compute 10 significant digits of the 20000-th iterate, `iRRAM` internally computes already with precision nearly 2^{-50000}, which is of course far from what is possible with any standard floating point data type.

4.1.1 The Shadowing Lemma

The next section deals with the relation between exactly computed and approximately computed orbits and how much numerical orbits have to do with real orbits.

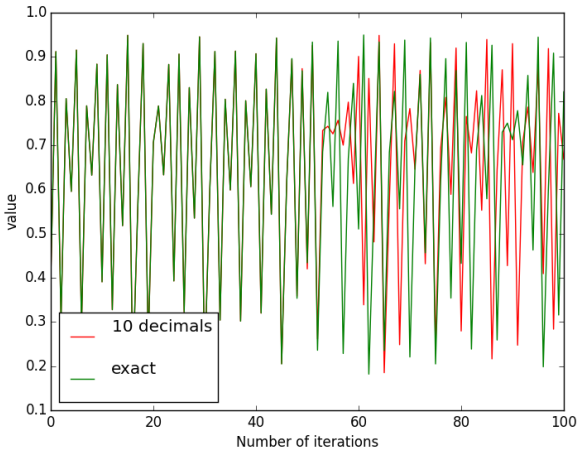


Figure 4.1: 100 Iterations of the logistic map with $a = 3.8$ and $x_0 = 0.4$ computed with 10 decimals precision and exactly.

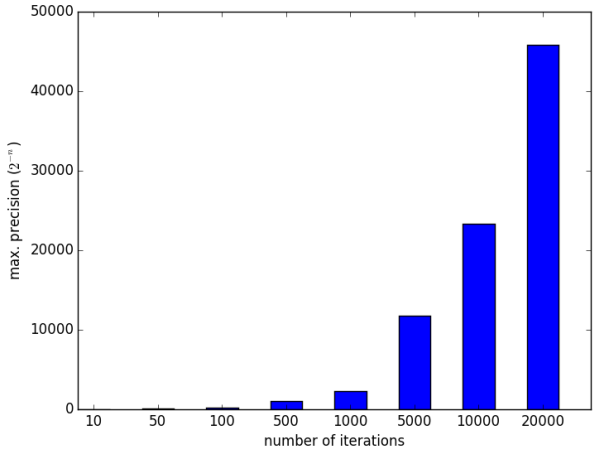


Figure 4.2: Maximal precision `iRRAM` uses to compute iterations of the logistic map and output the points with 10 decimal digits.

Definition 4.1.3: *A sequence* $(x_i)_{i\in\mathbb{N}}$ *is called an* α***-pseudo-orbit*** *for a map* f *if for all* $i \in \mathbb{N}$

$$\|x_{i+1} - f(x_i)\| < \alpha$$

One can think of a pseudo orbit as a numerically computed orbit, where small rounding errors can occur in every evaluation of f.

Definition 4.1.4: *A real orbit* $(y_i)_{i\in\mathbb{N}}$ β***-shadows*** *the pseudo-orbit* $(x_i)_{i\in\mathbb{N}}$ *if for all* $i \in \mathbb{N}$

$$\|x_i - y_i\| < \beta.$$

For systems that have a certain property, namely being uniformly hyperbolic, Anosov and Bowen could show the following result [Ano67] [Bow75] [HP08]:

Theorem 4.1.5 (Shadowing Lemma): Let $f : \mathbb{R} \to \mathbb{R}$ be a uniformly hyperbolic map. Then it holds that for all $\beta > 0$ there exists an $\alpha > 0$ such that for every α-pseudo-orbit $(x_i)_{i\in\mathbb{N}}$ there is a point p_0 so that the real orbit $(y_i)_{i\in\mathbb{N}}$ defined by $y_0 := p_0$ and $y_{i+1} := f(y_i)$ β-shadows $(x_i)_{i\in\mathbb{N}}$.

In other words, in the uniformly hyperbolic case, every numerically computed orbit is close to a true orbit with a slightly changed initial point.

Unfortunately, many maps (including the logistic map) are not uniformly hyperbolic.

For non-hyperbolic f, it can not be expected that there is a real orbit that stays close to the pseudo orbit forever. Instead, there will be an orbit staying close to (x_i) for some time (say up to some point n_0) and then starting to diverge.

The goal of this case study is to investigate such orbits, and in particular to find out how long there is an orbit staying close to the pseudo orbit. The basis for the case study is a paper by Hammel, Yorke and Grebogi from 1987 [HYG87]. The paper deals with the question, how long numerical orbits of the logistic map can be shadowed by true orbits.

The authors show that for $a = 3.8$ and $x_0 = 0.4$, there is a true orbit $(y_i)_{i\in\mathbb{N}}$ of the logistic map, so that $\|x_n - y_n\| < 10^{-7}$ for $n \leq 10^7$.

Their proof method is to compute bounds on the points of the true orbit with a computer by using a form of interval arithmetics. All their computations were done on a Cray X-MP supercomputer.

For the case study, their algorithm was first reimplemented on a modern computer in the `C++` programming language, both using fixed precision floating point numbers and `iRRAM`.

As a second step, `iRRAM` was used to compute the shadowing orbit exactly.

4.1.2 The Algorithm

In their paper Hammel, Yorke and Grebogi give an algorithm to compute a shadowing orbit $(y_i)_{i\in\mathbb{N}}$ of size N for a given pseudo orbit $(x_n)_{n\in\mathbb{N}}$ of the logistic map f.

Instead of computing the shadowing orbit from the first point, the algorithm computes the inverse orbit by starting with the last point and then iteratively computing the predecessor. Since the logistic map is not injective, there is not a unique choice for this predecessor.

For some point $f(x)$ there are normally two possible values for the inverse given by

$$f_{1,2}^{-1}(x) = 0.5 \pm \sqrt{0.25 + \frac{x}{a}}$$

An orbit can be computed by setting $y_N = x_N$ and then iteratively applying one of the inverse maps $y_{n-1} = f^{-1}(y_n)$.

To stay close to the pseudo orbit, in every step the point on the same side of 0.5 as x_n is chosen. In the following, this function will just be denoted by f^{-1} when the choice of index is obvious.

The backward procedure will give an orbit close to the pseudo-orbit since the inverse function on $(0, 1)$ is a contraction, i.e. $\|f^{-1}(x) - f^{-1}(y)\| \leq |x - y|$.

Let δ be the rounding error made when numerically computing $f(x_n)$. Then it holds $\|f^{-1}(x_n)\| = \|f^{-1}(f(x_{n-1}) + \delta)\| \leq \|f(x_n)\| + \varepsilon$ for some small ε. This leads to

$$\|x_{n-1} - y_{n-1}\| = \|x_{n-1} - f^{-1}(y_n)\| \leq \|f^{-1}(x_n) - f^{-1}(y_n)\| + \|f^{-1}(x_n) - x_{n-1}\| \leq \|x_n - y_n\| + \varepsilon$$

Of course, using only floating point arithmetics, the inverse can not be computed exactly and thus the shadowing orbit itself can not be computed.

Instead of computing the orbit, an error bound can be computed with the help of interval arithmetics. That is, a sequence of intervals $(I_n)_{0\leq n\leq N}$ giving upper and lower bounds for y_n is computed, i.e Intervals I_n such that

$$y_n \in I_n \text{ for } 0 \leq n \leq N.$$

The procedure is started with $I_N := [x_n, x_n]$ and then I_{n-1} is selected so that $I_n \subseteq f(I_{n-1})$ holds.
In each step, I_n is chosen as small as possible.

If at some point $x_n < \frac{a}{4}$ the inverse is not defined and the procedure fails.

In the other case, an upper bound on the maximal distance between all points in $I_n := [l_n, r_n]$ and x_n is computed.

That is

$$\delta = \max_{0\leq n\leq N} \min(|x_n - l_i|, |x_n - r_i|)$$

gives a shadowing bound.

4.2 Implementation

Several versions of the above algorithm were implemented.

The first one is a reimplementation of the original interval arithmetic version on a modern computer.

The Cray-MP's floating point data types had different sizes than the standard types in modern programming languages. To get similar results as in the original work, the cray double precision data type therefore had to be simulated using software implementations.

For the C++ implementation, the boost multiprecision library [MK12] was used. The library provides data types that replace the native C++ floating point types, but with a user defined precision.

The cray double data type used for the computations in the paper has machine epsilon $\varepsilon_\mu = 2^{-95}$.

To compute the intervals I_n from I_{n+1}, the inverse of the interval is computed as described above. As long as the condition

$$I_{n+1} \subseteq f(I_n) \tag{4.1}$$

is not fulfilled, I_n is enlarged on both sides by the minimal possible quantity ε_μ.

However, since both the computation of $f(I_n)$ and $f^{-1}(I_n)$ are done using floating point arithmetics, this is not enough to guarantee the condition 4.1.

Further enlarging the interval on both sides by the constant 10^{-25} takes care of this problem.

For the interval arithmetic a simple interval class template `fixed_precision_interval<prec>` was written to represent intervals of the form $[a, b]$ with a and b multiprecision floating point numbers with `prec` bits precision.

All needed operations on intervals were implemented for this class, making the implementation of the algorithm straight forward.

Instead of only implementing the algorithm in the original form, generic types where used such that experiments with different floating point precisions could be done.

The second implementation made is very similar to the first one but instead of using intervals with fixed precision floating point end points, `iRRAM`'s `INTERVAL` type was used.

The data type provides interval arithmetic for intervals with real numbered end points. The above algorithm can therefore be implemented by first computing a pseudo orbit using floating point arithmetics and then computing the inverse of a small interval around the last point exactly.

The last implementation uses `iRRAM` to compute a shadowing orbit exactly.

Again, a pseudo orbit is computed by using floating point arithmetic in the iterations of the logistic map.

Then, starting with the last point of the pseudo orbit the inverses can be computed exactly until the initial point is reached. In the end, the distance of this real orbit and the pseudo orbit can be computed and the maximum of all those distances taken to find the shadowing distance.

One advantage of the last approach is that the `iRRAM` code to compute the exact orbit is extremely simple. Since all operations can be performed exactly, there is no need for intervals or other complicated structures. The data type `REAL` takes care of everything.

Again, the implementation uses generic types, to make it possible to compute the shadowing distance for various different floating point precisions.

4.3 Evaluation

4.3.1 Breakdown points

As mentioned above, the algorithm only works if at each step $x_i \leq \frac{a}{4}$. If the algorithm fails, the first N for that there is some i such that $x_i > a_i$ is called the breakdown point.

The breakdown point depends on the precision that is used to compute the pseudo orbit.

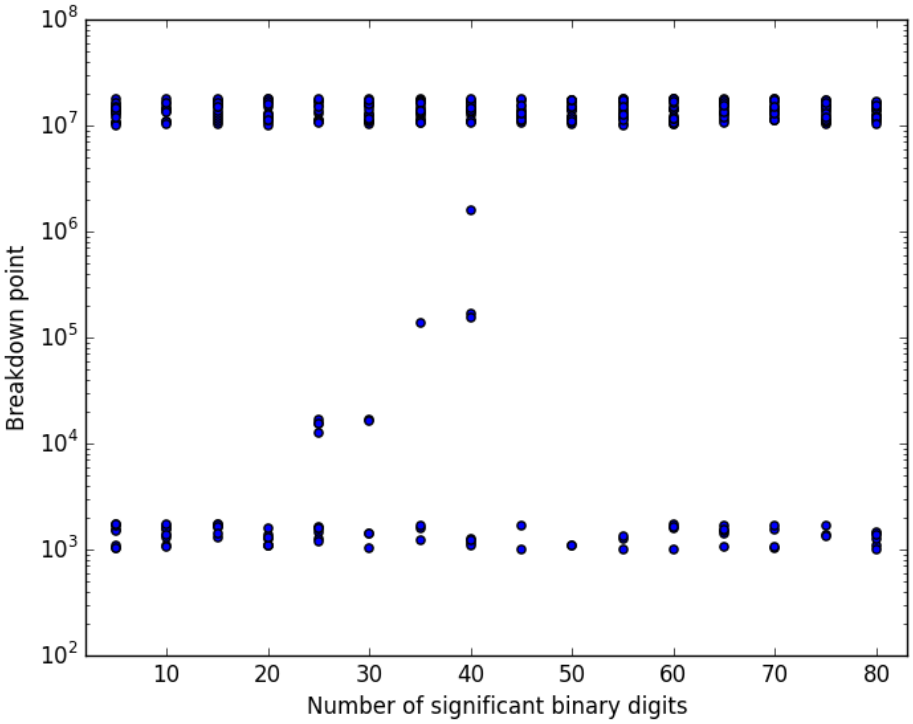


Figure 4.3: Breakdown points depending on N for different parameters a and p_0

Since the computation precision on the Clay X-MP was determined by the available floating point data types, it was not possible to just adjust it.

Instead, the noise was artificially increased by setting $x_{n+1} = ax_n(a - x_n) + \delta_p \cos(\Theta)$ with $\Theta = n(mod 997)$ and then the shadowing distance in relation to δ_p measured.

However, using the boost multiprecision library's `cpp_bin_float` data type, it is easy to change the number of binary digits of the used floating point data type instead of artificially increasing the noise.

In the original work the relation $N \approx \frac{1}{\sqrt{\delta_p}}$ between the noise amplitude and the breakdown points was found.

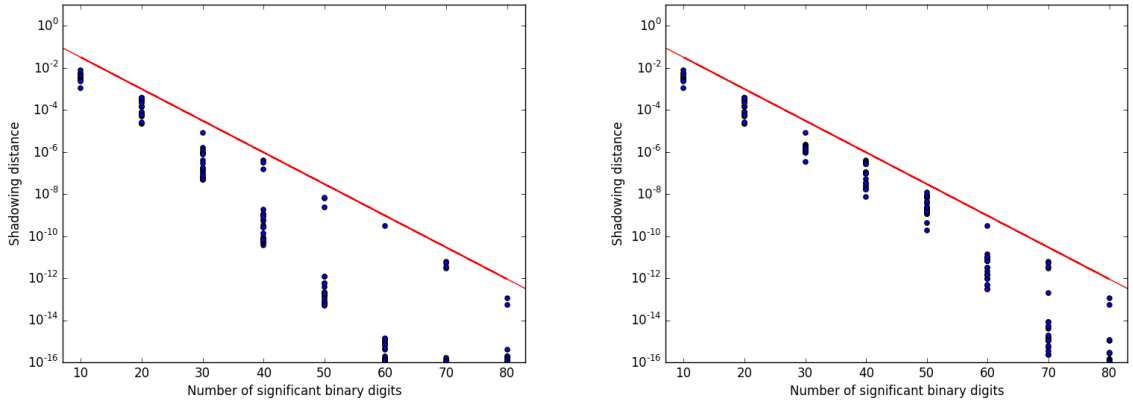
Interestingly no such relation could be found for the modified version that uses `cpp_bin_float` instead of a noise amplitude. As can be seen in 4.3, in this version breakdowns very rarely occur at all during the first 10^7 iterations. There are, however, still some parameters for that a breakdown occurs. In most of this cases it occurs rather quickly, during the first 1000 iterates.

The reason for that is probably that the correct rounding compensates much of the errors, which is not the case when the noise is artificially generated by the noise amplitude.

4.3.2 Shadowing bound and computation precision

The original paper also deals with the question, how close a pseudo-orbit is shadowed depending on the computation precision. The same was also done for the different implementations in this thesis.

Again, instead of artificially generating noise, the precision of the floating point type was changed.



(a) 10^3 iterations

(b) 10^7 iterations

Figure 4.4: Shadowing distance for different parameters a and p_0 of the logistic map. The distance is bound by $\frac{1}{\sqrt{\delta}}$ where $\delta = 2^{-n}$ is the precision.

The shadowing distance was computed for several different parameters a and starting points p_0. The number of iterations N were also varied.

The result can be seen in Figure 4.4.

The tests were made with all versions of the algorithm described above. The results obtained from the different versions were nearly identical, thus only the `iRRAM` version is depicted. Also, the running time of both program versions did not differ much for the performed evaluations.

In the original paper it is conjectured that for $2M$-digit accuracy the orbit stays close to 10^{-M} for around 10^M iterates.

As can be seen in the graph, for up to 10^7 iterates the conjecture holds.

5 Case Study: A Datatype for Analytic Functions

The following case study describes an implementation for a data-type for analytic functions in `iRRAM`. The implementation and theoretical analysis is joint work with Akitoshi Kawamura and Florian Steinberg [KST15].

The theoretical foundation for the case study can be found in [KMRZ12].

5.1 Introduction

In Chapter 2.2 it was shown many important operators on real functions are known to be computationally hard.

On the other hand, for many of those functions, fast numerical algorithms working with floating point arithmetic exist and are often very precise and reliable.

Thus, there is a kind of mismatch between the worst case results in Computable Analysis and the practice of Numerical Analysis.

The following questions arise

1. What kind of functions are important and naturally arise in numerical analysis?
2. What kind of implicit assumptions are made about the functions and in what way do they allow faster computations?

Consequently, it might be useful to consider subsets of real functions and investigate which of the operations become feasible.

A subset of real functions that has been thoroughly studied in computable analysis are the analytic functions. It could be shown that many operators that are hard in the general case become polynomial time computable when restricting to analytic functions.

Note, however, that being analytic is a quite strong restriction on a function and that many problems in numerical analysis deal with non-analytic functions.

An analytic function is locally defined by its Taylor series, that is

Definition 5.1.1: *For $z \in \mathbb{C}$ and $r \in \mathbb{R}$, $r > 0$, let $B(z,r) := \{x \in \mathbb{C} \mid |x - z| < r\}$.*
*A function $f : D \to \mathbb{C}$ with $D \subseteq \mathbb{C}$ is called **analytic**, if for every $x_0 \in D$ the function given by the Taylor-series around x_0 converges to $f(x)$ for every x in a neighborhood of x_0.*

That is, for every $x_0 \in D$ there is an $\varepsilon > 0$, there is a series

$$T(x) := \sum_{n=0}^{\infty} a_n (x - x_0)^n$$

such that $T(x) \to f(x)$ for all $x \in B(x_0, \varepsilon)$.

The above definition can also be generalized to higher dimension.

If f is analytic, all its derivatives $f^{(n)}(x)$ exist and it holds

$$a_n = \frac{f^{(n)}(x_0)}{n!}.$$

The set of functions analytic on D will be denoted by $C^\infty(D)$.
For the sake of simplicity, it will w.l.o.g. be assumed that $0 \in D$ and $x_0 = 0$ if not stated otherwise.

Below are some properties of analytic functions.
More detailed information on analytic function can for example be found in [KP02].

1. For $f, g \in C^\infty(D)$ the sum $f + g$ and the product $f\,g$ are analytic functions.
2. If f, g are analytic and $im(f) \subseteq dom(g)$ then their composition $f \circ g$ is analytic.

3. The roots of an analytic function that is not identical to 0 are isolated points.
4. The derivative and anti-derivative of an analytic function are analytic.
5. If the domain is connected, an analytic function is uniquely defined by a single Taylor-series around one point in its domain.

The next theorem shows that the property of being analytic is very useful when it comes to computability and complexity considerations.

Theorem 5.1.2 (Pour-El, Richards, Ko, Friedman [PER89]): Let $f \in C^\infty(B(x_0, \varepsilon))$ and $(a_n)_{n\in\mathbb{N}}$ the Taylor series valid on $B(x_0, \varepsilon)$. Then the following are equivalent

1. f is computable
2. The series $(a_n)_{n\in\mathbb{N}}$ is computable.

Theorem 5.1.3 (Müller [Mül87]): The equivalence in Theorem 5.1.2 also holds when the word computable is replaced by polynomial time computable.

Many operations on functions, like addition, multiplication, differentiation or anti-differentiation, can be reduced to simple manipulations on the Taylor-series for analytic function.

Thus, a direct consequence of Theorem 5.1.3 is the following

Corollary 5.1.4: If $f \in C^\omega(D)$ is polynomial time computable the following functions are also polynomial time computable:

1. The functions obtained by the sum, product or difference of the two functions
2. $i : D \to \mathbb{R}$, $i(x) = \int_0^x f(t)dt$
3. $d : D \to \mathbb{R}$, $d(x) = f'(x)$

Thus, restriction to analytic functions reduces the computational complexity of many important operations on real functions.

However, those theorems are non-uniform in the following sense:

The theorems only say that f being polynomial time computable implies the existence of a polynomial time computable sequence and the existence of such a sequence implies that there exists some algorithm computing f in polynomial time. In no way, however, do they say, how to compute the sequence from a given representation of the function and vice versa.

In fact the following shows that this is not a problem of those theorems, but it's inherently impossible to do so.

Theorem 5.1.5 (Müller [Mül87]): Let f given as in Definition 2.1.7 then the operator $f \to (a_n)_{n\in\mathbb{N}}$, computing the Taylor series around 0 (or any other point) is not computable.

Theorem 5.1.6 (Müller [Mül87]): Let $(a_n)_{n\in\mathbb{N}}$ be the series expansion around 0 for some $f \in C^\omega(D)$.

The evaluation operator $((a_n)_{n\in\mathbb{N}}, x) \to f(x)$ that, given a series and a point, computes the value of the corresponding function at that point, is not computable.

The reason for the non-uniformity is that any algorithm can only read a finite number of coefficients from the series. But from the power series alone, it is impossible to know how many coefficients are needed to make a good enough approximation. This information can not be (computably) extracted from the Taylor-series, thus the algorithms have to be given some additional information.

The goal of this case study was to write a general data type for analytic functions. Due to the above, any such data type will need more information than only the series of Taylor coefficients.

The next section will deal with the question, what additional information is needed to get uniform versions of the above theorems.

5.2 Representation of Analytic Functions

As seen in the previous section, the information given by the series expansion is not enough to make reasonable computations with analytic functions. However, by enriching the information by some finite discrete parameters, the translation between Taylor-series and function representation can be made uniform and even (parameterized) polynomial time computable.

Two possible representations for analytic functions on the closed unit disc are as follows

Definition 5.2.1: *A **series-name** ρ_s of $f \in C^\omega(\overline{B_1(0)})$ is a triple $((a_n)_{n\in\mathbb{N}}, k, A)$ where*

1. *$(a_n)_{n\in\mathbb{N}}$ is the series expansion of f around 0*
2. *$\sqrt[k]{2} \leq r$*
3. *$|a_j| r^j \leq A$ for all $j \in \mathbb{N}$*

and $r = (\limsup |a_j|^{\frac{1}{j}})^{-1}$ denotes the radius of convergence of the series.

The two additional parameters are useful, because they can be used to make a tail estimate

$$\left| \sum_{n\geq N} a_n x^n \right| \leq A \frac{(|z|/r)^N}{1-|z|/r} \tag{5.1}$$

A name as in Definition 5.2.1 can be found by choosing any appropriate k (note that the radius of convergence is always bigger than 1) and choosing A as an upper bound of f extended to $\overline{B_{\frac{k}{\sqrt{2}}}(0)}$.

Definition 5.2.2: *A **function-name** ρ_f of $f \in C^\omega(\overline{B_1(0)})$ is a triple (f, l, B) such that B is an upper bound of f on $\overline{B_{\frac{l}{\sqrt{2}}}(0)}$.*

Theorem 5.2.3: [KMRZ12] The mapping between ρ_s-name and ρ_f-names is computable in time polynomial in $n + k + \log(A)$ and the inverse mapping is computable in time polynomial in $n + l + \log(B)$

Note that the above translation becomes fully polynomial time when in the representations k resp. l are required to be encoded in unary, while A resp. B are given in binary. Thus, from now on the term polynomial time computable will be used when referring to running time bounds as in Theorem 5.2.3.

A full proof of Theorem 5.2.3 can be found in [KMRZ12].

Some details that are important for the implementation are given below as separate theorems.

Theorem 5.2.4: A series name can be used to evaluate the corresponding function in polynomial time.

Proof:

According to Equation 5.1, if the first $N := nk + \log A$ terms are summed up then for the error

$$\left| \sum_{j>N} a_j r^j \right| \leq 2^{-n}$$

holds. □

Theorem 5.2.5: Given a function name it is possible to compute any Taylor coefficient a_k around 0 of the function in polynomial time.

Proof:

To compute a series name from a function name, one has to compute the coefficients of the series expansion from a function name, i.e. from the function and the parameters l and B.

In [Mül93] Müller describes an algorithm for this task.

Let $\mathcal{M}(n)$ denote the time needed to multiply two n-bit numbers (e.g. $\mathcal{M}(n) = O(n \log n \log\log n)$ when using the Schoenhage-Strassen Algorithm [SS71]).

To approximate the coefficient a_k with precision at least 2^{-n} f is approximated by the Lagrangian interpolation polynomial

$$P_m(x) := \sum_{i=0}^{2m} f(x_i) \cdot L_{m,i}(x) \tag{5.2}$$

where

$$\begin{aligned} x_i &= (i-m) \cdot h, h \in \mathbb{R}, h > 0 \\ L_{m,i}(x) &= \prod_{i \neq j} \frac{x - x_j}{x_i - x_j} \end{aligned}$$

Differentiating Equation 5.2 k times yields

$$P_m^k(0) := \sum_{i=0}^{2m} f(x_i) \cdot L_{m,i}^{(k)}(0) \tag{5.3}$$

Let $\sigma = \lceil \log_2 B \rceil + 1$

It can then be shown that if f is evaluated on $2k+1$ points from the real interval $\{x \in \mathscr{R} \,||x| \leq \frac{1}{2}\}$ with precision $2n + 15k + \sigma + 6$ then a_k can be approximated by the above procedure with error less than 2^{-n} in $O((k+1) \cdot \mathscr{M}(n+k+\sigma) + k^2 \mathscr{M}(k \log k)$

The coefficients $L_{m,i}^{(k)}(0)$ in Equation 5.3 are rational numbers and can be computed by a recursion formula that is also given in [Mül93]. □

Theorem 5.2.6: The following is polynomial time computable when given ρ_s or ρ_f names for the input functions.

1. Evaluation $(f, z) \to f(z)$
2. Addition $(f_1, f_2) \to f_1 + f_2$
3. Multiplication $(f_1, f_2) \to f_1 \cdot f_2$
4. d-fold Differentiation $(f, d) \to f^{(d)}$ where d is given as unary
5. d-fold Anti-differentiation $(f, d) \to \int \ldots \int f$ where d is given in unary

Proof (Sketch):

Evaluation follows from 5.2.3.

The manipulations that have to be done on the Taylor series are straight forward, e.g. addition can be done by just adding the coefficients of the two series.

It remains to show how to compute the new parameters A' and k' from the parameters A_1, A_2, k_1, k_2. The results will just be listed here without proof.

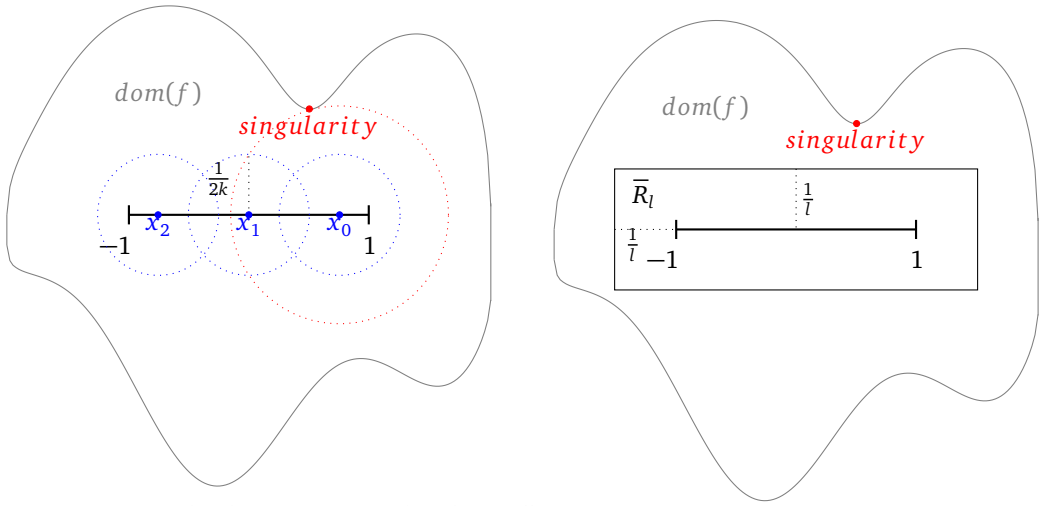
1. Addition: $A' = A_1 + A_2$ and $k' = max(k_1, k_2)$.
2. Multiplication: $k' = 2\max(k_1, k_2)$ and A' is some integer such that $A' \geq A_1 \cdot A_2 \cdot (1 + \frac{k'}{e \ln 2})$.
3. Differentiation: $k' = 2k$ and A' is some integer such that $A' \geq \frac{A}{r} \cdot (1 + \frac{2k}{e \ln 2})$. Note, that the parameters do not depend on d.
4. Anti-Differentiation: $k' = k$ and A' an integer such that $A' \geq A \cdot r^d$.

□

The above can be extended to a data type for analytic functions on arbitrary ball-shaped domains.

But of course, most domains are not ball-shaped and one would also like to represent such domains.

One possible way to represent functions on such a domain, is to cover the domain with balls and use the representation from Definition 5.2.1 on each ball. As an example, consider functions analytic on the real line $[-1, 1]$ (this implies being analytic on a rectangle around the line $[-1, 1]$ in the complex plane).



(a) For a series name the domain is covered with equally sized balls. For each ball, a Taylor series around the center of the ball is given and the properties of Definition 5.2.1 hold.

(b) A function name encodes a closed rectangle around $[-1,1]$ where the function is analytic and an upper bound for the function on that rectangle.

Figure 5.1: Representations for functions analytic on the real line $[-1,1]$

Definition 5.2.7: *Let $f \in C^\omega([-1,1])$. A **series-name** for f is a 5-tuple $(M,(x_m),(a_{n,j}),k,A)$ where $M \in \mathbb{N}$ $1 \le j \le M$, $n \in \mathbb{N}$, $x_m \in [-1,1]$ and it holds*

1. $[-1,1] \subseteq \bigcup_{m=1}^{M} [x_m - \frac{1}{4k}, x_m + \frac{1}{4k}]$
2. $(a_{n,i})_{n\in\mathbb{N}}$ *is the series expansion of f around x_i*
3. $|a_{n,i}| \le Ak^n$ *for all $n \in N$, $1 \le i \le M$*

To simplify the definition the parameters k and A were chosen global, i.e. they hold for each of the series. That means, the domain is covered by equally sized balls. Figure 5.1a shows a series name with three series.

The definition can be further simplified by requiring the series centers to be equidistantly distributed on $[-1,1]$ and thus making it unnecessary to store them in the representation.

A function name as in Definition 5.2.2 can also be defined

Definition 5.2.8: *Let $R_l := [-1-\frac{1}{l}, 1+\frac{1}{l}] \times [-\frac{1}{l}, \frac{1}{l}]$ the closed rectangle with distance $\frac{1}{l}$ around $[-1,1]$.*
*A **function-name** for a function $f \in C^\omega([-1,1])$ is a 3-tuple $(f|_{[-1,1]}, B, l)$ with $B, l \in \mathbb{N}$ such that*

1. $f \in C^\omega(R_l)$
2. *B is an upper bound for f on R_l*

where l is coded in unary and B in binary.

See Figure 5.1b for a picture describing the function name.

Again it can be shown that the two representations are polynomial time equivalent, and that the equivalent to Theorem 5.2.6 holds.

5.3 Analytic Continuation

One problem with the representation in Definition 5.2.7 is that when thinking of this representation as an interface for a data type, it is very cumbersome for the user to provide all the information needed. He would need to give an algorithm for as many Taylor series as needed to cover the interval.

As said before, there is no real gain of information by providing all those series. The analytic function is uniquely defined by the Taylor series around a single point.

A simplified representation can therefore be as follows

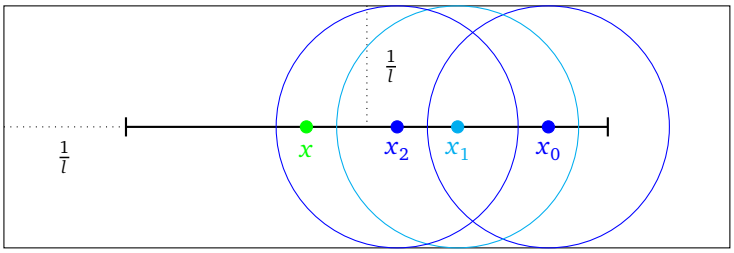


Figure 5.2: Evaluation by Analytic Continuation: to evaluate the function at point x given the series around x_0, first the series around point x_1 is computed. This series is used to compute one more series around x_2 which then finally can be evaluated at x.

Definition 5.3.1: *A **single series name** for $f \in C^\omega([-1,1])$ is a quadruple $(x_0,(a_n)_{n\in\mathbb{N}},k,A)$ such that*

1. *$(a_n)_{n\in\mathbb{N}}$ is the series expansion of x around x_0*
2. *A and k are such that $|a_m| \leq Ak^m$*
3. *The parameters A and k are valid on the entire rectangle R_l*

A series name can be obtained from a single series name by computing the series expansion on another point on the domain and then iterating this process until the series cover $[-1,1]$.

Since the parameters A and k are such that they hold on the entire rectangle, the new series will be valid on a ball with the same radius as the original series, effectively extending the domain. This process is called **analytic continuation**.

The series can be computed either by applying the algorithm from the proof of Theorem 5.2.3 or by directly computing the derivatives at an other point using Theorem 5.2.6.

Note, however, that iterating this process will break the polynomial time computability, as will be shown in the next section.

5.4 Runtime Analysis

The previous section gave an overview of the representations and how they can be used to yield efficient (in the sense of polynomial time computability) operations on analytic functions. Since for practical applications polynomial time computability is too coarse a measure for efficiency, in this section a more refined analysis on the algorithms is done.

The goal is to establish time bounds in terms of the O-notation.

Most of this has already been done in [KST15], thus proofs are omitted or only sketched.

Let $T(a_m,n)$ denote the running time needed to compute the coefficient a_m from the Taylor series with an error less than 2^{-n}. The running time of all operations on the Taylor series will depend on this factor.

The following theorems have been shown in [KST15]

Theorem 5.4.1: Given a series name of $f \in C^\omega(D)$. To evaluate f with error less than 2^{-n}

$$N = k(n + \lceil \log_2(\log_2(e^2)kA \rceil)$$

coefficients of the Taylor series are sufficient.
This yields evaluation computable in time

$$O(N\mathcal{M}(N) + N \cdot T(a_N, n + \log_2(N) + 1))$$

Theorem 5.4.2: The m-th coefficient of the d-times differentiated series is given by $a_m^{(d)} = a_{m+d}\prod_{i=1}^{d}(i+m)$.

Computing the m-th coefficient of the series for the d-fold derivative can thus be done

$$O(T(a_{m+d}, n + d\log_2(d+m))) + d\mathcal{M}(d\log_2(d+m)+n))$$

Theorem 5.4.3: Given a series name of $f \in C^{\omega}(D)$, the number of coefficients of the series of the d-fold derivative computed from f as in Theorem 5.2.6 needed to evaluate this derivative, is given by

$$N_d = k\left(n + \lceil \log_2(\log_2(e^4)kA\rceil + \left\lceil d\left(\log_2(d) + \log_2(\log_2(e)k+1) - \frac{1}{k}\right)\right\rceil\right).$$

Thus, evaluating the d-fold derivative of f is possible in time bound by

$$O(N_d \mathscr{M}(N_d) + N_d \cdot T(a_{N_d+d}, n + d\log_2(d+N_d))) + d\mathscr{M}(d\log_2(d+N_d)+n)$$

For computing the m-th coefficient of a Taylor series around some point the m-th derivative has to be divided by $m!$. This division reduces the needed precision leading to

$$N_{coeff}(m) = 2k\left(n + \lceil \log_2(\log_2(e^4)kA) + m\log_2(\log_2(e)k+1)\rceil\right)$$

as the number of coefficients that are needed to compute the m-th coefficients of the Taylor series around another point.

For stepwise analytic continuation it is necessary to iterate the process of differentiating. For example, using the single series name when one wants to evaluate the function at a point x that is not inside the ball of the given series, analytic continuation has to be applied until a series containing x is reached.

Assume for a computation the coefficients up to N of the last series is needed. The previous results lead to the recurrence relation relation

$$\begin{aligned} N^{(0)} &= N \\ N^{(l+1)} &= 2k\left(n + \lceil \log_2(\log_2(e^4)kA) + N^{(l)}\log_2(\log_2(e)k+1)\rceil\right) \end{aligned}$$

further leading to

Theorem 5.4.4: Applying analytic continuation l times, to compute N coefficients of the last series

$$N^{(l)} = O((2k\log_2(\log_2(e)k+1))^l N) \tag{5.4}$$

coefficients of the original series are needed.

The above leads to a running bound for evaluating using l-times iterated analytic continuation.

Theorem 5.4.5: The running time of evaluating the l times iterated series is bounded by

$$O(l(N^{(l)})^2 \mathscr{M}(N^{(l)}) + N^{(l)} T(a_{N(l)}, lN^{(l)}\log_2(N^{(l)}))) \tag{5.5}$$

Due to the number of parameters the running time depends on, the above bounds are quite complicated.

To get get simple bounds the following assumptions were made

1. A and k are usually very small compared to the other parameters, thus they are neglected except for the case when k is in the base of an exponentiation.
2. The time $T(a_m, n)$ to read a coefficient of the series is neglected. Although this time has a huge impact on the overall running time, for reasonably fast sequences it will be dominated by the other parameters.

This leads to the following simplified bounds

1. Evaluation of a series-name is possible in time $O(n\mathscr{M}(n))$.
2. Evaluation of the d-fold (anti-)derivative is possible in time $O((n+d)\mathscr{M}(n+d))$
3. Evaluation of the l times iterated analytic continuation is possible in time $O((2k\log k)^{2l} n^2 \mathscr{M}((2k\log k)^l n)$
4. The number of coefficients that have to be read from the original series to evaluate a series gained by l-times iterated analytic continuation is bounded by $O((2\log_2(\log_2(e)+1))^l) = O(2.56^l)$

5.5 Implementation

iRRAM was used as a framework to implement the ideas in the previous section. The goal was to add user friendly classes for analytic functions to iRRAM. In particular two classes were created for this purpose.

The class BA_ANA is a basic class for analytic functions on a closed disc with rational radius around 0, inspired by Definition 5.2.1.

The class ANA_RECT is a class for functions analytic on an arbitrary real interval $[a, b]$, inspired by Definitions 5.2.1 and 5.2.7.

Both classes provide a common set of operators and methods, in particular

1. It is possible to add, multiply and subtract two objects using the overloaded operators $+$, $*$, $-$.
2. It is possible to evaluate the function at $x \in D$ using the overloaded operator ()
3. There is a function `differentiate (unsigned int n)` to get a new object representing the d-th derivative of the analytic function.
4. There is a function `integrate (unsigned int n)` to get a new object representing the d-th anti-derivative of the analytic function.
5. It is possible to get the coefficients of the underlying Taylor series.

Since a power series is an infinite object, some thought has to be put into how to store it on a computer.

Some reasonable approaches are for example

1. A finite implementation. At every point during the execution of the program, a finite initial segment of the power series is precomputed and stored in memory. The length of the initial segment depends on the required precision and can change during the run of the program (e.g. with reiterations in iRRAM). All algorithms performing operations on analytic functions will be given only polynomials, and thus only operate on them.
2. A functional implementation. The coefficients are not precomputed and saved. All algorithms are given the power series as a function from an integer n to the coefficient a_n (or a pointer to such a function). If an algorithm needs to read a coefficient it calls the function.

The second method is very similar to the DAG approach, that is when performing operations that generate new power series pointers to all functions used have to be saved. It can therefore suffer from the same memory issues as the DAG approach. Despite that fact and even though the first method is arguably more in the spirit of the iRRAM framework, a decision was made to follow the second approach.

There are several reasons for that. Firstly, the number of coefficients needed from the power series depends heavily on the point where the analytic function is evaluated. If the length of the used initial segment grows with the internal precision of iRRAM, worst case assumptions have to be made on the needed number of coefficients. This would in most cases be a huge overestimation and therefore lead to longer computation times.

Secondly, the memory issue does not play such a big role in this case. Usually, operations like differentiation or analytic continuation that increase the size of the DAG are not applied hundred thousands of times, but orders of magnitude less. Thus, the DAGs stay rather small and easily fit into the main memory of a modern computer.

The pure functional approach, however, has the disadvantage that a coefficient has to be computed every time it is needed and thus the same work might have to be done many times. Depending on the algorithm, this can lead to very poor performance.

To prevent this, a form of caching was implemented. The coefficient function is only called the first time when a coefficient needs to be computed and then saved to the cache. The next time the coefficient is accessed it is taken from the cache instead of calling the function again. Thus the coefficient is only recomputed when the precision is not sufficient and iRRAM reiterates the computation.

5.6 Class Overview

In addition to the two classes for analytic functions, several helper classes were implemented. The design of the implementation is very close to the interface given by the theoretical framework, i.e. there are classes for different mathematical objects that work on representation classes for that object.

All classes were implemented with generic types using Templates whenever it made sense.

The following classes exist

5.6.1 POLY

POLY<coeff_type> is a class template for polynomials of a generic coefficient type. The class for the coefficient type should have implementations of the * and + operators and it should be possible to cast 0 to <coeff_type>.

The following has been implemented for POLY

1. Constructor from a vector<coeff_type> and default constructor (constant zero polynomial).
2. Copy constructor POLY(const POLY<coeff_type>& P) and Copy assignment constructor POLY& operator = (const POLY<coeff_type>& P).
3. Methods get_degree() and get_coeff(const unsigned int n) returning the degree and a specific coefficient.
4. Addition, Subtraction, Multiplication and Scalar Multiplication via the overloaded operators +, -, *, +=, *=.
5. Evaluation via the overloaded operator ().
6. symbolic differentiation and anti differentiation via the functions void diff(unsigned int) and void integrate(unsigned int n).
7. A function void rwrite(const POLY<coeff_type>& P, int precision) that outputs the polynomial in the form $a_n * X^n + \ldots a_n * X + a_0$ where the real numbers $a_0, \ldots, a_n$ are written with the number of decimals given by the parameter precision.

All methods were implemented in the most straight forward way.

5.6.2 FUNC

FUNC<RESULT(PARAM)> is a class template for functions with input of type PARAM and return type RESULT. It can be seen as a thin wrapper around std::function.

The idea behind FUNC is to have a class for functions in the mathematical sense, thus the classes for the template parameters should have the arithmetical operators implemented.

The following methods exists

1. Constructors from an std::function or a pointer an std::function.
2. Copy constructor FUNC(const FUNC<RESULT(PARAM)>&) and copy assignment constructor FUNC& operator = (const FUNC<RESULT(PARAM)>&)
3. Addition, Subtraction, Multiplication and Scalar Multiplication via the overloaded operators +, -, *, +=, *=
4. Evaluation with the operator ()
5. Function Composition with the operator ()

Methods are performed on a representation class called rep_rho_D.

Straight forward implementations of all operations are possible using C++11's lambda functions, e.g. addition is implemented the following way

```
alg_func_ptr<PARAM, RESULT> sum(new const alg_func<PARAM, RESULT>(
        [alg1, alg2](const PARAM& x) -> RESULT
                {return (*alg1)(x) + (*alg2)(x);}
));
```

5.6.3 POWERSERIES

POWERSERIES<coeff_type> is a class template for power series of a generic coefficient type, i.e. a (symbolic) series of the form $\sum_{i=0}^{\infty} a_n \cdot x^n$.

The same restrictions on the coefficient type as for POLY hold.

The series itself is represented as a FUNC<coeff_type(unsigned int)>, i.e. a function from the integers to coeff_type.

Note, that hereby the maximal number of coefficients is limited by the size of unsigned int, but the limit is large enough to have no practical relevance.

POWERSERIES has the following methods

1. Constructors from a shared_ptr to a FUNC<coeff_type(unsigned int)> and from a single object of coeff_type (giving a sequence with constant coefficient sequence).
2. Copy constructor POWERSERIES(const POWERSERIES<coeff_type>& P) and copy assignment constructor POWERSERIES& operator = (const POWERSERIES<coeff_type>& P).
3. A method coeff_type get_coeff(const unsigned int& n) returning the n-th coefficient
4. A method POLY<coeff_type> cut_of_at(const unsigned int& n) that returns the polynomial given by $\sum_{i=0}^{n} a_i x^i$.
5. Addition, Subtraction, Multiplication and Scalar Multiplication via the overloaded operators +, -, *, +=, *=
6. symbolic differentiation void differentiate(int n) [where did anti differentiation go?]
7. Composition of POWERSERIES using the operator ()

The methods are mainly implemented as realizer functions on the corresponding representation class rep_rho_dy_omega.

Again, the implementation is mostly straight forward transformations on the power series.

Since the objects that are manipulated are functions C++11's functional features come in handy and are used heavily.

This class also implements a form of caching for the power series coefficients.

The rep_rho_dy_omega class contains a vector<coeff_type> and whenever a coefficient is read, all coefficients up to this one are saved in the vector and retrieved from there when read the next time.

5.6.4 BA_ANA

BA_ANA<ARG> is a class template for functions analytic on a closed disc with some rational radius r around 0.

In the current form, for the template parameter ARG only REAL or COMPLEX makes sense.

Apart from the methods at the beginning of this section, the following methods exists

1. A constructor from a POWERSERIES, two INTEGERs k and A and a RATIONAL r. To work correctly r has to be strictly smaller than the radius of convergence R and it has to hold $\sqrt[k]{2r} < R$ and $|a_n| \leq \frac{A}{r^n \cdot 2^{n/k}}$.
2. A constructor from a FUNC, two INTEGERs k and A and a RATIONAL r.
3. A method POWERSERIES<ARG> series_around(const ARG&) that computes the series expansion around an arbitrary point in the domain.
4. Composition using the () operator.

BA_ANA uses two representation classes, rep_pi, the representation by a power series, and rep_rho_D_refined, the representation as a function. All operations except evaluation use the power series representation. If the constructor from a function object is called, the power series will also be computed, thus the rep_pi-name is always available.

Evaluation can also be realized from the rep_pi name, but it will usually be faster to use the function if it is available.

The methods are implemented as described in Theorems 5.2.6 and 5.2.3.

For evaluation it is very useful to have access to iRRAM's internal representation of the error of a REAL.
The value $f(x)$ is obtained summing up a finite initial segment of the Taylor series $\sum a_i x^n$.

The error consists of two parts, the error that is made because of the truncation (see 5.1) and the error that is made because of the coefficients and x only being available as finite approximations.

While summing up the terms in the sum, the second type of error might become bigger than the first type. In that case, it is better to stop the summation and return a REAL with a too big error, to make iRRAM start a new iteration with more precise approximations to the coefficients and x.

5.6.5 ANA_RECT

The class ANA_RECT represents complex functions that are analytic on an interval $[a, b] \subseteq \mathbb{R}$. The internal representation is very similar to the one in Definition 5.2.7. That is, a finite sequence of (overlapping) equidistant Taylor sequences covering the interval and parameters A and k that are valid for all the series.

Apart from the methods common with BA_ANA the following exist

1. A constructor from (a pointer to) a POWERSERIES, the two integers k and A and two REALs left and right that give the end points of the line where the ANA_RECT object will be defined.
2. A constructor from a function pointer, two INTEGERs l and B and left and right as above.
3. A method get_series_number(int n) that returns a pointer to the POWERSERIES for the n-th series.
4. The method cut_of_at(int n, int m) returns a POLY with the first m coefficients of series n.

The constructor only takes a single POWERSERIES. All other power series needed to cover the interval are computed from the provided one using analytic continuation.

ANA_RECT uses two representation, a function representation rep_fun and a series representation rep_ana_rect that can be transformed into each other.

The series representation rep_ana_rect contains a vector with the power series. When evaluating the rep_ana_rect-name at a point $x \in \mathbb{C}$, a Taylor series containing x is chosen and then a BA_ANA object of this series created and evaluated at the (appropriately shifted) point x.

For other operations, the according transformation of the first series is computed and then the other series are computed from that.

5.7 Usage

The following C++ program gives an example how the BA_ANA class can be used. ANA_RECT can be used similarly.

```
1  #include "iRRAM.h"
2  #include "Functions/Analytic/BA_ANA.h"
3  using namespace iRRAM;
4
5  REAL factorial(const int n){
6          if (n==0)
7                  return REAL(1);
8          return factorial(n-1)*REAL(n);
9  }
10
11 REAL sinseries(const int n){
12         if (n%2 == 0) return 0;
13         if (n % 4 == 3) return -1/factorial(n);
14         return 1/factorial(n);
15 }
16
17 void compute() {
18         POWERSERIES<REAL> ps_sin(sinseries);
19         BA_ANA<REAL> sinus(ps_sin, 2,2);
20         sinus.differentiate(2);
21         rwrite(sinus(0.2), 20);
22         iRRAM::cout << endl;
23 }
```

The snippet defines a BA_ANA object for the sine function.

Lines 5 to 15 define a function that returns with input an integer n the coefficient a_n of the sine series.

Instead of defining a function it is also possible to use C++11's lambda feature to define a function object.

The main part of the program happens in `iRRAM`'s compute function.

Line 18 defines a `POWERSERIES` object from the series function.

In Line 19 the `BA_ANA` object is defined from the power series and the parameters $A = 2$ and $k = 2$.

This object is then differentiated two times (line 20) and then evaluated at 0.2 (line 21).

The `rwrite` function is used to output the resulting real number with 20 valid decimal digits.

Note that there are much faster ways to implement the power series for the sine function, but for the example the most straight forward method was used.

5.8 Evaluation

Due to the chosen functional approach, most operations like differentiation do by themselves not perform any time critical operations but instead only define a new function to compute the coefficient series of the result.

As long as no coefficient is accessed, this function is not called and thus no operations performed. Thus, the only time critical operations are evaluation and printing coefficients.

Consequently the way to measure the complexity of the other operations, is evaluating the resulting function at some point.

For example for measuring the complexity of differentiation, one can first differentiate the function and then measure the running time of evaluating the differentiated function.

Running time measurements were performed for several different test functions. In particular, a number of tests were performed on the functions $x \to sin(x)$, $x \to \frac{1}{1+x^2}$, $x \to ln(2+x)$ and $x \to \sqrt{2+x}$.

The additional parameters A and k for those functions were chosen as small as possible and are between 1 and 2 for all functions.

As the running times are very similar for all the functions, only a few examples are included here.

To better understand the generated graphs, it is important to note that due to the iterative concept of the `iRRAM`, the running time is not continuous in the desired output precision, but grows stepwise with the iteration of the `iRRAM`.

All the evaluations were performed using a Mac-Book Pro with a 2 GHZ Intel Core i7 processor and 4GB of RAM. The code was compiled using gcc version 4.9.1 and `iRRAM` version 2013 01.

To get a reliable estimate for the running time, all evaluation scripts were run twice and the average of the two runs taken.

Below the results for both the evaluations performed on the both the `BA_ANA` and the `ANA_RECT` data-type are shown.

5.8.1 BA_ANA

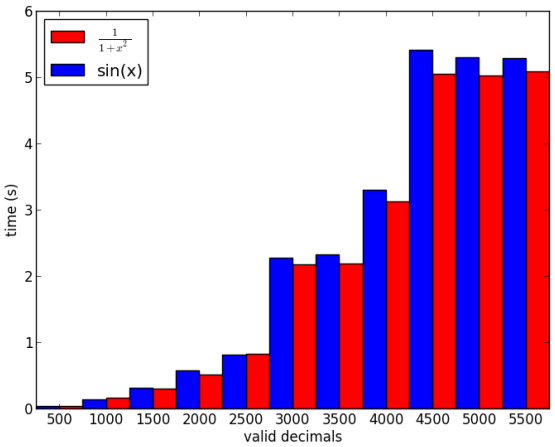


Figure 5.3: running time evaluating BA_ANA at fixed point $x = 0.8$ depending on the desired number of valid decimals

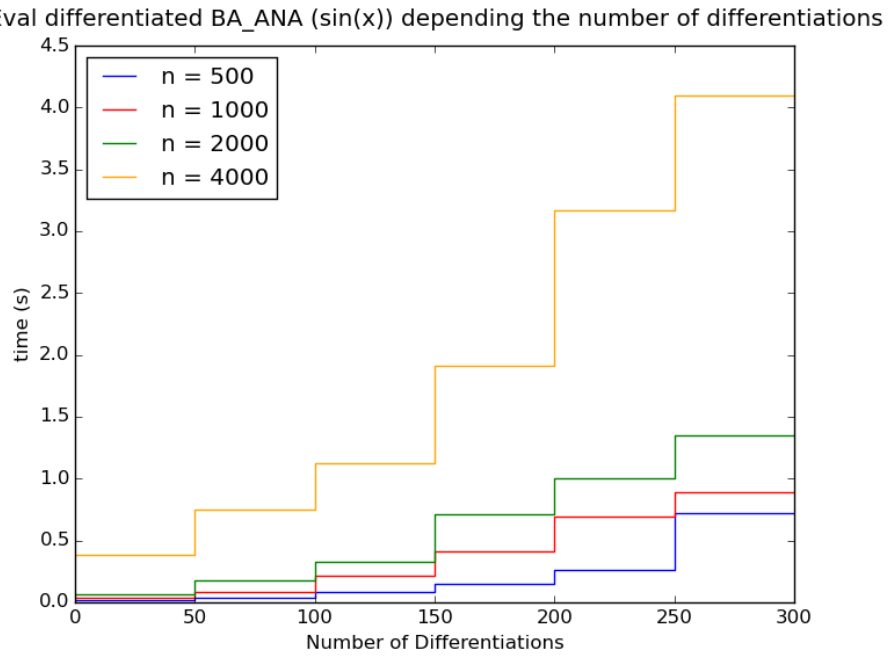


Figure 5.4: running time evaluating differentiated BA_ANA with series for $\sin(x)$ at fixed point $x = 0.8$ with different number of desired valid decimals depending on the number of differentiation

All operations on BA_ANA are expected to run in polynomial time. Thus, the algorithms are expected to run reasonably fast.

This could be verified in the performed tests. Figure 5.3 depicts the running time for the two real-valued functions $x \to \sin(x)$ and $x \to \frac{1}{1+x^2}$. Similar results were obtained for other functions.

Of course it's possible to compute those functions much faster, e.g. by using iRRAM's built-in sin function. However to obtain the above results no special properties of those functions were used, but only the information contained in the Taylor series and the two additional parameters. Thus, the results obtained are very general and should hold for all kind of functions (as long as the complexity to compute the series is similar).

5.4 shows how the running time depends on the number of differentiation.

Other operations like addition and multiplication yield similar running times and the results are therefore omitted.

All in all, the running times obtained for the tested examples were very close to what was expected from the theoretical analysis, and seem to be quite good, although due to the lack of similar approaches to compute analytic functions, it is hard to make any comparisons.

5.8.2 ANA_RECT

Since most operations on the power series on `ANA_RECT` and `BA_ANA` are identical, and are therefore expected to behave similarly, the crucial part of evaluating `BA_ANA` is the evaluation using analytic continuation.

In particular the following questions are interesting

1. How does the running time depend on the number of analytic continuations and on the output precision?
2. How many coefficients of each power series are needed?
3. How does the number of coefficients needed grow with the number of iterations and the output precision?

From the theoretical analysis, the expected behavior is that the running time for evaluating at a fixed point should depend polynomially on the desired output precision n, bounded by $O(n^4)$.

In the number of analytic continuations it should grow exponentially.

A similar dependency on output precision and number of analytic continuations should hold for number of coefficients that are read from the original series given to the constructor of the `ANA_RECT` object.

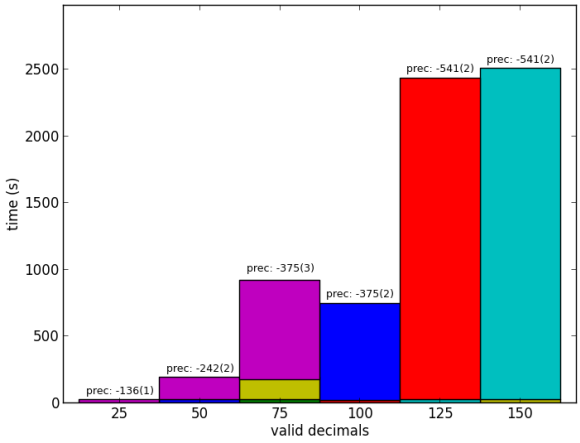


Figure 5.5: running time of `ANA_RECT` computing the sine function at the boundary of the third series, depending on the number of valid digits. The text above the bars shows the highest internal precision and the number of iterations `iRRAM` did.

Figure 5.6 shows, how the running time of evaluating at a fixed point x depends on the desired number of valid decimals. x is chosen so that 3 analytic continuations are necessary.

The plot also contains the information on how many iterations `iRRAM` performed and the maximum internal precision (text above the bars). Every bar is broken down into smaller bars that show how much time was spent in each iteration of the `iRRAM`.

The reason for the running time going down again between 75 and 100 decimals is that the iterations do not always have the same size, but `iRRAM` makes estimates on a good step size. The internal precision in the end is -375 for both $n = 75$ and $n = 100$, but for 75 the first estimate is too small, leading to a total number of 3 iterations instead of only 2 iterations for $n = 100$.

However, the plot also clearly indicates that the running time is always strongly dominated by the time spent in the last iteration.

Table 5.1 shows the number $N^{(l)}$ of coefficients accessed from the original series when evaluated at a point where l analytic continuations are necessary depending on the internal precision of `iRRAM`.

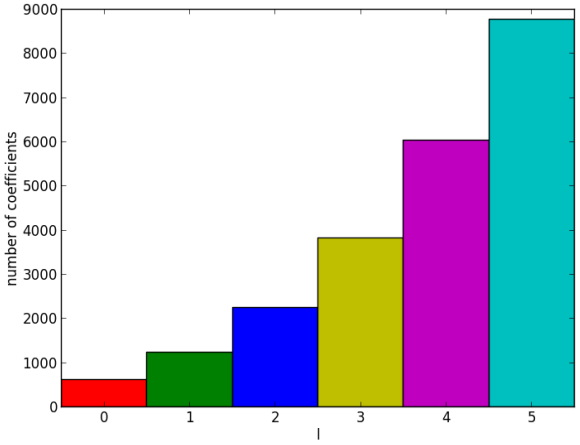


Figure 5.6: Number of coefficients read from the original series when evaluating sine-function with 150 valid decimals, depending on the number of analytic continuations

prec	$N^{(1)}$	sec
50	71	0
136	271	0
242	527	2
375	842	6
541	1235	20

(a) evaluating at series 1

prec	$N^{(0)}$	$N^{(1)}$	$N^{(2)}$	$N^{(3)}$	$N^{(4)}$
50	13	30	59	112	212
136	55	120	234	442	833
242	109	236	458	863	1626
375	175	378	733	1382	2603
541	258	556	1078	2031	3825

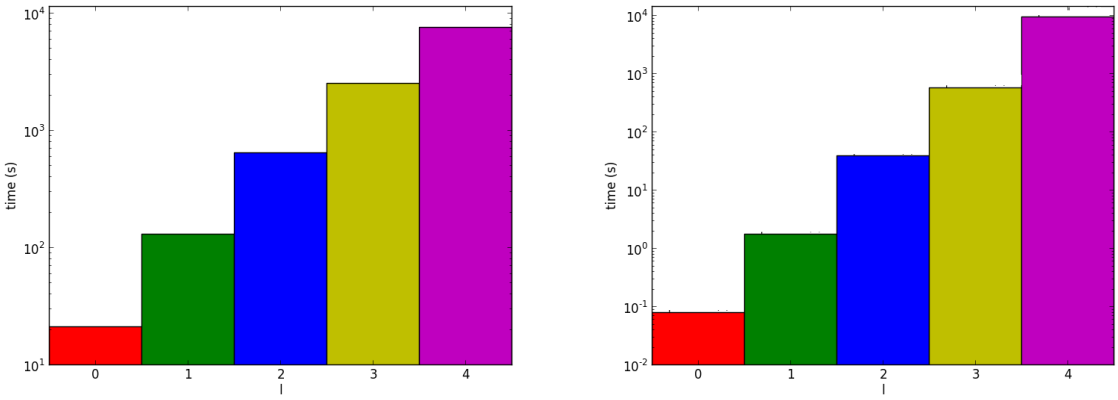
(b) evaluating at series 3

Table 5.1: Number of coefficients accessed in an `iRRAM` iteration of precision prec when evaluating at on the boundary of the first and third series.

How the running time and the number of coefficients depends on the number of analytic continuations can also be observed in Figure 5.7 and Table 5.2.

It can be seen that the number of coefficients read approximately doubles in each analytic continuation and thus grows a little slower than the upper bound established in the theoretical analysis.

All in all the figures are very well in accordance with the qualitative behavior expected from the analysis in the theory section.



(a) $\sin(x)$ **(b)** $\log(2+x)$

Figure 5.7: log-scale plot of the running time for `ANA_RECT` with 150 valid digits at the boundary of the l-th series.

precision	series 0	series 1	series 2	series 3	series 4	series 5	series 6
52	1914	1039	576	325	185	104	55
512	3721	2021	1121	633	361	204	109
2300	5978	3247	1801	1017	580	328	175
7550	8775	4766	2644	1493	851	481	257

Table 5.2: The number of coefficients read from each series depending on `iRRAM`'s precision. Results are for the sine function evaluated at the border of the series 6.

6 Conclusion

Two case studies have been presented, showing some of the possibilities and limitations of exact real arithmetic.

The first case study was a reimplementation of a algorithm on floating point numbers in exact real arithmetic. The automatic precision control in `iRRAM` made the implementation much simpler. Also, the `iRRAM` version makes it possible, to compute any point of the shadowing orbit with arbitrary little error, instead of just giving bounds for the position.

The second case study gave an example on how the theoretical results from computable analysis can be applied to yield practical algorithms. The empirical evaluation of the running time, showed that the bounds found with the help of real complexity theory correspond well with the implementation. This can be seen as evidence, that the models used in computable analysis map the reality quite well.

6.1 Possible Future Work

There are several possible extensions for both case studies. In general, it would be interesting to reimplement the algorithms in some other exact real arithmetic framework and compare the results.

Apart from that some possible extensions are listed below.

For the dynamical systems case studies the following is possible

1. **Forward algorithm**. An alternative to the algorithm implemented in the case study can be found in [CP91]. Instead of starting from the last point in the pseudo orbit and going backwards, this algorithm proceeds in forward direction. This algorithm could also be implemented in `iRRAM` and the results compared to the ones obtained in this thesis.
2. **Multi-dimensional shadowing**. Hammel, Yorke and Grebogi also briefly mention their results obtained for the two-dimensional case in their paper. They examined the Henon map

$$\begin{aligned} x_{n+1} &= 1 - a x_n^2 + y_n \\ y_{n+1} &= -J x_n \end{aligned}$$

   where J is the determinant of the Jacobian of the map. They found a similar shadowing result to the one mentioned in the case study for this map. In their implementation the true orbit is confined in a parallelogram to bound the position. `iRRAM` could be used to compute such an orbit exactly.

The data-type for analytic functions could be extended in the following ways

3. **Improvement of Running Time**. The current implementation of the data type for functions analytic on $[0,1]$ uses iterated analytic continuation to evaluate the function at points where the given Taylor series is not valid. This makes the representation simple, but the running time is exponential in the number of continuations. In practice, it is therefore not possible to have more than a few analytic continuations, Additional theoretical investigation should be done to find out, if it is possible to improve the running time, or if a high computational complexity is inherent to the problem.
4. **Multi-dimensional functions**. The implementation presented in this thesis only works with 1-dimensional functions. For many applications, however, the analytic functions are more-dimensional. The data-type should be extended to work with multi-dimensional Taylor series.
5. **Applications**. It would also be interesting, to use the implementation to solve a practical problem involving analytic functions. Such problems could for example be found in the theory of Ordinary Differential Equations.

Bibliography

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. IT Pro. Cambridge University Press, 2009.

[Ano67] Dmitry Victorovich Anosov. Geodesic flows on closed Riemannian manifolds of negative curvature. *Trudy Matematicheskogo Instituta im. VA Steklova*, 90:3–210, 1967.

[BHW08] Vasco Brattka, Peter Hertling, and Klaus Weihrauch. A Tutorial on Computable Analysis. *New Computational Paradigms: Changing Conceptions of What is Computable*, pages 425–491, 2008.

[Bow75] Rufus Bowen. ω-Limit sets for Axiom A diffeomorphisms. *Journal of Differential Equations*, 18(2):333–339, 1975.

[CP91] Shui Nee Chow and Kenneth J. Palmer. On the numerical computation of orbits of dynamical systems: The one-dimensional case. *Journal of Dynamics and Differential Equations*, 3(3):361–379, 1991.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[Fre05] Free Software Foundation. GNU GCC Manual. http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/, 2005.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. A Series of books in the mathematical sciences. W. H. Freeman, 1979.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic, 1991.

[HMU13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation: Pearson New International Edition*. Always learning. Pearson Education, Limited, 2013.

[HP08] B. Hasselblatt and Y. Pesin. Hyperbolic dynamics. 3(6):2208, 2008.

[HYG87] Stephen M. Hammel, James A. Yorke, and Celso Grebogi. Do numerical orbits of chaotic dynamical processes represent true orbits? *Journal of Complexity*, 3(2):136–145, 1987.

[iee08] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[J+10] Fredrik Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.14)*, February 2010. `http://code.google.com/p/mpmath/`.

[Kea96] R. Baker Kearfott. Interval Computations: Introduction, Uses, and Resources. *Euromath Bulletin*, 2:95–112, 1996.

[KH97] A. Katok and B. Hasselblatt. *Introduction to the Modern Theory of Dynamical Systems*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1997.

[KMRZ12] Akitoshi Kawamura, Norbert Mueller, Carsten Roesnick, and Martin Ziegler. Parameterized Uniform Complexity in Numerics: from Smooth to Analytic, from NP-hard to Polytime. *arXiv preprint arXiv:1211.4974*, 2012.

[Ko91] Ker-I Ko. *Complexity theory of real functions*. Progress in theoretical computer science. Birkhäuser, 1991.

[KP02] S. G. Krantz and H. R. Parks. *A Primer of Real Analytic Functions*. A Primer of Real Analytic Functions. Birkhäuser Boston, 2002.

[KST15] Akitoshi Kawamura, Florian Steinberg, and Holger Thies. Analytic Continuation in iRRAM: Implementations Inspired by Real Complexity Theory. In *LA Symposium, Winter 2015*, Kyoto, Japan, 2015.

[MK12] John Maddock and Christopher Kormanyos. Boost Multiprecision Library, 2012.

[Mül87] Norbert Th. Müller. *Uniform computational complexity of Taylor series*. Hagen Informatik. Fernuniv., 1987.

[Mül93] Norbert Th. Müller. Polynomial-Time computation of Taylor series. *Proc. 22 JAIIO-PANEL'93, Part 2, Buenos Aires*, 1993.

[Mül00] Norbert Th. Müller. The irram: Exact arithmetic in C++. In *Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers*, pages 222–252, 2000.

[PER89] M. B. Pour-El and J. I. Richards. *Computability in analysis and physics*. Perspectives in mathematical logic. Springer Verlag, 1989.

[Sch93] Joseph R. Schoenfield. *Recursion Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[Tur36] Alan Turing. On computable numbers with an application to the " Entscheidungsproblem". *Proceeding of the London Mathematical Society*, 1936.

[Wei00] Klaus Weihrauch. *Computable Analysis: An Introduction*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000.