hybr i2 d Theory of Hybrid Systems Informatik 2

RWTH AACHEN

Master of Science Thesis

Isolating Real Roots Using Adaptable-Precision Interval Arithmetic

Gereon Kremer

September 30, 2013

Supervisors:
Prof. Dr. Erika Ábrahám
Prof. Dr. Peter Rossmanith

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Aachen, den 30. September 2013

(Gereon Kremer)

Abstract

There are several decision procedures for the existential fragment of real algebra, for example the *cylindrical algebraic decomposition (CAD) method* or the *virtual substitution*. Both need efficient ways to *isolate real roots* of univariate polynomials and operate on these roots afterwards.
In this thesis we discuss different real-root representation techniques and isolation algorithms, and propose optimizations and efficient combinations of them to increase their applicability in real-algebraic decision procedures.
While real roots are traditionally isolated by bisection, we generalize this to an arbitrary partitioning. This allows for the integration of approximate methods, such as the *Aberth iteration* or the *companion matrix method*, in the first bisection step.
The real roots are usually represented by intervals with exact bounds. These are bounds represented by numbers of an arbitrary precision library. To speed up computations on such real roots, we explore the usage of *inexact intervals* that make use of native floating-point operations.
The content of this thesis is implemented and used in the CAD module of the SMT-RAT project. Experimental results in terms of comparisons between different configurations of the new algorithm and other algorithms are presented. While the new isolation algorithm proves to be an improvement for SMT-RAT, the performance of the CAD method as a whole still suffers from the preliminary integration of the new data types for inexact operations.

Contents

1 Introduction

Satisfiability checking (SAT) as well as its extension *Satisfiability modulo theories* (SMT) became widely used in practice in the last decades. While SAT checks formulas from the propositional logic for satisfiability, SMT combines a *SAT solver* with a *theory solver* to decide formulas from the existential fragment of first order logic over some theory.

1.1 Satisfiability Modulo Theories

The general architecture of an SMT solver is depicted in Figure 1.1. There are two major subroutines that are needed, a SAT solver and a theory solver for the underlying theory.

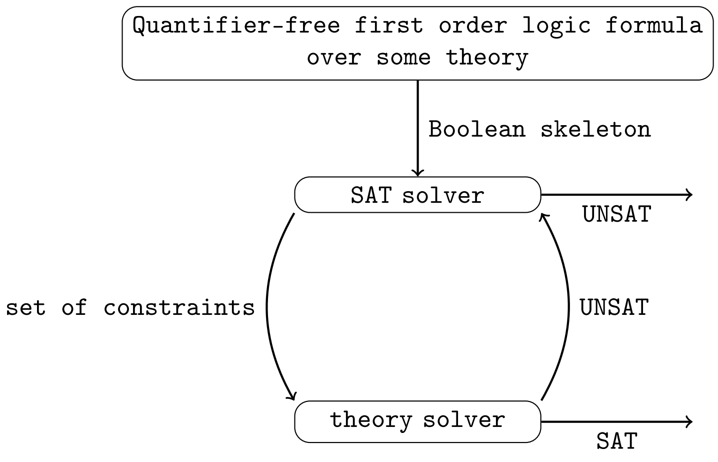

Figure 1.1: Architecture of a DPLL-style SMT solver

At first, the SMT solver transforms the input formula into *negation normal form* and pushes the negations into the constraints. Then it creates a *Boolean skeleton* from this formula replacing every constraint by a fresh Boolean variable. This results in a propositional formula that the SAT solver can check for satisfiability.

For every satisfying assignment found by the SAT solver, the theory solver checks whether this assignment is consistent in the theory. The assignment is transformed into a set of theory constraints by selecting all constraints from the original formula for which the corresponding Boolean variable is set to true by the current assignment. The theory solver can then check whether this combination of constraints is satisfiable in the theory.

The negation of those constraints whose corresponding Boolean variable is set to false need not be considered due to the special form of the formula. The formula would still satisfied, if such a Boolean variable was set to true.

If the combination of constraints is satisfiable, the whole formula is satisfiable and the SMT solver returns SAT. If the constraints are unsatisfiable, this result is returned to the SAT solver which searches for more satisfying assignments of the Boolean skeleton. Once the SAT solver cannot find further satisfying assignments, the whole formula is unsatisfiable and the SMT solver returns UNSAT.

Since different further issues like, for example, less lazy solving or the generation of explanations for unsatisfiability by the theory solver, are not closely related to this work, for a more detailed discussion refer to [KS08].

There are several theories that are supported by different SMT solvers. Examples are *linear real arithmetic* $(\mathbb{R}, 0, 1, +, <)$, *nonlinear real arithmetic* $(\mathbb{R}, 0, 1, +, \cdot, <)$ and *linear integer arithmetic* $(\mathbb{Z}, 0, 1, +, <)$. The focus of this thesis is in the nonlinear real arithmetic.

There exist several methods to decide this theory. Approaches like the *virtual substitution* are useful in practice but incomplete as there are always cases such that they cannot decide the satisfiability of a given problem.

A complete approach is the *cylindrical algebraic decomposition* (CAD) method, however it exhibits a doubly exponential running time in the number of variables. Our standard application throughout this thesis is the CAD method, but efficient methods for real root isolation are also used in the *virtual substitution* and in other decision procedures.

1.2 Cylindrical Algebraic Decomposition

Before explaining the idea of the CAD method, we first have a closer look at nonlinear real arithmetic. A constraint in canonical form has the form $p \sim 0$ with p being a polynomial expression and $\sim \in \{<, \leqslant, =, >, \geqslant\}$ a relation. As for the CAD the actual relation is not of interest, we only operate on the polynomials.

Given a set of polynomials P_n in n variables, a *cylindrical algebraic decomposition* is a partitioning of $\mathbb{R}^n$ into regions such that all polynomials are *sign invariant* on every region. A polynomial p is sign invariant on a region $R \subseteq \mathbb{R}^n$ if $p(r) \sim 0$ with $\sim \in \{<, >, =\}$ for all $r \in R$, that means the polynomial does not change its sign on R.

The consequence is that for any region, all points within this region are equivalent with respect to the satisfaction of real-arithmetic formulas containing only polynomials from P_n. If such a partitioning is computed, we only have to select an arbitrary *sample point* from each region and check whether this point is a solution to the problem.

A set of sample points therefore constitutes a finite representation of $\mathbb{R}^n$ with respect to the satisfiability of the given problem. To check the full infinite $\mathbb{R}^n$ for satisfiability, only these finitely many sample points have to be checked.

The *CAD method* is an algorithm to obtain such a partitioning. We do not explain this method in detail but only give a rough overview and show where real root isolations are involved. A visualization is given in Figure 1.2.

$$\begin{tikzcd}
P_n\,,\ |P_n| = k \arrow[d, "\texttt{project}"] & CAD_n \\
P_{n-1}\,,\ \mathcal{O}(k^2) \arrow[d, "\texttt{project}"] & \vdots \arrow[u, "\texttt{lift*}"'] \\
\vdots \arrow[d, "\texttt{project}"] & CAD_2 \arrow[u, "\texttt{lift*}"'] \\
P_1\,,\ \mathcal{O}(k^{2^n}) \arrow[r, "\texttt{sample*}"'] & CAD_1 \arrow[u, "\texttt{lift*}"']
\end{tikzcd}$$

Figure 1.2: Outline of the CAD method

The CAD method starts with a set P_n of polynomials in the variables $x_1, ..., x_n$ and iteratively eliminates the variables through projections and thereby constructs a new set P_{k-1} of polynomials in the variables $x_1, ..., x_{k-1}$ out of P_k until P_1 contains only univariate polynomials in the variable x_1. The projections preserves the roots of the polynomials such that if $\nu \in \mathbb{R}^k$ is a root of a polynomial from P_k, there is a $\nu' \in \mathbb{R}^{k-1}$ that is a root of a polynomial from P_{k-1} and ν equals ν' in the first $k-1$ dimensions.

These projections create a new polynomial for each pair of polynomials in P_n and hence produce quadratically many polynomials in each step. Therefore if $|P_n| = k$, then $|P_1| \in \mathcal{O}(k^{2^n})$.

For each univariate polynomial in P_1 the sample routine calculates one sample point for every sign-invariant CAD region. This set of sample points consists of the real roots, an additional point between each two neighboring roots, a point below the smallest real root and a point above the largest real root.

These real sample points are iteratively lifted to sample points in $\mathbb{R}^n$ by the lift operation. Given a polynomial $p \in P_k$ and a sample point $s \in \mathbb{R}^{k-1}$, lift substitutes s for $x_1, ..., x_{k-1}$ which results in a univariate polynomial. These substitutions require additions and multiplications on the sample points, hence we must provide such operations on the real roots.

Now sample is used once again to find sample points for these new polynomials. The sample point s is extended with all real sample points to new sample points in $\mathbb{R}^k$.

The sample routine, whose main part is finding the real roots, is called doubly exponentially often in the number of variables and hence optimizations at this point have a great impact on the overall performance. During the lifting, there are many operations on the roots, mostly the partial evaluation of polynomials, which can also be optimized.

1.3 Overview

In this thesis, we explore different ways to find real roots and analyze possibilities to speed up these methods as well as the subsequent operations on the roots in the lift routine of the CAD method.

The implementation of the described algorithms and optimizations was done within the *SMT-RAT* project [CLJA12], a toolbox that contains a SAT solver and several decision procedures for nonlinear real arithmetic, including *virtual substitution* and the *CAD method*. The root finding techniques described in this thesis were integrated into the CAD method but may also be used in other modules.

The CAD method, along with several other routines, is not implemented in SMT-RAT itself but in *GiNaCRA* [LA11], our extension to the *GiNaC* [BFK02] library.

We start in Chapter 2 with basic definitions of numbers, intervals and polynomials, and some general insights on how these structures may be represented in a computer.

Afterwards, in Chapter 3 we present several approaches for real root finding and highlight their benefits and drawbacks.

In Chapter 4 we continue with the operations on the real roots and see how we can perform them properly. An important point is performance, hence we look how we can use native floating point types maintaining correctness of our operations.

Some detail about the implementation is given in Chapter 5. We describe the software structure, present a few methods in more detail and explain a few decisions in our software design.

We have tested our implementation and compared to other options to measure performance and quality of the results. We present the results in Chapter 6.

Finally, we give a summary and the status of the implementation in Chapter 7. We then attempt an outlook about future work on this topic and possible applications outside of the CAD method.

The main contribution of this thesis are as follows:

- We propose optimizations and novel combinations of existing root finding algorithms for real root isolation.
- We dynamically connect two different representations for real roots, with the goal of decreasing the computational costs.
- We implement the proposed approaches within the SMT-RAT project.

1.4 Related Work

Solutions to find real roots of a polynomial, or a function in general, are mostly investigated in two different communities. On the one hand, there are *approximating* algorithms from numerical mathematics, on the other hand there are formal methods that need provable guarantees on the results. Combined approaches are rare, although they may be of interest for fast computations.

Approximating algorithms usually use a fixed-point iteration to refine approximations for the roots, usually also looking for complex roots. Examples are the Newton iteration, the Durand-Kerner method [Ker66], the Aberth method [Abe73] or the Jenkins-Traub algorithm [JT68]. Another non-iterative method is the companion matrix method [EM95].

The standard algorithm to find real roots in formal methods is bisection which in turn requires an estimation about the maximum size of a real roots, algorithms to count the number of real roots within a given interval and splitting heuristics for the bisection process.

Upper bounds on the absolute value of the real roots of a polynomial have been found, among others, by Cauchy [Cau28], Fujiwara [Fuj16] and Hirst and Macey [HM97].

As for the number of real roots within an interval, there are two popular possibilities. Sturm sequences [HM90] can be used to obtain the exact number of real roots whereas Descartes' rule of signs [CA76] yields an upper bound on the number of real roots.

The splitting heuristic is the essence of the bisection algorithm, hence there is some research about what constitutes a good splitting heuristic. Recent research analyzes the usage of the Newton method [Sag12, Ye94]. This thesis contributes several theoretical considerations about the usage of the Durand-Kerner method, the Aberth method and the companion matrix method as splitting heuristic and provides a few experimental results.

The representation of real roots by intervals is well-known also in the context of the CAD method [CJK02] and the possibility to use inexact computations for intervals has already been studied in [HJVE01, CJK02]. The necessity to switch between exact and inexact intervals on demand has been analyzed [She96, CJK02]. The scope of this thesis contains a practical implementation of this concept within the SMT-RAT [CLJA12] project.

2 Preliminaries

2.1 Representation of Numbers

Whenever we calculate with real numbers, we have two choices: Using the inexact native floating-point types or using a library offering an exact data type. The latter one is usually used whenever precision is crucial and the native precision is not enough.

Today almost all platforms support native floating-point arithmetic based on the IEEE 754 format [IEE08], usually called double. It offers eleven bits for the exponent and 52 bits for the mantissa, which fulfills most practical purposes.

However, for the correctness of our algorithms we often need exact computations. Hence, we perform many calculations using GiNaC [BFK02] which in turn uses CLN [HK].

CLN, the *Class Library for Numbers*, is a library that provides several arbitrary precision number types and efficient operations on them. Based on CLN is GiNaC, an acronym for *GiNaC is Not a CAS*, that provides a framework for symbolic operations and manipulations. We make use of these libraries to have consistent representations for numbers and polynomials and fast implementations for several operations.

CLN represents rational numbers as fractions using integers of arbitrary size. Using these numbers, we can represent every $q \in Q$, but we still cannot represent irrational numbers like $\sqrt{2}$.

It is possible to represent all algebraic numbers, the set of numbers that are roots of rational univariate polynomials, by arbitrary precision libraries. However, if we would represent such numbers by symbolic mathematical terms like $\sqrt{2}$, it would slow down the calculations significantly. Furthermore, we may be able to represent all possible roots, but we still may not be able to find them.

In the following, we call CLN numbers *exact* and double values *inexact*.

2.2 Intervals

We use a representation for real roots based on intervals.

Definition 2.1. *An* interval $I \subseteq \mathbb{R}$ *is a connected subset of the reals. Although there are several shapes of an interval, we only use two special types. For* $a, b \in \mathbb{R}$ *we define*

$$\begin{aligned}(a, b) &= \{x \in \mathbb{R} \mid a < x < b\} \text{ and} \\ [a, b] &= \{x \in \mathbb{R} \mid a \leqslant x \leqslant b\}.\end{aligned}$$

We call a and b the lower bound *and* upper bound *of the interval,* (a, b) *an* open interval *and* $[a, b]$ *a* closed interval.

An interval $I = [a, b]$ with $a = b$ is called a point interval.

Unless stated otherwise in the following, we only deal with open intervals or point intervals.

2.3 Polynomials

The most important structures throughout this work are polynomials.

Definition 2.2. *Let $x = \{x_1, ..., x_k\}$ for some $k \in \mathbb{N}$ be a set of variables and K a field. We define a* term *t as*

$$t = a \cdot m = a \cdot \prod_{i=1}^{k} x_i^{d_i}$$

where $d_1, ..., d_k \in \mathbb{N}$, m is a monomial *of degree* $\deg(m) = \sum_{i=1}^{k} d_i$ *and $a \in K$ the* coefficient *of m. A* polynomial *p is a sum of terms $t_1, ..., t_l$ and* $\deg(p) = \max_{i=1,...,l} \deg(t_l)$.

If the number of variables $k = 1$, a polynomial is called univariate*, otherwise* multivariate. *We denote the* polynomial ring *over K in the variables x by $K[x]$.*

In this thesis $K = \mathbb{Q}$ and all polynomials are called *rational.* In the following we write $p(x)$ to denote that $p \in \mathbb{Q}[x]$ and for some $v \in \mathbb{R}$ we write $p(v)$ for the evaluation $p[v/x]$ of p at v, that is substituting v for x in p.

Most of the time, we deal only with univariate polynomials. A univariate polynomial p of degree $\deg(p) = n \in \mathbb{N}$ can be written as

$$\sum_{j=0}^{n} a_j \cdot x^j$$

with $a_0, ..., a_n \in \mathbb{Q}$ and $a_n \neq 0$. Note, that the coefficients are the only data we need to store to uniquely represent a univariate polynomial.

For a univariate polynomial $p \in \mathbb{Q}[x]$, we define p' as the derivative of p.

As the evaluation of a univariate polynomial p at a given point v is used very frequently, we have a closer look at its complexity. Calculating $p(v)$ naively yields an algorithm that needs $n - 1$ multiplications to obtain $v, v^2, ..., v^n$ and another n multiplications for $a_1 v, a_2 v^2, ..., a_n v^n$. Including the following additions, we end up with $2n - 1$ multiplications and n additions.

However, using *Horner's method* we are able to calculate $p(v)$ using only n multiplications and n additions. It works as follows: Starting with $b_n = a_n$, we iteratively compute $b_{k-1} = a_{k-1} + b_k \cdot v$. Finally, $b_0 = p(v)$.

Definition 2.3. *A* real root *of a univariate polynomial $p \in \mathbb{Q}[x]$ is a number $\nu \in \mathbb{R}$ such that $p(\nu) = 0$. The* set of real roots *of p is given by $roots(p) = \{x \in \mathbb{R} \mid p(x) = 0\}$ and the* number of roots *by $\#^{roots}(p) = |roots(p)|$.*

Please note that $\#^{roots}(p)$ is bounded by $\deg(p)$. We write $roots_{(a,b)}(p)$ and $\#^{roots}_{(a,b)}(p)$ for the real roots of p and the number of real roots of p within the interval (a, b). They are defined by $roots_{(a,b)}(p) = \{x \in (a, b) \mid p(x) = 0\}$ and $\#^{roots}_{(a,b)}(p) = |roots_{(a,b)}(p)|$.

We know that for each real root ν of p, we can find a polynomial p^* such that $p(x) = (x - \nu) \cdot p^*(x)$. We call $(x - \nu)$ a *linear factor* of p. Hence, we can decompose p into

$$p(x) = (x - \nu_1)^{i_1} \cdots (x - \nu_n)^{i_n} \cdot p^*(x)$$

where $\{\nu_1, ..., \nu_n\} = roots(p)$, $i_1, ..., i_n \in \mathbb{N}$ and p^* is a polynomial that has no real roots. We call i_k the *multiplicity* of the root ν_k and call ν_k a *multiple root* if $i_k > 1$. Note that $\#^{roots}(p)$ counts the number of *distinct* roots.

2.4 Representation of Real Roots

Considering the possibilities to represent a real root, any root should usually be stored as an exact number, as any operation on an inexact number might produce rounding errors and therefore yields a wrong result.

Although exact numbers give us the theoretical possibility to calculate such roots to an arbitrary precision, it leaves two problems to solve: The mere possibility to represent it exactly and, if we can represent it, the costs to find it.

Operations on exact numbers are expensive. No matter how efficient the implementation might be, it cannot compete with inexact computations that are hard-wired in the CPU. Therefore, an inexact representation of a root with a guaranteed error bound seems reasonable.

Also, already the simple polynomial $x^2 - 2$ has irrational roots, that are roots not representable by a fraction. We may be able to calculate a value that is arbitrarily close to the exact value and we may as well be able to obtain the exact value for polynomials of small degree, given that we can represent algebraic numbers, but there is no general way to find the exact value. As we are bound to guarantee correctness, we must also handle such cases.

A possible solution for both problems is an *interval representation* for real roots. It consists of a polynomial and an interval such that the polynomial has exactly one root within this interval.

Definition 2.4. *Let $p \in \mathbb{Q}[x]$ and $\nu \in \mathbb{R}$ a real root of p. An* interval representation *of ν is a pair (p, I) such that I is an open interval or a point interval, $\nu \in I$ and there is no $v^* \in I$ with $v^* \neq \nu$ and $p(v^*) = 0$.*

We also use the terms *exact root* for a real root ν and *interval root* for an interval representation of a real root. Note that an interval root that consists of a point can be converted easily to an exact root.

Definition 2.5. *Let $\nu_1 = (p, I_1)$ and $\nu_2 = (p, I_2)$ be interval roots. We call ν_2 a* refinement *of ν_1, if I_2 is a strict subset of I_1.*

We also call a method that constructs a refined interval root ν_2 from a root ν_1 a *refinement*.

3 Isolating Real Roots

A basic task in many applications is to find all real roots of some polynomial. There are several fundamentally different solutions to this problem. However, considering that a real root might not even be representable by the number representation in use, the meaning of *finding all real roots* is still unclear. In our case, we only need to *isolate* real roots.

3.1 Isolation of Real Roots

Given a univariate polynomial $p \in \mathbb{Q}[x]$, isolating the real roots means to find interval representations for all real roots of p.

Definition 3.1. *Let $p \in Q[x]$ with $roots(p) = \{\nu_1, ..., \nu_n\}$. A* set of isolated roots *is a set $\{I_1, ..., I_n\}$ of disjoint intervals such that I_k is an interval representation of the real root ν_k for all $k = 1, ..., n$.*

Note that we can always construct isolated roots from exact roots or sufficiently precise approximations of the exact roots. This may be possible using guarantees on the error bounds provided by a specific algorithm or a combination of expansions and refinements of intervals around every root approximation.

In the following, we use the term *root finding* synonymously for isolating roots.

3.2 Optimal Root Finding

When comparing different root finding algorithms, we want to reason about their quality. As there is not a unique correct result but a infinite number of them, we are interested in the *performance* of the methods and the *quality* of their results.

Performance is a property that is conceptually easy to measure and to compare. The quality of the results is rather fuzzy and depends heavily on how the results are used afterwards. We give a metric to assess it.

The first observation is, that an exact root is better than an interval root in most cases, as an interval root in general requires more basic computations for a single operation on the root. However, operations on an interval root that uses inexact bounds might still be faster than the same operations on an exact root represented by an exact number with a large internal representation.

Finding an exact root also provides the opportunity to reduce the polynomial by eliminating the root through polynomial division. Given a root ν of a polynomial p, we know that $(x - \nu)$ is a factor of p. Hence there is a polynomial $p^*(x) = \frac{p(x)}{x-\nu}$ and for the set of roots it holds that $roots(p)\backslash\{\nu\} \subseteq roots(p^*) \subseteq roots(p)$, that means that all other roots of p are also roots of p^*. Note that ν may still be a root of p^*, if it was a multiple root of p.

Polynomial division can be performed quickly, as we only divide by linear factors of the form $x - \nu$. At the same time it speeds up all further operations on the polynomial, like evaluating it at some point, which is required in most high-level operations on a polynomial.

Interval roots should have small intervals. During the CAD method, all interval roots must stay disjoint, otherwise they must be refined. Smaller intervals are more likely to stay disjoint during arbitrary operations and thus are less likely to trigger a refinement process.

Since most operations need super-linear time in the size of the representation of the interval root, the size of the representation of an interval root is crucial. Hence, we must try to keep this size small.

3.2.1 Preprocessing

To avoid unnecessary work, we try to reduce a polynomial by eliminating all multiple roots from this polynomial. Multiple roots not only bloat our polynomial, they may also introduce problems with iterative algorithms: Most such algorithms converge significantly slower towards a multiple root and may be numerically instable.

However, we can eliminate multiple roots easily. If ν is a multiple root of a polynomial p, the derivative p' of p shares this root with the multiplicity decreased by one. Therefore, the greatest common divisor of p and p' contains all redundant linear factors and hence

$$p^*(x) = \frac{p(x)}{gcd(p, p')(x)}$$

produces a polynomial with the same real roots as p but without any multiple roots. From now on we only consider univariate polynomials without multiple roots.

3.3 Algorithms to Find Real Roots

There is a large number of algorithms dealing with the problem of root finding. In most scenarios, the goal is to get a good *approximation* of the roots, not finding exact *roots*. However, many of those algorithms also provide adjustable error bounds, such that they might be applicable in our context.

Next we give a rough overview of a few important root finding algorithms. Our assumption is, that we do not have any knowledge about the roots, especially no approximate values or knowledge about their distribution over some interval. However, we can compute bounds on the absolute values of all roots and the overall number of distinct real roots with little effort.

3.3.1 Newton iteration

Already discovered more than 300 years ago, the *Newton iteration* is probably the best-known algorithm to find real roots of arbitrary differentiable functions. It consists of a simple fixed-point iteration:

$$z^{k+1} = z^k - \frac{f(z^k)}{f'(z^k)}.$$

Given an initial approximation z^0, this iteration converges to a real root of the function f. The speed of convergence is quadratic, meaning that the number of correct decimals doubles in every step, if the initial approximation is "close enough" to the actual root. If the initial approximation is bad, this algorithm may converge slowly or even diverge.

However, due to our problem setting, we cannot provide any initial approximation. The usual approach is sampling: Initial approximations are chosen at random until the iterations converge to some root.

Unfortunately, divergence is not only a theoretical issue but may occur also on simple functions. Example 3.1 presents a case where the iteration gets stuck in an infinite loop between two values. The only solution here is to stop after a fixed number of iterations and continue with another initial approximation.

Example 3.1. *Let $p(x) = x^3 - 2 \cdot x + 2$ and hence $p'(x) = 3 \cdot x^2 - 2$. We try to find a real root of p using Newton iterations. We do not have any prior knowledge about the real roots of this polynomial, hence we decide to start with the initial approximation $z^0 = 0$.*

The first iteration produces $z^1 = 0 - \frac{2}{-2} = 1$. However, iterating a second time gives us $z^2 = 1 - \frac{1}{1} = 0$, that means we get our initial approximation again after the second iteration. This process is illustrated in Figure 3.1.

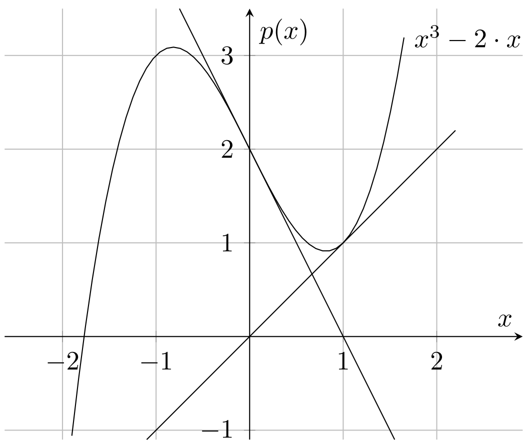

Figure 3.1: Example for a diverging Newton iteration

We can see that the Newton iterations may get stuck in an infinite loop already for simple polynomials using seemingly reasonable initial approximations.

Theoretical analysis has shown that each root has a set of numbers called *basin of attraction* [Den97]. The iteration converges against the root, if and only if the initial approximation is contained in its basin of attraction. Those sets can be arbitrarily small, hence finding all roots for a given polynomial can be very expensive.

There are several modifications of the original Newton iteration that reduce the computational effort, for example the *regula falsi method*, or provide faster convergence using further derivatives like *Halley's method* leading to the class of *Householder's methods* [Hou70]. Although these modifications may reduce the probability of divergence and accelerate the convergence, we give no further details here, as there are conceptually better methods available.

3.3.2 Aberth Method

The *Aberth method* [Abe73] is a way to find multiple roots at once and is similar to multiple Newton iterations performed in parallel. As an advantage to multiple independent Newton iterations, Aberth modifies the iteration for an approximation such that two approximations are unlikely to converge against the same root.

While Aberth is intended for complex arithmetic, we show that it can also be used on real numbers by selecting real initial approximations. We now start with an outline of the Aberth method on complex numbers.

The basic idea of an *Aberth iteration* is the following: We assume that the complex approximations $z_1, ..., z_n$ for the roots are good and therefore

$$p(x) \approx (x - z_1) \cdots (x - z_n).$$

Then we can divide p by the linear factors and thus get a linear polynomial that is almost identical to the linear factor of a single root.

$$(x - z_1) \approx \frac{p(x)}{(x - z_2) \cdots (x - z_n)}.$$

If we use this function instead of p for a Newton step, we make sure that this step does not converge towards a root that is already approximated by another z_k. The general form for a function used to approximate z_i is as follows:

$$F_i(x) = \frac{p(x)}{\prod_{j=1, j \neq i}^{n} (x - z_j)}.$$

An *Aberth step* consists of a Newton iteration for each approximation z_i. We can transform this to obtain a single formula for an *Aberth iteration* for a root z_i:

$$\begin{aligned}
z_i^{k+1} &= z_i^k - \frac{F_i(z_i^k)}{F_i'(z_i^k)} \\
&= z_i^k - \left(\frac{F_i'(z_i^k)}{F_i(z_i^k)}\right)^{-1} && \left| \frac{d}{dx}\ln|f(x)| = \frac{f'(x)}{f(x)} \right. \\
&= z_i^k - \left(\frac{d}{dz_i^k}\ln|F_i(z_i^k)|\right)^{-1} \\
&= z_i^k - \left(\frac{d}{dz_i^k}\left(\ln|p(z_i^k)| - \sum_{j=1,j\neq i}^{n}\ln|z_i^k - z_j^k|\right)\right)^{-1} \\
&= z_i^k - \left(\frac{p'(z_i^k)}{p(z_i^k)} - \sum_{j=1,j\neq i}^{n}\frac{1}{z_i^k - z_j^k}\right)^{-1} \\
&= z_i^k - \frac{1}{\frac{p'(z_i^k)}{p(z_i^k)} - \sum_{j=1,j\neq i}^{n}\frac{1}{z_i^k - z_j^k}} \\
&= z_i^k - \frac{\frac{p(z_i^k)}{p'(z_i^k)}}{1 - \frac{p(z_i^k)}{p'(z_i^k)}\cdot\sum_{j=1,j\neq i}^{n}\frac{1}{z_i^k - z_j^k}}.
\end{aligned}$$

Given reasonably good initial approximations, this method usually converges to the n complex roots of the polynomial p. The method terminates when the change of all approximations falls below a certain limit.

However, we are only interested in real roots. We can either apply this method on complex numbers and then use those results that are "almost real" as an approximation for real roots. These could be refined again by standard Newton iterations.

Another approach is to use only initial approximations from the real numbers. Note that the algorithm never produces any complex numbers in this case, hence we can perform all operations on real numbers. If we ensure that we only have as many approximations as the polynomial has real roots – we see in Section 3.4.2 how this is possible – the Aberth method approximates all real roots.

We assume for now that we have n approximations for a polynomial with n real roots. Then our initial modification to the polynomial changes to

$$(x - z_1)\cdot\hat{p}(x) \approx \frac{p(x)}{(x - z_2)\cdots(x - z_n)},$$

where $\hat{p}(x)$ is some polynomial without any real roots. While this modification may slow down convergence, the Newton iteration on the polynomial $(x - z_1)\cdot\hat{p}(x)$ still converges as it has only a single real root.

Note that this method suffers from the same problems as the Newton method in general, even though they are unlikely to occur in practice. The intervals for approximations leading to infinite loops are relatively small compared to the sections that lead to a successful iteration. Since the polynomial itself changes a lot in the first few iterations, it is unlikely that an approximation ends up in such an infinite loop once the other approximations are good.

Additionally such a bad approximation alters the polynomial for all other approximations heavily, all approximations change significantly as long as some other approximation is stuck in an infinite loop. This further reduces the probability that this happens at all.

There are several ways to generate initial approximations based on some upper bound on the absolute values of the roots. The approximations are usually distributed within the interval given by such an upper bound according to a heuristic like described in [TS02].

3.3.3 Other Iterative Methods

We now give a short overview over other iterative methods and their applications.

The *Durand-Kerner method* [Ker66], like the Aberth method, is also designed for complex roots and relies on the possibility to decompose a polynomial into linear factors. Given a polynomial p and approximations $z_1, ..., z_n$, we obtain the decomposition

$$p(x) = (x - z_1) \cdots (x - z_n).$$

A *Durand-Kerner iteration* is the following fixed-point iteration:

$$z_i^{k+1} = z_i^k - \frac{p(z_i^k)}{\prod_{j=1, j \neq i}^{n} (z_i^k - z_j^k)}.$$

Like the Aberth method, Durand-Kerner performs *steps* consisting of an iteration for each approximation z_i^k.

However, there is no obvious way to perform this on real numbers. Using only approximations of real roots, oftentimes less than the degree of the polynomial, completely breaks the algorithm. Using the correct number of approximations from the real numbers does not work either, as an approximation cannot converge to a complex root, as it cannot leave the real numbers and it is also pushed away from the real roots similar to the Aberth method. Hence, the complete iteration process cannot converge.

Performing the algorithm on complex numbers is significantly slower, as every operation on a complex number decomposes into multiple basic operations. Additionally, we have to decide which of the resulting approximated complex roots are real roots.

Finally, Durand-Kerner also requires additional parameters like a lower bound for the errors to handle termination and some way to obtain initial approximations.

A very popular algorithm is the *Jenkins-Traub algorithm.* It has the nice property that it always converges, but it may fail to find all roots. As it is quite complex, in terms of mathematical background and in the computational effort to obtain a result, we have not further investigated whether this algorithm may be of use for our application.

There are several further variants and optimizations for each of these iterative methods, which we do not use in this work.

3.3.4 Companion Matrix Method

A common problem in mathematics and many physical applications is the calculation of the eigenvalues of a matrix. We show how an approach to obtain the eigenvalues of a matrix can be reversed and thereby used to calculate the real roots of a polynomial.

Definition 3.2. *Let $A \in \mathbb{R}^{n\times n}$ be a square matrix. The* eigenvalues *of A are $\lambda_k \in \mathbb{C}$ and the* eigenvectors *are vectors $v_k \in \mathbb{C}^n$ such that $A \cdot v_k = \lambda_k \cdot v_k$ for every k. There are n eigenvalues and eigenvectors for every matrix $A \in \mathbb{R}^{n\times n}$, hence $k = 1, ..., n$.*

The characteristic polynomial *of an $n \times n$ square matrix A is defined as*

$$p_A(x) = \det(x \cdot I - A),$$

where I is the identity matrix. For the real roots of p_A it holds that $roots(p_A) = \{\lambda_k \mid k = 1, ..., n\}$.

This approach can be reversed: We construct a *companion matrix* for a given polynomial which is a matrix such that the characteristic polynomial of this matrix is identical to the original polynomial. This process is illustrated in Figure 3.2.

$$\begin{tikzcd}
\texttt{Matrix } A \arrow[r, shift right, "\texttt{characteristic polynomial}"'] \arrow[d, dash] & p(x) \arrow[l, shift right, "\texttt{companion matrix}"'] \arrow[d, dash] \\
\texttt{eigenvalues}(A) \arrow[r, leftrightarrow, "="] & roots(p)
\end{tikzcd}$$

Figure 3.2: Relation between eigenvalues and real roots

Definition 3.3. *Let $p(x) = a_n \cdot x^n + \cdots + a_0$ be a univariate polynomial. A* companion matrix *of p is a matrix $A \in \mathbb{R}^{n\times n}$ such that $p_A(x) = p(x)$.*

There are infinitely many possibilities to create such a matrix, one possible construction for a companion matrix from a polynomial p is as follows:

$$\begin{pmatrix}
0 & 0 & \cdots & 0 & -a_0/a_n \\
1 & 0 & \cdots & 0 & -a_1/a_n \\
0 & 1 & \cdots & 0 & -a_2/a_n \\
\vdots & \vdots & \ddots & & \vdots \\
0 & 0 & \cdots & 1 & -a_{n-1}/a_n
\end{pmatrix}$$

We can now apply different methods to calculate the eigenvalues of the companion matrix. These methods do neither require initial approximations nor error bounds to handle termination and therefore seem to be well-suited for our application. However, these algorithms may be numerically unstable and hence not reliable. Nevertheless, we can use such methods to obtain good initial approximations without the hassle of possible nontermination or divergence.

Another advantage of this method is a practical one: Calculating the eigenvalues of a matrix is a frequent and well-understood problem. Therefore, routines to calculate eigenvalues are part of the BLAS [Don02] standard and as such are part of several very stable and efficient libraries. Using well-established implementations is much easier and most probably much more efficient than implementing it on our own.

However, the eigenvalues may as well be complex, hence we must guess which of them correspond to our real roots. This makes it difficult to use this method to find real roots directly, but we can obtain approximations for the real roots. In this application we may return more guesses than real roots exist and thus use the real part of every eigenvalue as a guess.

3.4 Isolation by Bisection

Bisection is a completely different approach to isolating real roots. The basic idea is to start with an interval containing all real roots and split it recursively until each real root is contained in a different interval. While bisection usually means splitting an interval into exactly two new intervals, we use the term bisection to describe an arbitrary partitioning of an interval into multiple disjoint new intervals. We formalize this in Algorithm 1.

```
Input: Polynomial $p$
$Q \leftarrow \{\text{ContainingInterval}(p)\}$
$R \leftarrow \{\}$
while $\exists I \in Q$ do
  $Q \leftarrow Q \setminus \{I\}$
  if $\text{roots}(I) = 1$ then
    $R \leftarrow R \cup \{I\}$
  else if $\text{roots}(I) > 1$ then
    $Q \leftarrow Q \cup \{I_0, ..., I_k\}$ such that $\dot\bigcup_{i=0}^{k} I_i = I$
  end if
end while
return Isolated roots $R$
```

Algorithm 1: Pseudocode for bisection

Although this might look like a very natural way to isolate real roots in a recursive fashion, it poses a few problems. To start with bisection, we must calculate an interval that is guaranteed to contain all real roots and, to manage the recursion, we must be able to calculate the number of roots within an interval. Furthermore, we must ensure an exact bisection such that there are no regions contained in several intervals or not contained in any interval at all.

3.4.1 Upper Bounds for Real Roots

There are several possibilities to obtain an interval containing all real roots of a given polynomial. Most methods give an upper bound α on the absolute value $|\nu|$ of any real root and thereby yield an initial interval.

Note that such a bound only gives a closed interval $[-\alpha, \alpha]$, but as we see in Section 3.4.2, we may run into trouble if α is a root. To be on the safe side, we check $-\alpha$ and α separately and reduce the polynomial as described in Section 3.2 if possible.

Common bounds are the *Cauchy Bound* [Cau28]

$$|\nu| \leqslant \max\left\{1 + \left|\frac{a_0}{a_n}\right|, ..., 1 + \left|\frac{a_{n-1}}{a_n}\right|\right\} = 1 + \frac{\max\{|a_0|, ..., |a_{n-1}|\}}{|a_n|},$$

the *Hirst and Macey Bound* [HM97]

$$|\nu| \leqslant \max\left\{1, \sum_{i=0}^{n-1}\left|\frac{a_i}{a_n}\right|\right\},$$

or the *Fujiwara Bound* [Fuj16]

$$|\nu| \leqslant 2 \cdot \max\left\{\left|\frac{a_{n-1}}{a_n}\right|, \left|\frac{a_{n-2}}{a_n}\right|^{\frac{1}{2}}, ..., \left|\frac{a_1}{a_n}\right|^{\frac{1}{n-1}}, \left|\frac{a_0}{2 \cdot a_n}\right|^{\frac{1}{n}}\right\}.$$

All these roots are fairly easy to compute, hence we can even calculate multiple bounds and choose the smaller one for the initial interval.

3.4.2 Counting Roots within an Interval

The most important and computationally intensive task is counting the number of real roots within a given interval. To perform bisection, we do not need to know the exact number of roots within an interval. We only need to distinguish between zero, exactly one and at least two roots. We have a closer look at two possibilities to solve this problem. Both methods analyze *sign changes* of sequences of numbers.

Definition 3.4. *Let $(x_0, ..., x_k) \in \mathbb{R}^{k+1}$, $k \in \mathbb{N}$, be a sequence of numbers. The number of* sign changes *$\sigma(x_0, ..., x_k)$ over this sequence is defined recursively as*

$$\begin{aligned}
\sigma(x_0) &= 0\\
\sigma(x_0, x_1) &= \begin{cases} 1 & \textit{if } x_0 < 0 < x_1 \textit{ or } x_1 < 0 < x_0\\ 0 & \textit{otherwise}\end{cases}\\
\sigma(x_0, ..., x_k) &= \begin{cases} \sigma(x_0, x_2, ..., x_k) & \textit{if } x_1 = 0\\ \sigma(x_0, x_1) + \sigma(x_1, ..., x_k) & \textit{otherwise.}\end{cases}
\end{aligned}$$

Note that zero is considered neutral with respect to the sign change.

Sturm sequences A *Sturm sequence* of a polynomial p is a sequence of polynomials $p_0, ..., p_k$ with decreasing degree and $p_0 = p$ which fulfills the following properties:

- If $p(v) = 0$, then $sign(p_1(v)) = sign(p'(v))$ for all $v \in \mathbb{R}$.
- If $p_i(v) = 0$, then $sign(p_{i-1}(v)) = -sign(p_{i+1}(v))$ for all $0 < i < k$ and $v \in \mathbb{R}$.
- p_k has no sign changes, that means $p_k(v) \geqslant 0$ for all $v \in \mathbb{R}$ or $p_k(v) \leqslant 0$ for all $v \in \mathbb{R}$.

The number of distinct real roots of p within the *half-open* interval $(a, b]$ is then equal to $\sigma(p_0(a), ..., p_k(a)) - \sigma(p_0(b), ..., p_k(b))$. We restrict ourselves to open intervals (a, b) and make sure that a and b are no real roots of p. In case they are, we can reduce p by polynomial division as described in Section 3.2.

A Sturm sequence can be computed using Euclid's algorithm on p and its derivative p'. Starting with $p_0(v) = p(v)$ and $p_1(v) = p'(v)$, we continue with

$$p_i(v) := -rem(p_{i-2}, p_{i-1}) = p_{i-1}(v) \cdot q_{i-2}(v) - p_{i-2}(v)$$

where $rem(p_{i-2}, p_{i-1})$ is the remainder of the polynomial division of p_{i-2} and p_{i-1} and q_{i-2} its quotient.

Each iteration can be performed in at most $\mathcal{O}(n^2)$ using the Euclidean algorithm. As the degree of p decreases with each iteration, the overall complexity is at most $\mathcal{O}((n+1)^2)$.

To obtain the number of real roots for a single interval (a, b), we need to calculate $p_i(a)$ and $p_i(b)$ for each $i = 0, ..., k$ with $k \in \mathcal{O}(n)$. Evaluating a polynomial needs $\mathcal{O}(n)$ operations, hence we need $\mathcal{O}(n^2)$ operations overall. Counting the sign changes is linear and thus can be omitted. Note that the time needed for a single operation is by no means constant if we make use of exact representations for numbers. Hence, these considerations are only useful to compare methods using approximately the same basic operations.

Descartes' rule of signs There is another method that is, probably wrongly, attributed to Descartes to calculate an upper bound $var(p)$ on the number of positive real roots of a univariate polynomial p. Note that this rule gives an upper bound on the number of root including multiple roots, but as we always remove multiple roots as described in Section 3.2.1 we can use it as an upper bound on the number of distinct roots.

Descartes' rule of signs states that for $var(p) = \sigma(a_0, ..., a_n)$, where $a_0, ..., a_n$ are the coefficients of p, it holds that $var(p) \geqslant roots_{(0,\infty)}$ and $var(p) = roots_{(0,\infty)} \pmod 2$.

In order to apply this method for our purpose, we need to transform our polynomial p into a new polynomial p^* such that $\#^{roots}_{(0,\infty)}(p^*) = \#^{roots}_{(a,b)}(p)$ for a given interval $I = (a, b)$. To construct such a p^*, we use a continuous bijection $\varphi : (0, \infty) \to (a, b)$ on p such that $p(\varphi(x))$ is a function that behaves on $(0, \infty)$ like p on (a, b) and in particular possesses the same number of roots.

Though $p(\varphi(x))$ is not necessarily a polynomial, we are able to fix this with an additional factor $f(n)$. This factor does neither change the number, nor the position of the roots within $(0, \infty)$ and hence $p^*(x) := f(n) \cdot p(\varphi(x))$ fulfills $\#^{roots}_{(0,\infty)}(p^*) = \#^{roots}_{(a,b)}(p)$.

A well-known possibility to construct such a φ are Möbius transformations. Their general form is

$$\psi(x) = \frac{\alpha \cdot x + \beta}{\gamma \cdot x + \delta}.$$

We fix $\varphi(0) = a$ and $\lim_{x\to\infty} \varphi(x) = b$ and hence get $\frac{\beta}{\delta} = a$ and $\frac{\alpha}{\gamma} = b$ immediately. We now set $\gamma = \delta = 1$ and get

$$\varphi(x) = \frac{b \cdot x + a}{x + 1}.$$

Applying $p(\varphi(x))$ produces a factor of $\frac{1}{(x+1)^i}$ for each monomial $a_i \cdot x^i$. We remove the occurrences of x in the denominators by multiplying the result with $f(n) = (x + 1)^n$.

This polynomial $p^*(x) = (x + 1)^n \cdot p(\frac{b \cdot x + a}{x+1})$ with $\deg(p^*) = \deg(p) = n$ can be constructed efficiently in time $\mathcal{O}(n^2)$ as described in [Sag12]. To get the upper bound on $\nu_{(0,\infty)}(p^*)$, we must only compute the number of sign changes $\sigma(a_0, ..., a_n)$ of the coefficients of p^*.

Once again, like for the complexity of Sturm sequences, the basic operations may not have constant running time if exact number representations are used. We chose to do this simplified analysis as we only want to compare these two methods. Since they use similar basic operations, this analysis is sufficient for a meaningful comparison.

There is still one problem left to solve: Descartes' rule of signs only gives an upper bound, hence it may always return two while there is no real root in the given interval resulting in nontermination of the bisection. However, there exist bounds on the minimum distance between two roots of a polynomial. If an interval is smaller than this bound, we know that the number of roots within this interval is either zero or one. We can decide which one, as Descartes' rule of signs returns a result which is exact modulo two.

Although such a bound is important to guarantee termination, it turns out that these bounds are only necessary in rare cases. In fact, we did not encounter such a case during our tests. Nevertheless, we present a few bounds here taken from [Col01].

Let p be a polynomial of degree $\deg(p) = n$ and $\bar{a} = \max_{i=0}^{n} a_i$ the maximal coefficient of p. A lower bound for the minimal separation distance *sep*(p) of two roots due to *Collins* and *Horrowitz* is

$$sep(p) > \frac{1}{2} \cdot e^{\frac{-n}{2}} \cdot n^{\frac{-3n}{2}} \cdot \bar{a}^{-n}$$

and later *Mignotte* proved

$$sep(p) > \sqrt{6} \cdot n^{\frac{-n-1}{2}} \cdot \bar{a}^{-n+1}.$$

Collins conjectures, supported by extensive empirical analysis, that another bound might be

$$sep(p) > n^{\frac{-n}{4}} \cdot \bar{a}^{\frac{-n}{2}}.$$

All these bounds are quite unsatisfactory: They are in the order of $n^{-n} \cdot \bar{a}^{-n}$. It may be an alternative to use some other artificial bound, for example a size of one, and determine the number of roots with Sturm sequences for smaller intervals.

3.4.3 Partitioning Strategies

The last part of the bisection algorithm is the partitioning of an interval if it contains at least two roots. There is a wide range of possibilities to do this, so we discuss a few heuristics implemented in our application. We define a heuristic by its *pivot points*. Given pivots $P = \{p_1, ..., p_k\}$, the partitioning of an interval (a, b) is $\{(a, p_1), [p_1], (p_1, p_2), [p_2], ..., (p_k, b)\}$ as illustrated in Figure 3.3.

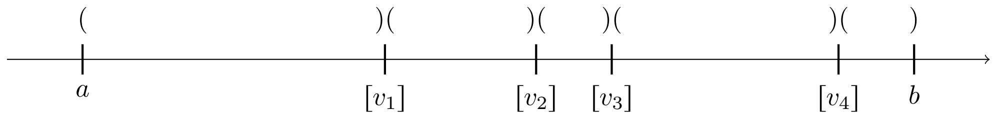

Figure 3.3: Partitioning by pivots $P = \{v_1, v_2, v_3, v_4\}$

The most obvious way, oftentimes implied by the name "bisection", selects a single pivot point being the *midpoint* of the interval, that is $P = \{\frac{a+b}{2}\}$. However, one might argue that we could use information about the polynomial to use better values to split. Also, we might want to stick to integer bounds as long as possible, as this reduces the size of the internal representation and speeds up later computations.

Attempting to stay within the integers yields the following procedure we call the *rounded midpoint*:

$$P = \begin{cases} \{\lceil\frac{a+b}{2}\rceil\} & \text{if } \lceil\frac{a+b}{2}\rceil \in (a, b) \\ \{\lfloor\frac{a+b}{2}\rfloor\} & \text{if } \lfloor\frac{a+b}{2}\rfloor \in (a, b) \\ \{\frac{a+b}{2}\} & \text{otherwise.} \end{cases}$$

It turns out that many problems actually induce polynomials having *nice* roots – meaning integers or fractions with small denominators – such that this actually hits an exact root significantly more often than the previous heuristic.

To exploit knowledge about our polynomial, we also try to use some of the previously presented algorithms for root finding. Using the Newton iteration, we perform one Newton step from the midpoint:

$$P = \left\{ \frac{a+b}{2} - \frac{p(\frac{a+b}{2})}{p'(\frac{a+b}{2})} \right\}.$$

Note that the Newton step may yield a value outside of (a, b). In this case, we choose the midpoint instead.

For the first partitioning, when we just started with the upper bound for all roots, it also makes sense to use a method that approximates all real roots, for example the companion matrix method or the Aberth method. Both yield approximations for all roots and we are able to create a set of intervals that might immediately isolate most roots. Given approximations $\nu_1^*, ..., \nu_k^*$ for the roots, we create

$$P = \{2 \cdot \nu_1^* - \nu_2^*, \frac{\nu_1^* + \nu_2^*}{2}, \frac{\nu_2^* + \nu_3^*}{2}, ..., \frac{\nu_{k-1}^* + \nu_k^*}{2}, 2 \cdot \nu_k^* - \nu_{k-1}^*\}.$$

As an initial approximation ν_i^* might as well already be an exact root, we check whether $p(\nu_i^*) = 0$ and add ν_i^* to P in this case.

These pivot points result in intervals around the approximation being as small as the approximations are close to each other. There are two additional intervals that range from the outer roots to the bounds of the initial interval. If the real roots are very close compared to the size of the initial interval, these two intervals fill most of the space but are likely to contain no real roots.

Example 3.2. *We assume a polynomial with three real roots* $\nu_1 = 5$, $\nu_2 = 7.2$ *and* $\nu_3 = 10.2$. *We use the companion matrix method (or the Aberth method) to get approximations for these roots and get* $\nu_1^* = 5$, $\nu_2^* = 8$ *and* $\nu_3^* = 9$. *Our partitioning is like shown in Figure 3.4.*

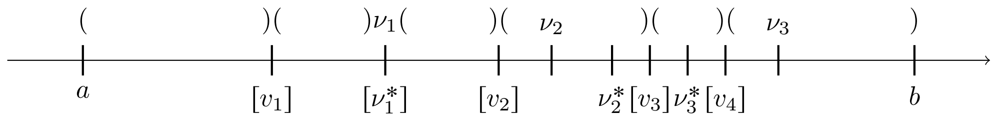

Figure 3.4: Partitioning with approximations for the real roots

The pivot points generated from $\nu_1^* = 5$, $\nu_2^* = 8$ *and* $\nu_3^* = 9$ *are* $P = \{v_1 = 3.5, v_2 = 6.5, v_3 = 8.5, v_4 = 9.5\}$.

This partitioning directly isolates all roots, it even determines the first root exactly. This is not always the case, but is more likely if the polynomial is of smaller degree and the roots are not too close to each other.

3.4.4 Incrementality

We call a method *incremental* or *lazy*, if it tries to avoid unnecessary work and only performs certain tasks when there is no other alternative left. This usually means, that the result can be split into multiple partial results which are calculated one after the other, hoping that a few partial results are sufficient to solve the overall problem. If not all partial results are needed, we can save computation time.

In many applications, this is impossible because there is no such thing as a partial result or no partial result can be computed without computing the complete result. In other applications, this approach involves a certain overhead, oftentimes because computing all partial results is more expensive than computing the complete result once.

However, bisection can be performed incrementally in a very natural way. Starting from Algorithm 1, we only need to return an isolated root as an intermediate result instead of adding it to the result set R.

Such an approach can be particularly useful in the context of satisfiability checking, because the whole problem is solved as soon as a single satisfying solution is found.

4 Adaptable Intervals

In our applications we need operations performed on intervals, either because our algorithms work on intervals or because we represent real numbers by intervals. Therefore, the overall performance depends heavily on the speed of operations on intervals.

Intervals are often used with exact bounds, that are bounds represented by exact numbers. While this ensures easy handling of the intervals, it also implies a major performance hit. Therefore, we examine how inexact numbers can serve as interval bounds. The main idea is to detect insufficient precision during the inexact operations and then dynamically fall back to exact computations.

4.1 Operations on Intervals

Among the fundamental interval operations, notably access to the bounds and checks for inclusion, for the CAD method only a few operations are used frequently and therefore have to be particularly efficient. For the lift operation, multivariate polynomials are evaluated at a sample point, hence fast *additions* and *multiplications* are needed.

As a real root in interval representation must never contain more than one real root or overlap with another root in interval representation, they have to be *refined* regularly.

We relax the common definition of interval addition and multiplication such that the resulting intervals can be larger than the exact result. We call such operations *over-approximating* in the sense that they approximate the exact operation. Whenever an interval becomes too large, we can use the refinement routine to reduce it. This gives us some freedom for the actual implementation of these operations.

Definition 4.1. *Let $I, J \subseteq \mathbb{R}$ be intervals. We define*

$$\begin{aligned} I + J &\supseteq \{v \in \mathbb{R} \mid \exists i \in I, j \in J : i + j = v\} \\ I \cdot J &\supseteq \{v \in \mathbb{R} \mid \exists i \in I, j \in J : i \cdot j = v\}. \end{aligned}$$

If we represent the interval bounds with exact numbers, we can implement these operations exactly, but as these operations are the basic operations within the CAD method, we have to consider their performance. Addition of intervals is easily reduced to addition of the bounds $(a, b) + (c, d) = (a + c, b + d)$, which conforms to the above definition.

Interval multiplication is a bit more complex:

$$(a, b) \cdot (c, d) = (\min X, \max X) \text{ with } X = \{a \cdot c, a \cdot d, b \cdot c, b \cdot d\}.$$

Thus, we can reduce addition and multiplication of intervals to addition and multiplication of exact numbers. If we represent exact numbers as fractions, addition and multiplication are not atomic operations but rather correspond to the following operations, where u and v are exact numbers, u_1, u_2, v_1 and v_2 are integers of arbitrary size and *cancel* is the standard cancellation method to reduce fractions.

$$u + v = \frac{u_1}{u_2} + \frac{v_1}{v_2} = \mathit{cancel}\left(\frac{u_1 \cdot v_2 + u_2 \cdot v_1}{u_2 \cdot v_2}\right)$$
$$u \cdot v = \frac{u_1}{u_2} \cdot \frac{v_1}{v_2} = \mathit{cancel}\left(\frac{u_1 \cdot v_1}{u_2 \cdot v_2}\right)$$

Addition of two integers is linear in the size of the numbers. As for multiplication, there exist several algorithms. Well known are the Karatsuba algorithm with complexity $\mathcal{O}(n^{\log_2(3)})$ and the Schönhage–Strassen algorithm [SS71] with complexity $\mathcal{O}(n \cdot \log(n) \cdot \log(\log(n)))$, while the theoretical lower bound for large integer multiplication is believed to be $\Omega(n \cdot \log(n))$ [Für09].

Anyhow, addition and multiplication on exact numbers require multiple integer additions or multiplications of (super-)linear complexity and we still need to cancel the fraction efficiently, even if we do not have to do this every time.

These considerations reveal a significant drawback of exact numbers: Already basic operations like addition and multiplication have a super-linear complexity in the size of the internal representation. And even if we are able to represent the integers with native integer types and hence can perform integer operations in constant time, we cannot compete with native processor operations.

4.2 Using Inexact Computations

As we have seen, interval operations making use of exact numbers exhibit an inherent performance hit. Therefore, it seems natural to investigate, if we can utilize of native floating point data types being computationally less expensive.

Today's processors, with the exception of microprocessors that are outside of our target group, usually support floating point numbers complying to the IEEE 754 standard [IEE08]. We use the 64 bit variant of this standard which is usually called double.

Consisting of eleven bits for the exponent, 52 bits for the mantissa and an additional bit for the sign, a double can represent numbers with absolute values ranging from $2.2251 \cdot 10^{-308}$ up to $1.798 \cdot 10^{308}$ with about 15 significant decimal places. The standard additionally provides reserved bit patterns to represent ∞, $-\infty$ and different kinds of arithmetical errors and defines several rounding modes.

These values show that most calculations seeming realistic in our setting are possible with double numbers so that we could simply use double numbers as interval bounds. However, we must ensure we maintain correctness with respect to our defined addition and multiplication operations.

The default rounding mode is to round to the nearest representable number, but modes to round up or round down are available. Using these modes properly, we are able to build an interval where the upper bound is a always rounded up and lower bound is always rounded down. However, we see in Section 5.3 that this is not trivial in practice.

4.3 Automatic Fallback

Nonetheless, in some rare cases, double might not be sufficient. We may need numbers exceeding the range of numbers that a double can represent. We may also face numbers that are so close to each other that the double type cannot distinguish them. As for numbers in the range of 10^{20}, the distance between to numbers that are representable by as double is already about 10^5.

If we are outside the bounds of double, the result of most operations is usually $\pm\infty$ or *not a number* (NaN) in some cases. Once this occurs, we must change to exact numbers, as further calculations cannot yield any meaningful result. One might want to investigate the reason for the $\pm\infty$ or NaN and might try to tweak the intervals to prevent this error. However, a problem instance that produces such a problem probably produces more of them, hence it seems reasonable to surrender at this point.

What we do instead is to fall back to exact computations. Whatever operation we wanted to perform, we convert the input intervals such that they use exact bounds and then perform the same operation again on these exact intervals. The conversion can always be done, as the set of representable exact numbers is a superset of the representable double numbers.

It may be possible to try converting exact intervals back to double intervals when the bounds have returned to representable values, but we do not investigate this any further here.

4.4 Using Interval Coefficients

There is another use case for double intervals in our application. Whenever we perform calculations on a polynomial, especially when we evaluate it at some point, we need to compute a significant number of operations: As we discussed in Section 2.3, we need n multiplications and n additions for an evaluation.

Although we only talked about exact coefficients, there is no reason why double intervals cannot be used to represent polynomial coefficients. Note that there is no inherent need for intervals, as the coefficients are given as finite representations and as such are representable by exact numbers.

We decided to allow for both. Operations on any univariate polynomial can either use exact coefficients or double interval coefficients. As only a few operations change the coefficients of a polynomial, we ensure that a polynomial always has exact and double interval coefficients and hence can always perform exact and inexact operations.

5 Implementation Details

Most of the presented ideas and algorithms have been implemented within GiNaCRA [LA11], an extension to GiNaC used within SMT-RAT [CLJA12]. As SMT-RAT strives to provide fast theory solvers for nonlinear real arithmetic, all this is implemented in C++ and special effort has been spent to produce a careful implementation. The functionalities described here have been implemented within the new rootfinder namespace in GiNaCRA.

5.1 Root Finding

The main goals for our C++ implementation are simplicity, modularity and performance. We decided to minimize the interface as much as possible to avoid any ambiguity on how to use it.

We introduced a new class called IncrementalRootFinder that is designed to isolate the roots of a given polynomial in an incremental fashion. It maintains a queue of intervals and a result set of isolated roots.

The constructor of this class accepts a polynomial and configuration options. It starts with transforming the polynomial into a reduced polynomial as described in Section 2.3. The polynomial is then checked for a root at zero, as this is a frequent root and needs only constant running time. If the polynomial has a small degree such that it can be solved using a closed formula, we do this and add the roots to the result set. Otherwise, a bound on the real roots is calculated and the resulting interval is added to the queue.

Once an instance of this class is created, calling next() checks the result set and returns a new root if there is any. As long as there is no new root found, it pops an interval from the queue and applies bisection to it, depending on additional information stored alongside the interval within the queue. We discuss the bisection functions in the next section. If the queue is empty, next() returns null and thereby indicates that the isolation process has finished.

As for the interval queue, we identified changing the order of the intervals in the queue as a possible optimization for the root isolation process. The order in which the intervals are processed is irrelevant for the correctness, hence we want to work on *good* intervals first, leaving *bad* ones for later, hoping find real roots as fast as possible. We decided to leave the decision how *good* an interval is to a heuristic which favors inexact intervals over exact intervals – operations on the former ones should be quicker – and secondly prefers smaller intervals hoping that they are more likely to produce an isolated root soon.

If we want to access the roots of a polynomial again, we can use the rootCache() method, which returns the list of all roots that have already been found. As soon as next() returns null, rootCache() returns a complete root isolation.

5.1.1 Bisection Functions

Our bisection routine works as described in Algorithm 1 always using Descartes' rule of signs to count real roots. The initial intervals are constructed from the Cauchy bound.

We implemented the bisection heuristics described in Section 3.4.3 as functors that can be used by the IncrementalRootFinder class. They all take the interval to bisect and an IncrementalRootFinder instance as input parameters and add intervals to the queue or isolated roots to the result sets through additional helper methods.

As for the companion matrix method, we make use of the C++ numeric library IT++ [ITP]. IT++ is a library containing matrix and vector operations mainly intended for simulations and communication systems. It implements the method eig() that takes a matrix and returns a vector containing the eigenvalues of the given matrix.

The eig() method uses the LAPACK [ABB$^+$99] routine DGEEV that is provided by any library that implements the LAPACK interface, for example ATLAS [WPD01] or Intel MKL [Int]. Therefore, we can safely assume that these eigenvalues are as exact as possible and their computation optimized.

5.2 Representation of Numbers

A tempting way to represent numbers that may be either exact or inexact is to create a class that either holds a double or an exact number. This class might overload necessary operators, like addition or multiplication, adapted to the given representation.

This way, only the class itself and its operators need to be aware of the different number representations that are contained in the number class. Furthermore, the fallback to an exact number whenever precision of the inexact number is insufficient could be performed within the methods implementing the operators. The actual type of a number would be transparent for all methods that use these numbers, in particular to the interval class.

However, there are some problems that must solved for such a number representation.

There are cases, for example if a real root has been found, where we need a number representation that guarantees exactness. Hence, we would force an adaptable number to store the number exactly and disallow representing it by a double. Since most of the CAD method operates either on exact numbers or on intervals, there is no advantage of such a class over adaptable intervals, that support exact and inexact bounds.

When we perform an operation on two inexact number representations, the result is rounded to a representable value that is close to the exact result. If such an operation is part of an interval root operation that needs to guarantee that it is overapproximating, we need to round either up or down, depending on the exact calculation that is done.

Hence, we have the provide the possibility to pass additional information to the basic number operations.

Finally, the performance of calculations on such a class representing an inexact number would be significantly worse compared to calculations on native inexact numbers. While an operation on double numbers is a native CPU instruction, the presented abstraction class exhibits a minimum of overhead that is significant if the operation itself lasts only a few processor cycles.

Because of these reasons, we decided not to implement such an abstract number type and use exact numbers from GiNaC or double numbers separately, although this means that many functions must be implemented for exact numbers and double numbers individually.

5.3 Intervals

We already touched on how to build adaptable intervals that stick to our definition of overapproximating operations. The basic idea is to take advantage of the different rounding modes available for double operations, which we discuss now in more detail.

5.3.1 Class Interface

As we want an adaptable interface to be an interface that can have either exact or inexact bounds, the class has to provide methods for both cases. That means that routines operating on these intervals may need to call different methods to exploit the advantages of inexact operations.

To ease the migration to these new intervals, each method checks the type of the interval internally. If the type of the interval does not match the type of the arguments or the desired return type, the interval performs the type conversions internally. Although these permanent conversion are a serious performance hit, it gives us the possibility to perform a continuous migration to the new intervals.

Most methods can be implemented in a way such that they can infer whether to calculate exactly or inexactly from their parameters. However, there are functions that only return a number, for example a bound on the real roots as described in Section 3.4.1, we want to provide them both for exact and inexact calculations.

These methods conceptually differ only in their result type and therefore must be distinguished by their method names, because C++ does not allow methods with identical names and identical parameter types. Note that this is not an arbitrary restriction from C++ but a consequence of the way modern object-oriented languages like C++ or Java implement classes.

Hence we end up with an interface where many methods are defined twice, once for exact and once for inexact computation. This is the penalty for handling exact and inexact computations within the interval class and not doing this within a number class as described in Section 5.2. But these methods also represent the internal structure of the interval class, as most functionality is implemented separately for exact and inexact bounds anyway.

5.3.2 double Operations

We now analyze how to use double numbers in our context. Though their operations are inexact, we must make sure that the interval operations are overapproximating. As we want the intervals to remain small, we achieve this through appropriate rounding.

While some *floating-point units*, called FPUs, support different rounding modes for single operations, most of them only implement operations to switch the mode that is valid for all operations to come. This usually results in flushing the FPU pipeline for every change of the rounding mode, thereby severely decreasing the overall performance of the double operations. Hence, we need a strategy to minimize the overall number of rounding mode switches.

As for the addition of (a, b) and (c, d), we would naively compute $\lfloor a+c \rfloor$ and $\lceil b+d \rceil$. We can replace the latter one by $-\lfloor(-b) + (-d)\rfloor$ and thereby, as the negation of a number involves no rounding, use only a single rounding mode instead of two. However, we should restore the original rounding mode as soon as we return the control flow to some other library, that might also perform double operations.

Some FPUs actually perform their calculations on registers larger than a double as an attempt to increase precision. The actual result is then the rounded value of the larger internal result value. Depending on the exact calculation, the above equality $\lceil b + d\rceil = -\lfloor -b - d\rfloor$ might not hold anymore. Therefore, we either have to drop this idea and continue to switch rounding modes more frequently or we have to force the FPU to calculate only on double precision, as that would maintain the above equality.

As such optimizations require a lot of knowledge about different FPUs and their specific mechanisms, we decided to use an existing library already handling all this. The boost C++ library contains a component called boost::interval that we employ due to its availability, documentation and reputation.

boost::interval implements an interval class with a templated bound type. However it is clearly intended to be used with double. A large number of operations is defined on this interval class, not only addition, subtraction, multiplication and division, but also for example the square root, sine, cosine and logarithm. There are policies to specify how rounding should be done. The default policy implements an overapproximating interval and therefore complies to our definition of addition and multiplication.

5.3.3 Error Handling

There are two possibilities how boost::interval signals an error or exceptional behavior, that are comparisons where the result is unclear or operations which result in an infinite value, possibly due to insufficient precision, or the double error value NaN.

The first possibility we are using in our implementation is throwing an exception. We attempt to catch these exceptions wherever they may be thrown and then change to exact arithmetic as described.

The second possibility returns a so-called tribool instead of a Boolean result, for example for comparison functions. A tribool can hold the values true, false and indeterminate. Whenever an operation cannot be performed correctly, indeterminate is returned.

A tribool result is checked using the C++ idiom described in Algorithm 2. Using tribool return values can result in cleaner code and a faster program, as exceptions always come with a significant overhead when thrown.

```
tribool x = method();
if (x) {
  /* x is true */
} else if (!x) {
  /* x is false */
} else {
  /* x is indeterminate */
}
```

Algorithm 2: Correct handling of tribool values.

However, there are cases where exceptions are more convenient. Imagine having a larger operation which is implemented twice, once for exact and once for inexact arithmetic, to include special optimizations. If the inexact operations fail at some point, the whole operation has to be canceled and restarted with exact arithmetic. When using exceptions, the operation method does not have to care about any exceptional cases: if they happen, the exception is caught by the calling function which can perform the conversion and call the operator again with the exact values.

Therefore, we decided to use exceptions. Since the mode can be switched also for single functions or modules, we plan to port certain parts of the code to tribool results, especially where exceptions are thrown very frequently.

5.3.4 Changing the Interval Type

There are two possible type conversions that may be used: From inexact to exact and from exact to inexact. The former one is used whenever we run into an error as described above, while the latter one is needed to initially convert an exact interval to an inexact interval. This is needed when our algorithm receives parameters from the rest of program which does not use inexact numbers yet.

Converting double values to exact values is straightforward. A double can always be exactly represented by a fraction and there are methods that return this fraction. As for GiNaC and CLN, there is GiNaC::rationalize().

The other way is a bit more involved, as we may have to drop some precision. One solution is to obtain some rounded double value and increase or decrease this value, until it is larger or smaller than the exact value.

GiNaC offers the method to_double() for exact values. We then increase, if it is the upper bound of the interval, or decrease, if it is the lower bound, this result using std::nextafter(), until the new inexact interval contains the old exact interval.

The std::nextafter() routine is given a floating point value v and a direction d which is also a floating point value. It returns a value r such that $|r-d| < |v-d|$, that means r is closer to d than v, and there is no representable value between v and r. This method allows us to efficiently iterate over the representable values in a safe way.

5.3.5 Integration in the CAD Method

The implementation of the CAD method in GiNaCRA uses a special class to represent a real root that may either hold an exact root or an interval root. Both provide a common interface such that most of the CAD method can operate on this interface and does not need to care about the type of a particular root.

The basic idea is to replace the interval used for the interval root by our new adaptable intervals and make the CAD method aware of this change. As an interval with adaptable bounds has to be treated differently than an interval having only exact bounds, this change implies a careful refactoring of all routines operating directly on an interval.

The interval roots were changed to use the new intervals, however the porting is not finished yet. While all operations on interval roots work with adaptable intervals, some of them are not aware of the possibility that the underlying bounds are inexact but used by methods for exact bounds. In such a case, these methods convert the bounds on every call and hence produce a significant overhead.

Some methods even rely on an exact bound type such that the interval must be converted before such a method may be called. This of course interferes with our goal to perform as many inexact operations as possible. Hence, to take full advantage of the possibility to calculate inexactly, most of the code used for the CAD method must be adapted.

5.3.6 Polynomials

We have seen that a frequent task is to evaluate a polynomial at a given point. All the effort to make inexact computations is futile though, if the polynomials can only operate exactly. Hence we need a polynomial class that can perform exact and inexact operations, similar to the adaptable polynomials.

Although in general the CAD method operates on multivariate polynomials, we started with a class for univariate polynomials. This class stores coefficients exactly and inexactly and hence provides exact and inexact operations.

The CAD method in SMT-RAT still uses a polynomial representation based on symbolic expressions from GiNaC, hence even a specialized class for multivariate polynomials working on exact coefficients only should be an improvement. Such polynomials are under development, but not used in the CAD method yet.

6 Experimental Results

So far, we discussed the theoretical background and some general ideas of the implementation. Now, we show some experimental results of the new implementation for different scenarios.

6.1 Isolating Real Roots

For a start, we compare the new implementation using the techniques described in this thesis with the implementation previously used in GiNaCRA. We compared the results with Wolfram Mathematica 8 to verify the correctness of our algorithm, but did not include Mathematica in our benchmarks as it was not available on our target platform. The tests were run on an AMD Opteron 6172 @ 2.1GHz using Debian Linux.

The previous algorithm consists of a recursive bisection algorithm and uses Sturm sequences to count real roots. As for the splitting, a heuristic similar to our rounded midpoint heuristic and multiple variants of the Newton heuristic are available.

Our scenario consists of $n = 1000$ random polynomials of a common *desired degree* that we generate as follows: For each polynomial, we choose a *minimal degree* between zero and the desired degree uniformly at random. This minimal degree shall be the minimal amount of real roots of this polynomial. We select real roots from $[-1000, 1000]$ with a granularity of $\frac{1}{100}$ uniformly at random and use them to build a polynomial of the selected minimal degree with the chosen real roots.

As for the difference between the desired degree and the minimal degree, we create a polynomial with random coefficients from a normal distribution with mean $\mu = 0$ and a standard deviation of $\sigma = 25$. However, we make sure that no coefficient is zero. We multiply these two polynomials and get a polynomial of the desired degree.

Note that these polynomials have almost no roots at zero while polynomials from real world examples often have such roots. However, all our bisection algorithms check for zero at the very beginning, as a root at zero is particularly easy to detect and to eliminate. Therefore such a root only increases the degree of a polynomial without adding any complexity for the isolation algorithm, hence we chose not to produce roots at zero intentionally.

Of course, this is only an artificial choice, but we think that this choice resembles the polynomials one encounters in real life examples. We create such a list of polynomials for every desired degree $\deg \in \{3, ..., 15\}$.

To get meaningful comparisons, we chose the new algorithm with different heuristics as well as the algorithm that was previously used in GiNaCRA to run on the same set of polynomials.

6.1.1 Speed of Isolation

The running times for all polynomials are estimated from the clock cycles used and measured with std::clock(). The results of this test are shown in Figure 6.1 on a logarithmic scale.

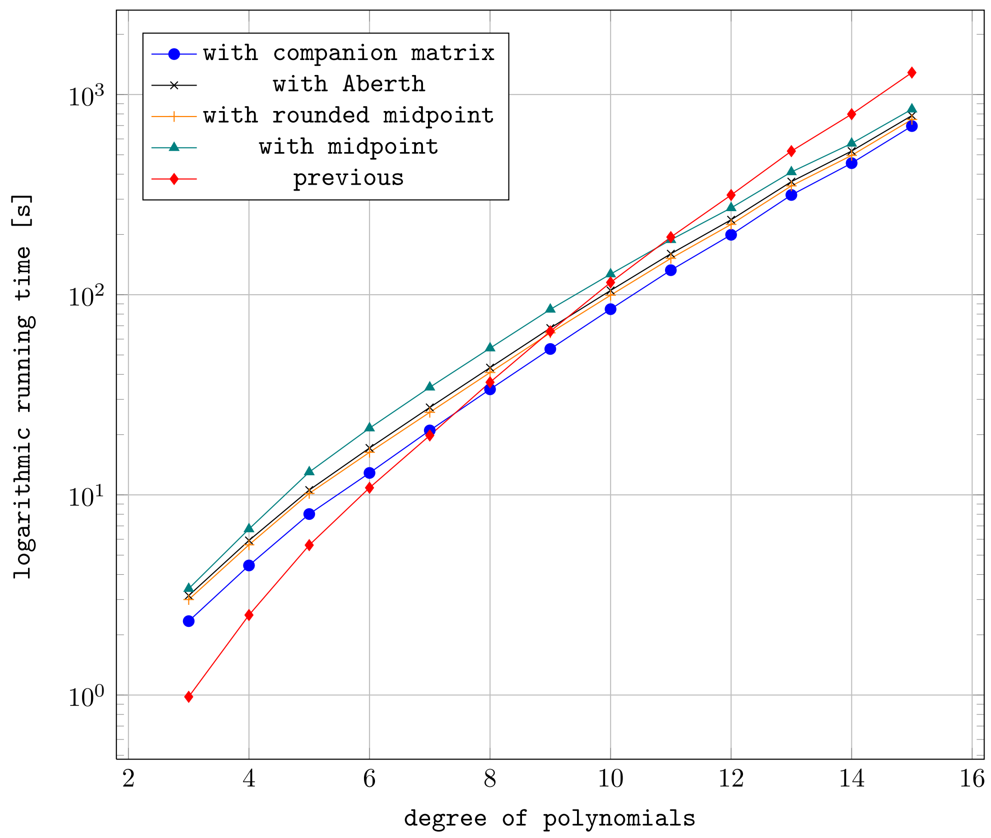

Figure 6.1: Comparison of running times for root isolation

While all algorithms show the expected exponential behavior, we can see a significant difference between the previous algorithm and our new algorithm. The previous algorithm is faster for polynomials of small degree but exhibits a faster growth. Starting with degree 8, the new algorithm using the companion matrix heuristic is the fastest out of all tested heuristics.

Note that due to the logarithmic scale, small differences in this graph already represent large differences in the actual running time. As for degree 12, the previous algorithm needs over 50% more time than the new algorithm with the companion matrix heuristic.

6.1.2 Number of Exact Roots

We have already discussed in Section 3.2 that an isolation is not only correct but has a certain quality for our application. One goal is to find as many exact roots as possible. As we chose the roots of the polynomials at random, most heuristics are unable to find such exact roots regularly. Figure 6.2 shows the average number of exact roots found for a polynomial.

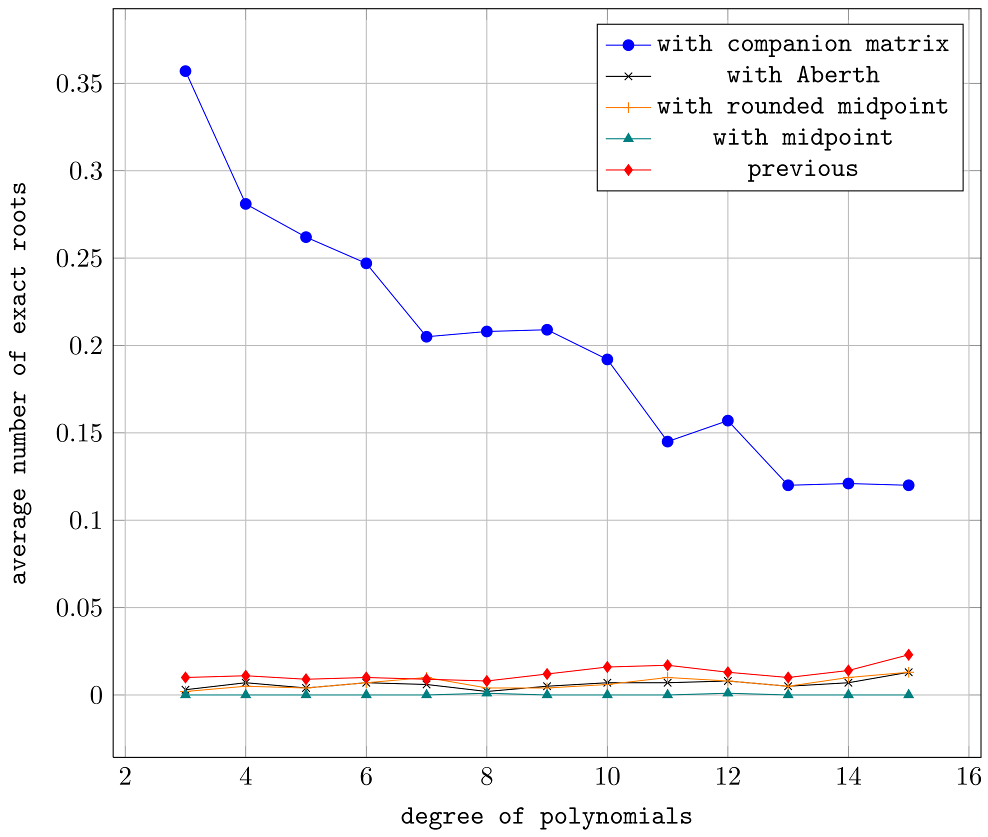

Figure 6.2: Comparison of probability to find exact roots

We see that the companion matrix heuristic is the only heuristic that is able to find exact roots systematically. Starting at 0.35 for degree three, the ratio of roots that are found exactly decreases to 0.10 for degree 15.

All other algorithms are significantly worse and constant for all degrees. The previous algorithm achieves at most 0.02, the midpoint heuristic finds almost no roots and the other heuristics are somewhere in between. Altogether, using the companion matrix heuristic might be an advantage within the CAD method, although the absolute number of exact roots still is very small.

6.1.3 Size of Isolations

Another goal for good isolations is the size of the isolating intervals. If the intervals of an isolation are smaller, the probability that the intervals need to be refined as they overlap after some operations during the CAD method is also smaller.

While one may argue about different metrics, we decided to use a logarithmic scale for Figure 6.3. For each polynomial, we calculated the average logarithmic size of all interval roots. Note that this produces negative numbers if the intervals are smaller than one.

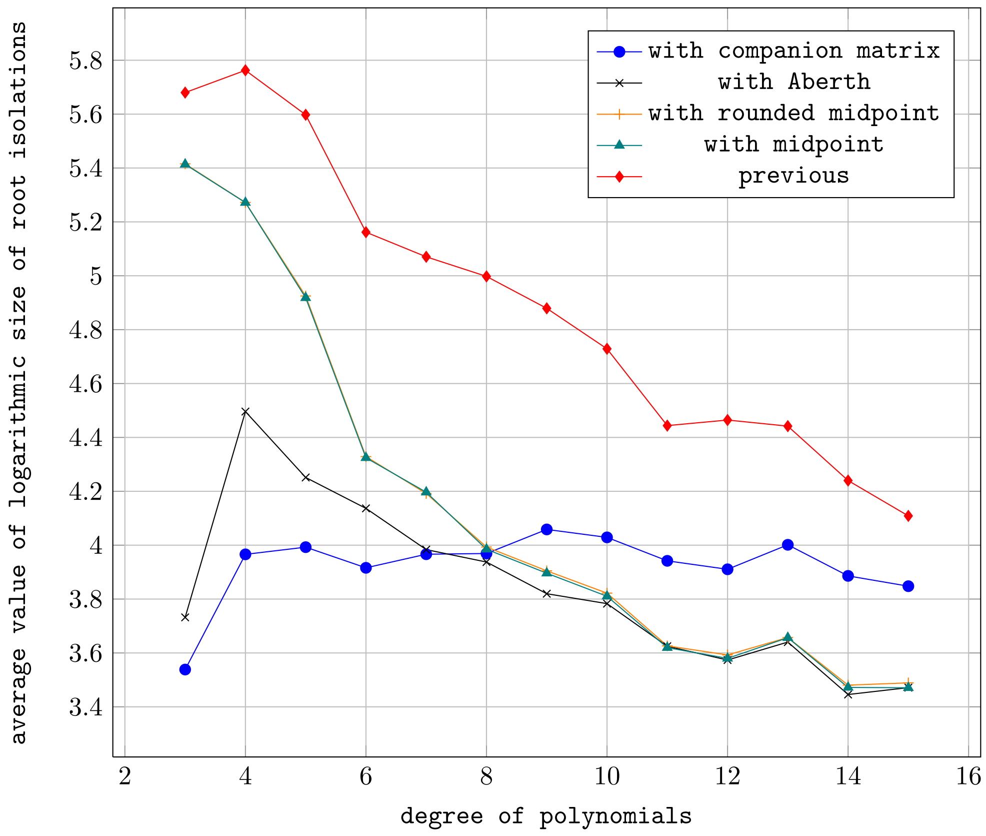

Figure 6.3: Comparison of size of real root isolations

While the data seems fairly arbitrary, we can see that our new algorithm produces significantly smaller isolations than the previous algorithm. As for the companion matrix heuristic, the average size seems to be almost constant, the other heuristics and the previous algorithms produce smaller isolations for polynomials of higher degree.

6.1.4 Separation of Intervals

While the size of the intervals gives an idea on how much the intervals grow during an operation, the distance between two intervals determines how much an interval may grow before it overlaps with another interval and has to be refined.

We call this distance between two intervals their *separation*. For each polynomial, we calculated the average separation between two consecutive roots, both exact and interval roots. The average separations is shown in Figure 6.4.

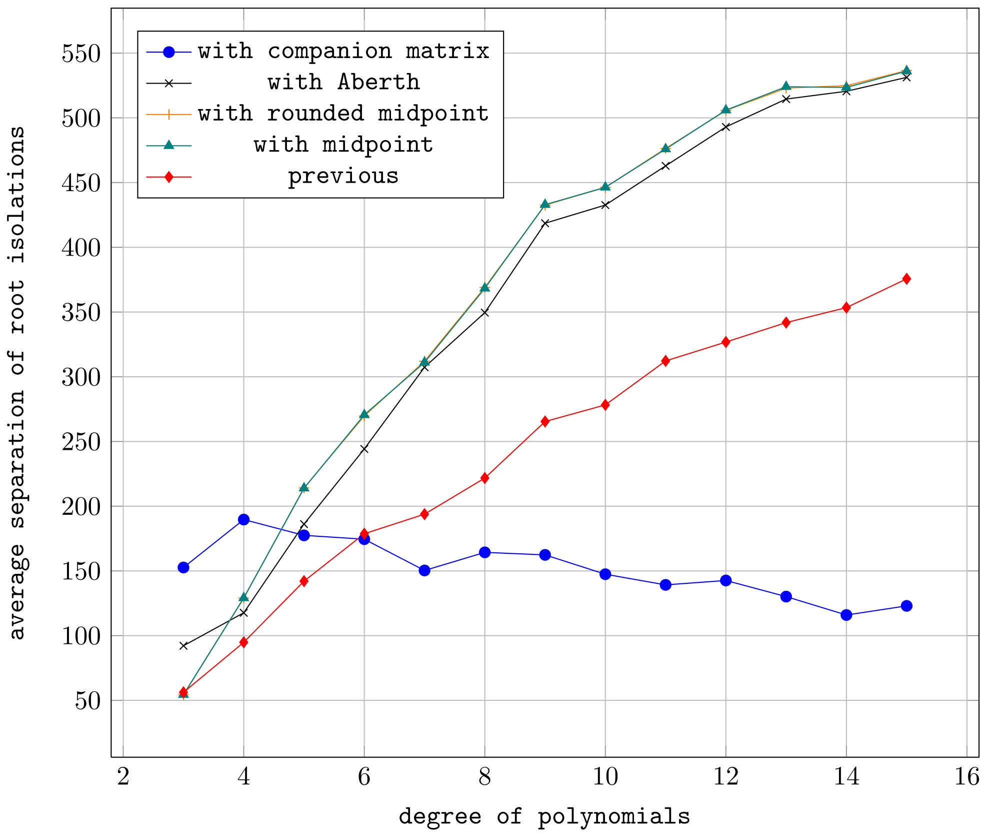

Figure 6.4: Comparison of separation of real roots

Since a small separation bears the risk of refinements in the CAD method, the goal is to maximize this separation. However, due to the way the roots are distributed, the separation is bounded by the two outermost roots and related considerably to the average size of the isolations.

Therefore, the separation exhibits a behavior alike to the isolation sizes. While the companion matrix heuristic yields results with an almost constant separation, the separation of all other heuristics and the previous algorithm grows with the degree of the polynomials.

6.2 The CAD Method

Although the results we have shown give an impression of the performance of the new algorithm, the ultimate question is how the application performs. Hence we checked the performance of the CAD method using the new algorithm with the companion matrix heuristic and using the previous algorithm. We decided for the companion matrix method because it performs similarly to the other heuristics but manages to find an outstanding number of roots exactly.

We built two solvers using the above root solving methods from the current state of the SMT-RAT project. We call them *new solver* and *old solver*. Both were configured to call only the CAD method. To test their performance, we used two standard benchmark sets for SMT solvers: *meti-tarski* and *keymaera* that are also used in [JdM12].

Since the integration of the double intervals into parts of the CAD method is not yet stable, we decided to remove all instances from our statistics where a solver crashes. These are however only 33 instances of 8276 instances in total for *meti-tarski* while there occurred no errors in *keymaera*.

To restrict the time needed for these benchmarks, we defined a time limit of 60 seconds. If a solver hits this time limit, it is stopped. The results of these tests are shown in Figure 6.5. The number of successfully solved instances is given in column *sat / unsat*, the time in seconds needed to solve them in *time*. *Timeouts* gives the number of instances where a timeout occurred.

	old solver			new solver		
benchmark set	sat / unsat	time	timeouts	sat / unsat	time	timeouts
meti-tarski	6806	5777	1434	5309	4856	2931
keymaera	303	324	118	290	346	131

Figure 6.5: Overview over running times

We can see that the new solver is slower and has significantly more timeouts. The average time for the successfully solved instances is smaller for the new solver on the *meti-tarski* set, however instances which take the old solver a long time will produce a timeout on the new solver. Hence, the solved instances of the new solver comprise easier instances on average than these of the old solver.

To get a visual impression of how the solvers perform on individual instances, we included Figure 6.6 and Figure 6.7. They show a comparison of the running times of the two solvers on both benchmark sets. Each point represents an instance of the benchmark set, its x coordinate represents the running time of the old solver and the y coordinate the running time of the new solver. Hence, the new solver is faster on all instances that reside below the diagonal line.

In comparison to the above table, we remove some more instances from these plots. Instances where both solvers produce a timeout would only result in a single point in the top right corner, therefore we do not include them here. Furthermore, instances where both solvers are fast create an opaque cloud close to the origin. Hence, we also removed all instances where the sum of the running times of both solvers do not exceed one second.

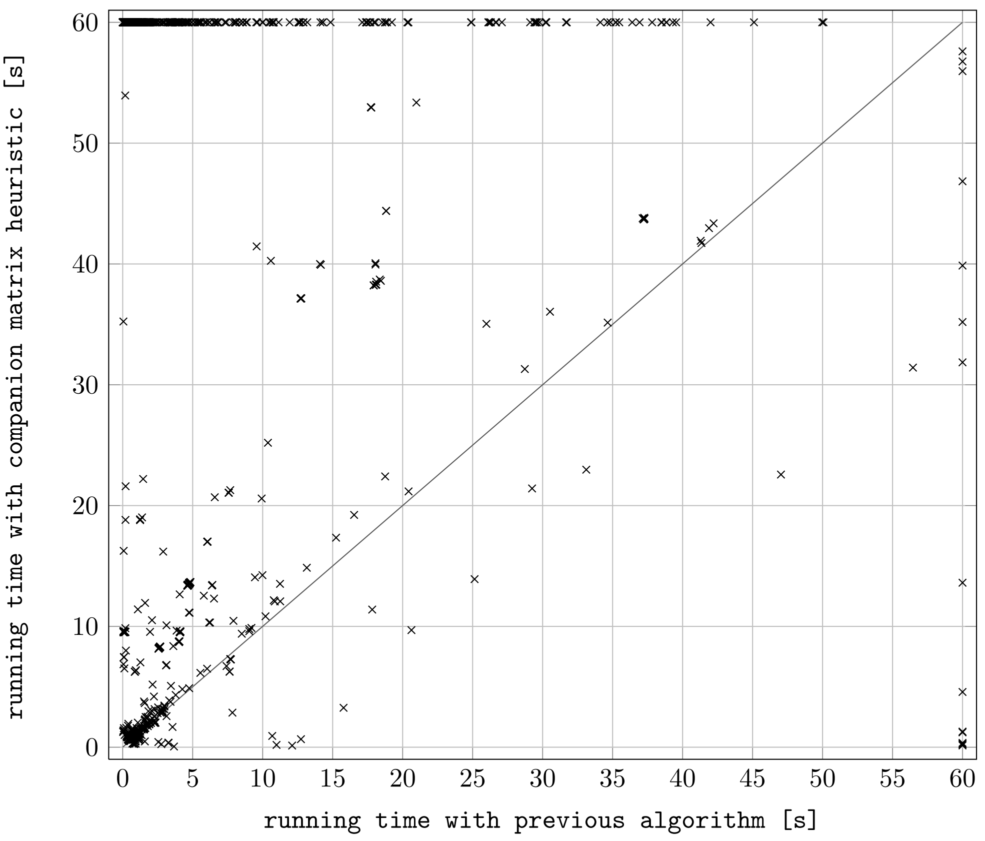

Figure 6.6: Comparison of running times on *meti-tarski* benchmark

Once again, we can see in both plots that the new solver is significantly slower in general. We now analyze this and explain a few reasons for this.

We can observe a large amount of instances in the *meti-tarski* set that are solved very quickly with the old solver but produce a timeout on the new solver. This pattern suggests the existence of a particular routine that is only called for specific instances and is poorly implemented in the new solver. This routine might either be much slower or lead to nontermination in specific situations.

We have seen that the root isolation algorithm itself is faster than its predecessor, hence the poor performance of the new solver must have other reasons. Either our sample polynomials did not match the polynomials that occur in practice or the integration of the new algorithm in the CAD method is bad. As we have already discussed a few problems in Section 5.3.5 we suspect that the integration must be optimized.

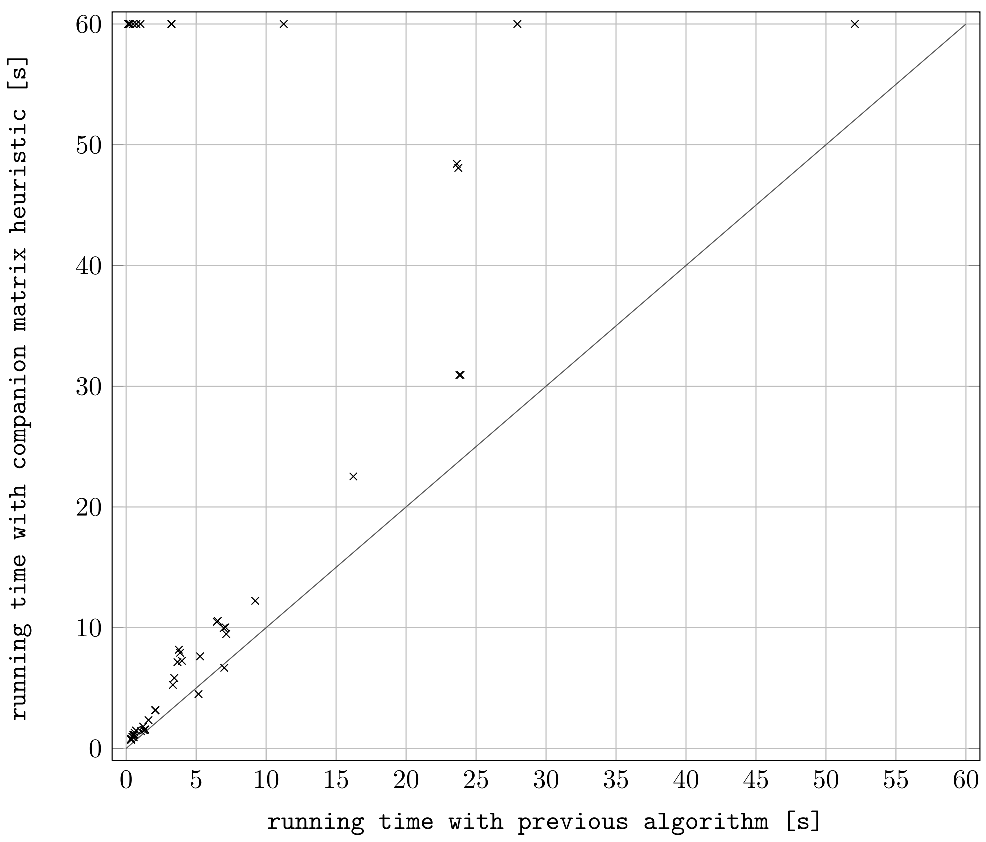

Figure 6.7: Comparison of running times on *keymaera* benchmark

We still use exceptions to detect errors during interval operations. Exceptions are relatively time-consuming in C++, hence they may also contribute to the bad performance of the new solver.

There are several issues that are conceptually simple but time-consuming and hence are not finished yet. We now give a short overview of these issues. We have also some ideas for further optimizations that are presented in Section 7.2.

The CAD method was refactored to use the new adaptable intervals, but most operations still use exact operations, even if inexact operations are available. Hence it not only fails to take advantage of faster inexact operations, it also produces additional overhead as the interval has to convert the inexact bounds to exact numbers every time. Once a routine within the CAD method is aware of the existence of inexact intervals, it can avoid this overhead and profit from inexact calculations.

We already mentioned that the CAD method uses polynomials built upon symbolical expressions provided by GiNaC. This has certain advantages, as GiNaC provides many algebraic operations on these expressions used within the CAD. Representing a polynomial by its coefficients however allows for faster operations and less memory usage, hence we implemented such polynomials for the univariate case.

A different representation implies however, that the representation of a polynomial needs to be converted whenever we search for its real roots. Although this overhead is relatively small, it may be relevant if it is invoked often. A new library for multivariate polynomials is being developed and will be used in SMT-RAT in the near future. This should provide a general speed-up for the CAD method but especially also reduce the overhead for our new algorithm.

7 Conclusion

The goal of this thesis was to improve existing methods to find real roots of univariate polynomials and to provide the possibility to operate efficiently on these roots afterwards in the context of SMT solving.

7.1 Summary

We started with guidelines on what we consider good isolations and looked at different approaches to find real roots. After discussing a few approximative methods we focused on bisection and presented several possibilities for upper bounds on the roots to get initial intervals, heuristics to split intervals and methods to count the number of real roots within an interval.

We continued with the representation of roots using intervals. Based on the operations that are required to use this representation within the CAD method, we analyzed how we can use double intervals while maintaining correctness of our computations. Furthermore, we presented ideas how to deal with cases where these double numbers cannot perform the desired operations and how we can apply the same ideas for faster polynomial evaluations.

As the presented algorithms were implemented within GiNaCRA we gave some details about their implementation. We showed how we made use of different libraries and how we detect errors during operations on inexact intervals.

Finally, we presented some experimental results for these algorithms. While there is clearly room for improvement, this new approach is already significantly faster for polynomials of degrees of nine or more and provides better results with respect to the number of exact roots at the same time.

7.2 Future Work

This thesis contains some topics that are only dealt with in the univariate case yet and still have potential for improvement. Some ideas that arised during the work on the presented topics could not be addressed at all. The implementation also offers a number of places for optimization.

7.2.1 Usage of double Intervals

In Section 4.4, we discussed the possibility to have polynomials with coefficients that are represented by double intervals. This is implemented for univariate polynomials used during the root finding, but multivariate polynomials used within the CAD method lack this functionality.

double intervals are already used in the generic number representation used in the CAD method and are therefore also available for all sample points. However, the polynomials need to support operations on double intervals to exploit this.

There is a new polynomial library under development within the SMT-RAT project supporting all this, but work on this library is not finished yet. As soon as it is available for use, the CAD method will be ported to the new library and should be modified such that inexact computations can be performed throughout the CAD method.

The overall process still suffers from several type conversions, especially between different polynomial types. The new library will provide polynomials whose representation is almost identical to those used within the presented algorithms, hence there will probably be less overhead at the interface between the CAD method and the root finding.

7.2.2 Find More Exact Roots

We can avoid interval operations altogether, if we find real roots exactly. We already argued that we cannot do this in general, however we may want to spend more effort for heuristics to guess exact roots.

As the results show, the current heuristics rarely find roots that are not equal to zero. The rounded midpoint finds all integer roots exactly, if the bisection process is continued sufficiently long. Hence, simply continuing the bisection process until the interval has at most size one may already be a promising heuristic.

7.2.3 Use of Iterative Algorithms

Although we have seen that the iterative algorithms known from numerical research cannot be applied directly, it may be possible to identify certain classes of polynomials where these algorithms work particularly well. These may be polynomials with a small degree or polynomials with roots that are well separated. We can identify such polynomials using bounds on the minimal distance between two real roots as described in [Col01].

Another possibility, as shown in Figure 3.4, is to use iterative algorithms to obtain initial approximations. We may want to use other algorithms that give worse results but are faster than the companion matrix method or the Aberth method.

7.2.4 Splitting Heuristics

While one may always try new ideas for splitting heuristics, there are also some possible optimizations for the existing ones. We look at a possible optimization for the partitioning based on approximations as described in Section 3.4.3.

If the approximations are particularly good, we might want to create smaller intervals around the approximations. This could look like shown in Figure 7.1.

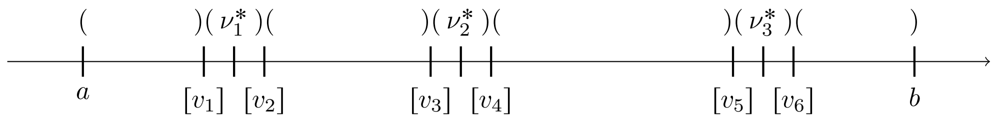

Figure 7.1: Better partitioning with approximations for all real roots

If the approximations are good and the exact roots lie within the small intervals around these approximations, this still yields an isolation directly and produces a much smaller isolation than before. However, the size of these small intervals around the approximations must be determined by some heuristic. This size may be constant or some fixed fraction of the distance between two neighboring approximations.

7.2.5 Algebraic Numbers

The fundamental reason to use interval arithmetic is the fact, that we cannot represent all polynomial roots with fractions. However, there are solutions how *algebraic numbers*, that are all numbers being a root of a polynomial with rational coefficients, can be represented.

While it might still be impossible to find these roots for polynomials with larger degree, we are able to obtain the roots in their algebraic form for degrees up to four using closed formulas.

7.3 Conclusion

Although finding the real roots of a polynomial seems to be an old problem for which many solutions exist, it turns out that there are still open questions for the application of satisfiability checking in an SMT solving framework. Most of the research had the goal to provide a result as precise as possible, however we require a result that is just guaranteed to be correct.

Using inexact computations is rather counterintuitive at first sight when correctness is required. Furthermore, using this in an actual implementation has several unexpected and delicate details. However the effort is probably worth it, as theoretical considerations and experimental results show.

Bibliography

[ABB$^+$99] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 3rd edition, 1999.

[Abe73] Oliver Aberth. Iteration methods for finding all zeros of a polynomial simultaneously. *Mathematics of Computations*, 27(122):339–344, 1973.

[BFK02] Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *Journal of Symbolic Computation*, 33(1):1–12, 2002.

[CA76] George E. Collins and Alkiviadis G. Akritas. Polynomial real root isolation using Descarte's rule of signs. In *Proceedings of the 3rd ACM Symposium on Symbolic and Algebraic Computation*, SYMSAC '76, pages 272–275. ACM, 1976.

[Cau28] Augustin-Louis Cauchy. *Exercises de mathématiques*, volume 3. Bure frères, 1828.

[CJK02] George E. Collins, Jeremy R. Johnson, and Werner Krandick. Interval arithmetic in cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 34(2):145 – 157, 2002.

[CLJA12] Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám. SMT-RAT: An SMT-compliant nonlinear real arithmetic toolbox. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing, SAT'12*, volume 7317 of *Lecture Notes in Computer Science*, pages 442–448. Springer, 2012.

[Col01] George E. Collins. Polynomial minimum root separation. *Journal of Symbolic Computation*, 32(5):467–473, 2001.

[Den97] Thomas Dence. Cubics, chaos and newton's method. *The Mathematical Gazette*, page 403, 1997.

[Don02] Jack J. Dongarra. Preface: Basic linear algebra subprograms technical (blast) forum standard. *International Journal of High Performance Computing Applications*, 16(1):1, 2002.

[EM95] Alan Edelman and H. Murakami. Polynomial roots from companion matrix eigenvalues. *Mathematics of Computations*, 64:763–776, 1995.

[Fuj16] Matsusaburô Fujiwara. Über die obere Schranke des absoluten Betrages der Wurzeln einer algebraischen Gleichung. *The Tohoku Mathematical Journal*, 10:167–171, 1916.

[Für09] Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009.

[HJVE01] Timothy J. Hickey, Qun Ju, and Maarten H. Van Emden. Interval arithmetic: From principles to implementation. *Journal of ACM*, 48(5):1038–1068, 2001.

[HK] Bruno Haible and Richard B. Kreckel. CLN - Class Library for Numbers. `http://www.ginac.de/CLN/`.

[HM90] D. G. Hook and P. R. McAree. Graphics gems. chapter Using Sturm sequences to bracket real roots of polynomial equations, pages 416–422. Academic Press Professional, Inc., 1990.

[HM97] Holly P. Hirst and Wade T. Macey. Bounding the roots of polynomials. *The College Mathematics Journal*, 28(4):292–295, 1997.

[Hou70] Alston S. Householder. *The numerical treatment of a single nonlinear equation.* McGraw-Hill, 1970.

[IEE08] *IEEE 754, IEEE Standard for Floating-Point Arithmetic.* The Institute of Electrical and Electronics Engineers, Inc., 2008. `http://dx.doi.org/10.1109/ieeestd.2008.4610935`.

[Int] Intel Corporation. Intel Math Kernel Library. `http://software.intel.com/en-us/intel-mkl/`.

[ITP] The IT++ library. `http://itpp.sourceforge.net`.

[JdM12] Dejan Jovanovic and Leonardo M. de Moura. Solving non-linear arithmetic. In *Proceedings of the 6th International Joint Conference on Automated Reasoning, IJCAR'12*, pages 339–354, 2012.

[JT68] Micheal A. Jenkins and Joseph F. Traub. A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration. Technical report, 1968.

[Ker66] Immo O. Kerner. Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen. *Numerische Mathematik*, 8(3):290–294, 1966.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View.* Springer, 1 edition, 2008.

[LA11] Ulrich Loup and Erika Ábrahám. GiNaCRA: a C++ library for real algebraic computations. In *Proceedings of the 3rd International Conference on NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 512–517. Springer, 2011.

[Sag12] Michael Sagraloff. When Newton meets Descartes: A simple and fast algorithm to isolate the real roots of a polynomial. In *Proceedings of the 37th*

International Symposium on Symbolic and Algebraic Computation, ISSAC '12, pages 297–304. ACM, 2012.

[She96] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1996.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3):281–292, 1971.

[TS02] Akira Terui and Tateaki Sasaki. Durand-Kerner method for the real roots. *Japan Journal of Industrial and Applied Mathematics*, 19(1):19–38, 2002.

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27:3–35, 2001.

[Ye94] Yinyu Ye. Combining binary search and Newton's method to compute real roots for a class of real functions. *Journal of Complexity*, 10(3):271 – 280, 1994.