Masaryk University
Faculty of Informatics

Root Isolation of High-Degree Polynomials

Master's Thesis

Adrián Elgyütt

Brno, Spring 2017

Masaryk University
Faculty of Informatics

Root Isolation of High-Degree Polynomials

Master's Thesis

Adrián Elgyütt

Brno, Spring 2017

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Adrián Elgyütt

Advisor: RNDr. Vojtěch Řehák, Ph.D

Acknowledgement

I would like to thank my advisor RNDr. Vojtěch Řehák, Ph.D for his invaluable guidance, advice and patience during writing this thesis. I also want to thank Ľuboš Korenčiak, for his helpful suggestions and advice.

Abstract

Solving polynomials is one of the oldest mathematical problems. Over the years, there have been many discoveries of fast and efficient root finding methods. In this thesis we present methods for approximating and isolating roots of high-degree polynomials with arbitrary precision. We implement these methods in the programming language Java. Lastly, we compare and evaluate the implemented methods on a set of high-degree polynomials with rational coefficients of high precision.

Keywords

polynomials, high-degree, root isolation, root finding, arbitrary precision, root approximation

Contents

1 Introduction

Finding the roots of polynomials is one of the oldest problems in mathematics. Over the centuries, many mathematicians pushed the boundaries on solving polynomials and algebraic equations including René Descartes in his book *La Géométrie* [1] and Isaac Newton in *De analysi per aequationes numero terminorum infinitas* and his many other publications. The advancement hasn't stopped and new discoveries and improvements are being made even in the present. In the past few decades, the technology has advanced rapidly, which allows us to implement these methods and use them to find roots of polynomials of very high degree.

Polynomials and root finding have many uses not only in mathematics, but in many other areas, including physics and computer science. One of the uses of polynomials can be found in the probabilistic model checker *PRISM* [2]. PRISM provides model checking for several types of probabilistic models including, but not limited to, discrete-time and continuous-time Markov chains and Markov decision processes [2]. PRISM is open source software, released under the GNU General Public License and thus there exist several branches of the program. One of the branches, fdCTMC PRISM, which stands for fixed-delay Continuous-time Markov chains, implements algorithm *Efficient Timeout Synthesis in Fixed-Delay CTMC Using Policy Iteration* [3], which requires to find the local extremes on an interval of a polynomial function. This can be done by differentiating the function and finding its roots. However, the current implementation relies on an external program to compute the roots. Thus the goal of this thesis is to create a library, which can be later used in the synthesis algorithm, that provides one or more time efficient algorithms, which solve the polynomials of high degree, with an arbitrary precision.

Structure of the thesis Chapter 2 introduces basic notions and we present three methods for a root approximation. Each of the methods is presented in a way to show how it was created. We show the convergence rate of each of the methods and describe their properties.

In the Chapter 3 we present several methods for isolating the roots, such that each interval contains exactly one root. We describe each method in detail and provide advantages and disadvantages for each method.

Chapter 4 describes the implementation of the presented methods.

Chapter 5 contains the experimental results and compares the algorithms.

Finally, Chapter 6 concludes the thesis and provides suggestions for future work.

2 Approximation of a single root

2.1 Definitions

2.1.1 Univariate polynomial

A *univariate polynomial* f is a mathematical expression of the form

$$f = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \tag{2.1}$$

where the $a_n, a_{n-1}, \ldots, a_1, a_0$ are the coefficients of the polynomial, n is any nonnegative integer and x is called an indeterminate or a variable. The highest $n \geq 0$ where $a_n \neq 0$ is called the degree of the polynomial (such n exists, since the set $\{i \,|\, a_i \neq 0\}$ is finite) and a_n is called the leading coefficient. Polynomials of degree 1 are called *linear*. We say that the polynomial is *monic*, if the leading coefficient is equal to 1. If all coefficients a_i are equal to 0, we call this special case *zero polynomial*. The degree of zero polynomial is usually defined as -1 or $-\infty$. If every a_k, $0 \leq k \leq n$, is a real number, we say that f is polynomial over $\mathbb{R}$ or simply real polynomial.

2.1.2 Roots of a univariate polynomial

Let $f = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ be a real polynomial and $c \in \mathbb{R}$. Then an element $a_n c^n + a_{n-1} c^{n-1} + \ldots + a_1 c + a_0$ is called a value of the polynomial and we denote it as $f(c)$.

Using this, we can create a polynomial function by mapping every element $x \in \mathbb{R}$, to the result of $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ [4].

Let f be a real polynomial, $c \in \mathbb{R}$. We say that c is a root of the polynomial f if $f(c) = 0$ [5].

2.2 Approximation of a root using iterative methods

The iterative methods generally require knowledge of one or more initial guesses for the desired root(s) of the polynomial. This often poses a problem itself and there are techniques and methods for finding them. The simplest method for finding a guess is by looking at the plot of the polynomial, which is often not possible (e.g. when dealing with very complex and long polynomials). Some of these methods will be shown later and thus for this section, we will assume we already have a guess.

2.2.1 Bisection method

The simplest method for finding a better approximation to a root is **bisection method**.

Theorem 2.2.1 (Intermediate value theorem [6]). *Let f be a function on $\mathbb{R}$. Consider an interval $I = [a, b]$ such that f is continuous on I and $\lambda \in \mathbb{R}$ such that λ lies between $f(a)$ and $f(b)$. Then there exists a $\gamma \in [a, b]$, such that $f(\gamma) = \lambda$.*

Now, assume a function $f : \mathbb{R} \to \mathbb{R}$ that is continuous on interval $[a, b]$ and that $f(a) \cdot f(b) < 0$. Then according to the intermediate value theorem there must be at least one root in $[a, b]$. The interval may be chosen large enough that there is more than one root, this is not a problem however, since the bisection algorithm will always converge to some root α in $[a, b]$ and a smaller interval containing only one root. Since all polynomial functions are continuous [7], we can use this theorem to create an algorithm.

Algorithm 1 Bisection algorithm

Precondition: f polynomial function, a, b interval bounds, ε precision error

```
1: function BISECTION($f, a, b, \varepsilon$)
2:     $x \leftarrow \frac{a+b}{2}$
3:     if $x - a \leq \varepsilon$ then
4:         return $c$
5:     if $f(a) \cdot f(x) < 0$ then
6:         return BISECTION($f, a, x, \varepsilon$)
7:     else
8:         return BISECTION($f, x, b, \varepsilon$)
```

Iteration	x_n	$f(x_n)$
1	1.500000	3.625000
2	1.250000	-0.867188
3	1.375000	1.023926
4	1.312500	-0.002411
5	1.343750	0.489595
6	1.328125	0.238424
7	1.320313	0.116730
8	1.316406	0.056843
9	1.314453	0.027138
10	1.313477	0.012344
11	1.312988	0.004961
12	1.312744	0.001275
13	1.312622	-0.000568
14	1.312683	0.000354
15	1.312653	-0.000106
16	1.312668	0.000124
17	1.312660	0.000010
18	1.312657	-0.000048
19	1.312659	-0.000019
20	1.312660	-0.000004

Table 2.1: Bisection algorithm on Example 2.2.1

Example 2.2.1. Find a root α of

$$2x^4 - 3x - 2 \tag{2.2}$$

with the precision $\varepsilon = 0.000001$.

It is fairly straightforward to show that there is a root located between $1 < \alpha < 2$, so we will use this interval as our initial guess. The iterations of the bisection algorithm are shown in the table 2.1.

The approximation of root on 14 decimal digits is

$$\alpha \approx 1.31265975467417 \tag{2.3}$$

The error of the final iteration is

$$|\alpha - x_{20}| \approx 0.00000024532583 \tag{2.4}$$

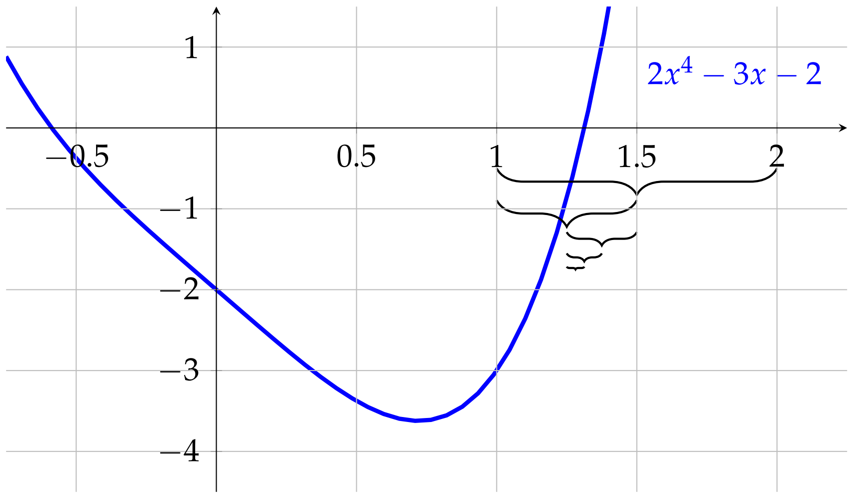


Figure 2.1: First 5 iterations of bisection algorithm on the example

which is smaller than the required error bound (0.000001). Upon closer inspection, it can be noticed that the algorithm already found a solution with enough precision in an earlier iteration (for example x_{19}) and it may seem as the computation could have been stopped right then. However, this fact was not known beforehand, because there is no possibility to predict the accuracy in an earlier iteration during computation.

Speed of convergence While every iteration gives a better approximation of the true solution, the algorithm is converging rather slowly. To examine the speed of convergence, we first need to characterize it.

Definition 2.2.2. Suppose we have a sequence of real numbers $x_0, x_1 \dots$ and a real number α such that for every $\varepsilon \in \mathbb{R}, \varepsilon > 0$ there exists $k \in \mathbb{N}$ such that $\forall l \geq k, |x_l - \alpha| < \varepsilon$. Then the sequence is said to converge to a point α, written as $\lim_{k\to\infty} x_k = \alpha$. The sequence is said to *converge linearly* if

$$\lim_{n\to\infty} \frac{|\alpha - x_{n+1}|}{|\alpha - x_n|} = c \tag{2.5}$$

for some $0 < c < 1$. The constant c is called the *rate of linear convergence* of x_n to α. The sequence is said to converge with *order p > 1* if

$$\lim_{n\to\infty} \frac{|\alpha - x_{n+1}|}{|\alpha - x_n|^p} > 0. \tag{2.6}$$

Let x_n denote the nth value of x in Algorithm 1. Then

$$\alpha = \lim_{n\to\infty} x_n \tag{2.7}$$

$$|\alpha - x_n| \leq \left[\frac{1}{2}\right]^n |b - a| \tag{2.8}$$

$$\lim_{n\to\infty} \frac{|\alpha - x_{n+1}|}{|\alpha - x_n|} = \lim_{n\to\infty} \frac{2^{-(n+1)}|b-a|}{2^{-n}|b-a|} = \frac{1}{2} \tag{2.9}$$

where $|b - a|$ denotes the length of the original interval input into the algorithm. Using Definition 2.2.2, we can say that the bisection algorithm has linear convergence with a rate of $\frac{1}{2}$. That does not necessarily mean that in every iteration the actual error decreases by a factor of $\frac{1}{2}$, but that the *average* rate of decrease is $\frac{1}{2}$. This means that it takes on average approximately 3.32 iterations ($\log_2 10$) to compute a single digit.

The major drawback of this algorithm is its very slow rate of convergence, specifically compared to other methods described in the following sections.

On the other hand, the *Bisection algorithm* has several advantages. The first of them being that it is guaranteed to converge if the prerequisites are met (i.e. the f is continuous - which is true for all polynomial functions - and $f(a) \cdot f(b) < 0$). The second one is the existence of a reasonable error bound. This method provides upper and lower bounds on the root α in every iteration and belongs in class of methods called *enclosure methods* [8].

2.2.2 Newton's method

Newton's method (sometimes also called **Newton-Raphson method**), named after Isaac Newton and Joseph Raphson, is another method for finding better approximations to a root of a real polynomial function (or in general, any real-valued function). The basic idea of the method is based on the fact that given a starting point x_0, that is sufficiently close to the root, one can approximate the function by computing its tangent line at the point $(x_0, f(x_0))$. Then, one can use the x-intercept of the tangent line, which typically provides a better approximation, and repeat this process ad infinitum [8] (or until sufficient precision is reached).

Assume that real-valued function $f(x)$ is differentiable on interval $[a, b]$ and that we have an initial approximation x_0. Then, using calculus, we can derive the formula for a better approximation x_1 as follows.

To compute the better approximation x_1, we need to use the formula that describes the slope m of our tangent line

$$m = \frac{f(x_1) - f(x_0)}{x_1 - x_0}. \tag{2.10}$$

Since we are interested in finding where the tangent line intersects the x-axis, we substitute $f(x_1)$ by 0 and manipulating the equation, we obtain

$$x_1 = x_0 - \frac{f(x_0)}{m}. \tag{2.11}$$

And since the slope m of the tangent line is the derivative of the function f at the point x_0, we obtain the formula

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{2.12}$$

The general formula is obtained by iterating the process, by replacing x_0 with x_1, ad infinitum, which gives us

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{2.13}$$

Since all polynomial functions are differentiable, we can apply this formula to our problem. Newton's method is exceptionally powerful and one of the most well known techniques for finding the roots, since it is rather easy to implement and converges very quickly.

Speed of convergence

Theorem 2.2.3 ([8, p. 60]). *Assume that $f(x), f'(x)$, and $f''(x)$ are all continuous for every x in some neighbourhood $I = [\alpha - \varepsilon, \alpha + \varepsilon]$ of α, α is a root of f, and $f'(x) \neq 0, \forall x \in I$. Then if x_0 is chosen sufficiently close to α, the iterates $x_n, n \geq 0$ of* (2.13) *will converge to α. Furthermore,*

$$\lim_{n\to\infty} \frac{\alpha - x_{n+1}}{(\alpha - x_n)^2} = \frac{-f''(\alpha)}{2f'(\alpha)} > 0 \tag{2.14}$$

proving that the convergence is quadratic.

Proof. (Sketch) First, since f is twice continuously differentiable around the root α, we can use a Taylor series expansion [8, p. 59] of second order about a point close to α, x_n, to represent $f(\alpha)$. The expansion of $f(\alpha)$ about x_n is

$$f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{f''(\xi_n)}{2!}(\alpha - x_n)^2 \tag{2.15}$$

where ξ_n is between x and α. Since α is root, we can replace $f(\alpha)$ with 0 and by rewriting the equation we get

$$-\alpha + x_n - \frac{f(x_n)}{f'(x_n)} = \frac{f''(\xi_n)}{2f'(x_n)}(\alpha - x_n)^2 \tag{2.16}$$

where we can use the definition of x_{n+1} from (2.13) and multiply the equation by -1 to get

$$\alpha - x_{n+1} = \frac{-f''(\xi_n)}{2f'(x_n)}(\alpha - x_n)^2 \tag{2.17}$$

Taking the absolute value of both sides of the equation we get

$$|\alpha - x_{n+1}| = \frac{|f''(\xi_n)|}{|2f'(x_n)|}(\alpha - x_n)^2 \tag{2.18}$$

Now we pick the sufficiently small interval $I = [\alpha - \varepsilon, \alpha + \varepsilon]$ on which $f'(x) \neq 0$ (which exists since $f'(x)$ is continuous) and then let

$$M = sup_{x \in I} \frac{|f''(x)|}{|2f'(x)|} \tag{2.19}$$

Pick x_0, such that $|\alpha - x_0| \leq \varepsilon$ and $M|\alpha - x_0| < 1$ [8, p. 61]. Then $M|\alpha - x_1| < 1$ and $M|\alpha - x_1| \leq M|\alpha - x_0|$. Using mathematical induction we can apply this argument for every $n \geq 1$, showing that $|\alpha - x_n| \leq \varepsilon$ and $M|\alpha - x_n| < 1$ for all $n \geq 1$. This, in combination with (2.18), gives us

$$|\alpha - x_{n+1}| \leq M|\alpha - x_n|^2 \tag{2.20}$$

$$M|\alpha - x_{n+1}| \leq (M|\alpha - x_n|)^2 \tag{2.21}$$

and

$$M|\alpha - x_n| \leq (M|\alpha - x_0|)^{2^n} \tag{2.22}$$

$$|\alpha - x_n| \leq \frac{1}{M}(M|\alpha - x_0|)^{2^n} \tag{2.23}$$

Since $M|\alpha - x_0| < 1$, this shows that $x_n \to \alpha$ as $n \to \infty$. The aforementioned point ξ_n is between x_n and α, which implies that $\xi_n \to \alpha$ as $n \to \infty$. Thus, from [8, p. 61] it holds that

$$\lim_{n\to\infty} \frac{\alpha - x_{n+1}}{(\alpha - x_n)^2} = -\lim_{n\to\infty} \frac{f''(\xi_n)}{2f'(x_n)} = \frac{-f''(\alpha)}{2f'(\alpha)} \tag{2.24}$$

□

However, this method has several deficiencies.

Firstly, just by observing the equation, we can see that if x_n is stationary, then x_{n+1} is undefined, since $f'(x_n) = 0$ (which means the tangent line is parallel to the x-axis).

Secondly, for some polynomial functions, the method may enter an infinite cycle. This happens if, for example, x_1 produces $x_2 = x_0$ as output, causing the method to alternate between these two results infinitely.

Thirdly, if the initial guess is not close enough or the first derivative does not behave well in the neighbourhood of a specific root, the method may skip this root as the tangent line will intercept the x-axis close to another root. This may pose a problem, if someone is looking for a root located specifically in an interval $[a, b]$, but does not pose a problem if one is simply looking for any root of a polynomial function.

Lastly, if the root (of the polynomial function) has a multiplicity greater than one (i.e. the first derivative of the polynomial function at the root is also zero), then the convergence to this root is only linear.

Theorem 2.2.4. *If the root of a polynomial function $f(x)$ has a multiplicity greater than one, then the convergence to this root is only linear.*

Proof. Let $f(x)$ be a polynomial function with a root α of multiplicity $m > 1$, i.e. $f(x) = (x-\alpha)^m g(x)$. Assume that x_0 is sufficiently close to α as stated in Theorem 2.2.3. Then

$$\begin{aligned}
x_{n+1} &= x_n - \frac{(x_n-\alpha)^m g(x_n)}{m(x_n-\alpha)^{m-1}g(x_n) + (x_n-\alpha)^m g'(x_n)} && (2.25)\\
&= x_n - \frac{(x_n-\alpha)g(x_n)}{mg(x_n) + (x_n-\alpha)g'(x_n)} && (2.26)\\
&= \frac{x_n(mg(x_n) + (x_n-\alpha)g'(x_n)) - (x_n-\alpha)g(x_n)}{mg(x_n) + (x_n-\alpha)g'(x_n)} && (2.27)\\
&= \frac{x_n(m-1)g(x_n) + x_n(x_n-\alpha)g'(x_n) + \alpha g(x_n)}{mg(x_n) + (x_n-\alpha)g'(x_n)} && (2.28)
\end{aligned}$$

which, as we get closer to the root, i.e. $x_n \approx \alpha$, gives us

$$x_{n+1} \approx x_n\frac{(m-1)}{m} + \frac{\alpha}{m} \tag{2.29}$$

or put differently

$$x_{n+1} \approx (x_n-\alpha)\frac{(m-1)}{m} + \alpha \tag{2.30}$$

from which we can conclude

$$\lim_{n\to\infty} \frac{|\alpha - x_{n+1}|}{|\alpha - x_n|} = \frac{(m-1)}{m} \tag{2.31}$$

which by Definition 2.2.2 proves the linear convergence. □

Error bounds To estimate the error and provide error bounds of our computation, we will need to use a variation of the *mean value theorem*.

Theorem 2.2.5 (Mean value theorem [9]). *Let f be a real-valued polynomial function and a, b real numbers, such that $a < b$. Then there exists some $c \in (a,b)$ such that*

$$f'(c) = \frac{f(b)-f(a)}{b-a}. \tag{2.32}$$

Roughly speaking, it states that given a curve, which starts at point a and ends in point b, there is at least one point at which the tangent to the curve is parallel to the secant connecting the two points. Using this theorem on a polynomial function f with a root α and an approximation x_n, $\alpha < x_n$ we get

$$f'(\xi_n) = \frac{f(x_n) - f(\alpha)}{x_n - \alpha} \tag{2.33}$$

where ξ_n being between α and x_n. Since α is the root of f, then we can remove $f(\alpha)$ and simplifying we get

$$\alpha - x_n = \frac{-f(x_n)}{f'(\xi_n)}. \tag{2.34}$$

Similarly, if $x_n < \alpha$, we arrive to the same formula. If $f'(x)$ is not changing rapidly between x_n and α, then $f'(\xi_n) \approx f'(x_n)$. Combining this with the definition of Newton's method we get

$$\alpha - x_n \approx \frac{-f(x_n)}{f'(x_n)} = x_{n+1} - x_n. \tag{2.35}$$

This gives us an absolute error estimate

$$\alpha - x_n \approx x_{n+1} - x_n \tag{2.36}$$

and a relative error estimate

$$\frac{\alpha - x_n}{\alpha} \approx \frac{x_{n+1} - x_n}{x_{n+1}}. \tag{2.37}$$

The Newton algorithm Using the Newton formula (2.13) and the error estimates (2.36), (2.37) provided above, we can create Algorithm 2.

The *max* variable rules out the possibility of getting stuck in an endless loop, when the method oscillates between two points and does not converge (meaning there is no root located in the interval (a, b)). However, in the case that the function iterates through the maximum amount of iterations *max* and returns no root (i.e. $i = max$ and *err* is 1), but one knows the root of the f is located in (a, b), one may try

Algorithm 2 Newton's algorithm

Precondition: f polynomial function, f' derivative of f, a, b interval bounds, ε precision error, max number of maximum iterations, err error flag

```
1: function Newton($f, f', a, b, \varepsilon, max$)
2:     $err \leftarrow 1$
3:     $x_0 \leftarrow \frac{a+b}{2}$
4:     for $i \leftarrow 1, max$ do
5:         $denom \leftarrow f'(x_0)$
6:         if $denom = 0$ then
7:             $err \leftarrow 2$
8:             return $err, x_0$
9:         $x_1 \leftarrow x_0 - \frac{f(x_0)}{denom}$
10:        if $\left(\left|\frac{x_1 - x_0}{x_1}\right| \leq \varepsilon\right)$ and $(|x_1 - x_0| \leq \varepsilon)$ then
11:            $err \leftarrow 0$
12:            return $err, x_1$
13:        if $(x_1 < a)$ or $(x_1 > b)$ then
14:            $err \leftarrow 3$
15:            return $err, x_1$
16:        $x_0 \leftarrow x_1$
17:    return $err, x_1$
```

to increase the max or narrow the interval (a, b) (e.g. using bisection) and try running the algorithm again.

The lines 13 to 15 of Algorithm 2 serve as a prevention against the case of overshooting the root. These lines may be omitted if one does not have particular interest in finding the root from the specific interval (in this case, input interval a, b may be switched for a single starting point x_0). As in previous case, if the algorithm ends prematurely because of the latest iteration being outside of the interval, but one is certain that the root is located inside the interval, one may try narrowing the interval (thus giving a better starting point).

The termination condition is a combination of the error bounds, both absolute and relative. The reason for this choice is that while relative error works well in small magnitude (i.e. $<< 1$) and worse when the root is large in magnitude (i.e. $>> 1$), the opposite is true for absolute error. That is because for error $\varepsilon = 10^{-e}$ the relative error of (2.37) gives at least $e - 1$ correct digits. This means that for an approximation in the floating point representation at least $e - 1$ significant digits are correct. Whereas the absolute error of (2.36) provides at least $e - 1$ correct digits after the decimal point (of a number in the decimal notation).

E.g. let $f(x) = x^3 - 100x^2 + x - 100$, which has a root $\alpha = 100$ and set $\varepsilon = 0.1$. Then, using only the relative error bound and setting $a = 0, b = 2000$, the algorithm returns as a result $x = 101.5$ after 7 iterations, ending too early. Using the absolute error bound, the algorithm ends after 9 iterations with the desired result $x = 100.0$. Now let $f(x) = x^3 - 0.0001x^2 + x - 0.0001$ which has a root $\alpha = 0.0001$ and set $\varepsilon = 0.00001$. Then, using only the absolute error bound and setting $a = 0, b = 100$, the algorithm returns as a result $x = 0.000104$ after 13 iterations. Then, using only the relative error bound and setting $a = 0, b = 100$, the algorithm returns as a result $x = 0.0001000$ after 14 iterations. While both results are within the precision error, the relative error gives a more precise result. Thus the best results are achieved combining both bounds.

Example 2.2.2. Find a root α of

$$2x^4 - 3x - 2 \tag{2.38}$$

with the precision $\varepsilon = 0.000001$.

The initial interval is the same as in bisection algorithm to make results comparable, i.e. $a = 1, b = 2$.

Iteration	c_n	$f(c_n)$
1	1.500000	3.625000
2	1.348958	0.575658
3	1.314358	0.025697
4	1.312664	0.000060
5	1.312660	0.000001

Table 2.2: Newton's algorithm on Example 2.2.2

With correct approximation being $\alpha \approx 1.31265975467417$, the error of the final iteration is $|\alpha - x_5| \approx 0.00000024532583$ which is within the allowed precision error.

While the Newton's algorithm solves some of the deficiencies of the bisection method, some remain. The method might sometimes fail (when the first derivative of x_n is zero) and the convergence for the roots of higher multiplicity is only linear.

On the other hand, as we can see from Table 2.2, the algorithm converges much faster than the previous method, finding the sufficient approximation in only 5 iterations. This makes Newton's method superior to the bisection method in most of the cases.

2.2.3 Halley's method

Halley's method, named after its creator Edmond Halley, is a method that in addition to first derivative uses a second derivative of the function as well. While Newton's method could be geometrically expressed as a series of tangent lines which roots (x-intercepts) converge to the root of function, there is no such obvious interpretation for Halley's method. However, as Newton's method can be derived via a first order Taylor polynomial, Halley's method can be derived via a second order Taylor polynomial [10],

$$y(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2!}(x - x_n)^2, \tag{2.39}$$

where x_n is an approximation of x, such that $f(x) = 0$. As the objective is to calculate a point x_{n+1} where the y function intersects x-axis, we set $y(x) = 0$ and the objective is then solving the equation

$$0 = f(x_n) + f'(x_n)(x_{n+1} - x_n) + \frac{f''(x_n)}{2}(x_{n+1} - x_n)^2 \qquad (2.40)$$

for x_{n+1}. Following [10] and simplifying the equation to express x_{n+1} on the left side we obtain

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n) + \frac{f''(x_n)}{2}(x_{n+1} - x_n)}. \qquad (2.41)$$

Substituting the difference $x_{n+1} - x_n$ with $-\frac{f(x_n)}{f'(x_n)}$ defined in (2.13), we obtain

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2f'(x_n)^2 - f(x_n)f''(x_n)}, \qquad (2.42)$$

what is known as Halley's method.

Speed of convergence The Halley's method converges to the root cubically [11, 12], as opposed to Newton's method, which converges quadratically. The proof of cubic convergence follows similarly as Newton's, using Taylor series expansion. The cubic convergence is achieved at the cost of more complex computation in each iteration. In addition, if the second derivative is close to or exactly 0 then the convergence speed is very similar to the Newton's method.

Halley's algorithm The Halley's algorithm has similar structure to Algorithm 2, only replacing lines 5 and 9 with the equation 2.42.

Example 2.2.3. Find a root α of

$$2x^4 - 3x - 2 \qquad (2.43)$$

with the precision $\varepsilon = 0.000001$.

The initial interval remains the same (i.e. $a = 1, b = 2$) as in the previous algorithms to make the results comparable.

Iteration	c_n	$f(c_n)$
1	1.318039	0.081800
2	1.312660	0.000001

Table 2.3: Halley's algorithm on Example 2.2.3

The algorithm finishes with the same result as the result provided by the bisection algorithm and the Newton's algorithm while ending in only 2 iterations, as opposed to 21 in Algorithm 1 (as seen in Table 2.1) and 5 in Algorithm 2 (as seen in Table 2.2). The Halley's algorithm has the same disadvantages as Newton's algorithm, while providing a higher speed of convergence, at the cost of higher computation time in every iteration. If the degree of the polynomial function is very high, the evaluation $f(x_n), f'(x_n), f''(x_n)$ takes a significant amount of time. This, combined with other additional computations may result in total computation time being similar or even higher than Newton's method. Additionally, if the second derivative is very close to 0 (with respect to error tolerance), the convergence rate is very similar to Newton's. Thus, it doesn't provide significant advantage over Newton's method (and may even result in higher total computational time).

3 Solving polynomials

While the previous chapter focused on getting better approximation of a single root, this chapter focuses on solving polynomial functions and isolating the roots.

3.1 Ruffini-Horner's method

The **Ruffini-Horner's** method relies on what is known as *fundamental theorem of algebra.*

Theorem 3.1.1 (Fundamental theorem of algebra [13])**.** *Given any positive integer $n \geq 1$ and any univariate polynomial f with complex coefficients such that the degree of f is n, the polynomial f has at least one complex root.*

This theorem has few important corollaries.

Corollary 3.1.2. *Every univariate polynomial f of degree $n \geq 1$ has exactly n complex roots.*

Corollary 3.1.3. *Any univariate polynomial f of degree $n \geq 1$ can be factored as $f = (x - z_1)(x - z_2) \dots (x - z_n)$, where $z_1, \dots, z_n$ are the complex roots of f.*

Corollary 3.1.4. *Any real univariate polynomial f of degree $n \geq 1$ can be factored as $f = (x - z_1)(x - z_2) \dots (x - z_m)g(x)$, where $z_1, \dots, z_m$ are the real roots of f, $m \leq n$ and $g(x)$ is irreducible in $\mathbb{R}$.*

Definition 3.1.5. Let f be a real univariate polynomial and $f = (x - z_1)^{k_1}(x - z_2)^{k_2} \dots (x - z_m)^{k_m} g(x)$ factorization of f such that $z_1, z_2, \dots, z_m$ are pairwise distinct, $m \leq n$, $\forall 1 \leq i \leq m : k_i \leq 1$ and $g(x)$ is irreducible in $\mathbb{R}$. Then if $k_i > 1$, $i \leq m$, we say that z_i is *multiple root* and if $k_i = 1$, we say that z_i is *simple root.*

The implication of Corollary 3.1.4 is that we can start by finding any root α of $f(x)$, then divide $f(x)$ by factor $(x - \alpha)$ to obtain $f_1(x)$ such that $f(x) = (x - \alpha) f_1(x)$ and repeat the process for $f_1(x)$ until we reach $f_m(x)$ such that it is irreducible in $\mathbb{R}$ (i.e. has no real roots).

Because in each step we are dividing $f_i(x)$, $0 \leq i < m$ by a linear monic polynomial, we can use the efficient technique for the division

Algorithm 3 Ruffini-Horner's algorithm

Precondition: f polynomial function, x_0 initial guess, ε precision error, max max amount of iter. for Newton's alg. (Algorithm 2)

```
function RuffiniHorner($f, x_0, \varepsilon, max$)
    $roots \leftarrow \emptyset$
    $f_0 \leftarrow f$
    $remainder \leftarrow 0$
    $error \leftarrow 0$
    while $remainder = 0 \& error = 0$ do
        $error, root \leftarrow Newton(f_0, f_0', x_0, \varepsilon, max)$
        if $error = 0$ then
            $roots \leftarrow roots \cup root$
            $f_1, remainder \leftarrow f_0/(x - root)$    using Ruffini/Horner rule
        if $f_1 = 1$ then
            return $roots$
        $f_0 \leftarrow f_1$
    return $roots$
```

known as *Ruffini's rule* or *Horner's method*, described in [14]. This gives us a simple Algorithm 3.

The advantages of Algorithm 3 are the efficiency of the division and simplicity, which in combination with Newton's method result in fast algorithm.

On the other hand, the division in each step produces a slight error in precision, which accumulates over the iterations and could cause larger discrepancy in the latter roots than desired.

3.2 Isolation of the roots

In this section, I present three algorithms that aim to isolate the roots into smaller separate intervals, each containing exactly one root. Then, using the combination of algorithms presented in the previous chapter, we approximate the roots to the allowed tolerance, thus achieving the desired result.

3.2.1 Isolation of roots using Sturm's theorem

Definition 3.2.1. A *Sturm chain* or *Sturm sequence* of a polynomial f is a finite sequence of polynomials $f_0, f_1, \dots f_m$ of decreasing degree with following properties:

- $f = f_0$
- f has simple roots
- if $f(\alpha) = 0$, then $sgn(f_1(\alpha)) = sgn(f'(\alpha))$
- $\forall 0 < i < m$: if $f_i(\alpha) = 0$, then $sgn(f_{i-1}(\alpha)) = -sgn(f_{i+1}(\alpha))$
- $sgn(f_m(x))$ is constant for all $x \in \mathbb{R}$.

The Sturm chain of f can be created by applying Euclid's algorithm to $f(x)$ and $f'(x)$ [15, 16]:

$$\begin{aligned}
f_0(x) &= f(x),\\
f_1(x) &= f'(x),\\
f_2(x) &= -r(f_0, f_1) = f_1(x)q_0(x) - f_0(x),\\
f_3(x) &= -r(f_1, f_2) = f_2(x)q_1(x) - f_1(x),\\
&\vdots\\
f_m(x) &= -r(f_{m-2}, f_{m-1}) = f_{m-1}(x)q_{m-2}(x) - f_1(x),\\
0 &= -r(f_{m-1}, f_m).
\end{aligned}$$

The $r(f_i, f_{i+1})$ represents the remainder and the q_i represents the quotient of the polynomial long division $f_i(x) = q_i(x)f_{i+1}(x) + r(f_i, f_{i+1})$ for every $0 \leq i < m$. As the degree of the remainder $deg(r(f_i, f_{i+1}))$ is at most $deg(f_{i+1}) - 1$, the maximum length of the chain is $m \leq deg(f_0)$.

Definition 3.2.2. Let $\lambda_0, \lambda_1, \lambda_2, \ldots$ be a finite or infinite sequence of real numbers. Suppose $a < b; a, b \in \mathbb{N}$ and the following condition holds

$$sgn(\lambda_a) = -sgn(\lambda_b) \land \forall i, a < i < b : \lambda_i = 0. \tag{3.1}$$

Then this is called a *sign variation* or *sign chance* between λ_a and λ_b.

To provide the Sturm's theorem, let $\sigma(f, \xi)$ represent the number of sign changes in the sequence $f_0(\xi), f_1(\xi), \ldots, f_m(\xi)$, where $f_0, f_1, \ldots, f_m$ is Sturm chain of f.

Theorem 3.2.3 (Sturm[15]). *Assume $f(x)$ is a real polynomial without a root of multiplicity greater than one and a, b are real numbers such that $a < b, f(a) \neq 0, f(b) \neq 0$. Then the number of real roots in interval $(a, b]$ is equal to $\sigma(f, a) - \sigma(f, b)$.*

Remark. The theorem can be strengthened (as proven in [17]) to any polynomial, including the ones with roots with higher multiplicity. Then the difference of sign changes $\sigma(f, a) - \sigma(f, b)$ provides the number of *distinct* roots in $(a, b]$.

Using this theorem and combining it with Newton's method and bisection, we can create Algorithm 5.

Algorithm 4 Sturm's algorithm

Precondition: f polynomial function and its first derivative f', a, b interval bounds

```
function Sturm($f, f', a, b$)
    $\sigma_a \leftarrow 0$
    $\sigma_b \leftarrow 0$
    $f_0 \leftarrow f$
    $f_1 \leftarrow f'$
    if $sgn(f_0(a)) \neq sgn(f_1(a))$ then
        $\sigma_a \leftarrow \sigma_a + 1$
    if $sgn(f_0(b)) \neq sgn(f_1(b))$ then
        $\sigma_b \leftarrow \sigma_b + 1$
    while $f_1 \neq 0$ do
        $r \leftarrow f_0 \ mod \ f_1$
        $f_0 \leftarrow f_1$
        $f_1 \leftarrow -r$
        if $sgn(f_0(a)) \neq sgn(f_1(a))$ then
            $\sigma_a \leftarrow \sigma_a + 1$
        if $sgn(f_0(b)) \neq sgn(f_1(b))$ then
            $\sigma_b \leftarrow \sigma_b + 1$
    return $\sigma_a - \sigma_b$
```

Algorithm 5 Root finding (sturm) algorithm

Precondition: f, f' polynomial and its derivative, a, b interval bounds, ε precision error, *max* number of maximum iterations

```
1:  function RootFindingSturm($f, f', a, b, \varepsilon, max$)
2:      $roots \leftarrow \emptyset$
3:      $root\_count \leftarrow Sturm(f, f', a, b)$
4:      if $root\_count > 0$ then
5:          if $root\_count = 1$ then
6:              $error, root \leftarrow Newton(f, f', a, b, \varepsilon, max)$
7:              if $error = 0$ then
8:                  $roots \leftarrow roots \cup root$
9:                  return $roots$
10:         $c \leftarrow \frac{a+b}{2}$
11:         $roots \leftarrow roots \cup RootFindingSturm(f, f', a, c, \varepsilon, max)$
12:         $roots \leftarrow roots \cup RootFindingSturm(f, f', c, b, \varepsilon, max)$
        return $roots$
```

The algorithm has several disadvantages. The major deficiency of this algorithm is the time complexity of evaluating the polynomials of the chain in points a and b to calculate the difference in sign changes. Since there can be up to n polynomials in the chain, where n is the degree of f, this can result up to $O(n^2)$ multiplications and additions in every iteration of bisection.

Another issue is that while creating the Sturm's chain, depending on the implementation, if an error is introduced during the polynomial long division, this leads to snowball effect in the subsequent divisions, which may result in a different number of the polynomials in the chain computed than the actual number of the polynomials in chain. This may happen if the coefficients are represented as floating-point numbers with set precision, instead of representing them as fractions of integers (which is often impractical). This can result into the number of sign changes $\sigma(\xi)$ being incorrect, leading to the number of roots located in the interval $(a, b]$ being wrong and causing some roots not being found. However, this can be avoided by reducing the tolerance of error sufficiently.

3.2.2 Isolation of roots using Vincent's theorem

Vincent published his theorem in 19th century [18, 19], but because of the appearance of Sturm's theorem, it was forgotten until the end of 20th century, when it was brought back by [20]. This led to the creation of a second version of the theorem in [21].

Theorem 3.2.4 (Vincent (continued fractions version) [18, 19])**.** *Given a polynomial function $f(x)$ with rational coefficients and without multiple roots, if one sequentially performs replacements of the form*

$$x \leftarrow \alpha_1 + \frac{1}{x}, x \leftarrow \alpha_2 + \frac{1}{x}, x \leftarrow \alpha_3 + \frac{1}{x}, \dots, \tag{3.2}$$

where $\alpha_1 \geq 0$ is an arbitrary nonnegative integer and $\alpha_2, \alpha_3, \dots$ are arbitrary positive integers, $\alpha_i > 0, i > 1$, then the resulting polynomial has either zero sign variations (see Definition 3.2.2) or one sign variation. In the former case, there are no positive roots, whereas in the latter case, the equation has exactly one positive root, which is represented by the continued fraction

$$\alpha_1 + \frac{1}{\alpha_2 + \frac{1}{\alpha_3 \frac{1}{\ddots}}}. \tag{3.3}$$

The negative root can be treated the same way, by simply performing substitution $x \leftarrow -x$ on $f(x)$. The requirement that the polynomial function $f(x)$ has no multiple roots does not restrict the generality of the theorem, because in the case that polynomial contains multiple roots, we can first apply square-free factorization (e.g. by dividing the polynomial f with GCD(f, f') to reduce all multiple roots to simple) and then isolate the roots of the square-free factor [22].

Theorem 3.2.5 (Vincent (bisection version) [21])**.** *Let $f(x)$ be a real polynomial of degree n which has only simple roots. It is possible to determine a positive quantity δ so that for every pair of positive real numbers a, b with $|b - a| < \delta$, every transformed polynomial of the form*

$$\phi(x) = (1 + x)^n f\left(\frac{a + bx}{1 + x}\right) \tag{3.4}$$

has exactly 0 or 1 sign variations. The second case is possible if and only if $f(x)$ has a single root within (a, b).

As in the previous case, the negative roots can be handled by substituting $x \leftarrow -x$ and the condition of $f(x)$ only containing simple roots can be resolved by applying square-free factorization.

Using this theorem, we can create a simple test that gives us the upper bound on the positive roots inside the interval (a, b):

$$\sigma_{(a,b)}(f) = \sigma\left((1+x)^n f\left(\frac{a+bx}{1+x}\right)\right) = \sigma_{(b,a)}(f), \qquad (3.5)$$

where $\sigma(\xi)$ is the number of sign variations of the sequence of coefficients $a_0, a_1, \ldots, a_n$ of ξ.

There are two major algorithms, each based on the different version of the theorem. We describe them in the following subsections.

Vincent–Akritas–Strzeboński (VAS) [23]

Let the *Möbius transformation* [24]

$$M(x) = \frac{ax+b}{cx+d} \qquad a, b, c, d \in \mathbb{N}; ad - bd \neq 0$$

represent the finite continuous fraction that substitutes x in $f(x)$ such that the substitution results in the transformed polynomial

$$g(x) = (cx+d)^{deg(f)} f\left(\frac{ax+b}{cx+d}\right) \qquad (3.6)$$

with one sign change. Then the positive root of $f(x)$ (located in $(0, \infty)$) corresponds to the root of $g(x)$ in $(min(\frac{a}{c}, \frac{b}{d}), max(\frac{a}{c}, \frac{b}{d}))$. Thus, to isolate the positive roots, one needs to compute a, b, c, d such that they result in the polynomial (3.6) with one sign change. The Descartes' rule of signs [1], states that the number of positive roots of $f(x)$ is equal to the number of sign changes of the coefficients $\sigma(f)$ or less than it by an even number. We can use this simple test to determine (line 2 and 4) if the f has exactly zero or one positive root or if it has more. If there are more, we first transform the polynomial by moving the lowest positive closer to 0 and then we split the polynomial into two transformed ones - the first representing the interval between $(0, 1)$ and the second representing the interval $(1, \infty)$. Every time we perform a substitution on f, we perform the corresponding substitution on M. This allows us to retroactively compute the corresponding interval in the original polynomial.

Thus Algorithm 6 returns intervals such that each contains exactly one positive root. To isolate the negative intervals, we simply substitute $x \leftarrow -x$, execute the algorithm again and then unite the sets of intervals. Then we can use one of the approximation algorithms (e.g. Newton's algorithm (2)) and approximate the root in every interval.

The lower bound (lb) of f can be computed by computing upper bound (ub) of $g(x) = x^{deg(f)} f(\frac{1}{x})$ and then $lb_f = \frac{1}{ub_g}$, as shown [16]. [16] also provides two efficient upper bounds.

Algorithm 6 VAS

Precondition: f polynomial function, M möbius transformation

```
1:  function VAS($f, \varepsilon$)
2:      if $\sigma(f) = 0$ then
3:          return $\emptyset$
4:      if $\sigma(f) = 1$ then
5:          $a \leftarrow min(M(0), M(\infty))$ where $M(\infty) = \frac{a}{c}$ if $c \neq 0$
6:          $b \leftarrow max(M(0), M(\infty))$ and if $c = 0$ then $M(\infty)$ =an upper
    bound on the positive roots
7:          return $\{(a, b)\}$
8:      $lb \leftarrow$ a lower bound on the positive roots of $f(x)$
9:      if $lb \geq 1$ then
10:         $f \leftarrow f(x + lb)$
11:         $M \leftarrow M(x + lb)$
12:     $f_{01} \leftarrow (x+1)^{deg(f)} f(\frac{1}{x+1})$
13:     $M_{01} \leftarrow M(\frac{1}{x+1})$
14:     $f_{1\infty} \leftarrow f(x + 1)$
15:     $M_{1\infty} \leftarrow M(x + 1)$
16:     $m \leftarrow M(1)$
17:     if $f(1) \neq 0$ then
18:         return $VAS(f_{01}, M_{01}) \cup VAS(f_{1\infty}, M_{1\infty})$
19:     else
20:         return $VAS(f_{01}, M_{01}) \cup VAS(f_{1\infty}, M_{1\infty}) \cup \{(m, m)\}$
```

Vincent–Collins–Akritas (VCA) [25]

While the previous algorithm required computing the upper and lower bound on the positive roots, this algorithm allows the user to enter the desired search interval (a, b). This was not possible in the previous case, as the Descartes' rule gives upper bound on all positive roots. However, using the test (3.5) (or rather a special case of it called Budan's test [26]) allows us to specify the interval we are interested in locating the roots.

The Budan's test [26] is a version of the test (3.5) where $a = 0, b = 1$, which results in

$$\sigma_{(0,1)}(f) = \sigma\left((1+x)^{deg(f)} f\left(\frac{1}{1+x}\right)\right). \tag{3.7}$$

If $\sigma_{(0,1)}(f)$ is equal to 0, there is no root located within $(0, 1)$, while if $\sigma_{(0,1)}(f) = 1$, there is precisely 1 root. Any number higher than 1 gives upper bound on the roots located within the interval $(0, 1)$ such that the exact number is equal to the $\sigma_{(0,1)}(f)$ or less by an even number (multiplicities counted), meaning if the upper bound is equal to even number, they may be zero roots in the interval.

The first step is then to make sure that all roots of $f(x)$ we want to isolate are within the entered interval (a, b). To do this, we first perform a substitution $x \leftarrow x + a$ on f to obtain f_a. Then we perform a substitution $x \leftarrow x \cdot (b - a)$ on f_a to obtain f_{ab}. This gives us a bijection of the interval (a, b) of f on $(0, 1)$ of f_{ab}. We use this as the initial input of 7, along with a, b.

Afterwards we perform the Budan's test (3.7) on the transformed polynomial f_{ab}. If there is 0 or 1 root, we can finish the search. Otherwise we split the f_{ab} into two polynomials, $f_{0\frac{1}{2}}$ and $f_{\frac{1}{2}1}$ such that there is bijection between the intervals $(0, \frac{1}{2})$ and $(\frac{1}{2}, 1)$ of f_{ab} and the intervals $(0, 1)$ of $f_{0\frac{1}{2}}$ and $f_{\frac{1}{2}1}$, respectively. Then we can use these new polynomials recursively.

If the original polynomial f is not square-free, this, in theory, will cause an endless recursion of the algorithm, since the Budan's test will always return a number greater than one. However, there are 2 solutions to this problem:

1. perform a square free factorization on f,

2. select a tolerance ε such that if $|a-b| < \varepsilon$ algorithm ends.

Since the second solution is much simpler and in line with the fact that we are approximating the roots to a certain precision, we apply it to obtain Algorithm 7. If the upper bound is odd, there are either multiple roots or a single root with odd multiplicity. If there are multiple roots, then this does not influence the result, since they are closer to each other than the desired tolerance and thus we are unable to distinguish them. If the upper bound is even, then there is no guarantee that the interval contains root. To test if there is a root, we could try executing the Newton's algorithm (Alg. 2) in this interval. However, if the function approaches 0 very close (i.e. closer than ε), this could result in a false positive. Thus, one should be vary of this when using Algorithm 7 on non square-free polynomial.

Algorithm 7 VCA

Precondition: f polynomial function, a, b searched interval, ε tolerance

```
function VCA($f, a, b, \varepsilon$)
    $bud \leftarrow \sigma\left((1+x)^{deg(f)} f\left(\frac{1}{1+x}\right)\right)$
    if $bud = 0$ then
        return $\emptyset$
    if $bud = 1$ then
        return $\{(a,b)\}$
    if $|a-b| < \varepsilon$ then
        return $\{(a,b)\}$
    $f_{0\frac{1}{2}} \leftarrow (2)^{deg(f)} f(\frac{x}{2})$
    $f_{\frac{1}{2}1} \leftarrow (2)^{deg(f)} f(\frac{x+1}{2})$
    if $f(\frac{1}{2}) \neq 0$ then
        return $VCA(f_{0\frac{1}{2}}, a, \frac{a+b}{2}, \varepsilon) \cup VCA(f_{\frac{1}{2}1}, b, \varepsilon)$
    else
        return $VCA(f_{0\frac{1}{2}}, a, \frac{a+b}{2}, \varepsilon) \cup VCA(f_{\frac{1}{2}1}, b, \varepsilon) \cup \{[\frac{a+b}{2}, \frac{a+b}{2}]\}$
```

4 Implementation

All of the methods presented are implemented in the programming language Java. The main reason for this choice is that the extension of the probabilistic model checker Prism [2] - Prism-fdCTMC (currently under development) is implemented in Java as well. The fdCTMC branch contains timeout synthesis [3] which requires finding local extremes of polynomial function within an interval. This problem then reduces to finding the roots of the first derivative of the polynomial. In the current implementation, the computation of the roots is done externally, either by connecting to a remote server, which computes the roots via Maple or by outputting Maple input data on screen (which requires the user to have a copy of Maple installed). This is quite impractical and thus our implementation can be used to replace the current one, so that the user doesn't have to rely on the internet connection, the stability of server or ownership of an external program.

4.1 State of the art

After the choice of programming language fell on Java, I first tried to find a mathematical library that already contains root finding of polynomials and supports arbitrary precision. The most notable mathematical or scientific libraries we found were Commons Math: The Apache Commons Mathematics Library[1], Jsience[2] and Efficient Java Matrix Library[3]. The criteria that the libraries needed to satisfy were arbitrary precision, root finding of polynomials and preferably representation of polynomials (with basic operations). As Figure 4.1 shows, none of the libraries satisfied all criteria and thus the choice was made to create an implementation which satisfies all of them.

1. `http://commons.apache.org/proper/commons-math/`
2. `http://jscience.org`
3. `http://ejml.org/wiki/index.php?title=Main_Page`

Library	Repr. of pol.	Root finding	Precision accuracy
JScience	yes	no	arbitrary
EJML	no	yes	double
Commons Math	yes	yes	double

Table 4.1: Comparison of Java libraries

4.2 PolynomialRoots implementation

The decimal numbers are represented by the class `BigDecimal`[4], which allows an arbitrary precision. A `BigDecimal` consists of an arbitrary precision integer unscaled value and a 32-bit integer scale. The value of the number represented by `BigDecimal` is unscaled value multiplied by 10^{-scale}.

The `BigDecimal` class provides basic mathematical operations and operations for comparison, rounding and manipulation of scale. With almost every division made, a scale and a type of rounding must be associated. In our implementation, we chose standard rounding (i.e. $< .5$ is rounded down, $\geq .5$ is rounded up) and the scale was chosen as scale of ε (if the algorithm contained one).

To represent the polynomials, we created the class `Polynomial`. The class provides basic operations with polynomials, such as addition or multiplication, or computing the first derivative. Full information can be found in the Javadoc documentation.

The implemented methods can be found in the class `PolynomialRootFinding`. During the implementation, we tried to avoid the duplication of code by reusing the methods we implemented. E.g. since the `BigDecimal` class doesn't provide $\sqrt[n]{a}$, where a is `BigDecimal`, we used Newton's algorithm to find the root of $x^n - a = 0$, with starting point in 1. Or to compute the square-free factorization of the polynomial, we used the Sturm's chain to find the GCD of f and f' and subsequently used the polynomial long division to remove the GCD factor from f.

Also when possible, we tried to evaluate some expressions efficiently. E.g. the computing the Sturm's chain in Algorithm 5 can be

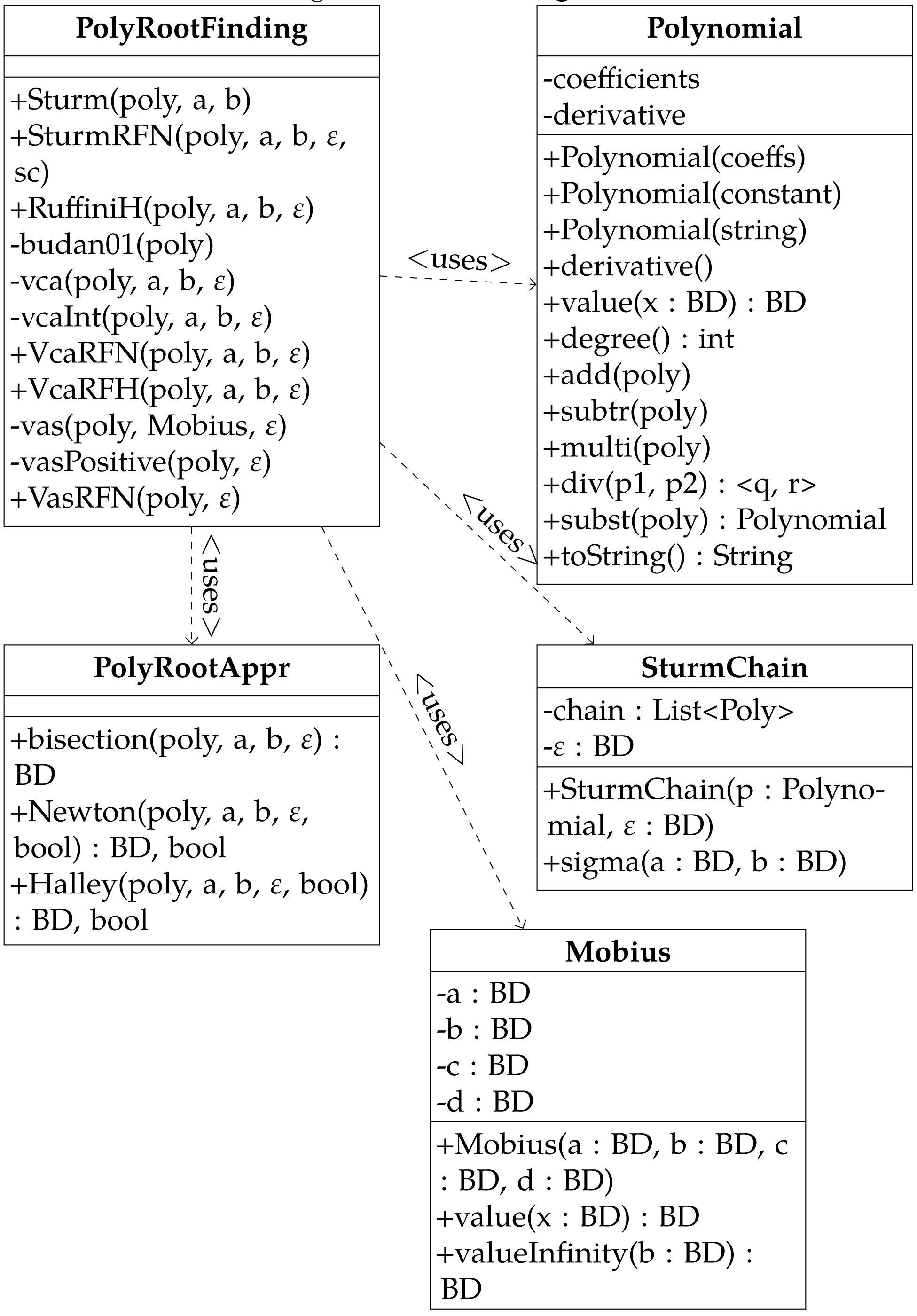
4. `https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`

done only once, by creating a class representing the chain instead of calling Algorithm 4 and computing the chain every time. Or when performing the Budan's test in Algorithm 6, instead of performing substitution $\frac{1}{1+x}$, expanding it and subsequently multiplying it by $(x+1)^{deg(f)}$, the more efficient way is to simply reverse the coefficients and perform substitution $(x+1)$.

The source code and Javadoc providing the full description of all implemented methods can be found in the attachment.

Figure 4.1 shows the class diagram. *BD* stands for `BigDecimal`, *poly* has type `Polynomial` and a, b, ε all have type `BigDecimal`.

Figure 4.1: **Class diagram**



5 Experiments

The implemented methods were tested with on a set of 40 polynomials with varying degrees from 34 up to 101. The polynomials were obtained as the output of the synthesis algorithm implemented in the fdCTMC branch of Prism. Thus, they have certain common characteristics - with the increasing degree, the coefficients shrink by approximately one decimal point (i.e. if the absolute coefficient a_0 is 1, the leading coefficient a_n is approximately 10^{-n}). All tested polynomials can be found in the attachment.

The tested algorithms were Sturm's (Algorithm 5), VAS (Algorithm 6) and VCA (Algorithm 7) to first isolate the roots within intervals. Subsequently, every isolated root was approximated using Newton's method. Algorithm 7 was also tested with Halley's method. The Algorithm 3 proved to be unreliable, as the rounding after division caused the latter roots to be not sufficiently precise and thus is not included.

The upper bounds used in Algorithm 6 were Kojima bounds [27], because they are very fast to compute, while providing reasonable bounds. The search interval for Algorithms 5 and 7 was $(-100, 100)$.

Tables 5.1 and 5.2 compare the computation time of polynomials with degree 34-67 and 68-101 respectively, with allowed error of precision $\varepsilon = 10^{-20}$.

Tables 5.3 and 5.4 compare the computation time of polynomials with degree 34-67 and 68-101 respectively, with allowed error of precision $\varepsilon = 10^{-50}$.

	VCA(Newton)	VCA(Halley)	VAS(Newton)	Sturm
Minimum	61	63	76	119
Median	235	240	297	716
Maximum	633	647	387	3158
Average	285	286	259	1075

Table 5.1: Polynomials with degree 34-67, precision $\varepsilon = 10^{-20}$ (time in miliseconds)

	VCA(Newton)	VCA(Halley)	VAS(Newton)	Sturm
Minimum	838	861	102	697
Median	997	1007	240	4434
Maximum	1362	1361	369	7173
Average	1021	1020	252	3901

Table 5.2: Polynomials with degree 68-101, precision $\varepsilon = 10^{-20}$ (time in miliseconds)

	VCA(Newton)	VCA(Halley)	VAS(Newton)	Sturm
Minimum	160	167	168	435
Median	631	648	988	2717
Maximum	1631	1657	1309	12559
Average	732	745	775	4209

Table 5.3: Polynomials with degree 34-67, precision $\varepsilon = 10^{-50}$ (time in miliseconds)

	VCA(Newton)	VCA(Halley)	VAS(Newton)	Sturm
Minimum	2283	2355	325	2735
Median	2897	2761	685	17534
Maximum	3572	3571	1252	28358
Average	2846	2870	741	15378

Table 5.4: Polynomials with degree 68-101, precision $\varepsilon = 10^{-50}$ (time in miliseconds)

From Tables 5.1-5.4 we can see that the difference between Newton's and Halley's method is insignificant. The reason behind this is the characteristic of the tested polynomials. For the smaller degrees (i.e. 34-67) VCA method (Algorithm 7) provides similar results to VAS method (Algorithm 6). However as the degree of polynomials rises, the difference between VCA and VAS starts to increase. Sturm's method proved to be the slowest one, by a large margin.

The computation time of VCA and Sturm's method could be further improved by providing a smaller search interval. Let us examine what happens if we restrict the search interval to $(-20, 40)$, which still contains all previous roots.

Table 5.5 shows that providing sufficient bounds for VCA can improve the computation time to the point it's significantly better than VAS. That's because VAS is not affected by the change of the search interval, as it computes the bounds itself every time (see Alg. 6).

On the other hand, if the search interval is small enough, VCA can finish much faster, as the search interval can serve as an input to VCA (see Alg. 7).

Sturm's method proved to be very inefficient in any case, since evaluation of up to 100 polynomials in every iteration (see Alg. 5) takes too much time.

Thus for finding roots without any restrictions on search interval, VAS is the fastest solution for root finding. In comparison, when locating the roots within an interval, VCA can provide faster solution.

	VCA+N	VCA+H	VAS+N	Sturm+N
Minimum	97	103	167	616
Median	456	489	985	4609
Maximum	1024	1057	1303	6862
Average	441	465	773	3871

Table 5.5: Polynomials with degree 34-67, precision $\varepsilon = 10^{-50}$ (time in miliseconds), interval $(-20, 40)$

	VCA+N	VCA+H	VAS+N	Sturm+N
Minimum	1495	1562	327	2844
Median	1690	1753	719	13392
Maximum	1978	2316	1320	22682
Average	1724	1785	763	12836

Table 5.6: Polynomials with degree 68-101, precision $\varepsilon = 10^{-50}$ (time in miliseconds), interval $(-20, 40)$

6 Conclusion

In this thesis, we presented, implemented and compared several methods and algorithms for root approximation and the isolation of roots. First, we presented three algorithms for approximation of a single root and described each of them in detail. Then, we presented four methods for finding all roots of the polynomial, either by factoring the polynomial or isolating the roots. Later, we implemented all of the methods in Java. Finally, we concluded experiments and compared the efficiency of the algorithms. Since the synthesis implemented in fdCTMC PRISM provides small enough interval, VCA method can be implemented as a default algorithm. Thus we accomplished the goal of this thesis.

Future work For future work, the implemented methods can be further optimized, e.g. by computing tighter and faster upper and lower bounds of the positive roots, as well as some of the operations performed on polynomials. There are also many other methods for finding the roots of polynomials that could be implemented and compared, such as root finding through the eigenvalue decomposition. Furthermore, the methods could be implemented in a faster programming language such as C++.

Bibliography

1. DESCARTES, R. *La Géométrie*. 1637.
2. KWIATKOWSKA, M.; NORMAN, G.; PARKER, D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In: GOPALAKRISHNAN, G.; QADEER, S. (eds.). *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer, 2011, vol. 6806, pp. 585–591. LNCS.
3. KORENCIAK, Lubos; KUCERA, Antonín; REHÁK, Vojtech. Efficient Timeout Synthesis in Fixed-Delay CTMC Using Policy Iteration. In: *24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016, London, United Kingdom, September 19-21, 2016*. IEEE Computer Society, 2016, pp. 367–372. ISBN 978-1-5090-3432-1. Available from DOI: `10.1109/MASCOTS.2016.34`.
4. LEUNG, K. T.; MOK, I. A. C.; SUEN, S. N. *Polynomials and Equations*. Hong Kong: Hong Kong University Press, 1992. ISBN 962-209-271-3.
5. ROSICKÝ, J. *Algebra*. 4. vyd. Brno: Masarykova univerzita, 2007. ISBN 978-80-210-2964-4.
6. BINMORE, K. G. *Mathematical Analysis: A Straightforward Approach*. Cambridge: Cambridge University Press, 1997. ISBN 9780521288828.
7. MILLER, K. *Polynomials are continuous functions*. Berkeley: The University of California, Berkeley, 2014. Available also from: `https://math.berkeley.edu/~kmill/math1afa2014/poly.pdf`.
8. ATKINSON, K. E. *An Introduction to Numerical Analysis*. 2nd ed. Iowa City: John Wiley & Sons, 1989. ISBN 0-471-62489-6.
9. JEFFREYS, H.; JEFFREYS, B. *Methods of Mathematical Physics*. 3rd ed. Cambridge: Cambridge University Press, 2000. ISBN 0521664020.
10. SCAVO, T. R.; THOO, J. B. On the Geometry of Halley's Method. *The American Mathematical Monthly*. 1995, vol. 102, no. 5, pp. 417–426. ISSN 00029890, 19300972. ISSN 00029890, 19300972. Available also from: `http://www.jstor.org/stable/2975033`.

11. MATHEWS, J. H. *Theory and Proof for Halley's Method*. 2003. Available also from: `http://mathfaculty.fullerton.edu/mathews/n2003/Halley'sMethodProof.html`.
12. ALEFELD, G. On the Convergence of Halley's Method. *The American Mathematical Monthly*. 1981, vol. 88, no. 7, pp. 530–536. ISSN 00029890, 19300972. ISSN 00029890, 19300972. Available also from: `http://www.jstor.org/stable/2321760`.
13. CAUCHY, A. L. *Analyse Algebrique*. Paris: Jacques Gabay, 1989. ISBN 2-87647-053-5.
14. FAN, L. A Generalization of Synthetic Division and A General Theorem of Division of Polynomials. *Mathematical Medle*. 2003, vol. 30, no. 1, pp. 30–37. ISSN 0217-2976. Available also from: `https://eprints.soton.ac.uk/168861/1/FLH_article_on_polynomial_division.pdf`.
15. STURM, J. C. F. Mémoire sur la résolution des équations numériques. *Bulletin des Sciences de Férussac*. 1829, vol. 11, pp. 419–425.
16. VIGKLAS, P. S. *Upper bounds on the values of the positive roots of polynomials*. Volos, 2010. Available also from: `https://www.e-ce.uth.gr/cced/wp-content/uploads/formidable/phd_thesis_vigklas.pdf`. PhD thesis. University of Thessaly.
17. BARTLETT, P. *Finding All the Roots: Sturm's Theorem*. 2013. Available also from: `http://web.math.ucsb.edu/~padraic/mathcamp_2013/root_find_alg/Mathcamp_2013_Root-Finding_Algorithms_Day_2.pdf`.
18. VINCENT, A. J. H. Mémoire sur la résolution des équations numériques. *Mémoires de la Société Royale des Sciences, de l' Agriculture et des Arts, de Lille*. 1834, pp. 1–34. Available also from: `http://gallica.bnf.fr/ark:/12148/bpt6k57787134/f4.image.r=Agence%20Rol.langEN`.
19. VINCENT, A. J. H. Sur la résolution des équations numériques. *Journal de Mathématiques Pures et Appliquées*. 1834, vol. 1, pp. 341–372.
20. AKRITAS, A. G. *Vincent's theorem in algebraic manipulation*. Raleigh, 1978. PhD thesis. North Carolina State University.

21. ALESINA, A.; GALUZZI, M. Vincent's Theorem from a Modern Point of View. *Rendiconti del Circolo Matematico di Palermo, Serie II*. 2000, no. 64, pp. 179–191. Available also from: `http://inf-server.inf.uth.gr/~akritas/Alessina_Galuzzi_b.pdf`.

22. AKRITAS, A. G. Vincent's Theorem from a Modern Point of View. *Journal of Mathematical Sciences*. 2010, vol. 168, no. 3, pp. 309–325. Available also from: `https://link.springer.com/article/10.1007/s10958-010-9982-1`.

23. AKRITAS, A. G.; STRZEBONSKI, A. W. Vincent's Theorem from a Modern Point of View. *Nonlinear Analysis: Modelling and Control*. 2005, vol. 10, no. 4, pp. 297–304. Available also from: `http://www.lana.lt/journal/19/Akritas.pdf`.

24. SHARMA, V. Complexity of real root isolation using continued fractions. *Theoretical Computer Science*. 2008, vol. 409, pp. 292–310. Available also from: `https://people.mpi-inf.mpg.de/~mehlhorn/ftp/VikramContinuedFractions.pdf`.

25. ROUILLIERA, F.; ZIMMERMANNB, P. Efficient isolation of polynomial's real roots. *Journal of Computational and Applied Mathematics*. 2004, vol. 162, no. 1, pp. 33–50. Available also from: `http://www.sciencedirect.com/science/article/pii/S0377042703007271`.

26. BUDAN, F. D. *Nouvelle méthode pour la résolution des équations numériques*. 2nd ed. Paris, 1807.

27. KOJIMA, T. On the limits of the roots of an algebraic equation. *Tohoku Mathematical Journal*. 1917, vol. 11, pp. 119–127. Available also from: `https://www.jstage.jst.go.jp/article/tmj1911/11/0/11_0_119/_pdf`.

7 Appendix

The source code and Javadoc documentation can be found in the attachment.