T.R.
GEBZE TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

IMPLEMENTATION AND COMPARISON OF METHODS FOR REAL ROOT ISOLATION OF UNIVARIATE POLYNOMIALS

HÜSEYİN SABRİ ALİEFENDİOĞLU
A THESIS SUBMITTED FOR THE DEGREE OF
MASTER OF SCIENCE
DEPARTMENT OF COMPUTER ENGINEERING

GEBZE
2023

T.R.

GEBZE TECHNICAL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

IMPLEMENTATION AND COMPARISON OF METHODS FOR REAL ROOT ISOLATION OF UNIVARIATE POLYNOMIALS

HÜSEYİN SABRİ ALİEFENDİOĞLU

A THESIS SUBMITTED FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER ENGINEERING

THESIS SUPERVISOR

Assist. Prof. ZAFEIRAKIS ZAFEIRAKOPOULOS

GEBZE

2023

T.C.
GEBZE TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

TEK DEĞİŞKENLİ POLİNOMLARDA REEL KÖK İZOLASYON YÖNTEMLERİNİN UYGULANMASI VE KARŞILAŞTIRMASI

HÜSEYİN SABRİ ALİEFENDİOĞLU

YÜKSEK LİSANS TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

DANIŞMANI
DR. ÖĞR. ÜYESI ZAFEIRAKIS ZAFEIRAKOPOULOS

GEBZE
2023

GEBZE TEKNİK ÜNİVERSİTESİ	**YÜKSEK LİSANS JÜRİ ONAY FORMU**

GTÜ Fen Bilimleri Enstitüsü Yönetim Kurulu'nun 07/02/2023 tarih ve 2023/11 sayılı kararıyla oluşturulan jüri tarafından 15/03/2023 tarihinde tez savunma sınavı yapılan Hüseyin Sabri Aliefendioğlu'nun tez çalışması Bilgisayar Mühendisliği Anabilim Dalında YÜKSEK LİSANS tezi olarak kabul edilmiştir.

JÜRİ

ÜYE (TEZ DANIŞMANI)	: Dr. Öğr. Üyesi Zafeirakis ZAFEIRAKOPOULOS
ÜYE	: Doç. Dr. Mehmet GÖKTÜRK
ÜYE	: Doç. Dr. Kağan KURŞUNGÖZ

ONAY

Gebze Teknik Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun ...../...../…… tarih ve ……/…..... sayılı kararı.

İMZA/MÜHÜR

SUMMARY

There are mainly two methods to isolate the real roots of a polynomial, i.e., to find separate intervals for all real roots so that each interval has only one real root. These are Sturm's method and method of continued fractions. The performance of the methods depends on the degree, root separation, and coefficient sizes of polynomial. The selection of the bounds also affects the performance. In this thesis we study how do these parameters affect the performance of the real root isolation methods. It is known that performance of continued fractions method is better in many cases.

Key Words: Continued fractions, real root isolation, algebraic number, univariate polynomial.

ÖZET

Sturm ve sürekli kesir yöntemi (CF) tek değişkenli polinomlarda reel kök izolasyonu (her kök için ayrık aralıklar bulmak) için yaygın olarak kullanılan iki yöntemdir. Yöntemlerin performansı polinom derecesine, köklerin yakınlığına ve katsayı büyüklüğüne bağlıdır. Kök sınırları seçimi de performansı etkilemektedir. Bu çalışmada bu parametrelerin kök izolasyon yöntemi performansını nasıl etkilediği incelenmiştir. CF yönteminin genel olarak daha iyi performans gösterdiği bilinmektedir.

Anahtar Kelimeler: Sürekli kesirler, reel kök izolasyonu, cebirsel sayılar, tek değişkenli polinomlar.

ACKNOWLEDGEMENTS

My sincere gratitude goes to my supervisor Zafeirakis Zafeirakopoulos for his patience, support and encouragement. I am also thankful to all my lecturers in the Department of Computer Engineering at Gebze Technical University.

Lastly I extend my thanks to my family for their support. I am dedicating this thesis to my beloved son Uzay.

TABLE of CONTENTS

	Page
SUMMARY	iv
ÖZET	v
ACKNOWLEDMENTS	vi
TABLE of CONTENTS	vii
LIST of ABBREVIATIONS and ACRONYMS	ix
LIST of FIGURES	x
LIST of TABLES	xi
1. INTRODUCTION	1
1.1. Exact Arithmetic	1
1.2. Real Root Isolation	1
1.3. Fast Algorithms	2
2. PREREQUISITES	3
2.1. Algebraic Structures	3
2.1.1. Groups	3
2.1.2. Rings	3
2.1.3. Fields	4
2.2. Polynomials	4
2.3. Recursion	7
2.4. Lazy Evaluation	8
2.5. Complexity Analysis	8
3. REPRESENTATION OF NUMBERS	9
3.1. Integer	9
3.2. Rational	9
3.3. Algebraic Numbers	9
3.4. Continued Fractions	10
4. REAL ROOT ISOLATION	15
4.1. Bounds on the Roots	15
4.2. Subdivision	15
4.2.1. DRS	15

4.2.2. Budan 16
4.2.3. Sturm 16
4.3. CF Method 18
5. EXACT ARITHMETIC 21
5.1. Integer Arithmetic 21
5.2. Rational Arithmetic 21
5.3. Algebraic Number Arithmetic 21
6. BENCHMARKS 26
6.1. Literature Survey 26
6.2. Implementation 27
6.3. Benchmarks 27
6.4. Analysis 42
7. CONCLUSION 45

REFERENCES 46
BIOGRAPHY 48
APPENDICES 49

LIST OF ABBREVIATIONS AND ACRONYMS

Abbreviations and Acronyms		**Explanations**
ALU	:	Arithmetic logic unit
CF	:	Continued fraction
DRS	:	Descartes' rule of signs
GCD	:	Greatest common divisor
LCM	:	Least common Multiple

LIST OF FIGURES

Figure No:		Page
1.1:	Pseudocode for Float Arithmetic.	1
1.2:	Root Isolation.	2
2.1:	Polynomial Long Division.	6
2.2:	Remainder of Polynomial Division.	6
2.3:	P(x-3) Transformation.	7
2.4:	Pseudocode for Factorial.	8
3.1:	Pseudocode for Rational Construction.	9
3.2:	Continued Fraction Construction.	11
4.1:	Counting Number of Sign Changes.	16
6.1:	Effect of Reduction of Polynomials.	32
6.2:	Number of Irrational Roots.	32
6.3:	Int Polynomials With Degree 12.	35
6.4:	Int Polynomials With Bit Size 150.	35
6.5:	Vincent Solving Polynomials With Degree 20.	36
6.6:	Vincent Solving Polynomials With Bit Size 200.	37
6.7:	Sage Solving Mignotte Polynomials.	37
6.8:	Cost of Sturm Sequence Preparation.	38
6.9:	Effect of Reduction of Polynomials.	39
6.10:	Effect of Converting to Integer (Rational Roots-Vincent).	40
6.11:	Effect of Bound Selection (Sturm).	41
6.12:	Effect of Bound Selection (CF).	41
6.13:	Coe-50 and Coe-200.	42
6.14:	Int-50 and Int-200.	42
6.15:	Irr-50 and Irr-200.	43
6.16:	Mig-50 and Mig-200.	43
6.17:	Mix-50 and Mix-200.	44
6.18:	Vincent-50 and Vincent-200.	44
6.19:	Sturm-50 and Sturm-200.	44

LIST OF TABLES

Table No:		Page
6.1:	Sample Polynomial of Degree 15 and Bit Size 100.	29
6.2:	Polynomials of Type irr, coe, mig.	30
6.3:	Polynomials of Type int, rat, mix.	31
6.4:	Types of Polynomial Roots.	31
6.5:	Roots of Sample Polynomial.	34
6.6:	Execution Results.	38

1. INTRODUCTION

1.1. Exact Arithmetic

Computers generally represent numbers as integer or floating point numbers within a fixed bit size. They are typically 32, 64 or 128 bits. Computers can make arithmetic operations on these types of numbers in one clock cycle of the processor. But both types of representation have some limitations. First of all they both have a capacity. When exceeded, they generate overflow error or the result will not be the correct one. For floating point numbers even if the maximum capacity is not exceeded, result of an arithmetic operation may not be correct. For example Julia code in Figure 1.1 gives the result of 1.0 which is not correct.

```
a= Float64(1.0) - 1e-20
print(a)
```

Figure 1.1: Pseudocode for Float Arithmetic

For applications like symbolic computation this is not tolerable. To eliminate these limitations many modern languages support big number types. Bit sizes of these numbers are limited by the memory of the hardware. Arithmetic on these type of numbers are called as exact arithmetic or arbitrary precision arithmetic. Algorithmic complexity of addition (and subtraction) is O(n), multiplication (and division) is $O(n^2)$. There are some algorithms like Karatsuba, Toom-Cook which give better performance.

1.2. Real Root Isolation

Real roots of polynomials with integer (or rational) coefficients up to degree 4 can be found. Evariste Galois proved that polynomials of degree 5 or more are not solvable by radicals. Although finding roots exactly may not be possible, determining whether there is a root or not inside an interval is possible. The process of finding disjoint intervals which include only one real root with rational endpoints

is defined as real root isolation. Intervals for all roots have to be found. Endpoints can be infinity. Two successive intervals may have a common endpoint. Real root isolation problem may have many different solutions. A sample root isolation can be seen in Figure 1.2.

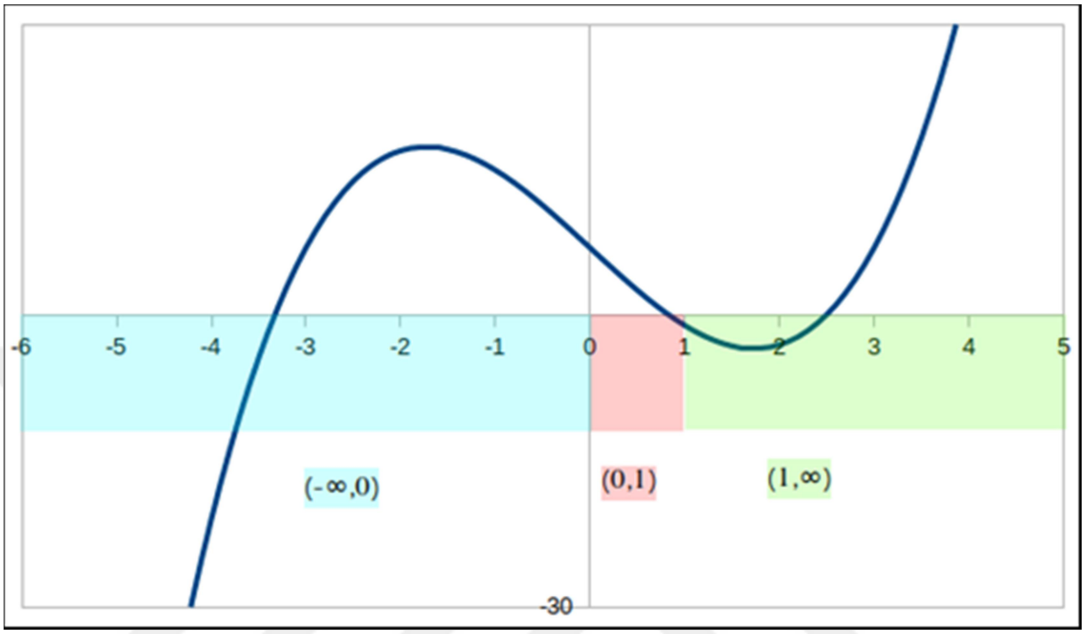


Figure 1.2: Root Isolation

There are two commonly used methods for root isolation: Sturm's method and continued fractions method.

1.3. Fast Algorithms

Performance of an algorithm highly depends on the sub operations (number multiplication or matrix multiplication) that are called frequently. There are algorithms which beat naive algorithms performance. These are called fast algorithms. Karatsuba, Schonhage–Strassen, Toom-Cook are well known fast multiplication algorithms.

2. PREREQUISITES

2.1. Algebraic Structures

2.1.1. Groups

A set G with a binary operation * satisfying following properties is called a group and denoted as (G,*)

- Associativity: $(a * b) * c = a * (b * c)$ $\forall$ a, b, c $\in$ S
- Identity element: $\exists e \in S$ such that $a * e = e * a = a$ $\forall$ a $\in$ S
  e is unique and called as identity element.
- Inverse element: $\exists b \in S$ such that $a * b = b * a = e$ $\forall$ a $\in$ S
  b is called as inverse of a and denoted as a^{-1}

Operation * has not to be commutative. For example set of 2x2 matrices forms a non-commutative group. If the operation is commutative, the group is called as commutative or abelian group.

Addition on integer numbers $(\mathbb{Z},+)$ forms a group whereas multiplication does not because each element does not have inverses. $(\mathbb{Q}-\{0\}, \times)$, $(\mathbb{R}, +)$, $(\mathbb{R}-\{0\}, \times)$, $(\mathbb{C}-\{0\}, \times)$ are samples of groups.

2.1.2. Rings

A set R with two binary operations +, × satisfying following properties is called a ring and denoted as (R, +, ×).

- (R, +) is a commutative group
- $(a\times b)\times c = a\times(b\times c)$ $\forall$a, b, c $\in$ S (Associative under ×)
- $(a+b)\times c = a\times c+b\times c$ and $a\times(b+c) = a\times b+a\times c$ $\forall$a, b, c $\in$ S (Distributive law)

$(\mathbb{Z},+,\times)$, $(\mathbb{Q},+,\times)$, $(\mathbb{C},+,\times)$ are samples of rings.

2.1.3. Fields

A set F with two binary operations +, × satisfying following properties is called a field and denoted as (F, +, ×).

- (F, +, ×) is a ring
- (F-{0}, ×) is an abelian group.

($\mathbb{Q}$, +, ×), ($\mathbb{R}$, +, ×), ($\mathbb{C}$, +, ×) are samples of fields

2.2. Polynomials

A function with following form is defined as a polynomial or to be more specific a univariate polynomial.

$$P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n = \sum_{k=1}^{n} a_k x^k \qquad (2.1)$$

n is defined as the degree of polynomial for $a_n \neq 0$. In most general case coefficients (a_0, a_1... a_n) are complex numbers. Polynomials with addition and multiplication operator form a ring.

A value x_0 which satisfies $P(x_0) = 0$ is called a root of the polynomial. Fundamental Theorem of Algebra states that a polynomial of degree n has n complex roots counted with multiplicities.

Most of the real world problems are related with the real roots of polynomials with rational coefficients. Multiplying a polynomial by a constant integer does not affect the roots of the polynomial. So an equivalent polynomial with integer coefficients can be obtained by multiplying by the greatest common divisor of all of denominators of the coefficients.

$$P(x) = \sum_{k=1}^{n} \frac{b_k x^k}{c_k} \tag{2.2}$$

$$M = \gcd(c_0, c_1, \ldots c_n) \tag{2.3}$$

$$M.P(x) = \sum_{k=1}^{n} d_k x^k \tag{2.4}$$

Complex Conjugate Root Theorem states that, if a complex number is a root of a polynomial with real coefficients, then the complex conjugate of it is also a root. So there must be an even number of complex roots. We can conclude that a polynomial with odd degree must have at least one real root.

A polynomial can be divided by a polynomial with smaller degree and quotient and remainder polynomials obtained.

If A is a polynomial of degree n and B is a polynomial of degree m where n≤m

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \tag{2.5}$$

$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_m x^m \tag{2.6}$$

Polynomial A can be written as:

$$A(x) = B(x)Q(x) + R(x) \tag{2.7}$$

Q is a polynomial of degree q≤n-m and R is polynomial of degree r<m

The quotient and remainder polynomials can be found by polynomial long division. In Figure 2.1 polynomial long division method is shown on a sample.

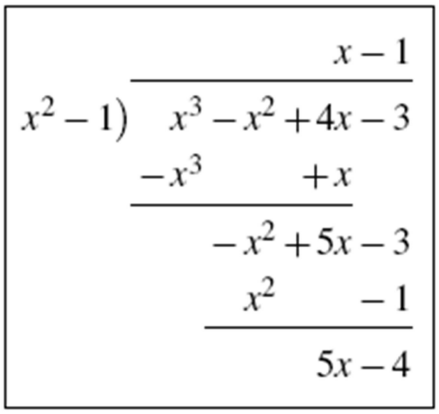


Figure 2.1: Polynomial Long Division

To find remainder of division by a polynomial of degree 1, synthetic division can be used. In Figure 2.2 obtaining remainder by polynomial long division and synthetic division methods are shown on a sample.

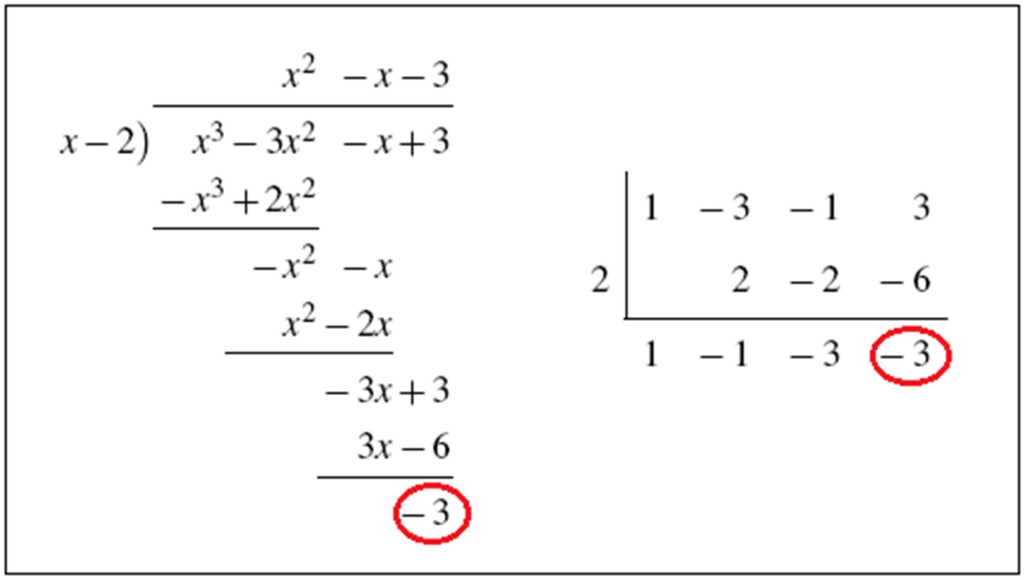


Figure 2.2: Remainder of Polynomial Division

P(x+a) transformation is obtained by applying previous method recursively. In Figure 2.3 transformation method is shown on a sample.

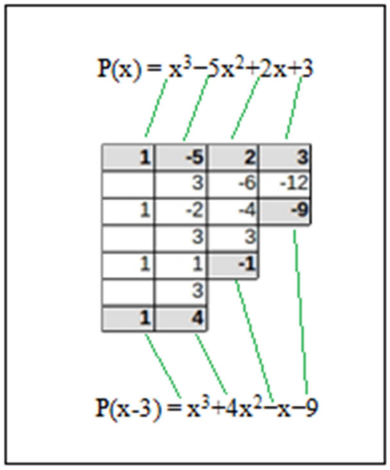


Figure 2.3: P(x-3) Transformation

P(1/x) transformation is done by simply reversing the coefficient order.

2.3. Recursion

Definition of an object in terms of itself is called recursion. Functions, series can be defined recursively. For example Fibonacci numbers is defined as follows.

$$ai = ai-2+ai-1 \quad (2.8)$$

To find an element in Fibonacci sequence, two initial elements namely a_0 and a_1 has to be given. It is called as base case.

Recursion is a very widely used method in computation problems. Functions calling theirselves are called recursive. Like mathematical recursion, there has to be a base case to terminate the recursion process. Julia code snippet in Figure 2.4 calculates factorial recursively.

```
function fact(n:: UInt16):: UInt16
    if (n<=1)
        return 1
    end
    return n * fact(n-1)
end
```

Figure 2.4: Pseudocode for Factorial

2.4. Lazy Evaluation

Lazy evaluation is a functional programming strategy in which evaluation is done when needed. It is also called as call-by-need. Lazy evaluation improves performance programs since it does not execute the code that does not affect the final result. By lazy evaluation it is possible to represent infinite structures like series or continued fractions.

2.5. Complexity Analysis

There can be many different solutions to a computation problem. Simply comparing two algorithms with respect to how much seconds they take may be misleading. First of all an algorithm which seems better may not be the best for large problem sizes. And this type of comparisons depend on the hardware that the algorithms running on.

Asymptotic notations are used for systematically compare algorithms. There are three types of asymptotic notations. These are Big O, Omega and Theta notations. Big O notation represents an upper bound for execution time of an algorithm with respect to problem size. For example for an algorithm of $O(n^2)$, if you double problem size (n), execution time will not increase more than 4 times. Memory used by an algorithm can also be represented like time and called as space complexity. Omega notation defines the lower bound. Theta notation represents both upper and lower bound.

3. REPRESENTATION OF NUMBERS

3.1. Integer

We use decimal system (base 10) to represent numbers and to make any operation on them. Whereas computers hold information as bits. A combination of bits (bit string) can represent anything (a character, a number etc.). To represent numbers, programming languages use certain bit sizes (32, 64 etc.). The decimal value which corresponds to a binary value can be calculated as following example.

$$(1101)_2 = 1\times23+1\times22+0\times21+1\times20 = (13)_{10} \quad (3.1)$$

3.2. Rational

Rational numbers can be represented as numerator and denominator as integer numbers. Modern programming languages support rational number representation. It is possible to represent infinity by setting denominator 0. While construction rational numbers are simplified. For example Julia code in Figure 3.1 gives output of 2//3.

```
print(Rational(4,6))
```

Figure 3.1: Pseudocode for Rational Construction

3.3. Algebraic Numbers

Numbers which is a root of a rational coefficient polynomial are called algebraic numbers. 3/4, √2, 1+i are algebraic numbers. Numbers like e, π cannot be expressed as a root of a polynomial. They are called transcendental numbers. Algebraic numbers can be represented by their minimal polynomial. Minimal polynomial is the polynomial whose root equals to the algebraic number and its degree is minimum.

The degree of an algebraic number is defined as the degree of its minimal polynomial.

3.4. Continued Fractions

An expression of following form is defined as continued fraction:

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{\ddots}}} \tag{3.2}$$

If all numerators of the continued fractions are 1 and all denominators except first one are positive, they are called as simple. General form of simple continued fractions is given below.

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{\ddots}}} \tag{3.3}$$

Continued fractions can be finite or infinite. Finite continued fractions correspond to rationals. Rationals can be converted to simple continued fraction by using Euclid's greatest common divisor method. Figure 3.2 shows the steps to obtain the continued fractions.

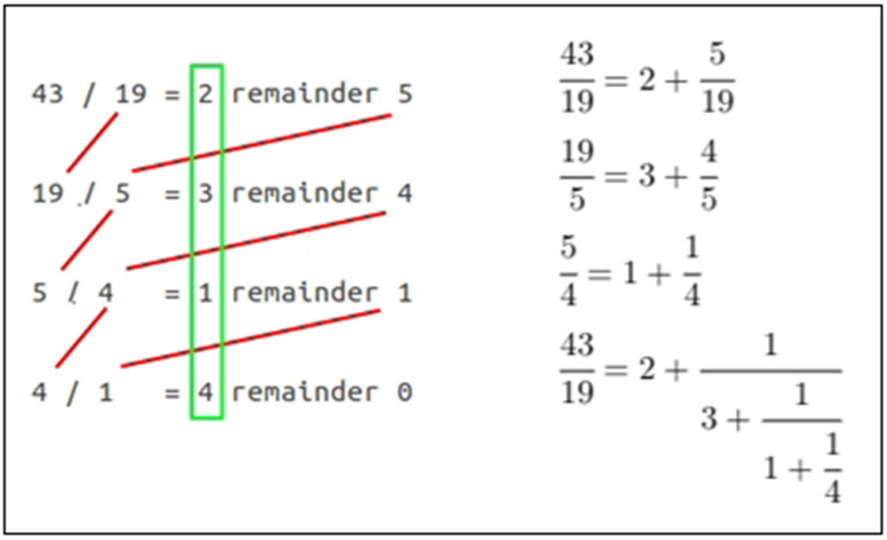


Figure 3.2: Continued Fraction Construction

Irrational numbers can be expressed as continued fractions but they yield infinite continued fractions (continued fractions with infinitely many terms).

$$\sqrt[3]{2} = 1 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{5 + \cfrac{1}{\ddots}}}} \tag{3.4}$$

By truncating the terms after some point one can obtain a rational approximation of the continued fraction. This is called convergent of continued fraction. This is the best rational approximation with such large denominator.

$$conv_n = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_{n-1} + \cfrac{1}{a_n}}}}} \tag{3.5}$$

Convergents can be calculated from bottom to top as follows.

$$conv_n = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_{n-2} + \cfrac{a_n}{a_n a_{n-1} + 1}}}}} \tag{3.6}$$

$$conv_n = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_{n-3} + \cfrac{a_n a_{n-1} + 1}{a_n a_{n-1} a_{n-2} + a_n + a_{n-2}}}}}} = \tag{3.7}$$

Or they can be calculated from top to bottom recursively as follows.

$$conv_0 = a_0 = \frac{a_0}{1} = \frac{p_0}{q_0} \tag{3.8}$$

$$conv_1 = \frac{a_0 a_1 + 1}{a_1} = \frac{p_1}{q_1} \tag{3.9}$$

$$conv_2 = \frac{a_0 a_1 a_2 + a_2 + a_0}{a_1 a_2 + 1} = \frac{p_2}{q_2} = \frac{p_1 a_2 + p_0}{q_1 a_2 + q_0} \tag{3.10}$$

$$conv_n = \frac{p_{n-1} a_n + p_{n-2}}{q_{n-1} a_n + q_{n-2}} \tag{3.11}$$

Simple continued fractions can be abbreviated as follows.

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots}}} = [a_0, a_1, a_2, \ldots] \tag{3.12}$$

4. REAL ROOT ISOLATION

4.1. Bounds on the Roots

A number which is larger than all the roots of a polynomial is defined as an upper bound. A number which is smaller than all the positive roots are defined as lower bound. If M is an upper bound for a polynomial f(x), then 1/M is a lower bound for the reverse polynomial $f_{rev}(x) = x^n f(1/x)$. So a method for finding upper bounds can be used for finding for lower bounds also.

If α is a root of polynomial $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$, then

$$|\alpha| < 1 + \max_{0 \le j \le n-1}\left(\left|\frac{a_j}{a_n}\right|\right) \text{ (Cauchy Theorem I)} \quad (4.1)$$

If α is a root of polynomial $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$, then

$$|\alpha| < \max_{0 \le j \le n}\left(n.\left|\frac{a_j}{a_n}\right|\right)^{\frac{1}{n-j}} \text{ (Cauchy Theorem II)} \quad (4.2)$$

If α_+ is a root of polynomial $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$, then

$$\alpha_+ < 2 \max_{1 \le j \le n;\ a_{n-j} \le 0}\left(-\frac{a_{n-j}}{a_n}\right)^{\frac{1}{j}} \text{ (Kioustelidis Theorem[9])} \quad (4.3)$$

4.2. Subdivision

4.2.1. DRS

Let $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$, be a polynomial of degree n (a_n non-zero), sign variations over coefficients of polynomial P is defined as δ(p(x)): sign changes in $[a_0, a_1 \ldots a_m]$.

$$\#\ \text{positive real roots of } P(x) \leq \delta(p(x)) - 2k \text{ where } k \in \mathbb{N} \quad \text{(Descartes Rule of Signs)} \tag{4.4}$$

In Figure 4.1 number of sign changes over coefficients of a polynomial is found as 2. So there are either 2 positive real roots or no positive real root.

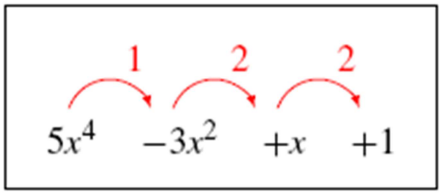


Figure 4.1: Counting Number of Sign Changes

For two cases number of real roots can be determined exactly. If δ p(x) is zero, there is no positive root. If δ p(x) is 1, there is only one positive root.

Descartes rule of signs defines a bound for only number of positive roots. But a similar bound can be found easily by transformed polynomial P(-x).

4.2.2. Budan

Budan's theorem can be considered as a generalization of Descartes rule of signs. Budan's theorem defines a bound for an interval.

Let P(x) be a polynomial $P_m(x)$ and $\delta P_m(x)$ are defined as $P_m(x) = P(x+m)$ and $\delta P_m(x)$ defined as sign changes over the coefficients of $P_m(x)$.

$$\#\ \text{real roots in } (a, b] \leq \delta P_a(x) - \delta P_b(x) - 2k \text{ where } k \in \mathbb{N} \quad \text{(Budan Theorem)} \tag{4.5}$$

4.2.3. Sturm

Sturm is the first mathematician who discovered a theorem which exactly defines the number of real roots in an interval. Before then Descartes stated a

theorem which gives an upper bound for the number of positive real roots. Sturm first define sequence of polynomials of decreasing degrees with following properties corresponding to polynomial of degree m.

- P_0 has no multiple roots
- P_1 has constant sign around root of P_0
- P_m non-zero and has constant sign
- if $P_i(a) = 0$ then $P_{i+1}(a) = -P_{i-1}(a)$ and they are non-zero

Constructing the Sturm's sequence can be done as follows.

- $P_0(x)$: original polynomial
- $P_1(x)$: derivative of original polynomial ($P'_0(x)$)
- $P_i(x)$: remainder of $P_{i-2}(x)/P_{i-1}(x)$ for $i > 1$

Let P(x) be a polynomial and $[P_0(x), P_1(x)... P_m(x)]$ be the Sturm's sequence of it. If ω(b): sign changes in $[P_0(b), P_1(b)... P_m(b)]$ then

$$\#\text{ real roots in } (a, b] = \omega(a) - \omega(b) \text{ (Sturm Theorem)} \tag{4.6}$$

Sturm sequence of polynomial $P(x) = x^5+5x^4-20x^2-10x+2$ are obtained as follows.

$$P0(x) = P(x) = x5+5x4-20x2-10x+2 \tag{4.7}$$

$$P1(x) = P'(x) = 5x4+20x3-40x-10 \tag{4.8}$$

$$P2(x) = -\text{Remainder}(P0(x), P1(x)) = 4x3+12x2-4 \tag{4.9}$$

$$P3(x) = -\text{Remainder}(P1(x), P2(x)) = 15x2+35x+5 \quad (4.10)$$

$$P4(x) = -\text{Remainder}(P2(x), P3(x)) = 68/9\,x+ 44/9 \quad (4.11)$$

$$P5(x) = -\text{Remainder}(P3(x), P4(x)) = 3285/289 \quad (4.12)$$

Counts of sign changes at different points are as follows.

- Sign at x= −∞: -+-+-+ sign changes: 5
- Sign at x= -1 -++–+ sign changes: 3
- Sign at x= 0 +–+++ sign changes: 2
- Sign at x= 1 –++++ sign changes: 1
- Sign at x= ∞: ++++++ sign changes: 0

So by Sturm's theorem the following conclusions can be drawn.

- between −∞ and -1 there are 2 real roots.
- between -1 and 0 there is 1 real root.
- between 0 and 1 there is 1 real root.
- between 1 and ∞ there is 1 real root.

4.3. CF Method

Let P(x) be a square free polynomial. If transformations of the form x← a_0+ 1/x are applied successively, the resulting polynomial has either no sign change or one (Vincent Theorem).

Isolated root obtained by Vincent's theorem can be represented as following continued fraction.

$$s(x) = a_0 + \cfrac{1}{a_1 + \cfrac{1}{. + \cfrac{.}{. + \cfrac{.}{. + \cfrac{.}{a_{n-1} + \cfrac{1}{a_n + x}}}}}}} \tag{4.13}$$

The first step in continued fractions (CF) method is to find the number of sign changes. If it is zero or one, algorithm terminates. If it is more than one, P(x+a) or P(1/x) transformations is done and process continues recursively. The selection of the constant a depends on the algorithm. The most trivial selection is 1. But it yields exponential computing time.

If in this step number of sign changes equals to one, there is a positive root in transformed polynomial. This root corresponds to a different value and interval in original polynomial. Corresponding interval of original polynomial can be found by substituting 0 and infinity to x. These are nth and (n-1)th convergents of the continued fraction.

$$s(0) = conv_{n-1} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{. + \cfrac{.}{. + \cfrac{.}{. + \cfrac{.}{a_{n-1}}}}}} \tag{4.14}$$

$$s(\infty) = conv_n = a_0 + \cfrac{1}{a_1 + \cfrac{1}{. + \cfrac{.}{. + \cfrac{.}{. + \cfrac{.}{a_n}}}}} \tag{4.15}$$

Details of Vincent method is explained by an example in Appendix B.

5. EXACT ARITHMETIC

5.1. Integer Arithmetic

Integer arithmetic is performed in bit level in ALU (Arithmetic Logical Unit) of processors. A combination of logic gates takes input bits and outputs result bits (and possible carry bit). A full adder for example takes two input bits and one carry bit, adds them up and generates one result bit and one carry bit. If carry output bit is given to next digit as carry input, multiple bit addition can be done. Modern processors can make integer arithmetic within one clock cycle. For numbers which exceeds ALU's bit size limit standard number types cannot be used. Many modern programming languages support big number types. Very large numbers can be represented by these types. algorithmic complexity of addition and subtraction is O(n), multiplication and division is $O(n^2)$.

5.2. Rational Arithmetic

Rational number addition and subtraction may require, multiplication, division and gcd operations. If numerator and denominator are represented as big integer type, complexity of all operations are bounded by $O(n^2)$.

5.3. Algebraic Number Arithmetic

Algebraic numbers can be represented as continued fractions. Addition, subtraction, multiplication and division on continued fractions can be done by using Gosper's method [10]. Method uses in each step one term from operands. For two operands (x, y) a general result (z) is defined.

$$z = \frac{a + xa_x + ya_y + xya_{xy}}{b + xb_x + yb_y + xyb_{xy}} \tag{5.1}$$

Or simply represent in matrix form as follows.

$$\begin{bmatrix} a & a_x & a_y & a_{xy} \\ b & b_x & b_y & b_{xy} \end{bmatrix} \tag{5.2}$$

For each operation the elements of the matrix is determined. Addition matrix is as follows.

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{5.3}$$

Subtraction matrix is as follows.

$$\begin{bmatrix} 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{5.4}$$

Multiplication matrix is as follows.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{5.5}$$

Division matrix is as follows.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{5.6}$$

In each step one term from operands are taken or one term of result is extracted. If a term from p is taken from x and substituted p+ 1/x, corresponding matrix is as follows.

$$\begin{bmatrix} a_x & (a + pa_x) & a_{xy} & (a_y + pa_{xy}) \\ b_x & (b + pb_x) & b_{xy} & (b_y + pb_{xy}) \end{bmatrix} \tag{5.7}$$

If a term from q is taken from y and substituted q+ 1/y, corresponding matrix is as follows.

$$\begin{bmatrix} a_y & a_{xy} & (a + qa_y) & (a_x + qa_{xy}) \\ b_y & b_{xy} & (b + qb_y) & (b_x + qb_{xy}) \end{bmatrix} \quad (5.8)$$

If a term r is extracted from z, corresponding matrix is as follows.

$$\begin{bmatrix} b & b_x & b_y & b_{xy} \\ (a - rb) & (a_x - rb_x) & (a_y - rb_y) & (a_{xy} - rb_{xy}) \end{bmatrix} \quad (5.9)$$

If $r = \left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{a_x}{b_x} \right\rfloor = \left\lfloor \frac{a_y}{b_y} \right\rfloor = \left\lfloor \frac{a_{xy}}{b_{xy}} \right\rfloor$, r is extracted from Z.

If $\left| \frac{a}{b} - \frac{a_x}{b_x} \right| > \left| \frac{a}{b} - \frac{a_y}{b_y} \right|$, next term is taken from X, otherwise from Y. If all terms are taken, next term is considered as infinite.

Continued fraction addition (7/3+2/11) is shown step by step as follows.

$$x = \frac{7}{3} = [2,3] \quad (5.10)$$

$$y = \frac{2}{11} = [0,5,2] \quad (5.11)$$

$$z = x + y \quad (5.12)$$

Matrix representation of addition is as follows.

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad (5.13)$$

$\left| \frac{0}{1} - \frac{1}{0} \right| \not> \left| \frac{0}{1} - \frac{1}{0} \right| \Rightarrow$ Substitute 0+1/y for y

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (5.14)$$

$\left|\frac{1}{0}-\frac{0}{0}\right| \not> \left|\frac{1}{0}-\frac{0}{1}\right| \Rightarrow$ Substitute 5+1/y for y

$$\begin{bmatrix} 0 & 1 & 1 & 5 \\ 1 & 0 & 5 & 0 \end{bmatrix} \quad (5.15)$$

$\left|\frac{0}{1}-\frac{1}{0}\right| > \left|\frac{0}{1}-\frac{1}{5}\right| \Rightarrow$ Substitute 2+1/x for x

$$\begin{bmatrix} 1 & 2 & 5 & 11 \\ 0 & 1 & 0 & 5 \end{bmatrix} \quad (5.16)$$

$\left|\frac{1}{0}-\frac{2}{1}\right| > \left|\frac{1}{0}-\frac{5}{0}\right| \Rightarrow$ Substitute 3+1/x for x

$$\begin{bmatrix} 2 & 7 & 11 & 38 \\ 1 & 3 & 5 & 15 \end{bmatrix} \quad (5.17)$$

$r = \left|\frac{2}{1}\right| = \left|\frac{7}{3}\right| = \left|\frac{11}{5}\right| = \left|\frac{38}{15}\right| = 2 \Rightarrow$ Extract 2 and substitute 1/(z-2)

$$\begin{bmatrix} 1 & 3 & 5 & 15 \\ 0 & 1 & 1 & 8 \end{bmatrix} \quad (5.18)$$

$\left|\frac{1}{0}-\frac{3}{1}\right| \not> \left|\frac{1}{0}-\frac{5}{1}\right| \Rightarrow$ Substitute 2+1/y for y

$$\begin{bmatrix} 5 & 15 & 11 & 33 \\ 1 & 8 & 2 & 17 \end{bmatrix} \quad (5.19)$$

$\left|\frac{5}{1}-\frac{15}{8}\right| > \left|\frac{5}{1}-\frac{11}{2}\right| \Rightarrow$ Substitute ∞ for x

$$\begin{bmatrix} 15 & 15 & 33 & 33 \\ 8 & 8 & 17 & 17 \end{bmatrix} \quad (5.20)$$

$r = \left|\frac{15}{8}\right| = \left|\frac{15}{8}\right| = \left|\frac{33}{17}\right| = \left|\frac{33}{17}\right| = 1 \Rightarrow$ Extract 1 and substitute 1/(z-1)

$$\begin{bmatrix} 8 & 8 & 17 & 17 \\ 7 & 7 & 16 & 16 \end{bmatrix} \tag{5.21}$$

$r = \left\lfloor \frac{8}{7} \right\rfloor = \left\lfloor \frac{8}{7} \right\rfloor = \left\lfloor \frac{17}{16} \right\rfloor = \left\lfloor \frac{17}{16} \right\rfloor = 1$ ⇒ Extract 1 and substitute 1/(z-1)

$$\begin{bmatrix} 7 & 7 & 16 & 16 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{5.22}$$

$\left\lfloor \frac{7}{1} - \frac{7}{1} \right\rfloor \not> \left\lfloor \frac{7}{1} - \frac{16}{1} \right\rfloor$ ⇒ Substitute ∞ for y

$$\begin{bmatrix} 16 & 16 & 16 & 16 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{5.23}$$

$r = \left\lfloor \frac{16}{1} \right\rfloor = \left\lfloor \frac{16}{1} \right\rfloor = \left\lfloor \frac{16}{1} \right\rfloor = \left\lfloor \frac{16}{1} \right\rfloor = 16$ ⇒ Extract 16 and substitute 1/(z-16)

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.24}$$

So continued fraction representation of z is [2, 1, 1, 16], which equals to 83/33

6. BENCHMARKS

6.1. Literature Survey

Lee E. Heindel [6] made first computer implementation Sturm's method. In this study in addition to standard root isolation, a preprocessing for eliminating possible multiple roots and refinement of intervals to an arbitrary error level are also done. Empirical computing times grow much slower than theoretical limits. Heindel concluded that it may be because of characteristics of Chebyshev polynomials or loose bounds used in deriving theoretical limits.

In 1976 Collins and Loos [4] developed a root isolation method using derivatives and Rolle's theorem. In this method roots of the derivatives of the polynomial are also isolated. It has a better performance than Sturm's method because derivative polynomials have smaller coefficients than Sturm's sequence.

In 1976 Collins and Akritas [3] developed the continued fraction method. The study is frequently cited (168 citation in ResearchGate.net). Developed algorithm is named as Uspenky's algorithm because Uspensky was the one who reminded Vincent's theorem in his book "Theory of Equations" in 1948 [13]. In 1978 Akritas [1] improves previous bound. In 1986 Akritas [2] showed that actually Uspensky doubled the computing time of Vincent's method and the method should be named as Vincent's method. The method has been improved by Johnson [8], Rouillier and Zimmermann [11]. In 2009 experimental comparison for different methods is published [7]. Computation of lower bounds has The worst-case complexity for both Sturm and continued fraction methods is $O_B(d^4\tau^2)$ for a polynomial of degree d and coefficient bit size of τ. Pan developed a numerical method with the best worst-case complexity of $O_B(d^2\tau)$ [10].

Bill Gosper [5] developed a method for arithmetic on continued fraction. Method takes a term from the operands (continued fractions) in each step and generates result as another continued fraction. S. Lang and H. Trotter [12] extracted the first 1000 terms of the continued fractions of cube root of 2, 3, 4, 5, 7.

6.2. Implementation

Implementation is done in Julia. Julia Polynomials, Big Integer and Rationals are used. For derivative, count of sign changes of coefficients, (x+a), (1/x), and (−x) transformations, lower and upper bounds a helper file called PolynomialUtility is created.

Vincent algorithm takes a polynomial as input. It first finds intervals for positive roots. It uses a recursive function to isolate intervals. In each recursive step number of sign changes is calculated. If it is zero it returns without adding any intervals. If number of sign changes is one it adds corresponding isolating interval and returns. If it is more than one it makes (x+lowerbound) transformation. It makes (1/1+x) and (1+x) transformations and calls itself. These transformations correspond to [0,1), [1,∞)] intervals. By making transformations a continued fraction is generated. By this, continued fraction intervals corresponding to original polynomial are extracted. Julia code of the Vincent algorithm is given in Appendix C.

Sturm algorithm takes polynomial as input. It creates Sturm sequence of polynomials. After taking Sturm sequence number of sign changes at −∞,0,∞. If number of sign changes is zero or one algorithm terminates for that interval. If it is more than one, for intervals [−upperbound, 0), (0, upperbound] bisection function called. Bisection function splits the interval to two parts in each time in order to get sign changes of 1. Julia code of the Sturm algorithm is given in Appendix D.

While implementing Vincent's algorithm a continued fraction is constructed. Instead of returning isolated interval, algorithm is continued lazily to represent algebraic numbers as continued fraction. It is used in Gosper's method for arithmetic on continued fractions. Julia code of the continued fraction arithmetic algorithm is given in Appendix E.

Cube roots of 2, 3, 4, 5, 7 are extracted by Lang, Trotter [12]. In Appendix F first 1000 elements of continued fraction of $\sqrt[3]{7}$ are given.

6.3. Benchmarks

For benchmark tests 6 types of polynomials are used. These are polynomials only have integer roots (int), polynomials only have rational roots (rat), polynomials

have irrational and complex roots (irr), polynomials have integer, rational, irrational and complex roots (mix), polynomials with random coefficients (coe) and Mignotte polynomials (mig). These types are abbreviated as int, rat, irr, mix, coe and mig respectively. All polynomials are square free. Their constant coefficients are non-zero. Except for mix and coe they are monic (leading coefficient 1).

• (int) Polynomials that have only integer roots are in this category. They are generated by random unique integer roots. Number of integer roots is equal to the degree of the polynomial. Coefficients of polynomials are integers.

• (rat) Polynomials that have strictly rational (a/b where $b \neq 1$) roots are in this category. They are generated by random unique rational roots. Number of integer roots is equal to the degree of the polynomial. Coefficients of polynomials are rationals.

• (irr) Polynomials that has strictly irrational ($a^{(1/n)} \notin \mathbb{Q}$, $2 \leq n \leq 5$) roots. Since remaining roots are complex, number of real roots is less than the degree of the polynomial. Coefficients of polynomials are integers.

• (mix) Polynomials that have integer, rational, irrational roots. Each group has ≈ 20% of the degree of the polynomial. Coefficients of polynomials are rationals. coe Polynomials that random rational coefficients. They may or may not have real roots. Number of real roots in this category is less than others. All real roots are irrational. Coefficients of polynomials are rationals.

• (mig) Mignotte polynomials are of the form $x^d-2(kx-1)^2$. They are generated by assigning a random integer to k value. Since in each category there are 100 of polynomials it is probable to have the same polynomial with same degree and bit size. Mignotte polynomials have two real roots which are close to each other. Although real root count is much small and there are some zero coefficients, root separation makes them hard to be isolated. Coefficients of this type polynomials are integers.

Sample polynomials of degree 15 and bit size 100 from each type are given below.

Table 6.1: Sample Polynomial of Degree 15 and Bit Size 100.

int	168105841536550470688751616000 - 369402063672325473004419072000*x - 187147874981329084874471424 0*x^2 + 456965292095996043932604 16*x^3 + 5100081193973724855705 60*x^4 - 156442863510293749362 00*x^5 - 438825742725050208 12*x^6 + 226929488452128118 2*x^7 + 321993980988291*x^8 - 165608334181943*x^9 + 123924003843*x^10 +6398836821*x^11 - 5480511*x^12 - 125877*x^13 + 69*x^14 + x^15
rat	-3920800//330429 + 7604985154//76329099*x + 2698965273299//4579745940*x^2 - 579056892613601//274784756400*x^3 - 17900093267500253//8243542692000*x^4 + 13373035143552259 3//9892251230400*x^5 - 2695340432445099 3//8992955664000*x^6 - 154985905206709//72737141400*x^7 + 4323788938932527//329741707680*x^8 - 3902816085878197//494612561520*x^9 - 31580731332769537//98922512304000*x^10 + 244512725706541//104128960320*x^11 - 3139790649535139//27478475640 00*x^12 + 69696080715817//274784756400*x^13 - 41086543//1542002*x^14 + x^15
irr	19846437182724197507134933548 00 - 95746273569048905323160*x^3 -47240239587248980777590*x^4 - 73779177543208242130800*x^5 + 1130106067433320*x^6 + 2279037220305403*x^7 + 3559370002642360*x^8 + 1756157032923390*x^9 - 26899781*x^10 - 42011720*x^11 - 84723263*x^12 + x^15
mix	162557959317435//67592 + 10210577805175007 7//11490640*x + 59089177034417871//3590825*x^2 + 43728562092451448619//574532000*x^3 + 32097691695288456152 1//1436330000*x^4 + 35489019352693446477 9//1436330000*x^5 + 39904800662932962123//5745320000*x^6 - 380560577708612823//9656000*x^7 - 15157128530349102957//1436330000*x^8 - 13962352437779164 07//1436330000*x^9 + 31516238962074796 61//5745320000*x^10 + 59242787852123304 7//5745320000*x^11 + 3166033188809 1//35908250*x^12 - 167892113467//4046000*x^13 - 229004//2975*x^14+ 4//1*x^15

Table 6.1: Continued.

coe	-2215216656224540//2189183348088033 - 1498588283240262//56386903344025*x - 1158068801775327//1148806526427665*x^2 - 87637607117485//525308538196214*x^3 - 1528323526451531//2047045853020991*x^4 - 429473103654531//1276992301168214*x^5 - 136146622157339//230649834002703*x^6 - 224882597783011//1500220280273835*x^7 - 281132158286015//360597060033717*x^8 - 1053514952710982//1609943428887509*x^9 + 709884772328801//2076424493622403*x^10 - 858487633776624//202279953823729*x^11 + 1361403712354043//1820191382084150*x^12 - 8773192478323//9179998604187*x^13 - 143212872196054//1267612030819849*x^14 + 1893104697767401//540699347532964*x^15
mig	-2 + 3443657305678112*x - 14823469548687792137845194855 68*x^2 +x^15

For each type there are 25 clusters. Each cluster corresponds to a degree and a coefficient bit size. Selected degrees are 10, 12, 15, 20 and 30. Coefficient bit sizes are 50, 70, 100, 150 and 200. In each cluster there are 100 of polynomials. So for each type there are 2500 polynomials. Table 6.2 shows polynomial counts with respect to clusters for types irr, coe and mig.

Table 6.2 Polynomials of type irr, coe, mig.

		Bit size				
		50	**70**	**100**	**150**	**200**
Degree	**10**	100	100	100	100	100
	12	100	100	100	100	100
	15	100	100	100	100	100
	20	100	100	100	100	100
	30	100	100	100	100	100

For some types polynomials with low bit size coefficient and high degree cannot be generated. Table 6.3 shows which types are missing for which clusters. Table 6.4 shows root types, for each type totally.

Table 6.3: Polynomials of type int, rat, mix.

		Bit size				
		50	**70**	**100**	**150**	**200**
Degree	**10**					
	12					
	15					
	20	~~rat~~				
	30	~~rat, int, mix~~	~~rat, int~~	~~rat~~		

Table 6.4: Types of polynomial roots.

	Count of real roots			**Count of complex roots**	**Count of polynoms**	**Total degree of polynoms**
	integer	**rational**	**irrat.**			
Polynoms with integer roots	37500	0	0	0	2300	37500
Polynoms with rational roots	0	32500	0	0	2100	32500
Polynoms with irrational roots	0	0	20150	23350	2500	43500
Polynoms with mixed roots	7920	8402	8990	15188	2400	40500
Polynoms with random coefficients	0	0	6526	36974	2500	43500
Mignotte polynoms	9500			34000	2500	43500

The cause of tendency to decrease at degree 15 is investigated (Figure 6.1) and concluded that it is because of the reduction of number of roots at degree 15 (Figure 6.2). Since there are less roots it takes less to isolate them.

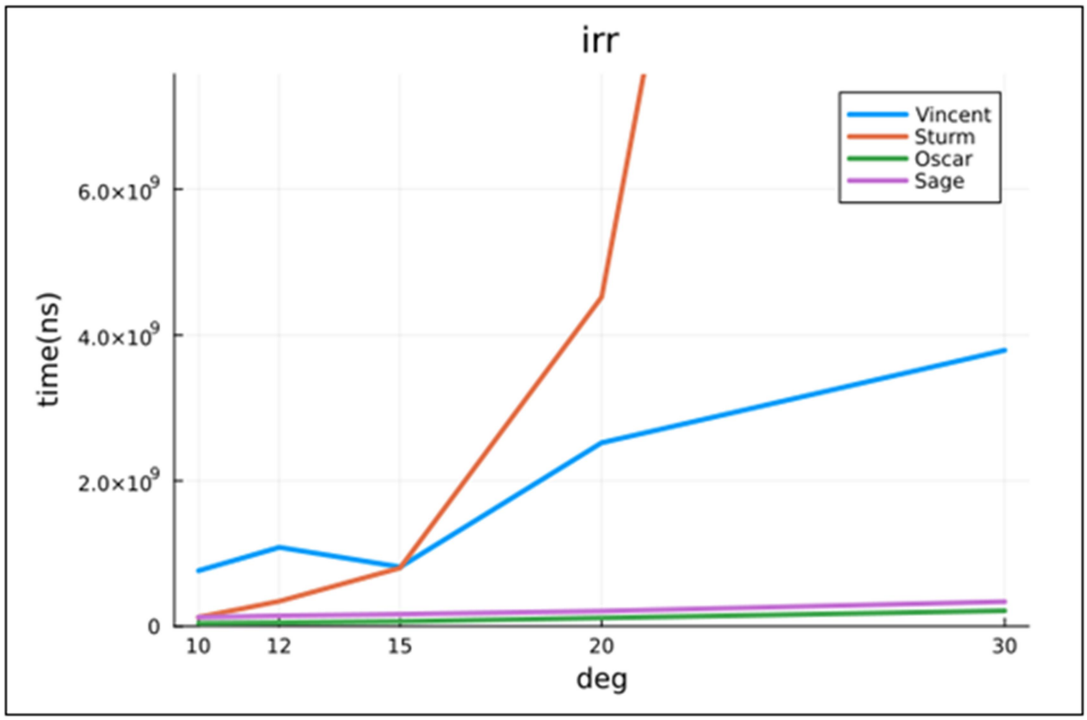


Figure 6.1: Effect of Reduction of Polynomials.

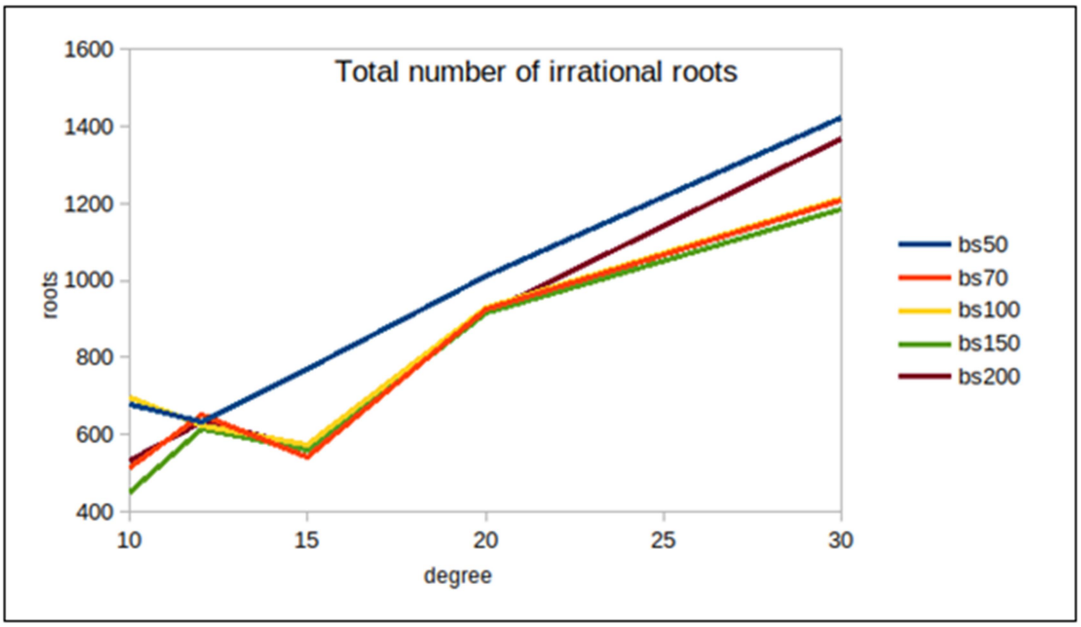


Figure 6.2: Number of Irrational Roots

For checking correctness and comparing the performance two libraries are used. Oscar (https://docs.oscar-system.org/dev/) is a computer algebra system which combines the capabilities of four cornerstone systems: GAP, Polymake, Antic and

Singular. In Oscar Polynomial Ring over integer or rational fields can be defined. Roots (including complex roots) can be found. It runs on Julia.

SageMath (https://www.sagemath.org/) is a free open-source mathematics software system. Main motivation is to create a viable free open source alternative to Magma, Maple, Mathematica and Matlab. SageMath library can be accessed by Python. Like Oscar Polynomial rings can be defined and real roots can found.

In benchmark results Vincent, Sturm, Oscar and SageMath are abbreviated as v, s, o and g respectively. Roots of Mignotte polynomials could not be found in defined time by Oscar and it is not evaluated in this category. SageMath generates error while execution of rat, mix and coe type. SageMath is evaluated for only int, irr and mig types.

Correctness of Vincent and Sturm methods are checked using Oscar library. All real roots found by Oscar are checked whether consistent to the intervals. Following are checked.

- Each root is in one internal.
- No extra intervals without a root.
- Intervals are disjoint.

Roots found by Oscar and intervals isolated by Vincent for a sample polynomial are in Table 6.5.

Table 6.5: Roots of Sample Polynomial

342989293524903187 5//288883370585876 - 425807336395697950687 5//88976078140449808*x + 106768323793762963562 5//38132604917335632*x^2 + 102661709274505696948450 25//1804434864688322106 24*x^3 -313187117218697790833 59//537034185919143484*x^4 + 453046450883925254607 3//537034185919143484*x^5 + 760314161919787829 57//14368807650010528*x^6 - 721469888674620779582 15//601478288229440702 08*x^7 - 125023059522069314149//1171710951096313056*x^8 + 412204048091660151 83//17350335237387712 56*x^9 + x^10		
Oscar	Vincent	Sturm
Root 4.75000 of 4x - 19	[4//1, 1//0]	[4, 6]
Root 3.80734 of 109x - 415	[3//1, 4//1]	[3, 4]
Root 2.51613 of 31x - 78	[2//1, 3//1]	[2, 3]
Root 1.60102 of 391x - 626	[1//1, 2//1]	[1//1,2//1]
Root 0.744986 of 349x - 260	[1//2, 1//1]	[1//2, 1//1]
Root 0.360632 of 696x - 251	[0//1, 1//2]	[0//1, 1//2]
Root -0.881874 of 491x + 433	[-1//1, 0//1]	[-3, 0]
Root -3.76033 of 121x + 455	[-4//1, -3//1]	[-6, -3]
Root -7.08791 of 91x + 645	[-8//1, -7//1]	[-12, -6]
Root -25.8077 of 26x + 671	[-1//0, -8//1]	[-48, -24]

The benchmark tests took place on a 64-bit Intel i7 with 16GB RAM memory, Ubuntu 22.04. Benchmark tests are carried for 100 polynomials in each cluster. Julia BenchmarkTools framework is used to systematically collect and compare test results. Each cluster is tested 5 times within 20 second time limit. BenchmarkTools framework controls each run and saves the time and memory results in benchmark suite which can be stored as JSON file.

Evaluation graphs are plotted by using Julia Plots library. Four main types of 2d graphs are prepared.

• Time (or memory) variation with respect to bit size for a fixed degree and polynomial type for each applicable method (Figure 6.3)

• Time (or memory) variation with respect to degree for a fixed bit size and polynomial type for each applicable method (Figure 6.4)

• Time (or memory) variation with respect to bit size for a fixed degree and method for each polynomial type (Figure 6.5)

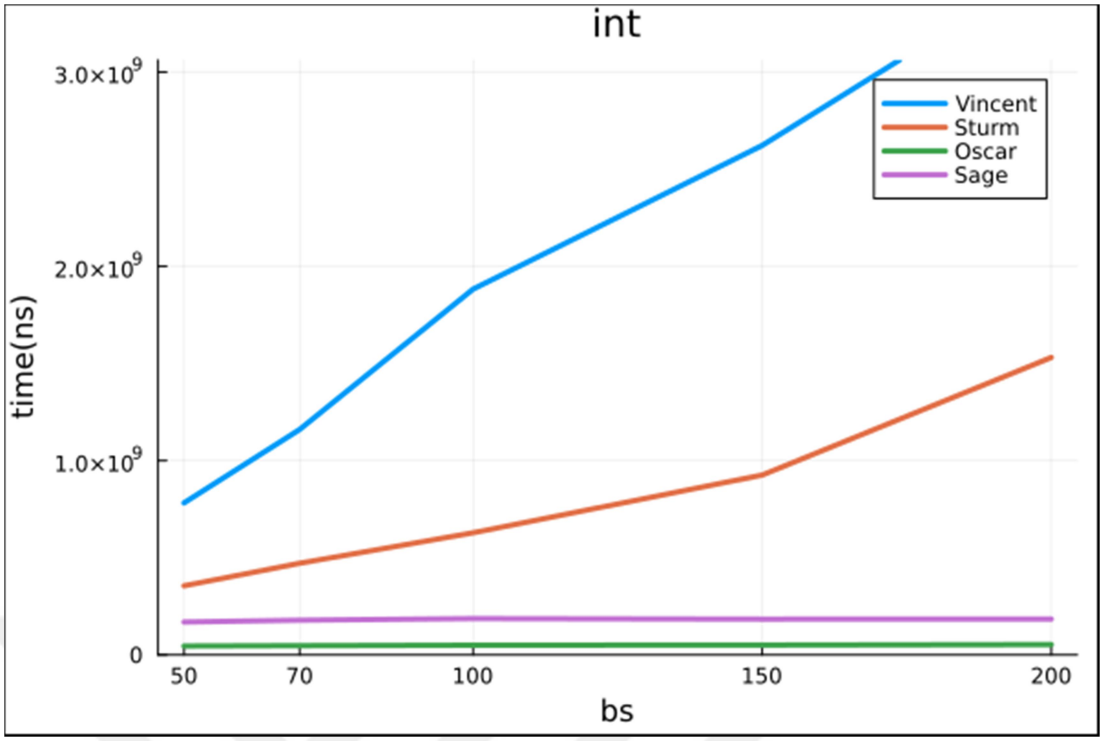


Figure 6.3: Int Polynomials with Degree 12.

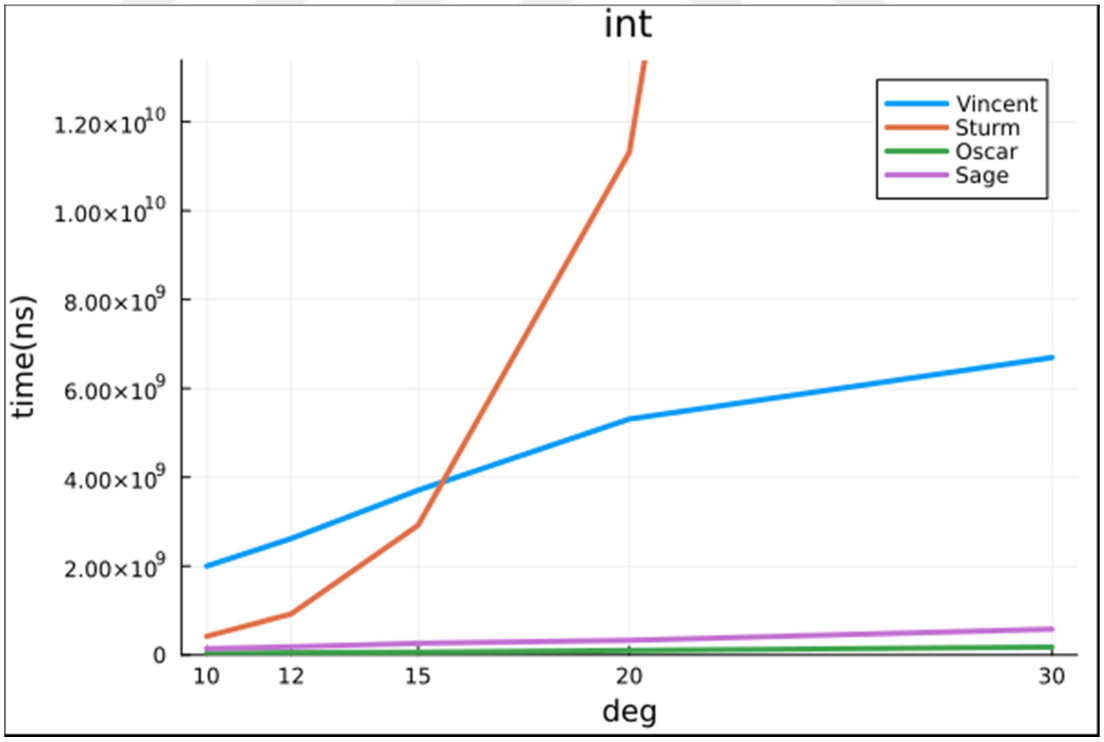


Figure 6.4: Int Polynomials with Bit Size 150.

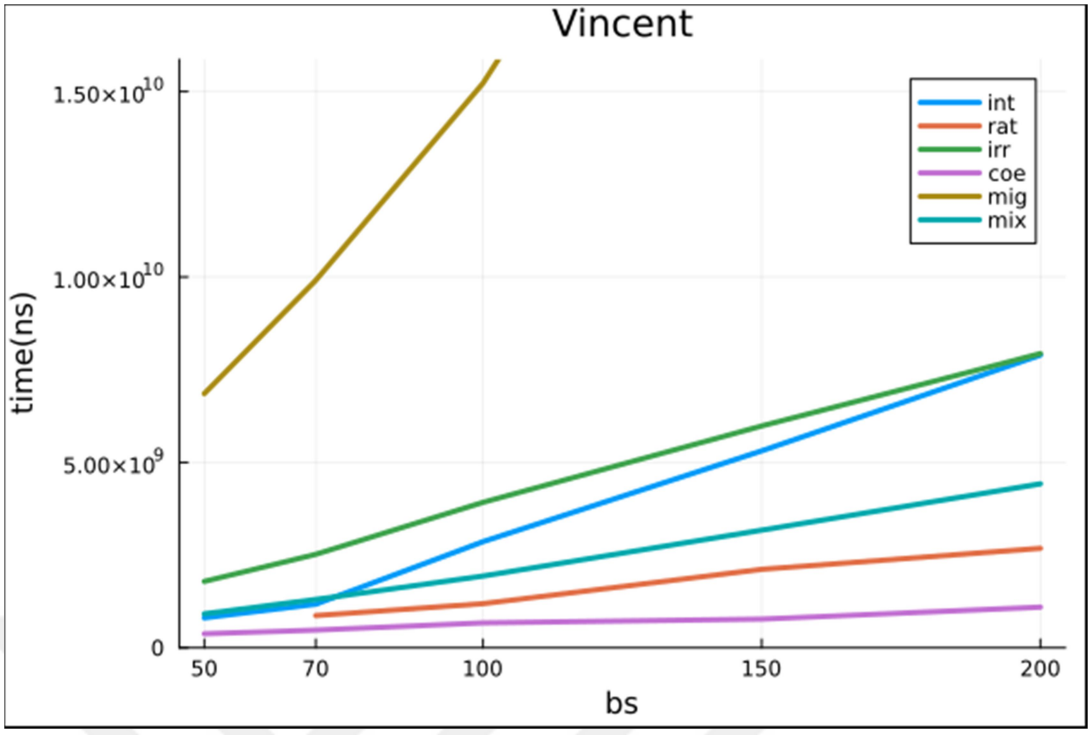


Figure 6.5: Vincent Solving Polynomials with Degree 20.

• Time (or memory) variation with respect to degree for a fixed bit size and method for each polynomial type (Figure 6.6)

One type of 3d graphs is prepared.

• Time (or memory) variation with respect to bit size and degree for a fixed polynomial type and method (Figure 6.7)

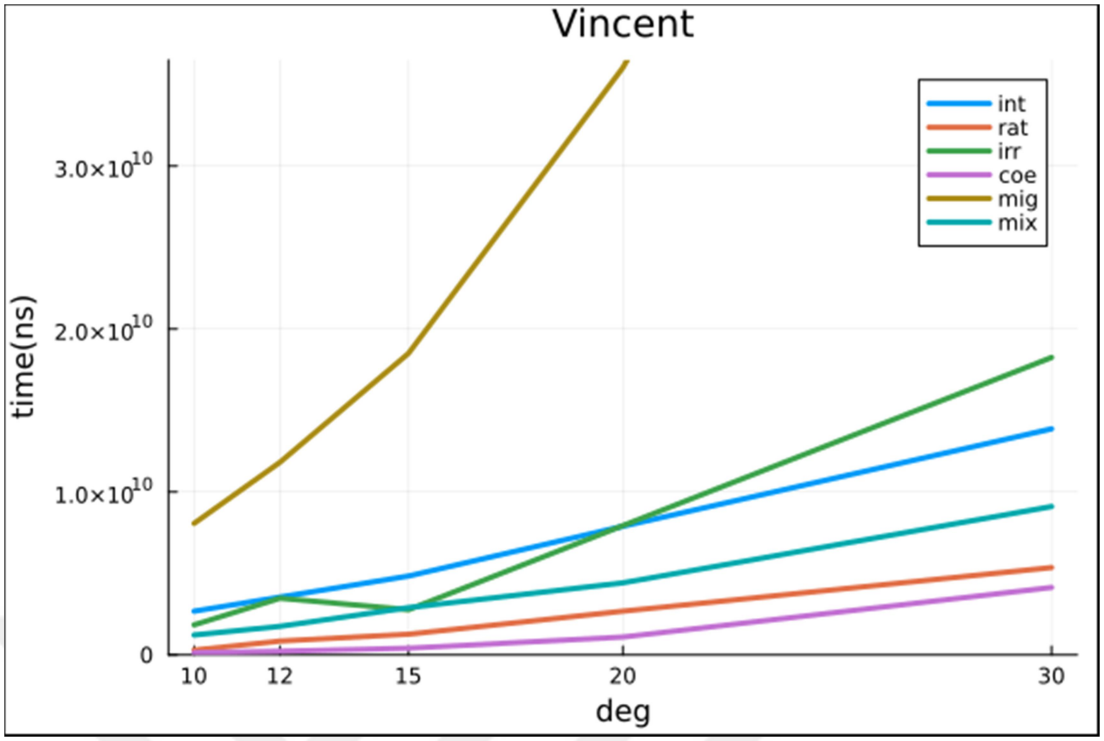


Figure 6.6: Vincent Solving Polynomials with Bit Size 200.

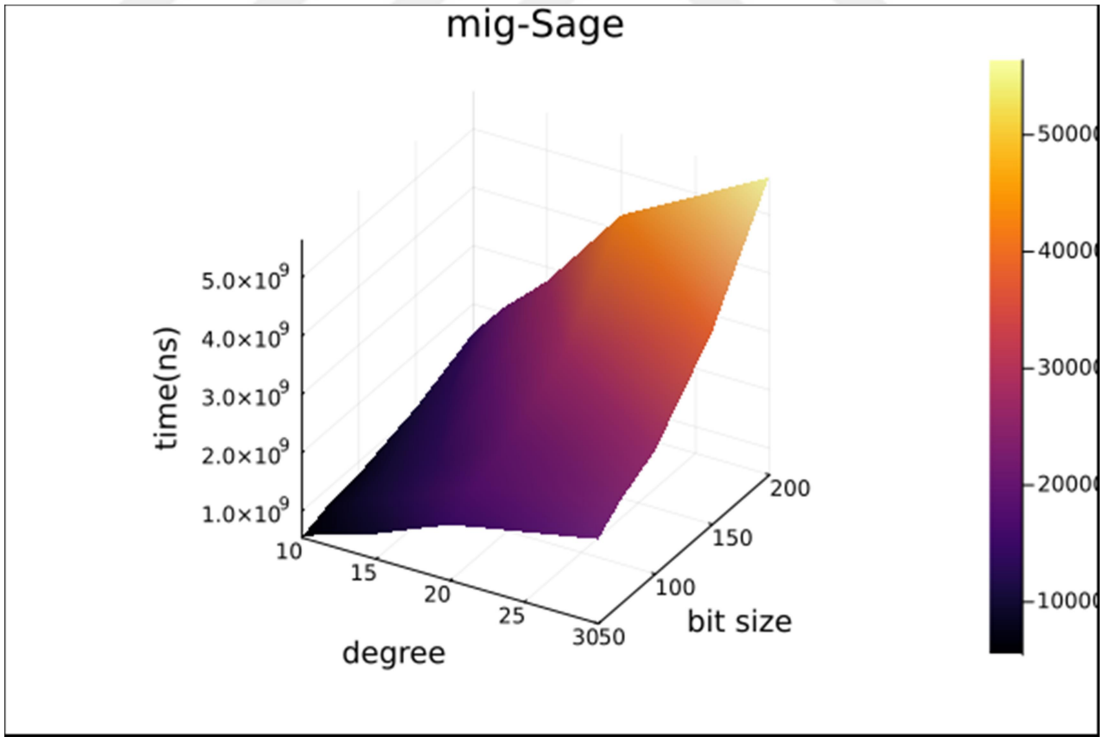


Figure 6.7: Sage Solving Mignotte Polynomials.

Vincent and Sturm algorithms can isolate intervals of polynomials of degree 30 and coefficient bit size of 200. Oscar and Sage cannot finish execution successfully for some types. Table 6.6 shows general results for each method.

Table 6.6: Execution Results.

	int	**rat**	**irr**	**mix**	**coe**	**mig**
Vincent						
Sturm						
Oscar						
Sage						

Sturm tends to increase by degree and bit sizes more. Preparation of Sturm sequence takes considerable time. Figure 6.8 shows the cost of sequence preparation.

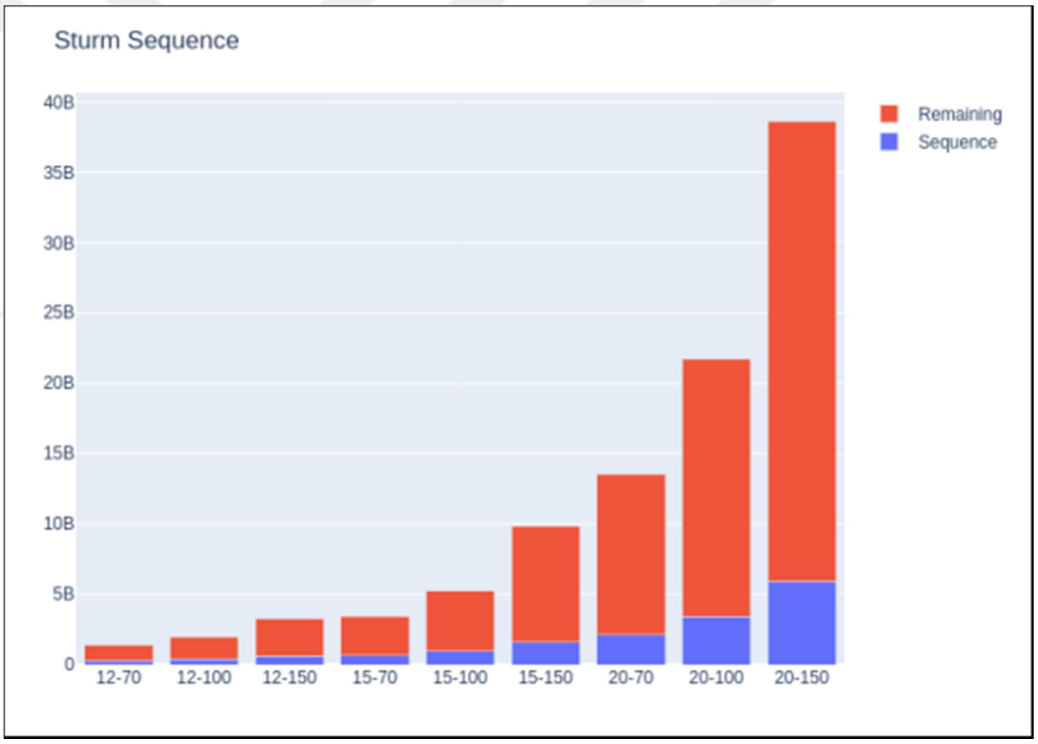


Figure 6.8: Cost of Sturm Sequence Preparation.

In Vincent implementation, if the root is found exactly (It can happen if the root is integer or rational) the polynomial can be reduced by dividing by (x−root). Its affect can be seen in Figure 6.9. After improvement performance improves considerably especially on int polynomials.

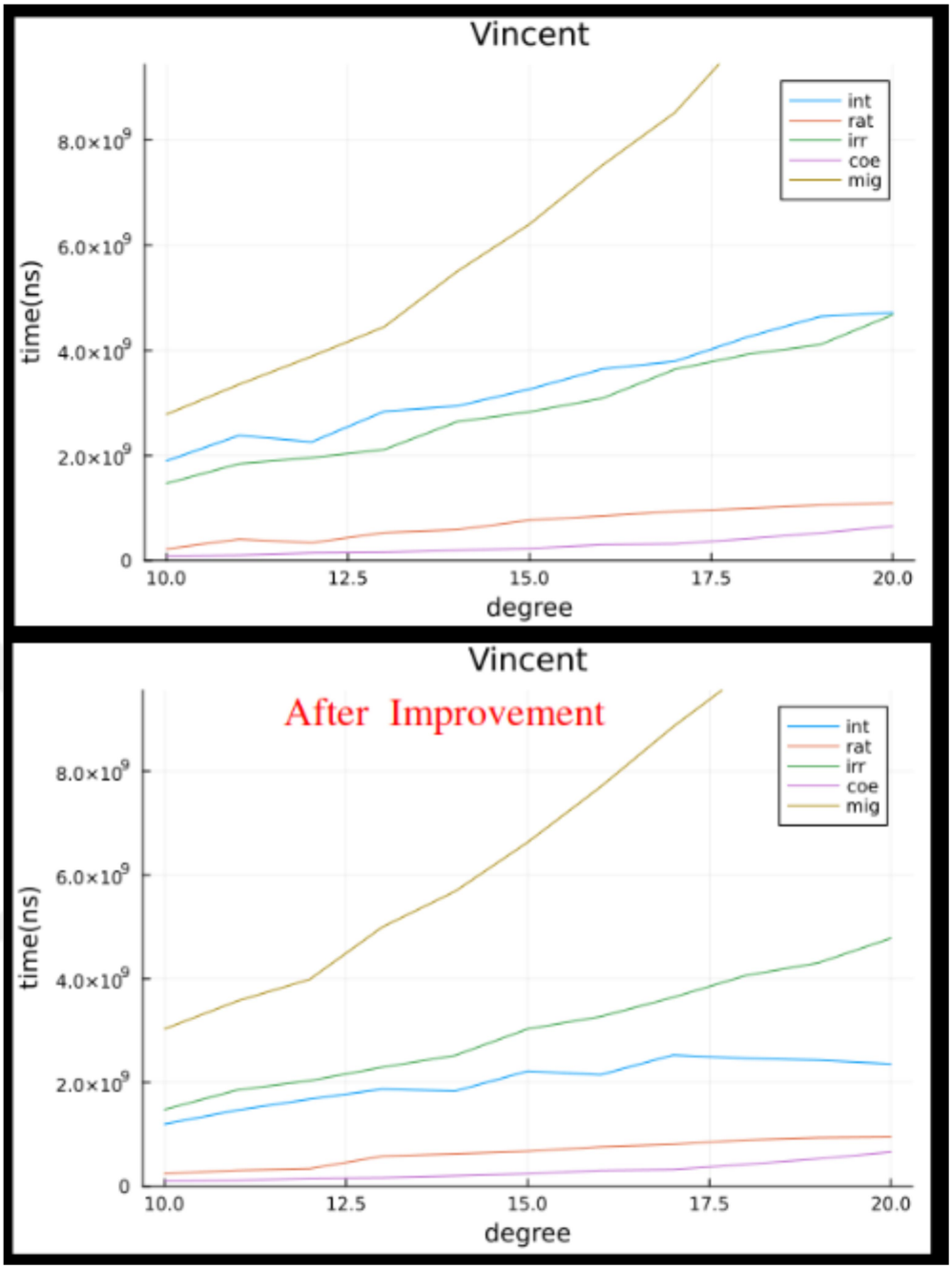


Figure 6.9: Effect of Reduction of Polynomials.

Multiplying polynomial by a constant, will not change the roots of the polynomial. If it is chosen as the least common multiple of the coefficients than resultant polynomial will have integer coefficients. For rational coefficient polynomials (rat, mix, coe) it is tested. But converting to integer and solving it totally takes more than original solution. Figure 6.10 shows both results together.



Figure 6.10: Effect of Converting to Integer (Rational Roots-Vincent).

Both Sturm and CF methods use polynomial root bounds. Sturm method only uses bounds in first step to get a finite end point. In each iteration step new end point are found by bisection. But CF method calculates lower bound in each iteration step. Cauchy bound is easier to calculate whereas Kioustelidis bound is closer to maximum of the roots. It can be seen that Sturm method does not depend on the type of bounds (Figure 6.11). For CF method Kioustelidis is much better especially for large bit sizes (Figure 6.12).

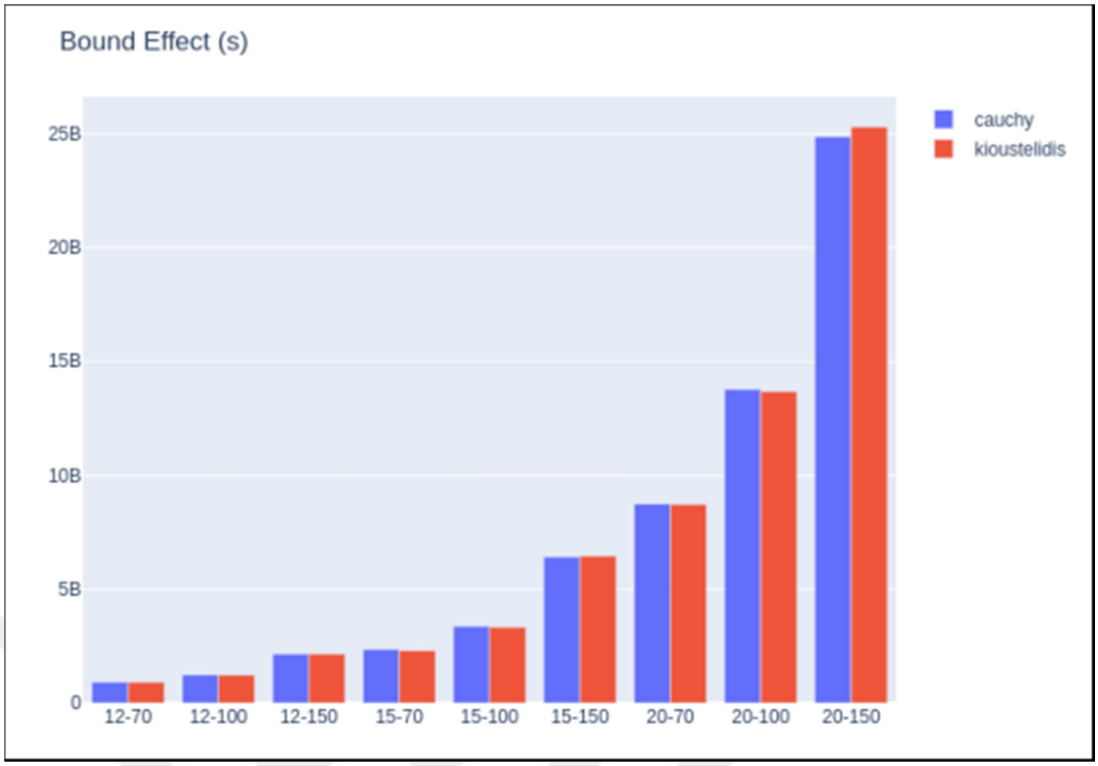


Figure 6.11: Effect of Bound Selection (Sturm).

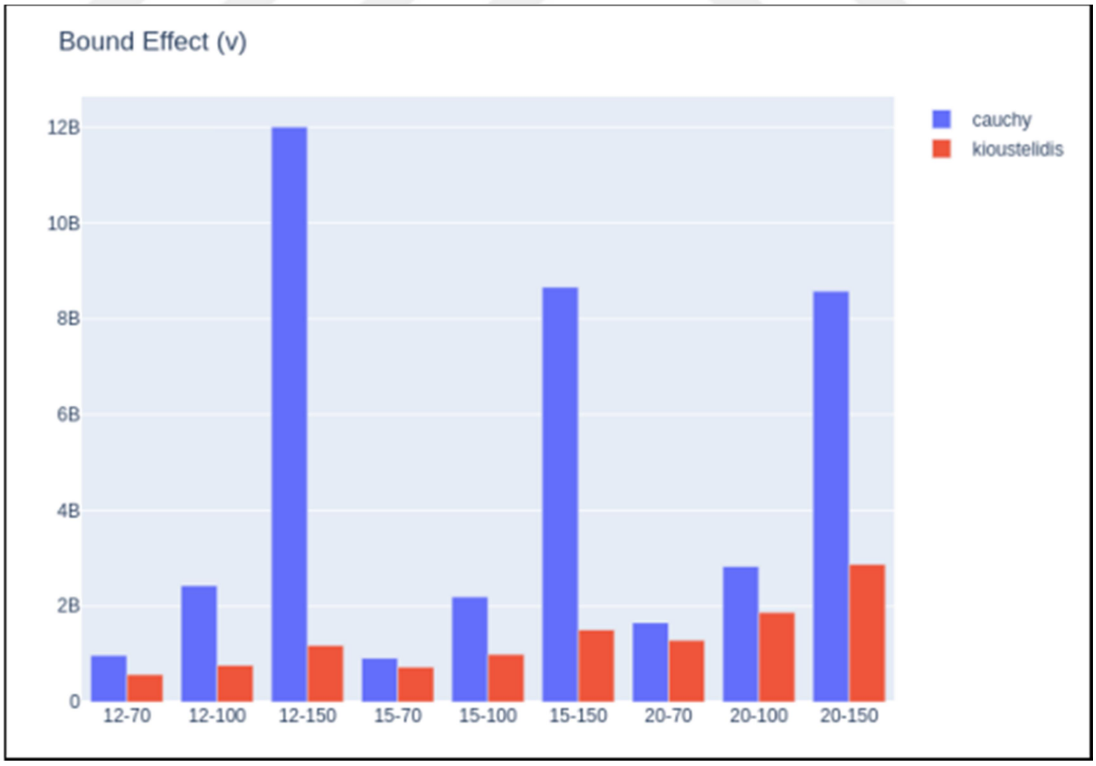


Figure 6.12: Effect of Bound Selection (CF).

6.4. Analysis

Random coefficient polynomials (coe) are solved by three methods (Vincent, Sturm, Oscar). For small degrees and bit sizes Vincent is the best whereas Oscar beats as bit size and degree increases (Figure 6.13).

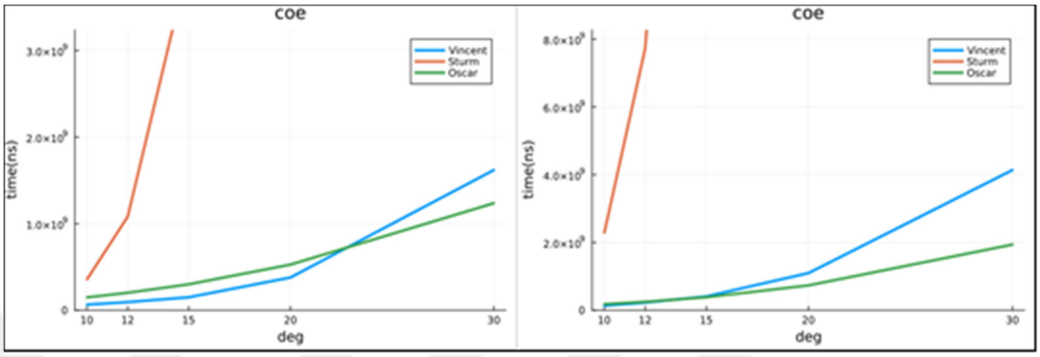


Figure 6.13: Coe-50 and Coe-200.

Integer root polynomials (int) are solved by all of the methods. Oscar is the best for all degrees and bit sizes. For high degrees and bit sizes Vincent is better than Sturm (Figure 6.14).

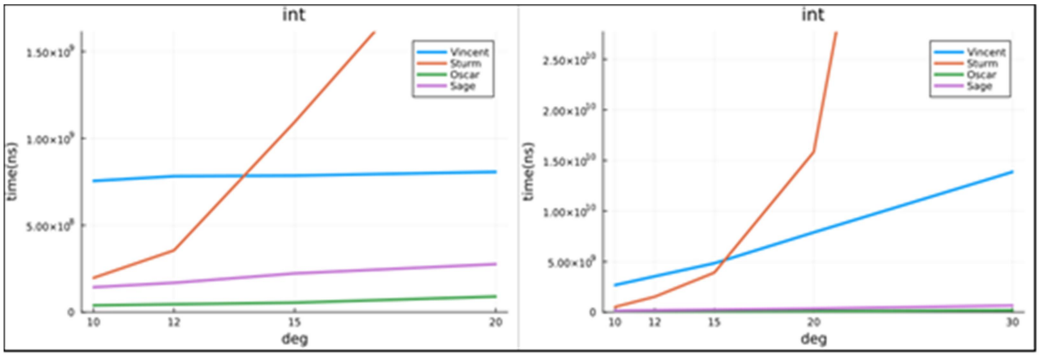


Figure 6.14: Int-50 and Int-200.

Irrational root polynomials (irr) are solved by all of the methods. Oscar is the best for all degrees and bit sizes. For high degrees and bit sizes Vincent is better than Sturm (Figure 6.15).

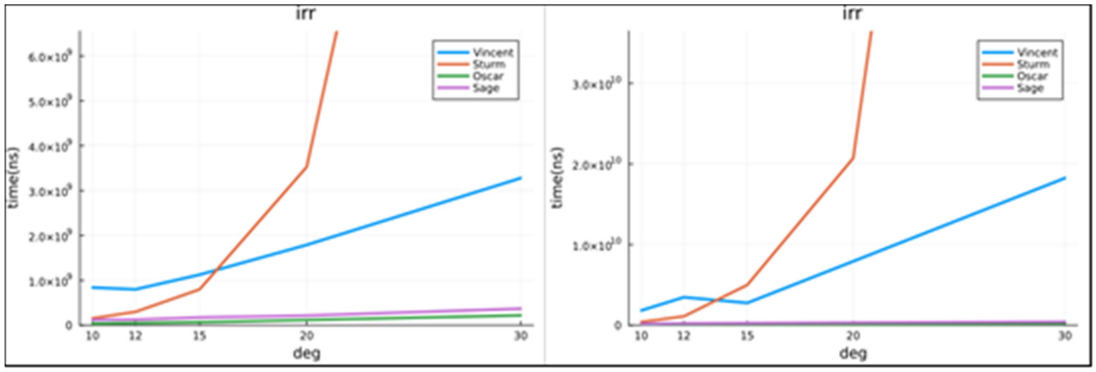


Figure 6.15: Irr-50 and Irr-200.

Mignotte polynomials (mig) are solved by three of the methods(Vincent, Sturm, Sage). Sage is the best for all degrees and bit sizes. Sturm is better than Vincent but for high degrees and bit sizes Vincent starts beating Sturm (Figure 6.16).

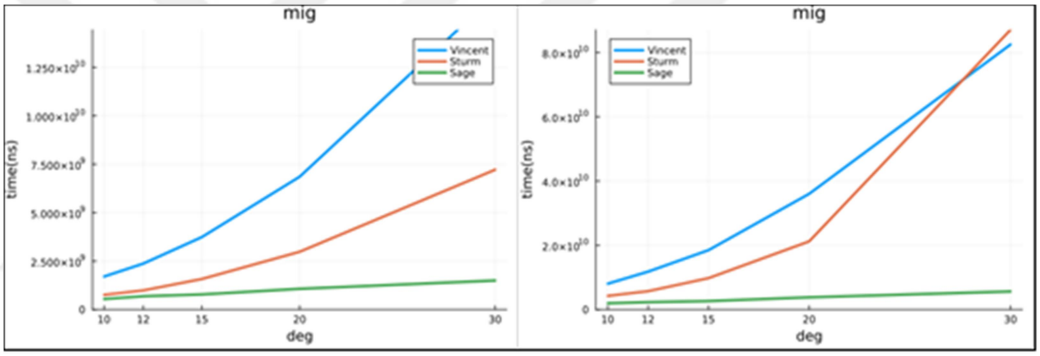


Figure 6.16: Mig-50 and Mig-200.

Mixed coefficient polynomials (mix) are solved by three methods (Vincent, Sturm, Oscar). Oscar is the best for all degrees and bit sizes. Vincent is more efficient than Sturm (Figure 6.17).

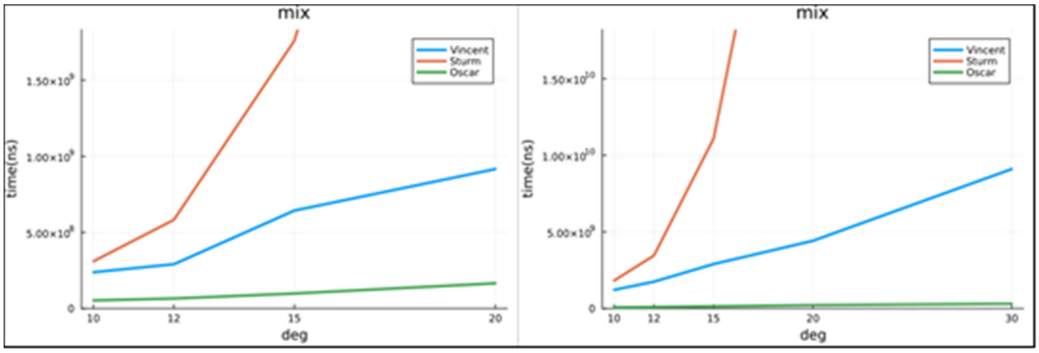


Figure 6.17: Mix-50 and Mix-200.

Best and worst performances of Vincent method are at random coefficient and Mignotte polynomials (Figure 6.18).

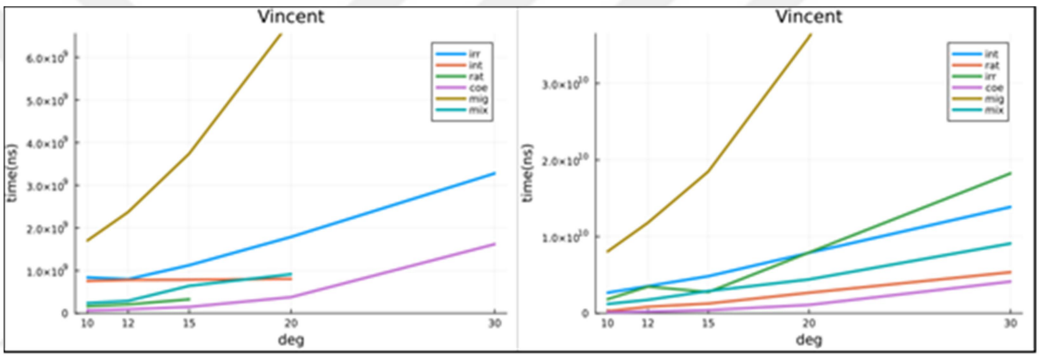


Figure 6.18: Vincent-50 and Vincent-200.

Best and worst performances of Sturm method are at Mignotte and random coefficient polynomials (Figure 6.19).

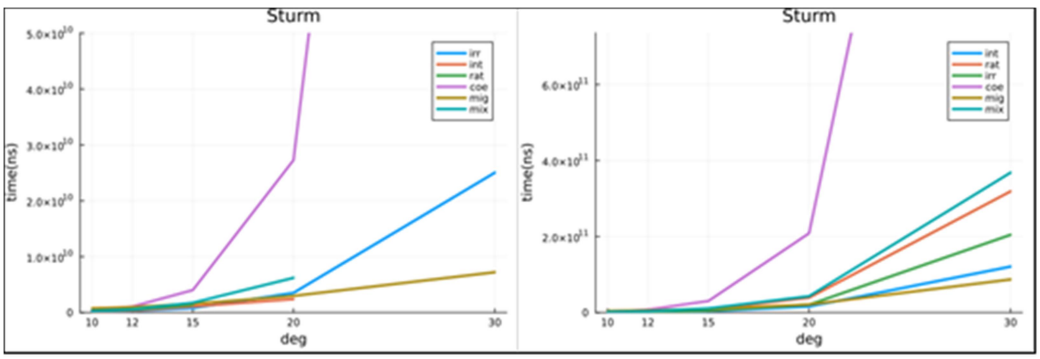


Figure 6.19: Sturm-50 and Sturm-200.

7. CONCLUSION

In this work Vincent and Sturm methods are implemented for isolation of real roots of polynomials. To test performance a data set containing 14300 square free polynomials are generated. The polynomials are up to degree 30 and coefficient bit size 200. 6 types of polynomials are used. These are polynomials only have integer roots (int), polynomials only have rational roots (rat), polynomials have irrational and complex roots (irr), polynomials have integer, rational, irrational and complex roots (mix), polynomials with random coefficients (coe) and Mignotte polynomials (mig).

Both CF and Sturm methods give correct intervals for all data set. Although Oscar and SageMath have better performance, they cannot reach solution for all types of polynomials.

Except for very small degree and bit sizes CF method gives better performance. Best and worst performances of Vincent method are at random coefficient and Mignotte polynomials. Best and worst performances of Sturm method are at Mignotte and random coefficient polynomials.

Although both Sturm and CF methods use polynomial root bounds, selection of root bound has nearly no effect for Sturm method since it is used only in first step. But Kioustelidis bound is much better for CF method.

Polynomial reduction (when found root exactly) improves CF performance. Converting rational coefficients to integer does not improve the performance.

CF method can be modified to represent real algebraic numbers and make arithmetic operations using Gosper's method.

REFERENCES

[1] Akritas G., (1978), “A new method for polynomial real root isolation”, In Proceedings of the 16th annual Southeast regional conference, 39–43, Atlanta/USA, 13 April 1978.

[2] Akritas G., (1986), “There is no Uspensky’s method.”, In Proceedings of the fifth ACM symposium on Symbolic and algebraic computation, 88–90, Waterloo/Canada, 21 July 1986.

[3] Collins E., Akritas G., (1976), “Polynomial real root isolation using Descarte’s rule of signs”, In Proceedings of the third ACM symposium on Symbolic and algebraic computation, 272–275, New York/USA, 10 August 1976.

[4] Buchberger B., Collins E., Loos R., (1983), “Computer Algebra Symbolic and Algebraic Computation”, Second Edition, Springer-Verlag.

[5] Beeler M, Gosper W., Schroeppel R., (1972), “Technical Report HAKMEM”, AIM 239, ITEM 101B Continued Fraction Arithmetic, Massachusetts Institute of Technology, USA.

[6] Heindel E., (1971), “Integer arithmetic algorithms for polynomial real zero determination”, In Proceedings of the second ACM symposium on Symbolic and algebraic manipulation, 415–426, Los Angeles/USA, 23 March 1971.

[7] Hemmer M., Tsigaridas E., Zafeirakopoulos Z., Emiris I., Karavelas M., Mourrain B., (2009), “Experimental evaluation and cross-benchmarking of univariate real solvers”, In Proceedings of the 2009 conference on Symbolic numeric computation, 45–54, Kyoto/Japan, 3 August 2009.

[8] Johnson J., (1991), “Algorithms for polynomial real root isolation”, PhD thesis, The Ohio State University.

[9] Kioustelidis B., (1986), “Bounds for positive roots of polynomials”, Journal of Computational and Applied Mathematics, 16 (2), 241–244.

[10] Pan Y., (2002), “Univariate polynomials: nearly optimal algorithms for factorization and root finding”, Journal of Symbolic Computation, 33 (5), 701-733.

[11] Rouillier F., Zimmermann P., (2004), “Efficient isolation of polynomial’s real Roots”, Journal of Computational and Applied Mathematics, 162 (1), 33-50.

[12] Trotter H., Lang S., (1972), “Continued fractions for some algebraic numbers”, Journal für die reine und angewandte Mathematik, 255, 112-134.

[13] Uspensky J., (1948), “Theory of Equations”, First Edition, McGraw-Hill.

BIOGRAPHY

Hüseyin Sabri Aliefendioğlu graduated from the Department of Mechanical Engineering, Middle East Technical University in 1997. He worked in Aselsan Microelectronics, Guidance & Electro-Optics Business Sector from 1999 to 2010. Between 2011 to 2022 he worked in Otokar Electromechanical Systems Department.

APPENDICES

Appendix A: Publications on the thesis

Aliefendioğlu H. S., Zafeirakopoulos Z., (2022), ”Continued Fractions Representation of Real Algebraic Numbers”, In 2022 GTU’s 6th Graduate Research Symposium.

Appendix B: Vincent Method Application Steps

Table B.1: Vincent Method Application Steps.

$$P(x) = 10x^4 - 33x^3 - 3x^2 + 99x - 81$$

$$P(x) = 10x^4 - 33x^3 - 3x^2 + 99x - 81 \quad \delta p(x) = 3 \quad lb = 0 \tag{1}$$

Substitute in (1) x $\leftarrow \frac{1}{1+x}$ $\quad cf : 0 + \dfrac{1}{1+x}$

$$P(x) = -81x^4 - 225x^3 - 192x^2 - 66x - 8 \quad \delta p(x) = 0 \rightarrow \text{no root}$$

Substitute in (1) x $\leftarrow 1 + x$ $\quad cf : 1 + x$

$$P(x) = 10x^4 + 7x^3 - 42x^2 + 34x - 8 \quad \delta p(x) = 3 \quad lb = 0 \tag{2}$$

Substitute in (2) x $\leftarrow \frac{1}{1+x}$ $\quad cf : 1 + \dfrac{1}{1+x}$

$$P(x) = -8x^4 + 2x^3 + 12x^2 - 7x + 1 \quad \delta p(x) = 3 \quad lb = 0 \tag{3}$$

$P(1) = 0 \rightarrow 1 + \dfrac{1}{1+1} = \frac{3}{2}$ is a root of (1)

Substitute (3) x $\leftarrow \frac{1}{1+x}$ $\quad cf : 1 + \dfrac{1}{1 + \dfrac{1}{1+x}}$

$$P(x) = x^4 - 3x^3 - 3x^2 + 9x \quad \delta p(x) = 2 \quad lb = 0 \tag{4}$$

Substitute in (4) x $\leftarrow \frac{1}{1+x}$ $\quad cf : 1 + \dfrac{1}{1 + \dfrac{1}{1 + \dfrac{1}{1+x}}}$

$$P(x) = 9x3 + 24x2 + 18x + 4 \quad \delta p(x) = 0 \rightarrow \text{no root}$$

Substitute in (4) x $\leftarrow 1 + x$ $\quad cf : 1 + \dfrac{1}{1 + \dfrac{1}{2+x}}$

$$x^4 + x^3 - 6x^2 - 2x + 4 \quad \delta p(x) = 2 \quad lb = 0 \tag{5}$$

Table B.1: Continued.

Substitute in (5) x $\leftarrow \frac{1}{1+x}$ $cf: 1+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+x}}}$

$$P(x) = 4x^4 + 14x^3 + 12x^2 - x - 2 \quad \delta p(x) = 1- > \text{ one root in interval}$$

Substitute 0 and ∞ in cf to get interval $[\frac{5}{3}, \frac{7}{4})$

Substitute in (5) x $\leftarrow 1+x$ $cf: 1+\cfrac{1}{1+\cfrac{1}{3+x}}$

$$P(x) = x^4 + 5x^3 + 3x^2 - 7x - 2 \quad \delta p(x) = 1- > \text{ one root in interval}$$

Substitute 0 and ∞ in cf to get interval $[\frac{7}{4}, 2)$

Substitute (3) x $\leftarrow 1+x$ $cf: 1+\cfrac{1}{2+x}$

$$P(x) = -8x^4 - 30x^3 - 30x^2 - 9x \quad \delta p(x) = 0 \rightarrow \text{no root}$$

Substitute (2) x $\leftarrow 1+x$ $cf: 2+x$

$$P(x) = 10x^4 + 47x^3 + 39x^2 + 11x + 1 \quad \delta p(x) = 0 \rightarrow \text{no root} \tag{6}$$

Substitute (3) x $\leftarrow (-x)$ for negative roots

$$P(x) = 10x^4 + 33x^3 - 3x^2 - 99x - 81 \quad \delta p(x) = 1- > \text{ one root in interval} \tag{7}$$

$interval(-\infty, 0)$

So isolating intervals : $(-\infty, 0), [\frac{3}{2}, \frac{3}{2}], [\frac{5}{3}, \frac{7}{4}), [\frac{7}{4}, 2)$

$(-\sqrt{3}, \frac{3}{2}, \sqrt{3}, \frac{9}{5}$ are the roots of the polynomial)

Appendix C: Vincent Algorithm

Table C.1: Julia Code of Vincent Method.

```
function Vincent(polynom::Polynomial{T}) where T<:Number
    cf = [Int128(0)]
    intervals = Vector{Vector{Number}}()
    sign = 1
    findIntervals(polynom, cf, intervals, sign)
        for interval in intervals
            if(interval[1] == interval[2])
                pq, rem = PolynomialUtility.syntheticDivision(polynom,
                interval[1])
                if(rem == 0)
                    polynom = pq
                end
            end
        end
    polynom_minus = PolynomialUtility.minusPolynomial(polynom)
    cf_minus = [Int128(0)]
    sign = -1
    findIntervals(polynom_minus, cf_minus, intervals, sign)
    sortIntervals(intervals)
    return intervals
end

function findIntervals(polynom::Polynomial{<:Number},
cf::Vector{<:Number}, intervals::Vector{Vector{Number}}, sign::Int)
    count_sign_changes = PolynomialUtility.
    signCount(coeffs(polynom))
        if(count_sign_changes == 0)
            return
        elseif(count_sign_changes == 1)
            addedInterval = getInterval(cf, sign)
            push!(intervals, addedInterval)
            return
        end
```

Table C.1: Continued.

```
        isOneRoot = (polynom(1)==0)
        if(isOneRoot)
            cf_root_1 = deepcopy(cf)
            cf_root_1[end] = cf_root_1[end]+1
            root_corresponding_to_1= getConvergents(cf_root_1)[2] * sign
            pq, rem = PolynomialUtility.syntheticDivision(polynom, 1)
                if(rem == 0)
                    polynom = pq
                end
            push!(intervals,
            [root_corresponding_to_1,root_corresponding_to_1])
        end

        lowerbound = PolynomialUtility.PLB(polynom)
        if lowerbound > 1
            modifiedlb = lowerbound-1
            polynom =
            PolynomialUtility.transformation(polynom,modifiedlb)
            cf[end] = cf[end] + modifiedlb
        end
        polynom01 = PolynomialUtility.revTransformation(polynom,1)
        cf01 = deepcopy(cf)
        push!(cf01,1)
        polynom1inf = PolynomialUtility.transformation(polynom,1)
        cf1inf = deepcopy(cf)
        cf1inf[end] = cf1inf[end] + 1
        findIntervals(polynom01, cf01, intervals, sign)
        findIntervals(polynom1inf, cf1inf, intervals, sign)
end
```

Appendix D: Sturm Algorithm

Table D.1: Julia Code of Sturm Method.

```
function Sturm(polynom:: Polynomial)
    sturmSequence = getSturmSequence(polynom)
    intervals = findIntervals(sturmSequence)
    return intervals
end


function getSturmSequence(polynom::Polynomial)
    sturmSequence = []
    p1 = polynom
    p2 = derivative(p1)
    push!(sturmSequence,p1)
    while(length(p2) > 1)
        push!(sturmSequence, p2)
        p3 = PolynomialUtility.minusRemainder(p1, p2)
        p1 = p2
        p2 = p3
    end
    push!(sturmSequence, p2)
    return sturmSequence
end


function findIntervals(sturmSequence)
    sc_minus_inf = signCount(sturmSequence, -1//0)
    sc_zero = signCount(sturmSequence, 0)
    sc_inf = signCount(sturmSequence, 1//0)
    intervals = Vector{Vector{Number}}()
    upper = PolynomialUtility.PUB(sturmSequence[1])
    lower = -upper
    if(sc_zero < sc_minus_inf)
        if(sc_zero == sc_minus_inf-1)
            push!(intervals,[-1//0,0//1])
        else
            lower = - PolynomialUtility.PUB(
            PolynomialUtility.minusPolynomial(sturmSequence[1]))
            while(signCount(sturmSequence,lower) < sc_minus_inf )
                lower= 2 * lower
            end
            bisection(sturmSequence, lower, sc_minus_inf, 0,
            sc_zero, intervals)
        end
    end
```

Table D.1: Continued.

```
    if(sc_inf < sc_zero)
        if(sc_inf == sc_zero-1)
            push!(intervals,[0//1,1//0])
        else
            upper = PolynomialUtility.PUB(sturmSequence[1])
            while( signCount(sturmSequence, upper)>sc_inf )
                upper = 2 * upper
            end
            bisection(sturmSequence, 0, sc_zero, upper, sc_inf,
            intervals)
            end
        end
    return intervals
end

function bisection(sturmSequence, lower, sc_lower,
upper, sc_upper, intervals)
    if(upper-lower > 2)
        mid = round(Int128, (upper+lower) * Float64(0.5))
    else
        mid = Rational{BigInt}(upper+lower) * (1//2)
    end
    sc_mid = signCount(sturmSequence,mid)
    if(sc_mid < sc_lower)
        if(sc_mid == sc_lower-1)
            if(sturmSequence[1](mid) == 0)
                push!(intervals,[mid, mid])
            else
                push!(intervals,[lower, mid])
            end
        else
            bisection(sturmSequence, lower, sc_lower,
            mid, sc_mid, intervals)
        end
    end
    if(sc_upper<sc_mid)
        if(sc_upper == sc_mid-1)
            if(sturmSequence[1](upper) == 0)
                push!(intervals,[upper, upper])
            else
                push!(intervals,[mid, upper])
            end
        else
            bisection(sturmSequence, mid, sc_mid,
            upper, sc_upper, intervals)
        end
    end
```

Appendix E: CF Arithmetic Algorithm

Table E.1: Julia Code of CF Arithmetic.

```
mutable struct ContinuedFraction{T}
    convergents::Array{T,1}
    op:: Union{Nothing, Operation{T}}
    x::Union{Nothing,ContinuedFraction}
    y::Union{Nothing,ContinuedFraction}
    ready::Bool
    ix::Int
    iy::Int
    p::Union{Nothing,Polynomial}
end

function ContinuedFraction(polynom::Polynomial{T}) where {T<:Number}
    return ContinuedFraction{T}(Array{T,1}(), nothing,
    nothing, nothing, false, 1, 1, polynom)
end

function ContinuedFraction(el::T) where {T<:Number}
    return ContinuedFraction{T}([T(el)],
    Operation(MArray{Tuple{2,4}, T}([T(0)
    T(0) T(0) T(0); T(0) T(0) T(0) T(0)])) ,nothing
    ,nothing,true,1,1,nothing)
end

function ContinuedFraction(numerator::T,denominator::T) where
{T<:Number}
    return ContinuedFraction{T}([],
    Operation(MArray{Tuple{2,4}, T}([T(numerator) T(0) T(0) T(0);
    T(denominator) T(0) T(0) T(0)])), nothing, nothing, false,
    1,1,nothing)
end

function ContinuedFraction(a::Array{T,1}) where {T<:Number}
    return ContinuedFraction{T}(a,
    Operation(MArray{Tuple{2,4}, T}([T(0) T(0) T(0) T(0);
    T(0) T(0) T(0) T(0)]))nothing ,nothing,true,1,1,nothing)
end
```

Table E.1: Continued.

```
function add(a::ContinuedFraction, b::ContinuedFraction)
    T = getCFGenericParameter(a)
    return ContinuedFraction{T}([],
    Operation(MArray{Tuple{2,4}, T}([T(0) T(1) T(1) T(0);
    T(1) T(0) T(0) T(0)])), a, b, false, 1,1,nothing)
end

function subtract(a::ContinuedFraction, b::ContinuedFraction)
    T = getCFGenericParameter(a)
    return ContinuedFraction{T}([],
    Operation(MArray{Tuple{2,4}, T}([T(0) T(1) T(-1) T(0);
    T(1) T(0) T(0) T(0)])), a, b, false,1,1,nothing)
end

function multiply(a::ContinuedFraction, b::ContinuedFraction)
    T = getCFGenericParameter(a)
    return ContinuedFraction{T}([],
    Operation(MArray{Tuple{2,4}, T}([T(0) T(0) T(0) T(1);
    T(1) T(0) T(0) T(0)])), a, b, false,1,1,nothing)
end

function divide(a::ContinuedFraction, b::ContinuedFraction)
    T = getCFGenericParameter(a)
    return ContinuedFraction{T}([],
    Operation(MArray{Tuple{2,4}, T}([T(0) T(1) T(0) T(0);
    T(0) T(0) T(1) T(0)])), a, b, false,1,1,nothing)
end
```

Appendix F: CF Representation of Cube Root 7

Table F.1: CF Elements of Cube Root 7 by Lang and Trotter.

$\sqrt[3]{7}$

	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10
0	1	1	10	2	16	2	1	4	2	1	21	1	3	5	1	2	1	1	2	11
20	5	1	3	1	2	27	4	1	*a	8	1	2	1	1	3	1	3	2	6	4
40	1	2	1	5	1	1	2	1	1	1	3	2	8	1	2	2	4	5	1	1
60	36	1	1	1	1	2	1	2	31	2	1	1	7	1	1	1	1	6	7	6
80	5	7	1	6	1	6	9	6	*b	5	33	8	2	1	1	1	6	2	1	1
100	1	1	2	13	7	1	1	1	17	3	2	1	1	1	23	*c	1	*d	1	72
120	1	4	6	6	1	11	6	12	7	3	5	1	4	6	2	2	1	2	4	4
140	1	1	9	11	6	2	1	15	6	2	1	1	2	1	3	5	1	2	1	1
160	1	1	1	15	1	35	1	2	1	2	4	2	1	11	2	1	1	1	1	15
180	4	1	1	2	22	2	1	31	2	2	1	1	3	1	10	2	1	2	1	1
200	2	1	1	8	1	5	1	1	6	2	1	14	1	4	7	5	2	6	6	1
220	4	1	1	3	2	10	1	3	2	16	2	34	1	1	1	18	3	1	7	1
240	3	12	1	1	15	3	2	1	2	2	1	2	2	17	3	2	2	3	5	8
260	1	56	1	2	1	25	1	3	1	3	3	8	20	8	1	3	1	1	1	8
280	1	4	12	1	3	6	3	1	3	7	19	3	1	1	13	1	1	1	1	10
300	2	2	1	1	3	15	1	4	1	*e	1	7	1	8	2	1	1	2	13	3
320	7	56	3	2	4	2	4	2	5	1	1	9	3	1	1	7	4	1	6	1
340	47	1	2	31	6	2	4	1	4	7	9	1	1	3	9	2	1	14	1	3
360	1	5	5	1	6	15	1	5	1	*f	6	1	1	11	4	1	6	8	19	1
380	1	1	8	2	6	4	19	3	14	14	3	1	11	1	6	1	1	2	2	1
400	1	1	1	5	2	1	1	1	1	4	2	3	2	1	1	1	1	1	2	3
420	1	2	1	4	1	2	12	1	1	13	3	1	1	2	*g	4	1	1	1	14
440	1	10	1	*h	7	60	9	32	1	6	2	13	1	1	1	2	1	98	1	1
460	1	2	1	2	1	*i	1	4	3	2	2	4	4	1	2	35	2	25	8	3
480	3	1	1	6	1	1	1	10	1	2	3	2	4	2	2	5	2	6	2	2
500	1	1	1	10	1	1	1	6	1	35	30	1	2	1	1	1	3	2	1	3
520	4	1	9	1	1	9	6	1	1	15	2	1	5	1	3	3	1	1	2	5
540	2	60	3	23	1	2	1	1	1	4	11	9	13	1	4	1	*j	2	1	11
560	3	2	1	1	5	2	52	1	4	1	8	2	3	2	3	25	1	14	1	1
580	1	7	8	1	2	11	15	1	2	14	2	1	6	1	2	7	1	1	2	2
600	1	1	1	2	1	1	1	1	1	9	*k	3	3	2	5	1	1	1	1	3
620	1	8	1	1	1	11	2	3	2	1	1	3	4	1	3	1	4	1	1	6
640	19	1	1	2	1	1	1	1	1	3	8	7	99	1	41	1	11	51	1	1
660	7	17	2	1	7	1	5	1	4	15	1	1	1	3	1	6	2	1	69	1
680	3	1	18	2	2	2	4	2	13	11	1	6	1	2	1	14	3	3	1	3
700	7	1	12	2	78	2	1	1	1	1	*m	5	1	10	5	2	1	2	5	1
720	6	3	2	2	10	13	4	27	1	2	1	1	44	19	4	3	1	3	1	1
740	2	2	5	3	1	1	1	1	1	1	1	2	1	1	1	11	1	8	6	1
760	7	1	1	2	19	1	5	3	1	1	3	1	3	1	1	44	1	34	1	5
780	3	8	1	2	3	*n	11	1	4	10	3	1	1	2	2	2	18	16	6	3
800	63	1	1	5	1	19	24	1	1	7	53	1	1	4	2	1	1	*p	2	1
820	5	2	7	5	4	1	1	3	1	30	1	1	1	2	1	7	1	7	1	1
840	24	2	1	4	1	1	3	2	1	3	21	1	1	1	73	1	4	2	1	1
860	4	9	1	2	4	2	3	1	1	3	7	2	11	2	1	4	8	8	15	11
880	18	17	1	1	2	7	43	2	3	1	1	6	2	1	10	1	5	3	3	3
900	8	22	4	3	1	4	4	4	6	1	7	1	1	9	1	18	3	4	1	1
920	2	1	2	7	1	2	2	1	15	1	1	2	5	1	2	52	3	1	87	1
940	3	1	1	1	6	10	1	1	1	3	2	4	1	6	1	2	1	3	2	3
960	1	1	1	5	2	1	1	8	1	1	8	1	2	4	1	*q	2	8	7	2
980	84	4	1	11	2	2	12	3	1	1	1	3	4	12	2	1	9	1	72	2

a = 282, b = 104, c = 277, d = 429, e = 303, f = 341
g = 110, h = 197, i = 118, j = 133, k = 689, m = 115
n = 111, p = 202, q = 628

Table F.2: CF Elements of Cube Root 7.

0	1	1	10	2	16	2	1	4	2	1	21	1	3	5	1	2	1	1	2	11
20	5	1	3	1	2	27	4	1	282	8	1	2	1	1	3	1	3	2	6	4
40	1	2	1	5	1	1	2	1	1	1	3	2	8	1	2	2	4	5	1	1
60	36	1	1	1	1	2	1	2	31	2	1	1	7	1	1	1	1	6	7	6
80	5	7	1	6	1	6	9	6	104	5	33	8	2	1	1	1	6	2	1	1
100	1	1	2	13	7	1	1	1	17	3	2	1	1	1	23	277	1	429	1	72
120	1	4	6	6	1	11	6	12	7	3	5	1	4	6	2	2	1	2	4	4
140	1	1	9	11	6	2	1	15	6	2	1	1	2	1	3	5	1	2	1	1
160	1	1	1	15	1	35	1	2	1	2	4	2	1	11	2	1	1	1	1	15
180	4	1	1	2	22	2	1	31	2	2	1	1	3	1	10	2	1	2	1	1
200	2	1	1	8	1	5	1	1	6	2	1	14	1	4	7	5	2	6	6	1
220	4	1	1	3	2	10	1	3	2	16	2	34	1	1	1	18	3	1	7	1
240	3	12	1	1	15	3	2	1	2	2	1	2	2	17	3	2	2	3	5	8
260	1	56	1	2	1	25	1	3	1	3	3	8	20	8	1	3	1	1	1	8
280	1	4	12	1	3	6	3	1	3	7	19	3	1	1	13	1	1	1	1	10
300	2	2	1	1	3	15	1	4	1	303	1	7	1	8	2	1	1	2	13	3
320	7	56	3	2	4	2	4	2	5	1	1	9	3	1	1	7	4	1	6	1
340	47	1	2	31	6	2	4	1	4	7	9	1	1	3	9	2	1	14	1	3
360	1	5	5	1	6	15	1	5	1	341	6	1	1	11	4	1	6	8	19	1
380	1	1	8	2	6	4	19	3	14	14	3	1	11	1	6	1	1	2	2	1
400	1	1	1	5	2	1	1	1	1	4	2	3	2	1	1	1	1	1	2	3
420	1	2	1	4	1	2	12	1	1	13	3	1	1	2	110	4	1	1	1	14
440	1	10	1	197	7	60	9	32	1	6	2	13	1	1	1	2	1	98	1	1
460	1	2	1	2	1	118	1	4	3	2	2	4	4	1	2	35	2	25	8	3
480	3	1	1	6	1	1	1	10	1	2	3	2	4	2	2	5	2	6	2	2
500	1	1	1	10	1	1	1	6	1	35	30	1	2	1	1	1	3	2	1	3
520	4	1	9	1	1	9	6	1	1	15	2	1	5	1	3	3	1	1	2	5
540	2	60	3	23	1	2	1	1	1	4	11	9	13	1	4	1	133	2	1	11
560	3	2	1	1	5	2	52	1	4	1	8	2	3	2	3	25	1	14	1	1
580	1	7	8	1	2	11	15	1	2	14	2	1	6	1	2	7	1	1	2	2
600	1	1	1	2	1	1	1	1	1	9	689	3	3	2	5	1	1	1	1	3
620	1	8	1	1	1	11	2	3	2	1	1	3	4	1	3	1	4	1	1	6
640	19	1	1	2	1	1	1	1	1	3	8	7	99	1	41	1	11	51	1	1
660	7	17	2	1	7	1	5	1	4	15	1	1	1	3	1	6	2	1	69	1
680	3	1	18	2	2	2	4	2	13	11	1	6	1	2	1	14	3	3	1	3
700	7	1	12	2	78	2	1	1	1	1	115	5	1	10	5	2	1	2	5	1
720	6	3	2	2	10	13	4	27	1	2	1	1	44	19	4	3	1	3	1	1
740	2	2	5	3	1	1	1	1	1	1	1	2	1	1	1	11	1	8	6	1
760	7	1	1	2	19	1	5	3	1	1	3	1	3	1	1	44	1	34	1	5
780	3	8	1	2	3	111	11	1	4	10	3	1	1	2	2	2	18	16	6	3
800	63	1	1	5	1	19	24	1	1	7	53	1	1	4	2	1	1	202	2	1
820	5	2	7	5	4	1	1	3	1	30	1	1	1	2	1	7	1	7	1	1
840	24	2	1	4	1	1	3	2	1	3	21	1	1	1	73	1	4	2	1	1
860	4	9	1	2	4	2	3	1	1	3	7	2	11	2	1	4	8	8	15	11
880	18	17	1	1	2	7	43	2	3	1	1	6	2	1	10	1	5	3	3	3
900	8	22	4	3	1	4	4	4	6	1	7	1	1	9	1	18	3	4	1	1
920	2	1	2	7	1	2	2	1	15	1	1	2	5	1	2	52	3	1	87	1
940	3	1	1	1	6	10	1	1	1	3	2	4	1	6	1	2	1	3	2	3
960	1	1	1	5	2	1	1	8	1	1	8	1	2	4	1	628	2	8	7	2
980	84	4	1	11	2	2	12	3	1	1	1	3	4	12	2	1	9	1	72	2