Real-Root Isolation of Polynomials

XINLONG YI,
Computer Science and Engineering Department, University of California Riverside, USA

1 ABSTRACT

Finding real roots of polynomials is a fundamental problem in scientific computing. This project aims to implement a real-root isolation program based on Budan's Theorem and Continued Fraction. Both methods are developed from Descartes' rule of signs and work only on square-free polynomials. Therefore, square-free decomposition is the very first step to isolate the roots of polynomials. This project applies Yun's algorithm to make the polynomials have no repeated roots. And in order to handle errors that come from floating point computation, interval arithmetic is introduced to replace exact numbers. Besides that, many methods are used to optimize the program and running time of these two algorithms are compared. This report also discusses how errors propagated through computation and how it affects the success of the program.

2 INTRODUCTION

One of the important scientific computing problems is computing the real roots of polynomials. It has wide applications in computer graphics. For the polynomials with low order, like quadratic or cubic, we can use formulas to get roots directly. However, according to the Abel–Ruffini theorem, there is no solution in radicals to general polynomial equations of five degrees or higher with arbitrary coefficients[5].

There are a lot of algorithms to find the real roots of polynomials. Although most root-finding algorithms, Newton's method for example, may produce some real roots, the convergence of algorithms are not guaranteed. Furthermore, these methods cannot generally certify having found all real roots. Which means if such methods do not find any root, one cannot know if there are real roots or not. Moreover, there are some cases where one does not need the exact roots of polynomial equations. Take ray tracing as an example. When computing the distance from intersection point to ray's end point, the object with smallest distance is important to us, instead of exact distance to point of intersection. In these cases, computing exact roots is not very essential.

Real-root isolation is an appropriate method for the above problems. It will generate a sequence of disjoint intervals. Each interval contains only one real root of the polynomial. And combining these intervals together, all real roots could be found. Besides that, isolating the roots instead of computing them out might speed up the cases that do not need exact roots, as mentioned above.

This project aims to implement a real-root isolation program based on Budan's theorem and continued fraction. These two methods are based on Descartes' rule of signs, which describes how to get information on the number of positive real roots of a polynomial and first introduced by René Descartes[3]. Budan's theorem, developed from Descartes' rule of signs, provides the methods to bound the number of real roots of a polynomial in an interval[1]. Vincent introduced the continued fraction method in his work in 1834[2]. Both methods work only on square-free polynomials. Therefore, this project takes Yun's algorithm to perform square free decomposition[8].

The organization of this report is as follows. Section 3 will introduce the theory basis of this project. Section 4 describes how the program implemented and what have been tried to optimize.

Author's address: Xinlong Yi, xyi007@ucr.edu,
Computer Science and Engineering Department, University of California Riverside, 900 University Ave, Riverside, California, USA, 92507.

Running time comparison and analysis of how error propagation through computation will be discussed in Section 5. Finally, a conclusion is drawn in Section 6.

3 METHODOLOGY

3.1 Square Free Decomposition

In mathematics, a square-free polynomial is a polynomial defined over a field that does not have a divisor any square of a non-constant polynomial[8]. Usually, a square-free polynomial refers to the polynomials with no repeated roots. This project applied Yun's algorithm to perform square-free decomposition. It's based one the succession of Greatest Common Divisor(GCD).

3.1.1 Greatest Common Divisor. In algebra, the greatest common divisor of two polynomials is a polynomial, of the highest possible degree, that is a factor of both the two original polynomials. This concept is similar to the GCD of two integers. Ideally, both polynomials divided by their greatest common divisor should have no remainder.

This project uses Euclidean algorithm to compute the GCD of two polynomials. In Algorithm 1, $rem(a, b)$ refers to the remainder of Euclidean division of polynomial a and polynomial b.

Algorithm 1: GCD of two polynomials

```
input  : $P1$: a univariate polynomial
         $P2$: a univariate polynomial
output : Greatest common divisor of $P1$ and $P2$
$r_0 = P1$;
$r_1 = P2$;
for $i = 1; r_i \neq 0; i = i + 1$ do
    $r_{i+1} = rem(r_{i-1}, r_i)$
end
return $r_{i-1}$
```

3.1.2 Yun's Algorithm. Based on the success of GCD, Yun developed a square-free decomposition algorithm for univariate polynomials.

Given a primitive polynomial P, assume $P = P_1 P_2^2 ... P_n^n$ is desired factorization. Yun's algorithm will compute square-free polynomials P_i and the subscript refers to the times this square-free polynomial appears. Which means P_i appears i times in original polynomial. This process is described in Algorithm 2 formally.

Algorithm 2: Yun's Square-free Decomposition Algorithm

```
input  : Primitive polynomial $P$
output : List of square-free polynomials
$G = GCD(P, dP/dx)$;
$C_1 = P/G$;
$D_1 = (dP/dx)/G - dC_1/dx$;
for $i = 1, C_i \neq 0; i = i + 1$ do
    $P_i = GCD(C_i, D_i)$;
    $C_{i+1} = C_i/P_i$;
    $D_{i+1} = D_i/P_i - dC_{i+1}/dx$;
end
return $P_1, P_2 ... P_n$
```

3.2 Budan's Theorem

Budan's theorem is a theorem used for bounding the number of real roots in a given interval. Given a univariate polynomial P, we denote $\#_{l,r}(P)$ as the number of real roots of P in half-open interval $(l, r]$. Then we denote $v_h(P)$ as the number of sign changes in coefficients of polynomial P_h, where $P_h(x) = P(x + h)$.

Budan's theorem states that $v_l(h) - v_r(h) - \#_{l,r}(P)$ is a nonnegative even integer. From this statement, we can know that if $v_l(h) - v_r(h) = 1$ or 0, there is only one or zero real root in interval $(l, r]$.

Based on this theorem, this project combines bisection method with Budan's theorem to isolate the real roots. This process described in Algorithm 3.

Algorithm 3: Real-root isolation based on Budan's Theorem

```
input  :A square-free polynomial $P$
output:List of intervals contains only one real root
$ret = []$;
$up\_bound = Upper(P)$;
$low\_bound = -up\_bound$;
$search = [(low\_bound, up\_bound)]$;
while $search$ $not$ $empty$ do
    $l, r = pop(search)$;
    $vl = sign\_change(P(x + l))$;
    $vr = sign\_change(P(x + r))$;
    if $vl - vr = 1$ then
        $ret.append([l, r])$;
    end
    else if $vl - vr > 1$ $and$ $r - l \geq MINIMAL\_RANGE$ then
        $mid = l + (r - l)/2$;
        $search.append(mid, r)$;
        $search.append(l, mid)$;
    end
end
return $ret$;
```

In Algorithm 3, $Upper(P)$ returns the upper bound of real roots of P. This project uses Lagrange's bound[4] in this project. Assuming $P = a_0 + a_1x + ... + a_nx^n$, Lagrange's bound is $max\{1, \sum_{i=0}^{n-1} |\frac{a_i}{a_n}|\}$.

3.3 Continued Fraction

Let's first introduce some notations used in this algorithm. Let $M(x)$ represent a Mobius transformation, which maps x to $\frac{ax+b}{cx+d}$. So that the number of positive roots of $P(M(x))$ equals the number of roots in interval $(\frac{b}{d}, \frac{a}{c}]$ of P. Denote the sign changes of coefficients of P as s.

We use $\{a, b, c, d, p, s\}$ to represent an interval. Where, $ad - bc \neq 0$ and the roots of original polynomial P in interval $(\frac{b}{d}, \frac{a}{c}]$ are images of positive roots of p.

Continued fraction method can be formalized as Algorithm 4.

Algorithm 4: Real-root isolation based on Continued Fraction

```
input  : A square-free polynomial with no zero root $P$
output : List of intervals contains only one positive real root
$s = sign\_change(P)$;
if $s = 0$ then
    return [];
end
else if $s = 1$ then
    return $[(0, \infty)]$;
end
$ret = []$;
$intervals = [\{1, 0, 0, 1, P, s\}]$;
while intervals not empty do
    $\{a, b, c, d, p, s\} = pop(intervals)$;
    $p' = p(x + 1)$;
    if $p'(0) = 0$ then
        $ret.append([\frac{b}{d}, \frac{b}{d}])$;
        $p' = p'/x$;
    end
    $s' = sign\_change(p')$;
    $intervals.append(\{a, a + b, c, c + d, p', s'\})$;
    if $s - s' = 1$ then
        $ret.append([\frac{a}{c}, \frac{b}{d}])$;
    end
    else if $s - s' > 1$ then
        $p'' = p(b/(1 + x))$;
        $intervals.append(\{b, a + b, d, c + d, p'', sign\_change(p'')\})$;
    end
end
return ret;
```

Algorithm 4 can only returns positive real roots, therefore, Algorithm 4 will be applied both on $P(x)$ and $P(-x)$. Zero roots should be handled before using Algorithm 4.

4 IMPLEMENTATION

4.1 Polynomial

At first, we tried to use *vector* data structure from the *std* library to implement polynomial class since it can handle coefficients dynamically. However due to the time consuming dynamic memory allocation process, it has been changed to static process.

We use template class with an integer template n to represent the maximum possible degree of polynomial. Which means although a polynomial class with template n, it's highest degree could be $n1$ where $n1 \leq n$. Although in this way, developers need to handle the possible change over the template when writing some operators, like $*, +$, programs can avoid dynamic memory allocation, which may save a lot of running time.

4.2 Replace x with $x + a$

Changing $P(x)$ to $P(x + a)$ in a polynomial is a very common operation in both methods, since both methods need to use the coefficients of $P(x + a)$ to get the sign variance.

At first, we tried a very primitive idea for this process. Initialize a very simple polynomial $p = x + a$. Then multiply it by coefficients and increase its order one by one. Add them together at last to get transformed polynomial.

In order to optimize this process, we apply Taylor expansion. This process is also called Taylor shift[6]. It runs as appling Taylor expansion at point a with $\Delta x = x$. Then we have:

$$P(x + a) = P(a) + P'(a) * x + ... + \frac{1}{n!}P^{(n)}(a) * x^n$$

This process avoids multiplication of two polynomials. Multiplication needs $O(n^2)$ operations in our implementation. In order to make sure Taylor expansion can improve the performance of the program, we made some experiments to compare the computation time of these two implementations. Results are in table 4.2. Coefficients of test polynomials are generated randomly from -1000 to 1000. Since this project does not pay attention to polynomials with degrees higher than 6, we only test polynomials with degree from 3 to 6.

Degree	Tylor Expansion	Original Method
3	1876 *ns*	2272 *ns*
4	2640 *ns*	3439 *ns*
5	3307 *ns*	4910 *ns*
6	4574 *us*	7103 *us*

As shown above, the running time of polynomial shifting with Taylor Expansion is about 25% faster than the primitive one in our implementation.

4.3 Approximate GCD

Errors in floating point computation are unavoidable. Therefore, it is hard to check if one polynomial is zero or not. Greatest common divisor and Square-free decomposition can not perform perfectly all the time.

In order to solve this error problem, we tried tolerance. If a number smaller than this tolerance, it will be treated as zero. At first we used fixed tolerance, which cannot perform well as the magnitude of the largest coefficient grows.

Then we tried the method introduced by Matu-Tarow[7]. It suggests to turn polynomials *regular* before performing *GCD*. A polynomial is *regular* means its leading coefficient is $O(1)$ and any other coefficients either $O(1)$ or 0. The notation $O(c)$ not same as the Landau's symbol and it means a number of approximately the same magnitude as c. Any univariate polynomial can be transformed to *regular* by scaling $P \longrightarrow \xi P$ and $x \longrightarrow \eta x$.

However, it is hard to define the η ξ when transforming a polynomial to *regular*. Besides that, if we transform x to ηx, roots will be changed accordingly. And the precision of the algorithm will be changed. Furthermore, this process will introduce extra computation and make the program less efficient.

4.4 Interval arithmetic

Due to the problems mentioned in the last part of the previous subsection, we discard approximate GCD and use *interval arithmetic*. Interval arithmetic represents each value as a range of possibilities so that we can keep track of errors from floating point computation in algorithms. We use interval arithmetic implemented in *boost* library.

For the numbers that can be represented by a computer perfectly, no error might be introduced through computation. However, if we get some numbers that not suitable for a computer to store, errors might appear. For example, $\frac{1}{3}$ might be represented as $0.33333333333 \pm \epsilon$, where ϵ might as small as $1e^{-17}$.

With interval arithmetic, the computation has been doubled since every computation need to calculate both ends of a interval. However, since it can handle the error from floating point computation, our program becomes more stable.

Although interval arithmetic can not make our program runs perfectly on all of cases, using interval arithmetic allows us to observe how errors propagated through computation. With this observation, we can figure out in what condition our program might fail. This part will be discussed in Section5.

5 RESULTS AND ANALYSIS

5.1 Running Time

With above implementation and operation, we test the running time of these two methods. Results can be found in table 5.1. Since we do not pay attention to the polynomials with degree more than 6, we only test the polynomials with degree 6 in this section. These test polynomials are generated with random coefficients from $-c$ to c.

c	Budan's Theorem	Continued Fraction
10	195 us	26 us
20	240 us	22 us
50	243 us	32 us
100	257 us	32 us
1000	259 us	32 us

From table 5.1, we can see that Budan's theorem method takes longer than the continued fraction method. This might be because in Budan's theorem, we need to get the upper bound of root up_b first and get $P(x + up\_b)$ to check how many positive real roots. However in continued fraction, sign variance of P can tell us the information about positive roots directly.

Furthermore, in Budan's theorem method, sometimes the search range needs to be small enough to figure out whether there are roots or not. Take $P(x) = x^4 + 1$ as an example. In Budan's theorem method with search range $[0 - \epsilon, 0 + \epsilon]$, $v_{0+\epsilon}(P) - v_{0-\epsilon}(P) = 4$ is always valid, no matter how small the ϵ is. Therefore, the bisection process will terminate only when $2\epsilon < MINIMAL\_RANGE$. However, the continued fraction method will check zero root first, then find no positive root since $v_0(P(x)) = 0$ and no negative roots since $v_0(P(-x)) = 0$. That's the reason why the Budan's theorem method takes longer than the continued fraction method.

Above polynomials are generated with random coefficients, which means it cannot guarantee that those polynomials have real roots. In order to figure out the relationship between the running time with magnitude of roots, we designed following experiments. Test polynomials have 6 roots and every root has $p = 0.5$ probability same as the previous one. Distinct roots are generated randomly in range $[-max\_root, max\_root]$. Degree of test polynomials is 6. Running times are shown in table 5.1.

max_root	Budan's Theorem	Continued Fraction
1	23 us	23 us
10	28 us	25 us
50	34 us	36 us
100	36 us	51 us
200	39 us	86 us
500	42 us	180 us
1000	47 us	300 us

First, we can observe that when roots go larger the continued fraction method takes longer than the Budan's theorem method. It is noticeable that $p' = p(x+1)$ step in Algorithm 4 only shifts polynomial by 1 each time. Therefore, when roots are large, it takes longer for the continued fraction method to shift the polynomial to the place around roots. This place needs to be optimized in the future. Shifting can be flexible according to the lower bound of roots of polynomials.

Furthermore, we can also notice that the Budan's theorem method runs faster than the previous one while the continued fraction method runs slower. This mainly because these polynomials are guaranteed to have 6 real roots. With 6 real roots, there are no conjugate complex roots, like $x^4 - 1$. This condition makes Budan's theorem method terminate earlier. Besides that, some factors of original polynomial might only have degree of 2 or 1. These factors can be solved very quickly. However, for the continued fraction method, it takes longer to shift polynomials to the place around roots.

5.2 Error Analysis

Although this implementation can work correctly in most cases, there are some conditions that could lead to failure.

Since both methods need to work on polynomials with no repeat roots, square-free decomposition becomes the most important step in the program. However, Yun's algorithm requires exact divisions and GCD success, which is hard to achieve in floating point computation.

As mentioned before, we introduce interval arithmetic to handle the errors from inexact computation and keep track of errors. We find that errors are related to the magnitude of coefficients. Errors might be enlarged during the process of GCD and long division. Let's simulate the first step of $GCD(P, P')$, which is $rem(P, P')$ where $P' = \frac{dP}{dx}$.

Given a polynomial $P = a_0 + a_1 * x + a_2 * x^2 ... + a_n * x^n$, and its derivative $P' = a_1 + 2 * a_2 * x ... + n * a_n * x^{n-1}$. We assume all of coefficients of P and P' can be represented by computer perfectly. Which means the width of interval a_i is zero.

Then the processes to calculate $rem(P, P')$ are:

$$
\begin{aligned}
div1 &= 1/n \\
rem1 &= P - div_1 * x * P' = a_0 + \frac{n-1}{n} a_1 * x + \frac{n-2}{n} a_2 * x^2 .... \frac{1}{n} * a_{n-1} x^{n-1} \\
&= \sum_{i=0}^{n-1} \frac{n-i}{n} a_i x^i \\
div2 &= \frac{a_{n-1}}{a_n} * \frac{1}{n^2} \\
&= div_1 * \frac{a_{n-1}}{na_n} \\
rem2 &= rem1 - div2 * P' = (a_0 - div2 * a_1) + (\frac{n-1}{n} * a_1 - div2 * 2 * a_2) + ... \\
&= \sum_{i=0}^{n-2} (\frac{n-i}{n} * a_i - \frac{a_{n-1} * a_{i+1}}{n^2 * a_n}(i+1))x^i \\
&= \sum_{i=0}^{n-2} ((1 - div1 * i) * a_i - div2 * a_{i+1}(i+1))x^i
\end{aligned}
$$

Since $rem2$ only has degree $n-2$ which is one less than P', it is the answer of $rem(P, P')$. Let's assume errors appear in computation of $div1$. Which means $div1 = \frac{1}{n} \pm \epsilon$. Then we assume $\xi = \frac{a_{n-1}}{na_n}$ is exact but very large, like $1e^{20}$. Then the $div2$ becomes $\frac{a_{n-1}}{a_n * n^2} \pm \epsilon * \xi$. Since ξ is very large, the error of $div2$ is not ignorable. And it will affect the coefficients of $rem2$, which makes the $rem2$ not accurate.

Therefore, if the magnitude of coefficients is very large, there is potential for this program to fail in *GCD* and long division processes and lead to failure in real-root isolation.

6 CONCLUSION

In this project, we implement a real-roots isolation program based on Budan's theorem and continued fraction. Besides that, we also use several methods to optimize the program, like Taylor Expansion for polynomial shift and interval arithmetic for error control.

Then we compare the running time of these two algorithms. It is noticeable that when there are conjugate complex roots, Budan's theorem will take longer. And continued fraction methods perform worse when polynomials have several large real roots due to slow shifts of polynomials. This can be optimized in the future.

With the application of interval arithmetic, our program becomes more robust to the errors from inexact computation and representation, although it is not perfect for all of the cases. We analyze the conditions that might cause failure of the program and find that the magnitude of coefficients might affect error's change during the processes of *GCD* and *remainder* computation. If the coefficients are large, the errors might become not ignorable and affect the accuracy of factors. Which might finally lead to failure of our program.

REFERENCES

[1] [n.d.]. *Budan's theorem.* https://en.wikipedia.org/wiki/Budan%27s_theorem Last edited: 2020-12-31.

[2] [n.d.]. *Continued fraction method.* https://en.wikipedia.org/wiki/Real-root_isolation#Continued_fraction_method Last edited: 2020-12-31.

[3] [n.d.]. *Descartes' rule of signs.* https://en.wikipedia.org/wiki/Descartes%27_rule_of_signs Last edited: 2021-3-9.

[4] [n.d.]. *Lagrange's bounds.* https://en.wikipedia.org/wiki/Geometrical_properties_of_polynomial_roots#Lagrange's_and_Cauchy's_bounds Last edited: 2021-3-9.

[5] Raymond G. Ayoub. 1980. Paolo Ruffini's Contributions to the Quintic. *Archive for History of Exact Sciences* 23, 3 (1980), 253–277. http://www.jstor.org/stable/41133596

[6] J. Gerhard. 2004. Modular Algorithms in Symbolic Summation and Symbolic Integration. In *Lecture Notes in Computer Science.* 42–49.

[7] Matu-Tarow Noda and Tateaki Sasaki. 1991. Approximate GCD and its application to ill-conditioned equations. *J. Comput. Appl. Math.* 38, 1 (1991), 335–351. https://doi.org/10.1016/0377-0427(91)90180-R

[8] David Y.Y. Yun. 1976. On Square-Free Decomposition Algorithms. In *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation* (Yorktown Heights, New York, USA) *(SYMSAC '76).* Association for Computing Machinery, New York, NY, USA, 26–35. https://doi.org/10.1145/800205.806320