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 most fundamental scientific computations is computing the real roots of polynomials. 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[3], there is no solution in radicals to general polynomial equations with five degrees or higher with arbitrary coefficients. Therefore, general root-finding algorithms are needed for arbitrary polynomials.

However, the usual root-finding algorithms, like Newton Method, cannot generally certify having found all real roots. Especially, if such algorithms does not find any root, one cannot know if there are real roots or not. In order to get all real roots, real-root isolation is useful. Real-root isolation can generate intervals, which contain only one real root of the polynomial, so that no real root will be missed.

In this project, I implemented two real-root isolation algorithms based on Budan's Theorem[1] and Continued Fraction. After basic implemented, I optimized the program with better data structure and interval arithmetic. After that, I compare the running time of two algorithms and analyze in what condition these methods will fail.

The organization of this report is as follows. Section 3 will introduce the methodologies that used to implement the real-root isolation. Section 4 has the way that I implemented this project and what I tried to optimize it. And running time comparison and how error propagated will be discussed in Section 5.

3 METHODOLOGY

This section introduce the methods that applied to this project. Since both two methods are only work on square-free polynomials, the first step of this project is applying square free decomposition to original polynomials to avoid repeat roots. After square free decomposition, methods based on Budan's Theorem and Continued Fraction will be applied. They both based on Descartes' rule of sign to check how many real roots in a interval. The continued fraction method also uses Mobius transformation, which will be introduce in Continued Fraction subsection.

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

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[5]. Usually, a square-free polynomial refers to the polynomials with no repeated roots. This project applied Yun's algorithm[5] 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 simillar to the GCD of two integers.

This project applied Euclid's algorithm to compute the GCD of two polynomials. In the 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 succession of GCD, Yun developed a square-free decomposition algorithm for univariate polynomials. Given a primitive polynomial P, the algorithm will compute square-free polynomials P_i and the subscript refers to the times this square-free polynomial appears. Which means $P = \prod_{i=1}^{k} P_i^i$.

Algorithm 2: Yun's Square-free Decomposition Algorithm

input : P: primitive polynomial
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 $P_1, P_2 ... P_k$

3.2 Budan's Theorem

Budan's Theorem is a theorem used for bounding the numnber 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 above statement, we can know that if $v_l(h) - v_r(h) = 1$ or 0, there is only one real root or no root in interval $(l, r]$.

Based on this theorem, this project applied bisection to isolate the real roots. This process described in Algorithm 3

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

input : P square-free polynomial
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 $vl - vr > 1$ **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. We are using Lagrange's bound[2] 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 notation used in this algorithm. Let $M(x)$ represent a Mobius Transformation, which map x to $\frac{ax+b}{cx+d}$. So that the number of positive roots of $P(M(x))$ equals to the number of roots in interval $(\frac{b}{d}, \frac{a}{c}]$ of P. And $s = sign\_change(P)$ represents the sign changes along coefficients of P.

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

The algorithm can be described as Algorithm 4.

Algorithm 4: Real-root isolation based on Continued Fraction

input : P square-free polynomial with no zero root
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*;

Above algorithm only accept the polynomials with non-zero roots, therefore before using the Algorithm 4, we need to remove the zero roots of original P. Then we used Algorithm 4 with $P(x)$ and $P(-x)$ to get the positive roots and negative roots.

4 IMPLEMENTATION

This section will introduce how I implement this project and what I tried to optimize it.

4.1 Polynomial

At first, I tried to use *vector* from *std* library to implement polynomial class. But due to time consuming of dynamic memory allocation process. I changed it to static process.

I use template class with an integer template n to represents 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, programmer need to handle the possible change over

the template when writing some operators, like $*, +$, program can get rid of dynamic memory allocation, which can save a lot of running time.

4.2 Replace x with $x + a$

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

At first, I used very primitive idea for this process. Initialize with very simple polynomial $x + a$. 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 the process that replace x to $x+a$ in a polynomial, I applied Taylor expansion. If I want to change $P(x)$ to $P(x + a)$, I can take Taylor expansion at point a with $\Delta x = x$. Then:

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

This process avoid multiplication of two polynomials, which needs $O(n^2)$ operations in my implementation. In order to make sure Taylor expansion can improve the perform of program, I made some experiments to compare the computation time of changing x to $x + a$. Reuslts are in table 4.2. Coefficients in test polynomial are generated randomly from $-n$ to n and degree of test polynomial is $n - 1$.

n	Tylor Expansion	Original Method
10	12us	16 us
20	42us	58 us
50	250us	340 us
100	986us	1345 us
150	2169us	3082 us
200	3801us	5784 us

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

Appling Fast Fourier Transform can speed up polynomial multiplication process. Maybe I will implement it in the future.

4.3 Approximate GCD

With above implementation, it cannot perform square-free decomposition correctly all the time. The main reason behind this is the representation of float in computer and computation error.

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

Then, I tried to make the original polynomial *regular*[4], which means make the leading coefficient of polynomial to $O(1)$ and remaining coefficients either $O(1)$ or 0 by scaling transformation $P \longrightarrow \xi P$ and $x \longrightarrow \eta x$. The notation $O(c)$ not same as the Landau's symbol and it means a number of approximately the same magnitude as c.

However, changing of coefficients and roots will introduce extra computation. Besides that, in some cases that have several extremely large roots, changing coefficients to make them $O(1)$ will make some coefficients very small and loose the percision.

4.4 Interval arithmetic

With above problem, I discard the methods introduced by Matu-Tarow[4] and applied *interval arithmetic* to the project. Every exact number in project has been changed to a interval that represents a set of possible values. If zero in such interval, this number will be seen as zero.

With interval arithmetic, the computation has been doubled since every computation need to computation both ends of interval. However, this implementation can handle the errors from float computation and data measuring.

By using interval arithmetic, I observed how error propagates during the GCD process. It's close related the magnitude of coefficient of polynomial. This will be discussed in Analysis 5 section.

5 RESUTL AND ANALYSIS

5.1 Running Time

Whit above implementation, I test the running time of these two methods. Result can be found in following table 5.1. Same as mentioned of Tylor Expansion test, test polynomials are generated with random coefficient from $-n$ to n with degree $n-1$.

n	Budan's Theorem	Continued Fraction
10	8.430ms	0.786ms
20	38.574ms	1.826ms
50	208.707ms	3.387 ms
100	988.346ms	11.301ms
150	2574.148ms	12.656ms
200	3918.112ms	21.634ms

From the table 5.1, we can see that Budan's Theorem method takes longer time than Continued Fraction method. This might because with bisection method, searching range need to be small enough before one can figure out if there is root or not. However, above polynomials are generated with random coefficients, which cannot guarantee it has real roots.

In order to figure out the relationship between the running time with magnitude of roots, I designed following experiments. Test polynomials has several roots and have every roots has $p = 0.5$ probability same as previous one. Then every root is in ranger $[-max\_root, max\_root]$. Running time is shown in table 5.1. Since with large degree, some number might be truncated, I test with polynomials with degree 5.

max_root	Budan's Theorem	Continued Fraction
1	38us	38us
20	31us	29us
50	38us	39us
100	42us	56us
150	46us	70us
200	46us	83us

There are several reasons that makes running time in this table is much lesser than the previous one. The first one is the degree of these polynomials are small, only 5 actually. Second, there are repeat roots in these polynomials and after square-free decomposition, square-free polynomials might only have small degree like 1 or 2, which can be solved with closed formula solution.

And we can see that, running time of Continued Fraction method is longer than Budan's Theorem. It's noticeable that $p' = p(x+1)$ line in Algorithm 4. It is only shift polynomial by 1 for each time. Therefore, when root is large, it takes longer for Continued Fraction method to shift to the

place around roots. This place is need to be optimized, if the shifting can be flexible and adjustable according to the lower bound of roots of polynomial it would be faster.

5.2 Fail Conditions

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 polynomial with no repeat roots, square-free decomposition becomes the most important step in the program. However, with the Yun's algorithm[5], *GCD* computation should be success and divisions should be exact, which cannot be achieved by floating number.

As mentioned before, introducing interval arithmetic can see how error propagated along with square-free decomposition process. Although can not know how exact errors changing over *GCD* process, it's related with the magnitude of the coefficients.

The magnitude of coefficients influence the error through the process computing the remainder of two polynomials.

Let polynomial $P = a_0 + a_1 * x + a_2 * x^2 ... + a_n * x^n$, and its derivative $P' = a_1 + 2 * a_2 * x ... + n * a_n * x^{n-1}$. Let's simulate the first step of *GCD* process, which is compute the $rem(P, P')$. All of coefficient in polynomial $a_0 ... a_n$ are intervals with no error if it can represented by computer perfectly

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

$rem2$ in above equations only has degree $n-1$ which is one less than P', therefore, it should be the remainder of P/P'. It's noticeable that $div2 = \frac{a_{n-1}}{a_n} * \frac{1}{n^2}$ might be very large. Although a interval $a_i = [a_i - \epsilon, a_i + \epsilon]$ might be small at first, when it multiplied by $div2$, the interval will be enlarged to $a_i * div2 = [(a_i - \epsilon) * div2, (a_i + \epsilon) * div2]$ and might not be ignorable. Besides that, with the intervals enlarged, some number might be regarded as zero accidentally.

Therefore, if magnitude of coefficients is very large and roots are close enough, there are potential for this program to failed in *GCD* process and lead to failure in real-root isolation.

6 CONCLUSION

In this project, I tried to implemented real-roots isolation program based on Budan's Theorem and Continued Fraction methods. Besides that, I also tried several methods to optimize the program, like Tylor Expansion for polynomial shift and interval arithmetic for error control.

Then we compare the running time of both methods. In general, continued fraction methods needs lesser computation time for polynomials with arbitrary coefficients. But, for the polynomials with large roots, continued fraction might take longer, since slow shift of polynomial and this can be optimized in the future.

With the interval arithmetic, I tried to analyze the conditions that might cause failure of the program. And found that the tolerance of intervals might be enlarged in the process of *remainder*

and *GCD* computation. If the magnitude of coefficient is large and the roots are close, program might failed.

REFERENCES

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

[2] [n.d.]. *Lagrange's bounds.* https://en.wikipedia.org/wiki/Geometrical_properties_of_polynomial_roots#cite_note-2 Last edited: 2021-3-9.

[3] 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

[4] 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

[5] 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