Nonlinear Analysis: Modelling and Control, 2005, Vol. 10, No. 4, 297–304

A Comparative Study of Two Real Root Isolation Methods

A. G. Akritas[1], **A. W. Strzeboński**[2]

[1]University of Thessaly, Department of Computer and Communication Engineering
GR-38221 Volos, Greece
akritas@uth.gr

[2]Wolfram Research, Inc., 100 Trade Center Drive, Champaign, IL 61820, USA
adams@wolfram.com

Received: 07.10.2005
Accepted: 10.11.2005

Abstract. Recent progress in polynomial elimination has rendered the computation of the real roots of ill-conditioned polynomials of high degree (over 1000) with huge coefficients (several thousand digits) a critical operation in computer algebra.
To rise to the occasion, the only method-candidate that has been considered by various authors for modification and improvement has been the Collins-Akritas *bisection* method [1], which is a based on a variation of Vincent's theorem [2]. The most recent example is the paper by Rouillier and Zimmermann [3], where the authors present

> ... a new algorithm, which is optimal in terms of memory usage and as fast as both Collins and Akritas' algorithm and Krandick variant ... [3]

In this paper we compare our own *continued fractions* method *CF* [4] (which is directly based on Vincent's theorem) with the best bisection method *REL* described in [3]. Experimentation with the data presented in [3] showed that, with respect to time, our continued fractions method *CF* is by far superior to *REL*, whereas the two are about equal with respect to space.

Keywords: root isolation, Vincent's theorem, coninued fractions method, bisection (or Collins-Akritas) method.

1 Description of the two algorithms

For completeness we briefly describe our implementation of both the continued fractions method CF and algorithm REL. The correctness of the first algorithm

along with an analysis of its computational complexity can be found in the literature [5–8]. A discussion of the second algorithm can be found elsewhere [3].

1.1 Description of the Continued Fractions Algorithm CF

Let us first introduce the notation used in the algorithm. Let $f \in Z[x] \setminus \{0\}$. By $sgc(f)$ we denote the number of sign changes in the sequence of nonzero coefficients of f. For nonnegative integers a, b, c, and d, such that $ad - bc \neq 0$, we put

$$intrv(a, b, c, d) := \Phi_{a,b,c,d}\big((0, \infty)\big),$$

where

$$\Phi_{a,b,c,d} \colon (0, \infty) \ni x \longrightarrow \frac{ax + b}{cx + d} \in (0, \infty)$$

and by *interval data* we denote a list

$$\{a, b, c, d, p, s\},$$

where p is a polynomial such that the roots of f in $intrv(a, b, c, d)$ are images of positive roots of p through $\Phi_{a,b,c,d}$, and $s = sgc(p)$.

The value of parameter α_0 used in step 4 below needs to be chosen empirically. In our implementation $\alpha_0 = 16$.

Algortihm Continued Fractions (CF)
Input: a squarefree polynomial $f \in Z[x] \setminus \{0\}$.
Output: the list $rootlist$ of positive roots of f.

1. Set $rootlist$ to an empty list. Compute $s \leftarrow sgc(f)$. If $s = 0$, return an empty list. If $s = 1$, return $\{(0, \infty)\}$. Put interval data $\{1, 0, 0, 1, f, s\}$ on $intervalstack$.

2. If $intervalstack$ is empty, return $rootlist$, else take interval data $\{a, b, c, d, p, s\}$ off $intervalstack$.

3. Compute a lower bound α on positive roots of p.

4. If $\alpha > \alpha_0$, set $p(x) \leftarrow p(\alpha x)$, $a \leftarrow \alpha a$, $c \leftarrow \alpha c$, and $\alpha \leftarrow 1$.

5. If $\alpha \geq 1$, set $p(x) \leftarrow p(x+\alpha)$, $b \leftarrow \alpha a + b$, and $c \leftarrow \alpha c$. If $p(0) = 0$, add $[b/d, b/d]$ to $rootlist$, and set $p(x) \leftarrow p(x)/x$. Compute $s \leftarrow sgc(p)$. If $s = 0$, go to step 2. If $s = 1$, add $intrv(a, b, c, d)$ to $rootlist$ and go to step 2.

6. Compute $p_1(x) \leftarrow p(x+1)$, and set $a_1 \leftarrow a$, $b_1 \leftarrow a + b$, $c_1 \leftarrow c$, $d_1 \leftarrow c + d$, and $r \leftarrow 0$. If $p_1(0) = 0$, add $[b_1/d_1, b_1/d_1]$ to $rootlist$, and set $p_1(x) \leftarrow p_1(x)/x$, and $r \leftarrow 1$. Compute $s_1 \leftarrow sgc(p_1)$, and set $s_2 \leftarrow s - s_1 - r$, $a_2 \leftarrow b$, $b_2 \leftarrow a + b$, $c_2 \leftarrow d$, and $d_2 \leftarrow c + d$.

7. If $s_2 > 1$, compute $p_2(x) \leftarrow (x+1)^m p(\frac{1}{x+1})$, where m is the degree of p. If $p_2(0) = 0$, set $p_2(x) \leftarrow p_2(x)/x$. Compute $s_2 \leftarrow sgc(p_2)$.

8. If $s_1 < s_2$, swap $\{a_1, b_1, c_1, d_1, p_1, s_1\}$ with $\{a_2, b_2, c_2, d_2, p_2, s_2\}$.

9. If $s_1 = 0$, goto step 2. If $s_1 = 1$, add $intrv(a_1, b_1, c_1, d_1)$ to $rootlist$, else put interval data $\{a_1, b_1, c_1, d_1, p_1, s_1\}$ on $intervalstack$.

10. If $s_2 = 0$, goto step 2. If $s_2 = 1$ add $intrv(a_2, b_2, c_2, d_2)$ to $rootlist$, else put interval data $\{a_2, b_2, c_2, d_2, p_2, s_2\}$on $intervalstack$. Go to step 2.

In the present paper we also address the issue of memory usage by the continued fraction method. We show that the algorithm can be so structured that the maximal number of transformed versions of the polynomial that need to be stored at any given time is bounded by $1 + \log_2 n$, where n is the degree of the input polynomial. This bound is based on the following conjecture, which we have not proven, but which we have extensively tested.

Conjecture. Let f be a polynomial of degree n, and let $sgc(f)$ denote the number of sign changes in the sequence of nonzero coefficients of f. Then

$$sgc\big(f(x+1)\big) + sgc\Big((x+1)^n f(\frac{1}{x+1})\Big) \leq sgc\big(f(x)\big)$$

Hence the number of sign changes in any interval on $intervalstack$ is at least equal to the total number of sign changes in all intervals above it. Since the number of sign changes in the top interval is at least 2, and the total number of sign changes in all intervals on stack is at most the degree n of f, the number of possible levels in $intervalstack$ is at most $\log_2 n$. Therefore the maximal number of transformed polynomials we need to keep at any given time is at most $1 + \log_2 n$.

1.2 Description of the Algorithm REL

As in [3], let

$$H_a(p)(x) = p(ax),$$
$$T_a(p)(x) = p(x + a).$$

The algorithm calls subprocedure *DesBound* which for a polynomial p of degree m returns $min(2, sgc((x+1)^m p(\frac{1}{x+1})))$. This is done by computing subsequent coefficients of the Taylor shift of $x^m p(\frac{1}{x})$, returning 2 as soon as we get two sign changes.

Algortihm REL

Input: a squarefree polynomial $f \in Z[x] \setminus \{0\}$.

Output: the list $rootlist$ of positive roots of f.

1. Set $rootlist$ to an empty list. Set $p \leftarrow f$. Compute an upper bound B on positive roots of p (a nonnegative power of 2). If $B > 1$, set $p(x) \leftarrow p(Bx)$.

2. Compute $s \leftarrow DesBound(p)$. If $s = 0$, return an empty list. If $s = 1$, return $\{(0, B)\}$. Put pairs $(1, 1)$ and then $(1, 0)$ on $intervalstack$. (Pair (k, c) corresponds to interval $[B\frac{c}{2^k}, B\frac{c+1}{2^k}]$.) Set $k \leftarrow 0$ and $c \leftarrow 0$.

3. If $intervalstack$ is empty, return $rootlist$, else take pair (k', c') off $intervalstack$.

4. Compute $p \leftarrow 2^{n(k'-k)} H_{2^{k-k'}}(T_{2^{k-k'}c'-c}(p))$. ([3] proves that the translation is either the identity or the Taylor shift.)

5. If $p(0) = 0$, add $[B\frac{c'}{2^{k'}}, B\frac{c'}{2^{k'}}]$ to $rootlist$, and set $p(x) \leftarrow p(x)/x$.

6. If $k' \leq k$, compute $s \leftarrow sgc(p)$. If $s = 1$, add $(B\frac{c'}{2^{k'}}, B)$ to $rootlist$. If $s = 0$ or $s = 1$, return $rootlist$.

7. Set $c \leftarrow c'$, $k \leftarrow k'$, and compute $s \leftarrow DesBound(p)$.

8. If $s > 1$, put pairs $(k + 1, 2c + 1)$ and then $(k + 1, 2c)$ on $intervalstack$. If $s = 1$, add $(B\frac{c}{2^k}, B\frac{c+1}{2^k})$ to $rootlist$. Go to step 3.

2 Empirical results

We compare performance of our continued fraction algorithm CF, and the algorithm REL described in [3]. We have implemented both algorithms as a part of *Mathematica* kernel. They both use the same implementation of Shaw and Traub's algorithm for Taylor shifts (see [9]). As benchmark examples we use Chebyshev, Laguerre, Wilkinson, and Mignotte polynomials used in [3], as well as three types of randomly generated polynomials used in [4].

All computations were done on a 850 MHz Athlon PC with 256 MB RAM. The memory used data was obtained using *Mathematica* MaxMemoryUsed command, so is includes the total memory used by *Mathematica* kernel. The startup size of *Mathematica* kernel is 1.6 MB.

In case of special polynomials, Table 1, CF is faster by factors ranging from around 3 for Chebyshev polynomials to 50000 for Mignotte polynomials. The case of Mignotte polynomials is especially advantageous for our continued fractions method, because there is a point with a very simple continued fraction expansion (namely $\frac{1}{5}$), which lies between the two close roots. For Chebyshev polynomials we used the fact that the polynomials are even and so with both methods we isolated only the positive roots.

Table 1. Special polynomials

Polynomial	Degree	No. of roots	CF $T(s)/M$ (MB)	REL $T(s)/M(MB)$
Chebyshev	1000	1000	2172/9.2	7368/8.5
Chebyshev	1200	1200	4851/12.8	15660/11.8
Laguerre	900	900	3790/8.7	22169/14.1
Laguerre	1000	1000	6210/10.4	34024/17.1
Wilkinson	800	800	73.4/3.24	3244/10
Wilkinson	900	900	143/3.66	5402/12.5
Wilkinson	1000	1000	256/4.1	8284/15.1
Mignotte	300	4	0.12/1.75	803/7.7
Mignotte	400	4	0.22/1.77	3422/15.8
Mignotte	600	4	0.54/1.89	26245/49.1

The results given for random polynomials, Table 2, were averaged over sets of 5 random polynomials each, both methods were tested on the same sets of

randomly generated polynomials. When all coefficients were randomly generated integers CF was faster by factors between 1.5 and 5.

Table 2. Polynomials with randomly generated coefficients

Coefficients (bit length)	Degree	No. of roots (average)	CF $T(s)/M$ (MB)	REL $T(s)/M$ (MB)
10	500	3.6	0.78/2.2	1.66/2.81
10	1000	4.4	6.67/3.75	34.2/7.5
10	2000	5.6	215/11.4	562/22.8
1000	500	3.2	0.56/2.28	2.19/2.97
1000	1000	3.6	12.7/5.1	31.4/6.5
1000	2000	6.0	329/14.2	510/24.3

The case of monic polynomials, Table 3, with randomly generated large integer coefficients, at lower terms proved to be especially hard for REL. In this case CF was several thousand times faster. This is because such polynomials tend to have both very large and small roots, so an isolation method based on interval bisection starts with a very large interval, and needs to bisect it many times before it isolates the small roots. CF does not have this problem, because the size of its each "step" is based on an estimate of how far the next root is.

Table 3. Monic polynomials with randomly generated coefficients

Coefficients (bit length)	Degree	No. of roots (average)	CF $T(s)/M$ (MB)	REL $T(s)/M$ (MB)
10	500	5.2	1.43/2.48	8.84/3.84
10	1000	4.8	7.12/3.74	80.7/10.1
10	2000	6.8	263/11.4	1001/37.1
1000	100	4.4	0.01/1.75	56.8/5.5
1000	200	6.0	0.086/1.93	252/17
1000	500	5.6	0.57/2.28	1917/96.8
1000	1000	6.0	25.5/5.2	>5000/?

For polynomials with all roots being randomly generated integers, Table 4, CF was up to 25 times faster for small roots, but REL was up to 4 times faster for large roots. The latter being the only case when we found CF to be slower than REL.

Table 4. Products of factors (x-randomly generated integer root)

Coefficients (bit length)	Degree	No. of roots	CF $T(s)/M$ (MB)	REL $T(s)/M$ (MB)
10	100	100	0.8/1.82	0.61/1.92
10	200	200	2.45/2.07	10.1/2.64
10	500	500	33.9/3.34	878/8.4
1000	20	20	0.12/1.88	0.044/1.83
1000	50	50	16.7/3.18	4.27/2.86
1000	100	100	550/8.9	133/6.49

3 Conclusions

We have shown that our continued fraction root isolation algorithm CF is almost always faster than the algorithms based on interval bisection. Its bound on memory usage, given in terms of the number of transformed polynomials it needs to keep, is not much worse then for the algorithm REL presented in [3], and in practice its memory usage is often smaller than that of REL.

The link `http://members.wolfram.com/webMathematica/Users/adams/RootIsolation.jsp` gives access to both isolation methods with one caveat: the memory comparison does not work too well. Probably due to the fact that webMathematica is using a kernel shared by several users, if somebody had run a memory intensive computation before, MaxMemoryUsed will return the memory used by that computation, and will not change after our test examples.

References

1. G. E. Collins, A. G. Akritas. Polynomial real root isolation using Descartes' rule of signs, in: *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computations*, Yorktown Heights, N.Y., pp. 272–275, 1976.

2. A. J. H. Vincent. Sur la resolution des équations numériques, *Journal de Mathématiques Pures et Appliquées*, **1**, pp. 341–372, 1836.

3. F. Rouillier, P. Zimmermann. Efficient isolation of polynomial's real roots, *Journal of Computational and Applied Mathematics*, **162**, pp. 33–50, 2004.

4. A. G. Akritas, A. V. Bocharov, A. W. Strzeboński. Implementation of real root isolation algorithms, in Mathematica, in: *Abstracts of the International Conference*

on Interval and Computer-Algebraic Methods in Science and Engineering (Interval'94), St. Petersburg, Russia, March 7–10, pp. 23–27, 1994.

5. A. G. Akritas. An implementation of Vincent's Theorem, *Numerische Mathematik*, **36**, pp. 53–62, 1980.

6. A. G. Akritas. The fastest exact algorithms for the isolation of the real roots of a polynomial equation, *Computing*, **24**, pp. 299–313, 1980.

7. A. G. Akritas. Reflections on a pair of theorems by Budan and Fourier, *Mathematics Magazine*, **55**, pp. 292–298, 1982.

8. A. G. Akritas. *Elements of Computer Algebra with Applications*, Wiley, New York, NY, 1989. Available also in Russian, MIR Publishers, Moscow, 1994 (with new material).

9. J. von zur Gathen, J. Gerhard. Fast Algorithms for Taylor Shifts and Certain Difference Equations, in: *Proceedings of ISSAC'97, Maui, Hawaii, U.S.A.*, pp. 40–47, 1997.