INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient isolation of a polynomial real roots

Fabrice Rouillier — Paul Zimmermann

N° 4113

1st February 2001

THÈME 2

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4113--FR+ENG

Efficient isolation of a polynomial real roots

Fabrice Rouillier* , Paul Zimmermann †*

Thème 2 — Génie logiciel
et calcul symbolique
Projet SPACES, `www-spaces.lip6.fr`

Rapport de recherche n° 4113 — 1st February 2001 — 16 pages

Abstract: This paper gives new results for the isolation of real roots of a univariate polynomial using Descartes' rule of signs, following work of Vincent, Uspensky, Collins and Akritas, Johnson, Krandick. The first contribution is a generic algorithm which enables one to describe all the existing strategies in a unified framework.

Using that framework, a new algorithm is presented, which is optimal in terms of memory usage, while doing no more computations than other algorithms based on Descartes' rule of signs. We show that these critical optimizations have important consequences by proposing a full efficient solution for isolating the real roots of zero-dimensional polynomial systems.

Key-words: univariate polynomial, real root isolation, Descartes' rule of signs, Uspensky's algorithm, polynomial system

* INRIA Lorraine and LIP 6, `rouillie@loria.fr`
† INRIA Lorraine and LORIA, `zimmerma@loria.fr`
This work was partly supported by the Frisco LTR European project, and by the *Action de recherche coopérative* "Outils pour un calcul numérique fiable" from INRIA

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Isolation efficace des racines réelles d'un polynôme

Résumé : Cet article propose de nouveaux résultats pour l'isolation des racines réelles d'un polynôme univarié via la règle de Descartes, dans la lignée des travaux de Vincent, Uspensky, Collins et Akritas, Johnson, Krandick. Une première contribution est un algorithme générique permettant une description unifiée de toutes les stratégies connues.

À l'aide de ce cadre unifié, nous présentons un nouvel algorithme, optimal en terme d'espace mémoire utilisé, sans effectuer plus de calculs que les autres méthodes basées sur la règle de Descartes. Nous montrons la pertinance des améliorations proposées, qui fournissent une méthode complète et efficace pour l'isolation des racines réelles de systèmes polynomiaux de dimension zéro.

Mots-clés : polynôme univarié, isolation de racine réelle, règle de Descartes, algorithme d'Uspensky, système polynomial

Introduction

The core of this article deals with a well known method for isolating the real roots of univariate polynomials currently (wrongly - see [1]) named Uspensky's algorithm [14]. This name covers all algorithms based on Descartes' rule of sign, which bounds the number of positive real roots of a polynomial by the number of sign changes in its coefficients; Vincent in 1836 proved some kind of reciprocal of Descartes' rule [15], and in 1976 Collins and Akritas published the first effective version of the algorithm [2]. Vincent's theorem was then improved by Collins and Johnson in 1989 [3, 8], and Krandick invented a variant of Collins and Akritas' algorithm working better for polynomials with very near roots. Another famous class of algorithms to isolate the real roots of a polynomial are Sturm-like methods (see for example [13]) which essentially compute derivatives of the initial polynomial, whereas Uspensky-like methods compute translations of it.

The main contributions of that paper are (i) a unified description of all variants of methods based on Descartes' rule of sign; (ii) a memory- and time-optimal algorithm running as fast as the best from Collins and Akritas' algorithm and Krandick variant, independent from the input polynomial, and performing better than both methods in terms of memory usage.

The paper is organized as follows: Section 1 recalls Descartes' rule of signs and Collins and Akritas' algorithm. Section 2 first shows that the polynomials computed by Collins and Akritas' algorithm are exactly of the form $P(\frac{x+c}{2^k})$ (Lemma 2.1), then proposes a generic algorithm that enables to describe both Collins and Akritas' algorithm and Krandick's strategy in a unified framework. The main results of the paper are Propositions 3.3 and 3.4, which show how to compute the required polynomials one from another, using translations by one only. It follows a new algorithm which is optimal in terms of memory usage, and performs no more computation than other algorithms based on Descartes' rule of signs.

1. Descartes' rule of signs and related methods

In this section, we recall Descartes' rule of signs, which gives a bound on the number of positive real roots of a polynomial, and the algorithm from Collins and Akritas, which isolates all real roots of a polynomial lying in $]0,1]$. For the description of the algorithm, we consider that the coefficients are in $\mathbb{R}$; in Section 3 we will restrict to integer coefficients for efficient computations.

1.1. Descartes' rule of signs.

Descartes' rule of sign provides a very simple method to compute a bound on the number of positive roots of any univariate polynomial. It is based on the study of the sign variations in the polynomial coefficients.

Notation 1.1. *We denote by* $\mathrm{sign}(a)$, *the* sign *of an element* $a \in \mathbb{R}$, *as being* 0 *if* $a = 0$, 1 *if* $a > 0$ *and* -1 *if* $a < 0$, *and define the* number of sign changes $V(a)$ *in the list* $a = (a_1, \cdots, a_k)$ *of elements of* $\mathbb{R} \setminus \{0\}$ *by induction over* k:

$$V(a_1) = 0, V(a_1, \cdots, a_k) = \begin{cases} V(a_1, \cdots, a_{k-1}) + 1 & \textit{if } \mathrm{sign}(a_{k-1}a_k) = -1 \\ V(a_1, \cdots, a_{k-1}) & \textit{otherwise} \end{cases}$$

We extend this notation to a list a *of elements in* $\mathbb{R}$ *that may contain zeroes: if* b *is the list obtained by removing zeroes in* a, *we define* $V(a) = V(b)$.

Using the above notations, let us recall Descartes' rule of signs:

Theorem 1.1 (Descartes' rule of signs). *Let* $P = \sum_{i=0}^{d} a_i x^i$ *be a polynomial in* $\mathbb{R}[x]$. *If we denote by* $V(P)$ *the number of sign changes in the list* $(a_0, \ldots, a_d)$ *and* $\mathrm{pos}(P)$ *the number of positive real roots of* P *counted with multiplicities, then* $\mathrm{pos}(P) \leq V(P)$, *and* $V(P) - \mathrm{pos}(P)$ *is even.*

Obviously, the bound provided by Theorem 1.1 gives the exact number of positive real roots when it is equal to 0 or 1.

Let us consider the following transformations:

Notation 1.2. *Let* P *be a polynomial in* $\mathbb{R}[x]$. *We define:*

- $R(P(x)) = x^{\deg(P)} P(1/x)$,
- $H_c(P(x)) = P(cx)$ *for* $c \in \mathbb{R}$,
- $T_c(P(x)) = P(x + c)$ *for* $c \in \mathbb{R}$.

The simple transformation $H_{b-a}T_a$ maps the interval $]0,1[$ to the interval $]a,b[$ while the transformation T_1R maps the real positive axis to the interval $]0,1[$, so Theorem 1.1 applied to $T_1RH_{b-a}T_aP$ provides a bound on the number of real roots of P in $]a,b[$.

The basic idea of algorithms based on Descartes' rule of sign is to construct sufficiently small intervals $]a,b[$ with the hope that the above mentioned bound equals 0 or 1 for all such intervals.

1.2. **Collins and Akritas' algorithm.** Let us consider the case of the interval $]0,1[$. One can show [15] that if P has no real root in the interval $]0,1[$ and if P has no non-real root outside the circle of center $(1/2,0)$ and diameter 1, then $V(T_1R(P))=0$. Moreover, a result due to Collins and Johnson [3], stronger than Vincent's theorem [15], gives a good criterion for deciding if a polynomial has a unique root in the interval $]0,1[$:

Theorem 1.2. *Let P be a square-free real polynomial with a single root in the interval $]0,1[$ and no non-real root in each of the two circles of radius 1 and centers $(0,0)$ and $(1,0)$. Then $V(T_1R(P))=1$.*

Suppose that P is a polynomial in $\mathbb{R}[x]$ whose roots are bounded by $B\in\mathbb{R}$. By applying the transformation H_B we may suppose that P has all its positive roots in $]0,1]$.

Definition 1.3. *Let P be any polynomial of degree n in $\mathbb{R}[x]$, k and c non-negative integers such that $c<2^k$. We define $I_{k,c}=]\frac{c}{2^k},\frac{c+1}{2^k}]$ and $P_{k,c}(P)(x)=2^{kn}P(\frac{x+c}{2^k})$. The intervals $I_{k,c}$ are called standard intervals in* [9].

A naïve algorithm would consist in splitting the interval $]0,1]$ in small intervals, for example the intervals $I_{k,c}$ of Definition 1.3, and for each of them, to compute $P_{k,c}(P)(x)=P(\frac{x+c}{2^k})$ so that Descartes' rule of sign applied to $P_{k,c}(P)$ gives 0 or 1 in any case, which would provide all the isolating intervals in the form $I_{k,c}$, according to Theorem 1.2.

Obviously, it is not suitable to use a bound for k and to consider all $I_{k,c}$ with $0\le c<2^k$, since it would induce an exponential number of computations for isolating the roots. In [2], Collins and Akritas propose an algorithm that avoids this exponential behavior by introducing a bisection step. The following recursive algorithm describes this method. It has to be called with a square-free polynomial $P=C_{0,0}$ with all its roots in $]0,1]$ and the pair of integers $(k=0,c=0)$, with the lists `Isol` and `Exact` being empty initially:

<u>CollinsAkritas</u>

Input: A square-free polynomial $C_{k,c}$ of degree n, and a pair of integers (k,c).
Output: Two augmented lists `Isol` $:=[(k_i,c_i)]_{i=1\ldots s'}$ and `Exact` $:=[(k'_i,c'_i)]_{i=1\ldots u'}$ such that: if $\alpha>0$ and $C_{0,0}(\alpha)=0$ then there exists $(k',c')\in$ `Exact` such that $\alpha=c'/2^{k'}$ or there exists $(k',c')\in$ `Isol` such that $\alpha\in I_{k',c'}$. Moreover $\forall(k',c')\in$ `Isol`, $I_{k',c'}$ contains one and only one root of $C_{0,0}$.
Auxiliary functions

- `DescartesBound`(P) computes the bound given by Descartes' rule of sign for $T_1R(P)$.
- `addInList`$((k,c),List)$ adds (k,c) to the list $List$.

Begin

- $s:=$ `DescartesBound`$(C_{k,c})$
- if $s>0$ then
  - if $s=1$ then `addInList`$((k,c),$ `Isol`$)$
  - else
    - $C_{k+1,2c}=2^nH_{1/2}(C_{k,c})$
    - `CollinsAkritas`$(C_{k+1,2c},\ (k+1,2c))$
    - $C_{k+1,2c+1}=T_1(C_{k+1,2c})$
    - if $C_{k+1,2c+1}(0)=0$ then `addInList`$((k+1,2c+1),$ `Exact`$)$
    - `CollinsAkritas`$(C_{k+1,2c+1},\ (k+1,2c+1))$

End

1.3. **Krandick's variant.** Johnson [8] shows that in general this method is better, in terms of computation times, than other algorithms; Krandick improved the algorithm by introducing several specific data structures [9], proposed a different computation order to optimize the memory usage, and studied precisely its complexity. Clearly, Collins and Akritas' algorithm is a backtracking one; it needs to store at least $k_{\max}$

polynomials, where $k_{\max}$ denotes the maximal value of k in the lists `Isol` or `Exact`. Note that an approximate value of $k_{\max}$ is such that the distance between two roots of the given polynomial is greater than $1/2^{k_{\max}}$. By computing the polynomials $C_{k,c}$ level by level, Krandick [10] proposed a method that stores at most n polynomials, where n is the degree of the given polynomial; his method is obviously better when $k_{\max}$ is larger than n.

2. A unified framework

2.1. A unified description of methods based on Descartes' rule of signs.

In the first part we described a naïve algorithm based on Descartes' rule of sign with an exponential behavior, and we recalled Collins and Akritas' method, which has a polynomial complexity. Let us show that the polynomials computed in both cases — $P_{k,c}(P)$ and $C_{k,c}$ — are the same:

Lemma 2.1. *Let $P \in K[x]$ be a degree n polynomial. Then, for any pair of integers $(k,c), k \geq 0, 0 \leq c < 2^k$, $P_{k,c}(P) = C_{k,c}$, where $P_{k,c}(P)(x) = 2^{nk}P(\frac{x+c}{2^k})$ from Definition 1.3, and the $C_{k,c}$ are the polynomials computed by Collins and Akritas' algorithm.*

Proof. Proof We proceed by induction on k and c.

- for $k = c = 0$, $P_{0,0}(P) = C_{0,0} = P$.
- suppose that for fixed values of k and c we have $P_{k,c}(P)(x) = C_{k,c}(x)$. Since

$$C_{k+1,2c}(x) = 2^n H_{1/2}(C_{k,c}(x)) = 2^n H_{1/2}(P_{k,c}(P)(x)) = 2^n 2^{kn} P(\frac{x+2c}{2^{k+1}}) = P_{k+1,2c}(P)(x),$$

  the relation is proved for $(k+1, 2c)$. In the same way,

$$C_{k+1,2c+1}(x) = T_1(C_{k+1,2c}(x)) = T_1(P_{k+1,2c}(P)(x)) = P_{k+1,2c+1}(x)$$

  and so the relation is proved for $(k+1, 2c+1)$.

□

Thus using Descartes' rule of signs for providing an algorithm isolating the real roots of a square-free polynomial $P \in \mathbb{R}[x]$ consists first in computing all the pairs (k,c) such that $\texttt{DescartesBound}(P_{k,c}(P)) > 1$, and then to deduce the pairs (k,c) such that $\texttt{DescartesBound}(P_{k,c}(P))$ gives 0 or 1, taking care of the points in the form $c/2^k$ that are roots of P. Let us introduce some notations:

Definition 2.2. *Let P be a univariate polynomial of degree n, k and $c < 2^k$ two nonnegative integers. We define:*

- $\texttt{Internal}(P) = \{(k,c), k \geq 0, 0 \leq c < 2^k, P_{k,c}(P)(0) \neq 0, \texttt{DescartesBound}(P_{k,c}(P)) > 1\}$,
- $\texttt{Tree}(P) = \{(0,0)\} \cup \{(k,c), (k-1, \lfloor c/2 \rfloor) \in \texttt{Internal}(P)\}$,
- $\texttt{Exact}(P) = \{(k,c) \in \texttt{Tree}(P), P_{k,c}(P)(0) = 0\}$,
- $\texttt{Isol}(P) = \{(k,c) \in \texttt{Tree}(P), P_{k,c}(P)(0) \neq 0, \texttt{DescartesBound}(P_{k,c}(P)) = 1\}$,
- $\texttt{Lost}(P) = \{(k,c) \in \texttt{Tree}(P), P_{k,c}(P)(0) \neq 0, \texttt{DescartesBound}(P_{k,c}(P)) = 0\}$.

From the correctness of Collins and Akritas' algorithm, together with Lemma 2.1, it follows:

- If P is square-free, then $\texttt{Internal}(P)$ is finite,
- $\texttt{Tree}(P) = \texttt{Internal}(P) \cup \texttt{Exact}(P) \cup \texttt{Isol}(P) \cup \texttt{Lost}(P)$,
- α is a positive root of P if and only if $(\exists (k,c) \in \texttt{Isol}(P), \alpha \in I_{k,c})$ or $(\exists (k,c) \in \texttt{Exact}(P), \alpha = c/2^k)$.

Thus, whatever the strategy used, isolating the real roots of P using Uspensky's method consists in computing $\texttt{Tree}(P)$. The differences between the implementations are:

1. the method for computing the polynomials $P_{k,c}(P)$,
2. the order of traversal of the tree: depth-first for Collins and Akritas' algorithm, breadth-first for Krandick's algorithm (but the nodes to check, which are exactly those of $\texttt{Tree}(P)$, do not depend on the order),
3. the way the nodes of $\texttt{Tree}(P)$ are stored. In Collins and Akritas' algorithm, a node is represented both by a pair (k,c) and the corresponding polynomial $P_{k,c}(P)$, but we may store (k,c) only and compute $P_{k,c}(P)$ on demand, using Lemma 2.1.

The computation time of the resulting implementation depends mostly on item 1, while the memory usage depends on items 2 and 3.

According to item 2, we need to fix an order of traversal of $\texttt{Tree}(P)$. Such an order can obviously be deduced from a total ordering over $\mathbb{N}^2$ — to order the pairs (k, c) — and will be a parameter of our generic algorithm, together with the following functions:

- $\texttt{initializeTree}(P)$ initializes at level $k = 0$ the set T representing the working subset of $\texttt{Tree}(P)$.
- $(k, c, Q) \leftarrow \texttt{getNewNode}(T, <)$ takes the lowest node (k, c) of T with respect to the ordering $<$, removes that node from T, and sets Q to $P_{k,c}(P)$.
- $(E, T) \leftarrow \texttt{addSuccessors}((k, c), Q, E, T)$ adds to T the successor nodes of (k, c), say $(k + 1, 2c)$ and $(k+1, 2c+1)$, taking care of the case where $Q(1/2) = 0$; in this case, the node $(k+1, 2c+1)$ is added to the set E.

Different implementations of these functions will lead to different strategies for computing or storing the final or intermediate results. Thus we consider them as parameters of the generic algorithm:

GenericDescartes

Input A square-free polynomial P with all its positive roots in $]0, 1]$, three functions `initializeTree`, `getNewNode` and `addSuccessors` and an ordering $<$ over $\mathbb{N}^2$.

Output The lists $E = \texttt{Exact}(P)$ and $I = \texttt{Isol}(P)$.

Auxiliary function: $\texttt{DescartesBound}(P)$ using Descartes' rule of sign for $T_1R(P)$.

Begin

- $E \leftarrow \emptyset, \quad I \leftarrow \emptyset, \quad T \leftarrow \texttt{initializeTree}(P)$
- while $T \neq \emptyset$ do
  - $(k, c, Q) \leftarrow \texttt{getNewNode}(T, <)$
  - $s \leftarrow \texttt{DescartesBound}(Q)$
  - if $s = 1$ then $I \leftarrow I \cup \{(k, c)\}$
  - if $s > 1$ then $(E, T) \leftarrow \texttt{addSuccessors}((k, c), Q, E, T)$

End

2.2. **Some strategies as a specialization of the Generic algorithm.** In this section it is shown that both Collins and Akritas' algorithm [2] and Krandick's strategy [10] can be described in terms of the generic algorithm, using the *same* functions `initializeTree`, `getNewNode` and `addSuccessors`, the only difference being the ordering used. We then deduce a new and simple algorithm which is optimal in terms of memory usage. In the rest of the paper, $P \in \mathbb{R}[x]$ is a square-free polynomial with all its roots in $]0, 1]$.

2.2.1. *Collins and Akritas' algorithm.* Collins and Akritas' algorithm consists in computing recursively the polynomials $P_{k,c}(P)$ for $(k, c) \in \texttt{Tree}(P)$ using a depth-first strategy. By introducing the following ordering over $\mathbb{N}^2$:

$$(k, c) <_{back} (k', c') \iff (\frac{c}{2^k} < \frac{c'}{2^{k'}}) \quad \text{or} \quad ((\frac{c}{2^k} = \frac{c'}{2^{k'}}) \text{ and } (k < k')),$$

we can easily show that this consists in computing the polynomials $P_{k,c}(P)$ in increasing order with respect to $<_{back}$.

With Collins and Akritas' strategy, the polynomial $P_{k,c}(P)$ is needed for testing the nodes $(k+1, 2c)$ and $(k + 1, 2c + 1)$ that require the computation of $P_{k+1,2c}(P)$ and $P_{k+1,2c+1}(P)$, so that we have to represent $\texttt{Tree}(P)$ by a list of elements of the form $(k, c, P_{k,c}(P))$.

We extend canonically the ordering used by taking:

$$(k, c, Q) <_{back} (k', c', Q') \iff (k, c) <_{back} (k', c').$$

Thus the specialization

$$\texttt{GenericDescartes}(\texttt{initializeTreeClassic}, \texttt{getNewNodeClassic}, \texttt{addSuccessorsClassic}, <_{back})$$

implements Collins and Akritas' algorithm with the following set of functions:

initializeTreeClassic

Begin
- $T \leftarrow \{(0, 0, P)\}$;

End

Since all polynomials $P_{k,c}(P)$ are stored in the set T, the function `getNewNode` is trivial:

getNewNodeClassic

Begin
- $(k, c, Q) \leftarrow \min_{<_{back}}(T)$
- remove (k, c, Q) from T

End

addSuccessorsClassic

Begin
- Compute $L = 2^n H_{1/2}(Q)$;
- Compute $R = T_1(L)$;
- if $R(0) = 0$ then $E \leftarrow E \cup \{(k+1, 2c+1)\}$
- $T \leftarrow T \cup \{(k+1, 2c, L), (k+1, 2c+1, R)\}$

End

2.2.2. *Krandick's algorithm.* In [10], Krandick proposed to improve Collins and Akritas' algorithm by computing the polynomials level by level instead of using a depth-first strategy. According to our definitions, Krandick's algorithm is defined as:

`GenericDescartes(initializeTreeClassic, getNewNodeClassic, addSuccessorsClassic,` $<_{lev}$`)`

where the ordering $<_{lev}$ over $\mathbb{N}^2$ can be defined as follows:

$$(k, c) <_{lev} (k', c') \iff (k < k') \quad \text{or} \quad ((k = k') \text{ and } (c < c')).$$

In both situations we can remark that `Tree`(P) is never fully stored. As noticed by Krandick [10], in Collins and Akritas' algorithm, the number of stored elements of `Tree`(P) is $O(k_{\max})$ where $1/2^{k_{\max}}$ denotes the minimal interval length in the final result, while in Krandick's algorithm it is $O(n)$ where n denotes the degree of the initial polynomial.

2.2.3. *A simple memory-optimal algorithm.* According to Definition 1.3 and Lemma 2.1, all the polynomials needed during the traversal of `Tree`(P) can be deduced from the initial polynomial P. We can thus represent `Tree`(P) simply by a list of pairs (k, c) and customize the functions `initializeTree`, `getNewNode` and `addSuccessors` as follows to get a naïve but memory-optimal algorithm:

initializeTreeConst

Begin
- $T \leftarrow \{(0, 0)\}$;

End

The polynomials $P_{k,c}(P)$ are not stored in the representation of `Tree`(P), but are computed at demand by the function `getNewNode`:

getNewNodeConst

Begin
- $(k, c) \leftarrow \min_{<}(T)$
- remove (k, c) from T
- $Q \leftarrow 2^{nk} T_c H_{1/2^k}(P)$

End

Due to the simple structure of T, the function `addSuccessors` is very short:

addSuccessorsConst

Begin
- if $Q(1/2) = 0$ then $E \leftarrow E \cup \{(k+1, 2c+1)\}$
- $T \leftarrow T \cup \{(k+1, 2c), (k+1, 2c+1)\}$

End

Whatever the ordering $<$ used, the algorithm

$$\texttt{GenericDescartes}(\texttt{initializeTreeConst}, \texttt{getNewNodeConst}, \texttt{addSuccessorsConst}, <)$$

is optimal in terms of memory usage, due to the way the nodes of $\texttt{Tree}(P)$ are stored (only one pair of integers for each node). Only two polynomials need to be stored: the initial polynomial P and the current one $Q = P_{k,c}(P)$.

This naïve algorithm is less efficient in terms of computation time than Collins and Akritas' one due to the method used for computing the polynomials $P_{k,c}$. Our goal in the next section is to propose another way of computing the polynomials $P_{k,c}$, with no overhead in terms of computation time, while keeping optimality in terms of memory.

3. Towards an optimal algorithm

In the above sections, we considered two well known algorithms (Collins and Akritas' method and Krandick's variant) based on Descartes' rule of sign. Both strategies have advantages and drawbacks in terms of memory usage (Collins and Akritas' method is better when $k_{\max}$ is less than the number of real roots, while Krandick's strategy is better in the other case) but have the same behavior in terms of computation time. We also proposed a naïve memory-optimal algorithm. However, none of these three methods is optimal both in terms of time and memory usage.

With the unified framework we proposed, one could easily switch from one strategy to another one by simply changing the order used during the computation; however we will not develop this idea further, since the algorithm described in the second part of this section will be shown to be optimal for both time and memory.

In the first part we show how to perform efficiently the two most time-consuming part of the generic algorithm: the translation $P(x+c)$ and computing Descartes' bound. From now on, we assume the coefficients of the polynomials are integers.

3.1. Efficient translation and computation of Descartes' bound.

Whatever the strategy used, the main computation in the algorithm `GenericDescartes` is that of $P_{k,c}$, which consists in applying an homothetic transformation of the form H_{2^l} and a translation of the form T_d. Obviously, the translation is the most expensive operation, since H_{2^l} requires only $O(n)$ arithmetic operations. When $d = 1$, several authors — for example in [10] — propose solutions for doing the translation $T_1(P)$ using $O(n^2)$ additions — asymptotically faster methods are also given in [4] — where n is the degree of the given polynomial. For example, the following transformation uses $O(n^2)$ additions and multiplications for computing $T_d(P)$ and only $O(n^2)$ additions when $d = 1$:

> Let $P(x) = \sum_{i=0}^{n} a_i x^{n-i}$ be a polynomial, and $h_j(x) = \sum_{i=0}^{j} a_i x^{j-i}$ be the j-th ($0 \le j \le n$) Horner's polynomial associated to P (note that $P = h_n$). For a scalar c and an integer $0 < j \le n$, we have:
>
> $$h_j(x+c) = xh_{j-1}(x+c) + ch_{j-1}(x+c) + a_j.$$

This formula gives a simple algorithm for computing efficiently $T_c(P)(x) = h_n(x+c)$. If $P[k]$ denotes the k-th coefficient of a polynomial P, an algorithm that computes destructively $T_c(P)$ can be described by:

`Translate`

Input A polynomial P of degree n represented by its coefficients $P[0], \ldots, P[n]$, and a scalar c.
Action Replace the coefficients of P by those of $P(x+c)$.
- **Begin**
  - for $i = 1$ to n for $k = n - i$ to $n - 1$ $P[k] = P[k] + cP[k+1]$;
- **End**

The particular case where $c = 1$ is intensively used by `DescartesBound`(Q) which applies Descartes' rule of sign on the polynomial $T_1R(Q)$. The fact that T_1 uses only additions leads to an important optimization that prevents from computing fully $T_1R(Q)$ — without loosing information — by testing if the coefficients that remain to be added all have the same sign than the last coefficient computed (in such a case, no additional variation of sign will appear):

<u>`DescartesBound`</u>

Input A polynomial Q of degree n.
Output The result of Descartes' Rule of sign for $T_1R(Q)$.
Begin

1. for $i = 1$ to n
   - for $k = i$ downto 1 do
     - $Q[k] = Q[k] + Q[k-1]$;
   - $s_1 \leftarrow$ number of sign changes in $Q[0], \dots, Q[i-1]$
   - if $s_1 = 0$ then goto 2
2. $s_2 \leftarrow$ number of sign changes in $Q[i-1], \dots, Q[n]$
3. return(s_2)

End

Remark 3.1. *We may also stop the computation as soon as $s_2 > 1$.*

Remark 3.2. *A different way of performing a translation by an integer c consists in writing $P(x+c) = P(c(x/c+1)) = H_{1/c}T_1H_c(P)(x)$. It may be faster than* `Translate` *since we may use fast algorithms for multiplying or dividing by c^i. Another advantage is that we need to write a translation procedure by one only, which can use fast algorithms too* [4].

3.2. **A memory- and time-optimal algorithm.** As seen in the above section, the cost of the operation T_c depends strongly on the integer c which must be as small as possible. Obviously, one has to consider in priority small values of c (0: nothing to compute, 1: an algorithm that uses only additions). This explains the overhead in terms of computation time between

$$\texttt{GenericDescartes}(\texttt{initializeTreeConst}, \texttt{getNewNodeConst}, \texttt{addSuccessorsConst}, <)$$

and

$$\texttt{GenericDescartes}(\texttt{initializeTreeClassic}, \texttt{getNewNodeClassic}, \texttt{addSuccessorsClassic}, <).$$

The idea now is to take care, at each step in the algorithm, of the last polynomial before computing a new one. In other words, one has to study the operations needed to compute $P_{k,c}$ from $P_{k',c'}$.

Proposition 3.3. *Let $P \in K[x]$ be a polynomial. Let (k,c) and (k',c') be two pairs of integers such that $k \geq 0, 0 \leq c < 2^k, k' \geq 0, 0 \leq c' < 2^{k'}$. Then using the notations of Lemma 2.1, we have:*

$$P_{k',c'} = 2^{n(k'-k)}H_{2^{k-k'}}T_{2^{k-k'}c'-c}(P_{k,c}),$$

and conversely:

$$P_{k,c} = 2^{n(k-k')}T_{c-2^{k-k'}c'}H_{2^{k'-k}}(P_{k',c'}).$$

Proof. Proof $P_{k',c'}(x)$ being defined as $2^{nk'}P(\frac{x+c'}{2^{k'}})$, we have:

$$P_{k',c'}(x) = 2^{nk'}P(\frac{2^{k-k'}x + 2^{k-k'}c' + c - c}{2^k}),$$

and therefore

$$P_{k',c'} = 2^{n(k'-k)}H_{2^{k-k'}}T_{2^{k-k'}c'-c}(P_{k,c}).$$

The second equation follows from inverting the first one, and using the equalities $T_c^{(-1)} = T_{-c}$ and $H_c^{(-1)} = H_{1/c}$. □

These formulae provide a new function `getNewNodeRel` for computing $P_{k,c}$ in algorithm `GenericDescartes`, taking advantage of the previous computations:

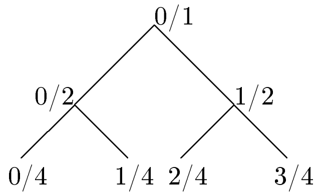

FIGURE 1. Successor nodes $c/2^k$ in depth-first traversal: starting from $0/1$, the node $0/2$ follows with $H_{1/2}$, then $0/4$ using $H_{1/2}$ again, then $1/4$ using T_1, then $1/2$ with H_2T_1, followed by $2/4$ using $H_{1/2}$, and finally $3/4$ using T_1.

getNewNodeRel

Begin
- let (k, c) be the previous node, and $Q = P_{k,c}$ the corresponding polynomial
- $(k', c') \leftarrow \min_<(T)$
- remove (k', c') from T
- $Q \leftarrow 2^{n(k'-k)} H_{2^{k-k'}} T_{2^{k-k'}c'-c}(Q)$

End

The algorithm

$$\texttt{GenericDescartes}(\texttt{initializeTreeConst}, \texttt{getNewNodeRel}, \texttt{addSuccessorsConst}, <),$$

is optimal — for any ordering $<$ — in terms of memory usage, since only one polynomial has to be stored if we replace $P_{k,c}$ in place by $P_{k',c'}$, as allowed by `Translate`.

We can optimize the computation time by choosing at each step in the algorithm the node that induces a translation by the smallest integer (by extending the ordering). But the following proposition shows that if we choose $<_{back}$ as ordering, then the resulting algorithm is optimal in terms of computation time, i.e. has the same practical behavior than

$$\texttt{GenericDescartes}(\texttt{initializeTreeClassic}, \texttt{getNewNodeClassic}, \texttt{addSuccessorsClassic}, <_{back}).$$

Proposition 3.4. *In the algorithm*

$$\texttt{GenericDescartes}(\texttt{initializeTreeConst}, \texttt{getNewNodeRel}, \texttt{addSuccessorsConst}, <_{back}),$$

all translations involved in the procedure `getNewNodeRel` *are either* T_0 *or* T_1*. In other words, if* (k, c) *and* (k', c') *are two consecutive nodes in the ordered list — with respect to* $<_{back}$ — $[(k_i, c_i)]_{i=0\ldots s}$ *that represents* `Tree`(P)*, then the expression* $2^{k-k'}c' - c$ *equals either* 0 *or* 1.

Proof. Proof This proposition comes from a general property of binary trees (see Figure 1). In a depth-first traversal of such a tree, the successor of a node n is:
- its left son if n is an internal node;
- otherwise the right brother of n's nearest parent which is itself a left son.

In the first case we have $k' = k+1$ and $c' = 2c$, therefore $2^{k-k'}c' - c = 0$; in the second case, the nearest parent which is itself a left son is of the form $k'' = k - i$, $c'' = (c+1-2^i)/2^i$. The successor of (k, c) is the right brother of (k'', c'') i.e. $k' = k'' = k - i$, $c' = c'' + 1 = (c+1)/2^i$. We then have $2^{k-k'}c' - c = 2^i(c+1)/2^i - c = 1$. □

The algorithm

$$\texttt{GenericDescartes}(\texttt{initializeTreeConst}, \texttt{getNewNodeRel}, \texttt{addSuccessorsConst}, <_{back})$$

computes exactly the same polynomials than Collins and Akritas' or Krandick's methods, performing the same number of homothetic transformations of the form $H_{1/2^k}$ and the same number of translations of the form T_1. Moreover, the representation of `Tree`(P) requires only the storage of the studied intervals and of the current polynomial $P_{k,c}$.

3.3. **Complexity.** The full study of the complexity of classical versions of the algorithm (Collins-Akritas and Krandick) can be found in [10].

3.3.1. *Theoretical complexity.* In this part, we give or recall the theoretical complexity (in time and memory) of the algorithms:

- `[Col]` `GenericDescartes(` `initializeTreeClassic`, `getNewNodeClassic`, `addSuccessorsClassic`, $<_{back}$)
- `[Kra]` `GenericDescartes(` `initializeTreeClassic`, `getNewNodeClassic`, `addSuccessorsClassic`, $<_{lev}$)
- `[Rel]` `GenericDescartes(` `initializeTreeConst`, `getNewNodeRel`, `addSuccessorsConst`, $<_{back}$) .

Proposition 3.5. *Let P be a polynomial of degree n with integer coefficients of size t. Any algorithm among* `[Col]`, `[Kra]` *and* `[Rel]` *needs $O(n^6 t^2 log_2^2(n))$ binary operations to isolate all real roots of P, and the corresponding memory usage is:*

- *$O(n^4 t^2 log_2^2(n))$ for algorithm [Col],*
- *$O(n^4 t log_2(n))$ for algorithm [Kra],*
- *$O(n^3 t log_2(n))$ for algorithm [Rel].*

Proof. Proof : Let us firstly compute the size[1] of the coefficients in the polynomials that appear during the computations. According to Lemma 2.1, whatever the algorithm used, these polynomials are $P_{k,c} = 2^{kn}P(\frac{x+c}{2^k}) = T_c(2^{kn}H_{1/2^k}(P))$. The size of the coefficient of degree i in $2^{kn}H_{1/2^k}(P)$ is at most $t+(n-i)k$. By using algorithm `Translate`, and the fact that $c < 2^k$, one can deduce that the size of coefficients in $P_{k,c}$ is bounded by $t + nk + n = O(t + nk)$.

The height of $\texttt{Tree}(P)$ is $O(n \log d + n \log n)$, where d is the euclidean norm of P (see [10]), thus the height $k_{\max} = O(nt \log n)$. It follows that the maximum coefficient size in any polynomial during the computations is $O(n^2 t \log n)$. According to Proposition 3.3, the only transformations used in the algorithms [Col], [Kra] and [Rel] are $H_{1/2^k}$, H_{2^k} and T_1. The transformations $H_.$ are linear in the size of $P_{k,c}$, and, according to algorithm `Translate`, the transformation T_1 needs $O(n^2)$ additions, so that whatever the chosen algorithm, the computation of any polynomial $P_{k,c}$ uses $O(n^4 t \log n)$ bit operations. The width of $\texttt{Tree}(P)$ is at most n so that the total number of nodes to be computed is $O(n^2 t \log n)$, then all polynomials are computed in $O(n^6 t^2 \log^2 n)$ bit operations. Algorithm [Col] stores a number of polynomials equal to the height of the tree, i.e. uses $O(n^4 t^2 \log^2 n)$ bits. Algorithm [Kra] stores a number of polynomials equal to the width of the tree, i.e. uses $O(n^4 t \log n)$ bits. Algorithm [Rel] stores only $O(1)$ polynomials, and therefore uses $O(n^3 t \log n)$ bits. □

3.3.2. *Practical complexity.* Let us study more precisely the transformations involved in `[Rel]`, which differ from those involved in `[Col]` when two successive nodes $(k,c) <_{back} (k',c')$ satisfy $k > k'$. Even if it doesn't affect the theoretical complexity, this transformation may induce, in practice, an overhead in algorithm `[Rel]`, compared with the transformation used in algorithm `[Col]` because of the sizes of the involved coefficients.

In such a situation, necessarily $c' \bmod 2 = c \bmod 2 = 1$, so that in algorithm `[Col]`, $P_{k',c'}$ is obtained using the identity $P_{k',c'} = T_1(P_{k',c'-1})$, while in algorithm `[Rel]`, it is obtained by using the formula of Proposition 3.3, which consists in computing $P_{k',c'} = 1/2^{nh} H_{2^h} T_1(P_{k,c})$ with $h = k - k'$. Since $P_{k',c'}$ has integer coefficients, then the coefficients of $H_{2^h}T_1(P_{k,c})$ are multiples of 2^{nh}.

We now propose an optimization for the computation of $P_{k',c'}$ in such situations, using the fact that all the coefficients of $H_{2^h}T_1(P_{k,c})$ are multiples of 2^{nh}. Since $P_{k',c'}(x) = 2^{-nh}P_{k,c}(2^h x+1) = 2^{-nh}P_{k,c}(2^h(x+2^{-h}))$, we can compute $P_{k',c'}$ using the following function:

HTranslate(P,h)

Input: $P(x)$ a polynomial of degree n, $P[j]$ its coefficient of degree j, h a positive integer.
Output: the coefficients of $2^{-h}H_{2^h}T_1(P)$, known to be integers

- $\mu = \lceil n/2^h \rceil + h + 2$
- **for** i **from** 0 **to** n **do** $P^{(0)}[i] = \lfloor P[i] 2^{\mu - h(n-i)} \rfloor$
- **for** i **from** 1 **to** n **do for** j **from** $n-i$ **to** $n-1$ **do** $P^{(i)}[j] = P^{(i-1)}[j] + \lfloor P^{(i-1)}[j+1] 2^{-h} \rfloor$
- **for** i **from** 0 **to** n **do** $P^{(n)}[i] = \lfloor P^{(n)}[i] 2^{-\mu} \rfloor$

[1] We consider here as size the number of bits of the coefficients.

Let us denote by e_i, $i = 0 \dots n$, the error done when using the truncated result of the divisions instead of their exact result, say $e_i = \max_{j=0\dots n} |\tilde{P}^{(i)}[j] - P^{(i)}[j]|$, where $\tilde{P}^{(i)}[j]$ would be the values obtained in `HTranslate` without taking the integer parts. Obviously, $e_0 < 1$ and $e_i \leq e_{i-1} + e_{i-1} 2^{-h} + 1, i = 1 \dots n$. It follows that for $i = 1 \dots n$, $e_i < (1 + 2^{-h})^i (1 + 2^h)$ so that $e_n < (1 + 2^{-h})^n (1 + 2^h)$. The last step of `HTranslate` will compute exactly the polynomial $P_{k',c'} = 2^{-\mu}(2^{\mu} P^{(n)}) = P^{(n)}$ if $\mu \geq \log_2((1 + 2^{-h})^n (1 + 2^h))$; the value $\mu = \lceil n/2^h \rceil + h + 2$ works.

4. Experiments

We consider here two kinds of polynomials. The first set contains classical examples of univariate polynomials known to be difficult to solve and currently used in benchmarks. The second set of examples comes from test-suites in polynomial system solving. As shown below, the isolation of the real roots of univariate polynomials can be viewed as a step in the full resolution of zero-dimensional systems. We will show that our method is well designed for this purpose.

4.1. **Isolation of the real roots of univariate polynomials.** We have tested 4 variants of Descartes' rule of sign based algorithms: SACLIB2.1 is a C implementation of Krandick's version in the SACLIB library, REL, COL and KRA are three variants of the generic algorithm described in this paper. They have been implemented in C using the GNU MP package for the multiprecision arithmetic [5]. We measured:

- T1, the computation time (in seconds, including the system time) obtained when using an optimized mecanism for the memory management,
- T2, the computation time (in seconds, including the system time) obtained when using a naïve mecanism for the memory management,
- M is the amount of memory (in mega bytes) used for storing the computation tree (the measured binary size of the computation tree). It is important to note that this represents much less memory than for the full algorithm since we do not take into account the intermediate computations and extra polynomial copies. The symbol `DA` means that the default heap size has been used.

For the algorithm `SACLIB2.1` we measured:

- T the computation time (including the system time),
- M the total amount of memory used by the program (according to the remark above, this can not be directly compared with the value observed for the algorithms REL, COL, and KRA).

The timings have been obtained on a 366 Mhz Pentium-III with 256 Mb of main memory, running under Linux.

Example	REL			COL			KRA			SACLIB2.1	
	T1	T2	M	T1	T2	M	T1	T2	M	T	M
Laguerre 100	0.58	0.7	0.01	0.58	0.82	0.06	0.60	1.80	0.27	1.66	DA
Laguerre 200	6.54	6.99	0.05	6.66	7.76	0.27	6.64	18.23	1.97	28.88	DA
Laguerre 300	29.94	31.98	0.12	30.46	34.52	0.75	30.79	77.23	6.91	150.49	27
Laguerre 400	91.19	95.68	0.21	92.31	100.07	1.30	92.97	212.38	16.05	490.13	62
Chebychev 100	0.39	0.44	0.01	0.40	0.64	0.065	0.40	0.84	0.08	1.32	DA
Chebychev 200	4.5	4.62	0.02	4.41	5.40	0.29	4.34	8.22	0.58	22.76	DA
Chebychev 300	22.3	23.05	0.05	21.34	24.90	0.71	21.64	40.86	2.64	123.56	12
Chebychev 400	65.04	68.12	0.09	63.40	70.08	1.39	63.13	104.82	4.71	413.54	20
Chebychev 500	169.63	176.57	0.15	166.21	176.96	2.42	167.00	272.63	12.78	1067.21	58
Wilkinson 100	0.54	0.66	0.01	0.54	0.78	0.07	0.55	1.89	0.42	2.19	DA
Wilkinson 200	6.54	6.68	0.05	6.31	7.39	0.3	6.39	19.74	3.11	31.81	DA
Wilkinson 300	28.02	29.83	0.12	28.22	32.14	0.76	28.40	80.58	10.60	167.90	40
Wilkinson 400	86.55	90.04	0.22	86.13	93.60	1.412	86.95	221.83	25.18	542.70	115
Wilkinson 500	219.07	222.44	0.34	214.48	223.88	2.29	215.80	505.34	50.00	1383.02	192
Mignotte 100	3.33	3.38	0.04	3.20	8.04	2.48	3.21	3.36	0.29	17.79	DA
Mignotte 200	127.92	122.96	0.30	124.00	193.70	36.18	123.00	121.75	2.323	626.55	DA

Memory usage: We can observe that the amount of memory needed in algorithm REL for storing the computation tree never exceeds 1 MB while it can be more than 50 MB with other strategies (which may induce 200 MB for the full process). As a consequence, the other algorithms may not terminate on huge examples due to exhausted memory. (We used a computer with 256 MB of memory.)

We can also observe that the strategy used for the memory management has almost no influence on algorithm REL, while it seriously affects the computation times in the other cases (compare T2 to T1).

In this test suite, there are two classes of examples:

- Mignotte polynomials for which the height of the computation tree is large while its width is small,
- Laguerre, Chebychev and Willkinson polynomials for which the height of the computation tree is small while its width is large.

As expected, algorithm KRA works better on the first class of examples, COL works better on the second class of examples; REL has a uniform good behavior.

Computation times: The computation times of all three algorithms REL, COL, and LEV are very similar when using an optimized memory management. In order to show that the algorithms were carefully implemented and optimized, we have compared them with the most efficient implementation we know (in a recent experimental version of the SACLIB library).

In the absolute, our implementations are faster, but the behavior of the different functions is similar. The difference of efficiency may be due to the use, in our code, of the multiprecision package (GNU MP), a dedicated memory management and to some additional tricks not described in this paper but allowing to decrease, in practice, the number of nodes to be visited.

4.2. **Application to polynomial system solving.** Our second set of tests deals with the resolution of zero-dimensional systems. Let $\{f_1(x_1, \ldots, x_n) = 0, \ldots, f_s(x_1, \ldots, x_n) = 0\}$ be a set of algebraic equations defined by polynomials of $\mathbb{Q}[X_1, \ldots, X_n]$. An efficient way of solving such an algebraic system is to compute a Rational Univariate Representation (RUR) of the solutions [12], that is to say an equivalent system of the form:

$$\begin{cases} f(t) = 0 \\ x_1 = g_1(t)/g(t) \\ \vdots \\ x_n = g_n(t)/g(t) \end{cases},$$

where $f, g, g_1, \ldots, g_n$ are univariate polynomials with coefficients in $\mathbb{Q}$.

Once such a representation of the system is computed, the problem reduces to the study of univariate polynomials. In particular, the number of real roots of the initial system exactly matches that of f. Moreover, by plugging an approximation of the roots of f in the rational functions g_i/g, one obviously obtains an approximation of all the solutions (f and g have no common root).

Since the RUR is computed using Gröbner bases, the size of the coefficients that appear in the various polynomials may be huge so that the study of f may be a hard task. Due to recent progress in the computation of Gröbner bases [7], the computation of the Rational Univariate Representation is possible for rather large examples so that the resolution of the univariate polynomial f may now become the blocking step in the full process. In this case, algorithm REL becomes very interesting due to the size of the initial polynomials.

Only few methods exist for isolating all the real roots of univariate polynomials using exact computations and they are mostly based on Descartes' rule of sign or Sturm-like methods like Sturm-Habicht sequences (a good overview of the recent improvements on these methods can be found in [13]). The main difference between these two sets of methods is that the Sturm-based algorithms can deal with polynomials over a non archimedean field, which is not the case of the algorithms described in this paper.

In Figures 2 and 3 we compare three softwares for counting the real roots of a univariate polynomial with integer coefficients (MPSolve based on Aberth's method, the PoSSo library implementing Sturm-Habicht algorithm, and RS using Uspensky's algorithm). If we except timings below one second, it appears that Uspensky's algorithm is always faster than Sturm-Habicht method, and faster than Aberth's method in all cases where that one guarantees the number of real roots, except for the `cyclic7` and `katsura9` examples. Even in several cases where Aberth's method cannot guarantee the number of real roots, Uspensky's algorithm is still faster (`cpdm5`, `ducos7_3` for example).

The univariate polynomials were obtained using Gb — a software implemented by J.C. Faugère, devoted to Gröbner bases computations — and RS — a software implemented by F. Rouillier, devoted to the study of the real roots of polynomial systems. Both software are written in C/C++ and are using GNU MP as arithmetic package.

Example	MPS	?	OK	ST	USP	TOT	NB	Deg	min	max
`Fabrice24`	**0.06**	0	24	155.86	0.11	39.26	24	40	2230	2265
`boon`	0	4	4	**0**	**0**	0.04	8	8	61	97
`camera1s`	0	*20*	0	6.21	**0.04**	7.24	16	20	1522	1559
`caprasse`	0.13	0	18	0.08	**0.02**	0.17	18	32	1	192
`cassou`	0	*16*	0	0.02	**0.01**	0.11	4	16	49	125
`conform1`	0	*16*	0	**0**	**0**	0.01	0	16	25	72
`cpdm5`	36.79	*55*	38	3007.79	**0.51**	64.79	43	163	143	851
`cyclic5`	0.02	*70*	0	0.36	**0.02**	0.26	10	70	1	265
`cyclic6`	0.1	*126*	10	35.56	**0.33**	3.6	24	156	1	1230
`cyclic7`	**26.5**	0	56	-	235.56	1669.59	56	924	57	5725
`d1`	**0.07**	0	16	78.24	0.16	21.81	16	48	2032	2091
`delta2`	0.05	*16*	4	1489.3	**0.36**	162.99	8	40	7968	8425
`des18_3`	0.03	0	6	10.86	**0.02**	2.53	6	46	285	423
`des22_24`	0.01	*23*	5	5.86	**0.03**	3.5	10	42	262	328
`ducos7_3`	9.52	0	168	1239.23	**2.48**	216.44	168	168	1	1017
`eco5`	0	*8*	0	**0**	0.01	0.03	4	8	1	17
`eco6`	0	*10*	2	**0**	**0**	0.04	4	16	1	38
`eco7`	0.01	*32*	0	0.09	**0**	0.15	8	32	1	83
`eco8`	0.01	*64*	0	2.91	**0.01**	1.09	8	64	1	180
`geneig`	0	9	1	**0**	**0**	0.13	10	10	16	21
`heart`	0	*4*	0	**0**	**0**	0.06	2	4	446	450
`hexa2`	0.05	*15*	7	1722	**0.43**	200.14	8	40	8707	9147
`hexaglide`	**0.03**	0	8	108.76	0.12	8.15	8	36	3906	4271
`i1`	1.1	*2*	16	-	**1.3**	523.34	16	66	5962	6147
`ilias_k_2`	6.88	*28*	0	463.96	**0.09**	705.5	14	72	359	897
`ilias_k_3`	9.02	*36*	0	684.01	**0.1**	1028.78	18	84	311	557
`ingersoll`	0.02	*34*	4	357.01	**0.11**	41.16	6	40	3490	3890
`kanukli4`	0.01	*12*	8	1.51	**0.01**	0.6	8	32	380	699
`katsura5`	0.08	0	12	0.33	**0.01**	0.12	12	32	69	110
`katsura6`	0.17	0	32	14.18	**0.07**	1.12	32	64	68	241
`katsura7`	0.92	0	44	832.16	**0.36**	12.61	44	128	227	581
`katsura8`	9.05	0	84	-	**4.41**	201.67	84	256	531	1248
`katsura9`	**74.36**	0	120	-	79.8	3095.92	120	512	1389	2834

FIGURE 2. Comparison of three methods (Aberth, Sturm-Habicht and Uspensky) to count the number of real roots of a polynomial. All times are in seconds. Column MPS is the time of MPSolve with options -Gc -SR -Dr (count roots, search on real line, detect real roots). Column ST gives the time of the Sturm-Habicht implementation from the PoSSo library. Column USP is the time of Uspensky's algorithm implemented in the RS software using multiprecision integers. The best timings with a guaranteed value are in bold face. Column TOT gives the total time of the elimination process — based on Gb and RS — that computes the polynomial to be studied. Column ? and OK indicate the number of probable and guaranteed real roots found by MPSolve (an italic value indicates a number of probable real roots leading to a wrong sum), while column NB gives the exact number of real roots; column Deg gives the number of complex roots. Column min and max give the binary size of the smallest and largest coefficients. A sign "-" indicates that the computation was aborted since it took more than 500 Mb or one hour. The timings have been obtained on a 733 Mhz Pentium-III with 512 Mb of main memory, running under Linux.

Example	MPS	?	OK	ST	USP	TOT	NB	Deg	min	max
`kin1`	0.11	0	16	234.93	**0.1**	113.55	16	48	3704	3767
`kinema`	0.13	0	8	1.84	**0.02**	0.5	8	40	107	244
`ku10`	0	2	0	**0**	**0**	0.02	2	2	18	21
`lorentz`	0	*11*	0	0.01	**0**	0.01	3	11	1	33
`mickey`	0	*3*	0	0.01	**0**	0.01	2	4	1	1
`mssm`	**0**	0	16	0.37	0.02	0.1	16	16	411	559
`noon3`	0	*21*	0	0.04	**0.01**	0.06	7	21	41	82
`noon4`	0.19	0	15	22.94	**0.05**	0.56	15	73	125	343
`noon5`	6.66	0	11	-	**0.56**	25.98	11	233	407	1338
`puma`	0	14	2	9.81	**0.03**	0.86	16	16	3278	3292
`quadfor2`	0	2	0	**0**	**0**	0.01	2	2	2	2
`quadgrid`	0	*3*	0	**0**	**0**	0.03	1	5	150	167
`rabmo`	0.05	0	8	0.3	**0.02**	3.49	8	16	373	501
`rbpl`	0.01	*21*	3	51.5	**0.01**	866.79	4	40	1178	1202
`rbpl24`	**0.06**	0	24	154.31	0.09	32.15	24	40	2230	2265
`redcyc5`	0.26	0	10	6.56	**0.06**	0.24	10	70	1	272
`redcyc6`	2.82	0	24	680.77	**0.49**	3.77	24	156	1	776
`redeco5`	0	*8*	0	**0**	**0**	0.01	4	8	9	13
`redeco6`	0	*10*	2	0.01	**0**	0.02	4	16	19	26
`redeco7`	0.01	*32*	0	0.15	**0.01**	0.07	8	32	42	59
`redeco8`	0.01	*64*	0	5.54	**0.02**	0.79	8	64	90	119
`rediff3`	0	*7*	1	0.01	**0**	0.05	2	8	185	189
`reimer5`	8.69	0	24	663.89	**0.32**	14.21	24	144	183	666
`robot_general`	0.02	*36*	2	491.12	**0.16**	339.44	8	40	4220	4618
`rose`	0.12	*4*	16	8.45	**0.19**	0.93	18	132	127	182
`s9_1`	0	*10*	0	**0**	**0**	0.03	4	10	39	63
`sendra`	0.01	*46*	0	1.44	**0.01**	0.11	6	46	87	103
`solotarev`	0.01	*4*	2	**0**	**0**	0.02	4	4	13	20
`sparse5`	0.09	*88*	0	542.88	**0.01**	2.17	0	160	1	613
`ssm`	0.01	*35*	0	79.46	**0.28**	5.79	12	36	3100	3669
`trinks`	0	*10*	0	**0**	**0**	0.05	2	10	71	79
`tssm`	0	*16*	0	0.43	**0**	0.46	4	16	861	1009
`virasoro`	147.57	0	224	-	**17.2**	268.71	224	256	1286	2830
`wright`	0.06	0	32	0.16	**0.04**	0.09	32	32	1	167

FIGURE 3. Comparison of three methods (Aberth, Sturm-Habicht and Uspensky) to count the number of real roots of a polynomial

The MPSolve program [11] implements a certified numerical method designed by D. Bini and G. Fiorentino. This algorithm, based on Aberth's method is designed for isolating all the complex roots of any univariate polynomial. Even if it may not distinguish real from complex roots in some cases, one can ask for a certified result (if some problem occurs, for example when the method can not decide if a root is real or not, a message appears in the result).

5. Conclusion

We have presented here a unified framework, which enables one to describe both Collins and Akritas' algorithm and Krandick's variant as different parameters choices from the same generic algorithm. We have deduced from this framework a new algorithm (REL), which is both optimal in terms of time and memory usage, for any kind of input polynomial. Our experiments with those three variants of Uspensky's algorithm shows that a good memory management algorithm can significantly reduce the computation time for Collins and Akritas' algorithm and Krandick's variant, whereas it has almost no effect on our new variant.

While breaking the memory barrier, the new method enables one to run Uspensky's algorithm on much larger polynomials, and thus becomes a serious concurrent of Sturm-like methods and perhaps even of numerical methods, although a complete comparison remains to be done in both cases. Still further ideas may decrease the computation time: use subquadratic algorithms like those described in [4] for Taylor shifts; use floating-point interval arithmetic instead of infinite-precision integer arithmetic, for example using the MPFR library [6]. Last but not least, the same algorithm can be applied to polynomials with algebraic or skewed coefficients.

References

[1] Akritas, A.-G. There is no "uspensky's" method. In honor of the 150th anniversary of Vincent's theorem,ACM, 1986.
[2] Collins, G., and Akritas, A. Polynomial real root isolation using descartes' rule of signs. In *SYMSAC* (1976), pp. 272–275.
[3] Collins, G., and Johnson, J. Quantifier elimination and the sign variation method for real root isolation. In *ACM-SYGSAM Symposium on Symbolic and Algebraic Coputation* (1989), pp. 264–271.
[4] Gathen, J. V. Z., and Gerhard, J. Fast algorithms for taylor shifts and certain difference equations. In *Proceedings of International Symposium On Symbolic And Algebraic Computations* (1997), pp. 40–47.
[5] GMP. http://www.swox.com/gmp/.
[6] INRIA-Lorraine, P. P. http://www.mpfr.org.
[7] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F4).-. *Journal of Pure and Applied Algebra 139*, 1–3 (June 1999), 61–88.
[8] Johnson, J. Algorithms for polynomial real root isolation. Tech. Rep. OSU-CISRC-8/91-TR21, Ohio State University, Department of Computer and Information Science, 1991.
[9] Krandick, W. A data structure for approximation. *Journal of Symbolic Computation 15* (1993), 143–167.
[10] Krandick, W. Isolierung reeller nullstellen von polynomen. In *Wissenschaftliches Rechnen* (1995), J. Herzberger, Ed., Akademie Verlag, Berlin, pp. 105–154.
[11] of Polynomial Equations : MPSolve v. 2.0, M. S. http://fibonacci.dm.unipi.it/pages/bini/public_html/software/.
[12] Rouillier, F. Resolution of zero-dimensional polynomial systems through the rational univariate representation. *Applicable Algebra in Engineering, Communication and Computing 9* (May 1999), 433–461.
[13] Roy, M.-F. Basic algorithms in real algebraic geometry: from sturm theorem to the existential theory of reals. Lectures on Real Geometry in memoriam of Mario Raimondo, Expositions in Mathematics, 23 , 1-67. Berlin, New York: de Gruyter, 1996.
[14] Uspensky, J. *Theory of equations.* McGraw-Hill Book Company, 1948.
[15] Vincent. Sur la résolution des equations numériques. *Journal de Mathématiques Pures et Appliquees* (1836), 341–372.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399