Faster Algorithms for Computing Hong's Bound on Absolute Positiveness

Kurt Mehlhorn and Saurabh Ray
Max Planck Institut für Informatik, Saarbrücken, Germany

Abstract

We show how to compute Hong's bound for the absolute positiveness of a polynomial in d variables with maximum degree δ in $O(n \log^d n)$ time, where n is the number of non-zero coefficients. For the univariate case, we give a linear time algorithm. As a consequence, the time bounds for the continued fraction algorithm for real root isolation improve by a factor of δ.

1 Introduction

Let A be a polynomial in d variables $x_1, \ldots, x_d$ with a maximum degree of δ in any of the variables. For any $\rho = (\rho_1, \ldots, \rho_d) \in \{0, \ldots, \delta\}^d$, we denote the quantity $x_1^{\rho_1} x_2^{\rho_2} \ldots x_d^{\rho_d}$ by X^ρ. Also, for any $\rho^1, \rho^2 \in \{0, \ldots, \delta\}^d$, we will write $\rho^1 \succ \rho^2$ if $\rho^1 \neq \rho^2$ and $\rho_k^1 \geq \rho_k^2$ for each $k \in \{1, \ldots, d\}$. Then, A has the form

$$A = \sum_{i=1}^{n} a_i X^{\rho^i}$$

where each a_i is a real number, $\rho^1, \ldots, \rho^n$ are distinct elements of $\{0, \ldots, \delta\}^d$ and n is the number of non-zero coefficients in A. We say that a monomial X^{ρ^i} of A is *dominant* if there is no other monomial X^{ρ^j} in A such that $\rho^j \succ \rho^i$.

Let B be a real number such that whenever $x_i \geq B$, for each $i \in \{1, \ldots, d\}$, the polynomial A and all its non-zero partial derivatives of every order have a positive value. Such a quantity B is called a bound on the absolute positiveness of A. For $\rho^1, \rho^2 \in \{0, \ldots, \delta\}^d$, let us denote by $||\rho^1 - \rho^2||$ the quantity $\sum_{i=1}^{d} |\rho_i^1 - \rho_i^2|$ and let

$$H(A) = \max_{\substack{i \\ a_i < 0}} \min_{\substack{j \\ a_j > 0,\ \rho^j \succ \rho^i}} \left(\frac{|a_i|}{a_j} \right)^{1/||\rho^j - \rho^i||}.$$

Hong [6] showed that if the coefficient of every dominant monomial in A is positive, then the quantity $(1 - 2^{-1/d})^{-1} H(A)$ is a bound on the absolute positiveness of A.

The expression for $H(A)$ gives an obvious way to compute it in $O(dn^2)$ time, where n is the number of non-zero coefficients, assuming that it takes $O(d)$ time to check for given i and j whether $\rho^j \succ \rho^i$. This is currently the best algorithm known. We show that $H(A)$ can be computed on a Real-RAM in $O(n \log^d \delta)$ time. It can also be approximated within a factor of four using rational arithmetic in linear time. For $d = 1$, we show that $H(A)$ can be computed in $O(n)$ time.

Sharma [7] analyzed the bit-complexity of the continued fraction methods for root isolation. The algorithms are recursive. In every node of the recursion tree, the current polynomial is transformed by

means of a Taylor shift and the Hong bound of the current polynomial is computed. Sharma remarks that fast Taylor shifts (with quasi-linear running time) have no advantage over standard Taylor shifts (with quadratic running time) as the computation of the Hong bound also takes quadratic time. This provides the motivation for our work: fast Taylor shift and our linear time algorithm for computing the Hong bound in the univariate case improve the running time of the continued fraction algorithm by a factor of n, where n is the degree[1] of the polynomial; this statement ignores logarithmic factors. For a square-free polynomial of degree n with integer coefficients of bit-length L, the bit complexity of Akritas' continued fraction algorithm [2] becomes $\tilde{O}(n^7L^3)$ and the bit-complexity of a formulation by Akritas and Strzeboński [1, 3] becomes $\tilde{O}(n^6L^2)$. The bit-complexity of Sharma's variant [7, Theorem 5.11][2] becomes $\tilde{O}(n^4L^2)$. The latter bound matches the bound for the Akritas-Collins bisection algorithm [4, 7].

We now review a few basic definitions and results from computational geometry that we will use in the description of our algorithms.

2 Preliminaries and Notation

Let $Q = \{q_1, \ldots, q_n\}$ be a set of n points in the plane so that for any $j > i$, the x-coordinate of q_j is strictly greater than the x-coordinate of q_i. We denote the convex hull of Q by $CH(Q)$. The boundary of $CH(Q)$ consists of two chains between q_1 and q_n (see Figure 1). The *lower hull* of Q is the chain

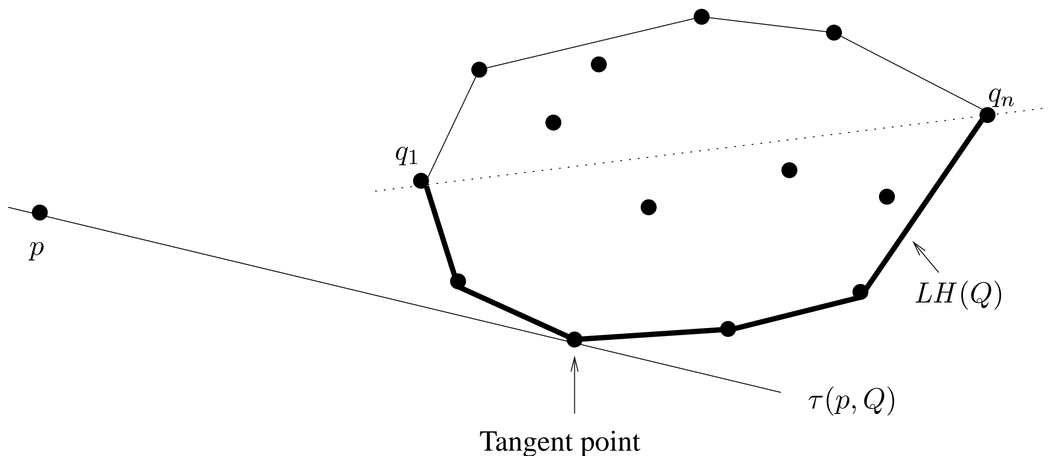

Figure 1: Lower Hull and Lower Tangent

consisting of points lying on or (vertically) below the line passing through q_1 and q_n. We denote the lower hull of Q by $LH(Q)$. For computational purposes, we will find it convenient to store lower hulls as an array of vertices on it from left to right i.e. in the increasing order their x-coordinates.

We say that a line l is a *lower tangent* of $CH(Q)$ if l passes through at least one of the points of Q and all points of Q lie on or above l. Among the points of Q that lie on l, let q be the leftmost

[1]Observe that here we are using n for the degree as in [7]. Elsewhere in this paper we use δ for the degree and n for the number of non-zero coefficients

[2]Theorem 5.11 in [7] refers to algorithm $\texttt{MCDS}(A, M)$ defined in Section 5.4 of [7]. The definition of the algorithm refers to the "ideal positive lower bound function" PLB. The next to last paragraph in Section 2.1 states that from now on the Hong bound is used for computing the PLB function.

i.e. the point with the smallest x-coordinate. Clearly, q lies on $LH(Q)$ and we will call it the *point of tangency* of l.

Let p be a point whose x-coordinate is smaller than the x-coordinate of q_1. We denote by $\tau(p, Q)$, the unique lower tangent to $CH(Q)$ that passes through p. Suppose that we have $LH(Q)$ available as an array $v_1, \ldots, v_k$ of the points in it from left to right and we want to compute the array corresponding to $LH(\{p\} \cup Q)$. To do this, we first find the point of tangency v_i of $\tau(p, Q)$. We then insert p in the position of v_{i-1} and mark this position as the beginning of the array. Effectively, we have removed $v_1, \ldots, v_{i-1}$ from the front of the array and then inserted p at the front. This gives us the array corresponding to $LH(\{p\} \cup Q)$. The point of tangency of $\tau(p, Q)$ is the unique point v_i satisfying the following conditions: (i) v_{i-1} lies strictly above the line through p and v_i unless $i = 1$ and (ii) v_{i+1} lies on or above the line through p and v_i unless $i = k$. There are two ways to find the point of tangency. One way is to simply traverse the list until we find a point v_i satisfying the above conditions. The time taken is $O(i + 1)$, i.e. it is proportional to the number of points that are removed from $LH(Q)$ in order to obtain $LH(\{p\} \cup Q)$. A second method is to do binary searching. For a point v_j, we can easily decide whether it is the tangent point and if not whether the tangent point lies to the left of it or to the right of it. This allows us to find the tangent point in time $O(\log k)$.

Suppose that we want to go over the points $q_n, q_{n-1}, \ldots, q_1$ in that order and we always want to maintain the lower hull of the set of points we have seen so far. In other words, if we let Q_i denote the set $\{q_i, q_{i+1}, \ldots, q_n\}$ for $i \in \{1, \ldots, n\}$, we want to enumerate $LH(Q_n)$, $LH(Q_{n-1})$, $\ldots$, $LH(Q_1)$ in that order. In how much time can we do this? $LH(Q_n)$ consists of just one point is easily constructed. Once $LH(Q_{i+1})$ is available for some i, we can use the method described above to construct $LH(Q_i) = LH(\{q_i\} \cup Q_{i+1})$. The total time taken depends on how we find the tangent point at each step. If we use binary searching, the time taken in each step is $O(\log n)$ and hence the total time is $O(n \log n)$. However, if we use the naive method of scanning the array from the front until we find the tangent point, the time taken is $O(n)$. This follows from the observation that if r_i vertices are removed from $LH(Q_{i+1})$ in order to obtain $LH(Q_i)$, then the time required in that step $O(r_i + 1)$. Since a point is removed from the lower hull at most once during the entire enumeration process, the total time taken is $O(n)$. The algorithm that we just described is a standard way of incrementally computing convex hulls of two dimensional point sets.

3 Computing Hong's Bound for Univariate Polynomials

Let $A = \sum_{i=1}^{n} a_i x^{d_i}$ be a polynomial in a single variable x where $d_1 < d_2 < \ldots < d_n = \delta$ are non-negative integers and $a_1, a_2, \ldots, a_n$ are non-zero reals with $a_n > 0$. Then, $2H(A)$ is a bound on the absolute positiveness of A, where

$$H(A) = \max_{\substack{i \\ a_i < 0}} \min_{\substack{j > i \\ a_j > 0}} \left(\frac{|a_i|}{a_j} \right)^{1/(d_j - d_i)}.$$

Since the $\log(\cdot)$ function is monotone, we have

$$\log H(A) = \max_{\substack{i \\ a_i < 0}} \min_{\substack{j > i \\ a_j > 0}} \frac{\log|a_i| - \log a_j}{d_j - d_i}.$$

Logarithms in this paper will always be with base 2. Let $b_k = -\log|a_k|$ for $k \in \{1, \ldots, n\}$. Our task

is to compute

$$\log H(A) = \max_{\substack{i \\ a_i<0}} \min_{\substack{j>i \\ a_j>0}} \frac{b_j - b_i}{d_j - d_i}.$$

For $i \in \{1, \ldots, n\}$, let p_i be the point (d_i, b_i). The quantity $\frac{b_j-b_i}{d_j-d_i}$ is then the slope of the line passing through p_i and p_j. For any i such that $a_i < 0$, let

$$s_i = \min_{\substack{j>i \\ a_j>0}} \frac{b_j - b_i}{d_j - d_i}.$$

Note that s_i is not defined for all $i \in \{1, \ldots, n\}$. Let $P = \{p_1, \ldots, p_n\}$. We will call a point $p_i \in P$ a *positive* point if $a_i > 0$ and we will call it a *negative* point if $a_i < 0$. Denote by P_i^+ the set $\{p_j : j \geq i \text{ and } a_j > 0\}$ and let $\mathcal{L}_i$ denote the lower hull of P_i^+. Notice that s_i is the slope of $\tau(p_i, P_i^+)$, the lower tangent to $CH(P_i^+)$ passing through p_i. We now describe two approaches to computing $\log H(A)$. The first approach is a naive approach that takes $O(n \log n)$ time. The second approach improves the running time to $O(n)$.

3.1 A Suboptimal Algorithm

The first approach is to compute s_i for all $i \in \{1, \ldots, n\}$ s.t. $a_i < 0$ and then take the maximum among them. We go over the points $p_n, p_{n-1}, \ldots, p_1$ in that order and we maintain the lower hull of the positive points seen so far exactly as described in Section 2. When we process p_i, we assume that we have $\mathcal{L}_{i+1}$ available as an array. If $a_i > 0$, we find the tangent point of $\tau(p_i, P_{i+1}^+)$ by sequentially scanning L_{i+1} and then update $\mathcal{L}_{i+1}$ to obtain $\mathcal{L}_i$. If $a_i < 0$, we set $\mathcal{L}_i = \mathcal{L}_{i+1}$ since $P_i^+ = P_{i+1}^+$. We then compute $\tau(p_i, P_i^+)$ by computing its tangent point using binary search on $\mathcal{L}_i$. This takes $O(\log n)$ time. The total time taken is $O(n \log n)$ since the time required to maintain the lower hulls of the positive points is $O(n)$ and the total time spent per negative point is $O(\log n)$. We next give an algorithm with linear running time.

3.2 An Optimal Algorithm

In the previous approach, we compute s_i for each negative point p_i. However we don't need to do so. As we process $p_n, p_{n-1}, \ldots, 1$, for any negative point p_i we need to compute s_i only if it is larger than the largest s_j we have computed so far because we are interested only in the maximum of all s_i's. Refer to Algorithm 1 for the pseudocode of the algorithm that we are about to describe.

For any $i \in \{1, \ldots, n-1\}$, let $\sigma_i = \max_{j \geq i, a_j < 0} s_j$ and let ℓ_i be the lower tangent to $CH(P_i^+)$ with slope σ_i. Further let t_i be the point of tangency. $\mathcal{L}_n$ consists of just p_n. We define σ_n to be $-\infty$ and ℓ_n to be a line of slope $-\infty$ passing through p_n and $(0, \infty)$. We set $t_n = p_n$. We will maintain these quantities along with $\mathcal{L}_i$, as we process the points. In the end, σ_1 will give us $\log H(A)$.

Suppose now that we are processing a point p_i, $i < n$. At this time, we have $\mathcal{L}_{i+1}$, σ_{i+1}, ℓ_{i+1} and t_{i+1} available.

Case 1: $a_i < 0$. We first set $\mathcal{L}_i = \mathcal{L}_{i+1}$ since $P_i^+ = P_{i+1}^+$. We then check whether p_i lies below ℓ_{i+1}. If p_i lies on or above ℓ_{i+1}, then we are sure that s_i is not larger than σ_{i+1} and we can just ignore p_i. In this case, we set $\sigma_i = \sigma_{i+1}$, $\ell_i = \ell_{i+1}$ and $t_i = t_{i+1}$. On the other hand if p_i lies below ℓ_{i+1}, we know that s_i is larger than σ_i. We also know that the tangent point t_i of $\ell_i = \tau(p_i, P_i^+)$ cannot lie to

the left of t_{i+1}. We find t_i (and hence ℓ_i) by scanning $\mathcal{L}_{i+1}$ starting from t_{i+1} and moving right along $\mathcal{L}_{i+1}$. The slope of ℓ_i gives us s_i and since $s_i > \sigma_{i+1}$, $\sigma_i = s_i$. Observe that edges of $\mathcal{L}_{i+1}$ that we traverse in searching for t_i have slopes in the interval $[\sigma_{i+1}, \sigma_i)$. We will use this fact to analyse the running time of the algorithm.

Case 2: $a_i > 0$. In this case we don't need to compute s_i since it is not defined. We set $\sigma_i = \sigma_{i+1}$, $\ell_i = \ell_{i+1}$ and $t_i = t_{i+1}$. We then scan $\mathcal{L}_{i+1}$ from the front to find the tangent point of $\tau(p_i, P_i^+)$ and update $\mathcal{L}_{i+1}$ to get $\mathcal{L}_i$ as before (see Section 3.1).

The pseudocode of the algorithm is shown in Algorithm 1. The time spent in all lines except lines 10 and 19 is constant. In line 19, we compute the tangent point in order to maintain the lower hull of the positive points. As we have seen before, the total time spent in this process is $O(n)$. Hence, to show that our algorithm runs in $O(n)$ time, we just need to argue that the total time spent in line 10 is $O(n)$. To see this, recall that as we search for the tangent point in line 10, we traverse vertices of $\mathcal{L}_{i+1}$ starting from t_{i+1} until we reach t_i. The edges between these vertices have slopes in the range $[\sigma_{i+1}, \sigma_i)$. This means that we traverse distinct edges each time we visit line 10. Each time, the time we spend in line 10 is $O(1 + \#$ of edges visited$)$. Since at most n edges ever appear in the lower hull of the positive points, it follows that the total time spent in line 10 is $O(n)$.

3.3 Approximating $H(A)$ using Rational Arithmetic

Since $H(A)$ cannot always be computed exactly using rational arithmetic, Sharma [7] proposes a procedure to compute a quantity $U(A)$ using rational arithmetic so that $U(A)/4 < 2H(A) < U(A)$. His procedure computes

$$u(A) = \max_{\substack{i \\ a_i<0}} \min_{\substack{j>i \\ a_j>0}} \left\lfloor \frac{\lfloor \log|a_i| \rfloor - \lfloor \log a_j \rfloor - 1}{j-i} \right\rfloor$$

and returns $U(A) = 2^{u(A)+3}$. The quantity

$$u'(A) = \max_{\substack{i \\ a_i<0}} \min_{\substack{j>i \\ a_j>0}} \frac{\lfloor \log|a_i| \rfloor - \lfloor \log a_j \rfloor - 1}{j-i}$$

can be easily computed using the linear time algorithm described earlier by just redefining the point p_i to be $(i, -\lfloor \log a_i \rfloor - 1)$ if $a_i > 0$ and $(i, -\lfloor \log|a_i| \rfloor)$ if $a_i < 0$. So we just compute $u'(A)$ and return $U(A) = 2^{\lfloor u'(A) \rfloor + 3}$.

The bit complexity of this algorithm is smaller than the bit complexity of Sharma's algorithm by a factor of $\Theta(n)$ since the only change we made to his algorithm is to use an $O(n)$ time algorithm instead of an $O(n^2)$ time algorithm for the Real RAM model. The sizes of the numbers that we deal with and the way we get the algorithm for the rational arithmetic model from the algorithm for Real RAM model are the same as in Sharma's case.

4 Computing the Bound for Multivariate Polynomials

We now describe our algorithm for computing the Hong Bound for multivariate polynomials. As we have seen before, for a multivariate polynomial

$$A = \sum_{i=1}^{n} a_i X^{\rho^i}$$

```
1  Algorithm: Compute log H(A)
   Input: Polynomial A(x) = \sum_i^n a_i x^{d_i}
   Output: log H(A)
   // Process p_n
2  t_n = p_n;
3  ℓ_n = LineThrough(p_n, (0, ∞)); // ℓ_n is the line through p_n and (0, ∞)
4  σ_n = −∞; // The slope of ℓ_n is −∞
5  L_n = {p_n};
   // Process p_{n−1}, ..., p_1
6  for (i = n − 1; i ≥ 1; i = i − 1) do
7      if a_i < 0 then
8          L_i = L_{i+1};
9          if p_i lies below ℓ_{i+1} then
               // s_i > σ_{i+1}
10             t_i = ComputeTangentPoint1(p_i, L_{i+1}); // Computes the tangent point
                   of τ(p_i, P^+_{i+1}) by scanning L_{i+1} starting from t_{i+1}
11             ℓ_i = LineThrough(p_i, t_i) // ℓ_i is the line through p_i and t_i
12             σ_i = s_i = Slope(ℓ_i) // σ_i and s_i are set to the slope of ℓ_i
13         end
14         else
               // p_i lies on or above ℓ_{i+1}
15             t_i = t_{i+1}; σ_i = σ_{i+1}; ℓ_i = ℓ_{i+1};
16         end
17     end
18     else
           // a_i > 0
19         t = ComputeTangentPoint2(p_i, L_{i+1});// Computes the tangent point of
               τ(p_i, P^+_{i+1}) by scanning L_{i+1} starting from the front
20         Replace points before t in L_{i+1} by p_i to obtain L_i;
21     end
22 end
23 return σ_1
```

Algorithm 1: Pseudocode of the optimal algorithm for univariate case

with d variables and a maximum degree of δ in any of the variables,

$$H(A) = \max_{\substack{i \\ a_i<0}} \min_{\substack{j \\ a_j>0,\ \rho^j \succ \rho^i}} \left(\frac{|a_i|}{a_j}\right)^{1/||\rho^j-\rho^i||}.$$

We first form groups of indices $U_1, \ldots, U_r$ and $V_1, \ldots V_r$ so that the following hold:

- For each $k \in \{1, \ldots, r\}$, if $i \in U_k$ and $j \in V_k$ then $a_i < 0$, $a_j > 0$ and $\rho^j \succ \rho^i$.
- For each pair of indices i, j such that $a_i < 0$, $a_j > 0$ and $\rho^j \succ \rho^i$, there is a unique k such that $i \in U_k$ and $j \in V_k$.
- $\sum_{k=1}^{r} |U_k| + |V_k| = O(n \log^d n)$, where, as before, n is the number nonzero coefficients.

These groups can be easily computed in $O(n \log^d n)$ time using a standard data structure for orthogonal range queries in d dimensions. The construction of the data structure that we will use can be found in [5]. We build a data structure on the set $\{\rho_j; a_j > 0\}$ which, given any query $\rho \in \{0, \ldots, \delta\}^d$, returns the set $\{j; a_j > 0$ and $\rho^j \succ \rho\}$. The answer to such a query is given as the disjoint union of $O(\log^d n)$ canonical subsets of $\{j; a_j > 0\}$. The total size of the canonical subsets stored by the data structure is $O(n \log^d n)$ and these canonical subsets form the groups $V_1, \ldots, V_r$. We then run a query with each i such that $a_i < 0$. The set U_k is formed by the set of indices i such that the answer to the query ρ^i contains the canonical subset V_k. It can be checked that these groups satisfy the conditions above.

For any index i, let y_i be the quantity $\sum_{j=1}^{d} \rho_j^i$. For each pair (i, k) such that $i \in U_k$, we define $\sigma_{i,k}$ as follows:

$$\sigma_{i,k} = \log \min_{j \in V_k} \left(\frac{|a_i|}{a_j}\right)^{1/(y_j-y_i)}.$$

Then,

$$\log H(A) = \max_{\substack{i \\ a_i<0}} \min_{\substack{k \\ i \in U_k}} \sigma_{i,k}.$$

For each index j, let p_j be the point (y_j, b_j) in the plane where $b_j = -\log|a_j|$ as defined earlier. Then we have the following expression for $\sigma_{i,k}$:

$$\sigma_{i,k} = \min_{j \in V_k} \frac{b_j - b_i}{y_j - y_i}.$$

The quantity $\frac{b_j-b_i}{y_j-y_i}$ is again the slope of the line passing through p_i and p_j.

For each k, we compute the quantities $\sigma_{i,k}$ for all $i \in U_k$ by running the sub-optimal algorithm (see section 3.1) for the univariate case on $P_k = \{p_i : i \in U_k \cup V_k\}$). This takes $O(((|U_k| + |V_k|) \log |V_k|)$ time. Thus, computing the $\sigma_{i,k}$'s for all k and all $i \in U_k$ takes time $O(\sum_k (|U_k| + |V_k|) \log |V_k|)$ $= O(n \log^{d+1} n)$. As a result, $H(A)$ can be computed in the same amount of time.

This running time can be easily improved to $O(n \log^d n)$ if we assume that the ρ^is are sorted along one of the dimensions and we form the groups using the other dimensions. We go over them in the decreasing order of their magnitude along the sorted dimension, just as we did in the univariate case, and we use a $(d-1)$-dimensional orthogonal range query data structure (instead of a d-dimensional data structure) for the remaining $d-1$ dimensions. More precisely, we run several one-dimensional

problems in parallel, one for each non-empty canonical subset of the $(d-1)$-dimensional data structure. This means that we have as many one dimensional problems as the number of canonical subsets in the data structure. As we process the points, for each point processed, we input a two dimensional point into one or more of the one dimensional problems. Recall that we process the points in the decreasing order of their magnitude in the sorted dimension. When we process a point p_i with $a_i > 0$, we insert $\bar{\rho}^i = (\rho_2^i, ..., \rho_d^i)$ into the data structure. This adds $\bar{\rho}^i$ to $\log^{d-1} n$ canonical subsets in the data structure. For each such subset, we add the point (y_i, b_i) to the corresponding one-dimensional problem. The one dimensional problems are solved using the sub-optimal algorithm just as we did above. The solutions to the one dimensional problems give the $\sigma_{i,k}$s we need. From these quantities we compute $H(A)$ as above.

Furthermore, if we exploit the fact that the coordinates of the ρ^is are integers between 0 and δ then we can replace $\log n$ by $\log \delta$ in the bound. So, we get a running time of $O(n \log^d \delta)$. The approximation using rational arithmetic can be done as before for each σ_k. This gives an approximation for $H(A)$.

5 Acknowledgements

The authors would like to thank Vikram Sharma for discussions and helpful suggestions on the problem and Chee Yap for pointing out a mistake in the original write-up.

References

[1] A. G. Akritas and A. Strzeboński. A comparative study of two real root isolation methods. *Nonlinear Analysis: Modelling and Control*, 10(4):297–304, 2005.

[2] A.G. Akritas. *Vincent's theorem in algebraic manipulation*. PhD thesis, North Carolina State University, 1978.

[3] A.G. Akritas, A. Bocharov, and A. Strzeboński. Implementation of real root isolation algorithms in Mathematica. In *Abstracts of the International Conference on Interval and Computer Algebra Methods in Science and Engineering, Interval '94*, pages 23–27, 1994.

[4] G. E. Collins and A. G. Akritas. Polynomial real root isolation using Descartes' rule of signs. In R. D. Jenks, editor, *SYMSAC*, pages 272–275, Yorktown Heights, NY, 1976. ACM Press.

[5] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications: Chapter 5*. Springer, 1997.

[6] Hoon Hong. Bounds for absolute positiveness of multivariate polynomials. *J. Symb. Comput.*, 25(5):571–585, 1998.

[7] Vikram Sharma. Complexity of real root isolation using continued fractions. *Theor. Comput. Sci.*, 409(2):292–310, 2008.