ETH zürich

ETH Library

A probabilistic zero test for expressions involving roots of rational numbers

Report

Author(s):
Blömer, Johannes

Publication date:
1998

Permanent link:
https://doi.org/10.3929/ethz-a-006652870

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
Technical report / Departement Informatik, ETH Zürich 299

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

A Probabilistic Zero-Test for Expressions Involving Roots of Rational Numbers

Johannes Blömer

May 27, 1998

Abstract. Given an expression E using $+, -, *, /$, with operands from $\mathbf{Z}$ and from the set of real roots of integers, we describe a probabilistic algorithm that decides whether $E = 0$. The algorithms has a one-sided error. If $E = 0$, then the algorithm will give the correct answer. If $E \neq 0$, then the error probability can be made arbitrarily small. The algorithm has been implemented and is expected to be practical.

Author's address:

Johannes Blömer
Institute for Theoretical Computer Science
ETH Zentrum
CH-8092 Zurich
Switzerland

Technical Report #299, Departement Informatik, ETH Zürich.
Electronically available from:
ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/

1 Introduction

In this paper we consider the following problem. Given a real radical expression without nested roots, that is, an expression E defined with operators $+, -, *, /$, with integer operands and operands of the form $\sqrt[d]{n}, d \in \mathbf{N}, n \in \mathbf{Z}, \sqrt[d]{n} \in \mathbf{R}$. We want to decide whether the expression E is zero. We describe an efficient, probabilistic algorithm to solve this problem. If the expression is zero, the algorithm will give the correct answer. If E is non-zero, the probability that the algorithms declares E to be zero, can be made arbitrarily small. The algorithm is not based on root separation bounds. Unlike algorithms based on root separation bounds, the algorithm has a worst-case running time that does not depend exponentially on the number of input roots. Similarly, the algorithm improves the algorithm in [2]. In that paper expressions are restricted to sums of roots. Of course, turning an arbitrary expression with k input roots into a sum of roots, creates a sum with up to 2^k terms. Again, the new algorithms avoids this behavior.

Tests in computer programs often can be reduced to determining the sign of a radical expression as described above. This is particularly true for problems in computational geometry (see for example [6],[12], [15],[16]). Computing the sign of a radical expression E obviously is a harder problem than deciding whether the expression is zero. Currently, any sign detecting algorithm is based on root separation bounds. That is, the algorithm first computes a bound b such that if E is non-zero then $|E| > 2^{-b}$ (see [7],[13] for the best bounds currently available). In a second step, it approximates E with absolute error less than 2^{-b} . However, experiments often show that if the expression E is close to zero, then E actually is zero. Here, by "close to zero" we mean that computing E with ordinary floating-point arithmetic does not allow to infer the sign of E. In these situations an efficient zero-test can be used as follows. To determine the sign of an expression E, first compute E using machine-provided floating-point arithmetic. If this allows you to detect the sign, stop. Otherwise, use the zero-test to determine whether E is zero. If this is the case, stop. Otherwise, approximate E with accuracy 2^{-b} to detect the sign of E. Here 2^{-b} is the accuracy required by the root separation bound. As mentioned, experiments indicate that in many situations the most expensive, third step hardly ever will be necessary.

A second application for a zero-test is in detecting degeneracies in geometric configurations. Here one needs to distinguish between two different types of degeneracies. One is caused by the use of finite-precision arithmetic. These degeneracies one usually wants to remove. The other degeneracies are problem-inherent degeneracies which one may want to keep. The problem-inherent degeneracies can often be detected by a zero-test as provided by the algorithm described in this paper. Degeneracies caused by finite-precision arithmetic then can be removed by some perturbation scheme.

Let us briefly outline the algorithm described in this paper. The basic idea, which originates in [8], is as follows. If α is an algebraic integer and α_i are its conjugates, then either $\alpha = 0$ or α and all its conjugates are non-zero. Therefore, rather than testing whether $\alpha \neq 0$, we may choose any conjugate α_i of α and check whether $\alpha_i \neq 0$. The simple but fundamental observation of [8] is, that although $|\alpha| \neq 0$ may be small, with high probability the absolute value of a random conjugate α_i of α is not too small. Hence a moderately accurate approximation to α_i will reveal that α_i, and therefore α, is non-zero.

Let us apply this idea to a radical expression E. For the sake of simplicity, we restrict ourselves to division-free expressions E involving only square roots $\sqrt{n_1}, \ldots, \sqrt{n_k}$. To

apply the method described above we need to be able to generate a random conjugate $\overline{E}$ of E. It is well-known that the conjugates of E can be obtained be replacing the roots $\sqrt{n_1}, \ldots, \sqrt{n_k}$ by roots $\epsilon_1\sqrt{n_1}, \ldots, \epsilon_k\sqrt{n_k}$, where $\epsilon_j = \pm 1$. Unfortunately, not all sign combinations lead to a conjugate. To see this, consider the following toy example, $E = \sqrt{2}\sqrt{3} - \sqrt{6}$. Interpreting all square roots as positive real numbers, we see that $E = 0$. However, choosing the sign combination $-1, -1, -1$ leads to $\sqrt{2}\sqrt{3} + \sqrt{6} \neq 0$. In particular, this sign combination does not lead to a conjugate of E. Of course, this combination fails to generate a conjugate because $\sqrt{2}\sqrt{3} = \sqrt{6}$. In general, we need to find multiplicative dependencies between the input roots. To determine these multiplicative dependencies we use a well-known procedure called factor-refinement (see for example [1]). Its running time is $\mathcal{O}(\log^2(n))$, where $n = \prod n_i$. Once the dependencies have been determined and a random conjugate $\overline{E}$ has been generated, the algorithm to check whether $E = 0$ simply approximates $\overline{E}$.

What is the accuracy required for this approximation in order to guarantee an error probability less than 1/2, say? By the result of [8] the approximation needs to have accuracy 2^{-B}, where 2^B is an upper bound on the absolute value of the conjugates of E. To obtain such an estimate we use a bound $\mathrm{u}(E)$ first introduced in [7]. A similar, but slightly worse bound is obtained in [13]. $\mathrm{u}(E)$ is easy to compute and, in the worst case, is the best possible upper bound on $|E|$ itself. Summarizing, except for an overhead of $\mathcal{O}(\log^2(n))$, the running time of our algorithm will be the time needed to compute $\overline{E}$ with absolute error $2^{-\mathrm{u}(E)}$. Since E and $\overline{E}$ differ only in the signs of the input radicals, in the worst case this is the time needed to compute E with absolute error $2^{-\mathrm{u}(E)}$.

As mentioned above, this compares favorably to algorithms based on separation bounds. Take the bounds in [7], which are the best bounds currently available. To decide whether an expression E as above is zero, an approximation to E with absolute error less than $2^{-2^k \mathrm{u}(E)}$ is computed, where k is the number of input square roots. Hence the quality of the approximation required by the algorithm in [7] differs by a factor of 2^k from the quality required by our algorithm. We achieve this reduction by a preprocessing step requiring time $\mathcal{O}(\log^2(n))$. Even for moderately small values of k this is time well spent.

This leads to the main question this paper raises. The bounds in [7] not only apply to expressions as defined above, these bounds also apply to expressions with nested roots[1]. That is, in the expression we not only allow the operations $+, -, *, /$, we also allow operators of the form $\sqrt[d]{\ }$. Although we feel that the class of expressions the algorithm described in this paper can handle is the most important subclass of the expressions dealt with in [7], it would be very interesting to generalize the algorithm to the class of nested radical expressions. Note that denesting algorithms as in [10],[9], and [4] implicitly provide a zero-test for these expressions. But denesting algorithms solve are a far more general problem than testing an expression for zero. Accordingly, denesting algorithms, if used as zero-tests, are less efficient than algorithms based on roots separation bounds.

The algorithm described in this paper has been implemented. So far no effort has been made to optimize its running time, but the algorithm seems to be practical. The main objective of the implementation was to compare the probabilistic behavior observed in practice with the probabilistic guarantees provided by the theory. The data set is still rather small. As expected, the algorithms performs better than predicted by theory. To give some specific numbers, we tested the algorithms on the determinant of 3×3 matrices

[1] It should be noted, that it is unknown whether the bounds in [7] can be improved, if nested roots are not allowed.

whose entries are sums of square roots. The integers involved are 5-digit integers. We generated matrices whose determinant is less than 10^{-6}. Random conjugates of these determinants consistently fell in the range from $10^5 - 10^7$. We never found an example where the random conjugate of a determinant was smaller than the determinant itself.

The paper is organized as follows. In Section 2 the main definitions are given, and we formally state the main result. In Section 3 we describe the algorithm for division-free expression and analyze its running time. In Section 4 we analyze the error probability of the algorithm. In Section 5 we show how to generalize the algorithm to expressions with divisions.

2 Definitions and statement of results

Throughout this paper, we only deal with roots of integers $\sqrt[d]{n}, d \in \mathbf{N}, n \in \mathbf{Z}$. The symbol $\sqrt[d]{n}$ does not specify a unique complex number. However, when we use this symbol, we will always assume that some specific d-th root of n is referred to. How this particular root is specified is irrelevant, except that the specification must allow for an efficient approximation algorithm. Usually we will require that a root $\sqrt[d]{n}$ is a real number. In this case, we assume that n is positive.

Our definition of a radical expression is the same as the definition of a straight-line program over the integers, except that we allow roots of integers as inputs. To be more specific, for a directed acyclic graph (dag) the nodes of in-degree 0 will be called *input nodes*. The nodes of out-degree 0 will be called *output nodes*. Nodes that are not inputs are called *internal nodes*.

Definition 2.1 *A depth* 1 *radical expression E over the integers is a directed acyclic graph (dag) with a unique output node and in-degree exactly* 2 *for each internal node. Each input node is labeled either by an integer or by a root of an integer. Each internal node is labeled by one of the arithmetic operations $+, -, *, /$. If no internal node is labeled by $/$, then E is called a division-free radical expression. If the inputs are labeled by integers and real roots of integers, then the expression is called a real radical expression. In either case, the input labels that are integers are called the input integers and the remaining input labels are called the input radicals.*

These expressions are called depth 1 expressions, since we do not allow operators of the form $\sqrt[d]{}$ for the internal nodes. Hence there are no nested roots in the expression. In this paper all expressions are depth 1 expression. In the sequel, we will omit the prefix "depth 1". Similarly, the suffix "over the integers" will be omitted.

For a radical expression E with e edges and with input integers $m_1, \ldots, m_k$ and input radicals $\sqrt[d_1]{n_1}, \ldots, \sqrt[d_l]{n_l}$ the *size of* E, denoted by $\text{size}(E)$, is defined as

$$e + \sum_{i=1}^{k} \log|m_i| + \sum_{j=1}^{l} \log|n_j| + \sum_{j=1}^{l} d_j,$$

where the logarithms are base 2 logarithms. Remark that $\text{size}(E)$ depends linearly on d_i rather than on $\log(d_i)$.

To each node v of a radical expression we can associate a complex number, called the *value* $\text{val}(v)$ of that node. The value of an input node is the value of its label. If v is an

internal node labeled with $\circ \in \{+, -, *, /\}$ and edges from nodes v_1, v_2 are directed into v, then $\mathrm{val}(v) = \mathrm{val}(v_1) \circ \mathrm{val}(v_2)$. The *value* $\mathrm{val}(E)$ *of a radical expression* is the value of the output node of E.

It is easy to construct an expressions with $\mathcal{O}(n)$ edges whose value is double-exponential in n. This shows that in general one cannot even write down $\mathrm{val}(E)$ in time polynomial in $\mathrm{size}(E)$. One way to avoid this problem is to restrict expressions E to trees. In this case, $\log(|\mathrm{val}(E)|)$ is bounded by a polynomial in $\mathrm{size}(E)$. In this work we follow a different approach. For a radical expression E we define an easily computable bound $\mathrm{u}(E)$ such that for a division-free expression $\mathrm{u}(E)$ is an upper bound on $|\mathrm{val}(E)|$. Later we will see that arbitrary expressions E can be written as the quotient of two division-free radical expressions E_1, E_2 such that $|\mathrm{val}(E_1)| \leq \mathrm{u}(E)$. The definition of the bound $\mathrm{u}(E)$ follows [7].

Let E be a radical expression and let v be a node of E. For an input node v the bound $\mathrm{u}(v)$ is the absolute value of its label. $\mathrm{l}(v)$ is defined to be 1. If v is an internal node and edges from v_1, v_2 are directed into v, then $\mathrm{u}(v), \mathrm{l}(v)$ are defined as follows:

$$\begin{array}{lcll} \mathrm{u}(v) & = & \mathrm{u}(v_1)\mathrm{l}(v_2) + \mathrm{u}(v_2)\mathrm{l}(v_1) & \text{if } v \text{ is labeled with } +, - \\ \mathrm{l}(v) & = & \mathrm{l}(v_1)\mathrm{l}(v_2) & \\ & & & \\ \mathrm{u}(v) & = & \mathrm{u}(v_1)\mathrm{u}(v_2) & \text{if } v \text{ is labeled with } * \\ \mathrm{l}(v) & = & \mathrm{l}(v_1)\mathrm{l}(v_2) & \\ & & & \\ \mathrm{u}(v) & = & \mathrm{u}(v_1)\mathrm{l}(v_2) & \text{if } v \text{ is labeled with } /. \\ \mathrm{l}(v) & = & \mathrm{u}(v_2)\mathrm{l}(v_1) & \end{array}$$

Finally, we define $\mathrm{u}(E)$ as the corresponding value of the output node of E. If E is division-free, then $|\mathrm{val}(E)| \leq \mathrm{u}(E)$. With these definitions we can state the main result of this paper.

Theorem 2.2 *Let E be a real radical expression. There is a probabilistic algorithm with one-sided error* $1/2$ *that decides whether* $\mathrm{val}(E) = 0$. *If the algorithm outputs* $\mathrm{val}(E) \neq 0$, *then the answer is correct. The running time of the algorithm is polynomial in* $\mathrm{size}(E) + \log \mathrm{u}(E)$.

We did not state the running time explicitly, because the running time depends on the way specific values for the input radicals are represented. As will be seen later, the running time of the algorithm is usually dominated by the running time of an algorithm approximating E with absolute error $2^{-\lceil \log \mathrm{u}(E) \rceil}$.

By running the algorithm e times with independent random bits, the error probability can be reduced to $\epsilon = 2^{-e}$. We will see later that there is a better way to achieve this error probability, if e is small.

3 The algorithm for division-free expressions

In this section we will describe a probabilistic algorithm that decides whether a division-free radical expression is zero. We will also analyze the running time of the algorithm. In the following section we will analyze the error probability of the algorithm.

Before we describe the algorithm recall that a *d-th root of unity*, $d \in \mathbf{N}$, is a solution of $X^d - 1 = 0$. The d-th roots of unity are given by $\exp(ik\pi/d), k = 0, \ldots, d-1$. Therefore, a random d-th root of unity corresponds to a random number between 0 and $d-1$.

Algorithm Zero-Test

`Input:` A real division-free radical expression E with input radicals $\sqrt[d_1]{n_1}, \ldots, \sqrt[d_k]{n_k}$.

`Output:` "zero" or "non-zero"

`Step 1:` Compute $m_1, \ldots, m_l \in \mathbf{Z}$, such that $\gcd(m_i, m_j) = 1$ for all $i \neq j, i, j = 1, \ldots, l$, and such that each n_i can be written as $n_i = \prod_{j=1}^{l} m_j^{e_{ij}}, e_{ij} \in \mathbf{N}$. Compute this representation for each n_i.

`Step 2:` For all $(i,j), i = 1, \ldots, k, j = 1, \ldots, l$, compute the minimal $d_{ij} \in \mathbf{N}$ such that $\sqrt[d_i]{m_j}^{d_{ij}} \in \mathbf{Z}$. For $j = 1, \ldots, l$, compute $t_j = \mathrm{lcm}(d_{1j}, \ldots, d_{kj})$.

`Step 3:` Compute $d = \mathrm{lcm}(d_1, \ldots, d_k)$ and choose l d-th roots of unity $\zeta_1, \ldots, \zeta_l$ uniformly and independently at random.

`Step 4:` In E replace the radical $\sqrt[d_i]{n_i}$ by

$$\prod_{j=1}^{l} \zeta_j^{d^2 e_{ij}/t_j d_i} \sqrt[d_i]{m_j}^{e_{ij}} = \left(\prod_{j=1}^{l} \zeta_j^{d^2 e_{ij}/t_j d_i} \right) \sqrt[d_i]{n_i}.$$

Call this new radical expression $\overline{E}$.

`Step 5:` Compute $\mathrm{u}(E)$. Approximate $\mathrm{val}(\overline{E})$ with absolute error less than $2^{-\lceil \log(\mathrm{u}(E)) \rceil - 1}$. If in absolute value this approximation is less than $2^{-\lceil \log(\mathrm{u}(E) \rceil - 1}$, output "zero", otherwise output "non-zero".

In the remainder of this section we will analyze the running time of this algorithm. For `Step 1` we can use a well-known procedure called factor-refinement. At any time during its execution factor-refinement maintains a list of integers m_j such that each n_i can be written as a product of the m_j's. Initially the list contains the n_i's. If there are two list elements m_s, m_t that are not relatively prime, factor-refinement computes $d = \gcd(m_s, m_t)$, replaces m_s and m_t by m_s/d and m_t/d, respectively, and adds d to its list. It is clear that eventually the list will contain integers that are relatively prime. An amortized analysis of factor-refinement was given by Bach et al. [1].

Lemma 3.1 *Let $n_1, \ldots, n_k$ be integers and denote their product by n. In time $\mathcal{O}(\log^2(n)$ integers $m_1, \ldots, m_l$ can be computed such that*

(i) $\gcd(m_i, m_j) = 1$ *for all* $i \neq j, 1 \leq i, j \leq l$,

(ii) Each n_i can be written as $n_i = \prod_{j=1}^{l} m_j^{e_{ij}}, e_{ij} \in \mathbf{N}$.

Within the same time bound the factorizations $n_i = \prod_{j=1}^{l} m_j^{e_{ij}}, e_{ij} \in \mathbf{N}$, can be computed.

Observe that l, the number of m_j's, can not be bounded by a function depending only on k, the number of input radicals. However, l is bounded by $\sum_{i=1}^{k} \log(|n_i|) \leq \text{size}(E)$.

In `Step 2` we are asked to compute for each $\sqrt[d_i]{m_j}$ the smallest d_{ij} such that $\sqrt[d_i]{m_j}^{d_{ij}} \in \mathbf{Z}$. For fixed i and j this can done in time polynomial in d_i and $\log m_j$ as follows. For $e = 1, \ldots, d_i - 1$ first approximate $\sqrt[d_i]{m_j}^e$ with absolute error less than $1/2$ to obtain the unique integer m with $|\sqrt[d_i]{m_j}^e - m| < 1/2$. Then check whether $m^{d_i} = m_j^e$.

`Step 3` and `Step 4` can clearly be done in time polynomial in size(E). To analyze `Step 5` observe that although we change the input radicals, the corresponding input radicals in E and $\overline{E}$ have the same absolute value. Therefore $\text{u}(E) = \text{u}(\overline{E})$. By definition of $\text{u}(\overline{E})$, this is an upper bound for val(v) of each internal node v of the expression $\overline{E}$. A straightforward error analysis shows that approximating the input radicals of $\overline{E}$ with absolute error less than 2^{-w}, where

$$w = 3\text{size}(E) + 2\lceil \log(|\text{u}(E)|) \rceil + 1,$$

leads to an approximation of $\text{val}(\overline{E})$ with absolute error less than $2^{-\lceil \log(\text{u}(E)) \rceil - 1}$, as required in `Step 5` of Algorithm Zero-Test. We assume that the radicals $\sqrt[d_i]{n_i}$ are represented in a way that allows for efficient approximation algorithms. The input radicals in $\overline{E}$ differ from the input radicals in E by powers of roots of unity. It follows from Brent's approximation algorithms for exp, log, and the trigonometric functions (see [5]) that these powers of roots of unity can be efficiently approximated. Hence, the input radicals of $\overline{E}$ can be approximated with absolute error $2^{-w}, w = 3\text{size}(E) + 2\lceil \log(|\text{u}(E)|) \rceil + 1$, in time polynomial in size(E) and $\log(|\text{u}(E)|)$. As mentioned, this implies that $\text{val}(\overline{E})$ can be approximated with absolute error $2^{-\lceil \log(\text{u}(E)) \rceil - 1}$ in time polynomial in $\log(|\text{u}(E)|)$ and size(E). Summarizing, we have shown

Lemma 3.2 *On input E the running time of Algorithm Zero-Test is polynomial in* size(E) *and* $\log(|\text{val}(E)|)$.

4 The error analysis

In this section we will analyze the error probability of Algorithm Zero-Test. We recall some basic facts and definitions from algebraic number theory. For readers not familiar with algebraic number theory we recommend [11]. A number $\alpha \in \mathbf{C}$ is called *algebraic*, if α is the root of some polynomial $f(X) \in \mathbf{Q}[X]$. A polynomial $f(X) = \sum_{i=0}^{n} f_i X^i \in \mathbf{Q}[X]$ is called monic, if $f_n = 1$. An algebraic number $\alpha \in \mathbf{C}$ is called an *algebraic integer*, if it is the root of a monic polynomial with coefficients in $\mathbf{Z}$. The *minimal polynomial* of an algebraic number $\alpha \in \mathbf{C}$ is the smallest degree monic polynomial in $\mathbf{Q}[X]$ with root α. If $f(X)$ is the minimal polynomial of α, then the roots $\alpha_0 = \alpha, \ldots, \alpha_{n-1}$ of f are called the *conjugates* of α. Product and sum of algebraic integers are algebraic integers. Product, sum, and quotients of algebraic numbers are algebraic numbers. Since arbitrary roots of integers are algebraic integers, we see that the value of an arbitrary radical expression is an algebraic number and that the value of a division-free algebraic expression is an algebraic integer.

The error analysis for Algorithm Zero-Test is based on the following two lemmas. The first one was originally formulated and used by Chen and Kao in [8].

Lemma 4.1 *Let α be an algebraic integer with conjugates $\alpha_0 = \alpha, \alpha_1, \ldots, \alpha_{n-1}$. Assume that $|\alpha_i| \leq 2^B, B \in \mathbf{N}$. For $b \in \mathbf{N}$, with probability at most $B/(b+B)$ a random conjugate α_i of α satisfies $|\alpha_i| \leq 2^{-b}$.*

Lemma 4.2 *Let E be a real division-free expression with input radicals $\sqrt[d_1]{n_1}, \ldots, \sqrt[d_k]{n_k}$ and let $\overline{E}$ be constructed as in Algorithm Zero-Test. Then* $\mathrm{val}(\overline{E})$ *is a random conjugate of* $\mathrm{val}(E)$ *chosen according to the uniform distribution.*

Both lemmas will be proven below. Let us show that they imply

Corollary 4.3 *Let E be a real division-free expression with input radicals $\sqrt[d_1]{n_1}, \ldots, \sqrt[d_k]{n_k}$. If* $\mathrm{val}(E) = 0$*, then on input E Algorithm Zero-Test will output "zero". If* $\mathrm{val}(E) \neq 0$*, then Algorithm Zero-Test will output "non-zero" with probability at least* $1/2$.

Proof: By Lemma 4.2 Algorithm Zero-Test generates an expression $\overline{E}$ whose value is a random conjugate of $\mathrm{val}(E)$. We already noted that $\mathrm{u}(E) = \mathrm{u}(\overline{E})$ (see page 7). In particular, the conjugates of $\mathrm{val}(E)$ are bounded in absolute value by $\mathrm{u}(E)$.

If $\mathrm{val}(E) = 0$ then its only conjugate is 0 itself. Hence the approximation in `Step 5` will result in a number bounded in absolute value by $2^{-\lceil \log(\mathrm{u}(E)) \rceil - 1}$. Therefore, the answer of Algorithm Zero-Test will be "zero".

If $\mathrm{val}(E)$ is non-zero, then the approximation to $\mathrm{val}(\overline{E})$ can be bounded in absolute value by $2^{-\lceil \log(\mathrm{u}(E)) \rceil - 1}$ if and only if $\mathrm{val}(\overline{E})$ is a conjugate of $\mathrm{val}(E)$ that is bounded in absolute value by $2^{-\lceil \log(\mathrm{u}(E)) \rceil}$. Applying Lemma 4.1 to $\mathrm{val}(E)$ with $B = b = \lceil \log(\mathrm{u}(E)) \rceil$ proves that this happens with probability at most $1/2$. ◻

Together with Lemma 3.2, Corollary 4.3 proves Theorem 2.2 for division-free real radical expressions.

We mentioned earlier (see page 5) that if the required error probability $\epsilon = 2^{-e}$ is not too small, in practice we can do better than run Algorithm Zero-Test e times with independent random bits. We now want to make this statement more precise.

Set $b = \lceil 1/\epsilon \rceil - 1$. Assume that in `Step 5` of Algorithm Zero-Test instead of approximating $\mathrm{val}(\overline{E})$ with absolute error $2^{-\lceil \mathrm{u}(E) \rceil - 1}$ we approximate it with absolute error $2^{-b\lceil \mathrm{u}(E) \rceil - 1}$. Furthermore, we output "zero" if and only if the approximation is in absolute value less than $2^{-b\lceil \mathrm{u}(E) \rceil - 1}$. With these parameters a non-zero $\mathrm{val}(E)$ will be declared 0 by Algorithm Zero-Test if and only if $|\mathrm{val}(\overline{E})| \leq 2^{-b\lceil \mathrm{u}(E) \rceil}$. By Lemma 4.1 this happens with probability less than ϵ.

The running time of this algorithm will be polynomial in $1/\epsilon$ rather than $\log 1/\epsilon$. But for small ϵ it should be more practical than running $\log 1/\epsilon$ times Algorithm Zero-Test with error probability $1/2$. Moreover, with this approach we save on the number of random bits (see [8] for a more detailed discussion). Of course, a hybrid approach may also be interesting, that is run Algorithm Zero-Test several times with independent random bits, but each time with guaranteed error probability γ.

In the remainder of this section we prove Lemma 4.1 and Lemma 4.2.

Proof of Lemma 4.1: Let d be the number of conjugates that are at most 2^{-b} in absolute value. $|\prod_{i=0}^{n} \alpha_i|$ is the absolute value of the constant term of the minimal polynomial of α. Hence the product is at least 1. Together with the upper bound 2^B on $|\alpha_i|$ we obtain

$$1 \leq |\prod_{i=0}^{n} \alpha_i| \leq 2^{(n-d)B} 2^{-db}.$$

This implies $d/n \leq B/(b+B)$. ◻

To prove Lemma 4.2 we need a few more definitions and facts from algebraic number theory. Again we refer to [11] for readers unfamiliar with algebra and algebraic number theory. To prove Lemma 4.2 we need a few more definitions and facts from algebraic number theory. Again we refer to [11] for readers unfamiliar with algebra and algebraic number theory. For an algebraic number field F an isomorphism of F into a subfield of $\mathbf{C}$ whose restriction to $\mathbf{Q}$ is the identity, is called an *embedding* of F. The basic fact about embeddings is the following lemma.

Lemma 4.4 *Let F be an algebraic number field and let γ be an algebraic number whose minimal polynomial f over F has degree n. Every embedding σ of F can be extended in exactly n different ways to an embedding of $F(\gamma)$. An extension is uniquely determined by the image of γ, which must be one of the n distinct roots of $\sigma(f)$.*

From this lemma one can deduce

Corollary 4.5 *Let α be an algebraic number and let F be an algebraic number field containing α. If σ is an embedding of F chosen uniformly at random from the set of embeddings of F, then $\sigma(\alpha)$ is a conjugate of α chosen uniformly at random from the set of conjugates of α.*

We specialize these facts to radical expressions. If E is a radical expression with input radicals $\sqrt[d_1]{n_1}, \dots, \sqrt[d_k]{n_k}$, then $\mathrm{val}(E)$ is contained in $\mathbf{Q}(\sqrt[d_1]{n_1}, \dots, \sqrt[d_k]{n_k})$, that is the smallest field containing $\sqrt[d_1]{n_1}, \dots, \sqrt[d_k]{n_k}$. It does not seem easy to directly generate a random embedding of this field. However, we also have $\mathrm{val}(E) \in \mathbf{Q}(\sqrt[d]{m_1}, \dots, \sqrt[d]{m_l})$, where $d = \mathrm{lcm}(d_1, \dots, d_k)$ and the integers m_j are as constructed in `Step 1` of Algorithm Zero-Test. For this extension we have

Lemma 4.6 *Let $m_1, \dots, m_l$ be positive integers that are pairwise relatively prime. Assume that the radicals $\sqrt[d]{m_1}, \dots, \sqrt[d]{m_l}$ are real numbers. Let t_i be the smallest positive integer such that there is an integer c_i with $\sqrt[d]{m_i} = \sqrt[t_i]{c_i}$. Then t_i divides d and the minimal polynomial of $\sqrt[d]{m_i}$ over the field $\mathbf{Q}(\sqrt[d]{m_1}, \dots, \sqrt[d]{m_{i-1}})$ is given by $X^{t_i} - c_i$.*

Proof: It was shown by Siegel [14] that the minimal polynomial of $\sqrt[d]{m_i}$ over the field $\mathbf{Q}(\sqrt[d]{m_1}, \dots, \sqrt[d]{m_{i-1}})$ has the form

$$X^{t_i} - q_i \prod_{j=1}^{i-1} \sqrt[d]{m_j}^{e_{ij}}, q_i \in \mathbf{Q}, 0 \leq e_{ij} < t_j, t_i \text{ divides } d.$$

Hence

$$m_i^{t_i} = q_i^d \prod_{j=0}^{i-1} m_j^{e_{ij}}.$$

Write $q_i = c_i/p_i, c_i, p_i \in \mathbf{Z}, \gcd(c_i, p_i) = 1$. Since $m_i^{t_i}$ is an integer, p_i^d must divide $\prod_{j=0}^{i-1} m_j^{e_{ij}}$. Since the m_j's are pairwise relatively prime, $p_i^d = \prod_{j=0}^{i-1} m_j^{e_{ij}}$. Hence $m_i^{t_i} = c_i^d$, and the minimal polynomial of $\sqrt[d]{m_i}$ over $\mathbf{Q}(\sqrt[d]{m_1}, \dots, \sqrt[d]{m_{i-1}})$ is given by

$$X^{t_i} - c_i.$$

The lemma follows. ∎

For the embeddings of $\mathbf{Q}(\sqrt[d]{m_1}, \dots, \sqrt[d]{m_{i-1}})$ this translates to

Corollary 4.7 *Let $m_i, \sqrt[d]{m_i}, t_i, c_i$ be as above. An embedding σ of $\mathbf{Q}(\sqrt[d]{m_1}, \ldots, \sqrt[d]{m_l})$ is uniquely determined by l d-th roots of unity $\zeta_1, \ldots, \zeta_l$ such that $\sigma(\sqrt[d]{m_i}) = \zeta_i^{d/t_i} \sqrt[d]{m_i}$. Moreover, if the ζ_i are chosen uniformly and independently at random, then σ is chosen uniformly at random.*

Proof: By Lemma 4.6 and by Lemma 4.4, an embedding σ of$\mathbf{Q}(\sqrt[d]{m_1}, \ldots, \sqrt[d]{m_l})$ is defined by mapping $\sqrt[d]{m_i}$ onto some root of $X^{t_i} - c_i, i = 1, \ldots, l$. These roots are given by $\eta_i \sqrt[t_i]{c_i} = \eta_i \sqrt[d]{m_i}$, where η_i is an arbitrary t_i-th root of unity. Since every t_i-th root of unity η_i can be written as ζ_i^{d/t_i} for a d-th root of unity ζ_i, the first part of the lemma follows.

To prove the second part, observe that for each t_i-th root of unity η_i there are exactly d/t_i d-th roots of unity ζ_i such that $\zeta_i^{d/t_i} = \eta_i$. □

Proof of Lemma 4.2: From the previous corollary we know that by choosing d-th roots of unity $\zeta_j, j = 1, \ldots, l$, uniformly and independently at random, we choose a random embedding σ of $\mathbf{Q}(\sqrt[d]{m_1}, \ldots, \sqrt[d]{m_l})$. By choice of d every radical $\sqrt[d_i]{m_j}$ is an element of this field. As before, t_j is defined as the smallest integer such that there is an integer c_j with $\sqrt[d]{m_j} = \sqrt[t_j]{c_j}$. Hence

$$\sqrt[d_i]{m_j} = \sqrt[d]{m_j}^{d/d_i} = \sqrt[t_j]{c_j}^{d/d_i}.$$

This implies

$$\sigma(\sqrt[d_i]{m_j}) = \sigma(\sqrt[d]{m_j})^{d/d_i} = \zeta_j^{d^2/d_i t_j} \sqrt[d_i]{m_j}.$$

We need to show that Algorithm Zero-Test correctly computes t_j. The algorithm computes t_j as $t_j = \mathrm{lcm}(d_{1j}, \ldots, d_{kj})$, where d_{ij} is the smallest integer such that $\sqrt[d_i]{m_j}^{d_{ij}} \in \mathbf{Z}$. Let $m_j = \prod p_h^{e_{hj}}$ be the prime factorization of m_j. Let e_j be the greatest common divisor of the exponents e_{hj}. Any positive integer t with $\sqrt[d_i]{m_j}^t \in \mathbf{Z}$ must satisfy

$$\frac{e_{hj} t}{d_i} \in N, \text{ for all } h.$$

Hence t must be a multiple of $d_i / \gcd(e_{hj}, d_i)$ for all h. Therefore d_{ij} is the least common multiple of the integers $d_i / \gcd(e_{hj}, d_i)$. This least common multiple is given by

$$d_{ij} = d_i / \gcd(d_i, e_j) = \mathrm{lcm}(d_i, e_j)/e_j.$$

Similarly $t_j = \mathrm{lcm}(d, e_j)/e_j$. We obtain

$$\begin{array}{rlcl} & e_j * \mathrm{lcm}(d_{1j}, \ldots, d_{kj}) & = & \mathrm{lcm}(e_j d_{1j}, \ldots, e_j d_{kj}) \\ = & \mathrm{lcm}(\mathrm{lcm}(d_1, e_j), \ldots, \mathrm{lcm}(d_k, e_j)) & = & \mathrm{lcm}(\mathrm{lcm}(d_1, \ldots, d_k), e_j) \quad = \quad \mathrm{lcm}(d, e_j). \end{array}$$

Hence

$$\mathrm{lcm}(d_{1j}, \ldots, d_{kj}) = \mathrm{lcm}(d, e_j)/e_j = t_j.$$

So far, we have shown that $\zeta_j^{d^2/d_i t_j} \sqrt[d_i]{m_j}, i = 1, \ldots, k, j = 1, \ldots, l$, is the image of $\sqrt[d_i]{m_j}$ under a random embedding of σ of $\mathbf{Q}(\sqrt[d]{m_1}, \ldots, \sqrt[d]{m_l})$. Then

$$\sigma(\sqrt[d_i]{n_i}) = \sigma\left(\prod_{j=1}^{l} \sqrt[d_i]{m_j}^{e_{ij}}\right) = \left(\prod_{j=1}^{l} \zeta_j^{d^2 e_{ij}/t_j d_i}\right) \sqrt[d_i]{n_i}, i = 1, \ldots, k,$$

as constructed in Algorithm Zero-Test, is the image of $\sqrt[d_i]{n_i}$ under the random embedding σ. Hence, $\sigma(\mathrm{val}(E)) = \mathrm{val}(\overline{E})$. Corollary 4.4 shows that $\mathrm{val}(\overline{E})$ is a random conjugate of $\mathrm{val}(E)$ chosen uniformly at random from the set of conjugates of $\mathrm{val}(E)$. □

5 Expressions with divisions

If a radical expression E contains divisions, Lemma 4.1 is not applicable, since $\mathrm{val}(E)$ need not be an algebraic integer. However, if E contains divisions, then E can be transformed into an expression E', in which only the output node is labeled with /. This can be done by separately keeping track of the numerator and denominator of E and applying the usual arithmetic rules for adding, multiplying, and dividing quotients.

To be more specific, to obtain E' replace every internal node v in E by two nodes $v^{(n)}$ and $v^{(d)}$. If v is a node with edges from nodes v_1, v_2 directed into v and if v is labeled $*$, then $v^{(n)}$ and $v^{(d)}$ are labeled $*$. Edges from $v_1^{(n)}$ and $v_2^{(n)}$ are directed into $v^{(n)}$. Analogously for $v^{(d)}$. If v is labeled /, then $v^{(n)}, v^{(d)}$ are labeled $*$. Edges from $v_1^{(n)}, v_2^{(d)}$ are directed into $v^{(n)}$, and similarly for $v^{(d)}$. If v is labeled $+$ or $-$, then nodes w_1, w_2, both labeled $*$, are added to E'. $v^{(n)}$ is labeled $+$ or $-$, $v^{(d)}$ is labeled $*$ and edges are constructed as in Figure 1. Finally add a new output node w that is labeled /. If v is the

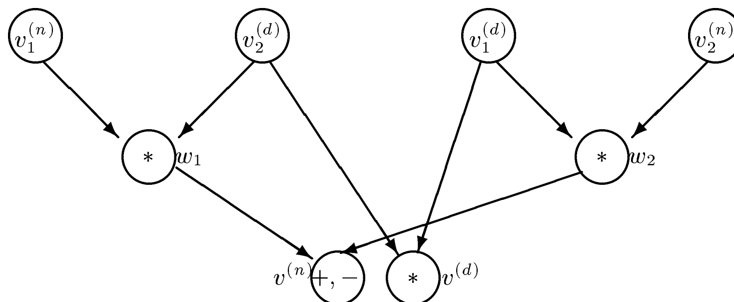

Figure 1: Replacing an addition

output node of E, then there are edges from $v^{(n)}$ and $v^{(d)}$ directed into w.

The size of E' is $\mathcal{O}(\mathrm{size}(E))$ and $\mathrm{val}(E') = \mathrm{val}(E)$. Furthermore, if v is the output node of E', and if w is the node where the numerator of v is computed, then $\mathrm{val}(E) = 0$ if and only if $\mathrm{val}(w)$ of node w in E' is zero. Restricting E' to the subgraph induced by the edges lying on paths from the input nodes to w, we obtain a division-free expression D with $\mathrm{val}(D) = \mathrm{val}(w)$. To check whether $\mathrm{val}(E) = 0$, we can use Algorithm Zero-Test with input D. By definition of $\mathrm{u}(E)$, we get $\mathrm{u}(D) \leq \mathrm{u}(E)$. Since E' and D can easily be constructed in time polynomial in $\mathrm{size}(E)$, the analysis for Algorithm Zero-Test given in the previous section proves Theorem 2.2 in the general case.

Acknowledgments I would like to thank Helmut Alt, Emo Welzl and Hans-Martin Will for stimulating and clarifying discussions.

References

[1] E. Bach, J. Driscoll, J. O. Shallit, "Factor Refinement", *Journal of Algorithms*, Vol. 15, pp. 199-222, 1993.

[2] J. Blömer, "Computing Sums of Radicals in Polynomial Time", *Proc. 32nd Symposium on Foundations of Computer Science* 1991, pp. 670-677.

[3] J. Blömer, "Denesting Ramanujan's Nested Radicals", *Proc. 33nd Symposium on Foundations of Computer Science* 1992, pp. 447-456.

[4] J. Blömer, "Denesting by Bounded Degree Radicals", *Proc. 5th European Symposium on Algorithms*, Lecture Notes in Computer Science, Vol. 1284, pp. 53-63, 1997.

[5] R. P. Brent, "Fast Multiple-Precision Evaluation of Elementary Functions", *Journal of the ACM*, Vol. 23, pp. 242-251, 1976.

[6] C. Burnickel, K. Mehlhorn, S. Schirra, "How to Compute the Voronoi Diagrams of Line Segments", *Proc. 2nd European Symposium on Algorithms*, Lecture Notes in Computer Science, Vol. 855, pp. 227-239, 1994.

[7] C. Burnikel, R. Fleischer, K. Mehlhorn, S. Schirra, "A Strong and Easily Computable Separation Bound for Arithmetic Expressions Involving Radicals", *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, 1997, pp. 702-709.

[8] Z. -Z. Chen. M. -Y. Kao, "Reducing Randomness via Irrational Numbers", *Proc. 29th Symposium on Theory of Computing*, 1997, pp. 200-209.

[9] G. Horng, M. -D. Huang, "Simplifying Nested Radicals and Solving Polynomials by Radicals in Minimum Depth", *Proc. 31st Symposium on Foundations of Computer Science* 1990, pp. 847-854.

[10] S. Landau, "Simplification of Nested Radicals", *SIAM Journal on Computing* Vol. 21, No. 1, pp 85-110, 1992.

[11] S. Lang, *Algebra*, 3rd edition, Addison-Wesley, 1993.

[12] G. Liotta, F. P. Preparata, R. Tamassia, "Robust Proximity Queries in Implicit Voronoi Diagrams", *Technical Report RI 02912-1910*, Center for Geometric Computation, Department of Computer Science, Brown University, 1996.

[13] M. Mignotte, "Identification of Algebraic Numbers", *Journal of Algorithms*, Vol. 3(3), 1982.

[14] C. L. Siegel, "Algebraische Abhängigkeit von Wurzeln", *Acta Arithmetica*, Vol. 21, pp. 59-64, 1971.

[15] C. K. Yap, "Towards Exact Geometric Computation", *Proc. Canadian Conference on Computational Geometry*, 1993, pp. 405-419.

[16] C. K. Yap, T. Dubé "The Exact Computation Paradigm", in D. Z. Du, F. Hwang, editors, *Computing in Euclidean Geometry*, World Scientific Press, 1995.