Interval-symbol method with correct zero rewriting: Reducing exact computations to obtain exact results

Kiyoshi Shirayanagi
kiyoshi.shirayanagi@is.sci.toho-u.ac.jp
Department of Information Science
Toho University
274-8510
Japan

Hiroshi Sekigawa
sekigawa@rs.tus.ac.jp
Department of Mathematical Information Science
Tokyo University of Science
162-8601
Japan

Abstract

We propose an *interval-symbol method with correct zero rewriting* or *ISCZ method* for short, which is an extension of the so-called interval method, to obtain exact results while reducing exact computations. Namely, this method uses not only a floating-point interval but also a symbol for each real coefficient of a polynomial. Symbols are used to keep track of the execution path of the original algorithm with exact computations. Moreover, the method has the rule of *zero rewriting* from stabilization techniques, which rewrites an interval into the zero interval if the interval contains zero. The key point is that at each stage of zero rewriting, one checks to see if the zero rewriting is really correct; namely, an interval considered to be zero is really zero by exploiting the associated symbol. Therefore, one can expect that it mostly uses floating-point computations; exact computations are only performed at the stage of zero rewriting and in the final evaluation to get the exact coefficients. As another important merit, one does not need to check the correctness of the output. The results of Maple experiments on convex hull construction indicate that our method is very effective for non-rational coefficients.

1 Introduction

While usually polynomials with exact coefficients are obtained by executing an algorithm with exact computations in all steps, the new method we propose makes the exact computations

as few as possible, or equivalently, it makes the floating-point computations as many as possible, but nevertheless obtains polynomials with exact coefficients. This method is based on stabilization techniques [10].

Stabilization techniques were proposed to moderate instability of algebraic algorithms derived from the use of floating-point computations. These techniques are fundamentally the interval method using zero rewriting, where if 0 lies within an interval, the interval is rewritten to an interval with center 0 and radius 0. When using a stabilized algorithm, if we increase the precision of the floating-point coefficients of the input, the output will converge to the true output. Here, precision denotes the number of decimal digits in the mantissa of a floating-point representation.

Let us briefly mention two categories of application of stabilization techniques. The first category is one in which we estimate the precision for stabilization in advance, and with that precision run the stabilized algorithm to get the reasonably approximate result. A typical example is floating point Gröbner basis [7]. However, there is the problem of estimating which precision will give a stabilized result. As the first success, Khungurn and the authors solved this problem for GCD computing algorithms [5].

The second category is one in which we repeatedly run the stabilized algorithm while increasing the precision until the correct result is obtained. An example of the second category is a method of Gröbner basis conversion using floating-point computations [8]. Another example is the *ISZ method* [11, 12]. In this method, we used not only a floating-point interval but also a symbol for each coefficient of a polynomial. Symbols are used to keep track of the execution path of the original algorithm with exact computations. The notion of *interval-symbols (IS's)*, which is thought of as an extension of intervals, will also be used in the new method we propose. The details on this notion will be described in Section 3.1. After the algorithm terminates, one evaluates the symbols of the output and get a result with exact coefficients. If necessary, one goes back to the starting point and increases the precision of the floating-point input and computation until a correct result is obtained. The details on the ISZ method can be found in [9].

The new method is similar to the ISZ method from the viewpoint of using symbols, but is definitely different in its treatment of zero rewriting. Whereas the ISZ method runs until the end regardless of whether zero rewriting is correct, the new method checks at each stage of zero rewriting if the zero rewriting is correct by evaluating the associated symbols. Here, zero rewriting is correct if the evaluated value is really zero. Therefore, one can expect that this method mainly involves floating-point computations; exact computations are only performed in zero rewriting and in the final evaluation to get the exact coefficients. In other words, one can reduce the exact computations of nonzero coefficients and still obtain correct results with exact coefficients. Moreover, unlike the ISZ method, *one does not need to check the correctness of the output*. The advantages of the new method will be discussed in detail in Section 3.2.

In the work [9] at SNC 2009, we reported experimental results on Buchberger's algorithm that computes Gröbner bases. At that time, we could not claim that the ISCZ method is always more effective than the purely exact approach which uses exact computations in all steps The main reason for this is that symbols tend to grow drastically due to the presence of recursion when executing reduction of polynomials. In the present paper, we perform new experiments on convex hull construction with Graham's algorithm that has no recursion, to show the real advantage of the ISCZ method over the purely exact method. Moreover, for Gröbner basis

computation, we introduced *symbol lists* to circumvent the growth of symbols as explained in Section 3.3. Surprisingly, in the case of convex hull construction, we discovered that the ISCZ method *not* using symbol lists is much more faster than one using symbol lists.

Related researches about using approximate computations to obtain exact results include [3, 4] for real algebraic number computations and real root isolation. Both of them employ interval arithmetic, and if interval arithmetic is not successful, they resort to exact computations without a symbol. On the other hand, *lazy arithmetic* [2] utilizes an IS-like notion called a *lazy number* in computational geometry. While they treat only rational numbers as inputs, we can treat non-rational numbers too.

This paper is organized as follows. Section 2 briefly reviews the theory of stabilization techniques, namely, the interval method with zero rewriting. Section 3 describes the proposed method, and Section 4 gives experimental results on convex hull construction. The conclusion mentions future work.

2 Review on the Interval Method with Zero Rewriting

We briefly describe the stabilization techniques, namely, the interval method with zero rewriting for the following class of algorithms:

- Input, intermediate, and output data are from the polynomial ring $R[x_1, \ldots, x_m]$, where R is a subring of the real numbers.
- Operations on data are polynomial functions over R.
- Predicates on data have *zero discontinuity*.

Let us explain the discontinuity set of a predicate. Predicates map polynomials to the set of two elements: {"TRUE", "FALSE"}. A predicate p is discontinuous at $f \in R[x_1, \ldots, x_m]$ if there exists a sequence $\{f_i\}_i$ in $R[x_1, \ldots, x_m]$ where $f_i \to f$ but $p(f_i) \not\to p(f)$ (i.e. for any M there is $M' \geq M$ s.t. $p(f_{M'}) \neq p(f)$). Here, $f_i \to f$ means that f_i converges *coefficientwise* to f, where convergence in the coefficient ring R is defined from the usual topology on the real numbers. A predicate is said to have *zero discontinuity* if it has no discontinuous points – i.e. it is continuous – or the only discontinuous point of the predicate is *zero*, i.e. a polynomial all of whose coefficients are 0, such as in the if statement "If $C = 0$ then ... else ...". Instead of "If $C = 0$", "If $C \geq 0$" or "If $C > 0$" can also be considered.

Let us call the class of algorithms that satisfy the above three conditions, *algebraic algorithms with zero discontinuity*.

Now the interval method with zero rewriting consits of the following three points:

R-to-Interval Make each input coefficient a into a (circular) interval $[\tilde{a}, \alpha]$ of a, where $\tilde{a}$ is a floating-point approximation of a with a specified precision μ and α is an error bound.

Interval Arithmetic Employ interval arithmetic (see [1]) for addition, subtraction, and multiplication between intervals.

Zero Rewriting For any intermediate interval $[E, \epsilon]$, rewrite $[E, \epsilon]$ to $[0, 0]$ if $|E| \leq \epsilon$ in the course of executing $\mathcal{A}$.

Let us call this procedure $Stab(\mathcal{A})$. The details can be found in [10].

Before we can discuss the effect of $Stab(\mathcal{A})$, we must define a few concepts. If we write an input $f \in R[x_1, \ldots, x_m]$ as $f = \sum_{\boldsymbol{\alpha}} a_{\boldsymbol{\alpha}} x^{\boldsymbol{\alpha}}$ using multi-index notation, an approximation sequence $\{Int(f)_j\}_j$ for f can be defined as $Int(f)_j = \sum_{\boldsymbol{\alpha}} [(a_{\boldsymbol{\alpha}})_j, (\epsilon_{\boldsymbol{\alpha}})_j]\, x^{\boldsymbol{\alpha}}$, where for all $\boldsymbol{\alpha}$ and for any j, we have $|a_{\boldsymbol{\alpha}} - (a_{\boldsymbol{\alpha}})_j| \leq (\epsilon_{\boldsymbol{\alpha}})_j$, and $(\epsilon_{\boldsymbol{\alpha}})_j \to 0$ as $j \to \infty$. In this case, we simply write $Int(f)_j \to f$. Most typically, $(a_{\boldsymbol{\alpha}})_j$ is a floating-point approximation of $a_{\boldsymbol{\alpha}}$ with precision j.

The fundamental theorem is as follows.

Theorem 1 (Stability Theorem [10]) *Let algorithm $\mathcal{A}$ be an algebraic algorithm with zero discontinuity, and let it terminate normally for an input $f \in R[x_1, \ldots, x_m]$. Then, for any approximation sequence $\{Int(f)_j\}_j$ for f, there exists n such that if $j \geq n$, $Stab(\mathcal{A})$ terminates normally for $Int(f)_j$, and $Stab(\mathcal{A})(Int(f)_j) \to \mathcal{A}(f)$.*

3 ISCZ method

3.1 Interval-Symbols

We introduce coefficients based upon intervals with formal symbols for polynomials. These were also used for the ISZ method. As before, intervals play the role of "reasonable" approximations with regard to the input, intermediate, and output coefficients. The intervals have the property that they always contain an exactly correct point. Symbols play the role of keeping track of the coefficients appearing in the course of a given algorithm. For example, let input coefficients be 1/3 and 1/9. The intervals for 1/3 and 1/9 are [.333, .0005] and [.111, .0005] with precision digits 3. Let the symbols for 1/3 and 1/9 be, say, s and t (indeterminates). Combining the interval and symbol for each, we can write $[[.333, .0005], s]$ and $[[.111, .0005], t]$ for these input coefficients. Next consider an operation, say, addition. We define addition between $[[.333, .0005], s]$ and $[[.111, .0005], t]$ as:

$$[[.333, .0005], s] + [[.111, .0005], t] = [[.333, .0005] + [.111, .0005], \dot{+}(s, t)]$$

We use interval arithmetic for $[.333, .0005] + [.111, .0005]$. The symbol part $\dot{+}(s, t)$ is still a formal symbol, namely a formal sequence of the symbols "$\dot{+}$", "(", "s", ",", "t", and ")". If necessary, one may replace $\dot{+}(s, t)$ by another symbol or code. In any case, one has only to keep track of operations and the associated arguments in each step of an algorithm. There might be some clever methods for efficient *symbolization* or *coding*. Section 3.3 gives one such example. When the algorithm finishes executing, one can recover or evaluate the final symbols of the output to see what their values actually are. Let us call such a pair of an interval and a symbol, an *interval-symbol*, or simply an IS. For an IS $u = [[a, \alpha], s]$, we shall call the interval part $[a, \alpha]$ *the I part of* u, and the symbol part s *the S part of* u.

Conversely, given a symbol, its exact value can be obtained by substituting the original values from R of the input coefficients for the respective input symbols. In this substitution, operation symbols such as "$\dot{+}$", "$\dot{\times}$" are given the mathematical meanings of their associated functions. The reason why the S part works is that it has information about how the coefficient arises from the original input coefficients, i.e., how it is generated from the input symbols. As a simple example, let $S = \dot{\times}(s, \dot{+}(t, u))$ be the S part of an IS, where s, t and u are input symbols for $\sqrt{2}$, 3, and 4, respectively. Then if one performs the substitution $s = \sqrt{2}$, $t = 3$, and $u = 4$

for S, one should obtain the result $\sqrt{2} \times (3+4) = 7\sqrt{2}$. This implies that the correct coefficient should be $7\sqrt{2}$. By *evaluating a symbol*, we mean obtaining the exact value from the symbol in this manner.

3.2 Theory

Given an algebraic algorithm with zero discontinuity, we propose a new method for reducing the steps of exact computations to obtain exact results. The fundamental idea is that like the ISZ method we use IS coefficients for keeping track of coefficients, but at each stage of zero rewriting we guarantee that the zero rewriting is really correct by evaluating the symbol. In contrast, the ISZ method runs until the end regardless of whether intermediate zero rewriting is correct.

The description of the method is as follows:

R-to-IS Make each input coefficient a into a pair $[[\tilde{a}, \alpha], Symbol_a]$, where $[\tilde{a}, \alpha]$ is an interval of a with a (preselected) precision, and $Symbol_a$ is an indeterminate representing a.

IS Arithmetic Perform arithmetic between IS's:

$$[[A, \alpha], s] + [[B, \beta], t] = [[A, \alpha] + [B, \beta], \dot{+}(s, t)]$$
$$[[A, \alpha], s] - [[B, \beta], t] = [[A, \alpha] - [B, \beta], \dot{-}(s, t)]$$
$$[[A, \alpha], s] \times [[B, \beta], t] = [[A, \alpha] \times [B, \beta], \dot{\times}(s, t)]$$

Correct Zero Rewriting For any intermediate IS $[[E, \epsilon], s]$, if $|E| \leq \epsilon$, then evaluate the symbol s to make the associated real number $r(s)$. If $r(s) = 0$, proceed to the next step; if otherwise, raise the precision and go back to **R-to-IS**.

IS-to-R Substitute the original input coefficient values for the respective symbols to evaluate the S part of the output.

We refer to this method as the ISCZ method (the IS method with *correct* zero rewriting).

The execution path of $Stab(\mathcal{A})$ evaluated on $Int(f)_j$ with some precision j will be exactly the same as that of $\mathcal{A}$ evaluated on f, when every zero rewriting throughout the algorithm is *correct*. In the ISCZ method, at each step of zero rewriting, the correctness of the zero rewriting is checked. From the stability theorem, there is a precision for which every zero rewriting is correct. Therefore, the ISCZ method terminates in a finite number of steps, and the symbol of each IS coefficient of the output gives the correct output.

We thus have the following theorem:

Theorem 2 (Termination and Correctness of the ISCZ Method) *Suppose that $\mathcal{A}$ terminates with an input I. Then, the ISCZ method for $\mathcal{A}$ always terminates in a finite number of steps and gives the correct result, i.e. the same result as the output of $\mathcal{A}(I)$.*

The advantage of the ISCZ method is that unlike the ISZ method *one does not need to check the correctness of the output.* One can skip exact computations (evaluation of s) of any IS $[[E, \epsilon], s]$ and use only floating-point computations unless $|E| \leq \epsilon$. In other words, one can reduce the exact computations of nonzero coefficients. Therefore, the ISCZ method is effective when there are more cases where $|E| > \epsilon$ than those where $|E| \leq \epsilon$.

3.3 Symbol List

The ISCZ method can obviously cause the S parts of IS's to grow. Although evaluations can be avoided in the intermediate stages, if the symbols themselves become too big, the method would fail because of memory exhaustion. Here, we discuss a way to circumvent the growth of the S parts.

If one always uses the IS's themselves, the size of the S parts would grow explosively and the method might be impractical. Here, we give an efficient method using integers – address numbers – instead of symbols themselves to keep track of the S parts.

We use pairs of the form $[[A, \alpha], M]$, where $[A, \alpha]$ is an interval and M is an indeterminate or integer, and a list containing information as to how each pair is constructed from other pairs. Let us call the list the *symbol list*. After executing the algorithm, we will recover the IS's from the symbol list.

1. Do **R-to-IS** conversion as described in Section 3.1.
2. (Initialization) Let N be 0 and the symbol list $\mathcal{L}$ be the empty list.
3. An operation $*$ ($+$, $-$, $\times$ etc.) between pairs $[[A, \alpha], L]$ and $[[B, \beta], M]$ is defined as follows.

   First, compute the interval part and apply zero rewriting to the resulting interval $[C, \gamma] = [A, \alpha] * [B, \beta]$.

   If $[C, \gamma]$ is rewritten to $[0, 0]$, the result is $\mathbf{0}$ and do not change $\mathcal{L}$.

   Otherwise, increment N (i.e., $N := N + 1$), make the new coefficient $[[C, \gamma], N]$ and add $(\dot{*}, L, M)$ to the end of $\mathcal{L}$.
4. (Recovery) Let $[[A, \alpha], N]$ be a coefficient of a polynomial in the output. Convert the pair $[[A, \alpha], N]$ into the associated IS as follows:

   If N is an indeterminate, do nothing. The pair $[[A, \alpha], N]$ is the IS.

   Otherwise (i.e., N is an integer), convert N into the associated symbol S by rewriting N to the Nth element of $\mathcal{L}$ recursively. The resulting pair $[[A, \alpha], S]$ is the IS.

The symbol list was sometimes useful for Gröbner basis computation, but as we will see in the next section, it may not always be necessary.

4 Experiments on Convex Hull Construction

We applied the ISCZ method to construct two-dimensional convex hulls. The convex hull of a set S of points is the smallest convex set containing S. Hereafter, we assume that S is a finite set in $\mathbb{R}^2$ so that the convex hull of S is a convex polygon. Therefore, for a given finite set S of points, "to find the convex hull of S" means "to give an ordered list of vertices of the convex hull of S."

There are several well-known algorithms for constructing two-dimensional convex hulls, and these include Graham's algorithm, Jarvis' algorithm, and Bentley-Shamos' algorithm (see [6] for instance). A basic routine in these algorithms is to determine whether P_3 is to the left of,

to the right of, or on the directed line from P_1 to P_2, for three given points $P_i = (x_i, y_i)$ ($i = 1$, 2, 3). This is equivalent to determining the signature of the determinant

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix},$$

because the determinant is positive, negative or zero, depending on whether P_3 is to the left of, to the right of or on the directed line P_1 to P_2.

Graham's algorithm is as follows.

Graham's algorithm.

Input: A finite set $S = \{P_1, P_2, \ldots, P_m\} \subset \mathbb{R}^2$.

Output: A list of vertices for the convex hull of S.

1. [Base point] Pick a point in S that will be a vertex of the convex hull (e.g., the point with the largest y-coordinate among the points with the smallest x-coordinate) and call it O.

2. [Sort] Sort the points in S, other than O, in increasing order of the arguments of the line segments from O to P_i. If more than one point have the same argument, select the one farthest from O and discard the others among them. Let Q_1, Q_2, ..., Q_n be the result of this operation, and put $Q_0 = O$.

3. [Sweep] Create the convex hull by determining whether Q_i is to the left of the directed line Q_{i-2} to Q_{i-1} or not. More precisely, the procedure can be described as follows:

$j := 0$
for i **from** 0 **to** n **do**
    $j := j + 1$
    $R_j := Q_i$
    while $j \geq 4$ **and** R_j is to the right of or on $\overrightarrow{R_{j-2}R_{j-1}}$ **do**
        $R_{j-1} := R_j$
        $j := j - 1$
return $[R_1, R_2, \ldots, R_j]$

Notice that Graham's algorithm is also an algebraic algorithm with zero discontinuity. We applied this algorithm to the following 8 examples through the implementation of our methods.

The first four examples consist of 5000 input points with integer coordinates.

Example 1 5000 *points with coordinates* (X, Y)*, where* X *and* Y *are randomly generated integers satisfying that* $0 \leq X, Y \leq 200$.

Example 2 5000 *points with coordinates* (X, Y)*, where* X *and* Y *are randomly generated integers satisfying that* $-200 \leq X, Y \leq 200$ *and* $X^2 + Y^2 \leq 200^2$.

Example 3 5000 *points with coordinates* (X, Y)*, where* X *and* Y *are randomly generated integers satisfying that* $0 \leq X, Y \leq 400$*,* $X^2 + Y^2 \leq 400^2$*, and* $Y^2 \leq 3X^2$*. That is, the points are bounded by a sector whose center is at the origin, radius is* 400*, and angle subtended by the arc at the center is* $\pi/3$.

Example 4 *The origin* $(0,0)$ *and* 4999 *points with coordinates* (X,Y)*, where* X *and* Y *are randomly generated integers satisfying that* $1 \leq X, Y \leq 6000$*,* $X^2+Y^2 \leq 6000^2$*, and* $\frac{9}{10} \leq \frac{Y}{X} \leq 1$*. That is, the points are bounded by a narrow sector whose center is at the origin and radius is* 6000*.*

The second four examples consist of 5000 input points with non-rational coordinates.

Example 5 5000 *points with coordinates* $(\sqrt{X}, \sqrt{Y})$*, where* X *and* Y *are randomly generated integers satisfying that* $0 \leq X, Y \leq 200$*.*

Example 6 5000 *points with coordinates* $(\mathrm{sign}(X)\sqrt{|X|}, \mathrm{sign}(Y)\sqrt{|Y|})$*, where* $\mathrm{sign}(X)$ *is the sign of* X*, and* X *and* Y *are randomly generated integers satisfying that* $-200 \leq X, Y \leq 200$ *and* $X+Y \leq 200$*.*

Example 7 5000 *points with coordinates* $(\sqrt{X}, \sqrt{Y})$*, where* X *and* Y *are randomly generated integers satisfying that* $0 \leq X, Y \leq 400$*,* $X+Y \leq 400$*, and* $Y \leq 3X$*. That is, the points are bounded by a sector whose center is at the origin, radius is* 20*, and angle subtended by the arc at the center is* $\pi/3$*.*

Example 8 *The origin* $(0,0)$ *and* 4999 *points with coordinates* $(\sqrt{X}, \sqrt{Y})$*, where* X *and* Y *are randomly generated integers satisfying that* $1 \leq X, Y \leq 6000$*,* $X+Y \leq 6000$*, and* $\frac{9}{10} \leq \frac{Y}{X} \leq 1$*. That is, the points are bounded by a narrow sector whose center is at the origin and radius is* $20\sqrt{15}$*.*

We performed the ISCZ method implemented in Maple 12, where the initial precision is 1 and the step of increasing precision is 1, on a computer with a dual core AMD(R) Opteron(R) processor (2.85GHz), 8GB RAM, and Linux(R) OS. The Maple code can be freely downloaded from the url http://www.rs.tus.ac.jp/~sekigawa/programs/.

We tried two versions of *using symbol lists* and *not using symbol lists*, because Graham's algorithm may not cause the symbols of IS's to grow so largely as Buchberger's algorithm does. Moreover, we introduced an *evaluation list* in which the value $r(s)$ is kept once s is evaluated, so that we can reuse it when we must evaluate s more than once. Using this list, we will count the number of evaluated symbols and how many times symbols were referenced.

Finally, we implemented Graham's algorithm in Maple 12, which gives convex hulls with exact coefficients by performing only exact computations in the real field. Let us call this function R_CH.

The experimental results are shown in Tables 1 and 2, respectively. Here, cpu time (1) means the total of the cpu times in seconds required for computations with the initial precision to the successful precision *using symbol lists*, and cpu time (2) means one *not using symbol lists*. "SP" and "ZR" stand for the successful precision and the number of intervals that were actually rewritten into zero by zero rewriting. "# of skipped nonzero coefficients" means the number of nonzero coefficients for which exact computations were skipped because of the "$|E| > \epsilon$" case at the precision that produced a success. "len. of symbol list" denotes the length of the final symbol list. "# of references" means the number of total references of the symbols, and the numbers in the braces mean the maximal reference times of a symbol.

From this we can conclude:

Table 1: Convex hull (integer coordinates)

Ex.	cpu time (1)	cpu time (2)	SP	ZR	# of skipped nonzero coefficients	len. of symbol list	# of evaluated symbols	# of references	R_CH
1	6510.6	8.2	6	1693	64920	130695	9403	9677 (22)	5.8
2	6262.9	9.3	6	774	67406	137748	4560	4662 (15)	6.6
3	5885.2	9.1	7	1059	66612	135189	5976	6167 (24)	6.3
4	4371.4	9.8	8	174	68978	142649	1062	1092 (18)	7.2

Table 2: Convex hull (non-rational coordinates)

Ex.	cpu time (1)	cpu time (2)	SP	ZR	# of skipped nonzero coefficients	len. of symbol list	# of evaluated symbols	# of references	R_CH
5	5039.5	10.6	9	660	67602	139214	2775	2806 (19)	72.1
6	6008.3	11.4	10	386	68439	141166	1697	1730 (31)	59.5
7	5127.4	11.1	9	560	68200	140508	2549	2582 (26)	72.7
8	5743.0	11.7	11	222	68695	141562	1215	1264 (29)	54.2

1. The ISCZ method without symbol lists is much more efficient than one with symbol lists.
2. The ISCZ method without symbol lists is more efficient than R_CH in the case of non-rational coefficients.
3. The number of evaluated symbols is much less than the length of the final symbol list, namely, *exact computations are essentially reduced.*

Moreover, in the case of convex hulls, it would be generally difficult to verify that the obtained result is a really correct convex hull without constructing the convex hull with an exact approach, but the ISCZ method theoretically *guarantees the correctness without any later verification.*

5 Conclusion

We proposed a method to obtain exact results using floating-point computations in most parts of the intermediate stages for the class of algebraic algorithms with zero discontinuity. Whether one should use symbol lists depends on the algorithm in question. When symbols grow largely due to recursion and so on, as in Buchberger's algorithm, we should use symbol lists. On the other hand, for algorithms whose structure is "flat" like Graham's algorithm, not using symbol lists would be better. Moreover, in the case of non-rational coefficients, the ISCZ method is generally more useful than it is in the case of rational coefficients, since it can avoid a significant amount of exact arithmetic in the intermediate stages. Our future work will include finding algorithms for which the ISCZ method is useful and practical examples in the real world.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 24500022.

References

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computations, Computer Science and Applied Mathematics*. Academic Press, 1983.

[2] M. O. Benouamer, P. Jaillon, D. Michelucci, and J-M. Moreau. A lazy exact arithmetic. In *Proc. 11th Symposium on Computer Arithmetic*, pages 242–249, 1993.

[3] J. R. Johnson. Real algebraic number computation using interval arithmetic. In *Proc. ISSAC 1992 (International Symposium on Symbolic and Algebraic Computation)*, pages 195–205, 1992.

[4] J. R. Johnson and W. Krandick. Polynomial real root isolation using approximate arithmetic. In *Proc. ISSAC 1997*, pages 225–232, 1997.

[5] P. Khungurn, H. Sekigawa, and K. Shirayanagi. Minimum converging precision of the QR-factorization algorithm for real polynomial. In *Proc. ISSAC 2007*, pages 227–234, 2007.

[6] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[7] K. Shirayanagi. Floating point Gröbner bases. *Mathematics and Computers in Simulation*, 42(4-6):509–528, 1996.

[8] K. Shirayanagi and H. Sekigawa. A New Groebner Basis Conversion Method Based on Stabilization Techniques. *Theoretical Computer Science*, 409:311–317, 2008.

[9] K. Shirayanagi and H. Sekigawa. Reducing Exact Computations to Obtain Exact Results Based on Stabilization Techniques In *Proc. SNC 2009 (Conference on Symbolic Numeric Computation)*, pages 191–198, 2009.

[10] K. Shirayanagi and M. Sweedler. A theory of stabilizing algebraic algorithms. Technical Report 95-28, Mathematical Sciences Institute, Cornell University, 1995. 92 pages. (http://www.lab.toho-u.ac.jp/sci/is/shirayanagi/msitr95-28.pdf)

[11] K. Shirayanagi and M. Sweedler. Automatic algorithm stabilization. In *ISSAC 1996 poster session abstracts*, pages 75–78, 1996.

[12] K. Shirayanagi and M. Sweedler. Remarks on automatic algorithm stabilization. *J. Symbolic Computation*, 26(6):761–766, 1998.