Remarks on automatic algorithm stabilization

Kiyoshi Shirayanagi
NTT Communication Science Laboratories

Moss Sweedler*
Department of Mathematics, Cornell University

Abstract

Approximate algebra typically deals with inexact input, perhaps due to constraints of physical measurement, and aims to arrive at appropriate approximate output. This article deals with inexact computation for another purpose. It is about a methodology utilizing inexact computation in conjunction with exact computation where the exact input is known and exact output is desired. The inexact computation is used to help avert the growth of intermediate expressions which frequently make simply using exact computation infeasible. The paper describes the methodology and how to make good use of it. The paper surveys several existing applications and proposes new areas or directions for future applications. It also briefly addresses frequently asked questions and misunderstandings.

1 Introduction

Most algorithms in computational algebra are intended to use exact computation. However, for practical purposes, in many cases, exact computation is useless because of the growth of intermediate expressions. It is helpful to

*Supported in part by the United States Army Research Office through the Army Center of Excellence for Symbolic Methods in Algorithmic Mathematics (ACSyAM), Mathematical Sciences Institute of Cornell University, Contract DAAL03-91-C-0027, and by the NSA.

be able to use some form of approximate computation, such as floating-point computation, to reduce the associated memory load and still obtain relevant results. Sasaki [25] proposed "approximate algebra" in 1988, and gave algorithms to do approximate polynomial factorization and to compute approximate greatest common divisors (GCD), [26, 20]. Additional suggested applications include computing numerical integrals of rational functions and plotting algebraic surfaces. The authors of the present paper proposed an alternative approach to the use of approximate computation [34] in 1995. [34] is a methodology leading to automatic algorithm stabilization and differs from approximate algebra, in premise and aim. Approximate algebra presupposes that inputs are inexactly given due to physical measurement etc., and aims to find a satisfactory approximate output from the scope of possible outputs. Research in this area is becoming increasingly active as found in, for example, [36, 11, 3, 14], and is extremely relevant to this special issue of *Symbolic-Numeric Algebra for Polynomials.* Our stabilization method presumes that inputs are given exactly. Thus an exact output exists and we want to obtain it. Our approach uses approximate computation in aid of exact computation in order to reduce computational expense while still yielding the exact output.[1]

In general if you invoke an algorithm repeatedly on a sequence of inputs converging to the exact input and utilize increasingly accurate approximate computation in the repeated invocations of the algorithm, the sequence of outputs so obtained will not approach the output obtained by invoking the algorithm with the exact input and utilizing exact computation. We call this output the exact output. An algorithm displaying such *instability* for some exact input is called an *unstable algorithm*[2]. Our approach converts an unstable algorithm into a new algorithm where if the new algorithm is run utilizing increasingly accurate approximate computation, the output will

[1]Earlier than Sasaki/Noda, Schönhage had proposed *Quasi-GCD* [27], where input polynomials are supposed to have exact coefficients and there is an oracle giving them to arbitrary accuracy. In that sense our approach may be similar to that of Schönhage. Another exciting approach is Sederberg-Chang's best linear common divisors [28]. Given a set of polynomials, their problem is to find a set of perturbation polynomials with minimum norm, where the perturbed polynomials have a linear common factor. Their method gives a strict solution using the Chebyshev polynomials.

[2]Note that this instability is a little different from what is called *numerical instability* in numerical analysis, see [37].

converge to the exact output of the original algorithm. The stabilization method has three points.

1. The syntactic structure of the algorithm is unchanged.
2. The coefficients are converted to *interval-coefficients* in the data set.
3. *Rewriting* is performed prior to predicate evaluation.

Interval-coefficients are coefficients which have the form of intervals from interval analysis, see, for example [1]. Steps (1) and (2) coincide with existing interval methods and in general (1) and (2) alone are not sufficient for stabilization. The key is (3). This is a method which rewrites an interval to a new interval at the *discontinuous point(s)* of a predicate. The discontinuous points of a predicate, such as 0 in a conditional instruction "If $X = 0$ then ...", are points where the execution path of the algorithm may branch upon evaluation of the predicate. A common cause of algorithm instability is that approximate computation causes a predicate to be evaluated incorrectly and the algorithm runs on a wrong execution path. *Rewriting* moderates the effect of predicate discontinuity. It rewrites an interval coefficient into (an interval signifying) the discontinuity point itself if the discontinuity point lies within the interval. Otherwise rewriting leaves an interval coefficient unchanged. This may have the same result as if exact computation with exact input had been done up until that point. In this case the modified algorithm passes through the branch-point in the same way as the original algorithm evaluated with exact computation on the exact input.

Of course, our stabilization techniques can not stabilize *all* algorithms. To be eligible, algorithms must satisfy some conditions. Roughly speaking, arithmetic operations used on the coefficients which appear in the algorithm must be extendable to interval-coefficients, and the set of discontinuity points (if any) of each predicate in the algorithm must be a discrete set. For example, using our techniques, it is not possible to stabilize the algorithm that outputs 1 if an input real number is rational, and 0 otherwise. On the other hand typical algorithms dealing with polynomials in computational algebra satisfy the above conditions. This is because addition, subtraction, multiplication, and division are most often used and they are easily extended to the domain of interval-coefficients, and in many cases the discontinuity point of a predicate is (or may be reduced to) zero (which of course is discrete).

There are two aims of this paper. One is to give a brief survey of the techniques presented in [34] and address frequently asked questions and misunderstood points about the methodology. The other aim is to propose new directions in which one can make effective use of the stabilization methodology in the future.

In the area of symbolic-numeric algebra for polynomials, we wish to emphasize that our techniques may be applied to other algorithms than the examples in this paper. We also wish to suggest that the reader apply them to an algorithm of personal interest where there is instability from a purely numerical approach.

This paper is organized as follows. In section 2, we briefly summarize the techniques for automatic algorithm stabilization [34] for a class of algebraic algorithms where the "coefficients" are from a field and each predicate has only the point zero as discontinuity (or no discontinuity). In section 3, we survey various applications which have appeared so far, and comment about each application. In section 4, we list frequently asked questions about the techniques and correct some of the misunderstood points. In section 5, we discuss how to make the best of the methodology, present a few examples, and propose new directions.

2 Automatic algorithm stabilization

We describe the techniques for automatic algorithm stabilization for the following class of algorithms:

1. The input data, intermediate data and output data are from the polynomial ring $R[x_1, \ldots, x_n]$, where R is a subfield of the real numbers.

2. Operations on data are: addition, subtraction, multiplication, and polynomial division with remainder.

3. Predicates on data have *zero discontinuity.*

Here polynomial division with remainder means computing the remainder of a *dividend* polynomial modulo a *divisor* polynomial as well as computing the *quotient* polynomial which gives that remainder. In one variable this is accomplished by the usual polynomial division algorithm. The only division between coefficients in the field R are divisions of the coefficients of the dividend polynomial by the coefficient of the lead term – i.e. the non-zero term of highest degree – of the divisor polynomial. In more than one variable the notion of *term order* is required to obtain suitable lead terms.

Next let us explain the discontinuity set of a predicate. Predicates map polynomials to the set of two elements:

$$\{\text{"TRUE"}, \text{"FALSE"}\}$$

A predicate p is discontinuous at $f \in R[x_1, \ldots, x_n]$ if there exists a sequence $\{f_i\}_i$ in $R[x_1, \ldots, x_n]$ 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 field R is defined from the usual topology in the real numbers. A predicate is said to have *zero discontinuity* if the only discontinuous point of the predicate is *zero*, i.e. the polynomial all of whose coefficients are 0, or the predicate has no discontinuous points – i.e. it is continuous.

Let us call the class of algorithms which satisfy the above three conditions, *algebraic algorithms with zero discontinuity.* Many algorithms in computational algebra are (or may be transformed into) algebraic algorithms with zero discontinuity. For example, the predicate which tests if "$X = 3$" may be transformed into two steps, the first being the operation (assignment to a new variable) "$Y = X - 3$" and the second being the predicate which tests if "$Y = 0$". This latter predicate has zero discontinuity.

For simplicity and clarity of description we have chosen an extremely limited set of permissible operations at (2). Nevertheless this still applies to a wide class of algorithms. We could permit other functions such as square root, differentiation, etc, and this is discussed in [34].

Let $\mathcal{A}$ be an algebraic algorithm with zero discontinuity. If $\mathcal{A}$ actually has a predicate with a discontinuity, $\mathcal{A}$ may be unstable with respect to limited precision computation. For example, let $\mathcal{B}$ be the following algorithm:

Initialize:	(X)
step_1:	$Y = 3X - 1$
step_2:	goto step_4 if $Y \geq 0$
step_3:	stop (0)
step_4:	stop (1)

Namely, given an input X, $\mathcal{B}$ returns 1 if $3X - 1 \geq 0$, and 0 otherwise. Consider the input $X = 1/3$. $\mathcal{B}(1/3)$ returns 1. However, with *any high* precision μ, floating point approximation $(1/3)_\mu$ of $1/3$ is $0.\overbrace{333\cdots 3}^{\mu\ digits}$, and hence $\mathcal{B}((1/3)_\mu)$ always returns 0. This is unstable behavior: $(1/3)_\mu \to (1/3)$ but $\mathcal{B}((1/3)_\mu) \not\to \mathcal{B}(1/3)$. Note that $\mathcal{B}$ has the predicate "if $Y \geq 0$" whose discontinuity set is $\{0\}$ and is an algebraic algorithm with zero discontinuity. The discontinuity causes the algorithm to be unstable.[3]

Automatic algorithm stabilization is done by interpreting an algorithm with modified semantics. The syntax of the algorithm is unchanged. Under the modified semantics the original coefficients are replaced by coefficient-pairs. Think of the new coefficient-pairs as intervals where the first component specifies the *center* of the interval and the second component – also called the error bound – specifies the *radius* of the interval[4]. We shall refer to the coefficient-pairs as *interval-coefficients*. Where the original algorithm specifies arithmetic operations on coefficients they are extended as interval operations to the new interval-coefficients. Immediately preceding evaluating predicates, *zero rewriting* is done. Zero rewriting changes certain interval-coefficients and leaves others unchanged. If 0 lies within the interval delineated by an interval-coefficient, the interval-coefficient may be rewritten or changed to the interval-coefficient with center 0 and radius 0. Once zero rewriting is done, the predicate is then evaluated on the first component or center of the interval-coefficient. The key property of zero rewriting is that it preserves convergence of a sequence of interval-coefficients and if the sequence converges to zero, then rewriting causes the sequence to *reach zero after a finite number of steps*. Hence the predicate will actually be evaluated at the discontinuity point zero. Without rewriting, the predicate may never get evaluated at the discontinuity point, and since it is discontinuous there, the algorithm may never have the original behavior[5].

[3]Using IEEE floating-point arithmetic (with a *guard digit*, [7]), $3 * (1.0/3.0) - 1$ may be exactly equal to 0 and there may be no problem. We choose "$3X - 1$" because it is easy to understand and also is correct if one requires a stabilization methodology which is independent of specific conventions such as IEEE floating-point arithmetic.

[4]If the coefficient field R were a subfield of the complex numbers then new coefficient pairs would be thought of as disks specified by a center and radius.

[5]Much is hidden in this word *behavior*. It actually refers to the path of execution flow of the algorithm. The path of execution flow is controlled by predicates and zero rewriting. This insures that when executed with high enough precision the modified algorithm will execute with the *exact execution path*, i.e. essentially the same path of execution flow as the original algorithm executed with exact computation. This is the key to why algorithm

Let us summarize the techniques for automatic algorithm stabilization. Given $\mathcal{A}$, an algebraic algorithm with zero discontinuity, let $\mathcal{I}(\mathcal{A})$ be the stabilized algorithm for $\mathcal{A}$. $\mathcal{I}(\mathcal{A})$ has the following features:

Interval Domain The domain is a set of polynomials having interval-coefficients as coefficients. An interval-coefficient is $[A, \alpha]$, where $A \in R$ and α is a non-negative real number. $[A, \alpha]$ should be thought of as $\{x \in R \mid |x - A| \leq \alpha\}$.

Interval Arithmetic Employ interval arithmetic (see [1]) for addition, subtraction, multiplication, and division between interval-coefficients. For binary operation $* \in \{+, -, \times, /\}$,

$$[A, \alpha] * [B, \beta] = [A * B, \gamma_*],$$

where γ_* is defined so as to satisfy:

$$|x - A| \leq \alpha, |y - B| \leq \beta \Rightarrow |x * y - A * B| \leq \gamma_*.$$

and

$$\alpha \to 0, \beta \to 0 \Rightarrow \gamma_* \to 0$$

Zero Rewriting Immediately preceding evaluating every predicate which has a discontinuity at zero, do zero rewriting:

For each interval-coefficient $[C, \gamma]$,

$$\text{rewrite } [C, \gamma] \text{ to } [0, 0] \text{ if } |C| \leq \gamma.$$

$\mathcal{I}(\mathcal{A})$ accepts interval polynomials as input and disseminates interval polynomials as output where *interval polynomials* are polynomial with interval-coefficient coefficients. Usually the original input $f \in R[x_1, \ldots, x_n]$[6] for $\mathcal{A}$ is approximated by a sequence of interval polynomials $\{Int(f)_j\}_j$. More specifically, let f be $\sum_{i_1,\ldots,i_m} a_{i_1 \ldots i_m} x_1^{i_1} \cdots x_m^{i_m}$. Then $Int(f)_j$ is defined to be

$$\sum_{i_1,\ldots,i_m} [(a_{i_1 \ldots i_m})_j, (\alpha_{i_1 \ldots i_m})_j]\, x_1^{i_1} \cdots x_m^{i_m},$$

stabilization works. See [34] for more detail.

[6]Or a list or an array of polynomials in $R[x_1, \ldots, x_n]$. For simplicity, we will treat a single polynomial in this section.

where $|a_{i_1\ldots i_m} - (a_{i_1\ldots i_m})_j| \leq (\alpha_{i_1\ldots i_m})_j$ for each j and $(\alpha_{i_1\ldots i_m})_j \to 0$ as $j \to \infty$ for all $i_1 \ldots i_m$. Let us call such $\{Int(f)_j\}_j$, an *approximation sequence* for f. In this case, we often write $Int(f)_j \to f$. For each j, we invoke $\mathcal{I}(\mathcal{A})$ on $Int(f)_j$ and obtain the output $\mathcal{I}(\mathcal{A})(Int(f)_j)$.

The following theorem states a property of $\mathcal{I}(\mathcal{A})$.

Theorem 1 (Coefficientwise convergence, [34]) *Suppose that $\mathcal{A}$ terminates normally when evaluated with input $f \in R[x_1, \ldots, x_n]$. Suppose that we have a sequence of polynomials with interval-coefficient $\{Int(f)_j\}_j$ where $Int(f)_j \to f$. Then, there exists n such that for $j \geq n$ $\mathcal{I}(\mathcal{A})$ terminates normally on input $Int(f)_j$, and we have:*

$$\mathcal{I}(\mathcal{A})(Int(f)_j) \to \mathcal{A}(f)$$

In many cases, however, coefficientwise convergence is not sufficient for the purpose of obtaining the correct "shape"[7] of an algebraic object. *Final zero rewriting* insures we will reach the correct shape for the output after a finite number of approximations. Namely, we do zero rewriting for each interval-coefficient of the output $\mathcal{I}(\mathcal{A})(Int(f)_j)$. We then take the first component of each interval-coefficient of the rewritten output to coerce it back to the original real coefficient polynomials. We refer to the algorithm – $\mathcal{I}(\mathcal{A})$ with final zero rewriting and coercing the output back to the R polynomials – as $\mathcal{I}(\mathcal{A})_R$.

The effect of the final zero rewriting is that it converts coefficientwise convergence into a stronger convergence called *supportwise convergence*. The *support* of a polynomial $f = \sum_{i_1,\ldots,i_m} a_{i_1\ldots i_m} x_1^{i_1} \cdots x_m^{i_m}$ is the set of power products $\{x_1^{i_1} \cdots x_m^{i_m} \mid a_{i_1\ldots i_m} \neq 0\}$, denoted $Supp(f)$. Similarly, for an interval polynomial F, $Supp(F)$ is the set of power products of F having interval-coefficient not equal to $[0, 0]$. *Supportwise convergence* is coefficientwise convergence with the added feature that coefficients which converge to zero must do so in a finite number of steps. Equivalently, $f_j \to f$ *supportwise* if $f_j \to f$ *coefficientwise* and there exists N such that for $j \geq N$: $Supp(f_j) = Supp(f)$. With *supportwise convergence* the "f_j's" take the shape of "f" after a finite number of steps.

Here is a stability property of $\mathcal{I}(\mathcal{A})_R$.

[7] Two polynomials have the same shape if they have the same support as defined in the next paragraph.

Theorem 2 (Supportwise convergence, [34]) *Under the hypotheses of Theorem 1, there exists n such that for $j \geq n$ $\mathcal{I}(\mathcal{A})_R$ terminates normally on input $Int(f)_j$ and we have:*

$$\mathcal{I}(\mathcal{A})_R(Int(f)_j) \rightarrow \mathcal{A}(f) supportwise$$

From now on we emphasize $\mathcal{I}(\mathcal{A})_R$ rather than $\mathcal{I}(\mathcal{A})$.

3 Applications

Table 1 presents several of the applications of the stabilization methodology. All of the examples are in terms of $\mathcal{I}(\mathcal{A})_R$. The left column in the table gives the algebraic objects computed by $\mathcal{I}(\mathcal{A})_R$ together with the algorithm used. The right column gives the related references. A floating-point approximation is used as an approximation sequence for input f. Namely, for $f = \sum_{i_1,\ldots,i_m} a_{i_1 \ldots i_m} x_1^{i_1} \cdots x_m^{i_m}$,

$$Int(f)_\mu = \sum_{i_1,\ldots,i_m} [fp_\mu(a_{i_1\ldots i_m}), er_\mu(a_{i_1\ldots i_m})] x_1^{i_1} \cdots x_m^{i_m},$$

where $fp_\mu(a)$ is a floating-point approximation to a with precision μ and $er_\mu(a)$ is the associated error. Note that not only is the input approximated but also in the course of executing each algorithm, approximate computation was done. This is also discussed in [34].

We have explained that zero rewriting is done preceding evaluating predicates. However, in implementation, it is often more convenient to perform zero rewriting after every function evaluation. This extra rewriting was done for the Buchberger and Sturm algorithms. This is good in the case where a predicate does not explicitly appear in the description of an algorithm. For example, the function of taking the lead term from a polynomial implicitly involves a predicate whether a coefficient is zero.

3.1 Gröbner bases

Gröbner basis (GB) techniques are a valuable tool for solving many problems in polynomial ideal theory. As is well known, the process of computing a GB may involve large numbers of intermediate coefficients - say from a

Table 1: Applications of the stabilization techniques

Gröbner bases (Buchberger)	(Shirayanagi 93) [31] (Ozaki 94) [21] (Hiyoshi 97) [10]
Counting real roots (Sturm)	(Sekigawa 95) [33]
Convex hull (Graham)	(Sekigawa 95) [29]
Smith normal form	(Niitsuma 96) [18]
Moore-Penrose generalized inverses (Greville)	(Minakuchi 96) [17]

field k - even when the final GB does not. In fact, the cost of performing exact arithmetic in k with the intermediate coefficients is a major factor determining the computational cost of computing a GB. When k is a subfield of the real numbers, using floating-point computation between coefficients may reduce the computational cost.

Buchberger's algorithm is an algorithm to compute Gröbner bases but it is unstable with respect to input coefficients because there is a predicate whether a remainder polynomial is zero or whether a coefficient is zero. We applied the stabilization techniques to it. One of our observation is: our approach may be unsuccessful if input polynomials contain a polynomial of too high degree or with too many terms, because even using floating-point computation does not solve the problem of the number of reductions or the number of polynomials that we must retain to make S-polynomials. But it will be useful when *the growth of coefficients* is the main reason why the computation of a Gröbner basis by the usual Buchberger algorithm is slow. For further details see [31, 32].

We refer the reader to [2, 5] for the theory of Gröbner bases and Buchberger's algorithm.

3.2 Counting real roots

Sturm's algorithm is an algorithm to count the number of all real roots of a given polynomial in one variable. Like Buchberger's algorithm, Sturm's algorithm also has the problem of intermediate expression swell in the size of the coefficients. There have been many efforts for improving the algorithm like the Sturm-Habicht sequence as found in [8]. But in the case where the coefficients are irrational, the above problem is still serious. What if we use floating-point computation without the methodology in [34]? Sturm's algorithm is then also unstable.

We applied the stabilization techniques to Sturm's algorithm based on floating-point computation. In many cases, the greatest advantage was found for polynomials having irrational coefficients and multiple real roots[8]. Namely, while a naive floating-point computation method could not give the correct number (the result varies or oscillates with the precision), our methodology stabilized and could give the correct number for a relatively low precision. Figure 1 illustrates this phenomenon for one polynomial of degree 53 having 26 terms and 3 real roots.

See [4] for example for the details of Sturm's algorithm.

3.3 Convex hull

Graham's algorithm is an algorithm to compute the two dimensional convex hull of a given finite set of points. The instability of the algorithm derives from the predicate whether a point is to the left or to the right to or directly on a line. This predicate has zero discontinuity. We applied the stabilization techniques to the algorithm and experimented on an input sample of 3,000 points, with success. In this application, in some cases one can verify the correctness *visually.*

[8]One might preprocess an input polynomial f to obtain a square free polynomial. The usual way to do this is to divide f by $\gcd(f, f')$, using the Euclidean algorithm to compute the GCD. However, the Euclidean algorithm is unstable.

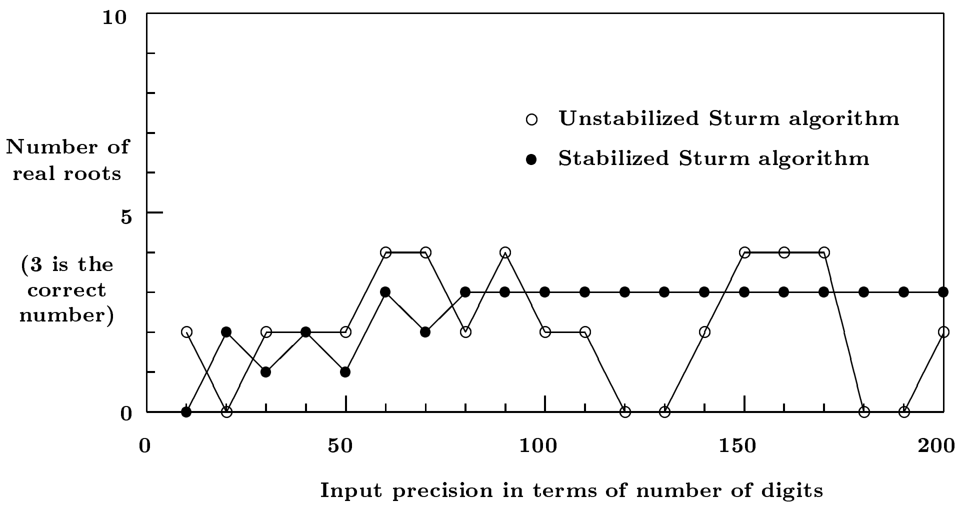

Figure 1: Effect of stabilizing Sturm's algorithm

However, as far as the two dimensional convex hull problem is concerned, in most cases a naive floating-point computation method seems to be sufficient as well.

The stabilization techniques would be more effective for higher dimensional cases or other problems in computational geometry.

We refer the reader to [22] for example for the details of Graham's algorithm and other topics in computational geometry.

3.4 Smith normal form

Computing the Smith normal form or Jordan canonical form of a given matrix is also unstable, especially when the matrix has multiple eigenvalues, see [15]. In that case the stabilization techniques were useful. For example, for the 4×4 matrix

$$A = \begin{bmatrix} \frac{107}{153} & \frac{202}{153} & \frac{8}{153} & -\frac{61}{153} \\ -\frac{29}{153} & \frac{421}{306} & \frac{25}{153} & -\frac{37}{306} \\ -\frac{71}{306} & \frac{577}{612} & \frac{325}{306} & -\frac{175}{612} \\ -\frac{22}{51} & \frac{19}{51} & \frac{26}{51} & \frac{44}{51} \end{bmatrix}$$

we want to compute the Smith normal form of the polynomial matrix $A(x) = xE - A$, where E is the identity matrix. A classical algorithm, [6], – which might be described as the marriage of row and column reduction with the Euclidean algorithm – is used to compute the Smith normal form.[9] Let us call it *smith*. The instability of *smith* derives from that of the Euclidean algorithm which has zero discontinuity.

Now we have

$$smith(A(x)) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (x-1)^2 & 0 \\ 0 & 0 & 0 & (x-1)^2 \end{bmatrix}$$

Consider a floating-point approximation $\tilde{A}$ of A to precision 5:

$$\tilde{A} = \begin{bmatrix} 0.69935 & 1.3203 & 0.052288 & -0.39869 \\ -0.18954 & 1.3758 & 0.16340 & -0.12092 \\ -0.23203 & 0.94281 & 1.0621 & -0.28595 \\ -0.43137 & 0.37255 & 0.50980 & 0.86275 \end{bmatrix}$$

[9] There are more efficient and practical algorithms to compute Smith normal forms, such as [13, 12, 39], etc. One could make further experiments on those algorithms to show the usefulness of the stabilization techniques. The article [23] discusses the problem of stability of the classical algorithm and also presents a method for computing Smith normal forms using limited precision p-adic arithmetic. This is closely related to our approach.

Merely applying *smith* to $\tilde{A}(x)$ does not yield the correct form:

$$smith(\tilde{A}(x)) = \begin{bmatrix} 1.0000 & 0 & 0 & 0 \\ 0 & 1.0000 & 0 & 0 \\ 0 & 0 & 1.0000 & 0 \\ 0 & 0 & 0 & p(x) \end{bmatrix}$$

where $p(x) = 1.0000x^4 - 3.9598x^3 + 5.9237x^2 - 3.9677x + 1.0040$. Making the precision higher until $\mu = 1000$ does not succeed. What about $\mathcal{I}(smith)_R$? As a convergent sequence $\{Int(A(x))_\mu\}_\mu$ of interval polynomial matrices for $A(x)$, we consider "$[fp_\mu(A(x)), er_\mu(A(x))]$", each coefficient of whose entry is an interval based on the floating-point approximation of the corresponding coefficient of $A(x)$ to precision μ and its error bound. Then,

$\mathcal{I}(smith)_R(Int(A(x))_2) =$

$$\begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0x^2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The precision of 2 is too small. But

$\mathcal{I}(smith)_R(Int(A(x))_3) =$

$$\begin{bmatrix} 1.00 & 0 & 0 & 0 \\ 0 & 1.00 & 0 & 0 \\ 0 & 0 & 1.00x^2 - 2.00x + 1.01 & 0 \\ 0 & 0 & 0 & 1.00x^2 - 2.00x + 0.989 \end{bmatrix}$$

And for $\mu = 4, \ldots, 10$ we observed supportwise convergence of $\{\mathcal{I}(smith)_R(Int(A(x))_\mu)\}_\mu$ to $smith(A(x))$.

3.5 Moore-Penrose generalized inverses

Given an $m \times n$ matrix A with entries in the reals, an $n \times m$ matrix G is said to be the Moore-Penrose generalized inverse of A, or for short MPG-inverse of A, if G satisfies the following four conditions:

(i) $AGA = A$

(ii) $GAG = G$

(iii) $(AG)^T = AG$

(iv) $(GA)^T = GA$

where B^T denotes the transpose of B. G is uniquely determined by A and often written as A^+.

Moore-Penrose generalized inverses are an important tool for solving a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$. When there is no strict solution, it is possible to find the *least square solution* by $\boldsymbol{x} = A^+\boldsymbol{b}$, where $\|A\boldsymbol{x} - \boldsymbol{b}\|$ is minimal with $\|\boldsymbol{x}\|$ minimal. The norm $\|\cdot\|$ is the Euclidean norm, i.e. the square root of the sum of squares of each component. For the details see [24]. Moreover, MPG-inverses are applied to engineering such as associative memory and pattern recognition, see [38].

There are many algorithms to compute Moore-Penrose generalized inverses, but we used Greville's algorithm, see [9]. Let us call it **Greville**.

Greville: Let us express an $m \times n$ input matrix A as $[a_1^T, a_2^T, \cdots, a_n^T]^T$, where each a_i is an m-dimensional vector. Define an $m \times i$ matrix A_i by

$$A_1 = a_1 \qquad A_i = \begin{bmatrix} A_{i-1} \\ a_i \end{bmatrix}$$

and for $i = 2, \ldots, n$, we compute an $i \times m$ matrix A_i^+ inductively by the following equation:

$$A_i^+ = [A_{i-1}^+ - b_i^T d_i \quad b_i^T]$$

where d_i, c_i, b_i $(2 \leq i \leq n)$ and the initial matrix A_1^+ are computed by the following:

$$\begin{aligned} d_i &= a_i A_{i-1}^+ \\ c_i &= a_i - d_i A_{i-1} \\ b_i &= \begin{cases} \frac{c_i}{c_i c_i^T} & \text{if } c_i \neq 0 \\ \frac{d_i (A_{i-1}^+)^T}{1 + d_i d_i^T} & \text{if } c_i = 0 \end{cases} \\ A_1^+ &= \begin{cases} \frac{a_1^T}{a_1 a_1^T} & \text{if } a_1 \neq 0 \\ a_1^T & \text{if } a_1 = 0 \end{cases} \end{aligned}$$

A_n^+ is the desired matrix A^+.

As can be seen, **Greville** is an algebraic algorithm with zero discontinuity, which has the predicates "$c_i = 0$" ($2 \leq i \leq n$) and "$a_1 = 0$".

This application exhibits an interesting, important property: it is computationally cheaply checkable. I.e. given a proposed MPG-inverse to A, obtained by stabilization methods or any other way, it is computationally inexpensive to determine whether the four conditions (i) - (iv) are satisfied, either strictly or approximately. The extended Euclidean algorithm is another such example. We will propose a new direction for application of the stabilization methodology, where again the example of MPG-inverses is considered.

In this experiment, we observed that if G_μ is the result obtained by executing **Greville** based on the stabilization techniques with precision μ, then $\|AG_\mu A - A\| \to 0$, $\|G_\mu AG_\mu - G_\mu\| \to 0$, $\|(AG_\mu)^T - AG_\mu\| \to 0$, and $\|(G_\mu A)^T - G_\mu A\| \to 0$ as μ increases, where $\|\cdot\|$ denotes the square root of the sum of square of each entry of the matrix. Of course, when we were able to compute A^+ by exact approach in advance, we observed $G_\mu \to A^+$.

4 Frequently asked questions

We list some of the questions which have been frequently asked about the stabilization methodology. We also address some of the misunderstood which we have encountered.

- **Q.** The stabilization methodology guarantees that the result will converge to the true result as the precision increases. Does this happen in a manner similar to Newton's method?
  A. Presumably the question is whether applying the stabilization methodology to an algorithm $\mathcal{A}$ yields an algorithm $\mathcal{B}$ where $\mathcal{B}$ is similar to Newton's method. In general this is not the case. Newton's method is a *successive approximation* method. One inputs an approximate solution and, when all goes well, Newton's method outputs a better approximate solution and the successive approximations converge to an exact solution. Using Newton's method one obtains a convergent sequence of results by using the output from one pass as the input to the next pass. In general $\mathcal{B}$ is not a successive approximation method and it would

not be possible to use the output from one iteration of $\mathcal{B}$ as the input for the next iteration.[10] By contrast an oversimplified description of $\mathcal{B}$ goes as follows:

- When $\mathcal{A}$ is executed with (exact) input and exact computation a particular "execution path" through the algorithm occurs, based upon the specific program branching resulting from predicate evaluation. Let us call this execution path the *exact execution path* resulting from the input.
- $\mathcal{B}$ together with some (exact) input does the following:
  * The *exact execution path* resulting from the input is ascertained.
  * The *exact execution path* is executed with increasingly accurate approximations to the exact input.
  * Increasingly accurate "arithmetic operations" are used in the passes with the increasingly accurate approximations to the exact input.

Of course when we say "increasingly accurate" we mean "increasingly accurate and convergent to the exact value."

- **Q.** Zero rewriting changes an interval $[E, \epsilon]$ to $[0, 0]$ if $|E| \leq \epsilon$. Is ϵ given in advance, or before running an algorithm?
  A. No. At an "internal" point in an algorithm one might be adding $[A, \alpha]$ and $[B, \beta]$. The result would be $[A+B, \alpha+\beta]$. Suppose that the next step is to apply a predicate to $[A+B, \alpha+\beta]$. Then zero rewriting is performed on $[A+B, \alpha+\beta]$ before the predicate is applied. Thus we are in the situation where $[E, \epsilon] = [A+B, \alpha+\beta]$ and we see that ϵ, the error bound of $[E, \epsilon]$, differs in general from the error bound of other terms.

  In case $[E, \epsilon]$ is an input and not the result of previous computation then E is an approximation to some exact input value and ϵ is the error

[10] Associated to Newton-like methods using interval computation, recently the area of self-validating numerics or numerical methods with guaranteed accuracy has been active to solve many nonlinear problems. This research was stimulated by Krawczyk's interval mapping [16] and Moore's interval iterative mapping [19], etc.

bound. If there are other input approximations it is not required that they all have the same error bounds.

When $[E, \epsilon]$ is an input, the size of ϵ depends upon at which stage we are in the approximation. If ϵ_i is the ϵ at the i^{th} stage then $\epsilon_i \to 0$.

- **Q.** What is the effect of zero rewriting? Is it really necessary?
  A. Yes. Let us suppose that when executed with a particular input and exact computation the algebraic algorithm $\mathcal{A}$ outputs the real number E. Then the algorithm $\mathcal{I}(\mathcal{A})$ when executed on approximation sequences for the input to $\mathcal{A}$ has output $\{[E_\mu, \epsilon_\mu]\}_\mu$ where $\{[E_\mu, \epsilon_\mu]\}_\mu$ is an approximation sequence for E, i.e. $|E - E_\mu| \leq \epsilon_\mu$ and $\epsilon_\mu \to 0$.[11] Consider the two cases, whether $E = 0$ or not. If $E = 0$ then $|E - E_\mu| \leq \epsilon_\mu$ implies that $|E_\mu| \leq \epsilon_\mu$ and so $[E_\mu, \epsilon_\mu]$ is rewritten to $[0, 0]$ and E is the first component of the rewritten $[E_\mu, \epsilon_\mu]$. Next suppose that E does not equal zero. In this case $\epsilon_\mu \to 0$ implies that there is an N where $\epsilon_\mu < |E|/2$ for $\mu > N$. Then $|E - E_\mu| \leq \epsilon_\mu$ implies that $\epsilon_\mu < |E_\mu|$ for $\mu > N$. Hence $[E_\mu, \epsilon_\mu]$ does not get changed by rewriting for $\mu > N$. Thus for $\mu > N$ the first component of the rewritten $[E_\mu, \epsilon_\mu]$ is E_μ and not 0 and hence converges to E.

  Now suppose that instead of E being the output of $\mathcal{A}$, E is an internal quantity computed by $\mathcal{A}$ in the course of execution of $\mathcal{A}$ and a predicate of A which has zero discontinuity gets evaluated at E. The same reasoning as above shows that the first component of the rewritten $[E_\mu, \epsilon_\mu]$ is 0 from some point onward if $E = 0$ and otherwise the first component is part of a sequence converging to E. Thus if the predicate does in fact have a discontinuity at 0 there will not be a problem because the (corresponding) predicate for $\mathcal{I}(\mathcal{A})$ will be evaluated at 0 and not on elements of a sequence converging to 0.

- **Q.** Suppose we wish to compute $x + z - x$, where z is some complicated but ultimately small (but not zero) quantity, and x is large (or has large coefficients). Then there is a good chance that zero rewriting will throw away the z term; giving the final answer of zero. Evaluating the resulting symbolic expression in exact arithmetic (namely $x - x$) will also give zero. The author's algorithm will conclude that all is well,

[11] Note that $\{[E_\mu, \epsilon_\mu]\}_\mu$ does not necessarily start with $\mu = 1$.

but the answer will be wrong. What is wrong with this analysis of the stabilization methodology?

A. Suppose at the i^{th} stage the interval coefficient corresponding to x is $[x_i, \alpha_i]$ and the interval coefficient corresponding to z is $[z_i, \beta_i]$. Then the interval coefficient for $x+z-x$ will be: $[z_i, 2\alpha_i+\beta_i]$. The questioner seems to think that α_i will be large because x is large but in fact α_i is a bound for $|x - x_i|$ and $\alpha_i \to 0$ as $i \to \infty$. Also, $\beta_i \to 0$ as $i \to \infty$. It may happen for low values of i that $|z_i| \leq 2\alpha_i + \beta_i$ in which case $[z_i, 2\alpha_i + \beta_i]$ will be rewritten to [0,0] but $\alpha_i \to 0$ and $\beta_i \to 0$ imply that for large values of i: $2\alpha_i + \beta_i < |z_i|$ since z is not zero. At this stage $[z_i, 2\alpha_i + \beta_i]$ is not rewritten to $[0, 0]$.

5 New directions

5.1 Hardness of the precision problem

We have described the stability methodology for various algorithms. At the present stage of development, bounds are lacking for the point where the execution path becomes the same as that made by an exact approach or where the correct support appears. This is an important issue. The following simple example illustrates one of the difficulties in obtaining such a bound and in particular how a bound can depend upon the input.

Consider the algorithm, $\mathcal{A}$:

Input	A, B
If	$A - B = 0$ then 1 else 0.

Suppose that the inputs A and B are positive real numbers. By the stabilization methodology, $\mathcal{I}(\mathcal{A})_R$ (or its equivalent) is:

Input	$[a, \alpha]$, $[b, \beta]$
If	$\|a - b\| \leq \alpha + \beta$ then 1 else 0.

where we are neglecting roundoff errors from performing interval arithmetic.

For input intervals, let a_μ be a floating-point approximation to A with precision μ and α_μ be an associated error, say $5 \times 10^{e(A)-\mu-1}$, where $e(A)$ is the exponent of A in base 10, namely $\lfloor \log_{10} A \rfloor$, where $\lfloor u \rfloor$ is the largest integer less than or equal to u. Similarly define $[b_\mu, \beta_\mu]$ for B.

Say $A = 0.11$ and $B = 0.10$. Obviously $\mathcal{A}(A, B) = 0$. What is

$$\mathcal{I}(\mathcal{A})_R([a_\mu, \alpha_\mu], [b_\mu, \beta_\mu])$$

For $\mu = 1$,

$$\begin{array}{rcl} [a_\mu, \alpha_\mu] & = & [0.1, 5 \times 10^{-2}] \\ a_\mu - b_\mu & = & 0 \end{array} \qquad \begin{array}{rcl} [b_\mu, \beta_\mu] & = & [0.1, 5 \times 10^{-2}] \\ \alpha_\mu + \beta_\mu & = & 10^{-1} \end{array}$$

Hence, for $\mu = 1$:

$$a_\mu - b_\mu < \alpha_\mu + \beta_\mu$$

and so $\mathcal{I}(\mathcal{A})_R([a_\mu, \alpha_\mu], [b_\mu, \beta_\mu]) = 1$, the wrong answer.

For $\mu \geq 2$,

$$\begin{array}{rcl} [a_\mu, \alpha_\mu] & = & [0.11, 5 \times 10^{-(\mu+1)}] \\ a_\mu - b_\mu & = & 0.01 \end{array} \qquad \begin{array}{rcl} [b_\mu, \beta_\mu] & = & [0.10, 5 \times 10^{-(\mu+1)}] \\ \alpha_\mu + \beta_\mu & = & 10^{-\mu} \end{array}$$

Hence, for $\mu = 2$:

$$a_\mu - b_\mu = \alpha_\mu + \beta_\mu$$

and so $\mathcal{I}(\mathcal{A})_R([a_\mu, \alpha_\mu], [b_\mu, \beta_\mu]) = 1$, still wrong.

For $\mu \geq 3$:

$$a_\mu - b_\mu > \alpha_\mu + \beta_\mu$$

and so $\mathcal{I}(\mathcal{A})_R([a_\mu, \alpha_\mu], [b_\mu, \beta_\mu]) = 0$, the right answer.

Thus $\mathcal{I}(\mathcal{A})_R([a_\mu, \alpha_\mu], [b_\mu, \beta_\mu])$ outputs the correct answer when $\mu \geq 3$ but not before. If $A_n = 0.1 + 10^{-n}$ and $B_n = 0.1$ the same reasoning shows that

$$\mathcal{I}(\mathcal{A})_R([a_{n,\mu}, \alpha_{n,\mu}], [b_{n,\mu}, \beta_{n,\mu}])$$

outputs the same answer as $\mathcal{A}(A_n, B_n)$ when $\mu \geq n + 1$ but not before. In general for $0 < A < 1$, $0 < B < 1$ and $A \neq B$ the worst case behaviour is that $\mathcal{I}(\mathcal{A})_R$ requires precision $\mu = \lfloor | \log_{10} |A - B|| \rfloor + 1$ to be certain of a correct answer. In general, it may be difficult to compute $A - B$ or its exponent exactly and if we knew the exact value of $A - B$, it would not have been necessary to use the stabilization methodology.

We do not mean to indicate that it is a wrong direction or unimportant to pursue the precision problem.[12] For some algorithms, it may be very

[12] The problem was a motivation for Sekigawa's work [30] for determining whether a given algebraic number is zero.

important to perform stochastic analysis on a large number of experiments regarding precisions, considering the worst case, the average case, and the best case. Is the stabilization methodology useless as long as the precision problem is unsolved? Our answer is NO, and in the next two sections we propose new directions in which one may make the best of the stabilization methodology.

5.2 Approximate solutions

Suppose that $I = (r_1, \ldots, r_n)$ is a vector in R^n where R is a subfield of the real numbers and suppose that $\mathcal{A}$ is an algorithm which given such I as input is supposed to give a length one vector $O = (s_1, \ldots, s_n)$ which is orthogonal to I as output. It is easy to check if the output of $\mathcal{A}$ is correct, check if substituting $(s_1, \ldots, s_n)$ for $(x_1, \ldots, x_n)$ satisfies the two equations:

$$\begin{array}{ccccccc} {x_1}^2 & + & \ldots & + & {x_n}^2 & = & 1 \\ r_1 x_1 & + & \ldots & + & r_n x_n & = & 0 \end{array}$$

Let $\mathcal{F}(I)$ denote the set consisting of the two functions:

$$\begin{array}{ccccccc} {x_1}^2 & + & \ldots & + & {x_n}^2 & - & 1 \\ r_1 x_1 & + & \ldots & + & r_n x_n & & \end{array}$$

The first of these two functions does not depend upon I but the second does and is an example of a function parameterized by I. O is a suitable output for $\mathcal{A}$ run with input I if and only if the functions in $\mathcal{F}(I)$ vanish when evaluated on (the coefficients of) O.

Let us consider a richer example. Given an $m \times n$ matrix A, with real number entries, it is relatively easy to check if an $n \times m$ matrix G which is purportedly a Moore-Penrose generalized inverse to A actually is a Moore-Penrose generalized inverse. One merely checks whether G satisfies the matrix equations[13]:

(i) $AXA - A = 0$

(ii) $XAX - X = 0$

(iii) $(AX)^T - AX = 0$

[13] See (i)–(iv) at the beginning of section 3.5.

(iv) $(XA)^T - XA = 0$

If the *matrix variable* X is expressed as an $n \times m$ matrix with entries $\{x_{i,j}\}_{i,j}$, which are variables, the preceding matrix equations translate into a number of equations which the coefficients of G must satisfy[14]. All of the equations are of the form: *a function of* ((*the coefficients of*) G) $= 0$. Drop off the "$= 0$" for each equation leaving a family $\mathcal{F}$ of polynomial functions.

Thus for the Moore-Penrose generalized inverse the question of whether G is an MPG inverse to A is equivalent to whether the functions in $\mathcal{F}$ vanish when evaluated on (the coefficients of) G. The functions in $\mathcal{F}$ are parameterized by A in the sense that expressions based upon the coefficients of A are used to form the coefficients of the functions in $\mathcal{F}$. Let us use $\mathcal{F}(A)$ to stand for the specific family of functions $\mathcal{F}$ arising from the matrix A.

Suppose that for input I we have a family of functions $\mathcal{F}(I)$ parameterized by I. We say that O is a *zero* of $\mathcal{F}(I)$ or $\mathcal{F}(I)$ vanishes on O if $E(O) = 0$ for each function $E \in \mathcal{F}(I)$. Beyond orthogonality of vectors and Moore-Penrose Generalized inverses, for many algebraic problems which depend upon an initial condition I, the issue of whether O solves the problem can be reduced to whether a family of functions parameterized by I vanishes on O. In such cases whether an algorithm $\mathcal{A}$ with input I provides a solution to the original problem is equivalent to whether the algorithm, with input I, gives an output which is a *zero* of the associated family of functions $\mathcal{F}(I)$. In fact we may shift perspective and think of the algorithm as (primarily) providing *zeros* of the parameterized families of functions $\{\mathcal{F}(I)\}_I$. I.e. given an input I the algorithm $\mathcal{A}$ computes a zero of $\mathcal{F}(I)$. This shift in perspective enables us to speak of *approximate solutions* to the extent that we may consider an *approximate zero* of the functions in $\mathcal{F}(I)$ to be an approximate solution to the original problem. O' being an *approximate zero* to the functions in $\mathcal{F}(I)$ is expressed by a distance or a valuation or a norm, a tolerance ϵ and the requirement that

$$\|E(O')\| < \epsilon \text{ for each function } E \in \mathcal{F}(I).$$

When this holds let us say that O' is an "ϵ *approximate solution*" of $\mathcal{F}(I)$. Note that for O' to be an ϵ *approximate solution* of $\mathcal{F}(I)$ there is no requirement that O' be near a *zero* of $\mathcal{F}(I)$.

[14]The equations on the coefficients of G which arise from (i), (iii) and (iv) will be linear but the equations on the coefficients of G which arise from (ii) will be second degree.

The first example in this section considered unit vectors orthogonal to I. If I were a unit vector or the length of I is taken into account in choosing ϵ one may control that an approximate solution lies within a desired angle of perpendicular.

The rest of this section is about how the stabilization methodology may be applied to an algorithm to obtain approximate solutions. Throughout the rest of the section we assume that:

i. $\mathcal{A}$ is an algebraic algorithm with zero discontinuity.

ii. $\mathcal{F}(I)$ is a parameterized family of continuous functions for inputs I of $\mathcal{A}$.

iii. If $\mathcal{A}$ is executed with input I, the output is a zero of $\mathcal{F}(I)$.

In the *orthogonal vector* example and Moore-Penrose example, for each I the functions in $\mathcal{F}(I)$ are of polynomials of degree two or less. The main property that we shall use about the functions in $\mathcal{F}(I)$ is that they are continuous functions.

The first step in obtaining an approximate solution is to prepare an approximation sequence $\{[I_n, \iota_n]\}_n$ for I. Then set n to an initial value, say $n = 1$. Invoke $\mathcal{I}(\mathcal{A})_R$ on $[I_n, \iota_n]$. Say the output is O_n. If one verifies that:

$$\|E(O_n)\| < \epsilon \text{ for each function } E \in \mathcal{F}(I)$$

then return O_n as the result. Otherwise increase n and repeat.

From the stability theorem (Theorem 2), it follows that $\{O_n\}_n$ forms an approximation sequence for O, that is, $O_n \to O$. Since the functions in $\mathcal{F}(I)$ are assumed to be continuous, it follows that for $E \in \mathcal{F}(I)$: $E(O_n) \to E(O) = 0$. Thus for large enough n, $\|E(O_n)\| < \epsilon$ for each function $E \in \mathcal{F}(I)$.

The significant issue here is the computational difficulty or expense of determining:

$$\|E(O_n)\| < \epsilon \text{ for each function } E \in \mathcal{F}(I).$$

5.3 Obtaining exact output

This section is about using the stabilization methodology to obtain an exact solution. The obvious difficulty is that $\mathcal{I}(\mathcal{A})_R$ produces approximation sequences which converge to the exact output of $\mathcal{A}$ but do not reach the exact output of $\mathcal{A}$ in a finite number of steps. On the other hand, two types of finiteness occur. The first is supportwise convergence (Theorem 2), whereby coefficients which will converge to zero do so in a finite number of steps[15]. The second type of finiteness concerns the execution path. As $\mathcal{I}(\mathcal{A})_R$ is executed on successive elements $[I_n, \iota_n]$ of an approximation sequence $\{[I_n, \iota_n]\}_n$ for I there comes an N where for $n \geq N$ the execution path through $\mathcal{I}(\mathcal{A})_R$ is the same as the execution path of $\mathcal{A}$ executed with input I, [34]. This execution path is what we have called the *exact execution path* in section 4. As will be seen this leads to an effective technique of obtaining exact output.

Supportwise convergence might be harnessed in order to obtain an exact solution. One would have to use the "shape" of elements in the approximation sequence outputs of $\mathcal{I}(\mathcal{A})_R$ to obtain an exact solution. At present, we do not have a useful example of this tactic nor a useful example along these lines for the more general "S rewriting", "algorithms with S discontinuity" and "S convergence" mentioned in the previous paragraph.

The fact that the execution path through $\mathcal{I}(\mathcal{A})_R$ stabilizes at the exact execution path may be utilized by keeping track of or *logging* the execution path. This can be found in [35]. The *win* of the log method is that predicates are not evaluated and one only has to perform evaluation with exact coefficients for computations leading to the *final* output. One can dispense with evaluation of *intermediate* data which does not enter into the final output. When the intermediate computation utilizes much memory and the the final result is of moderate size, this approach is useful. The computation of Gröbner bases is such an example, see [35].

Suppose that given a finite set $F \subset R[x_1, \ldots, x_n]$, we want to compute a purely lexicographic Gröbner basis of $\langle F \rangle$. In this case, verification whether a candidate G' is really the desired Gröbner basis is based on two matters: one is testing whether G' satisfies the property (definition) of Gröbner bases, and the other is testing equality of $\langle F \rangle = \langle G' \rangle$. We ran experiments. In one computation of a Gröbner basis in the case where the coefficients are

[15]A much better result holds. It is shown in [34]] that one may specify a discrete set S and do rewriting with regard to S. This has an additional side benefit of permitting algorithms where the predicates have discontinuities at elements of S, not just at zero. The result is that output coefficients which will converge to elements of S do so in a finite number of steps. We have restricted consideration to "zero rewriting", "algorithms with zero discontinuity" and "support convergence" in order to simplify the discussion. However, this is the special case where: S = $\{0\}$.

irrational, the approach computed the exact Gröbner basis (in about 100 seconds) whereas a direct exact computation method crashed after the intermediate expressions became too large (with over 2^{16} syntactic elements).

A second experiment involved polynomial division in $R[x_1, \ldots, x_n]$. Given a polynomial f and a list $F = (f_1, \ldots, f_s)$ of polynomials, we want to divide f modulo F with respect to a fixed term order, and compute the list of quotients $Q = (q_1, \ldots, q_s)$ and the remainder r where $f = q_1 \cdot f_1 + \cdots + q_s \cdot f_s + r$. As such an algorithm we refer the reader to the division algorithm in $R[x_1, \ldots, x_n]$ as found in [5]. Polynomial division occurs many times in computational algebra, such as the Gröbner basis computation and the ideal membership test. In this case, verifying whether candidates Q' and r' are the correct answer is simply based on testing the equality $f = q'_1 \cdot f_1 + \cdots + q'_s \cdot f_s + r'$. In most cases where the coefficients are complicated such as irrational or transcendental numbers, it was observed that a direct exact computation method took 2 to 3 magnitudes of computer resources and time as was required using the stabilization methodology.

6 Conclusion

For an algebraic algorithm $\mathcal{A}$ with zero discontinuity, not only does approximate computation with zero rewriting and interval arithmetic lead to a sequence of outputs which converge to the exact output, after a finite number of steps the "shape" of the exact output and the exact execution path of $\mathcal{A}$ are determined. This is directly effective for applications which can utilize approximate data which converges to the exact data. For some algorithms where the exact output satisfies a family of equations, the level of approximation may be measured by how well the equations are satisfied. This may be used as feedback to require a higher level of approximation. The stabilization methodology may also be used to help obtain exact output by utilizing knowledge of the exact execution path.

References

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

[2] Buchberger, B.: Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory, *Multidimensional Systems Theory (Ed. N. K. Bose), D. Reidel Publishing Company* (1985), 184–232.

[3] Corless, R. M., Gianni, P. M., Trager, B. M., and Watt, S. M.: The Singular Value Decomposition for Polynomial Systems, *Proc. of ISSAC'95* (1995), 195–207.

[4] Collins, G. E. and Loos, R.: Real zeros of polynomials, *Computer Algebra, Symbolic and Algebraic Computation, (Ed. B. Buchberger, G. E. Collins, and R. Loos), Springer-Verlag* (1982), 83–94.

[5] Cox, D., Little, J., and O'Shea D.: *Ideals, Varieties, and Algorithms, Springer-Verlag* (1992).

[6] Dummit, D. and Foote, R.: *Abstract Algebra, Prentice Hall* (1991).

[7] Goldberg, D.: What every computer scientist should know about floating-point arithmetic, *ACM Computing Surveys*, **23** 1 (1991), 5–48.

[8] Gonzàlez L., Lombardi H., Recio T., and Roy M.-F.: Sturm-Habicht sequence, *Proc. of ISSAC'89* (1989), 136–146.

[9] Greville, T. N. E.: Some Applications of the Pseudoinverse of a Matrix, *SIAM Review*, **2** 1 (1960), 15–22.

[10] Hiyoshi, H.: Applications of approximate computation to computational algebra and computational geometry (in Japanese), *Master thesis, Univ. Tokyo* (1997).

[11] Hribernig, V.: Sensitivity of Algebraic Algorithms, *Ph.D. thesis, Tech. Univ. Vienna* (1995).

[12] Kaltofen, E., Krishnamoorthy, M. S. and Saunders, B. D.: Fast parallel computation of Hermite and Smith forms of polynomial matrices, *SIAM Journal of Algebraic and Discrete Methods*, **8** (1987), 683–690.

[13] Kannan, R.: Polynomial-time algorithms for solving systems of linear equations over polynomials, *Theoretical Computer Science*, **39** (1985), 69–88.

[14] Karmarkar, N. and Lakshman, Y. N.: Approximate Polynomial Greatest Common Divisors and Nearest Singular Polynomials, *Proc. of ISSAC'96* (1996), 35–39.

[15] Kato, T.: *Perturbation Theory for Linear Operators, Springer-Verlag 2nd ed.* (1976).

[16] Krawczyk, R.: Newton-algorithmen zur bestimmung von nullstellen mit fehlerschranken, *Computing*, **4** (1969), 187–201.

[17] Minakuchi: private communication (1996).

[18] Niitsuma, H. and Shirayanagi, K.: Stabilizing floating-point Jordan normal forms (in Japanese), to appear in *RIMS Kokyuroku*.

[19] Moore, R. E.: A Test for Existence of Solutions to Nonlinear Systems, *SIAM J. Numer. Anal.*, **14** (1977), 611–615.

[20] Noda, M-T. and Sasaki, T.: Approximate GCD and Its Application to Ill-conditioned Algebraic Equations, *J. Computational and Applied Mathematics*, **38** (1991), 335–351.

[21] Ozaki, H. and Shirayanagi, K.: Using Maple's Interval Package to Compute Floating-point Gröbner Bases (in Japanese), *RIMS Kokyuroku* **920** (1995), 38–52.

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

[23] Ramachandran, V.: Exact reduction of a polynomial matrix to the Smith normal form, *IEEE Transactions on Automatic Control*, AC-24 (4), 638–641, (1979).

[24] Rao C. R. and Mitra S. K.: *Generalized Inverses of Matrices and its Applications, John Wiley & Sons* (1971).

[25] Sasaki, T.: Approximate Algebraic Computation (in Japanese), *RIMS Kokyuroku* **676** (1988), 307–319.

[26] Sasaki, T. and Noda, M-T.: Approximate Square-free Decomposition and Root-finding of Ill-conditioned Algebraic Equations, *J. Information Processing*, **12** (1989), 159–168.

[27] Schönhage, A.: Quasi-GCD computations, *J. Complexity*, **1** (1985), 118–137.

[28] Sederberg, T. W. and Chang, G-Z.: Best linear common divisors for approximate degree reduction, *Computer-aided Design*, **25** 3 (1993), 163–168.

[29] Sekigawa, H. and Shirayanagi, K.: Stabilizing an Algorithm to Compute Convex Hulls (in Japanese), *the 4-th annual meeting of JSSAC* (1995).

[30] Sekigawa, H.: Using Interval Arithmetic and Polynomial Norms to Determine Signs of Algebraic Numbers, *Proc. of the 1996 Asian Symposium on Computer Mathematics* (1996), 43–53.

[31] Shirayanagi, K.: An Algorithm to Compute Floating Point Gröbner Bases, *Mathematical Computation with Maple V: Ideas and Applications (Ed. T. Lee), Birkhäuser* (1993), 95–106.

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

[33] Shirayanagi, K. and Sekigawa, H.: An Interval Method Based on Zero Rewriting and its Application to Sturm's Algorithm (in Japanese), to appear in *IEICE Trans. A*.

[34] Shirayanagi, K. and Sweedler, M.: A Theory of Stabilizing Algebraic Algorithms, *Technical Report 95-28, Mathematical Sciences Institute, Cornell University* (1995), 92 pages.

[35] Shirayanagi, K. and Sweedler, M.: Automatic Algorithm Stabilization, *ISSAC'96 poster session abstracts* (1996), 75–78.

[36] Stetter, H. J.: Verification in Computer Algebra Systems, Validation Numerics, *Computing Suppl.* **9** (1993), 247–263.

[37] Stoer, J. and Bulirsch, R.: *Introduction to Numerical Analysis, Texts in Applied Mathematics* **12**, *Springer-Verlag* (1993).

[38] Therrien C. W.: Eigenvalue Properties of Projection Operators and Their Application to the Subspace Method of Feature Extraction, *IEEE Trans. Comput.* C-24 (1975), 944–948.

[39] Villard, G.: Computation of the Smith normal form of polynomial matrices. *Proc. of ISSAC'93* (1993), 208–217.