Proceedings of the Risa Consortium
1997 pp. 113– pp. 129

Automatic Algorithm Stabilization for Smith Normal Form of Polynomial Matrices*

Kiyoshi Shirayanagi †

NTT Communication Science Laboratories

(Received 1997/5/30 Revised 1997/9/4)

Abstract. Starting with a matrix A with real entries, the characteristic matrix of A is $A-xE$, the matrix whose determinant is the characteristic polynomial of A. The Smith normal form of $A-xE$ is a diagonal matrix with real polynomial entries and of fundamental use in linear algebra, [5]. Smith normal form is obtained by combining row and column reduction with the Euclidean algorithm. Such algorithms are not typically stable with respect to limited precision computation. I.e. if the algorithm is executed with limited precision computation, the output does not necessarily approach the Smith normal form as the precision increases. [10] presents a technique of stabilizing algorithms to allow limited precision computation in a *convergent* manner. In this paper we present an algorithm – which is amenable to the techniques of [10] – for computing Smith normal form. We then apply [10] to obtain an algorithm which when executed with limited precision will converge to Smith normal form as the precision increases. The convergence also has the property of being "fast convergence" at zero. I.e. any coefficient which will converge to zero reaches zero in a finite number of steps.

1. Introduction

Normal forms of matrices – such as rational canonical form and Jordan canonical form – play a fundamental role in linear algebra and its applications. They may be found in almost any book on linear algebra. Smith normal form is a normal form of matrices

* This research has been carried out in collaboration with Moss Sweedler of Cornell University and Hirotaka Niitsuma of Nara Institute of Science and Technology.

†shirayan@cslab.kecl.ntt.co.jp

with polynomial entries. Starting with a matrix A, taking the Smith normal form of the companion matrix to A[1] yields a diagonal matrix B whose entries are polynomials which are the invariant factors of A, [2]. The Jordan canonical form of A may be obtained from B, [5].

Smith normal form is typically computed by an algorithm which might be described as a marriage of row and column reduction with the Euclidean algorithm, [2], [5]. Such algorithms, for example the Euclidean algorithm itself, are not generally convergent[2] with respect to limited precision computation. [10] presents techniques by which an algorithm which meets certain requirements[3] may be automatically translated into a new algorithm and the new algorithm may be executed with limited precision computation in a convergent manner. I.e. for the new algorithm, the output converges to the *correct output of the new algorithm* as the precision increases. Furthermore the *correct output of the original algorithm* may be automatically obtained from the *correct output of the new algorithm*[4].

The fundamental contribution of this paper is to give an algorithm to compute Smith normal form which permits the use of limited precision computation in a convergent fashion. This does not mean that the algorithm may be directly computed with limited precision computation in a convergent fashion, but rather the algorithm is amenable to the techniques in [10]. In fact the algorithm is unstable with respect to limited precision computation, as is the original question: "What is the Smith normal form of a given matrix." The techniques of [10] are applied to the algorithm in section 3.1.. Section 3.2. supplies several specific simple examples illustrating how the stabilization works. Section 3.3. gives data about results for larger examples. Section 2. presents the needed background from [10] required for later sections.

---

[1] If A is an $n \times n$ matrix let E be the $n \times n$ identity matrix. Let x be a variable. The matrix $A - xE$ is the characteristic matrix of A. It is the matrix whose determinant is the characteristic polynomial of A.

[2] In other words, if the algorithm is executed with limited precision computation, the output does not necessarily approach the *correct output* as the precision increases. Here *correct output* means the output obtained when the algorithm is executed using exact computation. This is not because the algorithm is defective, the original question may not be convergent with respect to limited precision computation. For example in the case of the Euclidean algorithm, the original question is whether two polynomials are relatively prime and this is not convergent with respect to limited precision computation. See [10] for further explanation of this point.

[3] The requirements do not rule out algorithms where the original question is not convergent with respect to limited precision computation. For example, [10] applies to the Euclidean algorithm.

[4] In fact *outputs* from the new algorithm executed with limited precision computation yield values which converge to the *correct output of the original algorithm.*

2. Automatic algorithm stabilization

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

- All the data – input, intermediate, and output – are from the polynomial ring $R[x_1, \ldots, x_m]$, where R is a subfield of the real numbers.
- Operations on data are addition, subtraction, multiplication, and division with remainder in $R[x_1, \ldots, x_m]$.
- Predicates on data have *zero discontinuity.*

Here division with remainder in $R[x_1, \ldots, x_m]$ means computing the remainder of a *dividend* polynomial modulo a *divisor* polynomial. In one variable this is 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. However, for Smith normal form we only require division with remainder in one variable and go no further into term orders.

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

$$\{\text{“TRUE”, “FALSE”}\}.$$

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 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 “$X = 3$” may be transformed into the operation (assignment to a new variable) “$Y = X - 3$” and the predicate “$Y = 0$” which has zero discontinuity.

For simplicity, we have imposed the restriction that the operations which appear are polynomial functions or division with remainder. But we could treat other functions such as square root, differentiation, etc, and this is discussed in [10].

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 $0.\overbrace{333\cdots3}^{\mu \textit{ digits}}$, 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.

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[5]. 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 it 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-

[5] 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.

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[6].

Let us summarize the techniques for automatic algorithm stabilization. Given $\mathcal{A}$, an algebraic algorithm with zero discontinuity, let $Int(\mathcal{A})$ be the stabilized algorithm for $\mathcal{A}$. $Int(\mathcal{A})$ 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 $[C, \gamma]$,

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

$Int(\mathcal{A})$ accepts interval polynomials as input and disseminates interval polynomials as output where *interval polynomials* are polynomial with interval-coefficient coefficients.

[6] 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 insures that when executed with high enough precision the modified algorithm will execute with essentially the same path of execution flow as the original algorithm executed with exact computation. This is the key to why algorithm stabilization works. See [10] for more detail.

Usually the original input $f \in R[x_1, \ldots, x_m]$[7] 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

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

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 $i_1 \ldots i_m$. In this case, we simply write $Int(f)_j \to f$. For each j, we invoke $Int(\mathcal{A})$ on $Int(f)_j$ and obtain the output $Int(\mathcal{A})(Int(f)_j)$.

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

Theorem 1 (Coefficientwise convergence, [10])

Suppose that $\mathcal{A}$ terminates normally when evaluated with input $f \in R[x_1, \ldots, x_m]$. 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 $j \geq n$ $Int(\mathcal{A})$ terminates normally on input $Int(f)_j$, and we have

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

However, coefficientwise convergence is not sufficient for the purpose of obtaining correct Smith normal forms. The *final zero rewriting* is required. Namely, we do zero rewriting for each interval-coefficient of the output $Int(\mathcal{A})(Int(f)_j)$. 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 – $Int(\mathcal{A})$ with final zero rewriting and coercing the output back to the R polynomials – as $Int(\mathcal{A})_R$.

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 $[0, 0]$. Intuitively the support is the *shape* of a polynomial. The shape of $Int(\mathcal{A})_R(Int(f)_j)$ will be the same as that of $\mathcal{A}(f)$ after a *finite* number of steps. In other words, coefficients of $Int(\mathcal{A})_R(Int(f)_j)$ that are going to converge to zero, *reach* zero in a *finite* number of steps.

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

This is supportwise convergence, which will be required for obtaining the correct shapes of Smith normal forms.

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

Theorem 2 (Supportwise convergence, [10])

Suppose we have the same hypotheses as Theorem 1. Then, there exists n such that for $j \geq n$ $Int(\mathcal{A})_R$ terminates normally on input $Int(f)_j$ and we have:

1. $Int(\mathcal{A})_R(Int(f)_j) \to \mathcal{A}(f)$, *and*
2. *There exists N such that for all $j \geq N$,*

$$Supp(Int(\mathcal{A})_R(Int(f)_j)) = Supp(\mathcal{A}(f)).$$

For specific examples where coefficientwise convergence and supportwise convergence hold see [8, 9] where they are shown to hold for Buchberger's algorithm and [7] where they are shown to hold for Sturm's algorithm.

3. Stabilization of Smith normal form computation

3.1. $Int(smith)_R$

Now let us apply the techniques for automatic algorithm stabilization described in Section 2. to computing Smith normal forms. A classical algorithm, [2], – which might be described as the marriage of row and column reduction with the Euclidean algorithm – is used to compute the Smith normal form.[8)] Let us call it *smith*. Note that *smith* is an algebraic algorithm with zero discontinuity.

We shall explain $Int(smith)$. The domain of $Int(smith)$ is a set of *interval polynomial matrices* whose entries are polynomials in one variable with interval-coefficients. Interval arithmetic is performed between interval-coefficients, when applying elementary transformations in an interval polynomial matrix or dividing an interval polynomial by another interval polynomial to obtain remainder. Zero rewriting is performed preceding evaluating the predicates whether the remainder of a polynomial modulo another polynomial is zero,

[8)] There are more efficient and practical algorithms to compute Smith normal forms, such as [4, 3, 11], etc. One could make further experiments on those algorithms to show the usefulness of the stabilization techniques. The article [6] 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.

and whether a coefficient of a polynomial is zero. For the latter predicate, however, this is implicit and hence in implementation we do zero rewriting after elementary transformations or polynomial division. This "implicitly" enables us to do zero rewriting before evaluating the predicate. Moreover, $Int(smith)_R$ does the final zero rewriting for each coefficient of each entry of the output matrix of $Int(smith)$, and coerces it back to the polynomial matrix in $M_n(R[x])$[9] .

From Theorem 2 we have:

Theorem 3 (Stabilization of *smith*)

Given $A(x) = [a_{kl}] \in M_n(R[x])$, suppose that we have a sequence of interval polynomial matrices $Int(A(x))_j = [Int(a_{kl})_j]$ which entrywise converges to $A(x)$, i.e. for each (k, l), $Int(a_{kl})_j \to a_{kl}$. Then, we have

1. *$Int(smith)_R(Int(A(x))_j) \to smith(A(x))$ entrywise, and*
2. *There exists N such that for all $j \geq N$, entrywise*

$$Supp(Int(smith)_R(Int(A(x))_j) = Supp(smith(A(x))).$$

3.2. Examples

Here are some examples illustrating how $Int(smith)_R$ works. We implemented *smith* and $Int(smith)_R$ in Maple V Release 3 on an HP9000/735. Maple has the built-in function "smith" in the linear algebra package, but this does not accept floating-point coefficients as input. To compare $Int(smith)_R$ with a naive floating-point approximation of *smith*, we needed to implement *smith* so as to accept floating-points, as well as algebraic numbers.

Example 1

Consider A in $M_3(\mathbf{Q})$:

$$A = \begin{bmatrix} 2 & 0 & 1 \\ -1 & 1 & -1 \\ -1 & 0 & 0 \end{bmatrix}.$$

[9] $M_n(R[x])$ denotes the set of $n \times n$ matrices whose entries lie in $R[x]$, the ring of polynomials having coefficients in R in one variable x.

Then, the result of $smith(A(x))$ (the Smith normal form of $A - xE$) is:

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

Now as a perturbation direction matrix, consider

$$F = \begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & -1 & -1 \end{bmatrix}.$$

Let ϵ be a perturbation size, say 0.000001, and $\tilde{A}_\epsilon$ be the perturbed matrix $A + \epsilon F$. Then, $\tilde{A}_\epsilon(x)$ has the Smith normal form:

$$smith(\tilde{A}(x)) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1.0000 & 0 \\ 0 & 0 & p(x) \end{bmatrix}$$

where $p(x) = 1.0x^3 - 3.0x^2 + 3.0x - 0.999997$.

Note that for any small ϵ the "shape" of the Smith normal form does not coincide, in other words, $\epsilon \to 0$ does not imply that $smith(\tilde{A}_\epsilon(x)) \to smith(A(x))$.

Now, for example, as a convergent sequence of interval polynomial matrices for $A(x)$, we consider "$[A + 10^{-j}F, 10^{-j}] - x[E, 0]$", i.e., $Int(A(x))_j$ is

$$\begin{bmatrix} [2 - 10^{-j}, 10^{-j}] - x & [0, 10^{-j}] & [1, 10^{-j}] \\ [-1 + 10^{-j}, 10^{-j}] & [1, 10^{-j}] - x & [-1 + 10^{-j}, 10^{-j}] \\ [-1, 10^{-j}] & [-10^{-j}, 10^{-j}] & [-10^{-j}, 10^{-j}] - x \end{bmatrix}$$

where the coefficient of x should be $-[1, 0]$, but to save space it is omitted. Obviously $Int(A(x))_j \to A(x)$ and we apply $Int(smith)$ to $\{Int(A(x))_j\}_j$. By experimentation, for just $j = 2$ we have: $Int(smith)_R(Int(A(x))_2) =$

$$\begin{bmatrix} 1.0 & 0 & 0 \\ 0 & 1.0 & 0 \\ 0 & 0 & 1.0x^3 - 3.0x^2 + 3.0x - 1.0 \end{bmatrix}$$

which is not yet equal to $smith(A(x))$ in supports. However, for $j = 3$, we have:

$Int(smith)_R(Int(A(x))_3)) =$

$$\begin{bmatrix} 1.00 & 0 & 0 \\ 0 & 1.00x - 1.00 & 0 \\ 0 & 0 & 1.00x^2 - 2.00x + 0.997 \end{bmatrix}$$

which now reaches the same support as $smith(A(x))$. For $j = 4, \ldots, 10$ we observed coefficientwise convergence on that support.

As shown in Example 1, in general, $smith(A(x))$ may exhibit instability when A has a multiple eigenvalue α in the field R and more specifically when the last elementary divisor $e_n(x)$ of $A(x)$ has a multiple factor $(x - \alpha)^k$ $(k \geq 2)$. All of our examples will satisfy the condition.

Example 2

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

Then,

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

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

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

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 $p(x) = 1.0000x^4 - 3.9598x^3 + 5.9237x^2 - 3.9677x + 1.0039598$. We made significant digits higher until $j = 1000$ with no success. What about $Int(smith)_R$? As a convergent sequence $\{Int(A(x))_j\}_j$ of interval polynomial matrices for $A(x)$, we consider "$[float_j(A(x)), error_j(A(x))]$", each coefficient of whose entry is an interval based on the floating-point approximation of the corresponding coefficient of A(x) to significant digits j and its error bound. Then,

$Int(smith)_R(Int(A(x))_2) =$

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

Too small precision. But
$Int(smith)_R(Int(A(x))_3) =$

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

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

The final example involves irrational numbers.

Example 3

Consider A in $M_4(\mathbf{Q}(\sqrt{2}))$*:*

$$A = (3608\sqrt{2} - 8545)^{-1} \times$$

$$\begin{bmatrix} 6322-8545\sqrt{2} & 252\sqrt{2}-186 & -24\sqrt{2}-408 & -40\sqrt{2}+959 \\ -1788 & -8041\sqrt{2}+6844 & -48\sqrt{2}-816 & -80\sqrt{2}+1918 \\ -9834 & 2772\sqrt{2}-2046 & -8809\sqrt{2}+2728 & -440\sqrt{2}+10549 \\ -5364 & 1512\sqrt{2}-1116 & -144\sqrt{2}-2448 & -8785\sqrt{2}+12970 \end{bmatrix}.$$

Then, we have:

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

Let us again simply apply *smith* to $\tilde{A}(x)$, a floating-point approximation of $A(x)$ to significant digits 5.

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

where $p(x) = 1.0000x^3 - 7.8284x^2 + 16.141x - 9.9985$.

Unstable. Similarly, even with $j = 1000$, we did not get a stable result. However, using our techniques, although with precision less than 5 we were not successful, with precision 5 and more, we obtained the correct (reasonably approximate) result:

$Int(smith)_R([Int(A(x))_5]) =$

$$\begin{bmatrix} 1.0000 & 0 & 0 & 0 \\ 0 & p_1(x) & 0 & 0 \\ 0 & 0 & p_2(x) & 0 \\ 0 & 0 & 0 & p_3(x) \end{bmatrix}$$

where $p_1(x) = 1.0000x - 1.4144$, $p_2(x) = 1.0000x - 1.4143$, $p_3(x) = 1.0000x^2 - 2.8282x + 1.9995$.

3.3. Experimental results

We have observed stability of $Int(smith)_R$ for a few simple examples. In this section, for many examples of larger size, we present graphic data regarding:

- CPU time
- Comparison of our method with naive floating-point computation and exact computation
- Ratio of CPU time for exact computation and our method
- The number of interval-coefficients which get rewritten to zero by zero rewriting for an execution of $Int(smith)_R$
- Minimum precision giving the correct support (N as found in Theorem 3)

Input samples were randomly generated so as to have multiple eigenvalues. That is, let D be a diagonal matrix where the same real number appears more than one times on the diagonal line, and make $A = P^{-1}DP$ where P is a random non-singular integer matrix. Maple's random function was used for P. For simplicity, we experimented with two cases: 1) where all the eigenvalues of A are 2 and 2) where all the eigenvalues of A are $\sqrt{2}$. In case (2), the above P also involves $\sqrt{2}$ randomly at a few entries. Each graph is based upon the mean of the results for ten different samples for each size of the input matrix.

Let us explain the figures. In all figures the horizontal axis indicates the size n of an input $n \times n$ matrix. The vertical axis except for the last figure is on logarithmic scale in base 10.

For figure 1 the vertical axis is CPU time in seconds, in two cases: 1) where the eigenvalue is $\lambda = 2$ and 2) where the eigenvalue is $\lambda = \sqrt{2}$. `smith`, `Int(smith)`, `Naive_fp(smith)` denote exact computation of *smith*, computation of $Int(smith)_R$, and naive floating-point computation of *smith* (without any zero determination), respectively. For `Int(smith)` the precision used is the smallest precision yielding the correct support and `Naive_fp(smith)` is run at the same precision. See Figure 4 to see what this precision is.

Figure 2 shows the ratio of the CPU time of `smith` over the CPU time of `Int(smith)`, obtained from Figure 1.

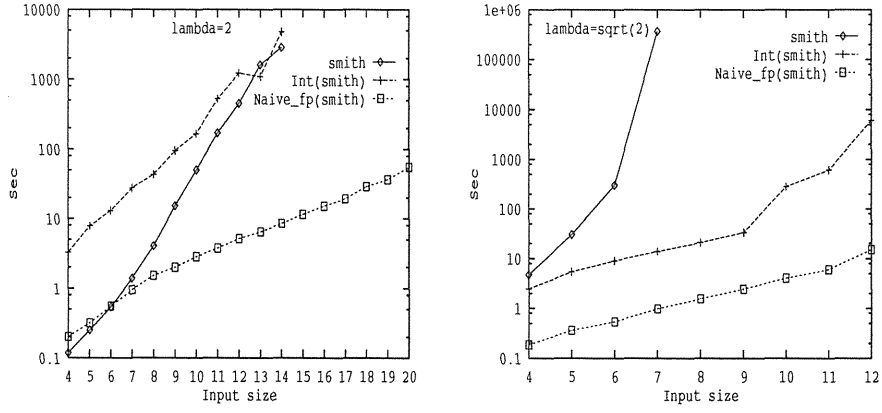

Fig. 1. Timings

Figure 3 shows the number of interval-coefficients $[C_j, \gamma_j]$ where C_j is *not* zero but $|C_j| \leq \gamma_j$ (so that $[C_j, \gamma_j]$ gets rewritten to $[0, 0]$), in one execution of `Int(smith)` in Figure 1.

Figure 4 shows at what precision the correct support first appears, i.e. precision digits N as found in Theorem 3, in `Int(smith)`.

Remarks and observations. From Figure 1, as we had expected, naive floating-point computation `Naive_fp(smith)` was fastest, but even with precision $j = 1,000$, it was not possible to obtain the correct support. Note that we also tested a *naive interval method* which uses interval arithmetic *without zero rewriting*, but the result was similar to that of `Naive_fp(smith)`. Namely, without zero rewriting the correct support was not obtained, again, with precision $1,000$. On the other hand, our method `Int(smith)` gave a stable result with a reasonable size of precision digits as shown in Figure 4. Hence, zero rewriting really works. This can also be confirmed in Figure 3. Roughly speaking, computing times of `smith`, `Int(smith)`, `Naive_fp(smith)` seem to grow as input size increases, linearly on logarithmic scale, i.e. exponentially on usual scale. From Figures 1 and 2 it may be stated that while for rational coefficients of moderate size, `Int(smith)` is not necessarily faster than `smith` (at least for a moderate size of input matrices), for complicated coefficients involving irrational numbers, `Int(smith)` is *almost always* far faster than `smith`. Moreover, although `Int(smith)` is slower than `Naive_fp(smith)` by the computational cost of performing interval arithmetic and zero rewriting, it is far more

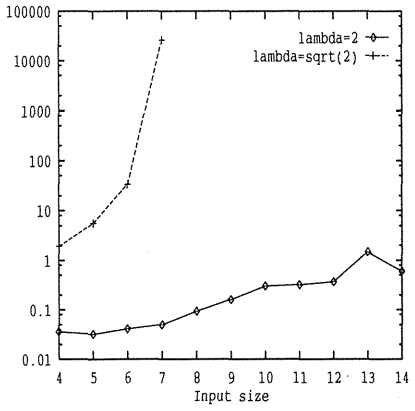

Fig. 2. Ratio of time(`smith`)/time(`Int(smith)`)

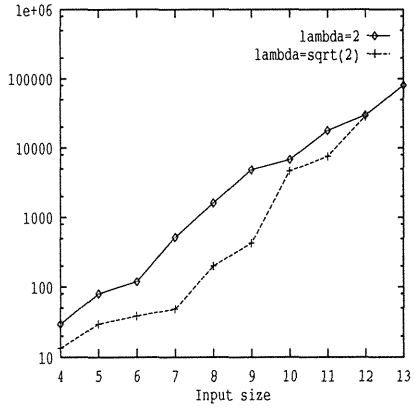

Fig. 3. Number of interval-coefficients rewritten to zero

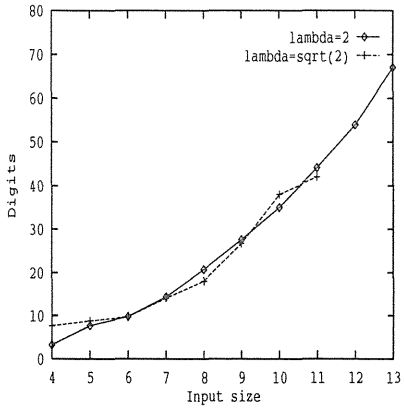

Fig. 4. Minimum precision giving the correct support

stable.

4. Conclusion

Our method gives a convergent or stable way to compute Smith normal forms of matrices in $M_n(R[x])$ without too much increased overhead beyond naive fixed precision floating point computation which is not stable.

The method also gives rise to a natural question for further study. The question is: given a matrix $A(x) \in M_n(R[x])$, what precision is necessary for $Int(smith)_R$ applied to $A(x)$ to yield the correct support? This could be answered in terms of a "bound" or "general case" and "worst case" behavior.

References

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

[2] D. Dummit and R. Foote. *Abstract Algebra.* Prentice Hall, 1991.

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

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

[5] S. Lang. *Introduction to Linear Algebra.* Addison-Wesley, 1965.

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

[7] H. Sekigawa and K. Shirayanagi. Zero rewriting in interval computation and its application to Sturm's algorithm. poster of ISSAC'95, 1995.

[8] K. Shirayanagi. An algorithm to compute floating point Gröbner bases. In T. Lee, editor, *Mathematical Computation with Maple V: Ideas and Applications*, pages 95–106. Birkhäuser, 1993.

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

[10] K. Shirayanagi and M. Sweedler. A theory of stabilizing algebraic algorithms. Technical Report 95-28, Mathematical Sciences Institute, Cornell University, 1995. 92 pages.

[11] G. Villard. Computation of the Smith normal form of polynomial matrices. In *Proceedings of ISSAC'93*, pages 208–217, Kiev, Ukraine, 1993.