To my mother and father,
And to all the magic makers.

Acknowledgements

Many people have made contributions along my doctoral journey and I wish to thank a special few in particular. Cleve Moler was responsible for introducing me to MACSYMA and the glorious possibilities of symbolic mathematics as well as instilling in me an interest in matrices and by example, good programming style. Steve Pruess provided much advice in the early stages of this work and first introduced me to the techniques of numerical analysis. Jeff Golden has helped me much with unraveling the inner workings of MACSYMA, and Ellen Golden greatly eased my immersion into the world of MIT-MC when I first became interested in this subject many moons ago and MACSYMA resided nowhere else. Daniel Finley, Deborah Sulsky and Mary Fuka furnished interesting sets of matrices that tested the limits of various symbolic algorithm implementations. Archie Gibson, Tom Kyner and Colston Chandler kindly allowed me to use (and abuse) their Sun workstations, and Jim Goodwin provided a variety of aid and inspiration in many of my computing endeavors. Jackie Damrau did a marvelous job of typing and making this thesis a wonder of LaTeX and I owe her a lot.

Finally, I would like to especially thank my advisor, Stanly Steinberg, who encouraged and endured me over the many years, who was primarily responsible for providing a superior computing environment in the Mathematics and Statistics Department, of which I took full advantage, and who helped me to see the links when at times I lost myself in the sandtraps. Thanks Stan!

Symbolic Calculation and Expression Swell Analysis of Matrix Determinants and Eigenstuff

Michael James Wester

B.S. Physics and Mathematics, University of New Mexico, 1978
M.A. Mathematics, University of New Mexico, 1980
Ph.D. Mathematics, University of New Mexico, 1992

Computer algebra systems (CASs) have become an important computational tool in the last few decades. They permit the possibility of exact, error-free solutions of long, complicated problems. One type of calculation especially suitable for a CAS is the computation of matrix invariants, in particular, determinants, characteristic polynomials and eigenvalues. Several different methods for calculating these quantities within a symbolic context are explored in depth here. Included are minor expansion procedures, various fraction-free Gaussian elimination algorithms, similarity transformations to a upper Hessenberg or tridiagonal form, and several miscellaneous methods. Implementation details are carefully examined as they can have profound effects on the success of a computation. For example, it is advantageous to minimize the complexity of the pivots used in a matrix reduction, and so an operational definition of expression complexity is presented. Also, the theory of performing permutations implicitly with the aid of an indexing array is discussed at some length.

One of the most serious problems connected with exact arithmetic is expression swell, in which the size of expressions involved in a calculation grow dramatically as the calculation progresses. Sometimes, drastic cancellation will occur at the end, resulting in a comparatively simple final answer. This is known as intermediate expression swell. In order to quantitatively estimate these phenomena for a given computation, a worst case arithmetic is developed which allows bounds on expression swell to be simply predicted for certain classes of calculations. A series of case studies is presented in conjunction with theoretical analyses, yielding findings that are often quite striking, even for small matrices.

An overview of what features make up a good symbolic matrix package (data structures, arithmetic, structural and mathematical operations, user interface) is given, using ideas from existing packages and one developed as part of this study. This is followed by a survey of the behaviors of various determinant and characteristic polynomial implementations in the above packages when applied to a number of different matrices. Each procedure had its problems, with some algorithms (such as single-step Bareiss) being moderately successful, while others (e.g., the method of Leverrier) were uniformly slow.

Contents

List of Figures

List of Tables

Prologue

The Eigenbeaver

'Twas burlap and the slimey cloves
Of garlic did putrify in the cake.
All messy was the coriander
When with the lone stash it was baked.

"Beware the Eigenbeaver my son!
The matrix that deceives, the roots that thrash!
Beware the characteristic polynomial, and shun
The furious system crash!"

He took his brackish manual in hand:
Long time the Macsyma foe he fought—
So tested he a problem tree
And became confused in thought.

And as in swirling symbols he stood,
The Eigenbeaver, with vectors askew,
Came grinding through the linear wood,
And eating determinant stew!

One, two! Three, four! Five, six, seven and more.
Each algorithm tried a different tack.
Some cut a slice, others weren't so nice,
But in the end the Eigenbeaver lay slack.

"And hast thou slain the Eigenbeaver?
It was a very good try, my son.
But I should have said, he won't stay dead,
And your task is far from done!"

'Twas burlap and the slimey cloves
Of garlic did putrify in the cake.
All messy was the coriander
When with the lone stash it was baked.

The Dance of the Symbol Man

The spring winds blow, blow, blow.
The symbolic computations go, go, go.
Pointers swirl, symbols flow,
But what at last is there to show?

Can one see the dancer for the dance?
Can one squint and see him at a glance?

The programmer programs madly,
Sometimes well, sometimes badly.
Computations, he always starts glady;
It's just the results he examines sadly.

Can one distinguish the dancer from the dance?
If one is fast enough, is there any chance?

One by one, fill the memory bins
As each symbol twists and swirls and spins,
While above the programmer does the same and grins.
Who's to tell which provides the greater sins?

Can one separate the dancer from the dance?
Is it all the same, each immersed in the other's stance?

Do it all again, faster, faster!
Find the answer and prevent disaster.
Tweak the code—just add plaster.
Programmer or program, which is the master?

Can one know the dancer without the dance?
Can one be wooed without the other perceived askance?

It is at last a scene of mad confusion,
Of broken algorithms and contusion.
Simplicity is revealed to be merely illusion,
And programmer and program, once apart, are now forever joined in fusion.

Can one live as the dancer and the dance?
Listen. The music starts. The audience chants.

Chapter 1

Introduction

One of the important developments in computational techniques in the last 30 years or so has been the evolution of computer algebra systems (CASs) or symbolic mathematical computer programs. These systems allow the direct manipulation of mathematical symbols, and can perform arithmetic, algebra, calculus, etc. on integers, polynomials, rational and transcendental functions, matrices, tensors and much more. CASs are quite distinct from the usual numerical program, whose goal typically is to use approximate arithmetic to produce approximate answers, numerical and possibly graphical, to a definite set of problems. Some numerical programs are extensible, providing a user level programming language with which features not originally anticipated by the designers can be added in a systematic way. MATLAB [Mol80] is one such example. CASs, on the other hand, use exact arithmetic to produce exact answers to varied sets of problems, with most also allowing user programmability. In addition, there is an increasing trend for general purpose CASs to include numerical and graphical features along with their symbolic capabilities in order to create integrated computational environments.

CASs originally began as special purpose routines for manipulating limited types of formulas. In the late 1960s–early 1970s, the first of the general purpose systems started to come out. These included REDUCE (1968) [Hea87] and MACSYMA (1970) [Bog83, Com88], both written in Lisp. Other important Lisp based systems appeared in the following decade, including Scratchpad II (1975) [Sut88] and muMATH (1979) [Sof83], the latter being the first general purpose CAS specifically developed for a personal computer. These last two systems have now been succeeded by Axiom (1991) and Derive (1988), respectively. The 1980s brought forth SMP (1982), whose successor is Mathematica (1988) [Wol88], and MAPLE (1985) [Cha88], both of which are based on the C language. In addition to the above general purpose systems, a number of special purpose packages have been developed. Five of these are FORM (for large-scale formula manipulation), Cayley (discrete algebra and combinatorial theory), PARI (number theory), SHEEP (relativity) and STENSOR (tensor and spinor manipulation) [Hör86]; however, there are many more. [Ged82, Chapter 1] contains further information on the history of CASs.

CASs provide several important advantages over humans when performing symbolic computations. They allow large, complicated calculations to be carried out that would not be feasible to do by hand. Part of this is the computer's innate ability to easily handle large chunks of data, with the example here being complex mathematical expressions. In addition, CASs can produce error-free results, given a consistent and correct set of rules appropriate for the mathematical objects being manipulated. Often, there is a great deal of flexibility about the way a particular problem can be approached. A good CAS can permit an answer to be checked by being able to redo the problem in a second way or by doing the inverse of the process that was originally performed (such as differentiating an indefinite integral). Moreover, once a result has been derived, the system can evaluate it numerically (and graphically) or write a program in a more conventional computer language that incorporates the expressions that were computed.

Some of the algorithms present in CASs are just simple adaptations of methods commonly applied by hand. However, in a number of instances, very sophisticated procedures have been implemented which allow large classes of problems to be solved automatically. Some examples are the Risch integration algorithm, methods for multivariate Taylor series, the use of multiple point evaluation and Hensel lifting in polynomial factorization, and Grobner basis schemes for solving systems of polynomial equations [Akr89, Ged82, Knu81]. For the most part, these procedures have been designed with large problems in mind, and are often quite different from the collection of heuristic techniques and simple algorithms a human might use to attack a question. Humans generally prefer to avoid the tedium of a long, messy computation, and this is exactly what computers do best.

One area of mathematics for which symbolic computation is especially suitable is linear algebra, in particular, operations involving matrices. A great deal of literature has been written on the theoretical aspects of matrices (and linear algebra), including such works as [Bel70, Bra75, Cul72, Gan60, Joh81, Str88]. In addition, much effort has been expended in trying to discover how to do efficient numerical calculations involving matrices. Some of the references here are [For67, For77, Fox65, Gol83, Hen64, Sch73, Wil65]. Techniques of linear algebra are used in just about every mathematical domain. Matrix applications crop up throughout science and engineering. Thus, although the subject has not exactly been neglected, it is somewhat surprising that more emphasis has not been placed on devising good robust symbolic matrix techniques that are appropriate for use in a CAS, like has been done for indefinite integration, for example.

A problem of especial interest concerning (square) matrices is the computation of their similarity invariants, in particular, determinants, characteristic polynomials and eigenvalues. Indeed, a major set of numerical routines is concerned with just the computation of eigenvalues (and eigenvectors) [Gar75]. These three quantities (and eigenvectors, which are related but are not themselves invariants) show up in all manner of situations, including in some rather unexpected places. As just one example, an eigensystem makes an appearance in one version of the method for determining the real number α that has a given continued fraction expansion which at some point starts to be periodic. The continued fraction algorithm produces a converging sequence of rational approximations $\left\{\frac{p_0}{q_0}, \frac{p_1}{q_1}, \ldots\right\}$ to α, which

are 'best' in some sense. These rational approximations are computed by

$$\frac{p_n}{q_n} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots + \cfrac{1}{a_{n-1} + \cfrac{1}{a_n}}}} ,$$

where $a_k = [\alpha_k]$ (that is, the greatest integer in α_k) and

$$\alpha_0 = \alpha , \quad \alpha_{k+1} = \frac{1}{\alpha_k - a_k} , \quad (k = 0, 1, \ldots).$$

If α is rational, then this procedure will eventually terminate; otherwise it will not. Suppose α is irrational, then it will be true that

$$M_0 \equiv \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} , \quad M_n \begin{pmatrix} 1 \\ \alpha_n \end{pmatrix} \equiv \begin{pmatrix} q_{n-2} & q_{n-1} \\ p_{n-2} & p_{n-1} \end{pmatrix} \begin{pmatrix} 1 \\ \alpha_n \end{pmatrix} = \mu_n \begin{pmatrix} 1 \\ \alpha \end{pmatrix} , \quad (n = 0, 1, \ldots),$$

where $\det M_n = (-1)^n$ and $\mu_n = q_{n-2} + \alpha_n q_{n-1}$ [Sta70, p. 226]. If the expansion of α is ultimately periodic, then $a_{n+\ell} = a_n$ for all $n \geq N$ for some N and ℓ. This will imply that $\alpha_{N+\ell} = \alpha_N$. Hence,

$$M_N^{-1} \mu_N \begin{pmatrix} 1 \\ \alpha \end{pmatrix} = \begin{pmatrix} 1 \\ \alpha_N \end{pmatrix} = \begin{pmatrix} 1 \\ \alpha_{N+\ell} \end{pmatrix} = M_{N+\ell}^{-1} \mu_{N+\ell} \begin{pmatrix} 1 \\ \alpha \end{pmatrix}$$

which results in the eigensystem

$$M_{N+\ell} M_N^{-1} \begin{pmatrix} 1 \\ \alpha \end{pmatrix} = \frac{\mu_{N+\ell}}{\mu_N} \begin{pmatrix} 1 \\ \alpha \end{pmatrix} ,$$

where α is to be determined. This formula can be used to show that α will be the (irrational) root of a quadratic equation with integer coefficients [Sta70, p. 231].

In the following chapters, some of the more prominent determinant and characteristic polynomial algorithms will be examined in depth within the context of a symbolic computing environment. Procedural details and operation counts will be discussed extensively in Chapter 2, in which some non-traditional methods are presented, as well as common schemes such as expansion by minors and Gaussian elimination. The non-traditional methods include a new sparse fraction-free determinant algorithm and direct characteristic polynomial procedures, involving similarity transformations to a Hessenberg or tridiagonal form, which have not previously been applied in a symbolic setting. It will be seen that many of the methods work by transforming the problem to a simpler domain where answers can be arrived at fairly easily, and then if the results are not invariants, the solution in the original domain can be constructed from the values determined.

Probably the most serious problem connected with exact arithmetic is expression swell. This is the dramatic growth in the size of expressions as a calculation progresses. Sometimes

at the end of a computation, drastic cancellation occurs and the final result is comparatively simple. In this case, the phenomenon is called intermediate expression swell. Expression swell, in either form, can have a profound influence on the success or failure of a calculation. In Chapter 3, a novel worst case arithmetic is developed which allows bounds on expression swell to be simply computed for certain classes of calculations. These procedures are then selectively applied to some of the methods of the previous chapter. For matrices that are initially filled with integers or rational numbers of a known size, this analysis is able to yield bounds on the size of the determinant and the entries of similar Hessenberg matrices that can be reasonably tight when compared with the actual results. Other case studies are also presented, and it is possible to predict that the reduction of a not very large Hermitian matrix into tridiagonal form will generate so much more expression swell than treating the matrix as general and transforming it into Hessenberg form, that the former method will be of doubtful utility.

Chapter 4 discusses what features would make up a good symbolic matrix package, covering such topics as data structures, arithmetic (including an operational definition of expression complexity), structural and mathematical operations that should be available, and the user interface. These comments are provided in the light of what existing systems furnish as well as that offered by a package implemented in MACSYMA as part of this study.

Finally, Chapter 5 looks at the results of a number of computer experiments that involved determinant and characteristic polynomial calculations. The implementations examined each had their strengths and weaknesses (although the strong points of some were not all that good!), with no one method being uniformly successful. A variety of observations are recorded, with one of the most pointed being that implementation details are often crucial, although algorithm choice is, of course, a very important decision.

This study provides a basis for the rational design of general purpose determinant and characteristic polynomial routines. To be reasonably effective, such routines need to be able to employ different approaches depending on the structure of the matrices handed to them. The observations and analyses in the following text establish a framework for writing generic procedures that can take advantage of the special properties of a given matrix.

To aid the reader, many of the common notations used in the following chapters have been collected together for easy reference in Table 1.1. The notations for the various algebraic structures frequently referred to (e.g., the integers) are listed separately in Table 1.2. Notation also will often be defined when it is first encountered in the text.

$\Longleftrightarrow$	if and only if
$\equiv$	is defined to be (or is congruent to, in Section 2.1.3)
$\sim$	is of the same order as
$\in$	is a member of
δ_{ij}	Kronecker delta (= 1 if $i = j$, and is 0 otherwise)
$\bar{x}$	complex conjugate of x
$\lfloor x \rfloor$	floor of x (the largest integer $\leq x$)
$\lceil x \rceil$	ceiling of x (the smallest integer $\geq x$)
$\mathrm{sign}(x)$	-1, 0 or 1 for $x <$, $=$ or > 0, respectively
$\deg(f(x))$	highest degree of x present in $f(x)$
$f^{(n)}(x)$	n^{th} derivative of f evaluated at x $[f^{(0)}(x) = f(x)]$
$O(x^n)$	terms of the order of x^n
$\gcd(x_1, x_2, \ldots)$	greatest common divisor of $x_1, x_2, \ldots$
$\mathrm{lcm}(x_1, x_2, \ldots)$	least common multiple of $x_1, x_2, \ldots$
$\binom{n}{m}$	number of ways of choosing m objects out of n in which the order of selection is unimportant
$\mathcal{N}_\beta(n)$	number of base β digits in the integer n
$\overline{n}$	equivalence class of integers containing exactly n digits
$\frac{\overline{m}}{\overline{n}}$	equivalence class of rational numbers with m-digit numerators and n-digit denominators
A	a matrix
$\mathbf{A}_i$	i^{th} column of A
a_{ij}	(i, j)-element of A
A^T	transpose of A
A^H	Hermitian of A $(= \bar{A}^T)$
$\mathrm{tr}\, A$	trace of A $(=\sum_i a_{ii})$
$\det A$	determinant of A
$m_A(\lambda)$	minimum polynomial of A
$c_A(\lambda)$	characteristic polynomial of A
O	zero matrix
I	identity matrix
P_{ij}	simple permutation matrix
$C_{p(\lambda)}$	companion matrix of the polynomial $p(\lambda)$
$\mathrm{diag}(d_1, d_2, \ldots)$	(block) diagonal matrix with diagonal elements (blocks) $d_1, d_2, \ldots$
$\mathbf{b}$	a column vector
b_i	i^{th} element of $\mathbf{b}$
$(\mathbf{b}_1 \vert \mathbf{b}_2 \vert \cdots)$	matrix formed from the horizontal concatenation of $\mathbf{b}_1, \mathbf{b}_2, \ldots$
$\mathbf{e}_i$	i^{th} elementary unit vector $[(\mathbf{e}_i)_j = \delta_{ij}]$

Table 1.1 Common notations.

Notation	Algebraic Structure
$\mathbf{Z}$	Integers
$\mathbf{Z}_p$	Integers modulo p
$\mathbf{Q}$	Rational numbers
$\mathbf{R}$	Real numbers
$\mathbf{C}$	Complex numbers
$\mathbf{Z}[x]$	Univariate polynomials with integer coefficients
$\mathbf{Z}_p[x]$	Univariate polynomials with coefficients that are integers modulo p
$\mathbf{Q}[x]$	Univariate polynomials with rational number coefficients
$\mathbf{Z}(x)$	Quotient field of univariate polynomials with integer coefficients
$\mathbf{Z}[x_1, \ldots, x_v]$	Multivariate polynomials with integer coefficients
$\mathbf{Z}_p[x_1, \ldots, x_v]$	Multivariate polynomials with coefficients that are integers modulo p
$\mathbf{Q}[x_1, \ldots, x_v]$	Multivariate polynomials with rational number coefficients
$\mathbf{Z}(x_1, \ldots, x_v)$	Quotient field of multivariate polynomials with integer coefficients

Table 1.2 Notations used for various algebraic structures associated with polynomials and rational functions [Ged82].

Chapter 2

Symbolic Algorithms

In this chapter, several different methods for calculating the determinant and characteristic polynomial of a square matrix will be examined in the context of a symbolic computing environment. It is appropriate to consider together algorithms for finding these two matrix invariants as their definitions are highly intertwined. Indeed, the characteristic polynomial of a matrix A, $c_A(\lambda)$, is defined in terms of the determinant of the characteristic matrix of A:

$$c_A(\lambda) = \det(A - \lambda I) \; . \tag{2.1}$$

In the other direction, the constant term of $c_A(\lambda)$, once computed by whatever means, will be equal to $\det A$ [take $\lambda = 0$ in (2.1)]. Therefore, the algorithms for computing one of these invariants will naturally lead to the algorithms for computing the other. The progression from determinant to characteristic polynomial is a natural one and so the development here shall be in that order.

Typically, the matrices under consideration will be assumed to have elements taken from $\mathbf{Z}(x_1, \ldots, x_v)$, i.e., rational functions of multivariate polynomials with integer coefficients. The algorithms will certainly all work with more general matrix elements, but detection of zeroes and arrangement of the result into a normal form cannot then be guaranteed in all circumstances. Performing an operation count on an algorithm will produce numbers that will tally rational operations, which are distinct from and more complex than polynomial operations (more on this later).

Many of the procedures discussed below are well known adaptations of common linear algebra techniques. However, some of the methods covered have not previously been applied in a symbolic situation. These include the computation of the characteristic polynomial via an intermediate Hessenberg or tridiagonal form (see Sections 2.2.2 and 2.2.3). In addition, I have developed a sparse variant of single-step fraction-free Gaussian elimination (see Section 2.1.2) which has proved to be of value in calculating the determinants of certain sparse matrices (see Chapter 5).

$+$	$n!-1$
$-$	$\frac{1}{2}n!$
$\times$	$(n-1)n!$

Table 2.1 Rational operation counts for computing the determinant of an $n \times n$ matrix from the definition.

2.1 Determinant Algorithms

The traditional definition of the determinant of an $n \times n$ matrix A [Bra75] is the weighted sum of all possible products of n elements such that every element in a given product is taken from a distinct row and a distinct column of the matrix. In other words, each term in the sum is a product of n matrix elements whose row and column indices, when read from left to right, separately form permutations of the integers $1, \ldots, n$. All rearrangements of the factors in a product are equivalent for algebraic systems possessing commutative multiplication, so a simple choice (e.g., the identity permutation) can be made for one of the sets of indices. Normally, the row indices are selected. In this case, the weights will be $+1$ for products exhibiting even permutations of the column indices and -1 for those showing odd permutations. [An even permutation of $1, \ldots, n$ can be transformed into $(1, \ldots, n)$ by an even number of swaps, and an odd permutation by an odd number of swaps.] This can be stated in symbols as

$$\det A = \sum_{\mathbf{j} \in \pi_n} (-1)^{\mathcal{P}(\mathbf{j})} \prod_{i=1}^{n} a_{ij_i} \ , \tag{2.2}$$

where π_n is the set of all permutations of $1, \ldots, n$ and $\mathcal{P}(\mathbf{j})$ is the parity (0 for even, 1 for odd) of the permutation $\mathbf{j} \equiv (j_1, \ldots, j_n)$.

Computing $\det A$ from equation (2.2) is not practical, except for very small n ($n \leq 3$). Conducting an operation count on this formula produces Table 2.1. Here, $+$, $-$ and $\times$ refer to the number of binary additions, unary negations and binary multiplications required to compute the determinant. These are all rational operations when the elements of A are themselves rational.

Since the definition (2.2) is a poor way to compute determinants except in the simplest cases, it is good that alternative methods are available which are much more efficient. The two procedures most commonly employed in CASs are expansion by minors and Gaussian elimination, both in various forms. These methods will be discussed in detail below in separate subsections. In addition, $\det A$ can be computed by transforming A into Smith normal form [Cul72, p. 229] and taking the product of the diagonal elements. This method, which makes use of elementary row and column operations, can sometimes lead to a partial factorization of the determinant; however, it is not a very efficient way to do this. The

Smith normal form is more commonly employed in characteristic polynomial computations and thus is discussed more fully in the next section.

A useful feature of determinants is that some of their properties can be extended to matrices in block form, that is, to matrices that have been partitioned into submatrices. The submatrices are treated in a manner analogous to the scalar elements in a regular matrix. In particular, the determinant of a block triangular matrix is the product of the determinants of the diagonal blocks, all of which must be square [Gan60, p. 43]. For example, if T is block upper triangular, of the form

$$T = \left(\begin{array}{ccc|c|cc} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \hline 0 & 0 & 0 & \times & \times & \times \\ \hline 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{array}\right) = \begin{pmatrix} T_{11} & T_{12} & T_{13} \\ 0 & T_{22} & T_{23} \\ 0 & 0 & T_{33} \end{pmatrix} , \tag{2.3}$$

then

$$\det T = (\det T_{11})(\det T_{22})(\det T_{33}) .$$

It would therefore be useful to be able to arrange an arbitrary matrix into block triangular form as that would decompose the calculation into simpler subproblems. It turns out that by performing row permutations (noting that each individual swap will change the sign of the determinant), any matrix can be put into this form, although it can turn out that the whole matrix may be a single block! The number of diagonal blocks will be maximized [Wan76] if the rows of the matrix are permuted so that no element along the diagonal is zero. If this cannot be done, then the matrix is singular (i.e., the determinant is zero). In the subsections that follow (including the characteristic polynomial algorithms), it is highly recommended to perform such a decomposition first before using any particular method.

2.1.1 Expansion by Minors

An m-minor of an $n \times n$ matrix A is the determinant of a square submatrix obtained from the intersections of m rows and m columns chosen from A. (Equivalently, the submatrix can be obtained by deleting $n - m$ rows and $n - m$ columns from A.) For example, the $(n-1)$-minor $\det A'_{ij}$ is associated with the submatrix of A, A'_{ij}, obtained by deleting row i and column j.

$\det A$ can be computed in terms of A's minors. In the general case [Gen74], let some m satisfying $1 \leq m < n$ be selected. Pick m columns from A. Now, there will be $\binom{n}{m}$ possible choices of m rows available. For each choice, there exists both an m-minor and an $(n-m)$-minor formed from the complementary sets of rows and columns. $\det A$ will then be the sum over all possible choices of m rows of the product of these two minors with each term weighted by ± 1. Obviously, this "expansion by minors" can occur along rows as well as down columns.

As a simple example, consider the expansion by 2-minors of the determinant of the general 4×4 matrix B. Introducing the minor notation [Gan60, p. 2]

$$A \begin{pmatrix} i_1 & i_2 & \dots & i_m \\ j_1 & j_2 & \dots & j_m \end{pmatrix} \equiv \begin{vmatrix} a_{i_1 j_1} & \dots & a_{i_1 j_m} \\ \vdots & & \vdots \\ a_{i_m j_1} & \dots & a_{i_m j_m} \end{vmatrix} \equiv \det \begin{pmatrix} a_{i_1 j_1} & \dots & a_{i_1 j_m} \\ \vdots & & \vdots \\ a_{i_m j_1} & \dots & a_{i_m j_m} \end{pmatrix} ,$$

then it is clear that

$$\det A = \begin{vmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{vmatrix} = A \begin{pmatrix} 1 & 2 & \dots & n \\ 1 & 2 & \dots & n \end{pmatrix} .$$

Therefore, expanding $\det B$ down the first two columns yields

$$\begin{aligned} \det B &= B \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix} B \begin{pmatrix} 3 & 4 \\ 3 & 4 \end{pmatrix} - B \begin{pmatrix} 1 & 3 \\ 1 & 2 \end{pmatrix} B \begin{pmatrix} 2 & 4 \\ 3 & 4 \end{pmatrix} + B \begin{pmatrix} 1 & 4 \\ 1 & 2 \end{pmatrix} B \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix} \\ &+ B \begin{pmatrix} 2 & 3 \\ 1 & 2 \end{pmatrix} B \begin{pmatrix} 1 & 4 \\ 3 & 4 \end{pmatrix} - B \begin{pmatrix} 2 & 4 \\ 1 & 2 \end{pmatrix} B \begin{pmatrix} 1 & 3 \\ 3 & 4 \end{pmatrix} + B \begin{pmatrix} 3 & 4 \\ 1 & 2 \end{pmatrix} B \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} . \end{aligned}$$

Each of these 2-minors can then be computed by expansion about a single row or column of the corresponding submatrix, producing formulas expressed in terms of 1-minors, which are simply the original matrix elements. Thus, expansion by minors is typically taken to be a recursive process. Minors created from the expansion of a higher level determinant are themselves expanded in terms of lower level determinants. The recursion terminates, as above, at 1-minors.

Usually, the recursive step in expansion by minors is used to express a k-minor in terms of k $(k-1)$-minors. In this situation, the procedure is often called "expansion by cofactors", where a cofactor is simply an appropriately weighted (signed) minor. In particular, the first stage in the recursive cofactor expansion of $\det A$ in which the expansion occurs along the i^{th} row is given by

$$\det A = \sum_{j=1}^{n} a_{ij} \mathcal{A}_{ij} , \tag{2.4}$$

while performing the expansion along the j^{th} column produces

$$\det A = \sum_{i=1}^{n} a_{ij} \mathcal{A}_{ij} . \tag{2.5}$$

Here, the cofactor $\mathcal{A}_{ij}$ is defined by

$$\mathcal{A}_{ij} \equiv (-1)^{i+j} \det A'_{ij} ,$$

where the weight $(-1)^{i+j}$ simply results in alternating signs as either i or j increases.

Operation counts for this procedure are given in Table 2.2a, where $e_k \equiv \sum_{\ell=0}^{k} 1/\ell!$.

$+$	$n! - 1$
$-$	$\sim \frac{1}{2}(e_{n-1} - 1)n!$
$\times$	$(e_{n-1} - 1)n!$

a. Recursive

$+$	$(n-2)2^{n-1} + 1$
$-$	$\sim \frac{1}{2}n(2^{n-1} - 1)$
$\times$	$n(2^{n-1} - 1)$

b. Iterative

Table 2.2 Rational operation counts for two different versions of computing the determinant of an $n \times n$ matrix by expansion by cofactors.

These were derived by considering the number of $(n-m)$-minors that will exist at the end of the m^{th} recursive stage $(m = 0, 1, \ldots, n-1)$ of the expansion. The initial stage at $m = 0$ has 1 n-minor (i.e., $\det A$). At $m = 1$, there will be n $(n-1)$-minors. At $m = 2$, there will be $n(n-1)$ $(n-2)$-minors. At the completion of the next to the last stage $(m = n-2)$, there will be $n(n-1)\cdots 3$ 2-minors. The total operation counts will then be the sum over all of these stages of the number of operations required to decompose the minors present at a given stage into minors of the next smaller size.

Specifically, the number of adds and multiplies needed to proceed from stage m to stage $m+1$ will be $n-m-1$ and $n-m$, respectively. Therefore, the total addition count will be

$$\begin{aligned} n - 1 + n(n-2) + n(n-1)(n-3) + \cdots + n(n-1)(n-2)\cdots 3 \cdot 1 \\ &= \sum_{m=2}^{n} \frac{n!}{m(m-2)!} = n! \sum_{m=2}^{n} \frac{m-1}{m!} \\ &= n! \left[\sum_{m=2}^{n} \frac{1}{(m-1)!} - \sum_{m=2}^{n} \frac{1}{m!} \right] = n! \left[\sum_{\ell=1}^{n-1} \frac{1}{\ell!} - \left(\sum_{\ell=1}^{n} \frac{1}{\ell!} - 1 \right) \right] \\ &= n! \left(1 - \frac{1}{n!} \right) = n! - 1 \ . \end{aligned}$$

Similarly, the total number of multiplies will be

$$\begin{aligned} n + n(n-1) + n(n-1)(n-2) + \cdots + n(n-1)(n-2)\cdots 3 \cdot 2 \\ &= \sum_{m=2}^{n} \frac{n!}{(m-1)!} = n! \sum_{m=2}^{n} \frac{1}{(m-1)!} \\ &= n! \sum_{\ell=1}^{n-1} \frac{1}{\ell!} = n! \, (e_{n-1} - 1) \ . \end{aligned}$$

Note that e_{n-1} is the sum of the first n terms in the Taylor series for e and converges very rapidly for increasing n. The number of negations will vary, depending exactly on which rows or columns are chosen for the expansion at each stage. However, since the signs will always alternate on any given cofactor expansion, the total number of negations will be roughly half that of multiplications.

Recursive cofactor expansion of $\det A$ can be nearly as expensive an operation as direct calculation from the definition (2.2). This analysis does not take into account, however, the fact that a minor (and all the subminors of that minor) need not be computed if the matrix element multiplier a_{ij} in any of the descendents of (2.4) or (2.5) is zero. Indeed, a clever implementation of this procedure would choose to expand each minor along the row or column that maximized the number of zeroes. If two or more rows and/or columns had this property, then the choice could either be arbitrary or in favor of the one that had the "simplest" set of elements. The latter option would be particularly desirable in a CAS if the computation of simplicity was not too expensive.

Nonetheless, even with a clever implementation, recursive cofactor expansion is still slow for most matrices (except for extremely sparse ones) since very many minors are recomputed over and over again in the course of the recursions. This is unavoidable in a recursive algorithm unless the values of the minors are preserved globally and so need not be recomputed after their first instantiation. Of course, time will then have to be spent in pattern matching (is this a new minor or has it been computed previously?) and memory reserved for the saved information. This idea is feasible but a purely iterative procedure is also possible [Gen74].

Iterative expansion by cofactors of $\det A$ starts out by choosing two rows (or columns) of A. All 2-minors determined by these rows are computed. Next, a third row is selected from A and all 3-minors determined by the expanded set of rows are calculated using the matrix elements on this new row and the values of the 2-minors previously computed. This process continues until $\det A$ (the n-minor) is produced. As soon as the set of $(m+1)$-minors is generated from the set of m-minors, the space used to store the values of the m-minors can be reclaimed as these quantities will no longer be needed.

Table 2.2b displays operation counts for the above method. They were derived by noting that at the m^{th} stage of the procedure ($m = 2, \ldots, n$), $\binom{n}{m}$ m-minors will be computed, each requiring $m-1$ adds and m multiplies over the number already performed. Therefore, the total number of multiplications required will be

$$\begin{aligned}\sum_{m=2}^{n} \binom{n}{m} m &= \sum_{m=2}^{n} \frac{n!\,m}{m!\,(n-m)!} = n \sum_{m=2}^{n} \frac{(n-1)!}{(m-1)!\,(n-m)!} \\ &= n \sum_{m=2}^{n} \binom{n-1}{m-1} = n \sum_{\ell=1}^{n-1} \binom{n-1}{\ell} = n(2^{n-1}-1)\ ,\end{aligned}$$

while the addition count will become

$$\begin{aligned}\sum_{m=2}^{n} \binom{n}{m} (m-1) &= n(2^{n-1}-1) - \sum_{m=2}^{n} \binom{n}{m} \\ &= n\ 2^{n-1} - n - (2^n - 1 - n) = 2^{n-1}(n-2) + 1\ .\end{aligned}$$

The number of negations will be approximately half the number of multiplies by an argument similar to the one used before. In [Gen74], it is shown that expansion by cofactors minimizes the number of operations (actually, multiplications) involved with respect to a general iterative minor expansion and so it is reasonable to consider just the specific algorithm discussed above rather than a more general one.

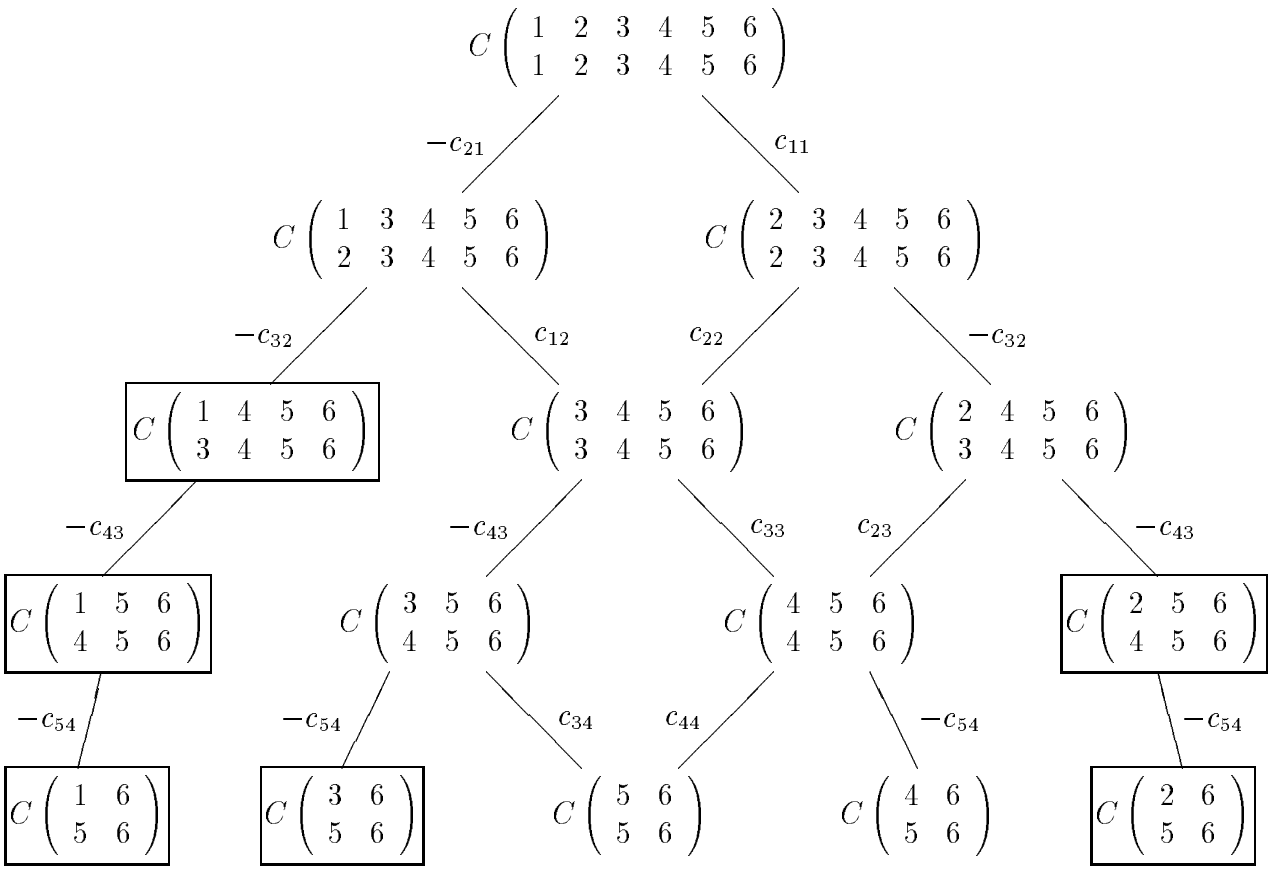


Figure 2.1 Minor interdependence of the 6×6 tridiagonal matrix C for recursive cofactor expansion about leftmost columns.

In a sense, iterative cofactor expansion is the reverse of recursive cofactor expansion in that it proceeds bottom-up instead of top-down. It has the advantage of avoiding altogether duplicate minor computations and the disadvantage of not being able to profit from knowing that the multiplier of a cofactor is zero. In [Wan77a], this latter flaw is remedied. Before the bottom-up minor computations are executed, a top-down analysis is performed to determine exactly which minors need to be calculated. This analysis produces a lattice-structure diagram which details exactly the pattern of minor interdependence. Figure 2.1 shows such a diagram for the 6×6 tridiagonal matrix

$$C = \begin{pmatrix} c_{11} & c_{12} & 0 & 0 & 0 & 0 \\ c_{21} & c_{22} & c_{23} & 0 & 0 & 0 \\ 0 & c_{32} & c_{33} & c_{34} & 0 & 0 \\ 0 & 0 & c_{43} & c_{44} & c_{45} & 0 \\ 0 & 0 & 0 & c_{54} & c_{55} & c_{56} \\ 0 & 0 & 0 & 0 & c_{65} & c_{66} \end{pmatrix}$$

in which expansion always takes place down the leftmost column of the relevant minor. The lines connecting pairs of minors are labeled with the appropriate scalar multiplier used in the cofactor expansion.

As can be seen from Figure 2.1, the top-down analysis computes the lattice that would be traversed by recursive cofactor expansion when zero multipliers are handled intelligently. Since no recursion is actually occurring, duplicate minors can be consolidated in advance. Once the analysis is completed, the needed minors can be computed in a bottom-up manner. Those minors that are boxed in the figure have a row of zeroes and hence are singular, so they do not need to be computed explicitly. A good implementation of this algorithm, such as that described in [Wan77b], should check minors for rows or columns of zeroes and immediately pare off any obviously singular branches from the recursion lattice.

No operation counts can be provided for the above hybrid algorithm since they would be intimately tied to the exact pattern of zeroes in the particular matrix under consideration. Nevertheless, it is clear that this procedure will best avail itself on sparse matrices. For large dense matrices, the overhead can become quite substantial.

2.1.2 Gaussian Elimination

The other primary method for computing determinants makes use of the properties of elementary row and column operations. Elementary row operations (with respect to an $n \times n$ matrix A) come in three forms:

$$\begin{array}{ll} \text{(i)} & \text{exchanging (swapping) rows } r \text{ and } s \quad (r \neq s), \\ \text{(ii)} & \text{multiplying row } r \text{ by } k \neq 0, \\ \text{(iii)} & \text{adding } q \text{ times row } s \text{ to row } r \quad (r \neq s). \end{array} \tag{2.6}$$

Elementary column operations are described similarly. These operations can be represented by the elementary matrices P_{rs}, $E_r(k)$ and $E_{rs}(q)$, respectively, which are defined by ($i = 1, \ldots, n$)

$$\begin{array}{ll} \text{(i)} & (P_{rs})_{ii} = 1 \quad (i \neq r, s), \quad (P_{rs})_{rs} = (P_{rs})_{sr} = 1, \\ \text{(ii)} & (E_r(k))_{ii} = 1 \quad (i \neq r), \quad (E_r(k))_{rr} = k, \\ \text{(iii)} & (E_{rs}(q))_{ii} = 1, \quad (E_{rs}(q))_{rs} = q, \end{array} \tag{2.7}$$

where any elements not explicitly specified are zero. Pre-multiplication by one of these matrices will perform the associated elementary row operation on A, while post-multiplication by the transpose will perform the corresponding column operation [$E_{rs}^T(q) \neq E_{rs}(q)$ necessitates the reference to transposes].

The idea is to use these operations to reduce A to a triangular form in which case the determinant will be simply the product of the diagonal elements. If row operations are used, the result will be upper triangular; column operations will produce a lower triangular form. The determinant of the resultant triangular matrix (call it T) will be det A times the product of the determinants of a number of elementary matrices. (This makes use of the property that the determinant of a product of matrices is the product of the determinants of the

individual matrices.) Thus, $\det A$ will be determined by $\det T$, which is easy to calculate, and the exact sequence of elementary operations used to transform A into T.

The determinant of each type of elementary matrix is given below:

$$\begin{aligned} \det P_{rs} &= -1, \\ \det(E_r(k)) &= k, \\ \det(E_{rs}(q)) &= 1. \end{aligned}$$

The last two findings are easy to see. The result for $\det P_{rs}$ follows from expansion by cofactors. The general form of the permutation matrix P_{rs} is

$$\begin{array}{c} \\ \\ \\ \\ r \\ \\ \\ \\ s \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccccccc} & & & & r & & & & s & & \end{array} \\ \begin{pmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & 0 & 0 & \cdots & 0 & 1 & & & \\ & & & 0 & 1 & & & 0 & & & \\ & & & \vdots & & \ddots & & \vdots & & & \\ & & & 0 & & & 1 & 0 & & & \\ & & & 1 & 0 & \cdots & 0 & 0 & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{pmatrix} \end{array} .$$

Expanding along the first $r-1$ columns yields factors of one. Expansion about the r^{th} column and then the r^{th} row will produce the multiplier $(-1)^{s-r}(-1)^{s-(r+1)} = (-1)^{2(s-r)-1} = -1$. The minor left over after this will be an identity matrix with determinant one.

The basic Gaussian elimination algorithm proceeds through a series of $n-1$ stages. Let $A^{(k)}$ designate the transformed matrix at the end of the k^{th} stage with $A^{(0)} \equiv A$ and suppose $T \equiv A^{(n-1)}$ is upper triangular. $a_{kk}^{(k-1)}$ will be the pivot element for the k^{th} stage. It will be used to eliminate entries below it in column k. If the pivot is zero, then if there exists $a_{\ell k}^{(k-1)} \neq 0$ for $\ell > k$, rows k and ℓ are swapped and $\tilde{A}^{(k-1)}$ is defined to be this new matrix. If no such element exists, then the entries in the column below the pivot are already zero and $A^{(k)} = A^{(k-1)}$. Otherwise, elimination proceeds on the designated elements of $A^{(k-1)}$ or $\tilde{A}^{(k-1)}$ (both of which will be referred to below as $A^{(k-1)}$ to simplify the notation).

In most numerical versions of this algorithm, the general elimination step for the i^{th} row $(i = k+1, \ldots, n)$ is implemented as

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)}, \quad (j = k, \ldots, n). \tag{2.8}$$

This corresponds to pre-multiplying $A^{(k-1)}$ by $E_{ik}(-a_{ik}^{(k-1)}/a_{kk}^{(k-1)})$. In a symbolic environment, (2.8) is not so desirable as it will transform, say, a polynomial matrix into a rational

matrix. Since the determinant must ultimately be polynomial if the matrix entries are initially polynomial, the introduction of rational functions will complicate the situation and may cause much unnecessary computational expense. Rational operations tend to be costly relative to polynomial ones of the same sort, primarily because of GCD (greatest common divisor) calculations, and so it is desirable to avoid rational arithmetic as much as possible in symbolic algorithms. This point will be discussed further in Section 4.2.2.

For the rest of this subsection, suppose A contains only polynomial entries, that is, elements from $\mathbf{Z}[x_1, \ldots, x_v]$. If the matrix R contains rational entries of the form $\mathbf{Z}(x_1, \ldots, x_v)$, it can be converted into a matrix of polynomials by applying the product of elementary matrices $\prod_{i=1}^{n} E_i(d_i)$ to R. If the entries of R are given by p_{ij}/q_{ij}, then $d_i = \mathrm{lcm}(q_{i1}, \ldots, q_{in})$ where lcm designates the least common multiple.

The obvious replacement for (2.8) that will yield a fraction-free algorithm is

$$a_{ij}^{(k)} = a_{kk}^{(k-1)} a_{ij}^{(k-1)} - a_{ik}^{(k-1)} a_{kj}^{(k-1)} \ , \quad (j = k, \ldots, n), \tag{2.9}$$

which corresponds to pre-multiplication by $E_{ik}(-a_{ik}^{(k-1)}) E_i(a_{kk}^{(k-1)})$ for each $i > k$. This formula can be written equivalently as

$$a_{ij}^{(k)} = \begin{vmatrix} a_{kk}^{(k-1)} & a_{kj}^{(k-1)} \\ a_{ik}^{(k-1)} & a_{ij}^{(k-1)} \end{vmatrix} \ , \quad (j = k, \ldots, n),$$

which makes obvious the determinantal relationship among the various elements involved.

The determinants of the final and original matrices will be related by

$$\det T = \left(\prod_{k=1}^{n-1} \left[a_{kk}^{(k-1)} \right]^{n-k} \right) \det A \ .$$

Now, $\det T = \prod_{k=1}^{n} a_{kk}^{(k-1)}$, hence

$$\det A = \frac{a_{nn}^{(n-1)}}{\prod_{k=1}^{n-2} \left[a_{kk}^{(k-1)} \right]^{n-k-1}} \ ,$$

which shows that the entries in the progression of matrices $A^{(k)}$ will grow substantially. Much of this growth can be eliminated, however, by taking advantage of the following observation [Fox65, p. 84].

Consider a general entry $a_{ij}^{(k-1)}$ in the portion of the matrix that will experience elimination as stage k begins, where $i > k+1$. The submatrix in which all the action occurs will

undergo the following transformations [the superscript $(k-1)$ is implied]:

$$\begin{pmatrix} a_{kk} & a_{k,k+1} & \cdots & a_{kj} & \cdots \\ a_{k+1,k} & a_{k+1,k+1} & \cdots & a_{k+1,j} & \cdots \\ \vdots & \vdots & & \vdots & \\ a_{ik} & a_{i,k+1} & \cdots & a_{ij} & \cdots \\ \vdots & \vdots & & \vdots & \end{pmatrix}$$

$$\xrightarrow{k} \begin{pmatrix} a_{kk} & a_{k,k+1} & \cdots & a_{kj} & \cdots \\ 0 & a_{kk}a_{k+1,k+1} - a_{k+1,k}a_{k,k+1} & \cdots & a_{kk}a_{k+1,j} - a_{k+1,k}a_{kj} & \cdots \\ \vdots & \vdots & & \vdots & \\ 0 & a_{kk}a_{i,k+1} - a_{ik}a_{k,k+1} & \cdots & a_{kk}a_{ij} - a_{ik}a_{kj} & \cdots \\ \vdots & \vdots & & \vdots & \end{pmatrix}$$

$$\xrightarrow{k+1} \begin{pmatrix} a_{kk} & a_{k,k+1} & \cdots & a_{kj} & \cdots \\ 0 & a_{kk}a_{k+1,k+1} - a_{k+1,k}a_{k,k+1} & \cdots & a_{kk}a_{k+1,j} - a_{k+1,k}a_{kj} & \cdots \\ \vdots & \vdots & & \vdots & \\ 0 & 0 & \cdots & a_{ij}^{(k+1)} & \cdots \\ \vdots & \vdots & & \vdots & \end{pmatrix}$$

where

$$\begin{aligned} a_{ij}^{(k+1)} &= \left(\boxed{a_{kk}}\, a_{k+1,k+1} - a_{k+1,k}a_{k,k+1}\right)\left(\boxed{a_{kk}}\, a_{ij} - a_{ik}a_{kj}\right) \\ &\quad - \left(\boxed{a_{kk}}\, a_{i,k+1} - a_{ik}a_{k,k+1}\right)\left(\boxed{a_{kk}}\, a_{k+1,j} - a_{k+1,k}a_{kj}\right) \ . \end{aligned}$$

The terms not involving $a_{kk}^{(k-1)}$ cancel, therefore, $a_{kk}^{(k-1)}$ will be a factor of $a_{ij}^{(k+1)}$ for $i, j > k+1$. Hence, a growth minimizing replacement for (2.8) is

$$a_{ij}^{(k)} = \frac{a_{kk}^{(k-1)}a_{ij}^{(k-1)} - a_{ik}^{(k-1)}a_{kj}^{(k-1)}}{a_{k-1,k-1}^{(k-2)}} = \frac{1}{a_{k-1,k-1}^{(k-2)}} \begin{vmatrix} a_{kk}^{(k-1)} & a_{kj}^{(k-1)} \\ a_{ik}^{(k-1)} & a_{ij}^{(k-1)} \end{vmatrix} , \quad (j = k, \ldots, n), \qquad (2.10)$$

where $a_{00}^{(-1)} \equiv 1$ and the division is exact.

The elementary matrix product corresponding to (2.10) is

$$E_i\left(\left[a_{k-1,k-1}^{(k-2)}\right]^{-1}\right) E_{ik}\left(-a_{ik}^{(k-1)}\right) E_i\left(a_{kk}^{(k-1)}\right) \ .$$

At the conclusion of the k^{th} stage, the determinantal relationship will then be

$$\frac{\left[a_{kk}^{(k-1)}\right]^{n-k}\left[a_{k-1,k-1}^{(k-2)}\right]^{n-k+1}\cdots\left[a_{11}^{(0)}\right]^{n-1}\det A}{\left[a_{k-1,k-1}^{(k-2)}\right]^{n-k}\cdots\left[a_{11}^{(0)}\right]^{n-2}\left[a_{00}^{(-1)}\right]^{n-1}} = \begin{vmatrix} a_{11}^{(0)} & \cdots & a_{1k}^{(0)} & a_{1,k+1}^{(0)} & \cdots & a_{1n}^{(0)} \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & a_{kk}^{(k-1)} & a_{k,k+1}^{(k-1)} & \cdots & a_{kn}^{(k-1)} \\ 0 & \cdots & 0 & a_{k+1,k+1}^{(k)} & \cdots & a_{k+1,n}^{(k)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k)} \end{vmatrix}$$

$$= a_{11}^{(0)}\cdots a_{kk}^{(k-1)}\begin{vmatrix} a_{k+1,k+1}^{(k)} & \cdots & a_{k+1,n}^{(k)} \\ \vdots & & \vdots \\ a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k)} \end{vmatrix},$$

and so

$$\det A = \frac{1}{\left[a_{kk}^{(k-1)}\right]^{n-k-1}}\begin{vmatrix} a_{k+1,k+1}^{(k)} & \cdots & a_{k+1,n}^{(k)} \\ \vdots & & \vdots \\ a_{n,k+1}^{(k)} & \cdots & a_{nn}^{(k)} \end{vmatrix}. \tag{2.11}$$

In particular, if $k = n-1$ then

$$\det A = a_{nn}^{(n-1)}\ . \tag{2.12}$$

Thus, when using (2.10) to perform the elimination along the i^{th} row, $\det A$ will be the lower right-hand element of the final resultant triangular matrix T.

This methodology for computing the determinant was first publicized widely in [Bar66, Bar68], so it will be referred to here as single-step fraction-free Bareiss Gaussian elimination (single-step since the method performs one stage at a time). Figure 2.2 presents an implementation of this algorithm, written in pseudo-code. The scope of the various control statements is indicated by the level of indentation. Since the reduction is destructive, the matrix A is first copied into C.

An operation count for the algorithm in Figure 2.2 is given in Table 2.3a. The formulas were derived for a dense matrix in which the **else** branch of the **if** statement was never taken. The actual count analysis of the pseudo-code (which is fairly easy here) is presented below. Future analyses of pseudo-code will follow exactly the same procedures and so their details will be omitted.

If c_{ik} is never zero, then only one statement will be executed which involves arithmetic on the matrix elements. Examining this statement, it is evident that the number of additions, negations and exact divisions will be the same and equal to half the number of multiplications. The total addition count for the algorithm is

$$\sum_{k=1}^{n-1}\sum_{i=k+1}^{n}\sum_{j=k+1}^{n} 1 \;=\; \sum_{k=1}^{n-1}\sum_{i=k+1}^{n} n-k = \sum_{k=1}^{n-1}(n-k)(n-k) = \sum_{k=1}^{n-1} n^2 - 2nk + k^2$$

$C = A$
$d = 1$
for $k = 1$ **to** $n - 1$
  for $i = k + 1$ **to** n
    if $c_{ik} \neq 0$ **then**
      for $j = k + 1$ **to** n
        $c_{ij} = (c_{kk}c_{ij} - c_{ik}c_{kj})/d$
      $c_{ik} = 0$
    else
      for $j = k + 1$ **to** n
        $c_{ij} = c_{kk}c_{ij}/d$
  $d = c_{kk}$
$\det A = c_{nn}$

Figure 2.2 Algorithm to compute the determinant of an $n \times n$ matrix A using single-step fraction-free Bareiss Gaussian elimination.

$+$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$
$-$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$
$\times$	$\frac{2}{3}n^3 - n^2 + \frac{1}{3}n$
exact $\div$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$

a. Single-step Bareiss

$+$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$
$-$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$
$\times$	$\frac{2}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$
exact $\div$	1

b. Single-step sparse

	$m = \left\lfloor \frac{n}{2} \right\rfloor$	$m = \frac{n}{2}$
$+$	$2n^2m - 4nm^2 + \frac{8}{3}m^3 - nm + m^2 + \frac{13}{3}m$	$\frac{1}{3}n^3 - \frac{1}{4}n^2 + \frac{13}{6}n$
$-$	$3nm - 3m^2 + 3m$	$\frac{3}{4}n^2 + \frac{3}{2}n$
$\times$	$3n^2m - 6nm^2 + 4m^3 + 8m$	$\frac{1}{2}n^3 + 4n$
exact $\div$	$n^2m - 2nm^2 + \frac{4}{3}m^3 + nm - m^2 + \frac{11}{3}m$	$\frac{1}{6}n^3 + \frac{1}{4}n^2 + \frac{11}{6}n$

c. Two-step Bareiss

Table 2.3 Rational operation counts for various algorithms which compute the determinant of an $n \times n$ matrix using fraction-free Gaussian elimination methods.

$$
\begin{aligned}
&= n^2(n-1) - 2n\frac{(n-1)n}{2} + \frac{(n-1)(n)(2[n-1]+1)}{6} \\
&= n^2(n-1) - n^2(n-1) + \frac{1}{6}(2n^3 - 3n^2 + n) = \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n \ .
\end{aligned}
$$

Since the calculation of a new value in (2.10) involves only old pivots and elements in the current and k^{th} row and column, the elimination will be completely unaffected by the substitution of the last row and column with a different set of values except in that last row and column (assuming no swapping occurs). Specifically, (2.12) will generalize to

$$
a_{ij}^{(k)} = \begin{vmatrix}
a_{11}^{(0)} & \cdots & a_{1k}^{(0)} & a_{1j}^{(0)} \\
\vdots & & \vdots & \vdots \\
a_{k1}^{(0)} & \cdots & a_{kk}^{(0)} & a_{kj}^{(0)} \\
a_{i1}^{(0)} & \cdots & a_{ik}^{(0)} & a_{ij}^{(0)}
\end{vmatrix} \tag{2.13}
$$

and the process that produced (2.11) will yield

$$
a_{ij}^{(k)} = \frac{1}{\left[a_{\ell\ell}^{(\ell-1)}\right]^{k-\ell}} \begin{vmatrix}
a_{\ell+1,\ell+1}^{(\ell)} & \cdots & a_{\ell+1,k}^{(\ell)} & a_{\ell+1,j}^{(\ell)} \\
\vdots & & \vdots & \vdots \\
a_{k,\ell+1}^{(\ell)} & \cdots & a_{kk}^{(\ell)} & a_{kj}^{(\ell)} \\
a_{i,\ell+1}^{(\ell)} & \cdots & a_{ik}^{(\ell)} & a_{ij}^{(\ell)}
\end{vmatrix} \tag{2.14}
$$

for $\ell \leq k$ [n and k in (2.11) correspond respectively to $k+1$ and ℓ here].

Taking $\ell = k-1$ in (2.14) returns the relationship (2.10). Values of $\ell < k-1$ will result in higher order methods; in particular, $\ell = 0$ gives back (2.13). In each case, since the transformation process is polynomial preserving, $a_{ij}^{(k)}$, the determinant on the right-hand side of (2.14) and the reciprocal of its multiplier must all be polynomials and so the division will always be exact. Thus, for smaller values of ℓ, more stuff can be divided out and hence the size of processed elements in intermediate stages of the elimination will tend to be reduced. Since larger expressions require both more memory and more time to manipulate, this will be a benefit in a CAS.

As a special case, consider $\ell = k-2$. (2.14) then becomes

$$
a_{ij}^{(k)} = \frac{1}{\left[a_{k-2,k-2}^{(k-3)}\right]^2} \begin{vmatrix}
a_{k-1,k-1}^{(k-2)} & a_{k-1,k}^{(k-2)} & a_{k-1,j}^{(k-2)} \\
a_{k,k-1}^{(k-2)} & a_{kk}^{(k-2)} & a_{kj}^{(k-2)} \\
a_{i,k-1}^{(k-2)} & a_{ik}^{(k-2)} & a_{ij}^{(k-2)}
\end{vmatrix} . \tag{2.15}
$$

This will be the basis for a two-step method in which two stages are processed together at a time. Elimination on rows $i > k$ will proceed by (2.15) while elimination on row k will

$C = A$
$d = 1$
for $k = 2$ **to** n [**step** 2]
  if $k < n$ **then**
    $K_0 = (c_{k-1,k-1}c_{kk} - c_{k-1,k}c_{k,k-1})/d$
  for $i = k+1$ **to** n
    $K_1 = (c_{k-1,k}c_{i,k-1} - c_{k-1,k-1}c_{ik})/d$
    $K_2 = (c_{k,k-1}c_{ik} - c_{kk}c_{i,k-1})/d$
    for $j = k+1$ **to** n
      $c_{ij} = (K_0 c_{ij} + K_1 c_{kj} + K_2 c_{k-1,j})/d$
    $c_{i,k-1} = 0$
    $c_{ik} = 0$
  for $j = k$ **to** n
    $c_{kj} = (c_{k-1,k-1}c_{kj} - c_{k-1,j}c_{k,k-1})/d$
  $c_{k,k-1} = 0$
  $d = c_{kk}$
$\det A = c_{nn}$

Figure 2.3 Algorithm to compute the determinant of an $n \times n$ matrix A using two-step fraction-free Bareiss Gaussian elimination.

occur last and use (2.10). Expanding the determinant in (2.15) up the last column yields [suppressing the $(k-2)$ superscript]

$$a_{ij}\begin{vmatrix} a_{k-1,k-1} & a_{k-1,k} \\ a_{k,k-1} & a_{kk} \end{vmatrix} - a_{kj}\begin{vmatrix} a_{k-1,k-1} & a_{k-1,k} \\ a_{i,k-1} & a_{ik} \end{vmatrix} + a_{k-1,j}\begin{vmatrix} a_{k,k-1} & a_{kk} \\ a_{i,k-1} & a_{ik} \end{vmatrix} .$$

Note that the first cofactor is independent of i and j, while the other two are independent of j.

Figure 2.3 presents a pseudo-code implementation of "two-step fraction-free Bareiss Gaussian elimination" using the above two formulas, where the divisions have been spread out in order to minimize the size of the expressions involved as quickly as possible. An operation count for this algorithm is given in Table 2.3c in which an exact count is presented in the second column and an approximate count (which is exact for n even) is displayed in the third for easy comparisons with other tallies. All of the expensive (i.e., binary) operations are of the same order or less than those for the single-step method.

In practice, single-step Bareiss procedures are commonly implemented in CASs, while two-step procedures are rarely done (the only one known to the author is the `det` function in MACSYMA 415). The usual judgement seems to be that the advantages of the two-step algorithms are not worth the additional complexity needed to implement them. It is not so clear whether this attitude is justifiable.

```
C = A
d = 1
for k = 1 to n − 1
    for i = k + 1 to n
        if c_ik ≠ 0 then
            d = d c_kk
            for j = k + 1 to n
                c_ij = c_kk c_ij − c_ik c_kj
            c_ik = 0
D = 1
for i = 1 to n
    D = D c_ii
det A = D/d
```

Figure 2.4 Algorithm to compute the determinant of an $n \times n$ matrix A using single-step fraction-free sparse Gaussian elimination.

The major problem with the Bareiss elimination methods is in how they handle sparse matrices. In traditional numerical Gaussian elimination exemplified by equation (2.8), anytime there is already a zero at position (i, k), the elimination on row i can be skipped for that particular stage [(2.8) becomes $a_{ij}^{(k)} = a_{ij}^{(k-1)}$]. However, although the elimination will be simplified in such a case for the Bareiss methods, it cannot be omitted entirely without destroying the exact divisibility properties implied in equation (2.14).

A solution is to go back to using (2.9), but only applying it on rows where $a_{ik}^{(k-1)} \neq 0$. In this case, it will be necessary to accumulate the product of the determinants of the elementary operations separately. After the elimination has been completed, this value will then need to be divided (exactly) out of the product of the diagonal elements of the final matrix T. This final division will generally involve the largest operands in the entire process and so can be expensive to perform. Nonetheless, in many cases, especially if implemented carefully, this "single-step fraction-free sparse Gaussian elimination" procedure can result in a great speed-up over the normal Bareiss methods.

One implementation of the above algorithm in pseudo-code is displayed in Figure 2.4 and the corresponding operation count in Table 2.3b. As can be seen by comparing with Table 2.3a, the two single-step methods treat dense matrices about the same except that the sparse procedure accrues all of its divisions into a single large one at the end.

Actual implementations of the two Bareiss and the above sparse method in a CAS are presented in Appendix F using the MAPLE language. The coding makes explicit certain techniques necessary to gain an efficient rendering of these schemes. In particular, partial pivoting is performed on the rows of the matrix at the beginning of each stage in order to choose the "best" pivot (i.e., the one with the "simplest" structure—this term is defined

precisely in Section 4.2.3) for the subsequent eliminations. Also, since the physical swapping of the rows in a matrix requires a large number of memory accesses and stores, it is substantially cheaper (and not much harder to design) using a scheme in which swapping is done implicitly through indexing arrays.

The concept of indexing arrays as a technique to avoid memory shuffling[1] by implying permutations is very useful. To consider this idea in a general context, let P_{R_k} be the product of k elementary row permutations of the form P_{ij} where now, $i = j$ is allowable (which will produce an identity matrix): $P_{R_k} \equiv P_{i_k j_k} \cdots P_{i_1 j_1}$. In a like manner, define P_{C_k} to be the sequence of k elementary column permutations $P_{\ell_1 m_1} \cdots P_{\ell_k m_k}$. If $B = P_{R_k} A P_{C_k}$ for some $n \times n$ matrix A, the question becomes: how are the elements of A reordered in B?

To answer this question, suppose that the vectors $\mathbf{I}_0$ and $\mathbf{J}_0$ are set to the identity permutation $[(\mathbf{I}_0)_i = (\mathbf{J}_0)_i = i,\ (i = 1, \ldots, n)]$. Now, let the vectors $\mathbf{I}_s$, $\mathbf{J}_s$ be recursively defined by $\mathbf{I}_s = P_{i_s j_s} \mathbf{I}_{s-1}$ and $\mathbf{J}_s^T = \mathbf{J}_{s-1}^T P_{\ell_s m_s}$ so that

$$\mathbf{I}_k = P_{R_k} \mathbf{I}_0 \quad \text{and} \quad \mathbf{J}_k^T = \mathbf{J}_0^T P_{C_k} \ . \tag{2.16}$$

Note that every application of an elementary permutation matrix to the vector $\mathbf{I}_s$ or $\mathbf{J}_s$ simply results in two elements being swapped.

An elementary permutation matrix, like P_{ij}, has exactly one 1 in each row and in each column; hence a product of these matrices, such as P_{R_k} (which will be a composite permutation matrix), will also have this property. The ones in P_{R_k} can be described by a list of their positions. If this list is ordered by the row coordinates, then the sequence of column coordinates will be precisely the contents of $\mathbf{I}_k$, which is equivalent to saying how P_{R_k} affects the identity permutation (given in $\mathbf{I}_0$). Therefore, $\mathbf{I}_k$ will describe exactly how the rows of A will be rearranged by P_{R_k}. Similarly, $\mathbf{J}_k$ will describe how the columns of A are shuffled by P_{C_k}.

All the above remarks lead up to the observation that

$$b_{ij} = a_{(\mathbf{I}_k)_i (\mathbf{J}_k)_j} \ , \quad (i, j = 1, \ldots, n). \tag{2.17}$$

This says that the (i, j)-element of B is equal to the element of A with row index $(\mathbf{I}_k)_i$ (i.e., the i^{th} element of $\mathbf{I}_k$) and column index $(\mathbf{J}_k)_j$.[2] A more compact notation (see the footnote) that will be used to mean the same thing is

$$B = A_{\mathbf{I}_k \mathbf{J}_k} \ , \tag{2.18}$$

[1] The idea represented by this phrase is so common as to warrant its own term: blit (from the PDP-10 block transfer instruction BLT) [Ste87]. The equivalent word for files is mung (frequently used in a malicious sense).

[2] In the matrix manipulation program MATLAB [Mol80], (2.17) would be expressed as

```
for i = 1:n, ...
   for j = 1:n, ...
      B(i,j) = A(Ik(i),Jk(j)); ...
      end; ...
   end;
```

or more compactly by `B = A(Ik,Jk)`.

where it will be made clear that the subscripts are n-element vectors.

The inverse relationship can also be described in a simple manner. A useful fact of elementary permutations P_{ij} is that $P_{ij}^T = P_{ij} = P_{ij}^{-1}$, where the last equality can be deduced by noting that a second exchange of the same sort will undo the effects of the first. Thus, a composite permutation like P_{R_k} will have as its inverse

$$\begin{aligned} P_{R_k}^{-1} &= (P_{i_k j_k} \cdots P_{i_1 j_1})^{-1} = P_{i_1 j_1}^{-1} \cdots P_{i_k j_k}^{-1} \\ &= P_{i_1 j_1}^T \cdots P_{i_k j_k}^T = (P_{i_k j_k} \cdots P_{i_1 j_1})^T = P_{R_k}^T \end{aligned}$$

(P_{R_k} need not be equal to $P_{R_k}^T$, however). This is equivalent to listing the positions of the ones in P_{R_k}, as before, but this time ordering by column coordinates. The sequence of row coordinates here will determine the inverse mapping $\mathbf{I}_k^{-1}$ (which can also be defined by $\mathbf{I}_k^{-1} = P_{R_k}^{-1}\mathbf{I}_0 = P_{R_k}^T\mathbf{I}_0$). $\mathbf{I}_k^{-1}$ is termed an inverse mapping since $(\mathbf{I}_k^{-1})_{(\mathbf{I}_k)_i} = i$ [or $\mathbf{I}_k^{-1}(\mathbf{I}_k(i)) = i$] and the complementary identity both hold. Therefore, as $A = P_{R_k}^{-1} B P_{C_k}^{-1} = P_{R_k}^T B P_{C_k}^T$ then

$$A = B_{\mathbf{I}_k^{-1}\mathbf{J}_k^{-1}} \ , \tag{2.19}$$

where $(\mathbf{J}_k^{-1})^T = \mathbf{J}_0^T P_{C_k}^T$ or $\mathbf{J}_k^{-1} = P_{C_k}\mathbf{J}_0$.

(2.18) and (2.19) are perfectly general formulas since k is arbitrary and an indefinite number of the elementary matrices composing either P_{R_k} or P_{C_k} can be identity matrices. An actual implementation, as in Appendix F, will represent the various vectors involved by one-dimensional arrays.

Appendix F exhibits a very typical situation in which permutations alternate with elimination type transformations. At the beginning of each stage of the reduction, the indexing array is updated to reflect any permutations that need to be performed and then the reduction proceeds using this array to index on the row. For the general case, it may also be necessary to index on the column.

To make these ideas precise, consider a sequence of transformations $A^{(1)}, \ldots, A^{(m)}$ to a matrix A $(= A^{(0)})$. At each stage in the transformation, both row and column permutations may occur. Following these will be the elimination steps which can, in general, also operate on both the rows and the columns of the matrix. Let the row and column permutation matrices applied at the k^{th} stage be denoted by P_{r_k} and P_{c_k}, respectively. Also, let X_k and Y_k designate the matrices that correspond to the row and column eliminations. The permutations will consist, in general, of one or more simple permutations, while X_k and Y_k will typically be products of elementary matrices. Thus, $A^{(k)}$ will be defined in terms of $A^{(k-1)}$ by

$$A^{(k)} = (X_k P_{r_k}) A^{(k-1)} (P_{c_k} Y_k) \ ,$$

and therefore, in terms of the original matrix A by

$$A^{(k)} = (X_k P_{r_k} \cdots X_1 P_{r_1}) A (P_{c_1} Y_1 \cdots P_{c_k} Y_k) \ . \tag{2.20}$$

Consider these transformations now in the context of indexing arrays. Let $\mathbf{I}_k$ and $\mathbf{J}_k$ hold the contents of arrays respectively indexing rows and columns for the k^{th} stage. $\mathbf{I}_0$ and $\mathbf{J}_0$

will be initialized to the identity permutation and the arrays (treating them as vectors) will be updated as

$$\mathbf{I}_k = P_{r_k}\mathbf{I}_{k-1} \quad \text{and} \quad \mathbf{J}_k^T = \mathbf{J}_{k-1}^T P_{c_k} \ .$$

(The second equation can also be written $\mathbf{J}_k = P_{c_k}^T \mathbf{J}_{k-1}$.) Hence,

$$\mathbf{I}_k = P_{r_k} \cdots P_{r_1}\mathbf{I}_0 \quad \text{and} \quad \mathbf{J}_k^T = \mathbf{J}_0^T P_{c_1} \cdots P_{c_k} \tag{2.21}$$

[compare with (2.16)]. The general elimination step that follows next will be of the form

$$\hat{A}^{(k)}_{\mathbf{I}_k\mathbf{J}_k} = X_k \hat{A}^{(k-1)}_{\mathbf{I}_k\mathbf{J}_k} Y_k \ , \tag{2.22}$$

where $\hat{A}^{(0)} \equiv A$. $\hat{A}^{(k-1)}$ is a physical, two-dimensional array that is being modified.

Expanding (2.22) using (2.21), this relationship becomes

$$(P_{r_k} \cdots P_{r_1})\hat{A}^{(k)}(P_{c_1} \cdots P_{c_k}) = X_k(P_{r_k} \cdots P_{r_1})\hat{A}^{(k-1)}(P_{c_1} \cdots P_{c_k})Y_k$$

or

$$\hat{A}^{(k)} = (P_{r_1}^T \cdots P_{r_k}^T)[X_k(P_{r_k} \cdots P_{r_1})\hat{A}^{(k-1)}(P_{c_1} \cdots P_{c_k})Y_k](P_{c_k}^T \cdots P_{C_1}^T) \ . \tag{2.23}$$

Setting $k = 1$ in (2.23),

$$\hat{A}^{(1)} = P_{r_1}^T(X_1 P_{r_1})A(P_{c_1}Y_1)P_{c_1}^T \ .$$

This will generalize to

$$\begin{aligned} \hat{A}^{(k)} &= P_{r_1}^T \cdots P_{r_k}^T(X_k P_{r_k} \cdots X_1 P_{r_1})A(P_{c_1}Y_1 \cdots P_{c_k}Y_k)P_{c_k}^T \cdots P_{c_1}^T \\ &= (P_{r_1}^T \cdots P_{r_k}^T)A^{(k)}(P_{c_k}^T \cdots P_{c_1}^T) \end{aligned} \tag{2.24}$$

as can be verified by inducting on (2.23) while making use of (2.24):

$$\begin{aligned} \hat{A}^{(k+1)} &= (P_{r_1}^T \cdots P_{r_{k+1}}^T)X_{k+1}(P_{r_{k+1}}P_{r_k} \cdots P_{r_1})\Big[P_{r_1}^T \cdots P_{r_k}^T(X_k P_{r_k} \cdots X_1 P_{r_1})A \\ &\quad \times (P_{c_1}Y_1 \cdots P_{c_k}Y_k)P_{c_k}^T \cdots P_{c_1}^T\Big](P_{c_1} \cdots P_{c_k}P_{c_{k+1}})Y_{k+1}(P_{c_{k+1}}^T \cdots P_{c_1}^T) \\ &= P_{r_1}^T \cdots P_{r_{k+1}}^T(X_{k+1}P_{r_{k+1}}X_k P_{r_k} \cdots X_1 P_{r_1})A(P_{c_1}Y_1 \cdots P_{c_k}Y_k P_{c_{k+1}}Y_{k+1})P_{c_{k+1}}^T \cdots P_{c_1}^T \ . \end{aligned}$$

In the Bareiss style algorithms presented here, no column operations are involved so $P_{c_k} = Y_k = I$ for $k = 1, \ldots, n-1$. Hence,

$$\hat{A}^{(n-1)} = P_{r_1}^T \cdots P_{r_{n-1}}^T(X_{n-1}P_{r_{n-1}} \cdots X_1 P_{r_1})A = P_{r_1}^T \cdots P_{r_{n-1}}^T A^{(n-1)} \ .$$

The final triangular matrix $T = A^{(n-1)}$ is then extracted from $\hat{A}^{(n-1)}$ by

$$T = P_{r_{n-1}} \cdots P_{r_1}\hat{A}^{(n-1)} \ ,$$

which is equivalent to the indexing operation $T = A^{(n-1)}_{\mathbf{I}_{n-1},\bullet}$ [or $T(i,j) = \hat{A}^{(n-1)}(\mathbf{I}_{n-1}(i), j)$]. A new notation is introduced here in which a dot is used as a placeholder to indicate an identity operation for matrices in which only one subscript is indexed.

One further comment needs to be made about the coding in Appendix F concerning the sparse Gaussian elimination algorithm. This implementation follows the method of the pseudo-code presented in Figure 2.4 except for the last four lines, where instead, an alternative procedure based on

$d = 1/d$
for $i = 1$ **to** n
    $d = dc_{ii}$
$\det A = d$

is followed. As before, d contains the accumulated factors from the elimination that need to be divided out of $\det T$. Although rational arithmetic is introduced here, it will be more than compensated by the reduced computational effort expended in performing several smaller exact divisions rather than one massive one.

2.1.3 Modular Algorithms

Modular algorithms are a variant, but an important one, of the polynomial preserving procedures discussed in the previous subsections. Instead of directly working with multivariate polynomials with integer coefficients, for example, the entries in a matrix are converted to representations in some number of independent, simpler domains (such as finite fields with different moduli). The computation is performed in each of these simpler domains and then the solution in the original domain is constructed from the results obtained. The key to this method lies in the simplicity of the arithmetic that occurs in the simpler domains.

To examine these ideas concretely, first consider the problem of finding the determinant of an $n \times n$ matrix A filled with integers. In all the integer preserving methods so far discussed, very large integers can be produced during the intermediate stages as well as for the final result. Manipulation of large integers can be expensive, however, especially if the integers are larger than the word size of the computer and so must be represented using data structures such as linked lists. The time consumed in the manipulation will be proportional to the size of the integers involved.

Integer growth can be limited by performing all arithmetical operations in a finite domain. $\mathbf{Z}_m$ (the integers modulo m with $m > 1 \in \mathbf{Z}$) is such a domain, defined in terms of the equivalence relation $\equiv$ (congruence). The elements of this ring consist of the sequence of integers $\{c, \ldots, c+m-1\}$, where the integer c can be chosen arbitrarily, forming a complete residue system. Any $b \in \mathbf{Z}$ corresponds to an element a of $\mathbf{Z}_m$ by

$$a \equiv b \pmod{m},$$

which is equivalent to saying that $b = a + km$ for some integer k, where $c \leq a < c + m$. The two most common choices for c are zero, which produces the positive representation $\{0, \ldots, m-1\}$ of $\mathbf{Z}_m$, and if m is odd, $c = -(m-1)/2$ which generates the symmetric representation of $\mathbf{Z}_m$

$$\left\{-\frac{m-1}{2}, \ldots, -1, 0, 1, \ldots, \frac{m-1}{2}\right\} .$$

$a \in \mathbf{Z}_m$ will have a unique multiplicative inverse, a^{-1}, if and only if $\gcd(a, m) = 1$ [Sta70, Chapter 3]. Now, if m is a prime number p, then $\gcd(a, p) = 1$ for all $a \in \mathbf{Z}_p$ except for $a \equiv 0 \pmod{p}$. Therefore, $\mathbf{Z}_p$ will form a finite field and if $p > 2$, can be described in symmetric representation by $|a| \leq \frac{p-1}{2}$ for $a \in \mathbf{Z}_p$.

All rational arithmetic operations available in $\mathbf{Z}$ (additions, subtractions, multiplications, divisions) are also available in $\mathbf{Z}_p$, and so all the algorithms discussed in the previous subsections can be performed equally well in this finite field. (Actually, if only exact divisions are required rather than full ones, it is sufficient to operate just in an integral domain, but $\mathbf{Z}_m$ for non-prime m is only a commutative ring and thus has inadequate algebraic structure.) Phrased differently, all the determinant procedures discussed previously can have their arithmetic performed (mod p) when applied to integer matrices. Of course, the final results will also be (mod p).

Arithmetic (mod p) is particularly nice since the reduction (mod p) can be performed at any time and as often as necessary. This observation comes from the easily proved statement

$$a \,\square\, b \equiv [a \,(\mathrm{mod}\ p)\ \square\ b \,(\mathrm{mod}\ p)] \pmod{p}$$

(and its variants) where $\square$ is one of $+$, $-$ or $\times$. Hence, the only time a number need have a magnitude $> \frac{p-1}{2}$ is the temporary result of one of the above binary operations before it is reduced back into its symmetric range. (The symmetric representation of $\mathbf{Z}_p$ is often preferable since it mirrors the symmetry of the integers; however, many people are just as happy to use the positive representation [How69].)

Moreover, $a^{-1} \pmod{p}$ is easily computable. The extended Euclidean algorithm can be used to find integers s and t such that $sa + tb = \gcd(a, b)$. If b is replaced by p and $a \not\equiv 0 \pmod{p}$, then this equation becomes

$$sa + tp = \gcd(a, p) = 1 \quad \text{or} \quad sa \equiv 1 \pmod{p}.$$

Thus, $a^{-1} \equiv s \pmod{p}$. Alternatively, by the Little Fermat Theorem [Sta70, p. 80], $a^{p-1} \equiv 1 \pmod{p}$, therefore $a^{-1} \equiv a^{p-2} \pmod{p}$. Either of these two methods is very straightforward to implement and relatively fast if organized with a little care.

Following [Ged82, Chapter 5], suppose $r_i \equiv \det A \pmod{m_i}$ has been computed for a series of distinct, odd prime moduli m_i ($i = 0, \ldots, N$ for some yet to be defined N). The question then becomes how to construct $d = \det A$ from the r_i. The answer lies in expressing d in a mixed radix representation, rather than in the usual single radix representation given by (for $0 \leq d \leq \beta^{N+1} - 1$)

$$d = d'_0 + d'_1\beta + d'_2\beta^2 + \cdots + d'_N\beta^N = \sum_{k=0}^{N} d'_k\beta^k \ . \tag{2.25}$$

Here, $\beta > 1 \in \mathbf{Z}$ is typically taken to be 10 or a power of 2, and $d'_k \in \mathbf{Z}_\beta$ with $\mathbf{Z}_\beta$ expressed in its positive representation. In the mixed radix representation, powers of β are replaced by products of radices yielding

$$d = d_0 + d_1(m_0) + d_2(m_0m_1) + \cdots + d_N \left(\prod_{i=0}^{N-1} m_i \right) = \sum_{k=0}^{N} d_k \left(\prod_{i=0}^{k-1} m_i \right) \ , \tag{2.26}$$

where $d_k \in \mathbf{Z}_{m_k}$. Letting $m_i = \beta$ for each i (and $d_k = d'_k$ for each k) will transform this equation back into (2.25).

The coefficients d_k in (2.26) are discovered sequentially from the congruences defining the r_i. Taking (2.26) (mod m_0) produces $d_0 \equiv d \equiv r_0 \pmod{m_0}$, which implies that $d_0 = r_0$ since $d_0, r_0 \in \mathbf{Z}_{m_0}$ and residues are unique within a given representation. In general, taking (2.26) (mod m_k) yields

$$d_k \equiv \left[r_k - \sum_{j=0}^{k-1} d_j \left(\prod_{i=0}^{j-1} m_i \right) \right] \left(\prod_{i=0}^{k-1} m_i \right)^{-1} \pmod{m_k} \tag{2.27}$$

for $k = 1, \ldots, N$. The inverse appearing here exists due to the supposition that the moduli are distinct primes so that $\prod_{i=0}^{k-1} m_i$ will be relatively prime to m_k.

The value of d constructed from (2.26) and (2.27) (by a process commonly known as the Chinese Remainder Theorem) will depend on the representations used by the $\mathbf{Z}_{m_k}$'s. If all the image fields are expressed in a positive representation, then $0 \leq d \leq M - 1$, where $M = \prod_{i=0}^{N} m_i$. That is, $d \in \mathbf{Z}_M$, where the positive representation of $\mathbf{Z}_M$ is used. This can be seen by choosing the $d_k = 0$ in (2.26) to obtain the lower bound. Letting $d_k = m_k - 1$, $(k = 0, \ldots, N)$ will result in a telescoping sum which will yield the upper limit. Similarly, if all the $\mathbf{Z}_{m_k}$'s exist in a symmetric representation, then $d \in \mathbf{Z}_M$ will satisfy $|d| \leq \frac{M-1}{2}$.

Now, if the elements of A originally lay in $\mathbf{Z}$ (the usual case), then for M chosen sufficiently large, the $d \in \mathbf{Z}_M$ (in symmetric representation) computed above will also be the representation of d in $\mathbf{Z}$. Thus, the moduli and their number $N + 1$ should be selected so that $\frac{M-1}{2}$ bounds $|\det A|$. This can be done in one of two ways. An explicit upper bound can be calculated, such as that given by Hadamard's inequality [Knu81, p. 414]:

$$|\det A| \leq \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij}^2 \right)^{1/2} . \tag{2.28}$$

This quantity can be computed easily. The problem here is that the bound computed may be unduly conservative. The alternative is to incrementally build up d one modulus at a time by alternatively calculating r_k and d_k, $(k = 0, 1, \ldots)$ until an N is discovered such that $d_k = 0$ for all $k > N$.

This latter methodology is often practiced by stopping at the first zero coefficient and verifying if the result is complete using a substitution check. If the moduli are fairly large, it is likely that the first zero coefficient will directly presage the end of the computation. A good choice for m_0 is the largest prime that will fit in a computer word (reserving one bit for the sign). Subsequent m_i's $(i \geq 1)$ can then be the next smaller primes, taken in descending order. One disadvantage of the incremental algorithm is that it is not as well suited to parallel machines as is the method that uses the Chinese Remainder Theorem only once. However, this can be remedied by computing the residues for a set of moduli at once rather than one at a time.

In an entirely analogous manner, a matrix of multivariate polynomials from $\mathbf{Z}_p[x_1, \ldots, x_v]$ can be transformed into a number of objects with entries in the simpler domain $\mathbf{Z}_p$, where it is

much easier to perform the arithmetic. The collection of values generated from application of determinant algorithms can then be assembled into a solution in the original domain by an interpolation procedure that is abstractly equivalent to the method underlying the Chinese Remainder Theorem. Therefore, in the typical case where the matrix A contains elements from $\mathbf{Z}[x_1, \ldots, x_v]$, A can first be transformed into $N+1$ matrices with entries from $\mathbf{Z}_{m_i}[x_1, \ldots, x_v]$, $i = 0, \ldots, N$ [the coefficient of each term in a given polynomial is reduced (mod m_i)], each of which can then be evaluated at a fixed number of points for each variable to be eliminated. Determinants can next be computed directly in the simplified domains and finally, the results uplifted by a pair of corresponding procedures into a solution back in the original domain.

As touched upon above, the number of variables in a polynomial can be reduced by simply choosing a set of evaluation points for each variable that is desired to be eliminated. Selecting sufficiently many will guarantee that the original polynomial can be uniquely reconstructed by interpolation. For instance, suppose that the highest degree of x_i in $f \in \mathbf{Z}_p[x_1, \ldots, x_v]$ is D_i. Choose $D_v + 1$ distinct values $\{\alpha_0, \ldots, \alpha_{D_v}\}$ and form $g_i(x_1, \ldots, x_{v-1}) = f(x_1, \ldots, x_{v-1}, \alpha_i)$, $i = 0, \ldots, D_v$. f can then be regenerated from the g_i's by a procedure known as the Newton interpolation algorithm.

For simplicity, consider the situation when $v = 1$ so that $f = f(x)$ and $g_i = f(\alpha_i)$, $i = 0, \ldots, D$ (dropping superfluous subscripts). Now, since the α_i are distinct, the binomials $x - \alpha_i$ will be pairwise relatively prime [i.e., $\gcd(x - \alpha_i, x - \alpha_j) = 1$ for $i \neq j$]. This is reminiscent of the prime moduli m_i in the integer case above which also satisfied this condition. In a like manner, f can be expanded in terms of these binomials by

$$\begin{aligned} f(x) &= f_0 + f_1(x - \alpha_0) + f_2(x - \alpha_0)(x - \alpha_1) + \cdots + f_D \left[\prod_{i=0}^{D-1} (x - \alpha_i) \right] \\ &= \sum_{k=0}^{D} f_k \left[\prod_{i=0}^{k-1} (x - \alpha_i) \right] . \end{aligned} \tag{2.29}$$

This mixed radix representation of f is called its Newton divided-difference form. (A single radix representation would be expressed as a finite Taylor series.)

To obtain the coefficients f_i, it is useful to make the notational convention that the evaluation homomorphism $x \to \alpha_i$ can be represented by congruence (mod $x - \alpha_i$). Thus, $g_i \equiv f(x) \pmod{x - \alpha_i}$, $i = 0, \ldots, D$ defines the residue of f at each evaluation point α_i. Taking (2.29) (mod $x - \alpha_0$) gives $f_0 \equiv f(x) \equiv g_0 \pmod{x - \alpha_0}$, which implies that $f_0 = g_0$ since both quantities are independent of x. In the general case, (2.29) (mod $x - \alpha_k$) produces (remembering that the coefficients of $f(x)$ and hence the α_i are contained in $\mathbf{Z}_p$)

$$f_k = \left[g_k - \sum_{j=0}^{k-1} f_j \left\{ \prod_{i=0}^{j-1} (\alpha_k - \alpha_i) \right\} \right] \left[\prod_{i=0}^{k-1} (\alpha_k - \alpha_i) \right]^{-1} \tag{2.30}$$

for $k = 1, \ldots, D$ where the inverse is taken in $\mathbf{Z}_p$. The above result is really a congruence (mod $x - \alpha_k$), but it becomes an equality since it is independent of x [of course, the f_k's are still only determined (mod p)].

There is a clear correspondence between (2.26)–(2.27) and (2.29)–(2.30). Essentially, the two pairs of equations only differ in the names of their variables with the most notable change being that the modulus m_i has been replaced by the basis element $x - \alpha_i$ in (2.29) and by $\alpha_k - \alpha_i$ in (2.30). Continuing the analogy, the coefficients f_k can be computed either incrementally along with the residues g_k or all at once after sufficient number of g_k have been determined. The latter requires knowledge of a bound on the final degree of the answer. One such bound is [McC73, p. 570]

$$\deg(\det A) \leq \sum_{j=1}^{n} \max_{1 \leq i \leq n} \deg(a_{ij}) \ ,$$

where $\deg(f(x))$ is the highest degree of x present in $f(x)$.

The essential idea of modular algorithms is that some operations are easier to perform on one representation of an expression than on another. It is easy to do arithmetic on residues and residues themselves are easy to compute from the standard, single radix representations of polynomials and integers (which have radices, respectively, of $x_1, \ldots, x_v$ and 10). It is more difficult to go in the other direction. To do so requires the construction of a mixed radix, intermediate form which is then expanded to obtain the standard representation.

Although conversions between various representations can be nontrivial, they are typically responsible only for a small amount of the time spent in a computation. The vast majority of the time is consumed by the calculation of the determinant on all the various images. If $N+1$ moduli are needed to represent the coefficients of $\det A$, and if each indeterminant x_i, $(i = 1, \ldots, v)$ in the final result has degree of at most D_i, then the minimal number of images of A necessary will be $(N+1)[\prod_{i=1}^{v}(D_i + 1)]$, which for $D_i = D$, $(i = 1, \ldots, v)$ becomes $(N+1)(D+1)^v$. Hence, the number of images will grow exponentially with respect to the number of indeterminants. Despite this, modular algorithms can be fast if reasonable methods are used to compute the determinants of the image matrices, such as some of the procedures that have been discussed previously.

2.2 Characteristic Polynomial Algorithms

In the great majority of present day CASs, the characteristic polynomial of a matrix A, $c_A(\lambda)$, is computed directly from its definition (2.1). Thus, the characteristic matrix $A - \lambda I$ is formed and then one of the determinant algorithms from the previous section is applied. There are methods, however, that can be used to compute $c_A(\lambda)$ that do not depend directly on determinant procedures and that is what will be discussed in this section. To set up the problem precisely, suppose A is dimensioned $n \times n$. The goal now is to find the polynomial

$$c_A(\lambda) = \lambda^n + a_{n-1}\lambda^{n-1} + \cdots + a_1\lambda + a_0$$

whose roots are the eigenvalues of A. In practice, a multiple of $c_A(\lambda)$ will do equally well since scalar multiplication will not affect the roots, which are the ultimate objective, after all, in most situations. [For instance, note that the above form of $c_A(\lambda)$ is actually computed from $\det(\lambda I - A) = (-1)^n \det(A - \lambda I)$.]

2.2.1 Method of Leverrier

One method for computing the coefficients of the characteristic polynomial involves combining traces of powers of the matrix. The trace of the matrix A is defined by

$$\operatorname{tr} A = \sum_{i=1}^{n} a_{ii}$$

(i.e., the sum of the diagonal elements). If $c_A(\lambda)$ is written in factored form in terms of its roots $\{\lambda_i\}_{i=1}^n$ (which are the eigenvalues of A) and then expanded:

$$\begin{aligned} c_A(\lambda) &= (\lambda-\lambda_1)(\lambda-\lambda_2)\cdots(\lambda-\lambda_n) \\ &= \lambda^n - (\lambda_1+\lambda_2+\cdots+\lambda_n)\lambda^{n-1} + \cdots + (-1)^n\lambda_1\lambda_2\cdots\lambda_n \ , \end{aligned}$$

it can be seen that $a_{n-1} = -\sum_{i=1}^n \lambda_i$ [and also that $\det A = a_0 = (-1)^n \prod_{i=1}^n \lambda_i$]. By using simple properties of traces, it can be shown that $\operatorname{tr} A = \sum_{i=1}^n \lambda_i$ [Gan60, p. 87] and thus $a_{n-1} = -\operatorname{tr} A$. Also, the eigenvalues of A^k, $(k = 0, 1, \ldots)$ will be given by $\{\lambda_i^k\}_{i=1}^n$ and therefore

$$\operatorname{tr} A^k = \sum_{i=1}^{n} \lambda_i^k \ , \quad (k = 0, 1, \ldots).$$

The coefficients of $c_A(\lambda)$ can be obtained from the symmetric functions of the above quantities for $k = 1, \ldots, n$. This procedure produces the recursion formula

$$a_{n-k} = -\frac{1}{k}\left[\operatorname{tr} A^k + \sum_{i=1}^{k-1} a_{n-i} \operatorname{tr} A^{k-i}\right] \ , \quad (k = 1, \ldots, n). \tag{2.31}$$

[(2.31) can also be derived using differential forms by considering the characteristic equation of the finite dimensional operator whose matrix representation is A [Sto72].] The above recurrence relation can be formulated as a matrix equation which makes the relationships involved particularly clear:

$$\begin{pmatrix} a_{n-1} \\ 2a_{n-2} \\ 3a_{n-3} \\ \vdots \\ (n-1)a_1 \\ na_0 \end{pmatrix} = - \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ a_{n-1} & 1 & 0 & 0 & \cdots & 0 \\ a_{n-2} & a_{n-1} & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ a_2 & a_3 & \cdots & a_{n-1} & 1 & 0 \\ a_1 & a_2 & a_3 & \cdots & a_{n-1} & 1 \end{pmatrix} \begin{pmatrix} \operatorname{tr} A \\ \operatorname{tr} A^2 \\ \operatorname{tr} A^3 \\ \vdots \\ \operatorname{tr} A^{n-1} \\ \operatorname{tr} A^n \end{pmatrix} . \tag{2.32}$$

If A is purely numerical, then the calculation of the coefficients $\{a_i\}_{i=0}^{n-1}$ will involve no polynomial arithmetic, which is a clear advantage over the direct use of the definition (2.1) which does involve polynomial arithmetic. However, the operation counts for Leverrier's method, listed in Table 2.4, show that it is not a very efficient procedure and despite claims to the contrary in the MACSYMA manual [Com88] (in particular, the so-called "fast characteristic polynomial algorithm" ncharpoly), this method is reasonable to use only for very

$+$	$n^4 - 2n^3 + \frac{5}{2}n^2 - \frac{3}{2}n$
$-$	n
$\times$	$n^4 - n^3 + \frac{1}{2}n^2 - \frac{1}{2}n$
$\div$	$n - 1$

Table 2.4 Rational operation counts for computing the characteristic polynomial of an $n \times n$ matrix using Leverrier's method.

small matrices (see also Chapter 5). The counts in the table were derived by noting that n^3 multiplications and $n^2(n-1)$ additions are needed to form the product of two $n \times n$ matrices. Therefore, computing $A, A^2, \ldots, A^n$ will require $n^3(n-1)$ multiplications and $n^2(n-1)^2$ additions, and taking the traces of these matrices demands $n(n-1)$ more additions. The rest of the analysis follows previously established methods.

2.2.2 Hessenberg Method

Here, it becomes useful to consider the notion of similarity. Two $n \times n$ matrices, A and B, are similar if there exists a nonsingular matrix S such that $B = S^{-1}AS$. The $n \times n$ matrices S and S^{-1} are often referred to as the similarity matrices corresponding to this transformation. Similarity is important because the invariants of the transformation include the major matrix concepts discussed previously: the trace, the determinant, the characteristic polynomial and the eigenvalues. (There are others but these are the most prominent.) Thus, if B is similar to A, then $\operatorname{tr} B = \operatorname{tr} A$, $\det B = \det A$ and $c_B(\lambda) = c_A(\lambda)$, which implies that the eigenvalues of A and B are the same (the eigenvectors are not, unfortunately).

If all the eigenvalues and (generalized) eigenvectors of a matrix are known, the matrix can be similarily transformed into Jordan canonical form (in symbols, $J = S^{-1}AS$ where J is the Jordan form of A). The Jordan form is characterized by a particularly simple structure in which all its eigenvalues (and thus the eigenvalues of the original matrix) lie along the diagonal and all other entries are zero, except perhaps along the superdiagonal (or alternatively, the subdiagonal) where ones may appear. The trouble is that although it is always theoretically possible to transform a matrix into Jordan form, in practice the procedure can be computationally intensive and the intermediate results quite messy for any but the simplest and smallest of matrices.

An alternative representation that is easier to compute and yet retains some of the attractive features of the Jordan form can be constructed by a procedure detailed in the proof of Schur's Theorem [Bel70, p. 202]. This theorem states that there always exists a unitary matrix U such that for a given square matrix A, the matrix $T = U^H AU$ will be triangular with the eigenvalues lying along the diagonal (U^H is the complex conjugate transpose of U). U unitary implies that $U^{-1} = U^H$ so this construction is really a specialized similarity transformation.

Both of the above matrix forms are convenient when calculating functions of matrices (such as the matrix exponential, e^A [Mol76]). However, knowledge of *all* the eigenvalues of A is required and for an exact problem, such information is not obtainable in general. A useful idea, though, is to consider a similarity transformation that will produce a matrix that is nearly triangular. The eigenvalues and hence the characteristic polynomial of a triangular matrix are trivial to discover, and so the hope is that the characteristic polynomial of an almost triangular matrix will be almost trivial to calculate and that this matrix will be almost as easy to use when computing matrix functions. As it turns out, this anticipation is almost correct (the matrix exponential is easier than for a general matrix but it is still hard—see Section 4.3).

A square matrix, H, is upper Hessenberg if $h_{ij} = 0$ whenever $j < i - 1$, and lower Hessenberg if $h_{ij} = 0$ whenever $j > i + 1$. Therefore, an upper Hessenberg matrix is upper triangular except that the subdiagonal can also have nonzero elements, and a lower Hessenberg matrix is lower triangular except that it can have a nonzero superdiagonal as well. The plan outlined here will be to take a general matrix A and transform it via a similarity reduction into Hessenberg form (H). This will always be possible and moreover, involve only rational arithmetic. A second procedure will then be used to compute $c_H(\lambda) = c_A(\lambda)$. This is very much like the idea behind the modular algorithms (Section 2.1.3). The original matrix is transformed into a simpler representation. The desired object is then computed for the simpler problem. The difference here is that this is sufficient, at least if only the characteristic polynomial is desired, since this object will be invariant under the (similarity) transformation. Other quantities, such as eigenvectors, matrix functions, etc., will need to be transformed back to the original space.

To accomplish the reduction of A into H, it is helpful to make use of elementary similarity transformations [Joh81, p. 167]. These are based on the elementary row (and column) operations (2.6) and their matrix representations (2.7). In particular, since $P_{rs}^{-1} = P_{rs}$, $P_{rs}AP_{rs}$ is a matrix similar to A in which both the rows r and s and the columns r and s of A have been interchanged. The other elementary similarity transformation needed here is defined by $E_{rs}(-q)AE_{rs}(q)$. $E_{rs}^{-1}(q) = E_{rs}(-q)$ as can be seen by forming the products $E_{rs}(q)E_{rs}(-q)$ and $E_{rs}(-q)E_{rs}(q)$ [for example, the result of the first multiplication is clearly the identity matrix, observing that the (r,s)-element is formed as $q \cdot 1 + 1 \cdot (-q) = 0$]. The effect of this transformation is to add $-q$ times row s to row r and $+q$ times column r to column s [since $E_{rs}(q) = E_{sr}^T(q)$].

The reduction of A into Hessenberg form will proceed through $n-2$ stages. In the same vein as the description of the Gaussian elimination algorithm in Section 2.1.2, let $A^{(k)}$ be the transformed matrix at the end of the k^{th} stage with $A^{(0)} \equiv A$, and suppose $H \equiv A^{(n-2)}$ is upper Hessenberg. The pivot element here for the k^{th} stage will be $a_{k+1,k}^{(k-1)}$. If the pivot is zero, then if there exists $a_{\ell k}^{(k-1)} \neq 0$ for $\ell > k+1$, swap rows $k+1$ and ℓ and swap columns $k+1$ and ℓ, producing the new matrix $\tilde{A}^{(k-1)}$ (i.e., $\tilde{A}^{(k-1)} = P_{k+1,\ell}A^{(k-1)}P_{k+1,\ell}$). These operations will not affect the previously established pattern of zeroes as can be seen below

for a typical situation in which $n = 6$ and $k = 3$:

$$\begin{array}{c} \\ k+1 \end{array} \begin{array}{c} \begin{array}{cc} \multicolumn{2}{c}{} & k \end{array} \\ \left(\begin{array}{cc|c|ccc} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ \hline 0 & 0 & \times & \times & \times & \times \\ \hline 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \end{array}\right) \end{array}$$

(the crosses represent arbitrary values). If the pivot and the elements below it are all zero, then there is nothing to do, so set $A^{(k)} = A^{(k-1)}$. Otherwise, it will be necessary to eliminate. To simplify the notation, $A^{(k-1)}$ will refer to both $A^{(k-1)}$ and $\tilde{A}^{(k-1)}$ below.

The first part of the elimination is similar to the general Gaussian elimination step (2.8): for the i^{th} row ($i = k+2, \ldots, n$),

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{k+1,k}^{(k-1)}} a_{k+1,j}^{(k-1)} , \quad (j = k, \ldots, n). \tag{2.33}$$

This is equivalent to premultiplying $A^{(k-1)}$ by $E_{i,k+1}(-a_{ik}^{(k-1)}/a_{k+1,k}^{(k-1)})$. To make this a similarity transform, it will be necessary to also postmultiply by the inverse, yielding

$$E_{i,k+1}\left(-\frac{a_{ik}^{(k-1)}}{a_{k+1,k}^{(k-1)}}\right) A^{(k-1)} E_{i,k+1}\left(\frac{a_{ik}^{(k-1)}}{a_{k+1,k}^{(k-1)}}\right) . \tag{2.34}$$

Since $i > k+1$, the column transformation will not disturb the pattern of zeroes already created and so the total effect will be to zero a_{ik}.

The similarity matrix

$$S = \prod_{k=1}^{n-2} \left[\prod_{i=k+2}^{n} E_{i,k+1}\left(\frac{a_{ik}^{(k-1)}}{a_{k+1,k}^{(k-1)}}\right) \right]$$

will have a simple form (ignoring permutations for the moment). This can best be demonstrated by an example. Let $r_{ik} \equiv a_{ik}^{(k-1)}/a_{k+1,k}^{(k-1)}$ and $R_k \equiv \prod_{i=k+2}^{n} E_{i,k+1}(r_{ik})$ so that $S = \prod_{k=1}^{n-2} R_k$. Suppose $n = 5$, then

$$R_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & r_{31} & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & r_{41} & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & r_{51} & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & r_{31} & 1 & 0 & 0 \\ 0 & r_{41} & 0 & 1 & 0 \\ 0 & r_{51} & 0 & 0 & 1 \end{pmatrix} .$$

$H = A$
$S = I$
$S^{-1} = I$
for $k = 1$ **to** $n-2$
  for $i = k+2$ **to** n
    $r_i = h_{ik}/h_{k+1,k}$
    $s_{i,k+1} = r_i$
    $s^{-1}_{i,k+1} = -r_i$
    for $j = 2$ **to** k
      $s^{-1}_{ij} = s^{-1}_{ij} - r_i s^{-1}_{k+1,j}$
    $h_{ik} = 0$
    for $j = k+1$ **to** n
      $h_{ij} = h_{ij} - r_i h_{k+1,j}$
  for $i = 1$ **to** n
    $h_{i,k+1} = h_{i,k+1} + \sum_{j=k+2}^{n} h_{ij} r_j$

Figure 2.5 Algorithm to similarily transform an $n \times n$ general matrix A into upper Hessenberg form H ($H = S^{-1}AS$).

R_2 will be fashioned similarly, and hence

$$S = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & r_{31} & 1 & 0 & 0 \\ 0 & r_{41} & 0 & 1 & 0 \\ 0 & r_{51} & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & r_{42} & 1 & 0 \\ 0 & 0 & r_{52} & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & r_{53} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & r_{31} & 1 & 0 & 0 \\ 0 & r_{41} & r_{42} & 1 & 0 \\ 0 & r_{51} & r_{52} & r_{53} & 1 \end{pmatrix} .$$

The structure of S^{-1} will also be lower triangular with ones along the diagonal, but it will be somewhat more complicated. Figure 2.5 presents the algorithm to compute H, S and S^{-1} given A, assuming all pivot elements are nonzero so that no permutations are necessary.

Of course, permutations will often be necessary or desirable (for example, to choose a simple pivot), but they can easily be handled with indexing arrays. Consider the general form of a reduction:

$$H = S^{-1}AS = (R^{-1}_{n-2}P_{n-2} \cdots R^{-1}_1 P_1) A (P^T_1 R_1 \cdots P^T_{n-2} R_{n-2}) . \tag{2.35}$$

Here, P_k represents a product of simple permutations, P_{rs}, that are applied to $A^{(k-1)}$ (and which may be an identity matrix in some instances). Comparing (2.35) with (2.20) and making note of the subsequent discussion, it is clear that only one indexing array (call it

$\mathbf{I}$) is needed here. If this array is updated as $\mathbf{I}_k = P_k \mathbf{I}_{k-1}$, $(k = 1, \ldots, n-2)$, then the intermediate matrix forms $\hat{H}^{(k)}$, $\hat{S}^{(k)}$ and $\hat{S}^{-1(k)}$ will be defined for $k = 1, \ldots, n-2$ by

$$\begin{aligned} \hat{H}^{(k)}_{\mathbf{I}_k \mathbf{I}_k} &= R_k^{-1} \hat{H}^{(k-1)}_{\mathbf{I}_k \mathbf{I}_k} R_k \ , \\ \hat{S}^{(k)}_{\mathbf{I}_k \mathbf{I}_k} &= \hat{S}^{(k-1)}_{\mathbf{I}_k \mathbf{I}_k} R_k \ , \\ \hat{S}^{-1(k)}_{\mathbf{I}_k \mathbf{I}_k} &= R_k^{-1} \hat{S}^{-1(k-1)}_{\mathbf{I}_k \mathbf{I}_k} \ , \end{aligned} \tag{2.36}$$

with $\hat{H}^{(0)} = A$ and $\hat{S}^{(0)} = \hat{S}^{-1(0)} = I$. These intermediate matrices will correspond to two-dimensional arrays (or their equivalents) in an actual implementation.

The recurrence formulas (2.36) are special cases of (2.22). This latter relation implies (2.24), therefore, the equations (2.36) will result in

$$\begin{aligned} \hat{H}^{(n-2)} &= (P_1^T \cdots P_{n-2}^T) H (P_{n-2}^T \cdots P_1^T) \ , \\ \hat{S}^{(n-2)} &= S (P_{n-2}^T \cdots P_1^T) \ , \\ \hat{S}^{-1(n-2)} &= (P_1^T \cdots P_{n-2}^T) S^{-1} \ , \end{aligned}$$

noting that $X_k = I$, $(k = 1, \ldots, n-2)$ when deriving the second formula and $Y_k = I$ when deriving the third. Thus,

$$H = \hat{H}^{(n-2)}_{\mathbf{I}_{n-2} \mathbf{I}_{n-2}} \ , \ S = \hat{S}^{(n-2)}_{\bullet, \mathbf{I}_{n-2}} \quad \text{and} \quad S^{-1} = \hat{S}^{-1(n-2)}_{\mathbf{I}_{n-2}, \bullet} \ . \tag{2.37}$$

A MATLAB implementation of the algorithm indicated above involving an indexing array is presented in Figure 2.6. A is assumed given and while no permutations are actually computed, the structure is in place to handle them. Introducing permutations will result in S and S^{-1} no longer necessarily being lower triangular (although $\hat{S}^{(n-2)}_{\mathbf{I}_{n-2} \mathbf{I}_{n-2}}$ and $\hat{S}^{-1(n-2)}_{\mathbf{I}_{n-2} \mathbf{I}_{n-2}}$ will always be).

By the way, if the final outcome of A's reduction is desired to be lower Hessenberg (H_L) rather than upper Hessenberg (H^U), this can be accomplished by simply folding in one more permutation into the indexing array just before the transformations (2.37) are applied. The similarity transformation $K H^U K$, where

$$K = \begin{pmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{pmatrix}$$

is a skew identity matrix (since the ones lie along its skew diagonal), will yield the desired form H_L. This is equivalent to applying K to $\mathbf{I}_{n-2}$ which has the effect of reversing the order of the elements in the indexing array.

After transforming A into H, it is still necessary to compute $c_H(\lambda)$. If any of the elements in the subdiagonal of H are zero [see, for example, the matrix T in equation (2.3)], then H is block upper triangular and so the problem can be broken down into simpler subproblems. The smaller upper Hessenberg matrices that will result will have nonzero subdiagonals and

```
n = sum(diag(eye(A)));
Indx = [1:n]';
H    = A;
S    = eye(n);
SINV = eye(n);
for k = 1:n-2, ...
   for i = k+2:n, ...
      ... Update Indx
      R(i) = H(Indx(i), Indx(k)) / H(Indx(k+1), Indx(k)); ...
      S(Indx(i), Indx(k+1)) = R(i); ...
      SINV(Indx(i), Indx(k+1)) = -R(i); ...
      for j = 2:k, ...
         SINV(Indx(i), Indx(j)) = SINV(Indx(i), Indx(j)) ...
                                  - R(i) * SINV(Indx(k+1), Indx(j)); ...
         end; ...
      H(Indx(i), Indx(k)) = 0; ...
      for j = k+1:n, ...
         H(Indx(i), Indx(j)) = H(Indx(i), Indx(j)) ...
                               - R(i) * H(Indx(k+1), Indx(j)); ...
         end; ...
      end; ...
   for i = 1:n, ...
      q = 0; ...
      for j = k+2:n, ...
         q = q + H(Indx(i), Indx(j)) * R(j); ...
         end; ...
      H(Indx(i), Indx(k+1)) = H(Indx(i), Indx(k+1)) + q; ...
      end;
   end;
for i = 1:n, ...
   for j = 1:n, ...
      HH(i, j)    = H(Indx(i), Indx(j)); ...
      SS(i, j)    = S(i, Indx(j)); ...
      SSINV(i, j) = SINV(Indx(i), j); ...
      end; ...
   end;
H = HH;   S = SS;   SINV = SSINV;
```

Figure 2.6 Partial MATLAB implementation of the algorithm to similarily transform an $n \times n$ general matrix A into upper Hessenberg form H ($H = S^{-1}AS$) in which an indexing array, `Indx`, is used to record permutations.

thus shall be in what is known as standard upper Hessenberg form. Hence, from now on, assume H to be standard upper Hessenberg.

The procedure for computing $c_H(\lambda)$ involves using H to generate a triangular system of equations. The unique solution of this system will then give the coefficients of the characteristic polynomial. To begin, let $\mathbf{w}_0 = \mathbf{e}_1$, where $\mathbf{e}_1$ is the $n \times 1$ unit vector $(1, 0, \ldots, 0)^T$. Next, define

$$\mathbf{w}_k = H\mathbf{w}_{k-1} \ , \quad (k = 1, \ldots, n)$$

so that $\mathbf{w}_k = H^k\mathbf{e}_1$, $(k = 0, \ldots, n)$. Finally, construct the matrix equation

$$W\mathbf{a} \equiv (\mathbf{w}_0|\cdots|\mathbf{w}_{n-1})\mathbf{a} = -\mathbf{w}_n \ , \tag{2.38}$$

where $\mathbf{a} \equiv (a_0, \ldots, a_{n-1})^T$. W will be upper triangular (as shall be seen below), so $\{\mathbf{w}_0, \ldots, \mathbf{w}_{n-1}\}$ forms a linearly independent set of vectors and thus a basis for $\mathbf{R}^n$. Hence, any vector such as $-\mathbf{w}_n \in \mathbf{R}^n$ can be expressed as a unique linear combination of these basis vectors. Since (2.38) can be rewritten as

$$\sum_{k=0}^{n-1} a_k\mathbf{w}_k = -\mathbf{w}_n \ ,$$

it is clear that $\mathbf{a}$ is indeed uniquely determined.

The above equation can be further rewritten as

$$\sum_{k=0}^{n-1} a_k H^k\mathbf{e}_1 = -H^n\mathbf{e}_1$$

or

$$p(H)\mathbf{e}_1 \equiv \left(H^n + \sum_{k=0}^{n-1} a_k H^k\right)\mathbf{e}_1 = \mathbf{0} \ .$$

$p(H)$ is the zero matrix. This can be seen by considering its effect on an arbitrary vector, $\mathbf{y} \in \mathbf{R}^n$, expanded in terms of the above basis [Joh81, p. 185]: $\mathbf{y} = \sum_{i=0}^{n-1} c_i\mathbf{w}_i = \sum_{i=0}^{n-1} c_i H^i\mathbf{e}_1$. For convenience, define a_n to be one, then

$$\begin{aligned} p(H)\mathbf{y} &= \sum_{k=0}^{n} a_k H^k \sum_{i=0}^{n-1} c_i H^i\mathbf{e}_1 = \sum_{i=0}^{n-1} c_i H^i \sum_{k=0}^{n} a_k H^k\mathbf{e}_1 \\ &= \sum_{i=0}^{n-1} c_i H^i p(H)\mathbf{e}_1 = \mathbf{0} \end{aligned}$$

since H^k obviously commutes with H^i. As $\mathbf{y}$ is arbitrary, $p(H) = 0$ is implied [for example, choose $\mathbf{y}$ to be the elementary unit vectors $\mathbf{e}_j$, $(j = 1, \ldots, n)$, where $(\mathbf{e}_j)_i \equiv \delta_{ij}$, $(i = 1, \ldots, n)$—this set of choices will pick out the columns of $p(H)$, one by one].

Suppose that $(\lambda, \mathbf{x})$ is an eigenvalue-eigenvector pair for H (so $\mathbf{x} \neq \mathbf{0}$), then $H\mathbf{x} = \lambda\mathbf{x}$, $H^2\mathbf{x} = H(H\mathbf{x}) = H(\lambda\mathbf{x}) = \lambda(H\mathbf{x}) = \lambda^2\mathbf{x}$, and in general, $H^k\mathbf{x} = \lambda^k\mathbf{x}$ for $k = 0, 1, \ldots$ Thus, $p(H)\mathbf{x} = p(\lambda)\mathbf{x} = \mathbf{0}$ since $p(H) = 0$, and so $p(\lambda)$ must be zero. This is true for each *distinct* eigenvalue λ of H, therefore, each distinct eigenvalue will be a root of $p(\lambda)$. If all

the eigenvalues are distinct, then since there are exactly n eigenvalues of H and n roots of $p(\lambda)$, the converse must also be true. Hence, $p(\lambda) = c_H(\lambda)$ since the coefficient of λ^n is one for both, and two polynomials with identical roots can differ by at most a scalar multiplier. If the eigenvalues of H are not all distinct, consider $H_\varepsilon = H + \varepsilon L$, where L is chosen so that H_ε has distinct eigenvalues for all $\varepsilon > 0$ (such an L can be constructed, for example, as SDS^{-1}, where D is diagonal and $S^{-1}HS$ is in Jordan form). Define $p_\varepsilon(\lambda)$ to be the polynomial devised by the same procedure as $p(\lambda)$ but using H_ε [thus, $p_\varepsilon(H_\varepsilon) = 0$]. By the earlier argument, $p_\varepsilon(\lambda) = c_{H_\varepsilon}(\lambda)$. Both $p_\varepsilon(\lambda)$ and $c_{H_\varepsilon}(\lambda)$ will depend continuously on ε, which ultimately will appear only in positive powers in either polynomial, therefore, the limits of these two functions as $\varepsilon \to 0$ exist and are equal. Thus,

$$p(\lambda) = \lim_{\varepsilon \to 0} p_\varepsilon(\lambda) = \lim_{\varepsilon \to 0} c_{H_\varepsilon}(\lambda) = c_H(\lambda) \ ,$$

and so the equality holds even for the case of repeated eigenvalues.

It now remains to show that W is upper triangular and furthermore, (2.38) is equivalent to the matrix equation

$$U\mathbf{a} = -\mathbf{u}_n \ , \tag{2.39}$$

where U is also upper triangular but with ones along its diagonal so that the solution (by back substitution) of this system will require no divisions. To do this, let s_i denote the subdiagonal element $h_{i+1,i}$ of the upper Hessenberg matrix ($i = 1, \ldots, n-1$) and define $s_{ij} = s_i s_{i+1} \cdots s_j$, ($1 \le i \le j < n$). If the columns of U are designated $\mathbf{u}_0, \ldots, \mathbf{u}_{n-1}$, then it will be seen below that $\mathbf{u}_j$ is related to $\mathbf{w}_j$ by $\mathbf{w}_j = D\mathbf{u}_j$, ($j = 0, \ldots, n$) where $D = \mathrm{diag}(1, s_{11}, \ldots, s_{1,n-1})$ [that is, D is a diagonal matrix and its diagonal consists of $\{1, s_1, s_1 s_2, \ldots, s_1 \cdots s_{n-1}\}$].

Suppose that H contains polynomial entries, that is, $h_{ij} \in \mathbf{Z}[x_1, \ldots, x_v]$. It is clear that $\mathbf{w}_j$, ($j = 0, \ldots, n$) will have polynomial entries as well. So, how should the $\mathbf{u}_j$ be defined? $\mathbf{w}_0 \equiv \mathbf{e}_1$ so take $\mathbf{u}_0 = \mathbf{e}_1$ also. $\mathbf{w}_1 = H\mathbf{w}_0 = (h_{11}, s_1, 0, \ldots, 0)^T$, so choose $\mathbf{u}_1 = (h_{11}, 1, 0, \ldots, 0)^T$. Consider now the computation of $\mathbf{w}_j$ assuming that $\mathbf{w}_{j-1}$ is of the form $(u_{1,j-1}, s_{11}u_{2,j-1}, \ldots, s_{1,j-2}u_{j-1,j-1}, s_{1,j-1}, 0, \ldots, 0)^T$ and all the factors involved are polynomial. If $\mathbf{w}_j$ can be put in a corresponding form, then the proposition put forth in the previous paragraph will be inductively verified as $\mathbf{w}_1$ has already been established to have such a structure. The actual calculation proceeds as follows:

$$\begin{aligned} w_j &= Hw_{j-1} \\ &= \begin{pmatrix} h_{11} & h_{12} & \cdots & & & & h_{1n} \\ s_1 & h_{22} & \cdots & & & & h_{2n} \\ & \ddots & \ddots & & & & \vdots \\ & & s_{j-2} & h_{j-1,j-1} & h_{j-1,j} & \cdots & h_{j-1,n} \\ & & & s_{j-1} & h_{jj} & \cdots & h_{jn} \\ & & & & s_j & h_{j+1,j+1} \ \cdots & h_{j+1,n} \\ & 0 & & & & \ddots \quad \ddots & \vdots \\ & & & & & s_{n-1} & h_{nn} \end{pmatrix} \begin{pmatrix} u_{1,j-1} \\ s_{11}u_{2,j-1} \\ \vdots \\ s_{1,j-2}u_{j-1,j-1} \\ s_{1,j-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \end{aligned}$$

$$= \begin{pmatrix} h_{11}u_{1,j-1} + h_{12}u_{2,j-1}s_{11} + \cdots + h_{1,j-1}u_{j-1,j-1}s_{1,j-2} + h_{1j}s_{1,j-1} \\ s_{11}(u_{1,j-1} + h_{22}u_{2,j-1} + \cdots + h_{2,j-1}u_{j-1,j-1}s_{2,j-2} + h_{2j}s_{2,j-1}) \\ \vdots \\ s_{1,j-2}(u_{j-2,j-1} + h_{j-1,j-1}u_{j-1,j-1} + h_{j-1,j}s_{j-1,j-1}) \\ s_{1,j-1}(u_{j-1,j-1} + h_{jj}) \\ s_{1j} \\ \vdots \\ 0 \end{pmatrix} \equiv \begin{pmatrix} u_{1j} \\ s_{11}u_{2j} \\ \vdots \\ s_{1,j-2}u_{j-1,j} \\ s_{1,j-1}u_{jj} \\ s_{1j} \\ \vdots \\ 0 \end{pmatrix} .$$

This proves the proposition.

The formal algorithm for computing $c_H(\lambda)$ is presented in Figure 2.7. Note that $u_{11} = h_{11}$ and $u_{jj} = u_{j-1,j-1} + h_{jj}$, $(j = 2, \ldots, n)$. Therefore, $u_{nn} = \sum_{j=1}^{n} h_{jj} = \operatorname{tr} H$ and since $u_{n,n-1} = 1$, (2.39) implies that $a_{n-1} = -\operatorname{tr} H$ [compare with equation (2.32)].

Operation counts for the two algorithms discussed above, transforming A into H and computing $c_H(\lambda)$, are given in Tables 2.5a and 2.6a, respectively. The counts in Table 2.5a are divided into two main sections. The number of operations required to compute S^{-1} are listed in the lower section, while those needed to form H are given in the upper. In calculating H, the elements of S are automatically created, so the number of these division operations are listed separately, although they must also be included as part of the work needed for the calculation of H.

The major problem with the Hessenberg algorithm presented here is that it involves divisions. Divisions are undesirable in a symbolic context since they can transform a purely polynomial problem into one requiring rational operations. Unfortunately, it is more difficult to make similarity reductions division-free than was the case for the determinant Gaussian elimination procedures, since any scaling applied to one side of a matrix must be balanced by applying the inverse scaling to the other side. In fact, the best that can be done is to multiply *both* sides of the matrix by scalars (or polynomials) at each stage of the transformation, and then try to be clever about using the accumulated product of these factors when computing the characteristic polynomial or the eigenvalues of the original matrix.

Equation (2.35) shows the relationship between H, S, S^{-1} and the matrices R_k and R_k^{-1}, $(k = 1, \ldots, n-2)$ [permutations will be ignored for now]. Let $r'_{ik} \equiv a_{ik}^{(k-1)}$ and $p_k \equiv a_{k+1,k}^{(k-1)}$, (so $r_{ik} = r'_{ik}/p_k$), then $R_k^{-1\prime} A^{(k-1)} R'_k$, where $R'_k \equiv p_k R_k$ and $R_k^{-1\prime} \equiv p_k R_k^{-1}$, will zero a_{ik}, $(i = k+2, \ldots, n)$, just as $R_k^{-1} A^{(k-1)} R_k$ does; however, this transformation will be division-free. R'_k will have the same structure as R_k, except that the entries r_{ik} will be replaced by r'_{ik} and the ones along the diagonal by p_k. Thus, if A has polynomial entries, then so will the iterates $A^{(k)}$ along with the final upper Hessenberg matrix $H' \equiv A^{(n-2)}$.

Form Triangular System

for $i = 1$ **to** $n-1$
    $s_{ii} = h_{i+1,i}$
    for $j = i+1$ **to** $n-1$
        $s_{ij} = s_{i,j-1} h_{j+1,j}$
$u_{11} = h_{11}$
for $j = 2$ **to** n
    for $i = 1$ **to** $j-1$
        $u_{ij} = h_{ii} u_{i,j-1} + \sum_{k=i+1}^{j-1} h_{ik} u_{k,j-1} s_{i,k-1} + h_{ij} s_{i,j-1}$
        if $i > 1$ **then**
            $u_{ij} = u_{i-1,j-1} + u_{ij}$
    $u_{jj} = u_{j-1,j-1} + h_{jj}$

Solve Triangular System

$a_{n-1} = -u_{nn}$
for $i = n-1$ **to** 1 [**step** -1]
    $a_{i-1} = -\left(\sum_{j=i}^{n-1} u_{ij} a_j + u_{in} \right)$

Compute Characteristic Polynomial

$$c_H(\lambda) = \lambda^n + \sum_{k=2}^{n-1} a_k \lambda^k + a_1 \lambda + a_0$$

Figure 2.7 Algorithm to compute the characteristic polynomial $c_H(\lambda)$ of a standard $n \times n$ upper Hessenberg matrix H.

H	$+$	$\frac{5}{6}n^3 - \frac{5}{2}n^2 + \frac{5}{3}n$
	$-$	$\frac{1}{3}n^3 - n^2 + \frac{2}{3}n$
	$\times$	$\frac{5}{6}n^3 - \frac{5}{2}n^2 + \frac{5}{3}n$
S	$\div$	$\frac{1}{2}n^2 - \frac{3}{2}n + 1$
S^{-1}	$+$	$\frac{1}{6}n^3 - n^2 + \frac{11}{6}n - 1$
	$-$	$\frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n$
	$\times$	$\frac{1}{6}n^3 - n^2 + \frac{11}{6}n - 1$

a. General into upper Hessenberg

H	$+$	$\frac{5}{6}n^3 - \frac{5}{2}n^2 + \frac{5}{3}n$
	$-$	$\frac{1}{3}n^3 - n^2 + \frac{2}{3}n$
	$\times$	$2n^3 - \frac{7}{2}n^2 - \frac{1}{2}n$
S	$\times$	$\frac{1}{3}n^3 - \frac{3}{2}n^2 + \frac{13}{6}n$
S^{-1}	$+$	$\frac{1}{6}n^3 - n^2 + \frac{11}{6}n - 1$
	$-$	$\frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n$
	$\times$	$\frac{1}{2}n^3 - 3n^2 + \frac{11}{2}n - 3$

b. General into upper Hessenberg (division free)

T^*	$+$	$\frac{2}{3}n^3 + n^2 - \frac{23}{3}n + 6$
	$-$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 - \frac{5}{6}n + 1$
	$\times$	$\frac{2}{3}n^3 + \frac{9}{2}n^2 - \frac{91}{6}n + 11$
	$\div$	$n^2 - 2n + 2$
	conj	$\frac{1}{3}n^3 - n^2 + \frac{2}{3}n$
	Re	$\frac{1}{2}n^2 - \frac{3}{2}n + 1$
S	$+$	$n^3 - 4n^2 + 5n - 2$
	$-$	$\frac{1}{2}n^3 - 2n^2 + \frac{5}{2}n - 1$
	$\times$	$n^3 - 4n^2 + 5n - 2$
S^{-1}	$\times$	$2n^2 - 4n + 3$
	conj	n^2

c. Hermitian into tridiagonal (modified fast Givens')

conj	$2(j-i)-1$

d. Similarity permutation P_{ij} applied to the upper triangle of a Hermitian matrix

Table 2.5 Rational operation counts for various transformation algorithms applied to an $n \times n$ matrix.

$+$	$\frac{1}{6}n^3 + n^2 - \frac{7}{6}n$
$-$	n
$\times$	$\frac{1}{3}n^3 + n^2 - \frac{7}{3}n + 1$

a. Standard upper Hessenberg

$+$	$\frac{3}{2}n^2 - \frac{3}{2}n$
$-$	n
$\times$	$2n^2 - 3n + 1$

b. Standard tridiagonal

Table 2.6 Rational operation counts for the algorithms which compute the characteristic polynomial of two different types of $n \times n$ matrices.

The matrix $S' \equiv \prod_{k=1}^{n-2} R'_k$ will resemble S, although it will be more complex. For example, again for $n = 5$,

$$S' = \begin{pmatrix} p_1 & 0 & 0 & 0 & 0 \\ 0 & p_1 & 0 & 0 & 0 \\ 0 & r'_{31} & p_1 & 0 & 0 \\ 0 & r'_{41} & 0 & p_1 & 0 \\ 0 & r'_{51} & 0 & 0 & p_1 \end{pmatrix} \begin{pmatrix} p_2 & 0 & 0 & 0 & 0 \\ 0 & p_2 & 0 & 0 & 0 \\ 0 & 0 & p_2 & 0 & 0 \\ 0 & 0 & r'_{42} & p_2 & 0 \\ 0 & 0 & r'_{52} & 0 & p_2 \end{pmatrix} \begin{pmatrix} p_3 & 0 & 0 & 0 & 0 \\ 0 & p_3 & 0 & 0 & 0 \\ 0 & 0 & p_3 & 0 & 0 \\ 0 & 0 & 0 & p_3 & 0 \\ 0 & 0 & 0 & r'_{53} & p_3 \end{pmatrix}$$

$$= \begin{pmatrix} p_1p_2p_3 & 0 & 0 & 0 & 0 \\ 0 & p_1p_2p_3 & 0 & 0 & 0 \\ 0 & p_2p_3r'_{31} & p_1p_2p_3 & 0 & 0 \\ 0 & p_2p_3r'_{41} & p_1p_3r'_{42} & p_1p_2p_3 & 0 \\ 0 & p_2p_3r'_{51} & p_1p_3r'_{52} & p_1p_2r'_{53} & p_1p_2p_3 \end{pmatrix} .$$

Defining $d_S = d_{S^{-1}} = \prod_{k=1}^{n-2} p_k$, S' and $S^{-1\prime}$ will be related to S and S^{-1} by $S' = d_S S$ and $S^{-1\prime} = d_{S^{-1}} S^{-1}$. Note that $S^{-1\prime} \neq S'^{-1}$ [$S^{-1\prime}S' = (d_{S^{-1}}S^{-1})(d_S S) = d_H I$ where $d_H \equiv d_{S^{-1}} d_S$]. Moreover,

$$H' = S^{-1\prime} A S' = (d_{S^{-1}} S^{-1}) A (d_S S) = d_{S^{-1}} d_S (S^{-1} A S) = d_H H \ .$$

There also exists simple relationships between $c_H(\lambda)$ and $c_{H'}(\lambda')$ as well as between the eigenvalues corresponding to these two polynomials ($\{\lambda_i\}_{i=1}^n$ and $\{\lambda'_i\}_{i=1}^n$, respectively). In particular,

$$\begin{aligned} c_H(\lambda) &= \det(H - \lambda I) = \det\left(\frac{1}{d_H} H' - \lambda I\right) = \det\left(\left[\frac{1}{d_H} I\right][H' - d_H \lambda I]\right) \\ &= \det\left(\frac{1}{d_H} I\right) \det(H' - d_H \lambda I) = \frac{1}{{d_H}^n} \det(H' - [d_H \lambda] I) = \frac{1}{{d_H}^n} c_{H'}(d_H \lambda) \ . \end{aligned}$$

Thus, a multiple of $c_H(\lambda)$ [i.e., $c_{H'}(d_H\lambda) = {d_H}^n c_H(\lambda)$] is computable in an entirely division-free manner. Also,

$$\prod_{i=1}^{n} (\lambda - \lambda_i) = c_H(\lambda) = \frac{1}{{d_H}^n} c_{H'}(d_H \lambda) = \frac{1}{{d_H}^n} \prod_{i=1}^{n} (d_H \lambda - \lambda'_i) = \prod_{i=1}^{n} \left(\lambda - \frac{\lambda'_i}{d_H}\right) \ ,$$

so $\lambda_i = \lambda'_i / d_H$.

Figure 2.8 displays the division-free algorithm for transforming A into H' (the primes on H, S and S^{-1} have been dropped to simplify the notation) and the corresponding operation counts are given in Table 2.5b. Comparing Figure 2.8 with Figure 2.5 shows that making the algorithm division-free has more than doubled its length. A corresponding comparison of the operation counts indicates that the number of multiplications in the division-free algorithm versus the number of multiplications plus divisions in the original method is roughly double, while the counts of additions and negations are exactly the same. Permutations can be incorporated into this new procedure in exactly the same manner as was done for the original algorithm.

The division-free algorithm can potentially generate very large quantities. This can be mitigated to some extent by reducing the elements of the R'_k (and the $R_k^{-1\prime}$) by any common factors that might appear in the course of the computation. The algorithm in Figure 2.8 can be modified by replacing the three lines at the beginning of the loop on k by the section of code listed in Figure 2.9. Even with this change, the size of the expressions produced can still become quite large (see Chapter 5).

2.2.3 Fast Givens' Method

The Hessenberg method is applicable to general matrices. As such, no advantage is taken of any symmetries that may be present. A very common type of symmetry occurs when a matrix is Hermitian, in which the matrix is equal to its complex conjugate transpose. In this situation, the number of independent elements is $\frac{n^2+n}{2}$ rather than n^2, and so it is common to represent just the upper (or lower) triangle of such a matrix. In what follows, A will be assumed to be Hermitian ($A = \bar{A}^T \equiv A^H$) with only its upper triangle explicitly specified.

The reduction to upper Hessenberg form in the previous subsection consisted of zeroing out the elements of the matrix there below the subdiagonal. It was then relatively easy to compute the characteristic polynomial of the Hessenberg matrix. In a like manner, the similarity reduction of the Hermitian matrix A can be arranged so that the elements below the subdiagonal and above the superdiagonal are simultaneously eliminated. It is then very simple to compute the characteristic polynomial of the tridiagonal matrix (T) that results.

$H = A$
$S = I \quad ; \quad d_S = 1$
$S^{-1} = I \quad ; \quad d_{S^{-1}} = 1$
for $k = 1$ **to** $n - 2$
  for $i = k + 2$ **to** n
    $r'_i = h_{ik}$
  $p = h_{k+1,k}$
  for $i = n$ **to** 1 [**step** -1]
    for $j = k + 1$ **to** n
      $h_{ij} = ph_{ij}$
      if $i > k + 1$ **then**
        $h_{ij} = h_{ij} - r'_i h_{k+1,j}$
    $h_{i,k+1} = ph_{i,k+1} + \sum_{j=k+2}^{n} h_{ij} r'_j$
    for $j = k + 2$ **to** n
      $h_{ij} = ph_{ij}$
    if $i \leq k + 1$ **then**
      for $j = \max(i - 1, 1)$ **to** k
        $h_{ij} = p^2 h_{ij}$
    else
      $h_{ik} = 0$
  for $j = 2$ **to** k
    for $i = n$ **to** $j + 1$ [**step** -1]
      $s_{ij} = ps_{ij}$
      $s^{-1}_{ij} = ps^{-1}_{ij}$
      if $i > k + 1$ **then**
        $s^{-1}_{ij} = s^{-1}_{ij} - r'_i s^{-1}_{k+1,j}$
  for $i = k + 2$ **to** n
    $s_{i,k+1} = d_S r'_i$
    $s^{-1}_{i,k+1} = -s_{i,k+1}$
  $d_H = p^2 d_H$
  $d_S = p d_S$
  $d_{S^{-1}} = d_S$
  for $i = 1$ **to** n
    $s_{ii} = d_S$
    $s^{-1}_{ii} = d_{S^{-1}}$

Figure 2.8 Division-free algorithm to similarily transform an $n \times n$ general matrix A into upper Hessenberg form. d_X is the common divisor of the elements of the matrix X.

$r'_{k+1} = h_{k+1,k}$
$g = r'_{k+1}$
for $i = k+2$ **to** n
    $r'_i = h_{ik}$
    $g = \gcd(g, r'_i)$
if $g \neq 1$ **then**
  for $i = k+1$ **to** n
      $r'_i = r'_i / g$
$p = r'_{k+1}$

Figure 2.9 Modification to the division-free algorithm of Figure 2.8 to remove unnecessary common factors in the matrices R'_k (this segment replaces the three lines following the definition of the k-loop).

One way to transform A into Hermitian tridiagonal form is to use plane rotations. The matrix representation of a typical plane rotation looks as follows:

$$R_{uv}(c,s) = \begin{array}{c} \\ \\ \\ \\ u \\ \\ \\ \\ v \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccccccc} & & & & u & & & & v & & \end{array} \\ \begin{pmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & \bar{c} & 0 & \cdots & 0 & -s & & & \\ & & & 0 & 1 & & & 0 & & & \\ & & & \vdots & & \ddots & & \vdots & & & \\ & & & 0 & & & 1 & 0 & & & \\ & & & \bar{s} & 0 & \cdots & 0 & c & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{pmatrix} \end{array} .$$

Any elements not explicitly indicated are zero. c and s are defined to have the property that $|c|^2 + |s|^2 = 1$. Thus,

$$\det(R_{uv}(c,s)) = \bar{c}c + (-1)^{v-u}(-s)(-1)^{v-u-1}\bar{s} = \bar{c}c + s\bar{s} = 1 \ ,$$

expanding by cofactors along rows. Multiplying $R^H_{uv}(c,s) = R_{uv}(\bar{c}, -s)$ with $R_{uv}(c,s)$ establishes that $R^{-1}_{uv}(c,s) = R^H_{uv}(c,s)$, therefore, $R_{uv}(c,s)$ is unitary and so $R^H_{uv}(c,s)AR_{uv}(c,s)$ can be classified as an elementary unitary transformation.

These elementary unitary transformations can be used to reduce A into T in a series of $n-2$ stages. The discussion of the algorithm closely parallels the description of the procedure in the previous subsection for transforming a general matrix into upper Hessenberg form and

so only the differences will be noted. In particular, let $A^{(k,j)}$ denote the transformed matrix at the end of the j^{th} step within the k^{th} stage ($j = k+2, \ldots, n$ for $k = 1, \ldots, n-2$). Also, let the notation $A^{(k,j)-1}$ refer to the matrix at the end of the previous step [this will correspond to $(k, j-1)$ if $j > k+2$ and $(k-1, n)$ if $j = k+2$]. If $c_{k+1,j} \equiv a_{k,k+1}^{(k,j)-1}/r_{k+1,j}$ and $s_{k+1,j} \equiv a_{kj}^{(k,j)-1}/r_{k+1,j}$ where $r_{k+1,j} \equiv \sqrt{|a_{k,k+1}^{(k,j)-1}|^2 + |a_{kj}^{(k,j)-1}|^2}$, then implicitly taking $R_{k+1,j} = R_{k+1,j}(c_{k+1,j}, s_{k+1,j})$,

$$A^{(k)} = R_{k+1,n}^H \cdots R_{k+1,k+2}^H A^{(k-1)} R_{k+1,k+2} \cdots R_{k+1,n}$$

will be the similarity transformation that zeroes a_{kj} and a_{jk} [$= \bar{a}_{kj}$], ($j = k+2, \ldots, n$). Of course, if any of the a_{kj}'s are already zero, then the corresponding $R_{k+1,j}^H$, $R_{k+1,j}$ pair can be omitted from the above formula [note that $R_{k+1,j}(c_{k+1,j}, 0) = I$ if $a_{k,k+1}^{(k,j)-1}$ is real and positive].

The above process is known as Givens' method [Wil65, p. 282]. The major problem with this algorithm (and also Householder's method, which is the other common procedure for transforming A into T) is that square roots are involved. Square roots in a symbolic context are generally non-rational operations and tend to introduce great complexity in the results produced. Thus, a square root free transformation is to be desired. It turns out that by suitably scaling the similarity matrices in the Givens' method, such a reduction can be accomplished. This variation is called the fast Givens' method [Gen73, Ham74, Mol77] and a modification of the version discussed in [Mol77] is described below.

Consider a generalization of the eigenproblem introduced earlier, namely $A\mathbf{x} = \lambda D'\mathbf{x}$, where A is Hermitian and D' is real diagonal with $d'_{ii} \neq 0$, ($i = 1, \ldots, n$). An equivalent formulation is $(D'^{-1}A)\mathbf{x} = \lambda\mathbf{x}$. The idea here is to create a matrix S such that $T = S^H A S$ is (Hermitian) tridiagonal while under the same transformation, $D = S^H D' S$ remains real diagonal. $S^{-1} \neq S^H$ normally, so T will not be similar to A. However, S^{-1} is simply related to S^H by $S^{-1} = D^{-1} S^H D'$ (since $DS^{-1} = S^H D'$), therefore,

$$T^* \equiv S^{-1}(D'^{-1}A)S = (D^{-1}S^H D')(D'^{-1}A)S = D^{-1}(S^H A S) = D^{-1}T$$

relates T to a tridiagonal (but not necessarily Hermitian) matrix T^* that *is* similar to $D'^{-1}A$. [Note that the d_{ii}, ($i = 1, \ldots, n$) must also be nonzero, which can be guaranteed, as shall be seen below, by choosing the d'_{ii} to all have the same sign.]

In a manner analogous to the regular Givens' method, S will be constructed from matrices $Z_{uv}(\alpha, \beta)$ which are used to eliminate one element at a time. These matrices are defined by (where $u \neq v$)

$$(Z_{uv}(\alpha,\beta))_{ii} = 1, \quad (Z_{uv}(\alpha,\beta))_{uv} = -\alpha, \quad (Z_{uv}(\alpha,\beta))_{vu} = \bar{\beta}.$$

Note that $Z_{uv}(\alpha, \alpha)$ is identical to $R_{uv}(1, \alpha)$, although this matrix is, of course, not a strict plane rotation since $\det(R_{uv}(1,\alpha)) = 1 + |\alpha|^2 > 1$ for $\alpha \neq 0$. Also, $Z_{uv}^H(\alpha,\beta) = Z_{uv}(-\beta, -\alpha)$.

The algorithm for the fast Givens' method starts out by initializing $A^{(0,n)} = A$ and $D^{(0,n)} = D'$. If $A^{(k,j)}$ and $D^{(k,j)}$ are the transformed matrices at the end of the j^{th} step of the k^{th} stage ($k = 1, \ldots, n-2$ and $j = k+2, \ldots, n$), then the matrices T and D defined

above will be given by $A^{(n-2,n)}$ and $D^{(n-2,n)}$, respectively. The superscripts will correspond to the element of A that has just undergone elimination.

The current value of $a_{k,k+1}$ will act as the pivot element in the elimination of a_{kj}. If this quantity is zero, then a (similarity) permutation will be needed to bring a nonzero element into the pivot position (selected from a_{kj}, $j = k+2, \ldots, n$) if possible. If there is no such element, then the algorithm proceeds to the next stage; otherwise, assume this operation has been done.

Let $\alpha_{k+1,j} \equiv a_{kj}^{(k,j)-1} / a_{k,k+1}^{(k,j)-1}$ and $\beta_{k+1,j} \equiv (d_{k+1,k+1}^{(k,j)-1} / d_{jj}^{(k,j)-1}) \alpha_{k+1,j}$ where the notation $(k,j)-1$ has the same meaning as before. The transformations

$$\begin{aligned} A^{(k,j)} &= Z_{k+1,j}^H(\alpha_{k+1,j}, \beta_{k+1,j}) A^{(k,j)-1} Z_{k+1,j}(\alpha_{k+1,j}, \beta_{k+1,j}) \\ D^{(k,j)} &= Z_{k+1,j}^H(\alpha_{k+1,j}, \beta_{k+1,j}) D^{(k,j)-1} Z_{k+1,j}(\alpha_{k+1,j}, \beta_{k+1,j}) \end{aligned}$$

will then introduce a zero at a_{kj} (and a_{jk}) while preserving the real diagonality of D (in particular,

$$\begin{aligned} d_{k+1,k+1}^{(k,j)} &= d_{k+1,k+1}^{(k,j)-1} + |\beta_{k+1,j}|^2 d_{jj}^{(k,j)-1} \\ d_{jj}^{(k,j)} &= d_{jj}^{(k,j)-1} + |\alpha_{k+1,j}|^2 d_{k+1,k+1}^{(k,j)-1} \quad). \end{aligned}$$

Notice that if $a_{kj}^{(k,j)-1}$ is already zero, then $\alpha_{k+1,j} = \beta_{k+1,j} = 0$ and $Z_{k+1,j}(0,0) = Z_{k+1,j}^H(0,0) = I$.

The correspondence between the fast and the classical Givens' method can be seen now by considering the matrix $U \equiv D'^{1/2} S D^{-1/2}$. U is unitary since

$$U^{-1} = D^{1/2} S^{-1} D'^{-1/2} = D^{1/2} (D^{-1} S^H D') D'^{-1/2} = D^{-1/2} S^H D'^{1/2} = U^H .$$

Rewriting the eigenproblem $A\mathbf{x} = \lambda D'\mathbf{x} = \lambda (D'^{1/2} D'^{1/2})\mathbf{x}$ as $(D'^{-1/2} A D'^{-1/2})\mathbf{y} = \lambda \mathbf{y}$ where $\mathbf{y} = D'^{1/2}\mathbf{x}$, it follows that

$$\begin{aligned} U^H (D'^{-1/2} A D'^{-1/2}) U &= (D^{-1/2} S^H D'^{1/2})(D'^{-1/2} A D'^{-1/2})(D'^{1/2} S D^{-1/2}) \\ &= D^{-1/2} (S^H A S) D^{-1/2} = D^{-1/2} T D^{-1/2} \end{aligned}$$

expresses the relationship between the Hermitian matrix $D'^{-1/2} A D'^{-1/2}$ and the Hermitian tridiagonal matrix $D^{-1/2} T D^{-1/2}$. Furthermore, simplifying the notation somewhat and considering the case when $D' = I$, the product of $Z_{k+1,j}$ with $\mathrm{diag}(\bar{a}_{k,k+1}/r, a_{k,k+1}/r)$ where r is a shorthand for $r_{k+1,j}$, i.e.,

$$\begin{pmatrix} 1 & -\dfrac{a_{kj}}{a_{k,k+1}} \\ \dfrac{d_{k+1,k+1}}{d_{jj}} \dfrac{\bar{a}_{kj}}{\bar{a}_{k,k+1}} & 1 \end{pmatrix} \begin{pmatrix} \dfrac{\bar{a}_{k,k+1}}{r} & 0 \\ 0 & \dfrac{a_{k,k+1}}{r} \end{pmatrix} = \begin{pmatrix} \dfrac{\bar{a}_{k,k+1}}{r} & -\dfrac{a_{kj}}{r} \\ \dfrac{d_{k+1,k+1}}{d_{jj}} \dfrac{\bar{a}_{kj}}{r} & \dfrac{a_{k,k+1}}{r} \end{pmatrix}$$

(here, only the intersection of rows $k+1$ and j with columns $k+1$ and j are explicitly shown), has almost the form of the plane rotation $R_{k+1,j}$ [and exactly if $d_{k+1,k+1} = d_{jj}$, which is true, for example, initially when $(k,j) = (1,3)$].

The details of the modified fast Givens' algorithm are presented in Figure 2.10. The operation counts for creating the individual matrices involved are given in Table 2.5c. Unfortunately, no clever tricks can be used in the construction of S and so its operation counts are rather large. Also, although the counts for T^* are less than those in Table 2.5a for computing H, it is certainly possible to spend significant amounts of time taking complex conjugates and real parts of large expressions (of course, if A is real symmetric, then this is no longer a problem).

If only the upper triangle of A is actually represented, which is desirable for savings both in storage and in arithmetic costs, then implicit permutations using indexing arrays become quite complicated. Indeed, it seems preferable to just do explicit permutations, at least on A (S will have no particular symmetry in general, thus all of its elements need to be represented, and so it is profitable to use indexing arrays for S). Figure 2.11 displays the result of the similarity permutation $P_{ij}AP_{ij}$, and the translation of this into an explicit procedure is given in Figure 2.12. (Notice that the similarity permutation preserves Hermitianness.) The corresponding operation count (all complex conjugations) is listed in Table 2.5d.

Once a tridiagonal matrix is available, it is quite easy to compute the characteristic polynomial. The procedure is exactly the same as for an upper Hessenberg matrix, except that it is now possible to take advantage of many additional zero entries. Figure 2.13 presents the algorithm for a general tridiagonal matrix T with no zero subdiagonal entries, and the corresponding operation counts (which are only of order n^2) are given in Table 2.6b. If T is Hermitian, the products $t_{i,i+1}t_{i+1,i}$ in Figure 2.13 can be replaced by $|t_{i,i+1}|^2$. Thus, the coefficients of $c_T(\lambda)$ will be real, which is appropriate since the eigenvalues of a Hermitian matrix are guaranteed to be real and so the characteristic polynomial must be too.

2.2.4 Conversion into Smith Canonical Form

Any $m \times n$ polynomial matrix A can be transformed into what is known as Smith normal form S through the application of nonscaling elementary row and column operations as described in (2.6) (i) and (iii) [Cul72, p. 229]. S will have a structure given by $\mathrm{diag}(f_1, \ldots, f_r, 0, \ldots, 0)$, where $r \leq \min(m,n)$ and $f_i \neq 0$ divides f_{i+1}, $(i = 1, \ldots, r-1)$. This is a very useful form since $r = \mathrm{rank}\, A$ and if $m = n$, the product of the diagonal elements will be $\det A$. Traditionally, scaling transformations are applied to the Smith normal form to obtain a canonical form. If the entries of A depend on a parameter, say λ, then $f_i = f_i(\lambda)$ will be made monic, while if the entries were originally all integers, it is more useful to simply make the $\mathrm{sign}(f_i) = 1$.

The special case when $S(\lambda)$ is the Smith canonical form of the characteristic matrix $A - \lambda I$ is of particular interest here. In this situation, $S(\lambda)$ will be of full rank and so having the form $\mathrm{diag}(f_1(\lambda), \ldots, f_n(\lambda))$. In addition, since A similar to B implies that $A - \lambda I$ and $B - \lambda I$ have the same Smith canonical form [Cul72, p. 233], $c_A(\lambda) = \prod_{i=1}^{n} f_i(\lambda)$. Moreover, the minimum polynomial of A, i.e., the monic polynomial $m_A(\lambda)$ of least degree such that $m_A(A) = 0$, will be given by $f_n(\lambda)$. There are several other interesting properties of $S(\lambda)$ as well (see [Cul72, Chapter 6]).

Perform Symmetric Reduction to Hermitian Tridiagonal Form

$T = A$
$D = D'$
$S = I$
for $k = 1$ **to** $n - 2$
    for $j = k + 2$ **to** n
        $r = d_{k+1,k+1}/d_{jj}$
        $\alpha = t_{kj}/t_{k,k+1}$
        $\beta = r\alpha$
        $\bar{\beta} = r\bar{\alpha}$
        $|\alpha|^2 = \alpha\bar{\alpha}$
        $|\beta|^2 = r^2|\alpha|^2$
        $d''_{k+1,k+1} = d_{k+1,k+1} \quad ; \quad d''_{jj} = d_{jj}$
        $d_{k+1,k+1} = d''_{k+1,k+1} + |\beta|^2 d''_{jj}$
        $d_{jj} = d''_{jj} + |\alpha|^2 d''_{k+1,k+1}$
        $t_{k,k+1} = t_{k,k+1} + \bar{\beta} t_{kj}$
        $t_{kj} = 0$
        $t'_{k+1,k+1} = t_{k+1,k+1}$
        $t'_{jj} = t_{jj}$
        $t_{k+1,k+1} = t'_{k+1,k+1} + |\beta|^2 t'_{jj} + 2r\mathrm{Re}(\bar{\alpha} t_{k+1,j})$
        $t_{jj} = t'_{jj} + |\alpha|^2 t'_{k+1,k+1} - 2\mathrm{Re}(\bar{\alpha} t_{k+1,j})$
        $t_{k+1,j} = t_{k+1,j} - \alpha\beta\bar{t}_{k+1,j} - \alpha t'_{k+1,k+1} + \beta t'_{jj}$
        for $i = k + 2$ **to** $j - 1$
            $t'_{k+1,i} = t_{k+1,i} \quad ; \quad t'_{ij} = t_{ij}$
            $t_{k+1,i} = t'_{k+1,i} + \beta\bar{t}'_{ij}$
            $t_{ij} = t'_{ij} - \alpha\bar{t}'_{k+1,i}$
        for $i = j + 1$ **to** n
            $t'_{k+1,i} = t_{k+1,i} \quad ; \quad t'_{ji} = t_{ji}$
            $t_{k+1,i} = t'_{k+1,i} + \beta t'_{ji}$
            $t_{ji} = t'_{ji} - \bar{\alpha} t'_{k+1,i}$
        for $i = 2$ **to** n
            $s'_{i,k+1} = s_{i,k+1} \quad ; \quad s'_{ij} = s_{ij}$
            $s_{i,k+1} = s'_{i,k+1} + \bar{\beta} s'_{ij}$
            $s_{ij} = s'_{ij} - \alpha s'_{i,k+1}$

Figure 2.10 Modified fast Givens' algorithm to similarily transform $D'^{-1}A$, where A is an $n \times n$ Hermitian matrix and D' is an $n \times n$ diagonal matrix, into general tridiagonal form T $[T = S^{-1}(D'^{-1}A)S]$.

Transform Back into a Similarity Calculation

$S^{-1} = S^H$
$d^{-1} = 1/d_{11}$
for $j = 1$ **to** 2
    $t_{1j} = d^{-1} t_{1j}$
$s_{11}^{-1} = d^{-1} d'_{11}$
for $i = 2$ **to** n
    $d^{-1} = 1/d_{ii}$
    for $j = i - 1$ **to** $\min(i+1, n)$
        $t_{ij} = d^{-1} t_{ij}$
    for $j = 2$ **to** n
        $s_{ij}^{-1} = d^{-1} s_{ij}^{-1} d'_{jj}$

Figure 2.10 (Concluded).

Since $S(\lambda)$ is such a convenient form, it is not surprising that the transformation from $A - \lambda I$ to $S(\lambda)$ is quite computationally intensive, although it does only require rational arithmetic operations (which is a major advantage over finding the Jordan form of A, which is an even more useful matrix). The idea is to first choose an element of A of minimal degree and swap it into the (1,1) position. Next, a_{i1} and a_{1i}, $(i = 2, \ldots, n)$ are eliminated using elementary operations of the type $E_{i1}(q(\lambda))$, where $q(\lambda)$ is a polynomial. If for any element, $a_{\ell i}$ say, there is a remainder left over of degree less than a_{11}'s [for example, defined by $a_{\ell 1}(\lambda) = q(\lambda) a_{11}(\lambda) + r(\lambda)$], then make this remainder into the new (1,1) element through a swap and continue with the elimination.

Once A has been converted into the form $\mathrm{diag}(a_{11}, A')$, where a_{11} is the current value of the (1,1) element and A' is the $(n-1) \times (n-1)$ remaining submatrix, it becomes necessary to ensure that all the elements of A' are divisible by a_{11}. If some element a_{ij} in A' is not, then row i is added to row 1 and the elimination is restarted. Eventually, a_{11} will have the required property and the entire procedure can then be repeated, operating on the submatrix. Since a_{11} now divides all the a'_{ij}, this attribute will continue to hold as A' itself undergoes a similar transformation. When A is finally diagonalized by recursive application of the above process, the diagonal elements can be simply converted into their canonical representations $f_i(\lambda)$ by the elementary scalings $E_i(1/k_i)$, where k_i is the leading coefficient of the current value of $a_{ii}(\lambda)$.

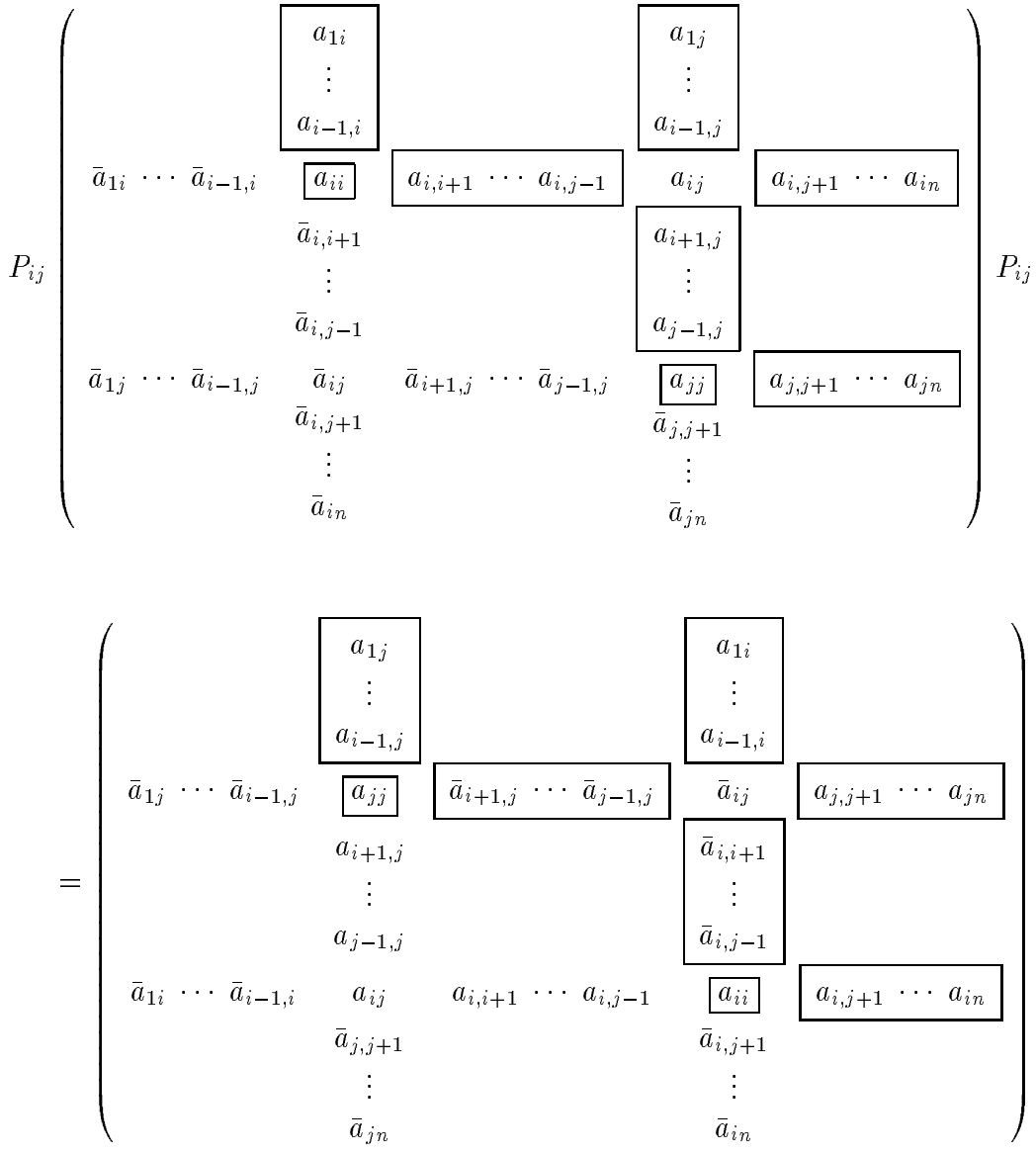


Figure 2.11 Similarity permutation P_{ij} applied to a Hermitian matrix A.

for $k = 1$ **to** $i - 1$
    $q = a_{ki}$
    $a_{ki} = a_{kj}$
    $a_{kj} = q$
$q = a_{ii}$
$a_{ii} = a_{jj}$
$a_{jj} = q$
for $k = i + 1$ **to** $j - 1$
    $q = a_{ik}$
    $a_{ik} = \bar{a}_{kj}$
    $a_{kj} = \bar{q}$
$a_{ij} = \bar{a}_{ij}$
for $k = j + 1$ **to** n
    $q = a_{ik}$
    $a_{ik} = a_{jk}$
    $a_{jk} = q$

Figure 2.12 Similarity permutation P_{ij} applied to the upper triangle of an $n \times n$ Hermitian matrix A ($A \leftarrow P_{ij} A P_{ij}$).

Form Triangular System

$u_{11} = t_{11}$
for $j = 2$ **to** n
    for $i = 1$ **to** $j - 1$
        $u_{ij} = t_{ii} u_{i,j-1}$
        if $i < j - 1$ **then**
            $u_{ij} = u_{ij} + t_{i,i+1} t_{i+1,i} u_{i+1,j-1}$
        else
            $u_{ij} = u_{ij} + t_{i,i+1} t_{i+1,i}$
        if $i > 1$ **then**
            $u_{ij} = u_{i-1,j-1} + u_{ij}$
    $u_{jj} = u_{j-1,j-1} + t_{jj}$

Solve Triangular System

$a_{n-1} = -u_{nn}$
for $i = n - 1$ **to** 1 [**step** -1]
    $$a_{i-1} = -\left(\sum_{j=i}^{n-1} u_{ij} a_j + u_{in} \right)$$

Compute Characteristic Polynomial

$$c_T(\lambda) = \lambda^n + \sum_{k=2}^{n-1} a_k \lambda^k + a_1 \lambda + a_0$$

Figure 2.13 Algorithm to compute the characteristic polynomial $c_T(\lambda)$ of a standard $n \times n$ tridiagonal matrix T.

Chapter 3

Expression Swell Analysis

A common phenomenon that occurs when performing exact computations is expression swell, in which the size of numbers and expressions involved in a calculation grow dramatically as the calculation progresses. A typical example of this phenomenon is the calculation of the roots of a third or fourth degree univariate polynomial which does not factor over the rational numbers and so the cubic or quartic formula must be used. As a particular example [Gro87], the characteristic polynomial of the 9×9 real symmetric Hankel matrix

$$\begin{pmatrix} -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 \end{pmatrix}$$

is given by

$$\lambda^9 + \lambda^8 - 40\lambda^7 - 24\lambda^6 + 240\lambda^5 + 144\lambda^4 \ ,$$

which can be factored over the rational numbers into

$$\lambda^4(\lambda + 6)(\lambda^4 - 5\lambda^3 - 10\lambda^2 + 36\lambda + 24) \ .$$

To complete the solution of the eigenvalue problem, the roots of the last factor must be extracted using the quartic formula. One of the roots is shown in Figure 3.1. An estimate of the size of this expression can be obtained by counting the number of operators and atomic operands involved in its construction. In this case, using the expression's internal representation in MACSYMA, a size of 300 was computed. The other three roots are of the same complexity and also have sizes of 300. Of course, if these roots were to be evaluated numerically, each root would be reduced to a single complex floating point number (which would have a size of 5).

$$-\frac{\sqrt{\left(18\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{2/3}-465\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{1/3}+1856\right)\sqrt{\frac{36\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{2/3}+465\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{1/3}+3712}{\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{1/3}}}+999\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{1/3}}}{\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{1/3}\sqrt{\frac{36\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{2/3}+465\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{1/3}+3712}{\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{1/3}}}}}}{6\sqrt{2}}$$

$$-\frac{\sqrt{\frac{36\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{2/3}+465\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{1/3}+3712}{\left(\frac{16\sqrt{8071}\,i}{3}+\frac{25136}{27}\right)^{1/3}}}}{12}+\frac{5}{4}$$

Figure 3.1 One root of $\lambda^4 - 5\lambda^3 - 10\lambda^2 + 36\lambda + 24$.

An important special case of the expression swell phenomenon is intermediate expression swell. This refers to a condition where, during the middle stages of a calculation, intermediate expressions can expand substantially, but the final results of the calculation are comparatively simple. A typical example here is the verification of a trigonometric or tensor identity. As an example of the latter, Figure 3.2 presents the results of MACSYMA computing the left-hand side of the Bianchi identity for a symmetric connection

$$K_j^{\ell}{}_{hk|p} + K_j^{\ell}{}_{kp|h} + K_j^{\ell}{}_{ph|k} \ ,$$

in terms of Christoffel symbols of the second kind. Here, K is the Riemann curvature tensor. This sum contains 72 terms, each of which is a product of 2 or 3 Christoffel symbols, for a total of 180 Christoffel symbols. However, upon simplifying this expression by consistently renaming the dummy indices, the simple result of zero is obtained, which verifies the identity.

Expression swell is a major problem in symbolic mathematical computations. As an expression grows in size, it takes up more and more memory and/or disk space, while also taking more and more time to be manipulated. A shortage of either of these resources can cause a computation to fail, even if the final result is known to be relatively simple. Sometimes a computation can be reorganized so that it uses less resources (or more of one and less of another) and thus succeeds where previously it failed. Sometimes nothing will help but the acquisition of more space (memory, disk, etc.) and/or an increase in the processing speed (bringing the time of a computation down to a reasonable level).

In the past, discussion of expression swell in symbolic mathematical computations has often been anecdotal. In this chapter, some quantitative results will be presented of the effects of expression swell when computing determinants and characteristic polynomials. These results will be a combination of case studies involving matrices consisting of integer and rational entries and some theoretical worst case (and other) analyses. The latter calculations will be done in terms of a worst case arithmetic which will be developed in detail. Many of the findings are quite striking, even for small matrices.

3.1 Theoretical Expression Swell Analysis

Since expression swell is such an important problem in symbolic mathematical computations, a very useful ability is to be able to predict the extent of this phenomenon during the course of a particular calculation. One way to do this is to set up an algorithm and run an extensive series of calculations under a variety of initial conditions. An analysis of the results will provide an idea (perhaps a good one) of the progress of expression growth during the calculation. In a subsequent section, a statistical survey of determinant and characteristic polynomial calculations under a limited set of initial conditions is presented. The problem with this approach for quantitatively assessing the expression swell inherent in a given calculation is that typically, a large number of sample calculations need to be made, which can be an expensive and time consuming proposition.

Generally, a better way of charting the course of expression growth is to make some kind of theoretical prediction. A common type of theoretical estimate is a bounding calculation.

$$
\begin{aligned}
&- \Gamma_{\#19}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#19}{}_{\#27}\Gamma_{p}{}^{\#27}{}_{k} + \Gamma_{j}{}^{\ell}{}_{h,\#25}\Gamma_{p}{}^{\#25}{}_{k} + \Gamma_{\#19}{}^{\ell}{}_{\#22}\Gamma_{j}{}^{\#19}{}_{h}\Gamma_{p}{}^{\#22}{}_{k} - \Gamma_{j}{}^{\ell}{}_{\#20,h}\Gamma_{p}{}^{\#20}{}_{k} \\
&+ \Gamma_{j}{}^{\ell}{}_{\#18,k}\Gamma_{p}{}^{\#18}{}_{h} + \Gamma_{\#10}{}^{\ell}{}_{k}\Gamma_{j}{}^{\#10}{}_{\#16}\Gamma_{p}{}^{\#16}{}_{h} - \Gamma_{\#10}{}^{\ell}{}_{\#12}\Gamma_{j}{}^{\#10}{}_{k}\Gamma_{p}{}^{\#12}{}_{h} - \Gamma_{j}{}^{\ell}{}_{k,\#11}\Gamma_{p}{}^{\#11}{}_{h} \\
&+ \Gamma_{j}{}^{\ell}{}_{\#9,h}\Gamma_{k}{}^{\#9}{}_{p} + \Gamma_{\#1}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#1}{}_{\#7}\Gamma_{k}{}^{\#7}{}_{p} - \Gamma_{\#1}{}^{\ell}{}_{\#3}\Gamma_{j}{}^{\#1}{}_{h}\Gamma_{k}{}^{\#3}{}_{p} - \Gamma_{j}{}^{\ell}{}_{h,\#2}\Gamma_{k}{}^{\#2}{}_{p} + \Gamma_{j}{}^{\ell}{}_{p,\#18}\Gamma_{k}{}^{\#18}{}_{h} \\
&+ \Gamma_{\#10}{}^{\ell}{}_{\#15}\Gamma_{j}{}^{\#10}{}_{p}\Gamma_{k}{}^{\#15}{}_{h} - \Gamma_{\#10}{}^{\ell}{}_{p}\Gamma_{j}{}^{\#10}{}_{\#13}\Gamma_{k}{}^{\#13}{}_{h} - \Gamma_{j}{}^{\ell}{}_{\#11,p}\Gamma_{k}{}^{\#11}{}_{h} - \Gamma_{h}{}^{\#20}{}_{k}\Gamma_{j}{}^{\ell}{}_{p,\#20} \\
&+ \Gamma_{\#9}{}^{\ell}{}_{k,h}\Gamma_{j}{}^{\#9}{}_{p} + \Gamma_{\#1}{}^{\ell}{}_{h}\Gamma_{\#7}{}^{\#1}{}_{k}\Gamma_{j}{}^{\#7}{}_{p} - \Gamma_{\#1}{}^{\ell}{}_{k}\Gamma_{\#4}{}^{\#1}{}_{h}\Gamma_{j}{}^{\#4}{}_{p} + \Gamma_{\#19}{}^{\ell}{}_{h}\Gamma_{\#27}{}^{\#19}{}_{k}\Gamma_{j}{}^{\#27}{}_{p} \\
&+ \Gamma_{\#20}{}^{\ell}{}_{k}\Gamma_{j}{}^{\#20}{}_{p,h} - \Gamma_{\#2}{}^{\ell}{}_{h,k}\Gamma_{j}{}^{\#2}{}_{p} + \Gamma_{\#19}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#19}{}_{p,k} - \Gamma_{\#19}{}^{\ell}{}_{\#26}\Gamma_{h}{}^{\#26}{}_{k}\Gamma_{j}{}^{\#19}{}_{p} \\
&+ \Gamma_{\#19}{}^{\#26}{}_{h}\Gamma_{\#26}{}^{\ell}{}_{k}\Gamma_{j}{}^{\#19}{}_{p} - \Gamma_{\#19}{}^{\#26}{}_{k}\Gamma_{\#26}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#19}{}_{p} + \Gamma_{\#19}{}^{\ell}{}_{h,k}\Gamma_{j}{}^{\#19}{}_{p} - \Gamma_{\#18}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#18}{}_{p,k} \\
&- \Gamma_{\#10}{}^{\ell}{}_{k}\Gamma_{\#16}{}^{\#10}{}_{h}\Gamma_{j}{}^{\#16}{}_{p} - \Gamma_{\#10}{}^{\ell}{}_{k}\Gamma_{j}{}^{\#10}{}_{p,h} + \Gamma_{\#10}{}^{\#15}{}_{h}\Gamma_{\#15}{}^{\ell}{}_{k}\Gamma_{j}{}^{\#10}{}_{p} - \Gamma_{\#10}{}^{\#15}{}_{k}\Gamma_{\#15}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#10}{}_{p} \\
&- \Gamma_{\#10}{}^{\ell}{}_{k,h}\Gamma_{j}{}^{\#10}{}_{p} + \Gamma_{h}{}^{\#9}{}_{p}\Gamma_{j}{}^{\ell}{}_{k,\#9} - \Gamma_{\#9}{}^{\ell}{}_{p}\Gamma_{j}{}^{\#9}{}_{k,h} - \Gamma_{\#1}{}^{\ell}{}_{h}\Gamma_{\#7}{}^{\#1}{}_{p}\Gamma_{j}{}^{\#7}{}_{k} \\
&- \Gamma_{\#19}{}^{\ell}{}_{h}\Gamma_{\#27}{}^{\#19}{}_{p}\Gamma_{j}{}^{\#27}{}_{k} + \Gamma_{\#25}{}^{\ell}{}_{h,p}\Gamma_{j}{}^{\#25}{}_{k} + \Gamma_{\#19}{}^{\ell}{}_{p}\Gamma_{\#23}{}^{\#19}{}_{h}\Gamma_{j}{}^{\#23}{}_{k} - \Gamma_{\#20}{}^{\ell}{}_{p,h}\Gamma_{j}{}^{\#20}{}_{k} \\
&+ \Gamma_{\#10}{}^{\ell}{}_{p}\Gamma_{\#13}{}^{\#10}{}_{h}\Gamma_{j}{}^{\#13}{}_{k} + \Gamma_{\#11}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#11}{}_{k,p} + \Gamma_{\#10}{}^{\ell}{}_{p}\Gamma_{j}{}^{\#10}{}_{k,h} - \Gamma_{\#10}{}^{\#12}{}_{h}\Gamma_{\#12}{}^{\ell}{}_{p}\Gamma_{j}{}^{\#10}{}_{k} \\
&+ \Gamma_{\#10}{}^{\#12}{}_{p}\Gamma_{\#12}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#10}{}_{k} + \Gamma_{\#10}{}^{\ell}{}_{p,h}\Gamma_{j}{}^{\#10}{}_{k} - \Gamma_{\#1}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#1}{}_{k,p} + \Gamma_{\#1}{}^{\ell}{}_{\#6}\Gamma_{h}{}^{\#6}{}_{p}\Gamma_{j}{}^{\#1}{}_{k} \\
&- \Gamma_{\#1}{}^{\#6}{}_{h}\Gamma_{\#6}{}^{\ell}{}_{p}\Gamma_{j}{}^{\#1}{}_{k} + \Gamma_{\#1}{}^{\#6}{}_{p}\Gamma_{\#6}{}^{\ell}{}_{h}\Gamma_{j}{}^{\#1}{}_{k} - \Gamma_{\#1}{}^{\ell}{}_{h,p}\Gamma_{j}{}^{\#1}{}_{k} + \Gamma_{\#1}{}^{\ell}{}_{k}\Gamma_{\#4}{}^{\#1}{}_{p}\Gamma_{j}{}^{\#4}{}_{h} \\
&- \Gamma_{\#25}{}^{\ell}{}_{k}\Gamma_{j}{}^{\#25}{}_{h,p} - \Gamma_{\#19}{}^{\ell}{}_{p}\Gamma_{\#23}{}^{\#19}{}_{k}\Gamma_{j}{}^{\#23}{}_{h} + \Gamma_{\#2}{}^{\ell}{}_{p}\Gamma_{j}{}^{\#2}{}_{h,k} - \Gamma_{\#19}{}^{\ell}{}_{p}\Gamma_{j}{}^{\#19}{}_{h,k} \\
&+ \Gamma_{\#19}{}^{\#22}{}_{k}\Gamma_{\#22}{}^{\ell}{}_{p}\Gamma_{j}{}^{\#19}{}_{h} - \Gamma_{\#19}{}^{\#22}{}_{p}\Gamma_{\#22}{}^{\ell}{}_{k}\Gamma_{j}{}^{\#19}{}_{h} - \Gamma_{\#19}{}^{\ell}{}_{p,k}\Gamma_{j}{}^{\#19}{}_{h} + \Gamma_{\#18}{}^{\ell}{}_{p,k}\Gamma_{j}{}^{\#18}{}_{h} \\
&+ \Gamma_{\#10}{}^{\ell}{}_{k}\Gamma_{\#16}{}^{\#10}{}_{p}\Gamma_{j}{}^{\#16}{}_{h} - \Gamma_{\#10}{}^{\ell}{}_{p}\Gamma_{\#13}{}^{\#10}{}_{k}\Gamma_{j}{}^{\#13}{}_{h} - \Gamma_{\#11}{}^{\ell}{}_{k,p}\Gamma_{j}{}^{\#11}{}_{h} + \Gamma_{\#1}{}^{\ell}{}_{k}\Gamma_{j}{}^{\#1}{}_{h,p} \\
&+ \Gamma_{\#1}{}^{\#3}{}_{k}\Gamma_{\#3}{}^{\ell}{}_{p}\Gamma_{j}{}^{\#1}{}_{h} - \Gamma_{\#1}{}^{\#3}{}_{p}\Gamma_{\#3}{}^{\ell}{}_{k}\Gamma_{j}{}^{\#1}{}_{h} + \Gamma_{\#1}{}^{\ell}{}_{k,p}\Gamma_{j}{}^{\#1}{}_{h} - \Gamma_{\#1}{}^{\ell}{}_{k}\Gamma_{h}{}^{\#4}{}_{p}\Gamma_{j}{}^{\#1}{}_{\#4} \\
&+ \Gamma_{h}{}^{\#25}{}_{k}\Gamma_{j}{}^{\ell}{}_{\#25,p} + \Gamma_{\#19}{}^{\ell}{}_{p}\Gamma_{h}{}^{\#23}{}_{k}\Gamma_{j}{}^{\#19}{}_{\#23} - \Gamma_{h}{}^{\#2}{}_{p}\Gamma_{j}{}^{\ell}{}_{\#2,k}
\end{aligned}
$$

Figure 3.2 Left-hand side of the Bianchi identity for a symmetric connection expanded in terms of Christoffel symbols of the second kind.

For an expression swell analysis of an algorithm, there are two important kinds of bounding calculations that can be performed. One involves worst case behavior in which the expression size is maximized for the outcome of each mathematical operation (the results of an addition, of applying a function to its arguments, of applying an operator to a function, etc.), while the other involves best case behavior in which the expression size is minimized for the outcome of each mathematical operation. In general, determining worst or best case behavior for expression swell of a general operation on general operands is extremely difficult. Even just the notion of expression size is not clearly defined. How is the size of an expression to be judged? Does it mean the number of terms in the expression, the number of operators and atomic operands, the number of bytes in some representation of the expression, or some other measure? Or perhaps, different measures are appropriate at different times.

Matters are considerably simplified if only the expression swell analysis of algorithms involving infinite precision integer and rational number expressions is considered. In these cases, expression size can be well-defined. For an integer, the number of digits in its base β ($\beta \in \mathbf{Z} \geq 2$) representation is a good indicator of its size. (One could also take the absolute value of the number itself to represent its size, but this definition does not provide enough generality to be useful—it is no easier nor much more informative to do an analysis using this definition than it is to just do the calculation with the original integers.) This definition of the size of an integer is essentially a logarithmic measure. Indeed, the number of base β digits comprising an integer n, $\mathcal{N}_\beta(n)$, can alternatively be defined by

$$\mathcal{N}_\beta(n) \equiv \begin{cases} \lfloor \log_\beta |n| \rfloor + 1, & n \neq 0 \\ 0, & n = 0 \end{cases} \tag{3.1}$$

where $\lfloor\ \rfloor$ denotes the floor function. (In practice, the common case of $\mathcal{N}_{10}(n)$ will be computed directly by actually counting the number of decimal digits in n.) Similarly, the size of a rational number (which need not be in lowest terms) can be defined as the sum of the sizes of its numerator and denominator (or possibly as the maximum of these two values). For example, the size (base 10) of $-\frac{327}{1955}$ would be $\mathcal{N}_{10}(-327) + \mathcal{N}_{10}(1955) = 3 + 4 = 7$.

In order to deal only with integer and rational number expressions in an algorithm, the operations permitted on these quantities must be restricted as well. In particular, the expression swell analysis of an algorithm with solely integer inputs can proceed in a simple manner only if all the operations in the algorithm are integer or at least rational number preserving. Therefore, confining the scope of algorithms studied to only those that just use rational arithmetic operations (except perhaps at the last stage, such as when computing a polynomial from its coefficients) will make expression swell analysis manageable for those problems with such algorithms. A large number of the calculations from linear algebra fall into this category.

In the following subsections, an expression swell arithmetic is developed. This arithmetic will operate on numbers that represent the number of digits in classes of integers and rational numbers. Integer preserving operations will be treated separately from rational number preserving ones. This division is made since exact rational number arithmetic exhibits a more complex behavior than exact integer arithmetic. After developing this expression swell

arithmetic and discussing an implementation of it in MACSYMA, examples of its use and comparisons of its predictions with the results of actual calculations will be presented in later sections.

3.1.1 Integer Calculations

Here, a worst case expression swell arithmetic will be developed for certain integer preserving operations. These include the rational arithmetic operations of addition, negation, multiplication, exact division and exponentiation to a nonnegative integer power. Also included are absolute values, exact square roots and greatest common divisors (GCDs). Precisely, worst case behavior under an integer preserving operation means that the result will contain the greatest number of digits that are possible (GCDs are a special case and will be discussed below).

To begin this development, note that $\mathcal{N}_\beta$ partitions the integers into an infinite set of equivalence classes. Each equivalence class can be designated by a nonnegative integer: $\bar{0}, \bar{1}, \bar{2}, \ldots$ Thus, $\overline{n}$ will represent the set of integers that have exactly n base β digits. For example, $\bar{1} = \{-(\beta-1), \ldots, -1, 1, \ldots, \beta-1\}$ (as a special case, $\bar{0} = \{0\}$). The intent behind classifying the integers in this way is to permit the analysis of algorithms where the inputs have specified numbers of digits (are members of specified equivalence classes). The analyses will produce upper bounds on the number of digits in the results for any representatives chosen from the respective equivalence classes that are used as inputs to the algorithms.

Operations on the equivalence classes defined by $\mathcal{N}_\beta$ can yield one of two types of results. An operation can produce an integer value. An obvious example is $\mathcal{N}_\beta(\overline{n}) \equiv n$. That is, the number of digits in any representative of $\overline{n}$ is no greater than n (actually, is exactly n in this simple usage). This defines the operation of $\mathcal{N}_\beta$ on an equivalence class. The other possibility is that an operation on one or more equivalence classes will itself yield an equivalence class. For example, suppose

$$f(\overline{n}_1, \overline{n}_2, \ldots) \equiv \overline{g(n_1, n_2, \ldots)} \; .$$

Then, for worst case expression swell behavior, this will mean that $g(n_1, n_2, \ldots)$ will bound the number of digits in $f(n_1', n_2', \ldots)$ over all possible choices of $n_1', n_2', \ldots$ such that $\mathcal{N}_\beta(n_1') = n_1, \mathcal{N}_\beta(n_2') = n_2, \ldots$ Another way of writing this is

$$\mathcal{N}_\beta(f(\overline{n}_1, \overline{n}_2, \ldots)) \leq \mathcal{N}_\beta(\overline{g(n_1, n_2, \ldots)}) = g(n_1, n_2, \ldots) \; . \tag{3.2}$$

For example, if $\beta = 10$, multiplying the equivalence class $\bar{2}$ with itself (i.e., multiplying together all possible pairs of 2-digit integers) will produce a set of 3 and 4-digit integers (ranging from $\pm 10 \cdot \pm 10 = \pm 100$ to $\pm 99 \cdot \pm 99 = \pm 9801$). Therefore, the product of $\bar{2}$ with itself will be defined to be $\bar{4}$ since no element of the product set will contain more than 4 digits. Note that the result could have been defined as $\bar{5}$ or $\bar{6}$ or ..., but these choices would not have been as good since they do not yield as strict a bound on the maximum number of digits as does $\bar{4}$.

The best choice for g in the above formulation will be a function that actually takes on the value of $\mathcal{N}_\beta(f(\overline{n}_1, \overline{n}_2, \ldots))$ [i.e., that maximizes the number of digits in $f(n_1', n_2', \ldots)$,

where $\mathcal{N}_\beta(n_i') = n_i$ for $i = 1, 2, \ldots]$. For unary and binary operations, this is not difficult. Consider the unary operations first.

Theorem 3.1 The unary operations of integer absolute value and negation, when applied to the equivalence class $\overline{n}$, possess optimal bounding functions given by

(i) $|\overline{n}| = \overline{n}$,
(ii) $\ominus\overline{n} = \overline{n}$.

This says that taking the absolute value or the negation of an integer does not change the number of digits it contains. The $\ominus$ is a "maximal expression swell" operator. This notation has been introduced to emphasize the distinction between worst case expression swell arithmetic and traditional integer arithmetic.

Theorem 3.2 The binary operations of integer addition, subtraction, multiplication and exact division, when applied to the equivalence classes $\overline{n}_1$ and $\overline{n}_2$, where $n_1 > 0$ and $n_2 > 0$, possess optimal bounding functions given by

(i) $\overline{n}_1 \oplus \overline{n}_2 = \overline{\max(n_1, n_2) + 1}$,
(ii) $\overline{n}_1 \ominus \overline{n}_2 = \overline{\max(n_1, n_2) + 1}$,
(iii) $\overline{n}_1 \odot \overline{n}_2 = \begin{cases} \overline{n_1 + n_2} & (n_1 > 1 \text{ and } n_2 > 1 \text{ for } \beta = 2) \\ \overline{\max(n_1, n_2)} & (\beta = 2 \text{ and } \min(n_1, n_2) = 1), \end{cases}$
(iv) $\overline{n}_1 \overset{\text{exact}}{\oslash} \overline{n}_2 = \overline{n_1 - n_2 + 1} \quad (n_1 \geq n_2)$.

If $n_1 = 0$ then the right-hand sides of (i) and (ii) become $\overline{n}_2$ and the right-hand sides of (iii) and (iv) become $\bar{0}$. If $n_2 = 0$ then the right-hand sides of (i)–(iv) are respectively, $\overline{n}_1, \overline{n}_1, \bar{0}$ and undefined.

Proof: Worst case expression swell addition and subtraction are equivalent since in the worst case for regular subtraction, the two operands will be of opposite signs and so it will really be an addition of two terms of the same sign, which is the worst case for regular addition. Symbolically, this can be stated

$$\overline{n}_1 \ominus \overline{n}_2 = \overline{n}_1 \oplus (\ominus\overline{n}_2) = \overline{n}_1 \oplus \overline{n}_2 \ .$$

The special cases involving $\bar{0}$ are straightforward, so assume $n_1 > 0$ and $n_2 > 0$. The worst cases for addition, multiplication and exact division can be achieved using all positive operands, so here $\overline{n}$ can be represented by the set of integers $\{\beta^{n-1}, \ldots, \beta^n - 1\}$ with $\beta \geq 2$. To demonstrate that $\overline{g(n_1, n_2)}$ optimally bounds $f(\overline{n}_1, \overline{n}_2)$, it is sufficient to show that the following relationship holds:

$$\beta^{g(n_1,n_2)-1} \leq f(n_1^*, n_2^*) \leq \beta^{g(n_1,n_2)} - 1 \ .$$

n_1^* and n_2^* are chosen so as to maximize $f(n_1', n_2')$, where $\mathcal{N}_\beta(n_1') = n_1$ and $\mathcal{N}_\beta(n_2') = n_2$. The right inequality expresses that $g(n_1, n_2)$ is indeed an upper bound of $\mathcal{N}_\beta(f(\overline{n}_1, \overline{n}_2))$, while the left inequality indicates that the bound is optimal so that $\mathcal{N}_\beta(f(\overline{n}_1, \overline{n}_2)) = g(n_1, n_2)$.

In the worst case for addition, n_1^* and n_2^* will be maximal, i.e., $n_1^* = \beta^{n_1} - 1$ and $n_2^* = \beta^{n_2} - 1$. Now,

$$\begin{aligned}(\beta^{n_1} - 1) + (\beta^{n_2} - 1) &= (\beta^{\max(n_1,n_2)} - 1) + (\beta^{\min(n_1,n_2)} - 1) \\ &\leq 2(\beta^{\max(n_1,n_2)} - 1) \leq \beta(\beta^{\max(n_1,n_2)} - 1) \\ &< \beta^{\max(n_1,n_2)+1} - 1 \ ,\end{aligned}$$

so $\max(n_1, n_2) + 1$ bounds $\mathcal{N}_\beta(\overline{n}_1 \oplus \overline{n}_2)$. Moreover,

$$(\beta^{\max(n_1,n_2)} - 1) + (\beta^{\min(n_1,n_2)} - 1) \geq \beta^{\max(n_1,n_2)}$$

since $\beta^{\min(n_1,n_2)} \geq 2$, thus this bound is optimal.

n_1^* and n_2^* will be defined similarly for worst case multiplication. Therefore,

$$(\beta^{n_1} - 1)(\beta^{n_2} - 1) = \beta^{n_1+n_2} - (\beta^{n_1} + \beta^{n_2}) + 1 < \beta^{n_1+n_2} - 1$$

shows that $\mathcal{N}_\beta(\overline{n}_1 \circledcirc \overline{n}_2)$ is bounded by $n_1 + n_2$. If

$$\beta^{n_1+n_2} - (\beta^{n_1} + \beta^{n_2}) + 1 \geq \beta^{n_1+n_2-1}$$

or

$$(\beta - 1)\beta^{n_1+n_2-1} \geq \beta^{n_1} + \beta^{n_2} - 1 \ , \tag{3.3}$$

then this bound is optimal. Note that

$$\beta^{n_1} + \beta^{n_2} - 1 = \beta^{\max(n_1,n_2)} + \beta^{\min(n_1,n_2)} - 1 < 2\beta^{\max(n_1,n_2)} \ . \tag{3.4}$$

Therefore, for $\beta \geq 3$,

$$2\beta^{\max(n_1,n_2)} \leq 2\beta^{n_1+n_2-1} \leq (\beta - 1)\beta^{n_1+n_2-1}$$

will be valid and (3.3) will hold as long as

$$\max(n_1, n_2) \leq n_1 + n_2 - 1 = \max(n_1, n_2) + \min(n_1, n_2) - 1$$

or $\min(n_1, n_2) \geq 1$, which is true by assumption.

For $\beta = 2$ [using (3.4) for the first inequality],

$$2^{n_1} + 2^{n_2} - 1 < 2^{\max(n_1,n_2)+1} \leq 2^{n_1+n_2-1}$$

will be correct and (3.3) will be satisfied whenever

$$\max(n_1, n_2) + 1 \leq n_1 + n_2 - 1 = \max(n_1, n_2) + \min(n_1, n_2) - 1$$

or $\min(n_1, n_2) \geq 2$. Note that $2^1 - 1 = 1$, hence

$$(2^{n_1} - 1)(2^{n_2} - 1) = (2^{\max(n_1,n_2)} - 1)(2^{\min(n_1,n_2)} - 1) = 2^{\max(n_1,n_2)} - 1$$

for the special situation when $\min(n_1, n_2) = 1$.

Finally, the worst case for exact division (with $n_1 \geq n_2$) occurs when n_1^* is as large as possible and n_2^* is as small as possible, so choose $n_1^* = \beta^{n_1} - 1$ and $n_2^* = \beta^{n_2 - 1}$. Now,

$$\frac{\beta^{n_1} - 1}{\beta^{n_2 - 1}} < \frac{\beta^{n_1}}{\beta^{n_2 - 1}} = \beta^{n_1 - n_2 + 1}$$

and since the result of the division must be an integer, this implies that

$$\frac{\beta^{n_1} - 1}{\beta^{n_2 - 1}} \leq \beta^{n_1 - n_2 + 1} - 1 \ ,$$

and so $n_1 - n_2 + 1$ bounds $\mathcal{N}_\beta(\overline{n}_1 \overset{\text{exact}}{\oslash} \overline{n}_2)$. Furthermore, this bound is optimal as can be seen from the following sequence of equivalent inequalities:

$$\frac{\beta^{n_1} - 1}{\beta^{n_2 - 1}} \geq \beta^{n_1 - n_2} \iff \beta^{n_1} - 1 \geq \beta^{n_1 - 1} \iff (\beta - 1)\beta^{n_1 - 1} \geq 1 \ .$$

(Both of the factors on the left-hand side of the last inequality are ≥ 1 since $\beta \geq 2$ and n_1 is at least 1.) ∎

To make the ideas presented in the above theorem more concrete, it is helpful to consider some decimal examples. For instance, the worst case of adding an n_1-digit number to an n_2-digit number occurs when all the digits in the larger number are 9's. Then adding a smaller or equal sized number of the same sign will at worst (in this particular case, will always) produce a result with one more digit than the original integer. For example, the validity of the assertion $\bar{4} \oplus \bar{3} = \bar{5}$ can be seen by looking at a worst case calculation like $9999 + 999 = 10998$. For multiplication, the worst case of expression size growth can be exhibited when both numbers consist of all 9's. In this case, the multiplication becomes $(10^{n_1} - 1)(10^{n_2} - 1) = 10^{n_1 + n_2} - (10^{n_1} + 10^{n_2}) + 1$. The two middle terms will always bring the final number of digits in the result down to $n_1 + n_2$. Thus, the worst case $9999 \cdot 999 = 9989001$ establishes the validity of writing $\bar{4} \odot \bar{3} = \bar{7}$. Finally, as an example of worst case exact division, it is easy to demonstrate that $\bar{4} \overset{\text{exact}}{\oslash} \bar{3} = \bar{2}$ by noting that the worst situations are $9900 \div 100 = 99$ and $9999 \div 101 = 99$.

Like their integer arithmetic counterparts, worst case expression swell addition and multiplication are commutative. However, the binary operation $\oplus$, at least, is not associative, unlike ordinary addition. A simple example shows this. $(\bar{1} \oplus \bar{2}) \oplus \bar{3} = \bar{3} \oplus \bar{3} = \bar{4}$ while $\bar{1} \oplus (\bar{2} \oplus \bar{3}) = \bar{1} \oplus \bar{4} = \bar{5}$. Remember that the motivation behind developing an expression swell arithmetic is to establish bounds on the rate of expression growth. Therefore, if one way of ordering operations produces a tighter bound on expression swell than other methods, then this way is to be preferred. The above example suggests that the optimal arrangement for adding equivalence classes is to have them ordered in terms of increasing size and then proceed from left to right [i.e., add $\overline{n}_1, \overline{n}_2, \overline{n}_3$, where $n_1 \leq n_2 \leq n_3$, by $(\overline{n}_1 \oplus \overline{n}_2) \oplus \overline{n}_3$]. $\odot$ is associative as can be easily seen and so needs no further elaboration.

$\oplus$ and $\odot$ can also be thought of as n-ary operators (operators that can be applied to an indefinite number of operands). $\odot$ can be easily extended to handle an arbitrary number of factors due to its associativity when considered as a binary operator. Extending $\oplus$ is more difficult; however, some properties can be stated.

Theorem 3.3 The n-ary operations of integer multiplication and addition, when applied to the equivalence classes $\overline{n}_1, \ldots, \overline{n}_k$ and $\overline{n}$, respectively, where $n_i > 0$ for all i and $n > 0$, possess bounding functions given by

$$\text{(i)} \quad \overline{n}_1 \odot \overline{n}_2 \odot \cdots \odot \overline{n}_k = \overline{n_1 + n_2 + \cdots + n_k} \qquad (k > 0),$$

$$\text{(ii)} \quad k \odot \overline{n} \equiv \underbrace{\overline{n} \oplus \overline{n} \oplus \cdots \oplus \overline{n}}_{k \text{ terms}} = \overline{n + \mathcal{N}_\beta(k-1)} \quad (k > 0).$$

If any of the $n_i = 0$ then the right-hand side of (i) becomes $\bar{0}$. If $n = 0$ then the right-hand side of (ii) is $\bar{0}$.

Proof: Again, the special cases involving $\bar{0}$ are clear, so assume all the $n_i > 0$ and $n > 0$. The first bound can be demonstrated using already established properties of worst case expression swell arithmetic. Thus,

$$\overline{n}_1 \odot \overline{n}_2 \odot \overline{n}_3 = (\overline{n}_1 \odot \overline{n}_2) \odot \overline{n}_3 = \overline{n_1 + n_2} \odot \overline{n}_3 = \overline{n_1 + n_2 + n_3}$$

is the first inductive step of a sequence that leads to formula (i).

Now, the left-hand side of (ii) defines a shorthand notation for the n-ary sum that follows. The bound on the right can then be obtained by appealing to the definition of $\mathcal{N}_\beta$ and the method implied in (3.2), noting that the actual worst case representative of $k \odot \overline{n}$ is given by $k(\beta^n - 1)$. First, observe that

$$\begin{aligned} \mathcal{N}_\beta(k[\beta^n - 1]) &= \lfloor \log_\beta(k[\beta^n - 1]) \rfloor + 1 = \lfloor \log_\beta k + \log_\beta(\beta^n - 1) \rfloor + 1 \\ &\leq \lfloor \log_\beta k + \log_\beta \beta^n \rfloor + 1 = n + (\lfloor \log_\beta k \rfloor + 1) \\ &= n + \mathcal{N}_\beta(k) \, . \end{aligned}$$

To show the result given in (ii), note that $\mathcal{N}_\beta(k) = \mathcal{N}_\beta(k-1)$ except when $k = \beta^m$ ($m \in \mathbf{Z} \geq 0$). Therefore,

$$\mathcal{N}_\beta(k[\beta^n - 1]) \leq n + \mathcal{N}_\beta(k-1)$$

is verified for all $k \neq \beta^m$. The situation for $k = \beta^m$ is

$$\begin{aligned} \mathcal{N}_\beta(\beta^m[\beta^n - 1]) &= \lfloor \log_\beta(\beta^m[\beta^n - 1]) \rfloor + 1 = \lfloor \log_\beta \beta^m + \log_\beta(\beta^n - 1) \rfloor + 1 \\ &= m + (\lfloor \log_\beta(\beta^n - 1) \rfloor + 1) = m + \mathcal{N}_\beta(\beta^n - 1) \\ &= m + n = n + \mathcal{N}_\beta(\beta^m - 1) \, , \end{aligned}$$

which completes the proof. ∎

To see that the bound on the n-ary sum is not optimal, consider what happens in the worst case for decimal n-digit numbers. For instance, suppose $n = 2$ then Table 3.1 shows

k	$k \cdot 99$	$\mathcal{N}_{10}(k \cdot 99)$	$\mathcal{N}_{10}(k \circledcirc \bar{2})$
1	99	2	2
2	198	3	3
10	990	3	3
11	1089	4	4
100	9900	4	4
101	9999	4	5
102	10098	5	5

Table 3.1 Actual versus theoretical worst case for base 10 n-ary addition.

the results of actual worst case sums for various values of k as well as, in the last column, the theoretical quantities $\mathcal{N}_{10}(k \circledcirc \bar{2})$. Notice that eventually $\mathcal{N}_{10}(k \circledcirc \bar{2})$ will occasionally exceed the actual worst case, but this occurs only for relatively large values of k.

The value of k for which the bound given above first exceeds the actual number of digits of the worst case can be computed in general. $n + \mathcal{N}_\beta(k-1)$ increments by one whenever $k = \beta^m + 1$ for $m \in \mathbf{Z} \geq 0$. At this value, the bound becomes

$$\mathcal{N}_\beta([\beta^m + 1] \circledcirc \overline{n}) = n + \mathcal{N}_\beta(\beta^m) = n + m + 1 \ .$$

The corresponding actual worst case is given by

$$(\beta^m + 1)(\beta^n - 1) = \beta^{m+n} + \beta^n - \beta^m - 1 = (\beta^{m+n} - 1) + (\beta^n - \beta^m) \ .$$

If $m < n$ then this will be an integer of $n + m + 1$ (base β) digits. When $m = n$ (and $k = \beta^n + 1$), however, the number of digits will drop to $n + m$ and so will be overpredicted (by one) by the bound calculated above.

The n-ary $\bigoplus$ operator introduced above is distinct from the binary version, although both versions do produce the same result of $\overline{n+1}$ for the common case of $\overline{n} \bigoplus \overline{n}$. For example, using n-ary $\bigoplus$, $\overline{n} \bigoplus \overline{n} \bigoplus \overline{n} = \overline{n+1}$ ($\beta \geq 3$), while under binary $\bigoplus$, this becomes $(\overline{n} \bigoplus \overline{n}) \bigoplus \overline{n} = \overline{n+1} \bigoplus \overline{n} = \overline{n+2}$. Again, the tighter bound is to be preferred when doing expression swell arithmetic, so when adding like terms, the n-ary operator will always be used. The infix representation of n-ary $\bigoplus$ is somewhat misleading, hence the alternative notation of $k \circledcirc \overline{n}$ for a sum of k like terms will be adopted whenever possible. This notation has the property that $(k_1 \circledcirc \overline{n}) \bigoplus (k_2 \circledcirc \overline{n}) = (k_1 + k_2) \circledcirc \overline{n}$ since the parenthesized terms on the left are really sums and for estimating bounds, it is best to make the whole expression into a single n-ary sum.

Whole number exponentiation can be defined in terms of n-ary multiplication. The bounds will be a special case of those derived above.

Corollary 3.4 The binary operation of integer exponentiation to a nonnegative integer power, when applied to the equivalence class $\overline{n}$, possesses a bounding function given by

(i) $\overline{n} \textcircled{\uparrow} 0 = \overline{1} \quad (n > 0)$,
(ii) $\overline{n} \textcircled{\uparrow} k \equiv \underbrace{\overline{n} \textcircled{\bullet} \overline{n} \textcircled{\bullet} \cdots \textcircled{\bullet} \overline{n}}_{k \text{ factors}} = \overline{kn} \quad (k > 0)$.

If $n = 0$ then the right-hand side of (i) becomes undefined.

Like n-ary addition, the results of expression swell exponentiation will also eventually exceed the actual worst case, but again, only for relatively large values of k. In particular, suppose $n > 0$ and consider

$$\begin{aligned} \mathcal{N}_\beta([\beta^n - 1]^k) &= \lfloor \log_\beta (\beta^n - 1)^k \rfloor + 1 = \left\lfloor \log_\beta \beta^{nk} \left(1 - \frac{1}{\beta^n}\right)^k \right\rfloor + 1 \\ &= \left\lfloor \log_\beta \beta^{nk} + \log_\beta \left(1 - \frac{1}{\beta^n}\right)^k \right\rfloor + 1 = nk + 1 + \left\lfloor \log_\beta \left(1 - \frac{1}{\beta^n}\right)^k \right\rfloor . \end{aligned}$$

The term on the far right of the last expression will be no greater than -1. Moreover, if

$$-(\ell + 1) \le \log_\beta \left(1 - \frac{1}{\beta^n}\right)^k < -\ell$$

for $\ell \in \mathbf{Z} \ge 0$, then the theoretical prediction, kn, will exceed the actual result by exactly ℓ digits. The right inequality can be transformed into the following equivalent relationships:

$$\left(1 - \frac{1}{\beta^n}\right)^k < \beta^{-\ell} \Longleftrightarrow k \ln \left(1 - \frac{1}{\beta^n}\right) < -\ell \ln \beta \Longleftrightarrow -k \ln \left(1 - \frac{1}{\beta^n}\right) > \ell \ln \beta \, .$$

Since $\beta^n \ge 2$, this last inequality has the Taylor expansion

$$-k \left[-\frac{1}{\beta^n} - O\left(\frac{1}{\beta^{2n}}\right) \right] > \ell \ln \beta \qquad \text{or} \qquad k > \ell \beta^n \ln \beta - O\left(\frac{1}{\beta^n}\right) \, .$$

Thus, the actual worst case number of digits will diverge from the theoretical bound starting at $k \approx \beta^n \ln \beta$. Table 3.2 shows results for $\beta = 10$. The values in the last column converge quite clearly to $\ln 10 = 2.30258\ldots$

Two more worst case expression swell operations need to be defined to complete the theoretical framework which will allow algorithms involving integer arithmetic to be analyzed.

Theorem 3.5 The unary operation of exact integer square root, when applied to the equivalence class $\overline{n}$, possesses an optimal bounding function given by

$$\sqrt{\overline{n}} = \overline{\left\lceil \frac{n}{2} \right\rceil} = \begin{cases} \overline{\left(\frac{n}{2}\right)} & (n \text{ even}) \\ \overline{\left(\frac{n+1}{2}\right)} & (n \text{ odd}), \end{cases}$$

where $\lceil \ \rceil$ denotes the ceiling function.

n	k	$\mathcal{N}_{10}([10^n-1]^k)$	$\mathcal{N}_{10}(\overline{n} \circledcirc k)$	$k/10^n$
1	22	21	22	2.2000
2	230	459	460	2.3000
3	2302	6905	6906	2.3020
4	23025	92099	92100	2.3025

Table 3.2 Actual versus theoretical worst case for base 10 exponentiation. k is chosen to be the first k such that $\mathcal{N}_{10}([10^n-1]^k) < \mathcal{N}_{10}(\overline{n} \circledcirc k) = kn$.

Proof: Consider the even case first. The demonstration for $n = 0$ is clear so assume $n \geq 2$. The worst case for taking a square root of an n-digit number occurs for $n^* = \beta^n - 1$. Now, $\sqrt{\beta^n - 1} < \sqrt{\beta^n} = \beta^{n/2}$, which implies that $\sqrt{\beta^n - 1} \leq \beta^{n/2} - 1$ since the result must be an integer. This establishes $n/2$ as a bound on $\mathcal{N}_\beta(\sqrt{\overline{n}})$. This bound is optimal as can be seen from the following equivalent inequalities:

$$\beta^{(n/2)-1} \leq \sqrt{\beta^n - 1} \Longleftrightarrow \beta^{n-2} \leq \beta^n - 1 \Longleftrightarrow \beta^{n-2}(\beta^2 - 1) \geq 1$$

(note $\beta \geq 2$). If n is odd (and so $n \geq 1$),

$$\sqrt{\beta^n - 1} < \sqrt{\beta^n} < \sqrt{\beta^{n+1}} = \beta^{(n+1)/2} ,$$

hence, $\sqrt{\beta^n - 1} \leq \beta^{(n+1)/2} - 1$. This bound is optimal, also, since

$$\beta^{(n-1)/2} \leq \sqrt{\beta^n - 1} \Longleftrightarrow \beta^{n-1} \leq \beta^n - 1 \Longleftrightarrow \beta^{n-1}(\beta - 1) \geq 1 ,$$

which is clearly true. ∎

Definition 3.6 The n-ary operation of integer greatest common divisor, when applied to the equivalence classes $\overline{n}_1, \ldots, \overline{n}_k$, is given by

$\gcd(\overline{n}_1, \overline{n}_2, \ldots, \overline{n}_k) \equiv \bar{1} \quad (k > 1)$.

The definition of the GCD seems at first inconsistent with the other operations, but it is really quite appropriate for worst case expression swell behavior. Typically in symbolic mathematical calculations, GCDs are used to find the common factors of sets of expressions in order to simplify subsequent computations in some way (e.g., removing the common factors from the numerator and denominator of a rational number, reducing it to lowest terms). Therefore, expression swell will be maximized if the quantities involved in a calculation are all relatively prime with respect to one another. This implies that the GCD is always one. As an example, consider the least common multiple (LCM) of an n_1-digit integer and an n_2-digit integer. In the worst case, the LCM becomes

$$\mathrm{lcm}(\overline{n}_1, \overline{n}_2) \equiv (\overline{n}_1 \circledcirc \overline{n}_2) \overset{\text{exact}}{\circledcirc\!\!\div} \gcd(\overline{n}_1, \overline{n}_2) = \overline{n_1 + n_2} \overset{\text{exact}}{\circledcirc\!\!\div} \bar{1} = \overline{n_1 + n_2} .$$

In a manner similar to the above, a best case expression swell arithmetic for integer preserving operations can be developed. Best case behavior under an integer preserving operation means that the result will contain the least number of digits that are possible. Hence, this arithmetic will provide a lower bound on expression growth during a computation. For example, under these circumstances, the definition of the GCD would yield $\min(n_1, n_2, \ldots, n_k)$ [assuming $n_i > 0$ for all i]. However, the best case for addition and subtraction of integers with the same number of digits is zero (catastrophic cancellation), so the results of such an analysis may be quite a bit less interesting than for worst case behavior. In this chapter, no analysis of best case expression swell behavior will be given.

3.1.2 Rational Number Calculations

Worst case expression swell arithmetic can also be developed for rational number operations. Essentially, rational expression swell arithmetic can be considered as an extension of integer expression swell arithmetic to ordered pairs of integer equivalence classes. These ordered pairs will be denoted like $\frac{\overline{m}}{\overline{n}}$, which represents the set of rational numbers with m-digit numerators and n-digit denominators. Since this is worst case arithmetic, the components of the ordered pairs will be assumed to be relatively prime, implying that the corresponding set of rational numbers is in lowest terms and cannot be reduced in size. This is consistent with defining integer GCDs to be one as was done in the previous subsection. Note that a given rational number (e.g., $\frac{2}{4} \in \frac{\overline{1}}{\overline{1}}$) need not actually be in lowest terms, but for worst case arithmetic operations, this assumption will always be made nevertheless. With these definitions, as with the integers, $\mathcal{N}_\beta$ can be seen to partition the rational numbers into a countably infinite set of equivalence classes. $\frac{\overline{m}}{\overline{n}}$ will then be the general notation for one of these equivalence classes where m and n can take on any integer value satisfying $m \geq 0$ and $n > 0$ (if $m = 0$ then n is only allowed to be 1). Note that $\mathcal{N}_\beta(\frac{\overline{m}}{\overline{n}}) \equiv m + n$ as was proposed earlier.

Now, worst case rational expression swell arithmetic can be defined by analogy with ordinary rational arithmetic in terms of the operations of the previously examined worst case integer expression swell arithmetic. This is done as follows:

Theorem 3.7 The unary operations of rational absolute value, negation and square root, when applied to the equivalence class $\frac{\overline{m}}{\overline{n}}$, possess bounding functions given by

(i) $\left|\frac{\overline{m}}{\overline{n}}\right| \equiv \frac{|\overline{m}|}{|\overline{n}|} = \frac{\overline{m}}{\overline{n}}$,

(ii) $\ominus \frac{\overline{m}}{\overline{n}} \equiv \frac{\ominus \overline{m}}{\overline{n}} = \frac{\overline{m}}{\overline{n}}$,

(iii) $\sqrt{\frac{\overline{m}}{\overline{n}}} \equiv \frac{\sqrt{\overline{m}}}{\sqrt{\overline{n}}} = \overline{\lceil \frac{m}{2} \rceil} / \overline{\lceil \frac{n}{2} \rceil}$.

Theorem 3.8 The binary operations of rational addition, subtraction, multiplication, division, exact division and exponentiation to an integer power when applied to the equivalence classes $\frac{\overline{m_1}}{\overline{n_1}}$, $\frac{\overline{m_2}}{\overline{n_2}}$ and $\frac{\overline{m}}{\overline{n}}$, where $m_1, m_2, m > 0$, possess bounding functions given by

(i) $\frac{\overline{m}_1}{\overline{n}_1} \oplus \frac{\overline{m}_2}{\overline{n}_2} \equiv \frac{(\overline{m}_1 \circledcirc \overline{n}_2) \oplus (\overline{n}_1 \circledcirc \overline{m}_2)}{\overline{n}_1 \circledcirc \overline{n}_2} = \frac{\overline{m_1+n_2} \oplus \overline{n_1+m_2}}{\overline{n_1+n_2}}$,

(ii) $\frac{\overline{m}_1}{\overline{n}_1} \ominus \frac{\overline{m}_2}{\overline{n}_2} \equiv \frac{(\overline{m}_1 \circledcirc \overline{n}_2) \ominus (\overline{n}_1 \circledcirc \overline{m}_2)}{\overline{n}_1 \circledcirc \overline{n}_2} = \frac{\overline{m_1+n_2} \oplus \overline{n_1+m_2}}{\overline{n_1+n_2}}$,

(iii) $\frac{\overline{m}_1}{\overline{n}_1} \circledcirc \frac{\overline{m}_2}{\overline{n}_2} \equiv \frac{\overline{m}_1 \circledcirc \overline{m}_2}{\overline{n}_1 \circledcirc \overline{n}_2} = \frac{\overline{m_1+m_2}}{\overline{n_1+n_2}}$,

(iv) $\frac{\overline{m}_1}{\overline{n}_1} \oslash \frac{\overline{m}_2}{\overline{n}_2} \equiv \frac{\overline{m}_1}{\overline{n}_1} \circledcirc \frac{\overline{n}_2}{\overline{m}_2} = \frac{\overline{m_1+n_2}}{\overline{n_1+m_2}}$,

(v) $\frac{\overline{m}_1}{\overline{n}_1} \overset{\text{exact}}{\oslash} \frac{\overline{m}_2}{\overline{n}_2} \equiv \frac{\overline{m}_1 \overset{\text{exact}}{\oslash} \overline{m}_2}{\overline{n}_1 \overset{\text{exact}}{\oslash} \overline{n}_2} = \frac{\overline{m_1-m_2+1}}{\overline{n_1-n_2+1}}$,

(vi) $\left(\frac{\overline{m}}{\overline{n}}\right) \textcircled{\uparrow} 0 \equiv \frac{\overline{m} \textcircled{\downarrow} 0}{\overline{n} \textcircled{\downarrow} 0} = \frac{\overline{1}}{\overline{1}}$,

(vii) $\left(\frac{\overline{m}}{\overline{n}}\right) \textcircled{\uparrow} k \equiv \frac{\overline{m} \textcircled{\downarrow} k}{\overline{n} \textcircled{\downarrow} k} = \frac{\overline{km}}{\overline{kn}} \quad (k > 0)$,

(viii) $\left(\frac{\overline{m}}{\overline{n}}\right) \textcircled{\uparrow} (-k) \equiv \left(\frac{\overline{n}}{\overline{m}}\right) \textcircled{\uparrow} k = \frac{\overline{kn}}{\overline{km}} \quad (k > 0)$.

If $m_1 = 0$ then the right-hand sides of (i) and (ii) become $\frac{\overline{m}_2}{\overline{n}_2}$, and the right-hand sides of (iii)–(v) become $\frac{\overline{0}}{\overline{1}}$. If $m_2 = 0$ then the right-hand sides of (i)–(v) are respectively, $\frac{\overline{m}_1}{\overline{n}_1}$, $\frac{\overline{m}_1}{\overline{n}_1}$, $\frac{\overline{0}}{\overline{1}}$, undefined and undefined. If $m = 0$ then the right-hand sides of (vi) and (viii) will be undefined and (vii) will be $\frac{\overline{0}}{\overline{1}}$.

A new operation, non-exact division, has been introduced but like the rest of the rational operations, it is adapted directly from ordinary rational arithmetic and so should come as no surprise. Finally, n-ary addition and multiplication for rational expression swell arithmetic are simple extensions of the binary versions.

Theorem 3.9 The n-ary operations of rational multiplication and addition, when applied to the equivalence classes $\frac{\overline{m}_1}{\overline{n}_1}, \ldots, \frac{\overline{m}_k}{\overline{n}_k}$ $(k > 0)$, where $m_i > 0$ for all i, possess bounding functions given by

(i) $\frac{\overline{m}_1}{\overline{n}_1} \circledcirc \frac{\overline{m}_2}{\overline{n}_2} \circledcirc \cdots \circledcirc \frac{\overline{m}_k}{\overline{n}_k} \equiv \frac{\overline{m}_1 \circledcirc \overline{m}_2 \circledcirc \cdots \circledcirc \overline{m}_k}{\overline{n}_1 \circledcirc \overline{n}_2 \circledcirc \cdots \circledcirc \overline{n}_k} = \frac{\overline{m_1+m_2+\cdots+m_k}}{\overline{n_1+n_2+\cdots+n_k}}$,

(ii) $\frac{\overline{m}_1}{\overline{n}_1} \oplus \frac{\overline{m}_2}{\overline{n}_2} \oplus \cdots \oplus \frac{\overline{m}_k}{\overline{n}_k} \equiv \frac{(\overline{m}_1 \circledcirc \overline{n}_2 \circledcirc \cdots \circledcirc \overline{n}_k) \oplus (\overline{n}_1 \circledcirc \overline{m}_2 \circledcirc \cdots \circledcirc \overline{n}_k) \oplus \cdots \oplus (\overline{n}_1 \circledcirc \overline{n}_2 \circledcirc \cdots \circledcirc \overline{m}_k)}{\overline{n}_1 \circledcirc \overline{n}_2 \circledcirc \cdots \circledcirc \overline{n}_k}$

$= \frac{\overline{n-n_1+m_1} \oplus \overline{n-n_2+m_2} \oplus \cdots \oplus \overline{n-n_k+m_k}}{\overline{n}}$ where $n \equiv \sum_{i=1}^{k} n_i$.

If any of the $m_i = 0$ then the right-hand side of (i) becomes $\frac{\overline{0}}{\overline{1}}$ and those terms with $m_i = 0$ are excluded from the sum in (ii).

3.1.3 MACSYMA Implementation

To obtain some practical experience with the above concepts, both integer and rational number worst case expression swell arithmetic were implemented in MACSYMA for $\beta = 10$. The top-level interface to expression swell arithmetic operations (there also exists a LISP-level interface which is accessed slightly differently) is provided by the functions abs_(x), neg_(x), add_($term_1$, $term_2$, ...), sub_(x,y), mul_($factor_1$, $factor_2$, ...), div_(x,y), ediv_(x,y)

[Exact DIVision], power_(x,y) and gcd_(x,y) [$\mathcal{N}_{10}$ is performed by ndigits(e)]. If the global variable exprswell is false (its default value), then these functions perform ordinary arithmetic. However, if exprswell is set to true, then these functions treat their arguments as equivalence classes of $\mathcal{N}_{10}$ (when appropriate) and perform worst case expression swell arithmetic on them (no mixing of integer and rational number equivalence classes is permitted). Thus, an algorithm can be implemented using this set of functions in place of the normal MACSYMA arithmetic operators except where expression swell operations are never appropriate, such as index calculations or incrementing loop indices. Then, with exprswell set to its default value, the algorithm can be run in a normal manner. However, by simply changing the value of exprswell, an expression swell analysis of the algorithm for inputs from a given set of equivalence classes can be performed.

The MACSYMA implementation of general integer worst case expression swell n-ary addition is an extension of the n-ary addition of like terms defined previously. The terms are first sorted into ascending order, then after eliminating any zeroes, the leading set of like terms (which may be a single term) is combined using the n-ary addition of Theorem 3.3. The result is tacked onto the beginning of the list containing the remainder of the terms, which is then resorted, if necessary, to maintain the terms in ascending order. Next, the leading set of like terms is combined, with perhaps an initial non-like term, once again using the n-ary addition of Theorem 3.3 (the non-like term, if there is one, will be treated as just another like term in the n-ary addition). These last two steps will then repeat until the list contains a single term. For example, $\overline{1}\oplus\overline{1}\oplus\overline{2}\oplus\overline{2}\oplus\overline{4}\oplus\overline{4} = \overline{2}\oplus\overline{2}\oplus\overline{2}\oplus\overline{4}\oplus\overline{4} = \overline{3}\oplus\overline{4}\oplus\overline{4} = \overline{5}$. This algorithm was chosen to try to minimize the results of general n-ary addition and thus provide as tight a bound as possible.

3.2 Matrix Algorithm Analysis

Here, worst case expression swell arithmetic is applied to the analysis of some of the matrix determinant and characteristic polynomial algorithms (and relationships) discussed in the previous chapter. The next section will then examine the results of a series of numerical case studies performed using both the MACSYMA implementation of expression swell arithmetic as well as various sets of numbers.

3.2.1 Determinant Computations

To start with, consider an analysis performed on the definition of the determinant [equation (2.2)]. For an $n \times n$ matrix (A), this will produce a sum of $n!$ products of n factors each, half of which are added and the rest subtracted. Therefore, if the entries of the matrix are all m-digit integers, then

$$\mathcal{N}_\beta(\det_d A) \;\leq\; \mathcal{N}_\beta\left(\left[\frac{n!}{2} \circledcirc (\overline{m} \circledast n)\right] \ominus \left[\frac{n!}{2} \circledcirc (\overline{m} \circledast n)\right]\right)$$

$$
\begin{aligned}
&\leq \mathcal{N}_\beta\left(\left[\frac{n!}{2} \circledcirc (\overline{m} \circledcirc n)\right] \oplus \left[\frac{n!}{2} \circledcirc (\overline{m} \circledcirc n)\right]\right) \\
&\leq \mathcal{N}_\beta(n! \circledcirc (\overline{m} \circledcirc n)) \leq \mathcal{N}_\beta(n! \circledcirc \overline{nm}) \leq nm + \mathcal{N}_\beta(n! - 1) \ . \qquad (3.5)
\end{aligned}
$$

One of the major schemes generally implemented in CASs for computing determinants is some variant of Gaussian elimination. The Bareiss style algorithms, especially, are common as they are integer preserving. The simplest (and most frequently seen) of these is the one-step method of equation (2.10). For the $n \times n$ matrix A consisting of m-digit integer elements, worst case expression swell analysis of this algorithm reveals that

$$
\mathcal{N}_\beta(\det_G A) \leq nm + (n-1)^2 \ . \qquad (3.6)
$$

This can be seen by performing the elimination on a representative matrix using expression swell arithmetic. In particular, for a 4×4 matrix of m-digit integers, the reduction proceeds as follows:

$$
\begin{pmatrix} m & m & m & m \\ m & m & m & m \\ m & m & m & m \\ m & m & m & m \end{pmatrix} \rightarrow \begin{pmatrix} m & m & m & m \\ 0 & 2m+1 & 2m+1 & 2m+1 \\ 0 & 2m+1 & 2m+1 & 2m+1 \\ 0 & 2m+1 & 2m+1 & 2m+1 \end{pmatrix}
$$

$$
\rightarrow \begin{pmatrix} m & m & m & m \\ 0 & 2m+1 & 2m+1 & 2m+1 \\ 0 & 0 & 3m+4 & 3m+4 \\ 0 & 0 & 3m+4 & 3m+4 \end{pmatrix} \rightarrow \begin{pmatrix} m & m & m & m \\ 0 & 2m+1 & 2m+1 & 2m+1 \\ 0 & 0 & 3m+4 & 3m+4 \\ 0 & 0 & 0 & 4m+9 \end{pmatrix} .
$$

For more detail, consider the second stage calculation of the (3,4) element:

$$
\begin{aligned}
\overline{a_{34}^{(2)}} &= [(\overline{a_{22}^{(1)}} \circledcirc \overline{a_{34}^{(1)}}) \ominus (\overline{a_{32}^{(1)}} \circledcirc \overline{a_{24}^{(1)}})] \overset{\text{exact}}{\oslash} \overline{a_{11}^{(0)}} \\
&= [(\overline{2m+1} \circledcirc \overline{2m+1}) \ominus (\overline{2m+1} \circledcirc \overline{2m+1})] \overset{\text{exact}}{\oslash} \overline{m} \\
&= [\overline{4m+2} \ominus \overline{4m+2}] \overset{\text{exact}}{\oslash} \overline{m} = \overline{4m+3} \overset{\text{exact}}{\oslash} \overline{m} = \overline{3m+4} \ .
\end{aligned}
$$

Thus, the right-hand side of (3.6) is a solution of the recurrence relation

$$
r_n = (2r_{n-1} + 1) - r_{n-2} + 1 = 2(r_{n-1} + 1) - r_{n-2}
$$

with initial conditions of $r_0 = 1$ and $r_1 = m$.

A comparison of the above two theoretical worst case determinations of the size of the determinant of an $n \times n$ matrix with m-digit integer entries shows that the former yields a tighter and thus a better bound. This can be seen by examining the difference

$$
\begin{aligned}
\mathcal{N}_\beta(\det_G A) - \mathcal{N}_\beta(\det_d A) &\sim (n-1)^2 - \mathcal{N}_\beta(n! - 1) \\
&= (n-1)^2 - \lfloor \log_\beta(n! - 1) \rfloor - 1 \qquad (n > 1) \\
&\approx n^2 - 2n - \left\lfloor \log_\beta \sqrt{2\pi n} \left(\tfrac{n}{e}\right)^n \right\rfloor \qquad (n \to \infty) \\
&\approx n^2 - n \log_\beta n + n(\log_\beta e - 2) \qquad (n \to \infty).
\end{aligned}
$$

The second line is a result of the definition of $\mathcal{N}_\beta$, and the third line is a result of approximating $n!$ (and hence $n! - 1$) for n large by Stirling's formula. Indeed, $\mathcal{N}_\beta(\det_G A)$ will strictly dominate $\mathcal{N}_\beta(\det_d A)$ for all $n > 2$. The primary reason for this behavior is that worst case expression swell analysis is performed one stage at a time while following the Gaussian elimination algorithm, but all at once when using the definition of the determinant. The latter analysis takes full advantage of the stricter bounds produced by combining as many n-ary arithmetical operations as possible at one time, and so yields the better estimate. The former analysis does produce bounds on the size of the matrix elements at each stage of the elimination, but any overestimates from previous stages tend to accumulate.

The determinant of a matrix can also be bounded using Hadamard's inequality (2.28). This relationship can be reexpressed without square roots for an $n \times n$ matrix A by

$$|\det A|^2 \leq \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij}^2 \right) .$$

If all the elements of A are m-digit integers, then here worst case expression swell analysis gives

$$\begin{aligned} \mathcal{N}_\beta(|\det_H A|^2) &\leq \mathcal{N}_\beta([n \odot (\overline{m} \oplus 2)] \oplus n) \leq \mathcal{N}_\beta([n \odot \overline{2m}\,] \oplus n) \\ &\leq \mathcal{N}_\beta(\overline{2m + \mathcal{N}_\beta(n-1)} \oplus n) \leq \mathcal{N}_\beta(\overline{2nm + n\mathcal{N}_\beta(n-1)}) . \end{aligned}$$

Hence,

$$\mathcal{N}_\beta(\det_H A) \leq nm + \left\lceil \frac{1}{2} n \mathcal{N}_\beta(n-1) \right\rceil . \tag{3.7}$$

This estimate is of about the same asymptotic order as the one produced by the definition of the determinant.

Rational expression swell arithmetic can also be applied fruitfully to Hadamard's inequality. Suppose B is an $n \times n$ matrix with rational number entries consisting of m-digit numerators and denominators, then

$$\mathcal{N}_\beta(|\det_H B|^2) \leq \mathcal{N}_\beta\left(\left[n \odot \left(\frac{\overline{m}}{\overline{m}} \oplus 2 \right) \right] \oplus n \right) \leq \mathcal{N}_\beta\left(\left[n \odot \frac{\overline{2m}}{\overline{2m}} \right] \oplus n \right) .$$

Now,

$$k \odot \frac{\overline{m}}{\overline{m}} \equiv \underbrace{\frac{\overline{m}}{\overline{m}} \oplus \cdots \oplus \frac{\overline{m}}{\overline{m}}}_{k \text{ terms}} = \frac{\overbrace{\overbrace{(\overline{m} \odot \cdots \odot \overline{m})}^{k \text{ factors}} \oplus \cdots \oplus \overbrace{(\overline{m} \odot \cdots \odot \overline{m})}^{k \text{ factors}}}^{k \text{ terms}}}{\underbrace{\overline{m} \odot \cdots \odot \overline{m}}_{k \text{ factors}}}$$

$$= \frac{k \odot \overline{km}}{\overline{km}} = \frac{\overline{km + \mathcal{N}_\beta(k-1)}}{\overline{km}} ,$$

therefore,

$$\mathcal{N}_\beta(|\det_H B|^2) \leq \mathcal{N}_\beta\left(\overline{\frac{2nm+\mathcal{N}_\beta(n-1)}{2nm}} \circledast n\right) \leq \mathcal{N}_\beta\left(\overline{\frac{2n^2m+n\mathcal{N}_\beta(n-1)}{2n^2m}}\right) .$$

Taking an exact square root once again,

$$\mathcal{N}_\beta(\det_H B) \leq \mathcal{N}_\beta\left(\overline{\frac{n^2m+\lceil n\mathcal{N}_\beta(n-1)/2\rceil}{n^2m}}\right) = 2n^2m + \left\lceil \frac{1}{2}n\mathcal{N}_\beta(n-1)\right\rceil .$$

3.2.2 Characteristic Polynomial Computations

Generally in CASs, the characteristic polynomial of a matrix A is calculated by forming the matrix $A-\lambda I$, where λ is a scalar variable, and then computing its determinant. The determinant will always involve polynomial arithmetic, even for a purely numerical matrix, so it is difficult to directly perform expression swell analysis on this method. However, in the previous subsection, expression swell analysis was performed on the determinant calculation of certain integer and rational number matrices. Since the determinant is plus or minus the constant term of the characteristic polynomial, which for nonsingular matrices is typically among the largest coefficients of the polynomial in gross size, these analyses should give a good bound on the size of the largest coefficient of the characteristic polynomial for matrices with entries of the same size.

A second method, which computes the coefficients of the characteristic polynomial of a purely numerical matrix using no polynomial arithmetic, involves taking the trace of powers of the matrix (the method of Leverrier—see Section 2.2.1). This method requires $O(n^4)$ operations and so is not practical for large matrices. Nevertheless, it is instructive to perform worst case expression swell analysis on this algorithm. Initially, consider the first portion of the algorithm, which computes the various traces. If the entries of the $n \times n$ matrix A are m-digit integers, then the number of digits in the entries of $AA = A^2$ will be bounded by $2m+\mathcal{N}_\beta(n-1)$ [i.e., $\mathcal{N}_\beta(n \circledcirc [\overline{m} \circledcirc \overline{m}])$]. For $AA^2 = A^3$, the number of digits in the entries will be bounded by $3m+2\mathcal{N}_\beta(n-1)$, and in general, the number of digits in the entries of $AA^{k-1} = A^k$ will be bounded by $km+(k-1)\mathcal{N}_\beta(n-1)$. Hence,

$$\begin{aligned}\mathcal{N}_\beta(\operatorname{tr} A^k) &= \mathcal{N}_\beta\left(\sum_{i=1}^{n}(A^k)_{ii}\right) \leq \mathcal{N}_\beta(n \circledcirc \overline{km+(k-1)\mathcal{N}_\beta(n-1)}) \\ &\leq k[m+\mathcal{N}_\beta(n-1)] .\end{aligned}$$

Now, the coefficient c_k of λ^k $(0 \leq k \leq n-1)$ in the characteristic polynomial $c_A(\lambda)$ of A will be composed of $p(n-k)$ terms, each of which will be proportional to a product of traces such that the total sum of powers involved is $n-k$. Here, $p(n)$ is the number of partitions of the integer n (i.e., the number of ways n can be written as a sum of positive integers where order does not matter), and is computable from Hardy and Ramanujan's formula [Coh78, p. 79] by

$$p(n) \approx \frac{1}{4\sqrt{3}\,n} e^{\pi\sqrt{2n/3}} .$$

For example, the coefficient of λ for A, 4×4, is

$$c_1 = \frac{\operatorname{tr} A^3}{3} - \frac{(\operatorname{tr} A^2)(\operatorname{tr} A)}{2} + \frac{(\operatorname{tr} A)^3}{6} ,$$

which consists of $p(4-1) = p(3) = 3$ terms. It is easily seen that $\mathcal{N}_\beta([\operatorname{tr} A] \circledast k) = \mathcal{N}_\beta(\operatorname{tr} A^k)$, therefore, ignoring divisions,

$$\begin{aligned} \mathcal{N}_\beta(c_k) &\leq \mathcal{N}_\beta(p(n-k) \circledcirc \overline{(n-k)[m + \mathcal{N}_\beta(n-1)]}) \\ &\leq (n-k)[m + \mathcal{N}_\beta(n-1)] + \mathcal{N}_\beta(p(n-k) - 1) . \end{aligned}$$

Note that $\mathcal{N}_\beta(c_{n-1}) \sim \mathcal{N}_\beta(\operatorname{tr} A)$ [since $p(1) = 1$], which is consistent with $c_{n-1} = -\operatorname{tr} A$ (again, see Section 2.2.1). Also,

$$\mathcal{N}_\beta(\det_L A) = \mathcal{N}_\beta(c_0) \leq nm + n\mathcal{N}_\beta(n-1) + \mathcal{N}_\beta(p(n) - 1) , \tag{3.8}$$

hence,

$$\begin{aligned} \mathcal{N}_\beta(\det_L A) &\approx nm + n(\lfloor \log_\beta(n-1) \rfloor + 1) + \left\lfloor \log_\beta \left(\frac{1}{4\sqrt{3}\, n} e^{\pi\sqrt{2n/3}} - 1 \right) \right\rfloor + 1 \qquad (n > 1) \\ &\approx nm + n \log_\beta n + n \qquad (n \to \infty). \end{aligned}$$

This estimate compares favorably to the one derived from the definition of the determinant (although it will be seen in the next section that it is somewhat higher).

A third way to compute the characteristic polynomial of a matrix is to transform the matrix into upper Hessenberg or tridiagonal form via a similarity reduction (which will preserve the characteristic polynomial), and then compute the characteristic polynomial of the transformed matrix using a computationally cheap algorithm. Worst case expression swell analysis of this kind of procedure is difficult to generalize and so will not be attempted. In the following section, results from the analyses of specific examples will be presented. These analyses were performed using the MACSYMA implementation of worst case expression swell arithmetic.

3.3 Results of Case Studies

It is all very well to derive theoretical bounds on the possible expression swell in a calculation, but when compared to the results of actual computations, how good of an estimate of real behavior do these bounds really provide? In order to answer this question, a number of case studies were performed involving matrices of various sizes with initial integer or rational number entries. Some of these results provide quite striking examples of intermediate expression swell in action.

To begin with, Table 3.3 presents the four worst case bounds [relations (3.7), (3.5), (3.8) and (3.6), respectively, of the previous section] derived for the determinant of an $n \times n$

n	Hadamard's inequality	Analysis of the Definition	Method of Leverrier	Gaussian elimination
3	14	13	16	16
4	18	18	21	25
5	23	23	26	36
6	27	27	32	49
7	32	32	37	64
8	36	37	42	81
9	41	42	47	100
10	45	47	52	121
20	100	99	123	441
30	150	153	184	961
40	200	208	245	1681
50	250	265	306	2601
60	300	322	367	3721
70	350	381	427	5041
80	400	439	488	6561
90	450	499	548	8281
100	500	558	609	10201

Table 3.3 Various predictions of the maximum number of digits comprising the determinant of an $n \times n$ integer general matrix ($m = 4$).

matrix containing m-digit integer entries, for a variety of values of n with $m = 4$ ($\beta = 10$ will be assumed throughout this section). Hadamard's inequality yields the lowest upper estimate on the number of digits in the determinant, although the bounds derived from the definition of the determinant and Leverrier's method fall within the same approximate asymptotic order. As noted before, worst case bounds derived from the fraction-free Gaussian elimination algorithm grow much faster than those computed from the other methods.

Table 3.4 presents the results of actual calculations performed on $n \times n$ matrices for values of n ranging from 3 through 10, where the initial matrix elements were 4-digit integers. The table is divided into three sections. The first section (a) exhibits the greatest number of digits encountered in the elements of the upper Hessenberg matrix produced by a similarity transformation of the original matrix using a division-free version of the algorithm. The column labeled exprswell shows the results produced by setting the global variable exprswell to true in MACSYMA. The next column displays the outcome of using an initial matrix composed of the n^2 largest 4-digit primes. The last three columns in this table present the results of a statistical sampling in which matrices consisting initially of random 4-digit integers were used. The first pair of numbers is the mean and standard deviation obtained

n	exprswell	primes	random numbers	N_{samples}	min → max
3	14	13	11.42 ± 0.75	100	9 → 12
4	43	27	30.73 ± 2.10	100	24 → 35
5	130	59	74.98 ± 2.66	100	68 → 81
6	391	98	142.38 ± 4.26	100	130 → 152
7	1174	155	249.75 ± 6.75	100	233 → 262
8	3523	240	366.50 ± 9.53	100	343 → 388
9	10570	334	585.30 ± 11.79	100	556 → 612
10	31711	469	825.92 ± 14.22	13	797 → 844

a. Division free transformation into Hessenberg form.

n	exprswell	primes	random numbers	N_{samples}	min → max
3	43	31	30.09 ± 2.28	100	22 → 34
4	172	92	115.60 ± 7.90	100	90 → 128
5	648	277	364.62 ± 13.36	100	331 → 393
6	2341	561	840.63 ± 25.23	100	765 → 898
7	8209	1058	1731.99 ± 47.74	100	1616 → 1824
8	28170	1882	2911.08 ± 77.09	100	2720 → 3081
9	95110	2959	5243.04 ± 106.39	100	4977 → 5483
10	317083	4644	8243.42 ± 147.58	12	7942 → 8422

b. Characteristic polynomial of the Hessenberg matrix.

n	exprswell	primes	random numbers	N_{samples}	min → max
3	17	7	11.40 ± 0.84	10	10 → 12
4	25	7	15.20 ± 0.63	10	14 → 16
5	34	8	18.70 ± 0.67	10	18 → 20
6	44	8	22.70 ± 0.95	10	21 → 24
7	55	11	26.50 ± 0.97	10	24 → 27
8	67	11	30.60 ± 0.52	10	30 → 31
9	80	13	34.20 ± 0.63	10	33 → 35
10	94	14	37.90 ± 0.57	10	37 → 39

c. Above divided by its leading coefficient.

Table 3.4 Maximum number of digits comprising the results of various transformations applied to an $n \times n$ integer general matrix ($m = 4$).

from a series of calculations whose number is given by N_{samples}.[1] The last column gives the minimum and maximum values for the largest number of digits attained.

Section (b) of Table 3.4 shows the number of digits contained in the largest coefficient of the characteristic polynomial that was computed directly from the corresponding upper Hessenberg matrix of section (a). Due to the division-free nature of the similarity reduction to upper Hessenberg form that was used, a straightforward computation of the characteristic polynomial will produce, in general, a non-monic result. However, the leading coefficient of this polynomial will exactly divide all the other coefficients, producing a monic polynomial whose constant term will be plus or minus the determinant of the original matrix. Essentially, this particular procedure for computing the monic characteristic polynomial of an integer matrix reserves all divisions until the final step of the calculation. The maximum number of digits found in a coefficient (nearly always the constant term) of the final, simplified characteristic polynomial is presented in the final section (c) of Table 3.4.

The trend displayed in Table 3.4 is quite typical of calculations involving intermediate expression swell. Even though the final numbers have relatively few digits, intermediate computations in this particular algorithm are already creating integers whose size is approaching 5000 digits when n is just 10. It is interesting to note that the matrices with prime entries pretty much provide the smallest actual results. The opposite conclusion prevails in most of the other calculations surveyed here.

The theoretical limits on the size of the determinant, tabulated in Table 3.3, nicely bound the results in Table 3.4c. The exprswell calculations tend to greatly overestimate the maximum number of digits actually produced in the first two phases of the computation, except for the lowest values of n. However, after the final exact divisions, these bounds drop down to much more reasonable estimates. One has to be careful, though, in interpreting the results of the exprswell computations for this last calculation. The coefficients of the non-monic characteristic polynomials in Table 3.4b were determined by a series of calculations that maximized the number of digits at each step. In order to truly maximize the number of digits in the coefficients of the final monic polynomials, the leading coefficients of the non-monic polynomials (which act as divisors) should really have been minimized in the course of their calculation (using a best case expression swell arithmetic). This action, however, could have impacted the results of the worst case expression swell arithmetic in earlier phases of the calculations. Thus, the occurrence of exact divisions in an algorithm such as this one can lead to theoretical uncertainties in verifying whether the analysis does indeed produce a bounding calculation, although in practice, no exceptions have been found so far.

Table 3.5 presents the maximum number of digits found in coefficients of the characteristic polynomials of upper Hessenberg matrices whose initial nonzero entries were 4-digit integers. The exprswell calculations and the results from the statistical survey for $n = 3$ through 10 are remarkably similar to those exhibited in Table 3.4c. These similarities ap-

[1] In some cases, when a sequence of calculations was performed as here, later calculations were done fewer times due to time constraints and/or random problems occurring as a result of running large MACSYMA jobs continuously for days at a time (such as occasional memory corruption resulting from the growth of a MACSYMA job).

n	exprswell	primes	random numbers	N_{samples}	min $\rightarrow$ max
3	17	9	11.46 $\pm$ 0.54	100	10 $\rightarrow$ 12
4	25	14	15.27 $\pm$ 0.60	100	13 $\rightarrow$ 16
5	34	15	19.13 $\pm$ 0.68	100	17 $\rightarrow$ 20
6	44	21	23.00 $\pm$ 0.70	100	21 $\rightarrow$ 24
7	55	23	26.51 $\pm$ 0.75	100	24 $\rightarrow$ 28
8	67	29	30.35 $\pm$ 0.73	100	28 $\rightarrow$ 32
9	80	33	34.12 $\pm$ 0.76	100	33 $\rightarrow$ 36
10	94	38	37.91 $\pm$ 0.79	100	36 $\rightarrow$ 39
20	289	—	75.40 $\pm$ 0.89	5	74 $\rightarrow$ 76
30	584	—	113.40 $\pm$ 1.14	5	112 $\rightarrow$ 115
40	979	—	150.60 $\pm$ 1.34	5	150 $\rightarrow$ 153
50	1474	—	187.60 $\pm$ 1.14	5	186 $\rightarrow$ 189
60	2069	—	227.60 $\pm$ 1.52	5	226 $\rightarrow$ 230
70	2764	—	264.40 $\pm$ 1.14	5	263 $\rightarrow$ 266
80	3559	—	301.80 $\pm$ 1.30	5	301 $\rightarrow$ 304
90	4454	—	341.60 $\pm$ 1.95	5	339 $\rightarrow$ 344
100	5449	—	378.00 $\pm$ 2.83	5	374 $\rightarrow$ 380

Table 3.5 Maximum number of digits comprising the coefficients of the characteristic polynomial of an $n \times n$ integer Hessenberg matrix ($m = 4$).

pear to imply that the results of computing characteristic polynomials of integer Hessenberg matrices will give a very good indication of the trends to be expected when computing monic characteristic polynomials of integer general matrices. This is useful since there are no theoretical uncertainties here as there were above, because this algorithm involves no divisions. Also, the computation of the characteristic polynomial of an integer Hessenberg matrix is reasonably fast as well as conservative of memory, so it was possible to perform calculations up through $n = 100$.[2]

The trends of Table 3.5 become more pronounced for $n > 10$. The worst case maximums derived from Hadamard's inequality continue to provide good bounds on the actual results, although they become less good for increasing n. For $n > 10$, these bounds, which are growing linearly, begin to leave the real recorded maximums, which are also growing about linearly, further and further behind. Generalizing these data tendencies (as well as those found for $m = 5$ and 6—see Table 3.6), a good empirical bound on the maximum number of digits in the determinant of an $n \times n$ integer matrix (A), which has entries consisting of no greater than m digits, appears to be given simply by

$$\mathcal{N}_{10}(\det A) \leq nm \ .$$

[2]All the calculations in this section were performed on Sun 3/160 workstations with 4 megabytes of memory running under a version of the UNIX operating system (SunOS 3.4).

n	$m = 4$	$m = 5$	$m = 6$
5	20	25	30
10	39	49	59
20	76	97	116
30	115	146	176
40	153	193	232
50	189	242	292

Table 3.6 Maximum number of digits comprising the coefficients of the characteristic polynomial of an $n \times n$ integer Hessenberg matrix.

n	exprswell	primes	random numbers	N_{samples}	min $\rightarrow$ max
3	107	93	78.07 $\pm$ 5.23	100	65 $\rightarrow$ 89
4	225	205	162.47 $\pm$ 11.43	100	123 $\rightarrow$ 187
5	404	376	283.81 $\pm$ 19.35	100	236 $\rightarrow$ 331
6	656	620	451.50 $\pm$ 30.81	100	346 $\rightarrow$ 507
7	993	946	657.26 $\pm$ 45.02	100	536 $\rightarrow$ 746
8	1427	1368	934.24 $\pm$ 56.08	100	804 $\rightarrow$ 1066
9	1970	1895	1245.57 $\pm$ 80.66	100	1080 $\rightarrow$ 1426
10	2634	2540	1612.45 $\pm$ 112.96	100	1245 $\rightarrow$ 1932

Table 3.7 Maximum number of digits comprising the coefficients of the characteristic polynomial of an $n \times n$ "derationalized" Hessenberg matrix, initially filled with 4-digit rational numbers.

Note that this is the worst case bound for the determinant of an $n \times n$ diagonal matrix ($\overline{m} \textcircled{\uparrow} n = \overline{nm}$).

Table 3.7 shows what happens when the Hessenberg matrix is initially filled with 4-digit rational numbers (the numerators and denominators are both 4-digit integers) and the matrix is first "derationalized." Derationalization is the process of converting a matrix of rational numbers into a matrix of integers by multiplying the matrix by the least common multiple of the denominators of its rational number entries. This number (call it d) is an implicit divisor of the integer matrix, and if the matrix is involved in subsequent calculations, d needs to be taken into account. In this particular example, the characteristic polynomial will again, in general, be non-monic, but in this case, dividing the coefficients by the leading coefficient is not guaranteed to be exact (since this is really the characteristic polynomial of a rational number matrix), and so is not performed.

In Table 3.7, the "prime" rational numbers (rational numbers whose numerators and denominators are prime integers) provided the worst actual results of expression swell. The

	exprswell			primes		
n	$m = 4$	$m = 5$	$m = 6$	$m = 4$	$m = 5$	$m = 6$
3	107	131	155	93	116	140
4	225	277	329	205	253	307
5	404	499	594	376	471	565
6	656	812	968	620	777	931
7	993	1231	1469	946	1186	1424
8	1427	1771	2115	1368	1716	2059
9	1970	2447	2924	1895	2380	2858
10	2634	3274	3914	2540	3194	3835

Table 3.8 Maximum number of digits comprising the coefficients of the characteristic polynomial of an $n \times n$ "derationalized" Hessenberg matrix, initially filled with m-digit rational numbers.

observation that "prime" rational numbers typically generate the greatest expression growth can also be noted in other calculations that start off with matrices of rational numbers, completely contrary to what was observed earlier for calculations that started with integer matrices. A nice discovery is that the exprswell calculations produced good bounds on the worst case expression swell. This may be due to the following consideration. The prime computations will be particularly large since the denominators of the initial rational number entries will all be big and relatively prime to each other, so that derationalization will create nearly the maximum possible integer entries for a matrix of a given size undergoing this transformation. The exprswell calculations will act as if they are computing the characteristic polynomials of integer Hessenberg matrices with large effective values of m (essentially, the original m times the number of nonzero elements in the matrix). Thus, any excesses in the exprswell computations (which seem to depend only on n) will be overwhelmed (at least, at these values of n) by the large effective values of m. Table 3.8 shows "prime" and exprswell results for three values of m, demonstrating that the relative error narrows as m increases, a result which is to be expected if the above argument is correct.

In Table 3.9 is presented the results of another characteristic polynomial computation via an intermediate Hessenberg transformation. This time, the entries of the initial general matrices were 4-digit rational numbers and all calculations were done in rational arithmetic. As an example, Figure 3.3 exhibits the complete results of a typical calculation. Note that the numbers in the table represent the number of digits in the maximal rational number (i.e., the largest value obtained by summing the number of digits in the numerator and denominator of each rational number under consideration).

Again, the calculations that started off with "prime" rational number matrices produced the greatest growth in expression size. Also, like the computations whose results were shown in Table 3.4, expression swell decreased dramatically between the intermediate results found

n	exprswell	primes	random numbers	N_{samples}	min → max
3	82	61	50.15 ± 4.84	100	36 → 60
4	911	216	176.29 ± 11.92	100	137 → 202
5	13534	622	504.29 ± 27.15	100	438 → 564
6	255497	1369	1096.85 ± 46.77	93	983 → 1223
7	5852292	2588	2013.28 ± 98.13	100	1722 → 2206
8	157563119	4313	3275.51 ± 130.41	57	3027 → 3584
9	4874278234	6671	4971.33 ± 170.71	12	4730 → 5310
10	170327525637	9717	7194.50 ± 198.44	6	6932 → 7488

a. Transformation into Hessenberg form.

n	exprswell	primes	random numbers	N_{samples}	min → max
3	711	69	60.34 ± 4.05	100	46 → 69
4	15812	125	105.24 ± 5.19	100	91 → 117
5	493670	198	163.02 ± 6.45	100	145 → 178
6	19254949	286	231.30 ± 8.34	93	208 → 250
7	898326065	390	309.65 ± 11.57	100	281 → 334
8	48860659874	508	399.82 ± 14.61	57	372 → 440
9	3039845981556	642	498.25 ± 11.41	12	484 → 518
10	213099621017855	791	611.50 ± 11.43	6	596 → 631

b. Characteristic polynomial of the Hessenberg matrix.

Table 3.9 Maximum number of digits comprising the results of transformations applied to an $n \times n$ general matrix, initially filled with 4-digit rational numbers.

in the entries of the Hessenberg matrices and the final numbers that comprised the coefficients of the characteristic polynomials. These latter numbers appear to be quite nicely bounded by $2n^2m$, which is very similar to the bound derived from Hadamard's inequality. (In this case, the worst case bound for a diagonal matrix does not quite match up: $\overline{\overline{m}} \textcircled{\uparrow} n = \overline{\overline{nm}} = \overline{2nm}$.) The bounds computed from the MACSYMA implementation of rational worst case expression swell arithmetic are huge and provide virtually no useful information.

As an experiment to see what effect GCDs have on rational number computations, the above calculations on "prime" rational number matrices were repeated with the GCD function forced always to return one, thus allowing no cancellation of common factors. These results are compared with those from Table 3.9, in which GCDs were taken freely, in Table 3.10. It is quite clear from this comparison of maximal numbers of digits that GCDs, which are taken every time a new rational number is formed in MACSYMA, have important effects in minimizing expression swell which, in the samples reviewed in Table 3.10, is

$$A = \begin{pmatrix} \frac{9533}{9539} & \frac{9547}{9551} & \frac{9587}{9601} & \frac{9613}{9619} & \frac{9623}{9629} \\ \frac{9631}{9643} & \frac{9649}{9661} & \frac{9677}{9679} & \frac{9689}{9697} & \frac{9719}{9721} \\ \frac{9733}{9739} & \frac{9743}{9749} & \frac{9767}{9769} & \frac{9781}{9787} & \frac{9791}{9803} \\ \frac{9811}{9817} & \frac{9829}{9833} & \frac{9839}{9851} & \frac{9857}{9859} & \frac{9871}{9883} \\ \frac{9887}{9901} & \frac{9907}{9923} & \frac{9929}{9931} & \frac{9941}{9949} & \frac{9967}{9973} \end{pmatrix}$$

a. Initial matrix.

$$\mathcal{N}_{10}(H) = \begin{pmatrix} \frac{4}{4} & \frac{33}{32} & \frac{105}{104} & \frac{175}{175} & \frac{4}{4} \\ \frac{4}{4} & \frac{33}{32} & \frac{106}{105} & \frac{175}{175} & \frac{4}{4} \\ 0 & \frac{53}{56} & \frac{113}{117} & \frac{187}{190} & \frac{13}{16} \\ 0 & 0 & \frac{155}{158} & \frac{229}{231} & \frac{69}{72} \\ 0 & 0 & 0 & \frac{309}{313} & \frac{143}{146} \end{pmatrix}$$

b. The sizes of the entries in the upper Hessenberg matrix H that is similar to A.

$$\mathcal{N}_{10}(c_H(\lambda)) = \lambda^5 + \frac{21}{20}\lambda^4 + \frac{98}{100}\lambda^3 + \frac{95}{100}\lambda^2 + \frac{92}{100}\lambda + \frac{88}{100}$$

c. The sizes of the coefficients of the characteristic polynomial $c_H(\lambda) = c_A(\lambda)$.

Figure 3.3 Stages in the expression swell analysis of the computation of the characteristic polynomial of a 5×5 general matrix containing "prime" 4-digit rational numbers.

	Hessenberg matrix		Characteristic polynomial	
n	GCDs	no GCDs	GCDs	no GCDs
3	61	61	69	122
4	216	379	125	565
5	622	2360	198	3129
6	1369	14203	286	18365
7	2588	82576	390	——

Table 3.10 Maximum number of digits comprising the results of the transformation of an $n \times n$ general matrix, initially filled with 4-digit "prime" rational numbers, into Hessenberg form and then taking the characteristic polynomial of the Hessenberg matrix.

n	primes	random numbers	N_{samples}	min $\rightarrow$ max
3	29	24.31 $\pm$ 1.82	100	17 $\rightarrow$ 27
4	135	118.96 $\pm$ 6.02	100	96 $\rightarrow$ 133
5	451	377.51 $\pm$ 21.17	100	334 $\rightarrow$ 424
6	1299	1066.30 $\pm$ 57.21	80	873 $\rightarrow$ 1199
7	2856	2237.50 $\pm$ 107.46	20	2074 $\rightarrow$ 2475
8	5302	3997.60 $\pm$ 120.55	5	3882 $\rightarrow$ 4163

a. Transformation into tridiagonal form.

n	primes	random numbers	N_{samples}	min $\rightarrow$ max
3	30	30.64 $\pm$ 2.49	100	18 $\rightarrow$ 34
4	56	53.45 $\pm$ 3.11	100	43 $\rightarrow$ 60
5	89	82.22 $\pm$ 4.49	100	71 $\rightarrow$ 91
6	130	118.61 $\pm$ 5.55	80	105 $\rightarrow$ 129
7	178	155.80 $\pm$ 5.82	20	142 $\rightarrow$ 166
8	236	197.20 $\pm$ 6.22	5	190 $\rightarrow$ 207

b. Characteristic polynomial of the tridiagonal matrix.

Table 3.11 Maximum number of digits comprising the results of transformations applied to an $n \times n$ symmetric matrix, initially filled with 4-digit rational numbers.

growing exponentially for the "no GCDs" cases. The reason that the results for "no GCDs" are complete only through $n = 6$ is because the MACSYMA computations for bigger cases ran out of available memory.

One last computation of interest is the transformation of a symmetric matrix into tridiagonal form (the modified fast Givens' method for a real matrix—see Section 2.2.3). Table 3.11 displays the results for six values of n. A notable feature is that although the final results consist of less than half the number of digits compared to those for a general matrix (Table 3.9), the intermediate forms, starting at $n = 7$, produce larger maximal expression swell. Thus, it can be seen that the modified fast Givens' algorithm will quickly become unwieldly for increasing n, and it will be better to just use the general purpose Hessenberg algorithm.

3.4 Concluding Remarks

Expression swell is a significant and in the long run, an inevitable problem of symbolic computation. Since expression swell is ultimately unavoidable, one goal should be to make it at least manageable. One way to accomplish this is to develop a collection of procedures

which will allow a user to be able to predict the progress of expression growth for a given calculation. In this chapter, an attempt has been made to develop some tools to pursue the above goal for a certain class of computations. The application of these tools has had mixed success.

One tool for charting expression growth is to perform a variety of calculations with varying initial conditions, and measure the sizes of the final and various intermediate results. This procedure, although tedious, has produced some interesting conclusions about the maximum size of coefficients in the characteristic polynomials of certain integer and rational number matrices, where the nonzero entries were initially uniform in size.

A second tool is worst case (and best case) expression swell arithmetic. This can be used both for deriving general bounds on expression swell over a class of calculations, and also for determining such bounds for specific computations. Some of the bounds looked at above have produced good estimates on real maximal expression size, while others have given outrageous overestimates. At this preliminary stage in the development of a useful expression swell arithmetic, some observations can be made based on the experience gained here.

For integer worst case expression swell arithmetic, the best results (the tightest bounds) have come about when analyses have been performed on algorithms or inequalities that have minimized the mix of operations needed to obtain an estimate. For example, the analysis of the Gaussian elimination algorithm of Section 3.2.1 and many of the computed bounds in the previous section required a complex, multi-stage calculation and in many cases, this resulted in excessive estimates, even for small values of the matrix dimension. On the other hand, simple formulas like Hadamard's inequality, where the various arithmetical operations are reasonably consolidated, gave good bounds on the expression swell. A major reason for this behavior is that expression swell arithmetic ignores past history. Thus, if a calculation produces a worst case outcome that just barely exceeds n digits [e.g., $(\beta+1) \circledcirc \overline{n-2}$], the result will normally be treated as if it was the largest possible n-digit integer in the very next calculation, effectively neglecting the number's origin. Thus, the more steps there are in a computation, the more often these jumps in (interpreted) value can occur. Of course, this phenomenon also allows the analysis to be greatly simplified.

Working with an arbitrary integer base β instead of 10 (the latter was chosen in the previous section only for convenience) can help to soften the effect of jumps in value if a base smaller than 10 is selected. This is because integers having the same number of base β digits will form more and smaller sets as β decreases, thus producing finer divisions of the integers. Therefore, expression swell bounds can be made more precise and jumps in value can be lessened (for instance, 1111_{10} would jump to 9999_{10} using the decimal definition $\mathcal{N}_{10}(n)$, but it would only jump to 2047_{10} with a binary definition, where $\mathcal{N}_2$ would count the number of bits in n).

Another factor in why the expression swell arithmetic developed here sometimes tends to greatly overestimate expression growth is that no provision has been (or is easily able to be) made to account for the effects of subtractive cancellation (which can be significant for procedures such as the solution of a triangular system of equations) and cancellation of

m	$+$	$\times$	GCD	iterations	GCD $= 1$
1	.00253	.00259	.02161	1.982	0.617
2	.00253	.00268	.02268	3.951	0.595
3	.00264	.00278	.02303	5.837	0.594
4	.00263	.00279	.02349	7.811	0.587
5	.00440	.00283	.02545	9.781	0.617
6	.00268	.00281	.02353	11.785	0.614
7	.00272	.00490	.02378	13.777	0.617
8	.00270	.00287	.02377	15.723	0.611
9	.00270	.00288	.02589	17.525	0.619
10	.00278	.00294	.02437	19.304	0.615

Table 3.12 The number of seconds needed to add, multiply or take the GCD of a pair of random m-digit integers in MAPLE 4.1 running on a Sun 3/160, where the results have been averaged over 1000 trials. The GCDs were computed using the Euclidean algorithm, taking the average number of iterations listed. The last column displays the fraction of the GCDs found to be unity.

common factors in non-exact quotients. A theorem from number theory due to Dirichlet states that the probability that two integers chosen at random are relatively prime is $6/\pi^2$ or about 60.8% [Knu81, p. 324]. (In Table 3.12, the results of doing this experiment in MAPLE are shown, along with some relevant timing data.) Performing rational arithmetic operations again and again on elements of a given matrix involves numbers that are, after a while, far from random and so GCDs play a significant role, as was seen in Table 3.10.

The other major reason why the rational worst case expression swell arithmetic often did so poorly is that rational operations involve extensive integer arithmetic, so that the jump in value effect is greatly magnified. Nonetheless, this arithmetic did provide bounds on the size of other smaller expressions involved in the calculations which were more reasonable, as well as indicating the relative ranking by size of the various expressions present at a given stage (e.g., matrix elements or polynomial coefficients). These last remarks also apply to integer expression swell arithmetic.

Worst case expression swell arithmetic is an attempt to effectively systematize bounding calculations for a certain class of problems, while providing a great deal of flexibility in their application (e.g., the elements in a matrix need not be considered initially uniform in size in order to obtain a bound). In this chapter, only numerical computations were considered, but these or like techniques could also be applied to the coefficients and exponents of polynomial or other calculations. One has to be careful with worst (or best) case analyses for complicated algorithms, as what is worst case at one stage may be best case (or more usually, a mix) at another. Finally, these analyses are useful for not only bounding expression swell (and thus computer memory usage), but also for setting limits on CPU time consumption. For example,

the time required to multiply n_1 by n_2 using the simplest algorithm is $O(\mathcal{N}_\beta(n_1) \cdot \mathcal{N}_\beta(n_2))$ [Akr88].

Chapter 4

A Symbolic Matrix Package

A very useful feature of a CAS is the ability to operate on matrices. Many sorts of operations can be implemented. It is desirable that these operations be well integrated into the rest of the CAS. Moreover, since matrix operations are often generalizations of scalar ones, operators need to be able to handle matrix as well as scalar expressions. This includes mixed mode expressions that involve both scalars and matrices.

All major CASs have some ability to deal with matrices. At least one major numerical package, MATLAB [Mol80], has a prime emphasis on matrix operations and a well-designed interface so that these operations can be applied in a reasonably 'natural' manner. In MATLAB, it is quite easy to extract rows, columns and subblocks of matrices in various combinations, and rearrange the results into new matrices. (This is encouraged by allowing vector indices to select a sequence of objects, such as matrix elements.)

For example, suppose A is a 4×4 matrix. Then the MATLAB statement

```
B = [A(1:3,2:4), A([1,2,4],[3,1,4]); A, [A(1:2,3:4); A([4,1],[3,2])]]
```

will define B to be the 7×6 matrix

$$B = \left(\begin{array}{ccc|ccc} a_{12} & a_{13} & a_{14} & a_{13} & a_{11} & a_{14} \\ a_{22} & a_{23} & a_{24} & a_{23} & a_{21} & a_{24} \\ a_{32} & a_{33} & a_{34} & a_{43} & a_{41} & a_{44} \\ \hline a_{11} & a_{12} & a_{13} & a_{14} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} & a_{43} & a_{42} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{13} & a_{12} \end{array}\right).$$

Square brackets (`[ ]`) delimit matrices and vectors in MATLAB, semicolons (`;`) separate rows, and colons (`:`) are used to make integer sequences (e.g., `1:3` is equivalent to `[1,2,3]`). Such notation can be used on the left side of assignment statements as well. For instance, `B([2,5],:) = B([5,2],:)` will swap rows 2 and 5 of B (the colon here expands to all relevant columns). This flexibility in manipulating matrices (along with the apparent ease in doing so) should be a vital ingredient in any system that deals with matrices.

Generalized operations are also handled well in MATLAB. For example, exp(x) produces the exponential of x, for x either a scalar or a (square) matrix. In contrast, if x is a vector, then exp(x) takes the exponential of each element. Other operations act similarly. Again, this is a good model to be imitated by other systems.

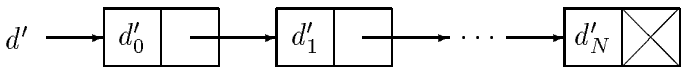
In this chapter, various ideas involving the design of a symbolic matrix package will be considered. These ideas will be examined in the light of existing packages present in current CASs (and programs like MATLAB), with an emphasis on determinant and eigenproblems. In this regard, Appendix D surveys what functions and methods are available in most of the major general purpose CASs. Development has been rapid in the last few years, and so this list expands every time a new release of a CAS appears, making it difficult to keep this summary up-to-date!

In addition, I have developed an extension to MACSYMA's matrix package (see [Bog83, Com88] and Appendix D), in which some of the methods of the previous chapters and some of the ideas to be discussed below were implemented. This collection of functions and variables is called "The Matrice Package"[1] and its MACSYMA level interface is described in Appendix E. This package is written in Lisp [Fod83] (in which language MACSYMA is itself written).

4.1 Data Structures

A fundamental issue in the design of a symbolic matrix package is the choice of the data structures to be used for representing various objects. For a given mathematical object, the data structure should be chosen so that the object can be easily manipulated, in particular, allowing rapid access to its components. The data representation should also be economical spacewise, with redundant information minimized. This latter is especially important since expression swell is such a common phenomenon, as was seen in the last chapter.

As an example, in CASs infinite precision integers are typically represented by linked lists (arrays are sometimes used, but these have the disadvantage of imposing a static limit on the length of an integer). A linked list, in its most basic form, consists of a linearly connected series of nodes, where each node contains some item of data and a pointer to the next node in the line. Thus, if the item of data is a base β digit, then the integer $d' = \sum_{k=0}^{N} d'_k \beta^k$ [see equation (2.25)] can be represented by the linked list



Note that the last node has a null pointer (marked by a cross). The Lisp language provides the alternative dotted-pair notation[2] for this structure given by

$$d' \rightarrow (d'_0 \ . \ (d'_1 \ . \ (\ldots \ . \ (d'_N \ . \ \text{nil})))) \ . \tag{4.1}$$

[1] "Matrice" is *a* singular of matrices, and this term is used to indicate a data structure implemented in MACSYMA as opposed to a mathematical matrix.

[2] The Lisp functions car and cdr, if applied to the dotted-pair $(a \ . \ b)$, will return a and b, respectively, and thus compositions of these functions can be used to select any data item in the linked list pointed to by d'.

In this case, the null pointer is designated by nil. (4.1) is conventionally written in a more economical manner in terms of a list by

$$d' \rightarrow (d'_0 \ d'_1 \ \ldots \ d'_N) \ ,$$

where here the terminating null pointer is implied by the right parenthesis.[3]

The data structure for a rational number r can then simply be represented as $r \rightarrow (n \ . \ d)$, where the numerator n and the denominator d are both infinite precision integers. As an example, if $\beta = 100$ and $r = \frac{327}{19550}$, then the corresponding representation of r will be given by

$$r \rightarrow ((27 \ 3) \ . \ (50 \ 95 \ 1)) \ .$$

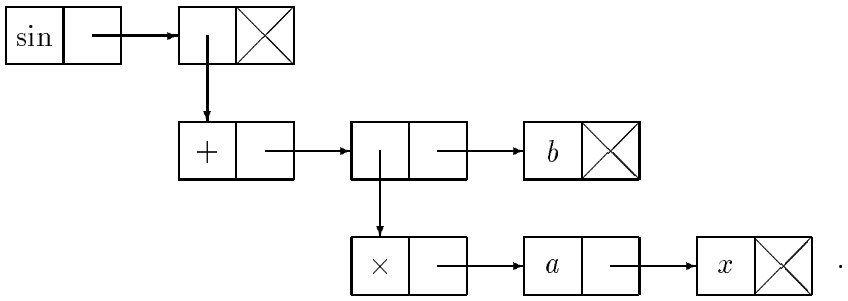
The above notions can be extended to the representation of polynomials and rational functions. Lisp prefix operator notation lends itself naturally to this task. A function application in Lisp is written as a list in which the first element is the operator and the rest of the elements are its operands.[4] The operands may be atomic (e.g., x) or may be themselves function applications. For example,

$$(\sin \ \underbrace{(+ \ \overbrace{(\times \ a \ x)} \ b)}) \tag{4.2}$$

is the prefix operator equivalent of $\sin(ax + b)$. By the way, (4.2) can also be displayed in dotted-pair notation as

$$(\sin \ . \ (\underbrace{(+ \ . \ (\overbrace{(\times \ . \ (a \ . \ (x \ . \ \text{nil})))}^{} \ . \ (b \ . \ \text{nil})))}^{} \ . \ \text{nil})) \ ,$$

and as a recursive application of linked lists by



Most of the major, general purpose CASs use some variant of the above data structure type to represent polynomials, as well as more general expressions (typically, the denominators of rational functions are taken to be multiplicative factors raised to a negative power). For instance, in MACSYMA, an expression with an internal form of the type exhibited in

[3]The car and cdr of d' in list-notation will yield, respectively, d'_0 and $(d'_1 \ \ldots \ d'_N)$. Note that nil is considered equivalent to the empty list (), and so $(a \ . \ \text{nil})$ and (a) both represent the same data structure (they have the same cars and cdrs).

[4]Thus, the car of a function call is the function, while the cdr is the list of its arguments.

(4.2) is said to be in "general representation".[5] The majority of the calculations presented in Appendix B involved operations on expressions in general representation or the equivalent for other CASs.

In MACSYMA, there is, in addition, a specialized data structure especially suitable for expanded (or partially factored) polynomials and rational functions. A polynomial in CRE (Canonical Rational Expression) form is represented solely by its variable, nonzero coefficients and the corresponding exponents. For example, the CRE form of $x^7+4x^3-2x+11$ is

$$((x\ 7\ 1\ 3\ 4\ 1\ -2\ 0\ 11)\ .\ 1)$$

(the 1 on the right-hand side of the dotted pair is the denominator, which will be non-unity for rational functions). This succinctness contrasts with the general representation equivalent

$$(+\ (\uparrow\ x\ 7)\ (\times\ 4\ (\uparrow\ x\ 3))\ (\times\ -2\ x)\ 11)\ .$$

Multivariate polynomials are generated by allowing coefficients to be CREs in additional variables; e.g.,

$$((x\ 3\ (y\ 2\ (z\ 1\ 4)\ 0\ 5)\ 1\ -7)\ .\ 1)$$

is the CRE representation of $(4zy^2+5)x^3-7x$ in which the variables are ordered as x, y, z in terms of 'main'ness. Further discussion will await the next section.

The next higher level of data abstraction encompasses lists, arrays and various other forms used to organize collections of data objects, such as groups of expressions or functions. This is where matrices finally come in. In most systems, matrix data structures are constructed from lists or arrays (MAPLE is an exception—see Table 4.1). Arrays provide faster accessing of their components (these are stored sequentially in memory so one does not have to follow pointers as is necessary with lists), but they have fixed dimensions (normally). The latter is usually not much of a disadvantage, however.

Typically, all the elements in a matrix are explicitly stored. For large matrices that exhibit sparsity and/or some sort of symmetry, this can be very wasteful of memory. A remedy is to store only non-redundant elements, such as the upper triangle of a Hermitian matrix or the nonzero bands of a banded matrix. This idea can be implemented in two different ways. For example, in MAPLE, only those elements that have been explicitly defined are stored (along with their indices). The indices are hashed (for rapid retrieval of the associated values) and the elements are accessed via an indexing function (for instance, symmetric will reorder the indices to be nondecreasing so that elements in the lower triangle will be mapped onto corresponding elements in the upper triangle).

The alternative implementation is provided in the Matrice Package. A matrice is arranged as a one-dimensional array of one-dimensional arrays. In order to minimize redundant information, several different data structures exist, representing various kinds of matrices (see

[5]The actual MACSYMA internal representation of (4.2) is [Fat79]

((%sin simp) ((mplus simp) ((mtimes simp) $a $x) $b)),

but for simplicity's sake, gory details like these will be passed over as they are irrelevant to the discussion.

MACSYMA	• list of lists (generally) • 2-D array (internally in newdet, det, and determinant with RATMX:true
MAPLE	sparse collection of defined elements, internally hashed, associated with index bounds and an optional indexing function (e.g., symmetric, anti-symmetric, sparse [unassigned elements are assumed to be zero], diagonal, identity, sparsymm [sparse symmetric])
Mathematica	list of lists (however, internally, a list is implemented by an array)
MATLAB	segment of 1-D array
REDUCE	list of lists
Scratchpad II	1-D array of 1-D arrays

Table 4.1 Matrix data structures used in five different CASs and one numerical matrix package.

Figure 4.1). Associated with each type of matrice are its dimensions ($m \times n$ or $n \times n$), the half-bandwidth (if the associated matrix is banded—this is the number of super- or subdiagonal bands including the main diagonal), the mathematics (whether the matrice contains complex or exclusively real elements), and a flag indicating whether the matrice elements are numerical, CREs or general expressions. A function, transmute, is used to change between the various forms and modify the associated properties (except for the dimensions, which are not allowed to vary). This particular methodology seems to be a reasonable compromise in terms of space and time, while also providing good functionality. For instance, specifying the mathematics of a matrice permits simplified algorithms to be used when the mathematics is real (e.g., $|x|^2$ becomes simply x^2). This can be of considerable benefit for symbolic computations.

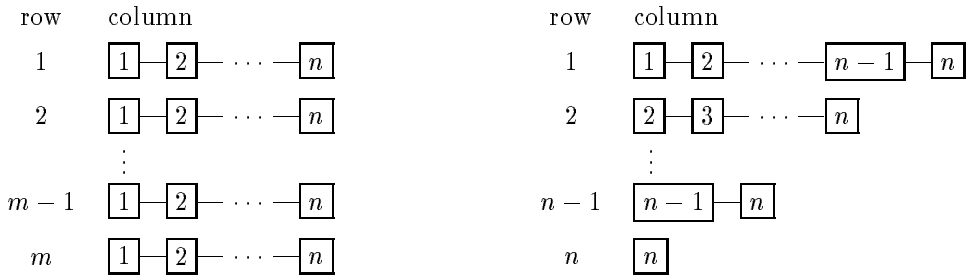
4.2 Arithmetic

An important consideration in the efficiency and accuracy of an algorithm is how arithmetic is performed on the inputs to (and the intermediate expressions generated by) the algorithm. These values may be numeric or symbolic, polynomial or rational, simple or complex. Often, they will be some mix of these choices. In this section, issues pertaining to performing arithmetic on these various possibilities will be discussed.

4.2.1 Numeric versus Symbolic

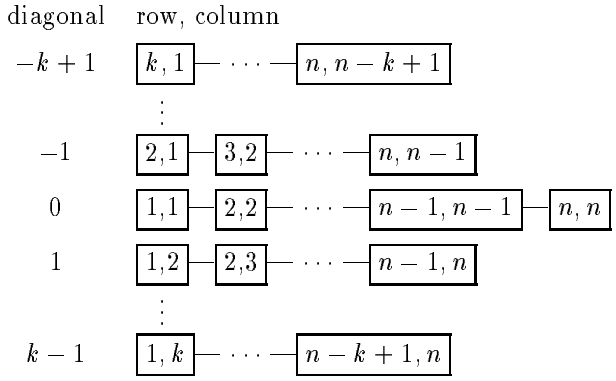
All of the algorithms discussed in Chapter 2 can operate on numeric (floating point) as well as symbolic quantities.[6] There are fundamental differences, however, between the arithmetic

[6]Here, symbolic quantities will include exact integers and rational numbers.

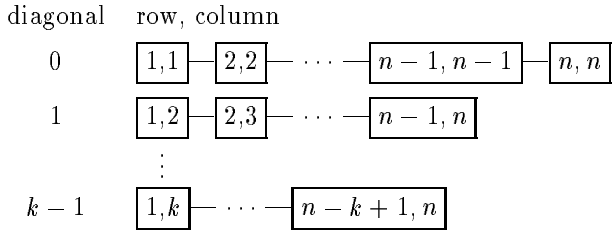


a. General

b. Hermitian

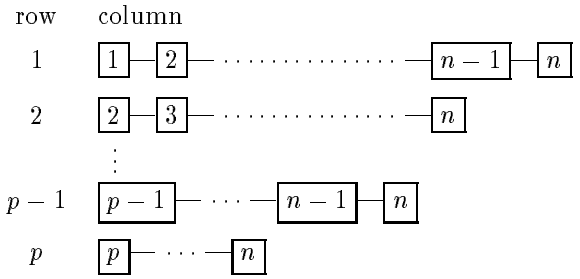


c. General banded

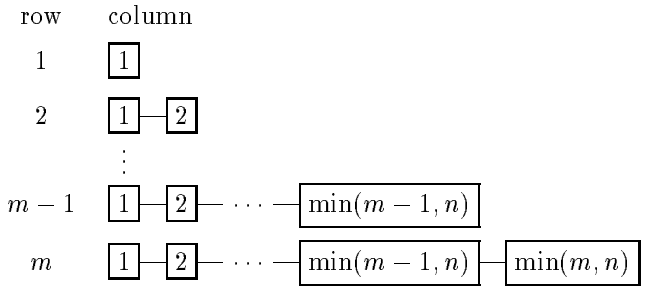


d. Hermitian banded

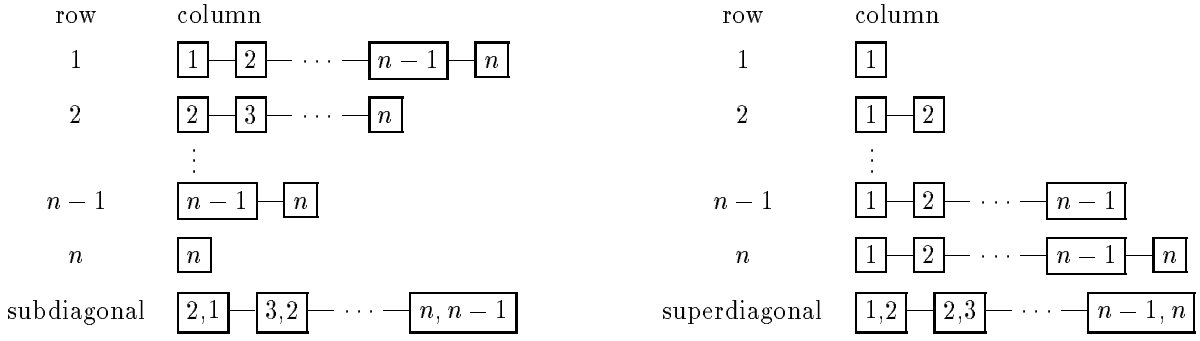
Figure 4.1 Matrice data structures.



e. Upper triangular, $p = \min(m, n)$



f. Lower triangular



g. Upper Hessenberg

h. Lower Hessenberg

Figure 4.1 Matrice data structures (concluded).

in these two contexts. One is normally inexact, while the other is always exact. Thus, to contend properly in each situation, adjustments will necessarily need to be made in the algorithms.

Probably one of the biggest differences concerns the detection of zeroes. In a floating point computation, a number is often taken to be zero if it is smaller in magnitude than the machine unit round-off. The question in a symbolic context is to determine when an expression (which may be quite complicated) is equivalent to zero. In the general situation, this problem is undecidable. However, there exist classes of expressions for which the detection of zero is always possible.

Knowledge of when an expression is zero is crucial since certain operations on an undetected zero (such as dividing by one) can lead to spurious results. For example, many CASs have had difficulty in the past recognizing the zero equivalence of $\cos^2\theta + \sin^2\theta - 1$ and related forms. Thus, it was quite possible to produce zero pivots unknowingly when performing eliminations on trigonometric matrices, for instance, such as those commonly associated with rotations.

Completely expanding a polynomial or the numerator of a rational function and collecting like terms will result in an expression that is either identically zero or definitely not equal to zero. Such a process puts these classes of expressions into what is known as a normal form, in which zero equivalence (but not expression equivalence) is trivially detectable. If, in addition, a regular ordering is imposed on the terms generated above (and the same process is applied to rational function denominators), then the equivalence of any two expressions can be trivially detected and the transformed expression will be in a canonical form. Therefore, as an example, tests for zero equivalence as well as expression equivalence are trivial operations in MACSYMA for polynomial and rational function CREs, but difficult (in fact, not possible in all situations) for more general expressions.

By the way, sometimes even detection of *syntactic* zeroes can be difficult in a CAS. One example is MACSYMA which has three basic 'zeroes': 0, 0.0 and 0.0b0. These correspond to integer, floating point and extended precision floating point zero, respectively. Now, in MACSYMA, an expression can be in general representation, CRE form, extended CRE (Taylor series) form or Poisson series form. Putting the three basic zeroes into the four different representations (using rat and taylor with KEEPFLOAT:true, and intopois) will produce nine[7] distinct internal representations for zero. Not all MACSYMA predicates will treat these forms equivalently when asked to determine their zero equivalence (in fact, most will *not*). Therefore, program coding must be done carefully to avoid unanticipated results in unusual circumstances. The Matrice Package function zerop (q.v.) was developed to quickly detect all syntactic forms that might reasonably exist for zero since no cheap way is available in MACSYMA to do just that.

The idea of approximate versus exact but possibly messy arithmetic has many repercussions. Typically, one can perform a floating point calculation and produce some sort of answer. (There are many exceptions, but this is true as a general rule.) It is quite possible, of course, that the answer obtained may be completely meaningless if care was not taken with

[7]Ten in MACSYMA 417.

the numerical analysis. For instance, lack of sufficient precision can lead to the loss of all significant digits in a calculation. In a matrix context, this may result in small residues but large errors (such as when solving a linear system of equations) or the loss of orthogonality (such as when computing eigenvectors) among many other examples.

Symbolic computations, on the other hand, frequently will not produce an answer. Some problems are completely intractable (e.g., finding the exact roots of a general 5^{th} degree polynomial), while others may fail as a result of space or time exhaustion due to expression swell. Even when a calculation succeeds, the outcome may often be so complicated as to be effectively useless.

Floating point operations are relatively fast (frequently, they are implemented in hardware) and so explicit zeroes are not given special treatment for the most part, except when it is necessary to avoid computational singularities. Commonly, the overhead of testing for zero is greater than the expense of just performing an operation directly (this is even more true for vector processors). Arithmetic performed on symbolic operands, however, is not nearly so cheap (especially when dealing with nonnumeric quantities), and so it is often worthwhile to check explicitly for zeroes in order to avoid unnecessary computations. Such checks are generally implemented by CASs as part of their basic arithmetic operations, although tests for zeroes at higher levels, where it is possible to exclude a whole sequence of calculations, are not uncommon.

A difference between numeric and symbolic computations that is important in matrix elimination methods is the choice of a pivot. In floating point procedures, numerical stability is promoted by selecting a pivot that is maximal in absolute value along a given column (or row). This is so because the ratio of a column element to the pivot [the factor $a_{ik}^{(k-1)}/a_{kk}^{(k-1)}$ in equation (2.8)] will then be less than or equal to one in absolute value, and so fractions of the pivot row will be subtracted from the other rows in the column undergoing elimination. These subtractions will minimize the loss of significant digits more so than under any other pivot choice, hence, this is the preferred pivot selection strategy for numerical elimination algorithms.

In symbolic elimination procedures, it is more beneficial to minimize the 'complexity' of a pivot (an operational definition of complexity will be given in Section 4.2.3). The reason for this is that no matter whether division-free or fraction-producing algorithms are employed, the pivot will tend to cause growth proportional to its size in the elements that are affected (but not zeroed) in the associated elimination stage (for Gaussian elimination, these will be the elements to the right of the pivot and below). Moreover, at each subsequent stage, this phenomenon will increase in intensity. Thus, the selection of a 'simple' pivot will always be a desirable goal when performing exact eliminations.

To conclude here, it is important to note that there are also many similarities in numeric and symbolic computations. The most fundamental is that in both types of activities, it is desirable to minimize space and time consumption. This implies, for instance, that certain algorithms like the method of Leverrier, which requires $O(n^4)$ additions and multiplications, will be inherently slow compared to $O(n^3)$ methods like the Hessenberg transformations. Of course, other characteristics of a given procedure may make it undesirable in a numeric

$+_n^r$	$3\times_n^p + 1 +_{2n}^p + 1\,\mathrm{GCD}_{2n+1,2n}^p$
$-_n^r$	$-_n^p$
$\times_n^r$	$2\,\mathrm{GCD}_n^p + 2\times_n^p$
$\div_n^r$	$2\,\mathrm{GCD}_n^p + 2\times_n^p$
$\overset{\text{exact}}{\div}{}_n^r$	$2\ \overset{\text{exact}}{\div}{}_n^p$

Table 4.2 Rational operations expressed in terms of polynomial operations (superscripts r and p, respectively), assuming worst case expression swell. The subscripts refer to the size of the operands (e.g., n means that they are all n-digit numbers [rationals or integers]; m, n means that the first operand has m digits while the second has n digits). Also, note that $-$ denotes unary negation and $\mathrm{GCD}^p = 1$ actual $\gcd^p + 2\ \overset{\text{exact}}{\div}{}^p$ if $\gcd^p \neq 1$, where $\gcd^p$ indicates the physical operation of taking a polynomial GCD.

and/or symbolic setting, but algorithms possessing high operation counts and/or requiring large amounts of memory will definitely have little to favor them.

4.2.2 Polynomial versus Rational

In the algorithms discussed in Chapter 2, the primary operations are negation, addition, multiplication and division (both exact and nonexact). Typical operands are polynomial and rational expressions (frequently numbers). Thus, rational arithmetic is generally being performed, although for the division-free methods, the arithmetic will be pure polynomial if the initial data likewise is polynomial.

In general, rational arithmetic operations are significantly more complex than their corresponding polynomial counterparts, a rational operation typically being composed of two or more polynomial ones. The relationships can be seen explicitly in Table 4.2. One thing to note in this table is that many of the rational operations include the taking of polynomial GCDs. These are optional, but are often helpful in reducing the size of newly constructed numerators and denominators by detecting any common factors that may be present, which can then be divided out.

A bound on the time required to perform a (modular) polynomial GCD is given in [McC77, p. 6]. This bound can be expressed most easily in terms of the following generalized multivariate polynomial notation. Suppose p is a polynomial in the v variables $\mathbf{x} = (x_1, \ldots, x_v)$. Let $\mathbf{e} = (e_1, \ldots, e_v)$ be a vector of exponents corresponding to these variables. p can then be written as

$$p(x_1, \ldots, x_v) = \sum_{(e_1,\ldots,e_v)\in S_p} p_{e_1,\ldots,e_v} x_1^{e_1} \cdots x_v^{e_v} \ ,$$

or more concisely as

$$p(\mathbf{x}) = \sum_{\mathbf{e} \in S_p} p_{\mathbf{e}} \mathbf{x}^{\mathbf{e}} ,$$

where S_p is a finite set of distinct vectors (corresponding to the nonzero coefficients $p_{\mathbf{e}}$ of p) for which $\mathbf{e} \in S_p$ implies that $e_i \in \mathbf{Z} \geq 0$, $(i = 1, \ldots, v)$. The degree of x_i in p and the norm of p are defined by

$$\partial_i(p) \equiv \max_{\mathbf{e} \in S_p} e_i \quad \text{and} \quad \|p\| \equiv \sum_{\mathbf{e} \in S_p} |p_{\mathbf{e}}| ,$$

respectively.

The time needed to compute the $\gcd(p, q)$ for $p, q \in \mathbf{Z}_{p'}[x_1, \ldots, x_v]$ can now be given. This time will be of the order of

$$\mathcal{N}_\beta(K)(D+1)^v[\mathcal{N}_\beta(K) + D + 1] ,$$

where $K = \max(\|p\|, \|q\|)$ and $D = \max_{1 \leq i \leq v}(\partial_i(p), \partial_i(q))$. As a comparison, the time required to add two integers n_1 and n_2 is $O(\mathcal{N}_\beta(n_1) + \mathcal{N}_\beta(n_2))$, while the time needed to multiply them is $O(\mathcal{N}_\beta(n_1) \cdot \mathcal{N}_\beta(n_2))$.

By applying the results of Table 4.2 to the rational operation counts in Chapter 2, tables of polynomial operation counts can be obtained. Table 4.3 is one such example in which the entries in Tables 2.5a, 2.5c and 2.6 have been so converted. Of course, the actual run times of these algorithms depend not only on the operation counts, but also on the size of the operands. An estimate of these latter quantities can be obtained through the expression swell analyses of the previous chapter.

A final comment about rational arithmetic is that by default in most general-purpose CASs, putting the results of sums over common denominators and reducing rational expressions to lowest terms is not done automatically unless the quantities involved are strictly numerical. Normally, the user has to intervene explicitly when there are variables present. The advantage here is that rational operations are very quick and the user has a large amount of control over simplification. However, the user frequently makes only minimal use of this control and the expressions generated can become extremely messy.

On the other hand, full rational arithmetic is automatically performed when operating in REDUCE or with MACSYMA's CREs. Polynomials and rational functions are maintained in a canonical form and so zero detection and expression equivalence determination are trivial operations. There is a significant computational overhead, however, associated with this arithmetic, especially when dealing with large expressions. This overhead is due to the frequent GCDs which occur so that numerators and denominators of rational expressions remain relatively prime, and also the necessity to expand out any products that might appear in order to maintain the canonicality of polynomial expressions. If significant cancellation may occur under either of these operations (which is often the case for univariate calculations, in particular), then it may be very advantageous to perform full rational arithmetic. The time spent may be very much less than for quickly computing an enormously messy expression and then attempting to simplify it all in one fell swoop.

4.2.3 Simple versus Complex

It is always desirable, of course, to produce *simple* results. Unfortunately, except for simple problems (and often not even then) or lucky occasions, this usually will not happen. A very common phenomenon is expression swell, as was noted in the introduction to Chapter 3.

In many practical situations, the final answer, although not necessarily simple, may be much less complex than the intermediate results used to obtain it. This is the familiar concept of intermediate expression swell. The general strategy in performing a chain of calculations is to arrange the calculations so that expression growth is minimized at each step. The idea is that by producing simpler results at one step, the results of subsequent steps will also be simpler, and this will lead through a quicker and less resource-hungry path to the ultimate answer.

As an example of the above tactic, the various matrix reduction algorithms in the Matrice Package are designed to take advantage of zeroes already present in a matrix and to choose pivots that are as simple as possible. For instance, consider what happens in the implementation of the Hessenberg procedure. Suppose A is an $n \times n$ matrix and let $A^{(k)}$ designate the transformed matrix at the end of the k^{th} stage with $A^{(0)} \equiv A$ (this is all just as before—see Section 2.2.2). At the *beginning* of the k^{th} stage, one of two actions may occur depending on the value of $a_{k,k-1}^{(k-1)}$ (the element in the pivot position for the previous stage). A typical situation is given below for $n = 6$:

$$
\begin{array}{c} \\ k+1 \end{array}
\begin{array}{c}
\begin{array}{cccccc} \multicolumn{6}{c}{k} \end{array} \\
\left(\begin{array}{cc|c|ccc}
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
0 & \boxed{p_{k-1}} & \times & \times & \times & \times \\
\hline
0 & 0 & \times & \times & \times & \times \\
\hline
0 & 0 & \times & \times & \times & \times \\
0 & 0 & \times & \times & \times & \times
\end{array}\right)
\end{array}
$$

Here, p_{k-1} is a shorthand for $a_{k,k-1}^{(k-1)}$.

If p_{k-1} is zero, then columns k through n are searched. The objective is to find the column that maximizes the number of naturally occurring, off-diagonal zeroes located in rows numbered $k, \ldots, n$. (If more than one column has this property, then the one possessing the simplest future pivot will be selected—see below.) The concept is that this column (j) will be traded with the current column which is about to undergo elimination (k), and the more zeroes there are, the less work that will need to be done in the elimination. To preserve similarity, rows k and j will also have to be swapped, thus forming $A^{(k-1)'} = P_{kj} A^{(k-1)} P_{kj}$. An effect of this permutation will be to switch $a_{kk}^{(k-1)}$ and $a_{jj}^{(k-1)}$. In other words, diagonal elements will remain on the diagonal. The diagonal elements do not participate in the reduction, and so this is why these elements were bypassed in the search for the column containing the most zeroes.

If p_{k-1} is nonzero, then no column exchanges can be allowed as the corresponding row swap here would destroy the regular pattern of zeroes below the subdiagonal that had been

established during the previous stages. In this case, $A^{(k-1)'} = A^{(k-1)}$. This will be the more usual situation, by the way. In general, selecting a column for its zeroes is only assured to happen at the very beginning of the reduction. On the whole, zero pivots will be uncommon and will correspond to a partial factorization of the characteristic polynomial over the rational numbers.

Once a column has been chosen and has been put into place, the elements in rows $k+1$ through n are examined. The 'simplest' nonzero element, say $a_{ik}^{(k-1)'}$, will be selected to be the pivot and then swapped into the subdiagonal position. The corresponding column permutation will act on columns to the right of this (the k^{th}) column and so will have no effect on zeroes intentionally created in previous stages of the elimination. Symbolically, $A^{(k-1)''} = P_{k+1,i} A^{(k-1)'} P_{k+1,i}$ will be formed. If all the possible pivots are zero, then $A^{(k-1)''} = A^{(k-1)'}$. Next, $A^{(k-1)''}$ is renamed to be $A^{(k-1)}$ once again, and the elimination is finally permitted to begin.

The procedure for transforming a Hermitian matrix into tridiagonal form does not try to maximize zeroes, but does try to choose a good pivot (although not by default since all permutations must be explicitly applied to the matrix undergoing reduction in this algorithm—the pivoting strategy is controlled by the setting of the global variable MINHERMPIVOT). The implementation of the various Gaussian elimination methods of Section 2.1.2 in Appendix F also employ a variant of the above pivoting methodology. There, the pivot is in the diagonal position and only row exchanges need to be performed, but the quest for simplicity is still the essential characteristic.

So far, no definition of simplicity (or complexity) has been given. These terms refer to somewhat elusive concepts and so any statements about them will have a heuristic flavor. For example, consider the two expressions

$$\begin{aligned} (x-1)^{10} &= x^{10} - 10x^9 + 45x^8 - 120x^7 + 210x^6 - 252x^5 \\ &\qquad + 210x^4 - 120x^3 + 45x^2 - 10x + 1 \,, \\ x^{10} - 1 &= (x-1)(x+1)(x^4 - x^3 + x^2 - x + 1)(x^4 + x^3 + x^2 + x + 1) \,. \end{aligned}$$

Clearly, the forms on the left-hand side are smaller in size than the corresponding ones on the right-hand side. Does this make them simpler? If simplicity means minimizing memory usage (i.e., the number of bytes consumed), then the answer is yes. If, on the other hand, the simplicity of an expression refers to the ease with which certain mathematical operations can be performed on it, then the choice of the preferred form depends on the operation. Combining like terms when adding or subtracting works best when the expressions involved are expanded, while collecting or cancelling common factors when multiplying or dividing is easier to do when the expressions are factored. Note that of the two compact left-hand side expressions above, the first is fully factored while the second is completely expanded.

A nice example of an expression whose most compact arrangement is neither of the standard polynomial forms mentioned above is exhibited in Table 4.4. (The meaning of the numerical values for the complexity will be explained below.) Simplicity in one manner, thus, does not necessarily imply simplicity in other aspects. Nonetheless, expressions with terse structures are usually desirable in symbolic computations for a couple of reasons. Can-

H	$+$	$\frac{5}{6}n^3 - \frac{5}{2}n^2 + \frac{5}{3}n$
	$-$	$\frac{1}{3}n^3 - n^2 + \frac{2}{3}n$
	$\times$	$\frac{25}{6}n^3 - \frac{25}{2}n^2 + \frac{25}{3}n$
	GCD	$\frac{5}{2}n^3 - \frac{15}{2}n^2 + 5n$
S	$\times$	$n^2 - 3n + 2$
	GCD	$n^2 - 3n + 2$
S^{-1}	$+$	$\frac{1}{6}n^3 - n^2 + \frac{11}{6}n - 1$
	$-$	$\frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n$
	$\times$	$\frac{5}{6}n^3 - 5n^2 + \frac{55}{6}n - 5$
	GCD	$\frac{1}{2}n^3 - 3n^2 + \frac{11}{2}n - 3$

a. General into upper Hessenberg

T^*	$+$	$\frac{2}{3}n^3 + n^2 - \frac{23}{3}n + 6$
	$-$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 - \frac{5}{6}n + 1$
	$\times$	$\frac{10}{3}n^3 + 14n^2 - \frac{172}{3}n + 33$
	GCD	$2n^3 + 12n^2 - 42n + 21$
	conj	$\frac{1}{3}n^3 - n^2 + \frac{2}{3}n$
	Re	$\frac{1}{2}n^2 - \frac{3}{2}n + 1$
S	$+$	$n^3 - 4n^2 + 5n - 2$
	$-$	$\frac{1}{2}n^3 - 2n^2 + \frac{5}{2}n - 1$
	$\times$	$5n^3 - 20n^2 + 25n - 10$
	GCD	$3n^3 - 12n^2 + 15n - 6$
S^{-1}	$\times$	$4n^2 - 8n + 6$
	GCD	$4n^2 - 8n + 6$
	conj	n^2

b. Hermitian into tridiagonal (modified fast Givens')

$+$	$\frac{1}{6}n^3 + n^2 - \frac{7}{6}n$
$-$	n
$\times$	$\frac{7}{6}n^3 + 5n^2 - \frac{49}{6}n + 2$
GCD	$\frac{5}{6}n^3 + 3n^2 - \frac{35}{6}n + 2$

c. Standard upper Hessenberg characteristic polynomial

$+$	$\frac{3}{2}n^2 - \frac{3}{2}n$
$-$	n
$\times$	$\frac{17}{2}n^2 - \frac{21}{2}n + 2$
GCD	$\frac{11}{2}n^2 - \frac{15}{2}n + 2$

d. Standard tridiagonal characteristic polynomial

Table 4.3 Polynomial operation counts for selected rational transformation and characteristic polynomial algorithms applied to an $n \times n$ matrix.

Form	Expression	complexity
factored sum	$(x+1)^{10} - (x-1)^{10}$	13
expanded	$20x^9 + 240x^7 + 504x^5 + 240x^3 + 20x$	24
factored	$4x(x^4 + 10x^2 + 5)(5x^4 + 10x^2 + 1)$	25

Table 4.4 Three different forms of the same expression and their corresponding complexities.

Symbol	Classification	Complexity	Procedure
R	rational numbers (integers are considered to have a denominator of 1)	sum of the absolute value of the numerator plus the denominator	minimize
F	floating point numbers	absolute value of the number	maximize
B	extended precision floating point numbers (bigfloats)	absolute value of the number	maximize
E	expressions that are not simple numbers	size of the expression (the number of operators and atomic operands)	minimize

Table 4.5 The scheme used by complexity for rating the complexity of a MACSYMA expression, and the procedure for choosing the 'simplest' object within a given classification to act in the capacity of a pivot.

cellation in a big way, especially when dealing with multivariate formulas, typically will not be all that common of an occurrence, and so expressions will tend to expand as a calculation proceeds. Any stinting of this growth may make the difference between completing the calculation and giving up because of a lack of memory or of time. Also, large expressions are just more expensive to manipulate and/or manage than smaller ones.[8]

In the Matrice Package, the pivots in a matrix reduction are chosen to minimize structural 'complexity'. This is done by assigning a letter *classification* and a numerical *complexity* to every element that is examined and then selecting the one which has the simplest evaluation. This ordering scheme is detailed in Table 4.5. In short, the *classification* of an object gives a gross estimate of its complexity (for example, non-numbers are considered to be more complex than numbers), while the *complexity* of an object within a given *classification* provides a more refined evaluation of its complicatedness (for example, within the integers, 1 is simpler than 10). The criteria for simplicity implied by Table 4.5 were established in order that a simple object which would act as a pivot in multiple calculations (usually in the capacity of a divisor) could be easily and quickly found.

Note that floating point numbers are chosen so as to provide numerical stability during a floating point matrix reduction. In many CASs, such as MACSYMA, floating point numbers are 'contagious' over rational numbers, and in MACSYMA at least, extended precision floats are also contagious over machine precision floats (contagious in this context means that

[8]The latter condition is manifested in Lisp-based systems like MACSYMA by increased garbage collection (GC) times. A garbage collection occurs whenever Lisp decides that there is not enough usable storage left, and so it acts to claim back some of its previously employed space. On some memory intensive computations, the GC time can be a significant fraction or even exceed the CPU time.

```
(defun EXPRSIZE (e)
  (if (atom e) 1
      (do ((ee (cdr e) (cdr ee)) (k 1))
             ;the second case below occurs when a dotted pair is encountered
          ((atom ee) (if (null ee) k (1+ k)))
          (setq k (+ k (exprsize (car ee)))))))
```

Figure 4.2 Lisp code for calculating the size of the MACSYMA expression `e`. The size is an integer which counts the number of operators and atomic operands in the expression.

operations involving a contagion will produce results of the same form as the contagion). If a floating point number is converted into a rational number, it will typically be quite complex (this is even more true for an extended precision float) and so as a class, floating point numbers can be considered to be more complex than rationals.

The function complexity will compute the *classification* and *complexity* of any MACSYMA object using the scheme described in Table 4.5. One comment needs to be made about computing the size of a non-numerical expression. This value is calculated based on the internal form of the expression. (The Lisp function that does this is shown in Figure 4.2.) Some types of expressions are represented internally in a different way than they may normally be presented to a user, and so the computed values may not always be completely obvious, as might be the case for the first entry in Table 4.4, for example. Any confusion can be cleared up by realizing that $x - y$ is represented internally in MACSYMA (as well as in several other CASs) by $(+\ x\ (\times\ -1\ y))$. [Similarly, the internal form of x/y is $(\times\ x\ (\uparrow\ y\ -1))$.] This choice of internal representation makes for easier arithmetic and aids in simplification.

4.3 Available Operations

Any symbolic matrix package should have a wide variety of available operations. These should include things from simple matrix arithmetic to the computation of various canonical forms and invariant properties of matrices. Moreover, these calculations should, if possible, take advantage of any special matrix data structures that might be available. Finally, all operations should perform equally well with complex matrices as they do with real matrices.

To begin, it should be easy to create, delete and modify matrices of various types. Simple structural operations like row, column, diagonal and block extraction should all be readily available. Matrix elements should be individually accessible as well as modifiable. An extremely useful ability is to be able to map a function onto the elements of a matrix. This function may, in general, depend on the indices of the elements as well as on their values. More formally, if A is a matrix and $f_{ij}(x)$ is a function application, then $\mathrm{map}(f_{ij}, A)$ will yield a matrix of the same size as A with elements given by $f_{ij}(a_{ij})$.

The next operations of importance are those concerned with basic matrix arithmetic. The most fundamental are multiplication by a scalar, and the addition, subtraction and multiplication of matrices of compatible dimensions. In a decent implementation, these operations should try to take advantage of known matrix structure as much as possible. For instance, the addition or subtraction of matrices of the same form will often result in a matrix also of this form. Other examples include: multiplying a matrix with a specified pattern of zeroes by a diagonal matrix will cause the given zero structure to be preserved, the product of two not necessarily square lower (or upper) triangular matrices will be a lower (or upper) triangular matrix, multiplying together two block diagonal matrices with the same block structure will yield a third matrix of like structure, and so forth. Thus, it is quite possible to avoid many unnecessary operations (mostly involving known zeroes) by just keeping track of some basic matrix types.

At this point, a comment about vectors is appropriate. All matrix packages can form column and row vectors from $n \times 1$ and $1 \times n$ matrices. Many of them will also treat lists or one-dimensional arrays as vectors in operations involving matrices. However, these latter objects do not always behave in a mathematically consistent manner. For example, in both MACSYMA and Mathematica, a list can sometimes act as a row vector and sometimes as a column vector depending on the context (this is most noticeable when multiplying a matrix and a list together, with the list first on the left and then on the right). On the other hand, MAPLE always takes a one-dimensional array to be a column vector. The former treatment can certainly be convenient at times, but the potential for confusion and errors is ever present, especially for complicated formulae, so the approach taken by MAPLE is generally to be preferred.

A very common and important operation that is available in almost every matrix package is the transpose. A less common, but still important, operation that is available in almost no (symbolic) matrix package (the collection of functions in Appendix E is an exception) is the Hermitian or complex conjugate transpose. This lack is not so surprising considering that CASs in general do not deal really well with complex numbers (several of the major ones do not even have a built in complex conjugate function). Any decent matrix package needs to be able to handle complex matrices in as natural a manner as it does for matrices with real entries. In MATLAB, for instance, the natural operation is the Hermitian, and then when the matrices are real, this will specialize into the transpose.

Computing the norm of a matrix is another useful operation that is typically absent in CASs, but common in numerical matrix packages. The norm of a matrix A, $\|A\|$, is a single nonnegative real number that characterizes the overall size of a matrix. There are various sorts of norms with the 2-norm (e.g., $\|A\|_2$) having the most interesting theoretical properties, but unfortunately also being among the most difficult to compute.[9] The norms typically used in numerical analysis, although not quite so nice theoretically, are easy to calculate and should be available in any matrix toolkit. In particular, for A an $m \times n$

[9]The $\|A\|_2$ is essentially the maximum stretch factor supplied by A when A acts as a linear transformation in a finite dimensional vector space. $\|A\|_2$ is computed as the square root of the largest eigenvalue of $A^H A$, that is, the largest singular value of A.

matrix, this set should include

$$\begin{aligned} \|A\|_1 &\equiv \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| \ , \\ \|A\|_\infty &\equiv \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}| \ , \\ \|A\|_F &\equiv \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2} \ . \end{aligned}$$

All of the above matrix norms are implemented by the norm function in the Matrice Package. $\|A\|_F$, also known as the Euclidean, Frobenius or Schur norm, can also be defined by $\sqrt{\mathrm{tr}(A^H A)}$. The equivalence of the two forms can be seen by noting that $(A^H A)_{ij} = \sum_{k=1}^{m} \bar{a}_{ki} a_{kj}$, therefore,

$$\mathrm{tr}(A^H A) = \sum_{j=1}^{n}\sum_{k=1}^{m} \bar{a}_{kj} a_{kj} = \sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2 \ .$$

Another quantity of interest, derivable from the norm but more expensive to compute, is the condition number of a matrix: $\mathrm{cond}(A) \equiv \|A\|\,\|A^{-1}\|$. Under the 2-norm, this is equal to the ratio of the largest singular value of A to the smallest one, and so the 2-norm condition number is a measure of the maximum distortion induced by A when A acts in the capacity of a linear transformation.

The next capability of importance for a matrix package is the ability to solve the linear system $AX = B$ for X, where in general, the coefficient matrix A can be $m \times n$ and the matrix B of right-hand sides can be $m \times \ell$ (so the matrix X of unknowns will be $n \times \ell$). Solutions can be unique, not exist at all, or depend on one or more arbitrary parameters. The last possibility can be handled especially well by a CAS. If $m = n$ then $AX = B$ will have a unique solution if and only if $\det A \neq 0$. If this condition is true and $B = I$, then solving the linear system will produce $X = A^{-1}$. This is probably the best general method to compute A^{-1}, although for many special matrix forms, simpler procedures are known (and should be used when practical).

$AX = B$ is normally solved by performing some form of Gaussian elimination, the details being very similar to computing $\det A$ by such methods. Indeed, by initially setting $C = (A|B)$ in Figures 2.2, 2.3 and 2.4, and extending the upper limits on all the j-loops from n to $n + \ell$, all that is necessary to complete the fraction-free solution of $A(Z/\det A) = B$ for the pair $(\det A, Z)$ is the back substitution algorithm listed in Figure 4.3. All this is actually done in Appendix F for the $n \times n$ matrix A.

The effect of the elimination and back substitution procedures, as with the more common, fraction-producing variety [see (2.8)], is equivalent to decomposing A into the product LU, where L is lower triangular and U is upper triangular. Solving $AX = B$ is thus turned into computing the 'triangular decomposition' of A (which requires most of the labor) and the solution of two simpler problems involving only back substitution: $LY = B$ and $UX = Y$.

for $i = n-1$ **to** 1 [**step** -1]
  for $j = n+1$ **to** $n+\ell$

$$c_{ij} = \left[(\det A) c_{ij} - \sum_{k=i+1}^{n} c_{ik} c_{kj} \right] \Big/ c_{ii}$$

for $i = 1$ **to** n
  for $j = 1$ **to** ℓ
    $z_{ij} = c_{i,n+j}$

a. Algorithm

$+$	$\left(\frac{1}{2}n^2 - \frac{1}{2}n\right)\ell$
$-$	$(n-1)\ell$
$\times$	$\left(\frac{1}{2}n^2 + \frac{1}{2}n - 1\right)\ell$
exact $\div$	$(n-1)\ell$

b. Corresponding rational operation counts

Figure 4.3 Fraction-free back substitution algorithm to determine $Z = (\det A)X$, where X is the solution to the linear system $AX = B$ with known matrices A $(n \times n)$ and B $(n \times \ell)$. C, initially $(A|B)$, has already undergone fraction-free Gaussian elimination.

The Gaussian elimination algorithm simplifies greatly for certain special forms of A. In particular, if A is tridiagonal (a not infrequent situation, especially when dealing with finite difference grids), then the reduction can be performed in place on a tridiagonal data structure representation of A, like the one in Figure 4.1c with $k = 2$ (see [Str74, p. 39]). Such exploitation of matrix structure should occur whenever possible since the solution of linear systems of equations is an extremely common operation in many applications.

An especially interesting linear system is $A\mathbf{x} = \mathbf{0}$. The solution of this equation yields the null space of A whose dimension is equal to the column dimension (n) of A (taken to be $m \times n$) minus the rank of A. By continuing to apply the elementary row operations in (2.6), A can be put into a convenient form as a reduced row echelon matrix. Such a matrix has rows that are either all zero or whose first nonzero entry is a one. The columns containing these ones will have no other nonzero entry and correspond to the linearly independent column vectors comprising the original matrix (their number will also equal A's rank). The entries in the remaining columns indicate the exact form of the linear dependence of the corresponding vectors composing A in terms of the linearly independent vectors. Thus, much useful information about both the range and null space of A can be obtained from its reduced row echelon form, and so this reduction is typically available in CAS matrix packages.

Probably the most important attributes of a (square) matrix are its invariants under similarity transformations. The most prominent of these are the trace, determinant, characteristic polynomial, eigenvalues and eigenvectors. The trace is simple to calculate, and algorithms to compute the determinant and characteristic polynomial were covered in great

detail in Chapter 2. (Calculating arbitrary minors is also covered by these procedures.[10]) The eigenvalues can be obtained by factoring the characteristic polynomial over the complex numbers. The eigenvectors can then be determined by finding the null space of $A - \lambda I$ for each distinct eigenvalue λ of A.

In practice, CASs can factor polynomials only to a limited extent. The factoring algorithms can find all rational roots of univariate polynomials. In most systems, factoring over specified algebraic field extensions is also available so, in particular, factoring is easily extended to the complex rationals. Roots of irreducible factors over this latter field can be computed for polynomials of degree 2, 3 and 4 via the quadratic, cubic and quartic formulas. These factors, by the way, can be taken to be square free, for if $p(x)$ is monic and has r distinct roots $\{\lambda_i\}_{i=1}^r$ with λ_i of multiplicity m_i [so that $p(x) = \prod_{i=1}^r (x - \lambda_i)^{m_i}$], then the associated polynomial

$$\tilde{p}(x) \equiv \frac{p(x)}{\gcd(p(x), p'(x))} = \prod_{i=1}^{r} (x - \lambda_i) \ ,$$

i.e., $\tilde{p}(x)$ will have all the same roots as $p(x)$, but each will only have a multiplicity of one [Knu81, p. 421]. (The prime indicates differentiation.) Therefore, the problem of factoring a general polynomial with possibly repeated roots can always be reduced to the task of factoring an associated polynomial with roots guaranteed to be distinct.

Even with all the techniques noted above, however, and additional specialized tricks, univariate polynomials cannot generally be factored completely, and if the coefficients contain parameters, then a numerical solution is not even possible. Moreover, applying any of the radical producing formulas often produces results that are so complicated as to be essentially useless (see Figure 3.1). In the Matrice Package by default, meigenvalues will solve for all the roots that it can, and return any remaining irreducible factors consolidated into a residual polynomial. Optionally, only roots of linear factors can be solved for explicitly. Also, a flag can be set (the global variable NUMEIGS) to numerically solve for the roots of any univariate residual polynomials produced by this process. Many current CASs provide a mix of these capabilities as well. In MAPLE, implicit zeroes of polynomials can be indicated through the use of the RootsOf operator, which is a particularly nice notation (since it has the potential of allowing further manipulations).

Probably the best way to symbolically compute the eigenvectors of A is to solve the characteristic equation $(A - \lambda I)\mathbf{x} = \mathbf{0}$ for each distinct eigenvalue λ of A. This works well if λ is of simple form. If λ is complicated or only known implicitly, then it is better (or

[10] Two related quantities that are occasionally useful are also—the permanent of a matrix (similar to the determinant, but with all terms added in its definition:

$$\text{perm } A \equiv \sum_{\mathbf{j} \in \pi_n} \prod_{i=1}^{n} a_{ij_i}$$

[cf. equation (2.2)]) and the adjoint, adjugate or matrix of cofactors of the matrix A, $(\text{adj } A)_{ij} \equiv \mathcal{A}_{ji}$ (the cofactor $\mathcal{A}_{ij}$ is defined in Section 2.1.1), which has the property that $A^{-1} = (\text{adj } A)/\det A$.

necessary) to work with λ as a symbolic parameter. This will involve solving the modular equation $(A - \lambda I)\mathbf{x} \equiv \mathbf{0} \pmod{p(\lambda)}$, where $p(\lambda) = 0$ implicitly defines the eigenvalues under consideration. Congruence $(\mathrm{mod}\ p(\lambda))$ is an extension of the notation introduced in Section 2.1.3, and indicates that polynomials of degree $\geq \deg(p(\lambda))$ $(= k$, say) can be reduced to degree $< k$ by iteratively replacing occurrences of λ^ℓ $(\ell \geq k)$ by $\lambda^{\ell-k}(\lambda^k - p(\lambda))$.

As an example of the latter procedure, suppose

$$A = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 5 & 4 \end{pmatrix} ,$$

then $c_A(\lambda) = (\lambda-3)(\lambda^2-5\lambda-6) = (\lambda-3)(\lambda+1)(\lambda-6)$. The eigenvectors corresponding to the eigenvalues -1 and 6 can be determined by solving $(A-\lambda I)\mathbf{x} \equiv \mathbf{0} \pmod{\lambda^2-5\lambda-6}$. A fraction-free reduction of the characteristic matrix to triangular form will proceed as follows:

$$\begin{pmatrix} 3-\lambda & 0 & 0 \\ 0 & 1-\lambda & 2 \\ 0 & 5 & 4-\lambda \end{pmatrix} \rightarrow \begin{pmatrix} 3-\lambda & 0 & 0 \\ 0 & 5(1-\lambda) & 10 \\ 0 & 0 & (1-\lambda)(4-\lambda)-10 \end{pmatrix}$$
$$\overset{\mathrm{mod}}{\rightarrow} \begin{pmatrix} 3-\lambda & 0 & 0 \\ 0 & 5-5\lambda & 10 \\ 0 & 0 & 0 \end{pmatrix} .$$

Thus, the components of $\mathbf{x}$ will satisfy $(3-\lambda)x_1 = 0$ and $(5-5\lambda)x_2 + 10x_3 = 0$. This implies that $x_1 = 0$ and $x_3 = \frac{\lambda-1}{2}x_2$, yielding the generic eigenvector $\mathbf{x} = (0, 1, \frac{\lambda-1}{2})^T$, where λ can be either -1 or 6.

In general, a matrix will not necessarily have a full set of eigenvectors. There will be only a single eigenvector associated with each Jordan block of repeated eigenvalues in the Jordan form of the matrix. However, every matrix will have a complete complement of generalized eigenvectors which can be determined systematically from the eigenvectors. A theoretical result is that the Jordan form J of an $n \times n$ matrix A is completely characterized (up to the order of the diagonal blocks) by the Segre characteristics $\{\lambda_i, m_i, s_i, \{\ell_{ij}\}_{j=1}^{s_i}\}_{i=1}^{r}$ [Cul72, p. 197]. Here, r is the number of distinct eigenvalues λ_i (so $r \leq n$), each of which is of algebraic multiplicity m_i. Also, s_i is the number of simple Jordan blocks associated with λ_i, the individual sizes being given by ℓ_{ij}. (A simple Jordan block $J_\ell(\lambda)$ is an $\ell \times \ell$ diagonally placed submatrix of J with λ's along its diagonal, ones along its super- [or sub-] diagonal and zeroes elsewhere.) Hence,

$$n = \sum_{i=1}^{r} m_i , \quad m_i = \sum_{j=1}^{s_i} \ell_{ij} , \quad c_A(\lambda) = \prod_{i=1}^{r} (\lambda - \lambda_i)^{m_i} .$$

Furthermore, if the ℓ_{ij} are ordered so that $\ell_{i1} \geq \cdots \geq \ell_{is_i} \geq 1$, $(i = 1, \ldots, r)$, then the minimum polynomial of A will be given by

$$m_A(\lambda) = \prod_{i=1}^{r} (\lambda - \lambda_i)^{\ell_{i1}} . \tag{4.3}$$

This is the polynomial of least degree for which $m_A(A) = 0$ is true. $m_A(\lambda)$ will be a divisor of $c_A(\lambda)$.

The total number of regular eigenvectors will be $\sum_{i=1}^{r} s_i \leq n$ (note that $s_i - 1$ will count the number of free parameters in the general expression for the eigenvector associated with λ_i). If $\ell_{i1} = 1$, $(i = 1, \ldots, r)$, then $J = \mathrm{diag}(J_{\ell_{11}}(\lambda_1), \ldots, J_{\ell_{rs_r}}(\lambda_r))$ will be diagonal and all eigenvectors will be regular. Otherwise, generalized eigenvectors $\mathbf{x}_{ijk}$ can be defined by $(i = 1, \ldots, r;\ j = 1, \ldots, s_i)$

$$\begin{aligned} (A - \lambda_i I)\mathbf{x}_{ij1} &= \mathbf{0} \\ (A - \lambda_i I)\mathbf{x}_{ijk} &= \mathbf{x}_{ij,k-1}\ , \quad (k = 2, \ldots, \ell_{ij}). \end{aligned} \tag{4.4}$$

(One has to make a proper choice for the $\mathbf{x}_{ij,k-1}$.) Equivalently (although not practical computationally), $\mathbf{x}_{ijk}$ will satisfy $(A - \lambda_i I)^k \mathbf{x}_{ijk} = \mathbf{0}$, $(k = 1, \ldots, \ell_{ij})$. The values held by ℓ_{ij} can be determined by noting for which values of k the second system of equations in (4.4) either becomes inconsistent or yields a solution which depends linearly on the previously computed independent generalized eigenvectors $\{\mathbf{x}_{ij1}, \ldots, \mathbf{x}_{ij,k-1}\}$.

The matrix formed from the complete set of generalized eigenvectors of A will be the similarity matrix S which transforms A into its Jordan from by $J = S^{-1}AS$. Knowing this matrix allows any sufficiently differentiable function of a matrix argument, $f(A)$, to be easily computed. This is so because the value of $f(A)$ can be defined as the value of the matrix polynomial $p(A)$, where $p(\lambda)$ is the Lagrange-Sylvester interpolation polynomial which satisfies

$$p^{(k)}(\lambda_i) = f^{(k)}(\lambda_i)\ , \quad (k = 0, \ldots, \ell_{i1} - 1;\ i = 1, \ldots, r)$$

[Gan60, p. 96]. Note that $p(\lambda)$ interpolates $f(\lambda)$ at values which correspond to the eigenvalue based zeroes of the minimum polynomial (4.3) and its derivatives [i.e., those k for which $m^{(k)}(\lambda_i) = 0$ necessarily, $(i = 1, \ldots, r)$]. $p(\lambda)$ will be of finite degree $(< \sum_{i=1}^{r} \ell_{i1})$, so $p(A)$, and hence $f(A)$, is well defined. Therefore, since

$$A^k = (SJS^{-1})^k = (SJS^{-1})(SJS^{-1}) \cdots (SJS^{-1}) = SJ^kS^{-1}\ ,$$

then

$$f(A) = p(A) = p(SJS^{-1}) = Sp(J)S^{-1} = Sf(J)S^{-1}\ ,$$

and the problem can thus be reduced to computing $f(J)$ with J being of simple form.

If A is block diagonal, of the form $A = \mathrm{diag}(A_1, \ldots, A_s)$, then $f(A) = \mathrm{diag}(f(A_1), \ldots, f(A_s))$ [Gan60, Chapter 5]. Hence, $f(J)$ can be easily calculated by further noting that for a simple Jordan block $J_\ell(\lambda)$,

$$f(J_\ell(\lambda)) = f\overbrace{\begin{pmatrix} \lambda & 1 & & O \\ & \ddots & \ddots & \\ & & \ddots & 1 \\ O & & & \lambda \end{pmatrix}}^{\ell} = \begin{pmatrix} f(\lambda) & \frac{f'(\lambda)}{1!} & \cdots & \frac{f^{(\ell-1)}(\lambda)}{(\ell-1)!} \\ & \ddots & \ddots & \vdots \\ & & \ddots & \frac{f'(\lambda)}{1!} \\ O & & & f(\lambda) \end{pmatrix}\ .$$

This result is derived by expanding $f(J_\ell(\lambda))$ in a Taylor series about λI:

$$f(J_\ell(\lambda)) = \sum_{k=0}^{\infty} \frac{f^{(k)}(\lambda)}{k!} (J_\ell(\lambda) - \lambda I)^k = \sum_{k=0}^{\ell-1} \frac{f^{(k)}(\lambda)}{k!} (J_\ell(\lambda) - \lambda I)^k \ .$$

The second equality follows from $J_\ell(\lambda) - \lambda I$ being a nilpotent matrix of index ℓ so that all powers of the matrix $\geq \ell$ will yield O. This final series is the Sylvester-Lagrange interpolating polynomial corresponding to the minimum polynomial $m_{J_\ell(\lambda)}(x) = (x - \lambda)^\ell$.

Probably the most important transcendental matrix function is the matrix exponential. This operation commonly arises when computing the solution of a system of linear, first order ordinary differential equations (ODEs). For example, defining $\mathbf{x}'(t)$ to be the componentwise derivative of $\mathbf{x}(t)$,[11] the initial value problem

$$\mathbf{x}'(t) = A\mathbf{x}(t) \ , \quad \mathbf{x}(0) = \mathbf{x}_0 \ . \tag{4.5}$$

will be solved by $\mathbf{x}(t) = e^{At}\mathbf{x}_0$, assuming A is independent of t. This can be seen by noting that e^{At} has the power series representation $\sum_{k=0}^{\infty} (At)^k/k!$, therefore,

$$\frac{d}{dt}(e^{At}) = \frac{d}{dt} \sum_{k=0}^{\infty} \frac{t^k A^k}{k!} = \sum_{k=0}^{\infty} \frac{kt^{k-1}A^k}{k!} = A \sum_{k=1}^{\infty} \frac{t^{k-1}A^{k-1}}{(k-1)!} = Ae^{At} \ .$$

The non-homogeneous problem can also be solved using matrix exponentials [Cul72, p. 268].

Although it is convenient to calculate a matrix function from its Jordan form, computing the Jordan form in the first place is often a very expensive process. One possible alternative is to use the procedure detailed in the proof of Schur's theorem to construct a triangular matrix unitarily similar to the original matrix (see the introductory paragraphs of Section 2.2.2). Now, since a matrix function can be defined in terms of a finite degree polynomial, and recalling that the sum or product of two compatible triangular matrices will still be triangular, it is clear that a function of a triangular matrix will again be triangular. Moreover, if T $(n \times n)$ is the matrix in question, it is clear that

$$\mathrm{diag}(f(T)) = \mathrm{diag}(f(t_{11}), \ldots, f(t_{nn})) \ . \tag{4.6}$$

The rest of the nonzero elements of $f(T)$ will be more complicated than was the case with $f(J)$, but they can still be computed by a straightforward procedure, this time involving recursive divided differences of the diagonal elements (4.6) [Mol76, Method 17].

Reduction of a matrix to either Jordan or Schur form is, of course, a non-rational process in general, and may not always be symbolically possible or desirable. Transformation of a matrix into Hessenberg form, as discussed in Section 2.2.2, involves a completely rational algorithm, however, and so it is worthwhile to consider what properties a function of a Hessenberg matrix may have. Suppose H is a standard $n \times n$ upper Hessenberg matrix. H will

[11] The derivative and integral of a matrix function are defined in terms of the derivatives and integrals of their elements: $(X'(t))_{ij} = x'_{ij}(t)$ and $(\int X(t)\,dt)_{ij} = \int x_{ij}(t)\,dt$. These operations should be readily available in any symbolic matrix package of worth.

commute with $f(H)$ as can be easily seen from the polynomial definition of $f(H)$. Therefore, if $F \equiv f(H)$, then $HF = FH$, or writing F in terms of its columns as $(\mathbf{F}_1|\cdots|\mathbf{F}_{n-1}|\mathbf{F}_n)$, this becomes

$$\begin{aligned}(H\mathbf{F}_1|\cdots|H\mathbf{F}_{n-1}|H\mathbf{F}_n) &= (\mathbf{F}_1|\cdots|\mathbf{F}_{n-1}|\mathbf{F}_n)\begin{pmatrix} h_{11} & \cdots & h_{1,n-1} & h_{1n} \\ s_1 & \cdots & h_{2,n-1} & h_{2n} \\ & \ddots & \vdots & \vdots \\ O & & s_{n-1} & h_{nn} \end{pmatrix} \\ &= \left(h_{11}\mathbf{F}_1 + s_1\mathbf{F}_2 \middle| \cdots \middle| \sum_{i=1}^{n-1} h_{i,n-1}\mathbf{F}_i + s_{n-1}\mathbf{F}_n \middle| \sum_{i=1}^{n} h_{in}\mathbf{F}_i\right) .\end{aligned}$$

Thus,

$$H\mathbf{F}_k = \sum_{i=1}^{k} h_{ik}\mathbf{F}_i + s_k\mathbf{F}_{k+1} , \quad (k = 1, \ldots, n-1), \tag{4.7}$$

so

$$\mathbf{F}_{k+1} = \frac{H\mathbf{F}_k - \sum_{i=1}^{k} h_{ik}\mathbf{F}_i}{s_k} = \frac{(H - h_{kk}I)\mathbf{F}_k - \sum_{i=1}^{k-1} h_{ik}\mathbf{F}_i}{s_k} , \quad (k = 1, \ldots, n-1).$$

Hence, if the first column of $f(H)$, $\mathbf{F}_1$, is known, then the remainder of the columns can be computed from the above recurrence. Notice that for $f(H) = e^H$, $\mathbf{F}_1$ is equal to $\mathbf{x}(1)$, where $\mathbf{x}(t) = e^{Ht}\mathbf{e}_1$ is the solution to (4.5) with $A = H$ and $\mathbf{x}_0 = \mathbf{e}_1$ (see also [Mol76, Method 13]).

A good matrix package should be able to transform matrices into various forms such as those noted above. Forms that require only rational arithmetic to compute are especially preferable when operating in a symbolic context. One such transformation of interest is the deflation of a Hessenberg matrix [Wil65, p. 465]. The idea here is that if some but not all of the eigenvalues of a Hessenberg matrix are known (say, k out of n), then this matrix can be similarily transformed so as to place the known eigenvalues on the diagonal, while simultaneously producing an $(n-k) \times (n-k)$ Hessenberg block, also lying on the diagonal, possessing the remainder of the eigenvalues. For example, in the 4×4 case, if H has eigenvalues $\{\lambda_i\}_{i=1}^4$, then the first application of the deflation process will go like

$$H \equiv H^{(0)} = \left(\begin{array}{cccc} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} \times & \times & \times & 0 \\ \times & \times & \times & 0 \\ 0 & \times & \times & 0 \\ \hline 0 & 0 & \times & \lambda_1 \end{array}\right) \equiv \left(\begin{array}{ccc|c} & H^{(1)} & & 0 \\ & & & 0 \\ & & & 0 \\ \hline 0 & 0 & \times & \lambda_1 \end{array}\right) .$$

The eigenvalues of $H^{(1)}$ will then be $\{\lambda_i\}_{i=2}^4$. If all of the eigenvalues are known, then this method will convert a Hessenberg matrix into near Jordan canonical form. For further details, see [Wil65].

A second purely rational conversion is the similarity transformation of a Hessenberg matrix into companion form. The companion matrix of the polynomial $p(\lambda) = \lambda^n + \sum_{k=0}^{n-1} a_k \lambda^k$,

$$C_{p(\lambda)} = \begin{pmatrix} 0 & 0 & \cdots & 0 & -a_0 \\ 1 & 0 & \cdots & 0 & -a_1 \\ & 1 & & \vdots & \vdots \\ & & \ddots & \vdots & \vdots \\ O & & & 1 & -a_{n-1} \end{pmatrix} ,$$

has the property that its characteristic polynomial $c_{C_{p(\lambda)}}(\lambda) = p(\lambda)$. The minimum polynomial, $m_{C_{p(\lambda)}}(\lambda) = p(\lambda)$ as well [Cul72, p. 237]. Moreover, due to the sparse structure of companion matrices, the relation (4.7) will take a particularly simple form for $H = C_{p(\lambda)}$:

$$\mathbf{F}_{k+1} = C_{p(\lambda)} \mathbf{F}_k \ , \quad (k = 1, \ldots, n-1).$$

Thus, companion matrices have many nice properties, and so make a worthwhile goal for rational transformations.

By the way, companion matrices can show up directly, as when converting an n^{th} order linear ODE into a system of first order ODEs. Suppose $x = x(t)$ and

$$\frac{d^n x}{dt^n} + a_{n-1} \frac{d^{n-1} x}{dt^{n-1}} + \cdots + a_1 \frac{dx}{dt} + a_0 x = 0 \ .$$

Making the substitutions

$$x_0 = x \ , \quad x_1 = \frac{dx}{dt} \ , \quad \ldots \ , \quad x_{n-1} = \frac{d^{n-1} x}{dt^{n-1}}$$

produces the system

$$\begin{aligned} \frac{dx_0}{dt} &= x_1 \\ &\cdots \\ \frac{dx_{n-2}}{dt} &= x_{n-1} \\ \frac{dx_{n-1}}{dt} &= -a_0 x_0 - a_1 x_1 - \cdots - a_{n-1} x_{n-1} \ , \end{aligned}$$

which has the matrix representation

$$\frac{d}{dt} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 & & & O \\ 0 & 0 & 1 & & \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & \ldots\ldots & & 1 \\ -a_0 & -a_1 & \ldots\ldots & & -a_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} \ .$$

The above matrix is simply $C^T_{p(\lambda)}$ and is also a companion matrix of $p(\lambda)$ in the sense that $c_{C^T_{p(\lambda)}}(\lambda) = p(\lambda)$.

Let H be an $n \times n$ standard upper Hessenberg matrix. To transform H into a companion matrix (C), it is necessary to first eliminate all the elements above the subdiagonal using the subdiagonal elements as pivots, and then to scale the resultant matrix so that the subdiagonal contains only ones. The reduction phase of this process will proceed through $n-1$ stages. As usual, let $H^{(k)}$ designate the transformed matrix at the end of the k^{th} stage with $H^{(0)} \equiv H$. The situation at the beginning of the k^{th} stage for a typical example in which $n = 6$ and $k = 3$ will look like

$$\begin{array}{c} \\ k+1 \end{array} \begin{array}{c} \begin{array}{cccccc} & & k & & & \end{array} \\ \left(\begin{array}{cc|c|ccc} 0 & 0 & \times & \times & \times & \times \\ \times & 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ \hline 0 & 0 & \times & \times & \times & \times \\ \hline 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{array}\right) \end{array} .$$

The pivot element will thus be $h^{(k-1)}_{k+1,k}$, and this is guaranteed to be nonzero since H was assumed to be *standard* upper Hessenberg.

The elimination will complete the path that was started in the reduction of a general matrix into upper Hessenberg form, with many of the same indices involved but with different ranges. Thus, the elimination step for the i^{th} row, $(i = 1, \ldots, k)$,

$$h^{(k)}_{ij} = h^{(k-1)}_{ij} - \frac{h^{(k-1)}_{ik}}{h^{(k-1)}_{k+1,k}} h^{(k-1)}_{k+1,j} , \quad (j = k, \ldots, n),$$

is the same as (2.33) except that the elimination is now occurring above the subdiagonal rather than below. In terms of the elementary row operations (2.7), and adding in the corresponding inverse elementary column operations for similarity, this becomes

$$E_{i,k+1}\left(-\frac{h^{(k-1)}_{ik}}{h^{(k-1)}_{k+1,k}}\right) H^{(k-1)} E_{i,k+1}\left(\frac{h^{(k-1)}_{ik}}{h^{(k-1)}_{k+1,k}}\right) ,$$

which is essentially the same as (2.34). Note that neither the row nor the column operations will interfere with the previously established pattern of zeroes.

This time around, the similarity matrix

$$S'^{-1} = \prod_{k=1}^{n-1} \left[\prod_{i=1}^{k} E_{i,k+1}(-r_{ik}) \right] ,$$

where $r_{ik} \equiv h^{(k-1)}_{ik} / h^{(k-1)}_{k+1,k}$, will be of a simple form. For example, in the case $n = 4$,

$$S'^{-1} = \begin{pmatrix} 1 & 0 & 0 & -r_{13} \\ 0 & 1 & 0 & -r_{23} \\ 0 & 0 & 1 & -r_{33} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -r_{12} & 0 \\ 0 & 1 & -r_{22} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & -r_{11} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & -r_{11} & -r_{12} & -r_{13} \\ 0 & 1 & -r_{22} & -r_{23} \\ 0 & 0 & 1 & -r_{33} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The structure of S' will also be upper triangular with ones along the diagonal, but it will be somewhat more complicated.

Now, $C' = S'^{-1} H^{(n-1)} S'$ will be in a companionlike form but with subdiagonal elements that are not necessarily unity. To produce a true companion matrix, a final scaling needs to be applied. If $p_k \equiv h_{k+1,k}$, $(k = 1, \ldots, n-1)$, then this scaling will be given by $C = D^{-1} C' D$, where $D = \mathrm{diag}(1, p_1, p_1 p_2, \ldots, p_1 \cdots p_{n-1})$. In more detail,

$$\begin{aligned} C &= \begin{pmatrix} 1 & & & & \\ & \frac{1}{p_1} & & O & \\ & & \ddots & & \\ & O & & \frac{1}{p_1 \cdots p_{n-2}} & \\ & & & & \frac{1}{p_1 \cdots p_{n-1}} \end{pmatrix} \begin{pmatrix} & & & & c'_0 \\ p_1 & & O & & c'_1 \\ & \ddots & & & \vdots \\ & & p_{n-2} & & c'_{n-2} \\ O & & & p_{n-1} & c'_{n-1} \end{pmatrix} \\ &\qquad \begin{pmatrix} 1 & & & & \\ & p_1 & & O & \\ & & \ddots & & \\ & O & & p_1 \cdots p_{n-2} & \\ & & & & p_1 \cdots p_{n-1} \end{pmatrix} \\ &= D^{-1} \begin{pmatrix} & & & & c'_0 p_1 \cdots p_{n-1} \\ p_1 & & O & & c'_1 p_1 \cdots p_{n-1} \\ & \ddots & & & \vdots \\ & & p_1 \cdots p_{n-2} & & c'_{n-2} p_1 \cdots p_{n-1} \\ O & & & p_1 \cdots p_{n-1} & c'_{n-1} p_1 \cdots p_{n-1} \end{pmatrix} \\ &= \begin{pmatrix} & & & c'_0 p_1 \cdots p_{n-1} \\ 1 & & O & c'_1 p_2 \cdots p_{n-1} \\ & \ddots & & \vdots \\ & & 1 & c'_{n-2} p_{n-1} \\ O & & & 1 \quad c'_{n-1} \end{pmatrix}. \end{aligned}$$

The complete algorithm for computing C, S and S^{-1} given H is presented in Figure 4.4a (cf. Figure 2.5). The corresponding operation counts are given in Figure 4.4b. Note that there are approximately the same number of additions and half as many multiplications here as for the characteristic polynomial algorithm described in Section 2.2.2 (see Table 2.6a), however, this procedure is not division-free which can be an important feature in some computations. The method of Danilewski [Wil65, p. 409] can be used to reduce a general matrix directly into companion form, and is essentially a combination of the algorithm for transforming a general matrix into upper Hessenberg form (Figure 2.5) and the above procedure.

Reduction to a Companionlike Matrix

$C = H$
$S = I$
$S^{-1} = I$
for $k = 1$ **to** $n - 1$
  for $i = 1$ **to** k
    $r_i = c_{ik}/c_{k+1,k}$
    $s_{i,k+1} = r_i$
    for $j = 1$ **to** $i - 1$
      $s_{j,k+1} = s_{j,k+1} + r_i s_{ji}$
    $s^{-1}_{i,k+1} = -r_i$
    $c_{ik} = 0$
    for $j = k + 1$ **to** n
      $c_{ij} = c_{ij} - r_i c_{k+1,j}$
    $c_{i+1,k+1} = c_{i+1,k+1} + r_i c_{i+1,i}$

Convert to True Companion Form

$p = 1$
for $i = 2$ **to** n
  $p = pc_{i,i-1}$
  $s_{ii} = p$
  for $j = 1$ **to** $i - 1$
    $s_{ji} = ps_{ji}$
  for $j = i$ **to** n
    $s^{-1}_{ij} = s^{-1}_{ij}/p$
$p = 1$
for $i = n - 1$ **to** 1 [**step** -1]
  $p = pc_{i+1,i}$
  $c_{i+1,i} = 1$
  $c_{in} = pc_{in}$

a. Algorithm

C	$+$	$\frac{1}{6}n^3 + \frac{1}{2}n^2 - \frac{2}{3}n$
	$-$	$\frac{1}{6}n^3 - \frac{1}{6}n$
	$\times$	$\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{4}{3}n - 2$
	$\div$	$\frac{1}{2}n^2 - \frac{1}{2}n$
S	$+$	$\frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n$
	$\times$	$\frac{1}{6}n^3 + \frac{5}{6}n - 1$
S^{-1}	$-$	$\frac{1}{2}n^2 - \frac{1}{2}n$
	$\div$	$\frac{1}{2}n^2 - \frac{1}{2}n$

b. Corresponding rational operation counts

Figure 4.4 Algorithm to similarily transform an $n \times n$ standard upper Hessenberg matrix H into companion form C ($C = S^{-1}HS$).

It is quite easy, by the way, given the eigenvalues of a companion matrix C and assuming that there is only one Jordan block associated with each eigenvalue in the Jordan form J, to construct the similarity matrix V that will transform C into J. V, usually referred to as a Vandermonde matrix, will take an especially simple form if all the eigenvalues of C, $\{\lambda_i\}_{i=1}^n$, are distinct: $v_{ij} = \lambda_j^{i-1}$ in this case [Wil65, p. 13]. This can be illustrated by looking at $J = V^{-1}CV$ or $VJ = CV$, for, say, $n = 3$. (Note that this C is simply the transpose of the companion matrix produced above.)

$$
\begin{aligned}
VJ &= \begin{pmatrix} 1 & 1 & 1 \\ \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1^2 & \lambda_2^2 & \lambda_3^2 \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix} = \begin{pmatrix} \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1^2 & \lambda_2^2 & \lambda_3^2 \\ \lambda_1^3 & \lambda_2^3 & \lambda_3^3 \end{pmatrix} . \\
CV &= \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -a_0 & -a_1 & -a_2 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1^2 & \lambda_2^2 & \lambda_3^2 \end{pmatrix} \\
&= \begin{pmatrix} \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1^2 & \lambda_2^2 & \lambda_3^2 \\ -(a_2\lambda_1^2 + a_1\lambda_1 + a_0) & -(a_2\lambda_2^2 + a_1\lambda_2 + a_0) & -(a_2\lambda_3^2 + a_1\lambda_3 + a_0) \end{pmatrix} .
\end{aligned}
$$

Equality of the two right-hand sides follows from $c_C(\lambda) = \lambda^3 + a_2\lambda^2 + a_1\lambda + a_0 = 0$ for $\lambda \in \{\lambda_1, \lambda_2, \lambda_3\}$. If J is not diagonal, then V will be more complicated, but not unreasonably so (see [Wil65, p. 15]). If more than one Jordan block is associated with any eigenvalue, then it will be necessary to transform C into a form where the Vandermonde matrices can operate blockwise, so that V will now be a block diagonal matrix with diagonal blocks Vandermonde matrices.

If H is not a standard upper Hessenberg matrix and so has some zeroes along its subdiagonal (say, ℓ in number), then the reduction discussed above will need to be broken down into $\ell + 1$ parts. The idea is that each standard Hessenberg block along the diagonal of H will be reduced to companion form, while the blocks of elements above the diagonal will be made zero (by simply starting the elimination process always at row 1 of H, all the elements above the subdiagonal of a companion block can be systematically zeroed—eliminating elements above the right-hand column of a companion block while also preserving similarity is not quite as straightforward, however). This decomposition into diagonal companion blocks produces a partial factorization of the characteristic polynomial.

The Frobenius or rational canonical form of a matrix is similar to the above, but corresponds to a complete factorization of the characteristic polynomial (within a specified field). The exact block structure will be determined by the Segre characteristics of the original matrix, however, it is also possible to derive the structure of the Frobenius form from the Smith canonical form of the characteristic matrix (see Section 2.2.4). Specifically, suppose A is an $n \times n$ matrix with elements contained in the field $\mathbf{G}$, and furthermore assume that $A - \lambda I$ has the Smith canonical form $\mathrm{diag}(f_1(\lambda), \ldots, f_n(\lambda))$. The invariant factors of $A - \lambda I$, $f_k(\lambda)$, $(k = 1, \ldots, n)$ will themselves be factorable as follows [remembering that $f_k(\lambda)$ divides

$f_{k+1}(\lambda)$, $(k = 1, \ldots, n-1)$]:

$$\begin{aligned} f_1(\lambda) &= p_1(\lambda)^{\ell'_{1n}} \cdots p_{r'}(\lambda)^{\ell'_{r'n}} \\ &\cdots \\ f_n(\lambda) &= p_1(\lambda)^{\ell'_{11}} \cdots p_{r'}(\lambda)^{\ell'_{r'1}} \ . \end{aligned} \tag{4.8}$$

Here, each $p_i(\lambda)$, $(i = 1, \ldots, r')$ will be of degree ≥ 1, distinct, monic and irreducible over $\mathbf{G}$. Also, $\ell'_{i1} \geq \cdots \geq \ell'_{in} \geq 0$, where there will be at least one strict inequality in this sequence. Note, by the way, that $r' \leq r$, where r is the number of distinct eigenvalues of A. The factors $p_i(\lambda)^{\ell'_{ij}}$, $(i = 1, \ldots, r';\ j = n, \ldots, 1)$ with nonzero exponent ℓ'_{ij} are commonly known as the elementary divisors of A.

Let $\{d_i(\lambda)\}_{i=1}^{t}$, $(t \leq nr')$ be the elementary divisors of A ordered as above. A can then be similarily transformed in operations purely rational over $\mathbf{G}$ into the block diagonal form $\mathrm{diag}(C_{d_1(\lambda)}, \ldots, C_{d_t(\lambda)})$ [see [Cul72, Chapter 7] for pertinent theorems concerning this and the following statements]. Algorithms to perform this reduction (in part) have been given previously. However, one can do even better. A is also similar to $F = \mathrm{diag}(Y_{d_1(\lambda)}, \ldots, Y_{d_t(\lambda)})$, where $Y_{d(\lambda)}$ is the hypercompanion matrix associated with the polynomial $d(\lambda)$. If $d(\lambda) = p(\lambda)^{\ell}$ for $\ell \in \mathbf{Z} > 0$ and $\deg(p(\lambda)) = m$, then $Y_{d(\lambda)} = Y_{p(\lambda)^{\ell}}$ will be the (ℓ block) $\times$ (ℓ block) matrix

$$Y_{p(\lambda)^{\ell}} = \begin{pmatrix} C_{p(\lambda)} & & & O \\ W & C_{p(\lambda)} & & \\ & \ddots & \ddots & \\ O & & W & C_{p(\lambda)} \end{pmatrix} ,$$

where W $(m \times m)$ is zero except for the upper right-hand element $w_{1m} = 1$. Hence, if the elementary divisors are written as in (4.8), then F will be in Frobenius canonical form with respect to $\mathbf{G}$. As an example, Figure 4.5 shows the Frobenius forms of a matrix when $\mathbf{G}$ is successively the rational numbers, the real numbers and the complex numbers, so that the invariant factors are broken down into more and more fundamental components. Note that the last matrix is also in (lower) Jordan canonical form.

A good characterization of the Frobenius form is that it is a generalization of the Jordan canonical form for fields where complete factorization of the characteristic polynomial may produce nonlinear, irreducible factors. If all the factors are linear, then $r' = r$, $p_i(\lambda) = \lambda - \lambda_i$ and $\ell'_{ij} = \ell_{ij}$, where the terms on the right-hand side were defined earlier in the discussion of the Segre characteristics. Thus, in this case, the Frobenius and Jordan forms are equivalent. Transforming a matrix into Frobenius canonical form is particularly appealing for CASs since only rational operations are needed (possibly supplemented by algebraic field extensions), and the resulting matrix will be as close to Jordan in character as it is possible to be in such a context.

If A is a non-square matrix (say, $m \times n$), then it will not have eigenvalues, however, there will be an associated set of singular values, $\{\sigma_i\}_{i=1}^{\min(m,n)}$. These values will be the diagonal elements of Σ, where Σ is a matrix of the same dimensions as A produced by a procedure known as the singular value decomposition (SVD). The SVD is an important

$$\text{diag}(0, \left(\begin{array}{cc|cc} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{array}\right), \left(\begin{array}{cc|cc} 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{array}\right), \begin{pmatrix} 0 & -3 \\ 1 & 0 \end{pmatrix}, -4)$$

a. $\mathbf{G} = \mathbf{Q}$ so elementary divisors are λ, $(\lambda^2+1)^2$, $(\lambda^2-2)^2$, λ^2+3, $\lambda+4$.

$$\text{diag}(0, \left(\begin{array}{cc|cc} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{array}\right), \begin{pmatrix} -\sqrt{2} & 0 \\ 1 & -\sqrt{2} \end{pmatrix}, \begin{pmatrix} \sqrt{2} & 0 \\ 1 & \sqrt{2} \end{pmatrix}, \begin{pmatrix} 0 & -3 \\ 1 & 0 \end{pmatrix}, -4)$$

b. $\mathbf{G} = \mathbf{R}$ so elementary divisors are λ, $(\lambda^2+1)^2$, $(\lambda+\sqrt{2})^2$, $(\lambda-\sqrt{2})^2$, λ^2+3, $\lambda+4$.

$$\text{diag}(0, \begin{pmatrix} -i & 0 \\ 1 & -i \end{pmatrix}, \begin{pmatrix} i & 0 \\ 1 & i \end{pmatrix}, \begin{pmatrix} -\sqrt{2} & 0 \\ 1 & -\sqrt{2} \end{pmatrix}, \begin{pmatrix} \sqrt{2} & 0 \\ 1 & \sqrt{2} \end{pmatrix}, -i\sqrt{3}, i\sqrt{3}, -4)$$

c. $\mathbf{G} = \mathbf{C}$ so elementary divisors are λ, $(\lambda+i)^2$, $(\lambda-i)^2$, $(\lambda+\sqrt{2})^2$, $(\lambda-\sqrt{2})^2$, $\lambda+i\sqrt{3}$, $\lambda-i\sqrt{3}$, $\lambda+4$.

Figure 4.5 The Frobenius canonical form under three different fields $\mathbf{G}$ of a 12×12 matrix with invariant factors $f_1(\lambda) = \cdots = f_{11}(\lambda) = 1$ and $f_{12}(\lambda) = \lambda(\lambda^2+1)^2(\lambda^2-2)^2(\lambda^2+3)(\lambda+4)$.

algorithm commonly found in numerical matrix packages (such as MATLAB) in which A is decomposed as $A = U\Sigma V^H$ where $U\,(m \times m)$ and $V\,(n \times n)$ are unitary (so that $U^{-1} = U^H$ and $V^{-1} = V^H$), and Σ is diagonal. The singular values will always be real and non-negative. Furthermore, the nonzero singular values of A will be the positive square roots of the nonzero eigenvalues of AA^H and A^HA as can be seen from

$$\begin{array}{rcl} AA^H & = & (U\Sigma V^H)(U\Sigma V^H)^H = U\Sigma V^H V\Sigma^T U^H = U(\Sigma\Sigma^T)U^H \\ A^HA & = & (U\Sigma V^H)^H(U\Sigma V^H) = V\Sigma^T U^H U\Sigma V^H = V(\Sigma^T\Sigma)V^H \end{array}, \tag{4.9}$$

[$\Sigma\Sigma^T$ $(m \times m)$ and $\Sigma^T\Sigma$ $(n \times n)$ will be diagonal, and clearly possess the same nonzero elements].

Note that if A is Hermitian (so $A^H = A$), then (4.9) implies that $A^2 = U\Sigma^2U^H = V\Sigma^2V^H$. Therefore, in this situation, the singular values of A will be the absolute values of the eigenvalues of A [if $\{\lambda_i\}_{i=1}^n$ are the eigenvalues of A, then $\sigma_i^2 = \lambda_i^2$ so $\sigma_i = |\lambda_i|$, $(i = 1, \ldots, n)$]. Thus, one can think of the SVD as a generalization (although not a perfect one) of the eigendecomposition of a Hermitian matrix applied to general rectangular matrices.

The SVD can be used to produce generalizations in other matrix realms as well. For example, by replacing A with $U\Sigma V^H$ in the general linear system $A\mathbf{x} = \mathbf{b}$ [yielding $(U\Sigma V^H)\mathbf{x} = \mathbf{b}$], the alternative system $\Sigma\mathbf{z} = U^H\mathbf{b}$ can be created with $\mathbf{x}$ now given by $V\mathbf{z}$. The original

system will often be over- or under-determined, and it will turn out [For77, Chapter 9] that this transformation will result in a direct solution which will be the best possible in a least squares sense (i.e., that minimizes the residual $\|A\mathbf{x} - \mathbf{b}\|$). If $p = \min(m, n)$, then this solution will be given by $(i = 1, \ldots, n)$

$$z_i = \begin{cases} \frac{(U^H \mathbf{b})_i}{\sigma_i} & (i \leq p \text{ and } \sigma_i \neq 0), \\ 0 & (i > p \text{ or } \sigma_i = 0). \end{cases}$$

As a second example, one can define the pseudoinverse of A, $A^\dagger$, to be the matrix that satisfies the Moore-Penrose conditions: $AA^\dagger A = A$, $A^\dagger A A^\dagger = A^\dagger$, $AA^\dagger$ and $A^\dagger A$ are Hermitian. If A is square and nonsingular, then $A^\dagger$ is simply A^{-1}. Otherwise, the SVD can be used to define $A^\dagger$ by $A^\dagger = (U\Sigma V^H)^\dagger \equiv V\Sigma^\dagger U^H$, where $\Sigma^\dagger$ is $n \times m$ and diagonal with diagonal elements given by $(i = 1, \ldots, p)$

$$\sigma_i^\dagger = \begin{cases} \frac{1}{\sigma_i} & (\sigma_i \neq 0), \\ 0 & (\sigma_i = 0). \end{cases}$$

This matrix will permit the least squares problem above to be solved by $\mathbf{x} = A^\dagger \mathbf{b}$.

The usual algorithm for computing the SVD consists of using elementary unitary transformations to first reduce A to bidiagonal form (only the diagonal and the super- [or sub-] diagonal have nonzero elements), and then performing some sort of iterative process (such as using the QR method) to force the off-diagonal elements to become numerically negligible. The first portion of this procedure is exact and so relevant in a symbolic context, particularly as the resultant matrix will be almost diagonal and hence easy to work with. As was the case in Section 2.2.3, the transformation to bidiagonal form can be performed using square root free Givens' rotations [Gen73], and so will be a purely rational process. An alternative procedure is to actually try to compute the eigenvalues of the smaller of AA^H and $A^H A$, and then take square roots. The major disadvantage to this method is that forming the matrix product will require $\min(m, n)^2 \max(m, n)$ multiplies and $\min(m, n)^2(\max(m, n) - 1)$ additions, which can be significant for large m and n.

To round out this survey, a matrix package should be able to compute a variety of special-purpose matrices. Some especially relevant to a CAS are the Jacobian or change of coordinate matrix with components $\frac{\partial f_i}{\partial x_j}$ for $\mathbf{f} = \mathbf{f}(\mathbf{x})$, the Hessian or second derivative matrix of $f = f(\mathbf{x})$ with elements $\frac{\partial^2 f}{\partial x_i \partial x_j}$, and the Wronskian matrix of $\mathbf{f}(x)$ specified componentwise by $f_j^{(i-1)}(x)$. The determinants of these matrices can provide valuable information, respectively, about the scaling of coordinate transforms, function maxima and minima, and the linear independence of solutions to n^{th} order, linear, homogeneous ODEs. Finally, it is very useful to be able to perform manipulations on whole matrices treated as symbols (e.g., A) as well as on their components. This amounts to directly managing the elements of a vector space (with respect to addition and scalar multiplication) if the matrices are taken to be general $m \times n$, and a ring with identity (with respect to addition and matrix multiplication) if they are assumed to be square. Of course, matrices of all different sizes (and this sometimes even includes infinite dimensional ones!) are frequently found together in one formula, so additional loose algebraic structures will be needed to cope with all the possibilities that may occur.

4.4 User Interface

A last but extremely important feature of a matrix package (or for that matter, of any piece of software) is its user interface. A noble goal is that interactions with the computer should be easy yet flexible, succinct yet descriptive, logical yet tolerant. This includes such apparent common sense (which is not always so apparent, it seems) as consistent syntax, helpful feedback and informative error messages.[12] There are many possible implementations (Table 4.6 shows how widely syntaxes can vary, for example), however, to be concrete, the following discussion will center around some of the ideas used (or at least considered) in the design of the Matrice Package.

One of the first decisions made was to always associate a matrix with a symbol (the matrice[13]), and to not display its contents unless the user specifically requested this action. The former makes it easy to associate mathematical, structural and computational properties with a given matrix by attaching them, and the matrix, to a unique symbol. The latter decision was motivated by the observation that displays of matrix contents are difficult to read if they exceed the size of a page (either electronic or paper), and this will be the typical situation except for an occasional simple result. Producing intelligible displays of large-scale structures, such as complex matrices, is a difficult problem (MACSYMA tends to do a little better job than other CASs but still not great), however, the user probably should have some sort of option to automatically see what is there if desired, with an ability to easily abort the output if it becomes too messy. A reasonable heuristic under such an option might be to display just an outline of the contents of complicated matrices unless the user really wants to see all the gory details. A handy function here would be something like MACSYMA's reveal command, which shows the anatomy of an expression down to a specified depth in its tree-structured general representation (e.g., $ax + b$ has a top-level description by reveal of `Sum(2)` and a secondary-level description of `Product(2) + b`).

One place where the user interface can be especially helpful is making it easy to enter matrices of various types. As discussed at the beginning of this chapter, MATLAB provides great flexibility in both creating general matrices and in being able to modify and extract pieces from them. The matrice function tries another tack. One argument, the name of the matrice, is required. All other information, if not explicitly or implicitly provided in the remaining arguments, will be interactively prompted for. This information will be a description of the matrix and its contents.

The matrice descriptors specify the type of the matrix (whether general, Hermitian, upper or lower triangular, etc.), its dimensions, whether the elements are to be considered as real or complex, and what form they may take (numbers, CREs, general expressions). Each descriptor has a definite format and so can be readily identified by the matrice function (ba-

[12] Two all time losers in this category are "Error while printing error message" generated by early versions of MACSYMA when a user attempted to load a file with a period in its name and forgot to enclose the name in quotes, and "Bad" which was the generic error message that TIPs (Terminal Interface Processors) on the old ARPANET used to produce when a user made most any sort of mistake.

[13] See the footnote near the end of the introductory material at the beginning of this chapter; also, refer to the first section of Appendix E.

	Comment	Line continuation	Displaying output	Suppressing output
MACSYMA	`/*` *comment* `*/`	*input*<cr>*more input*	*input*;	*input*$
MAPLE	`#` *comment*	*input*<cr>*more input*	*input*;	*input*:
Mathematica	`(*` *comment* `*)`	*input*<cr>*more input*	*input*	*input*;
MATLAB	`%` *comment*	*input* ..<cr>*more input*	*input*	*input*;
REDUCE	`%` *comment*	*input*<cr>*more input*	*input*;	*input*$
Scratchpad II	`--` *comment*	*input* -<cr>*more input*	*input*	*input*;

	Loading a file	Timing a command	Quitting
MACSYMA	`load("`*file*`")$`	`showtime: all$`	`quit();`
MAPLE	`read(‘`*file*`‘):`	`readlib(showtime): on;`	`quit`
Mathematica	`<<` *file*	`Timing[`*command*`]`	`Quit[]`
MATLAB	`exec('`*file*`');`	`t0 = clock;` *command*, `dt = etime(clock, t0)`	`exit`
REDUCE	`in "`*file*`"$`	`on time;`	`quit;`
Scratchpad II	`)read "`*file*`" )quiet`	`)set messages time long`	`)quit`

	Defining a matrix	Imaginary unit	Previous expression
MACSYMA	`m: matrix([1, 2],[3, 4]);`	`%i`	`%`
MAPLE	`m:= array([[1, 2],[3, 4]]);`	`I`	`"`
Mathematica	`m = {{1, 2},{3, 4}}`	`I`	`%`
MATLAB	`m = [1 2; 3 4]`	`sqrt(-1)`	`ans`
REDUCE	`m:= mat((1, 2),(3, 4));`	`i`	`ws`
Scratchpad II	`m:= [[1, 2],[3, 4]] :: M[2]`	`%i`	`%`

Table 4.6 Various synonyms for operations in five different CASs and one numerical matrix package. <cr> indicates a carriage return.

sically, by checking through a series of cases), therefore, any descriptors that are supplied can be given in an arbitrary order. A descriptor designating a banded matrix type is allowed to be furnished with an argument [e.g., GENERAL_BANDED(3)], in which case the value will be taken to be the half-bandwidth (which is needed for completing the structural specification of this type of matrix). To make life easier, certain common aliases are available. Thus, SYMMETRIC_TRIDIAGONAL is equivalent to REAL HERMITIAN_BANDED(2), for example. Reasonable defaults are provided whenever possible (matrices are assumed to be real and general with general expressions for elements unless matrice is told otherwise). This is a good idea normally anyway. All descriptors are checked for consistency, and a short but complete description of the newly created matrice (produced by matriceinfo) will be returned to the user as feedback. This and similar forms of feedback, by the way, are

common throughout the Matrice Package, and are an attempt to always indicate to the user what exactly has been changed (or created or deleted).

If the last argument supplied to matrice is an already existing MACSYMA matrix or two-dimensional array, then the elements (and the dimensions) of the matrice will be extracted from this object. The matrice can also be initialized from a list of lists[14] as well as to a given constant value. If the last argument is none of these forms, then entermatrice will be called to prompt for the non-redundant elements of the matrice one by one (for example, the upper triangle of a Hermitian or upper triangular matrix), although the user can provide a list if he wishes to enter a whole row at once. Thus, the elements of a matrice can be supplied in a variety of ways, allowing the user a great amount of flexibility.

Once a matrice has been created, it is easy to modify its characteristics (except for its dimensions) with the transmute function. This provides a simple mechanism for type conversion. Individual matrice elements can be examined and modified with gmat and pmat, respectively, although in a somewhat primitive fashion. A comprehensive solution might involve an extension of MATLAB's syntax to handle arbitrary diagonals, as well as permit block matrices to be easily constructed and manipulated.

The Matrice Package tries to provide a variety of utility and mathematical functions. Also, a number of options for controlling the progress of algorithms are available. Of special note is MRALTSIMP, which allows the user to supply an alternative to the default simplification that occurs during a computation. The user routine is permitted to act in different ways depending on the declared type of the matrice elements. Also of note is ZEROTEST, which grants the user the power to establish his own test for determining if an expression is truly zero. This may be useful in avoiding zero pivots (see Section 4.2.1). Not to be left out are MVERBOSE and MVERYVERBOSE, which if set will cause a (possibly detailed) trace of the progress of certain Matrice Package functions as they execute. Setting printlevel to be an integer > 1 in Appendix F will accomplish the same task for the functions there. In either case, such a trace will often provide positive feedback to the user who is wondering what the calculation is doing!

A global and a local aspect of the user interface will conclude this discussion. The hessenberg function is set up to produce the transformed matrix H given A (where $H = S^{-1}AS$), and optionally the similarity matrix S and possibly S^{-1}. Making the computation of one or both of the similarity matrices always an option is a desirable feature, and the algorithms presented up to now have generally provided either explicit details on how to calculate the similarity matrices or at least an indication of where to start. It would be very useful if this practice became as widespread in symbolic matrix packages as it has in numeric ones.

Finally, all arithmetic operations in the Matrice Package are funneled through a set of local (Lisp) routines. This permits complete control (in theory, at least) over simplifica-

[14]The top-level list will correspond to the matrice's primary array and the sublists to the secondary arrays in its data structure (for instance, the diagonals of a banded matrix—see Figure 4.1). Shorthands exist, so, for example, an input like `[2, 1]` to a symmetric banded matrice, will assign 2's to its diagonal and 1's to the super- (and hence, sub-) diagonal.

tion, the nemesis of CASs. Moreover, by simply setting a flag, it becomes trivial to count arithmetic operations as they are performed (see countops). Even more intriguing, it now becomes possible to completely replace normal arithmetic operations by an entirely different set of rules, such as those of expression swell arithmetic, for example. This is a really nice feature that is now coming into its own in object-oriented systems like Scratchpad II.

Chapter 5

Experiments

So far, the discussions have been for the most part mainly of a theoretical nature. It is now time to consider some actual symbolic experiments. In Appendix A, a random collection of test matrices is presented, ranging from fairly simple forms to some moderately complex examples. Some extremely nasty matrices [Cam82] were initially considered for testing purposes as well, but the set of matrices in Appendix A were more than sufficient to demonstrate that no automated method likely would have worked well, so the idea was dropped.[1]

Appendix B displays the timing results of a large number of determinant, characteristic polynomial and eigenvalue calculations performed under a variety of conditions by the major, general purpose CASs (MACSYMA, MAPLE, Mathematica, REDUCE and Scratchpad II—see Table B.1). Included in this study are results of standard, supplied functions (see Appendix D) and of the Matrice Package functions (see Appendix E). Some of the more interesting computations from this appendix have been collected together and reorganized in Appendix C. The times in each section of Appendix C have been normalized with respect to a specific calculation for easy comparison of results within a section. Moreover, the base calculation for each section has been chosen in a fairly consistent manner, so it is possible to make some more wide ranging comparisons as well. Certain of the computational series show a definite exponential (or greater) trend, and so in these cases, base 10 logarithms of the normalized timings are also included to make these trends more obvious.

The calculations in Appendix C are arranged by section in order of increasing complexity. For instance, the first two sections are concerned with diagonal matrices with symbolic entries. One would not expect that CASs would have much trouble dealing with such simple matrices, and the fact that some do reveals aspects of these systems' behavior that can cause significant problems in more complicated cases. Mathematica, for example, performs a recursive expansion by cofactors on the characteristic matrix in order to compute the characteristic polynomial. This polynomial is then factored to yield the eigenvalues. The whole process is incredibly slow, even for a very simple symbolic diagonal (see Section C.1).

[1] [Cam82] does provide some useful comments on making initially intractable symbolic problems tractable, the most generally relevant being to rename 'heavily used' subexpressions every so often in order to reduce the total space consumed by symbols (i.e., replace all occurrences of a common subexpression with a unique symbol).

This was due in part to an inefficient implementation of the Eigenvalues routine,[2] and in part to the insistence by the determinant function to fully expand its results, even when the algorithm used could have produced a factored form, thus making it necessary to factor a somewhat lengthy multivariate polynomial consisting of $N+1$ parameters given an $N \times N$ matrix.

The trait of producing a fully expanded characteristic polynomial is present to some extent in all of the CASs examined (MACSYMA's ncharpoly and charpoly with RATMX:true or the matrix elements initially in CRE form, MAPLE under most circumstances, REDUCE in its default mode, and Scratchpad II). Therefore, many of the computations in Section C.2 for a 100×100 Sulsky1 Diagonal matrix were either very time consuming or failed completely (entries preceded by a $>$ indicate calculations that were intentionally aborted because they were taking too long, while those superscripted with letters reference computations that died a violent death, as evidenced by the tables of fatal error messages later on to which the letters refer).

Retaining any factors that occurred naturally as the calculations progressed (with the Matrice Package functions or RATFAC:true in MACSYMA, and with the sparse MAPLE functions) or performing continual factorizations (by setting factor on in REDUCE) sped things up considerably, although in some instances there was still a substantial overhead associated with operating on a 100×100 matrix. Using sparse methods, especially in conjunction with factor retention, was of even greater benefit since these procedures were able to take maximum advantage of the zero structure of the diagonal matrices. Sparse methods in this situation involved various sorts of minor expansion (invoked explicitly by SPARSE:true in MACSYMA, det(,*sparse*) in MAPLE, and mateigen in REDUCE) and recognition of block triangular matrices (as done by the Matrice Package functions) which could then be treated blockwise.[3]

Some procedures failed because they were just too space or time consumptive. MACSYMA's charpoly with no special options set will use a bottom-up minor expansion algorithm when computing the determinant of $A - \lambda I$. This is fine in applications like the current one involving sparse matrices. At the same time, however, only the most trivial of simplifications are performed as the expression is computed. Even for purely numerical matrices, these default simplifications are not sufficient to collect together the coefficients corresponding to a given power of λ. To do so, aid must be provided in the form of a subsequent invocation of a

[2]It was not uncommon that the time required to explicitly calculate the characteristic polynomial and factor it would be half as much as that taken by Eigenvalues.

[3]For a diagonal matrix, these two schemes are effectively equivalent, however, for a matrix with as simple a structure as

$$\begin{pmatrix} \times & \times & 0 & 0 \\ \times & \times & 0 & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{pmatrix},$$

the latter procedure will guarantee a partial factorization of the determinant/characteristic polynomial while the former cannot. None of the standard functions supplied with the CASs tested could automatically produce such a factorization for a generic example of the above matrix, however, the Matrice Package functions readily provided one.

command like ratsimp which performs rational simplification. The Sulsky1 Diagonal matrices involve a parameter, making the situation even worse, and hence the results of charpoly for large N can be extremely long and complicated. In the 100×100 run, this was apparently so much the case that the calculation ran out of memory.

As a second example, ncharpoly, which uses the method of Leverrier to compute the characteristic polynomial (see Section 2.2.1), also failed at its task with the 100×100 Sulsky1 Diagonal matrix. The algorithm is slow [$O(n^4)$—see Table 2.4] as can be seen even from the 10×10 case. Moreover, no special simplifications on any of the expressions formed are done until the very end when there is a call to ratsimp. Thus, adding in even a single symbolic parameter, as was the case here, can cause huge, unmanageable expressions to be thrown to ratsimp, leading to complete, catastrophic collapse of the procedure for not all that large of an N.

Some final demonstrations involved computing the characteristic matrix by hand (these are the "loop" times in which a loop is used to subtract λ from each of the diagonal elements of the matrix), and then invoking a couple of additional determinant implementations. One was det, which suffered both from being an algorithm better suited to dense matrices (two-step Bareiss elimination), and also from being originally written in the top-level MACSYMA language rather than Lisp. Even after being translated into Lisp and compiled by MACSYMA, there is significant overhead associated with calls to MACSYMA functions in such code. This overhead is mainly a result of numerous and often largely redundant calls to the general simplifier acting on the arguments and return values of each function invoked. The general simplifier will be applied to nearly every operation of any sort unless specifically inhibited, either explicitly (by setting SIMP:false, which turns off *all* simplifications) or implicitly. As an example of the latter, a variable (such as a loop index) can be declared to be an integer. Arithmetic operations on it (after compilation) will then be released from many of the extra precautions that are taken with a generic variable, which has the possibility of possessing a symbolic value and so must be treated more carefully. More on the topic of high level versus low level coding will be presented when discussing the Fuka matrices.

The coding of det makes no effort to avoid unnecessary references to MACSYMA's general simplifier. Moreover, the total number of binary arithmetic operations performed by a two-step Bareiss elimination method is very nearly N^3 (see Table 2.3c), which is 10^6 for $N = 100$. Therefore, even if each operation takes only a few milliseconds (a typical value), the large number of operations involved (also including many index calculations and list manipulations) will result in substantial CPU times for sizable values of N.

newdet, on the other hand, tries to perform a bottom-up minor expansion using CREs. This failed miserably for the Sulsky1 Diagonal matrices as the intermediate results quickly got out of hand, even for the relatively low value of $N = 20$. For a slightly larger value of N ($N = 25$), the calculation failed almost immediately when newdet was unable to allocate enough space for its internal arrays (the largest, which is of a size that depends exponentially on N, requested a capacity of $> 10^8$ elements for this value of N!).

The next series of calculations involved the Wilkinson matrices, which have integer entries and are tridiagonal (see Section C.3). Mathematica was again unable to complete a $100 \times$

100 problem in a reasonable amount of time and so the computation was aborted. This was also true for the MACSYMA functions ncharpoly and det. MAPLE and REDUCE were able to produce an answer, however, and at fairly equivalent speeds. To gain some feeling of how complicated the results produced were, these systems were then asked to factor the polynomials that they had just computed. Although the times required certainly depended on the details of the particular factoring implementations, an order of magnitude difference, as was the case here, strongly implies that the MAPLE results were quite a bit more complicated than were the REDUCE ones.

In MACSYMA, the complexity of the characteristic polynomials generated was examined in several ways. length was a quick and dirty method for counting the number of terms (or possibly factors) in an expression. complexity measured an expression's complexity as defined in Table 4.5 (for general expressions, this is the number of operators and atomic operands). meigenvalues (using factor) would factor the polynomial and solve for the roots of any linear factors it found.

As was the case previously, MACSYMA's sparse algorithm ran well. Setting RATFAC:true slowed things down a little, but produced shorter expression lengths. With the Matrice Package's mcharpoly, three sets of trials were performed in which different amounts of knowledge about the Wilkinson matrix's structure were provided to the function. Matrices initially indicated to be tridiagonal or symmetric were quickly sent to the $O(n^2)$ tridiagonal characteristic polynomial algorithm (Figure 2.13), resulting in relatively quick runtimes. (In the modified fast Givens' reduction, if an element to be eliminated is already zero, then the reduction will proceed immediately to the next elimination step; hence, a tridiagonal matrix, which is already in final form, will be processed rapidly.)

Matrices initially specified only as general were sent to be converted to upper Hessenberg form by the algorithm given in Figure 2.5, and then run through the $O(n^3)$ upper Hessenberg characteristic polynomial algorithm (Figure 2.7). Even though most of the entries in the matrices were zero, it was not as easy as it was with the Givens' reduction to completely bypass all the calculations in unneeded elimination steps since entire columns are processed all at once. In particular, the second loop on i in Figure 2.5 contributes $\frac{1}{2}n^3-\frac{3}{2}n^2+n$ additions and multiplications, or slightly more than half of the total counts of these operations given in Table 2.5a. Therefore, computing the characteristic polynomial of a Wilkinson matrix via an intermediate Hessenberg form required an aggregate sum of approximately $\frac{3}{2}N^3$ binary operations (albeit mostly ones involving zeroes).

The next computational series again dealt with tridiagonal matrices, but now with symbolic diagonal elements (the Sulsky1 matrices of Section C.4). The only calculations which really had much success for large N were those that made use of MACSYMA's CRE representation for polynomials. charpoly did best when the CREs were maintained in a partially factored form (RATFAC:true) as this substantially reduced the size of the expressions involved. The sparse algorithm again ran faster. mcharpoly did not do so well this time around, generating huge expressions. Starting with matrix entries that were CREs, however (see Table C.17), sped the computations up considerably by substantially reducing the size of the expressions being manipulated. The performance of mcharpoly in this circumstance

improved on charpoly using CREs and one-step Bareiss elimination (i.e., RATMX:true only), but lagged behind the partially factored calculations.

The results of computing the characteristic polynomial for a somewhat more structurally complicated matrix than any so far discussed are found in Section C.5. The 11×11 Schwarz matrix is pentadiagonal and has all integer entries. Complexity here is measured by the number of lines of output produced when the polynomial was displayed. The only extremely excessive number was associated with MACSYMA's bottom-up minor expansion (plain charpoly). One Gaussian elimination and two expansion by minors implementations (charpoly with RATMX:true, and procedures from Mathematica and REDUCE, respectively) ran especially fast on this problem, but most other methods did not fall too far behind. Some items to note were that MACSYMA's sparse algorithm was not as fast as Gaussian elimination, retaining factors (setting RATFAC:true) was a detriment in the CRE calculations, introducing CREs slowed down the Matrice Package functions, and the fast Givens' reduction was particularly slow.

The Sulsky4 matrices are the next to be studied (Section C.6). These matrices are block tridiagonal with diagonal blocks that are themselves tridiagonal—all other blocks being of the form of identity matrices. The calculations were all done under MACSYMA as it was especially convenient to create various sizes of Sulsky4 matrices using the symmetric banded entry option of the matrice function.

Once again, ncharpoly, newdet and det ran quite slowly, with the order listed being from fastest to slowest. Even though the Sulsky4 matrices are very sparse, charpoly with SPARSE:true turned out to be even more sluggish than these three procedures in the 16×16 case (and the calculations died a violent death in the 25×25 examples). In fact, the only method slower was regular charpoly, which was taking close to three hours to do the 16×16 calculation. Success in terms of reasonable computation times came with the one-step Bareiss elimination method (RATMX:true). Setting RATFAC:true as well improved the response even more.

By making use of the fast Givens' reduction, mcharpoly was able to beat the best times above up through the 25×25 Sulsky4 matrix. However, starting with the 36×36 case, the times became slower and lagged further and further behind with increasing N. Introducing CREs was unhelpful, and performing occasional or frequent searches for optimal pivots (MINHERMPIVOT:true or always, respectively) was counterproductive except for the smallest matrices. Unlike the situation for the Hessenberg algorithm, a similarity permutation applied to a Hermitian matrix requires explicit swapping of elements. The overhead associated with this process apparently became quite substantial for matrices that were not all that large ($N \geq 25$).

The best method for computing the characteristic polynomials of the Sulsky4 matrices with dimensions of 81×81 or less was the mcharpoly procedure of converting to an intermediate upper Hessenberg form. The Bareiss elimination algorithm finally won out in the 100×100 case. It is interesting to note that the times required to do the matrix transformations followed a definite exponential trend as a function of $\sqrt{N}$ for $N \geq 36$ (see Table C.26); most other sequences in this section exhibited similar behaviors but were not so clear cut.

The introduction of CREs was again unhelpful. The use of a division-free algorithm also did not improve the running time, and eventually led to disaster when the quantities generated became too huge to handle. If GCDs were suppressed altogether with the division-free procedure, disaster came very quickly.

The next two matrices tabulated in Appendix C (Wester in Section C.7 and JR in Section C.8) are both numerical and dense. Many similarities exist between the two sets of calculations, and so they will be discussed together. To begin, the lengthiest computations involving either matrix were those performed by Mathematica, Scratchpad II and MACSYMA's charpoly, in either standard mode, with RATFAC:true or with SPARSE:true. This last is not surprising since these matrices are far from sparse, and it would be unexpected if a sparse algorithm did work well.

The best (or close to the best) times were also due to charpoly, but with RATMX:true only. It is interesting to note that as a general trend, the same calculations run under Common Lisp MACSYMA 415.69 were always somewhat slower than those run under Franz Lisp MACSYMA 309. This may be due to the fact that the compiled binary of MACSYMA 415.69 is more than twice as large as that for MACSYMA 309 (13 Mb versus 5 to 6 Mb). Since the machines used had limited memory, the bigger MACSYMA likely induced a greater amount of swapping and garbage collection, and so resulted in elevating the total time consumed.

The hybrid routine of MAPLE also ran well. The Hessenberg method, on the other hand, was variable. For the Wester matrix, the intermediate upper Hessenberg form had mainly rational entries, the most complicated containing a sum of 190 digits in its numerator and denominator. The Hessenberg form corresponding to the JR matrix was much nicer, however, with the top two rows containing integers of less than 6 digits and all other rows were zero except for -1's along the subdiagonal. In fact, this form was so nice that the associated characteristic polynomial computation was the fastest of all the ones performed for this matrix. In contrast, letting mcharpoly treat the JR matrix as Hermitian resulted in an intermediate tridiagonal form that had primarily rational entries—the most complicated possessing an 11-digit numerator and a similarly sized denominator. In this case, the CPU time was much greater.

Next in the line-up is the SU3 matrix (Section C.9). This 8×8 matrix has 8 symbolic parameters and 41% of its elements are zero. It is quite representative of the types of matrices that are encountered by physicists. Even though this matrix is not all that large and is somewhat sparse, its characteristic polynomial is fairly complicated (just trying to factor it often was overwhelming). REDUCE's cofactor expansion algorithm was the winner here, however, similar procedures from MACSYMA, MAPLE and Mathematica did not trail far behind. (MAPLE's det actually performed a hybrid calculation, doing four stages of one-step Bareiss elimination and then resorting to minor expansion for the other half of the computation when no sufficiently simple pivot was detected.) Full one-step Bareiss elimination (charpoly with RATMX:true) also produced a result in an acceptable amount of time. In addition, charpoly with no options set and the combination of hessenberg plus mcharpoly were both able to compute an answer (the former as fast as REDUCE), however, the results were monster polynomials (so big in the second case that tremendous amounts of

time were needed just to format the output—the calculation was aborted before this could complete).

The last series of calculations to be considered are determinant computations for the Fuka matrices (Section C.10). These matrices are tridiagonal off shifted by one diagonal, with an extra triangle of nonzero elements in the lower left-hand corners. The Fuka matrices come in both general and "symmetric" forms. The $2n \times 2n$ general form possesses $3n + 2$ distinct variables, while for the symmetric forms, the 6×6 matrix has 6 and the 34×34 matrix has 27 independent parameters.

Table C.32 shows the results of calculations done with the MAPLE det function and the three varieties of the Det function listed in Appendix F in which partial pivoting was turned off.[4] The two one-step Bareiss implementations ran in roughly the same amount of time with about the same memory usage. (Note that the length of an expression in MAPLE is a measure of its complexity, and will be equal for equivalent calculations.) The situation improved somewhat with the two-step Bareiss procedure, and quite dramatically with the one-step sparse algorithm in which a step was completely skipped if the element to be eliminated was already zero. None of the calculations succeeded for the 34×34 symmetric matrix, as can be seen in Table C.33; however, the sparse method did at least get further along in the computations than did the Bareiss runs (failing at last when trying to perform the final exact divisions at the end of the algorithm). The difference between the "time" and the "accumulated time" was what was spent trying to process the "max stage"+1 of the elimination (or the final divisions) before the calculation crashed.

The next pair of tables display the outcomes of applying MACSYMA's determinant function to the smaller Fuka matrices. The sparse procedure ran somewhat quicker than did the Bareiss method on the larger of these matrices, while factor retention sometimes helped and sometimes hurt. In general, the MACSYMA calculations ran considerably faster than any of the MAPLE ones, with the notable exception of bottom-up minor expansion, which performed extremely sluggishly. As an aside, ratsimp, MACSYMA's general purpose simplifier, seemed to have great difficulties dealing with the partially factored CRE forms.

Further determinant computations, this time by the MACSYMA versions of the routines in Appendix F, are given in Table C.36. Partial pivoting was again turned off. As was the case with MAPLE, the speed and the simplicity of the results was the most favorable for the sparse method and the least favorable when the one-step Bareiss algorithm was employed. The three methods all ran relatively quickly when the elements of the matrices were given as general expressions. The final determinants, however, were somewhat complicated as evidenced by the comparatively long running times of the ratsimp operations, as well as by the noticeable differences in the before and after values of the complexity.

If the matrix elements were first made into CREs, the three Det procedures produced reasonably uncomplicated results, however, the algorithms then ran rather slowly. This negative outcome is largely an artifact of Det being written in the top-level MACSYMA language; by applying a few simple techniques, it is quite possible to eliminate much of

[4] `better_pivot:= proc(x, y) false end:`

the associated overhead. To demonstrate this idea, the Det code was condensed and then modified in various ways. Table 5.1 shows what happened in the 10×10 case.

First, just eliminating references to linear equation solving and making Det a single consolidated function sped up the three base calculations by 16–22%. Next, replacing all references to polynomial arithmetic (e.g., $ax + b$) by calls to user-defined functions [e.g., add(mul(a, x), b)], where these functions were defined at top-level in terms of primitive CRE arithmetic operations,[5] caused the Bareiss procedures to run 20% faster than in the base calculations (listed immediately above). The sparse computation did slow down by 5%; however, after rewriting the user arithmetic functions directly in Lisp, there was a substantial gain in speed by all three algorithms. The situation was further improved by temporarily setting SIMP:false during any explicit polynomial arithmetic operation as this avoided unnecessary calls to simplifya, MACSYMA's general simplifier, since the CRE package handles its own simplifications. These are the strategies, by the way, used by determinant when RATMX:true, although these techniques were easier to implement there since the function was written directly in Lisp. It seems likely that additional optimizations (such as rewriting Det entirely in Lisp) would eventually produce comparable behavior to the built-in MACSYMA procedures.

The final set of calculations involving the Fuka matrices was attempting to compute the determinant of the 34×34 symmetric matrix (see Table C.37). Unlike the MAPLE results, three calculations actually completed here. Their success was almost certainly due to the fact that essentially no intermediate simplifications were performed by MACSYMA so that, for instance, the exact quotient of a pair of complicated polynomials was not further processed, the result being left in the form of a rational function. This presumption is supported by the huge complexitys [$O(10^9)$] of the determinants and the very long run times made by ratsimp (> 1 CPU day per run!) in vain attempts to simplify the results. All other calculations invoked CRE forms in some fashion with a total lack of success.

5.1 Summary

Now that the calculations presented in Appendix C have been discussed in detail, some conclusions can be drawn from these results. In particular, it is useful to summarize which methods worked best in a given situation, as well as identify those methods that were pretty much failures except when dealing with the smallest and simplest of matrices. Consider some of the latter procedures first.

The method of Leverrier (as manifested in ncharpoly) was the only $O(n^4)$ procedure that was tested and not surprisingly, it ran slowly or not at all. The various eigenvalue routines were also not all that successful in the runs studied, except for REDUCE's mateigen, and a few did not even return some sort of residual polynomial when they could not solve for all the roots of the characteristic polynomial (the offenders here were MACSYMA's eigenvalues,

[5]The syntax `?pplus(x, y)` in MACSYMA is used to refer to Lisp-level functions not normally directly accessible by the user.

METHOD	times		ratio
rat	0.316		0.025
Bareiss1	73.213	[37.017]	5.893
Bareiss1 (M)	58.206	[33.036]	4.685
Bareiss1 (L)	39.360	[20.346]	3.168
Bareiss1 (MS)	35.856	[17.503]	2.886
Bareiss1 (LS)	29.950	[13.973]	2.411
Bareiss2	73.066	[37.880]	5.882
Bareiss2 (M)	58.076	[32.823]	4.675
Bareiss2 (L)	34.966	[18.553]	2.815
Bareiss2 (MS)	35.026	[17.230]	2.819
Bareiss2 (LS)	26.483	[13.306]	2.132
Sparse	50.613	[24.346]	4.074
Sparse (M)	53.093	[30.956]	4.274
Sparse (L)	41.630	[23.263]	3.351
Sparse (MS)	38.366	[21.060]	3.088
Sparse (LS)	35.560	[19.436]	2.862
determinant (RATMX)	12.423	[5.823]	1.000

Key: Polynomial arithmetic was replaced by calls like
M = add(x, y) where `ADD(x, y):= ?pplus(x, y)`, etc.
L = add(x, y) where `(defun $ADD (x y) (pplus x y))`, etc.
S = SIMP:false, add(x, y), SIMP:true; etc.

Table 5.1 The overall number of seconds [and GC times], averaged over 5 trials, that were taken to compute the determinant of the 10×10 general Fuka matrix [gen(10)] using the indicated methods after first converting the matrix entries into CREs with rat. These calculations were performed on a Sun 3/160 using specially tweaked versions (as described by the key) of the MACSYMA routines equivalent to those found in Appendix F. In the last column, the overall times are normalized with respect to that taken by determinant with RATMX:true for purposes of comparison.

Mathematica's Eigenvalues and Scratchpad II's eigenvalues). A major part of the problem with the eigenvalue routines was that any natural factorization of the characteristic polynomials that they computed tended to be lost, since the underlying polynomial arithmetic usually did not retain factors. This was demonstrated dramatically for both characteristic polynomial and eigenvalue functions by the poor showing of so many routines when given a large symbolic diagonal matrix, which ironically is easy for any (mathematically inclined) human to deal with! It was not uncommon for a function to compute the completely factored characteristic polynomial and then expand the entire result out. The expansion might have occurred as each step of the procedure was accomplished rather than all at once at the end, but in any case, a great deal of unnecessary (and counterproductive) work was performed. Not all expansion is bad, by the way, as a number of calculations were helped by doing all internal computations in a fully expanded canonical form (e.g., CRE form). However, for block triangular matrices, factors corresponding to distinct diagonal blocks should always be retained by determinant and characteristic polynomial algorithms.

MACSYMA's bottom-up minor expansion, as implemented in the default method of determinant and by newdet (in which matrix elements are first converted into CREs), did not fare well with the characteristic matrices that were tested, a not too surprising conclusion considering that this algorithm has a dense operation count of $O(n2^{n-1})$. A similar story was true for det, which is a top-level and unsophisticated implementation of the 2-step Bareiss algorithm (unsophisticated in that no attempt is made to minimize pivots and all swapping occurs explicitly, rather than implicitly with the aid of an indexing array). Other procedures that had difficulties were variations on the more successful methods to be discussed next, and so will be referenced as appropriate in the following contexts.

Table 5.2 summarizes the characteristic polynomial calculations in Appendix C, presenting the methods that were most successful when applied to a selected set of matrices. Brief descriptions of the matrices and the methods employed by the various routines are given in Tables 5.3 and 5.4, respectively. In Table 5.2, dashes indicate runs that were not performed. 'Failed' computations were those that either spontaneously aborted or were stopped when the time they were taking was deemed excessive. Some calculations were not run, but presumably would have failed if they had been, because like ones involving smaller or simpler matrices of the same type had been unsuccessful. In one case, the computation might not have actually failed but certainly would have been very slow; this entry is indicated by a tilde ($\sim$).

The times in Table 5.2, as in Appendix C, have been normalized with respect to the 10×10 (or size closest to 10×10) MACSYMA charpoly calculation for each computational series with RATMX:true (i.e., one-step Bareiss elimination operating on CREs). This provides a global basis for cost comparison but is somewhat difficult to interpret, so a second version of this table with the times in each column normalized to the fastest calculation therein was also produced and is presented as Table 5.5. Under this normalization strategy, numbers close to one indicate relatively quick results, while large values are a sign of computations that are slow relative to the base times. Determinant calculations for the 16×16 general Fuka matrix were treated similarly, and the results are listed separately in Table 5.6.

METHOD	Sulsky1 Diagonal ($N = 100$)	Wilkinson (100)	Sulsky1 (100)	Schwarz (11)	Sulsky4 (49)	Sulsky4 (100)	Wester (10)	JR (12)	SU3 (8)
MACSYMA									
charpoly(R)	♠	1819.592	♠	1.006	92.698	1446.977	1.048	1.000	1.066
charpoly(FR)	56.735	♠	505.696	3.150	62.868	912.394	54.442	♠	2.252
charpoly(RS)	31.786	191.872	♠	3.072	♣	♣	47.034	493.874	0.312
charpoly(FRS)	8.005	194.864	105.007	7.963	♣	♣	72.703	544.060	0.546
mcharpoly(D/T)	0.100	69.675	♠	—	—	—	—	—	—
mcharpoly(H)	0.177	68.684	♣	26.284	266.387	♣	—	20.326	—
mcharpoly(G)	4.362	805.782	♣	3.401	46.754	1242.474	21.305	0.776	♠
mcharpoly(G:C)	—	—	♣	6.117	117.746	2099.605	24.035	4.488	♠
MAPLE									
charpoly	5.138	1155.698	♠	2.510	—	—	1.829	2.520	0.328
Mathematica									
charpoly	34.680	♠	♣	0.680	—	—	29.581	♠	0.314
REDUCE									
charpoly	♠	1051.908	♠	0.462	—	—	5.889	7.673	0.106
mateigen	0.129	∼	♠	6.104	—	—	16.608	2.699	♠
Scratchpad II									
characteristicpol	—	—	♠	4.878	—	—	13.208	27.112	7.238
1 time unit =	48.391	1.983	6.800	3.016	5.067	5.067	3.816	4.750	74.100

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
D = diagonal, T = tridiagonal, H = Hermitian, G = general, C = CRE
♠ = calculation failed, ♣ = not run but would have failed

Table 5.2 A summary of the most significant calculations and successful methods of Appendix C for determining the characteristic polynomial. The times are normalized as in the appendix. Boxed entries designate the fastest computation in a given series.

matrix	N	elements	symmetric	structure
Sulsky1 Diagonal	100	1 parameter	$\times$	diagonal
Wilkinson	100	integer	$\times$	tridiagonal
Sulsky1	100	1 parameter	$\times$	tridiagonal
Schwarz	11	integer	$\times$	pentadiagonal
Sulsky4	49	integer	$\times$	simple block tridiagonal
Sulsky4	100	integer	$\times$	simple block tridiagonal
Wester	10	integer		dense
JR	12	integer	$\times$	dense
SU3	8	8 parameters		semi-sparse
Fuka gen(16)	16	26 parameters		'tridiagonal' + a "corner"

Table 5.3 Dimension, type of elements and physical structure of the matrices specified in Table 5.2.

Routine	Method
MACSYMA	
charpoly(R)	one-step Bareiss elimination
charpoly(FR)	one-step Bareiss elimination (RATFAC)
charpoly(RS)	conservative minor expansion
charpoly(FRS)	conservative minor expansion (RATFAC)
mcharpoly(T)	tridiagonal algorithm
mcharpoly(H)	fast Givens' method
mcharpoly(G)	Hessenberg method
mcharpoly(G:C)	Hessenberg method (invoking CREs)
MAPLE	
charpoly	one-step Bareiss + cofactor expansion
Mathematica	
charpoly	cofactor expansion
REDUCE	
charpoly	cofactor expansion
mateigen	cofactor expansion
Scratchpad II	
characteristicpol	one-step Bareiss elimination

Table 5.4 Methods used to compute the characteristic polynomial/determinant of the characteristic matrix for the matrices specified in Table 5.2 by the routines listed there. See also Appendix D.

METHOD	Sulsky1 Diagonal ($N = 100$)	Wilkinson (100)	Sulsky1 (100)	Schwarz (11)	Sulsky4 (49)	Sulsky4 (100)	Wester (10)	JR (12)	SU3 (8)
MACSYMA									
charpoly(R)	♠	26.492	♠	2.177	1.983	1.586	1.000	1.289	10.057
charpoly(FR)	567.350	♠	4.816	6.818	1.345	1.000	51.948	♠	21.245
charpoly(RS)	317.860	2.793	♠	6.649	♣	♣	44.880	636.436	2.943
charpoly(FRS)	80.050	2.837	1.000	17.236	♣	♣	69.373	701.108	5.151
mcharpoly(D/T)	1.000	1.014	♠	—	—	—	—	—	—
mcharpoly(H)	1.770	1.000	♣	56.892	5.698	♣	—	26.193	—
mcharpoly(G)	43.620	11.732	♣	7.361	1.000	1.362	20.329	1.000	♠
mcharpoly(G:C)	—	—	♣	13.240	2.518	2.301	22.934	5.784	♠
MAPLE									
charpoly	51.380	16.826	♠	5.433	—	—	1.745	3.247	3.094
Mathematica									
charpoly	346.800	♠	♣	1.472	—	—	28.226	♠	2.962
REDUCE									
charpoly	♠	15.315	♠	1.000	—	—	5.619	9.888	1.000
mateigen	1.290	∼	♠	13.212	—	—	15.847	3.478	♠
Scratchpad II									
characteristicpol	—	—	♠	10.558	—	—	12.603	34.938	68.283
1 time unit =	4.839	136.200	714.048	1.393	236.903	4623.100	3.999	3.686	7.855

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
D = diagonal, T = tridiagonal, H = Hermitian, G = general, C = CRE
♠ = calculation failed, ♣ = not run but would have failed

Table 5.5 A summary of the most significant calculations and successful methods of Appendix C for determining the characteristic polynomial, in which the times in each computational series have been normalized with respect to the fastest (boxed) run in that series.

METHOD	times	complexity/ length	ratio
MACSYMA			
determinant	2778.912	20873	1375.018
determinant(R)	10.451	8355	5.171
determinant(FR)	10.116	8355	5.005
determinant(RS)	8.989	7370	4.448
determinant(FRS)	9.084	7370	4.495
Det(1)	4.921	31017	2.435
Det(2)	4.718	27684	2.334
Det(s)	2.021	24692	1.000
Det(1r)	233.188	10411	115.382
Det(2r)	236.782	10411	117.161
Det(sr)	210.847	10411	104.328
MAPLE			
det	122.411	38630	60.570
Det(1)	126.865	38630	62.773
Det(2)	114.717	38630	56.762
Det(s)	38.455	38630	19.028
1 time unit =	12.933		26.137

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69), r = RAT
1 = METHOD:BAREISS1, 2 = METHOD:BAREISS2, s = METHOD:SPARSE
1 = METHOD:=bareiss1, 2 = METHOD:=bareiss2, s = METHOD:=sparse

Table 5.6 A summary of the results of Section C.10 for computing the determinant of the 16×16 general Fuka matrix [gen(16)]. The times in the second column are normalized as in Appendix C, while those in the last column are the ratios with respect to the fastest (boxed) calculation for this matrix.

matrix	N	$\max(\mathcal{N}_{10}(A))$	$\max(\mathcal{N}_{10}(H))$	$\max(\mathcal{N}_{10}(T))$	$\frac{\max(\mathcal{N}_{10}(T))}{\max(\mathcal{N}_{10}(H))}$
Rosser	8	3	39	140	3.590
Hankel	9	1	3	14	4.667
Schwarz	11	1	48	186	3.875
JR	12	2	5	22	4.400

Table 5.7 The maximum number of digits taken by the entries of the $N \times N$ symmetric matrices A, their upper Hessenberg forms H and their tridiagonal forms T, the latter two matrices being derived from the A's through similarity reductions. See Section B.6.13 for additional data.

Looking at the data, some definite trends can be picked out; however, it is also true that the results in many cases are somewhat muddled. As noted previously, methods that employed automatic expansion of polynomials fared poorly for the most part when handling the large diagonal and tridiagonal matrices. For both types of Sulsky1 matrices (however, not for the Wilkinson matrix), retaining polynomial factors was of definite benefit in reducing the computational overhead, and in the case of the Sulsky1 tridiagonal matrix, making the calculations even possible. Factor retention was of mixed utility for the other sparse matrices though, and a major impediment in the dense computations. Its effect was at least consistently helpful or not helpful for a given series of calculations, the Fuka matrices being the sole exception.

The speediest calculations of the characteristic polynomials for the Sulsky1 Diagonal and the tridiagonal Wilkinson matrices were due to mcharpoly when told that its matrix argument was symmetric and perhaps diagonal/tridiagonal. Elimination was either trivial or could be avoided altogether, and so for the Wilkinson matrix, the fast $O(n^2)$ tridiagonal characteristic polynomial algorithm was almost immediately invoked. This procedure was less fortunate with the Sulsky1 matrices, however, even with the speed up gained by using CRE forms (see Tables C.16 and C.17), and was completely unsuccessful in the 100×100 trial. It is quite possible that the overhead of passing all expressions through simplifya, as in the case of the Fuka matrices, was a major contributing factor to these negative results, as the algorithm itself should have been very similar to the one used by the first successful instance of charpoly in this computational series.

The actual fast Givens' procedure, when applied to matrices less simple than tridiagonal, was not particularly fast and was always slower than the more general Hessenberg method, for example. This latter observation is not too surprising once the two different intermediate forms are examined. For instance, Table 5.7 presents the comparative sizes of the largest entries found in the similar Hessenberg and tridiagonal forms of four of the symmetric matrices analyzed in Appendix B. In each case, the largest tridiagonal matrix entry is around four times the size of the largest Hessenberg one. The cost of creating and manipulating these more complicated values will substantially outweigh the cost associated with the Hessen-

berg matrix entries, even though the number of nonzero elements in the tridiagonal matrix is considerably less than for the Hessenberg matrix ($3n-2$ versus $\frac{1}{2}n^2+\frac{3}{2}n-1$). Trying to minimize the pivots in the Givens' reduction (by setting MINHERMPIVOT:true) did not help as too much time was spent searching for better pivots and then physically swapping elements if one was found. Doing this procedure frequently (MINHERMPIVOT:all) was an even worse idea.

The conservative minor expansion algorithm of MACSYMA (invoked when SPARSE:true) also did well with the tridiagonal matrices, and performed quite decently for the semi-sparse SU3 matrix and the Fuka matrices. Other sparse matrices, such as the Schwarz and Sulsky4 matrices, were apparently not sparse enough as this method bogged down on them. Not surprisingly, conservative minor expansion ran very slowly on the dense numerical matrices (Wester and JR).

The other minor expansion implementations were quite variable in their abilities. Mathematica's charpoly and REDUCE's mateigen were generally slow routines overall, although Mathematica did reasonably well with the smaller sparse matrices, while mateigen cleverly handled the diagonal case and performed decently on the JR matrix. Mathematica seemed to have major difficulties dealing with large matrices, and the minor expansion did not work well on dense matrices either. REDUCE's charpoly, on the other hand, excelled with the smaller sparse matrices, had acceptable performance (a little slow) on the dense matrices, and at least managed an answer for the 100×100 tridiagonal Wilkinson matrix. The differences in performance among the five minor expansion procedures discussed here (if one includes regular charpoly) show how subtle algorithmic variations and the quality of the underlying polynomial arithmetic can have profound effects on the results produced for a given problem.

The Hessenberg method had notable success with the JR matrix (which had a particularly simple upper Hessenberg similarity form) and the Sulsky4 matrices (where it was easy to eliminate the lone diagonal of ones in the lower triangle). Otherwise, though, this method lacked versatility and yielded only mediocre to poor performance for the rest of the calculations tabulated in the summary. (However, it should be pointed out that more information was also produced than in most of the other methods examined, i.e., the Hessenberg form similar to the original matrix.) Although employing CREs was of significant aid in the Sulsky1 calculations, in general, their use was counterproductive in the Hessenberg runs. A similar statement applies to the fast Givens' method, by the way. Running the division-free version of the Hessenberg algorithm (HDIVFREE:true) also was not a help, quickly generating huge expressions (which became even worse when GCDs were suppressed with HDIVFREE:nogcds).

The Hessenberg method was the leading procedure for computing the Sulsky4 characteristic polynomials up until $N = 100$ when the one-step Bareiss implementation of charpoly with RATMX:true came out ahead. The fraction-free Gaussian elimination strategy was on the whole the most universal of the pure methods that were engaged for solving a random problem in a reasonable amount of time (or at all). The worst cases occurred when certain types of sparse matrices (i.e., diagonal, tridiagonal, SU3) were involved. For dense matrices,

in particular, Bareiss elimination was a clear winner over expansion by minors as should be expected from the $O(n^3)$ versus $O(n!)$ operation counts in this situation. MACSYMA's version of the one-step Bareiss elimination procedure ran the fastest, as characteristicpol seemed to be considerably weighed down by overhead from the massive Scratchpad II system.

The problem that the Bareiss methods have with very sparse matrices is that in order for the procedures to remain fraction-free, *every* row below a given diagonal pivot must be operated upon, even when the element to be eliminated in that row is already zero. For the diagonal example, this overhead was compounded by the common practice of expanding all polynomials out, resulting in a great deal of arithmetic being performed. Applying a sparse fraction-free Gaussian elimination algorithm that does not operate on a row unless it needs to, to the very sparse but highly symbolic Fuka matrices proved to be of definite benefit. (In contrast, using a two-step Bareiss procedure was only of minor advantage, if at all, for a considerably more complicated algorithm.) This statement is tempered by noting that the behavior of the MACSYMA determinant function in all but its default mode was quite decent when acting on the smaller Fuka matrices. In large part, this was due to determinant possessing a direct interface to the CRE package, where a very efficient implementation of sparse polynomial arithmetic is available. Once again, underlying details of implementation played a major role in performance considerations.

Finally, the routine that appeared to have the most general success was MAPLE's hybrid procedure, which for the characteristic matrices tested here, would start off with one-step Bareiss elimination and then switch over to minor expansion if the pivots became too complicated. This idea worked quite effectively for the smaller matrices, and only ran into serious trouble (as nearly every method did) on the 100×100 Sulsky1 matrix. The results for the 100×100 Wilkinson matrix were also not that good, but they were better than those for MACSYMA's pure Gaussian elimination routines. In conclusion, each method had its faults, and so the user needs to beware (and be aware)!

Epilogue

The Final Simplification

Once before a console dreary, while I pondered, weak and weary,
Over many a quaint and curious volume of Macsyma lore—
While I wavered, nearly napping, suddenly there came a snapping,
A snap, snap, snap reverberating throughout the core.
"'Tis some daemon," I muttered, "examining my network door—
Only this and nothing more."

It was a long ago hour in May, I still remember to this day,
When this apparition first upon me bore.
I had eagerly sought to reduce untold numbers of symbols loose,
Loose throughout my machine's memory store—
Just this and nothing more.

Vainly, I had tried to crunch my symbols down to a decent bunch,
But with no success; there was no clever method I could adore.
I had wished to bring to fruition the final goal of my tuition,
The very last cause that drove me now ahead and to the fore—
Simply this and nothing more.

The mad shimmering of my screen was the last thing I had seen,
Thrilling me with fantastic errors I had never encountered before.
And now came this gentle snapping, almost as if something was quietly tapping,
Tapping into my computer's core,
Seeking its way in—to be released, nevermore!

I knew then this was no turtle, but instead another hurdle,
Impeding me from the goal I could only now abhor.
It was a symbolic processor, its algorithms mere gossamer,
Poking its ever expanding heap into the my system's memory store—
Planning to hold onto each and every bit forevermore!

Soon, I would lose all ability for swapping, my calculations would all be flopping,
And yet it would still demand even more!
I knew then there was no turning back; I would soon have to make my attack,
My attack on this monstrous memoryivore—
Now or nevermore!

So, strengthening my emaciated will, I stalked my potential kill,
Making monitoring its process status my only chore.
Suddenly, with an outrageous cry, I knew finally it would have to die;
I threw an interrupt at it like the Hammer of Thor—
To stop it now and forevermore!

Unfortunately, the system hung, and I knew right away I would be done,
Done, unless I lept to a distant shore.
"So it's to be hardball," I thought. "Then that's how it will be fought,
Man versus symbol manipulator, a contest not unlike war—
Raw power to the victor; to the vanquished, nothing more!"

With a nervous twitch, I hunted for the switch
To reboot the system and wipe clean the core.
I pushed the rocker cleanly and smiled a little meanly
With full knowledge that I had finally cleared the score,
For now and forevermore!

Ah, but the system did not reboot or make even the least little hoot,
And this made my anger wax very sore.
I thought then, "I will remove its damned felicity by eliminating its source of electricity,"
And squatted purposefully to the floor—
"Once the cord goes, this beast will haunt me nevermore!"

Well, I tugged and twisted, but the cord easily resisted,
And out of nowhere, it appeared I had become embued with gore.
My own body was now struggling against the process still juggling
Symbols in an ever widening sweep around the computer's core—
Ever wider, they began to seep into my body's pores.

Many years since have passed, but I have yet to see my last,
And the last of the symbols which I now abhor.
The only revolution has been my traumatic evolution
Into a symbiotic memory hog, thrashing on the floor—
Rationally simplifying for now and forevermore!

Appendix A

Test Matrices

Below are presented the nontrivial matrices which were used to test out various determinant and characteristic polynomial algorithms implemented in existing computer algebra systems. These matrices were obtained from a variety of sources. The most interesting examples (i.e., the larger matrices and those with several symbolic parameters) are typical of applications in physics and engineering. Of course, these are the more difficult to deal with since they can be quite complex. Subsequent appendices will present timing data for which these matrices were the objects of the calculations.

A.1 TRIDIAGONAL MATRICES

A.1.1 FMM Matrices

These matrices were taken from [For77, p. 61]. They occur in the determination of cubic splines over a region divided into equal-sized intervals ($n-1$ intervals corresponds to an $n \times n$ matrix). The matrices are tridiagonal with super/subdiagonal elements equal to 1 and diagonal elements $\{-1, 4, 4, \ldots, 4, -1\}$. For example, the 5×5 FMM matrix is

$$\begin{pmatrix} -1 & 1 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 \\ 0 & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 1 & -1 \end{pmatrix} .$$

A.1.2 Wilkinson Matrices

These matrices were taken from [Wil65, p. 307]. Their diagonal elements are all equal to 2 and their super/subdiagonals are all 1's. The eigenvalues of an $n \times n$ Wilkinson matrix are given explicitly by

$$\left\{ 4 \sin^2 \left[\frac{\pi k}{2(n+1)} \right] \right\}_{k=1}^{n} .$$

A.2 SULSKY4 MATRICES

These matrices were described in a private communication from Deborah Sulsky. They come up in a finite difference problem in which the time evolution at a grid point depends on the point's nearest horizontal and vertical neighbors. The matrices are block tridiagonal with super/subdiagonal blocks filled in with identity submatrices. The diagonal blocks are themselves tridiagonal with diagonal elements all equal to -4 and all super/subdiagonal elements equal to 1. These matrices further have the property that if the dimension of each block in one is $n \times n$ then the dimension of the matrix as a whole is $n^2 \times n^2$. For example, the 9×9 Sulsky4 matrix is

$$\left(\begin{array}{ccc|ccc|ccc} -4 & 1 & 0 & & & & & & \\ 1 & -4 & 1 & & I & & & O & \\ 0 & 1 & -4 & & & & & & \\ \hline & & & -4 & 1 & 0 & & & \\ & I & & 1 & -4 & 1 & & I & \\ & & & 0 & 1 & -4 & & & \\ \hline & & & & & & -4 & 1 & 0 \\ & O & & & I & & 1 & -4 & 1 \\ & & & & & & 0 & 1 & -4 \end{array}\right) .$$

A.3 MESSY MATRICES

A.3.1 Tournier Matrix (5×5)

Taken from [Tou79, p. 161]. This matrix has an order 5 elementary divisor given by

$$x(x^3 - 2x^2 - x + 2) .$$

Tournier =

$$\left(\begin{array}{ccc} 5x^4 - 10x^3 - 5x^2 + 10x - 1 & x^4 - 2x^3 - x^2 + 2x - 1 & -x^4 + 2x^3 + x^2 - 2x + 1 \\ 5x^4 - 10x^3 - 5x^2 + 4x + 1 & x^4 - 2x^3 - x^2 + 4x + 1 & -x^4 + 2x^3 + x^2 - 4x + 3 \\ 0 & 0 & 4 \\ 5x^4 - 10x^3 - 5x^2 + 7x - 2 & x^4 - 2x^3 - x^2 + 3x - 2 & -x^4 + 2x^3 + x^2 - 3x + 2 \\ 3x(x-3) & x(-x+3) & x(x-3) \end{array}\right.$$

$$\left.\begin{array}{cc} 2(-x^4 + 2x^3 + x^2 - 2x + 1) & 0 \\ 2(-x^4 + 2x^3 + x^2 - 1) & 0 \\ 0 & 0 \\ 2(-x^4 + 2x^3 + x^2 - x + 2) & 0 \\ 2x(-x+3) & 4x(x-2) \end{array}\right)$$

A.4 SMALL MATRICES

A.4.1 Fox Matrix (4×4)

Taken from [Fox65, p. 82]. The determinant of this matrix is 1042.

$$\text{Fox} = \begin{pmatrix} 7 & 9 & -1 & 2 \\ 4 & -5 & 2 & -7 \\ 1 & 6 & -3 & -4 \\ 3 & -2 & -1 & -5 \end{pmatrix}$$

A.4.2 Wilkinson1 Matrix (5×5)

Taken from [Wil65, p. 212]. Under Gaussian elimination, this matrix produces maximally sized elements in its right hand column.

$$\text{Wilkinson1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{pmatrix}$$

A.4.3 Moler Matrix (5×5)

Taken from a private communication from Cleve Moler. All the eigenvalues of this matrix are zero.

$$\text{Moler} = \begin{pmatrix} -9 & 11 & -21 & 63 & -252 \\ 70 & -69 & 141 & -421 & 1684 \\ -575 & 575 & -1149 & 3451 & -13801 \\ 3891 & -3891 & 7782 & -23345 & 93365 \\ 1024 & -1024 & 2048 & -6144 & 24572 \end{pmatrix}$$

A.4.4 Rosser Matrix (8×8)

Taken from a private communication from Cleve Moler. This matrix is notorious for causing numerical eigenvalue routines to fail. The eigenvalues of this matrix are, in order of descending value,

$$\{10\sqrt{10405},\ 1020,\ 510 + 100\sqrt{26},\ 1000,\ 1000,\ 510 - 100\sqrt{26},\ 0,\ -10\sqrt{10405}\}\ .$$

$$\text{Rosser} = \begin{pmatrix} 611 & 196 & -192 & 407 & -8 & -52 & -49 & 29 \\ 196 & 899 & 113 & -192 & -71 & -43 & -8 & -44 \\ -192 & 113 & 899 & 196 & 61 & 49 & 8 & 52 \\ 407 & -192 & 196 & 611 & 8 & 44 & 59 & -23 \\ -8 & -71 & 61 & 8 & 411 & -599 & 208 & 208 \\ -52 & -43 & 49 & 44 & -599 & 411 & 208 & 208 \\ -49 & -8 & 8 & 59 & 208 & 208 & 99 & -911 \\ 29 & -44 & 52 & -23 & 208 & 208 & -911 & 99 \end{pmatrix}$$

A.4.5 Hankel Matrix (9×9)

Taken from [Gro87]. This matrix has four eigenvalues equal to zero.

$$\text{Hankel} = \begin{pmatrix} -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 \end{pmatrix}$$

A.4.6 Wester Matrix (10×10)

Invented by the author.

$$\text{Wester} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 \\ 12 & 23 & 34 & 45 & 56 & 67 & 78 & 89 & 90 & 1 \\ -1 & 2 & -3 & 4 & -5 & 6 & -7 & 8 & -9 & 0 \\ 5 & 8 & 5 & 4 & 0 & 2 & 6 & 8 & 9 & -9 \\ 3 & 2 & 7 & 1 & 9 & 5 & 5 & 1 & 2 & 3 \\ 0 & 11 & 22 & 33 & 44 & 55 & 66 & 77 & 88 & 99 \\ 0 & 11 & 22 & -33 & -44 & -55 & 66 & 77 & 88 & -99 \\ 5 & 0 & 5 & 2 & 5 & 5 & 8 & 4 & 0 & 8 \\ 1 & 0 & 1 & 0 & 1 & 9 & 8 & 6 & 1 & 4 \\ 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 \end{pmatrix}$$

A.4.7 Schwarz Matrix (11×11)

Taken from [Sch73, p. 174].

$$\text{Schwarz} = \begin{pmatrix} 5 & -4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -4 & 6 & -4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 6 & -4 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 6 & -4 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -4 & 6 & -4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -4 & 6 & -4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -4 & 6 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -4 & 6 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -4 & 6 & -4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -4 & 5 \end{pmatrix}$$

A.4.8 JR Matrix (12×12)

Taken from [Joh81, p. 312].

$$\text{JR} = \begin{pmatrix} 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 11 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 10 & 10 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 9 & 9 & 9 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 8 & 8 & 8 & 8 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 7 & 7 & 7 & 7 & 7 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 5 & 4 & 3 & 2 & 1 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 4 & 3 & 2 & 1 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 3 & 2 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 2 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

A.4.9 Wilkinson2 Matrix (6×6)

Taken from [Wil65, p. 425]. The eigenvalues of this matrix are all on the order of one.

$$\text{Wilkinson2} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 2^{-k} & 2^{k} & 2^{k} & 0 & 0 \\ 0 & 0 & 2^{k} & 2^{k} & 2^{-k} & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

A.4.10 Cullen Matrix (3×3)

Taken from [Cul72, p. 231].

$$\text{Cullen} = \begin{pmatrix} 2x+5 & 2x^2-x-15 & 4x^2-5x-27 \\ x+3 & x^2+x-8 & x^2-2x-15 \\ x+2 & x^2-x-6 & 2x^2-3x-11 \end{pmatrix}$$

A.4.11 Buckley Matrix (3×3)

$$\text{Buckley} = \begin{pmatrix} -2u & -iv & -w \\ w & ix & \frac{u}{\sqrt{3}+y} \\ -iv & \frac{u}{\sqrt{3}-y} & ix \end{pmatrix}$$

A.4.12 SU3 Matrix (8×8)

Taken from a private communication from Daniel Finley. This is the matrix of commutators for the algebra SU(3). The eigenvalues of this matrix will be invariants of the algebra.

$$\text{SU3} = \begin{pmatrix} 0 & -2a_2 & 2a_3 & 0 & a_5 & -a_6 & -a_7 & a_8 \\ -a_3 & 2a_1-a_4 & 0 & 0 & 0 & -a_8 & -a_5 & 0 \\ a_2 & 0 & a_4-2a_1 & 0 & -a_7 & 0 & 0 & -a_6 \\ 0 & a_2 & -a_3 & 0 & -2a_5 & 2a_6 & -a_7 & a_8 \\ 0 & 0 & a_8 & -a_6 & 2a_4-a_1 & 0 & a_2 & 0 \\ 0 & a_7 & 0 & a_5 & 0 & a_1-2a_4 & 0 & a_3 \\ a_8 & -a_6 & 0 & a_8 & a_3 & 0 & a_4+a_1 & 0 \\ -a_7 & 0 & -a_5 & -a_7 & 0 & a_2 & 0 & -a_4-a_1 \end{pmatrix}$$

A.5 FUKA MATRICES

These matrices were described in a private communication from Mary Fuka. They are used in the stability analysis of the periodic orbits of a nonlinear physical system, which consists of a charged particle trapped in a one-dimensional square well moving under the influence of an external cosine dependent forcing field. A periodic trajectory of this particle is described by $2n$ nonlinear equations, where $2n$ is the total number of contacts made (in alternation) with the well walls. Considering small deviations in the contact times of an arbitrary periodic trajectory of this system, a set of $2n$ recursion relations can be derived for the pertubations. The determinant of the coefficient matrix will then produce a quadratic polynomial, whose roots will determine a cumulative out of phase factor. Stability or instability of a particular periodic orbit will depend on the form of this phase factor, which in turn will depend on the specific contact times which go into generating the coefficient matrix. These matrices have the general form (for a $2n \times 2n$ matrix) of

$$\mathrm{gen}(2n) = \begin{pmatrix}
x_1 & y_1 & z_1 & & & & & & \\
 & x_2 & y_2 & z_2 & & O & & & \\
 & \ddots & \ddots & \ddots & & & & & \\
 & & x_{n-1} & y_{n-1} & z_{n-1} & & & & \\
 & & & x_n & y_n & x_n & & & \\
 & & & & z_{n-1} & y_{n-1} & x_{n-1} & & \\
 & & O & & & \ddots & \ddots & \ddots & \\
 & & & & & & z_2 & y_2 & x_2 \\
ux_1 & & & & & & & z_1 & y_1 \\
uy_{2n} & ux_{2n} & & & & & & & x_{2n}
\end{pmatrix}.$$

There are also "symmetric" forms [designated by sym$(2n)$] in which many of the above parameters are not independent. These correspond to degeneracies in the periodic orbits. One such matrix used in the testing was

$$\mathrm{sym}(6) = \begin{pmatrix}
x_1 & y_1 & z_1 & 0 & 0 & 0 \\
0 & -z_1 & -y_1 & -x_1 & 0 & 0 \\
0 & 0 & x_3 & y_3 & x_3 & 0 \\
0 & 0 & 0 & -x_1 & -y_1 & -z_1 \\
ux_1 & 0 & 0 & 0 & z_1 & y_1 \\
-uy_3 & -ux_3 & 0 & 0 & 0 & -x_3
\end{pmatrix}.$$

Appendix B

Raw Data Tables

The following tables present raw data obtained by performing a lengthy series of eigenproblem related timing tests on a variety of matrices. These tests included computing determinants, characteristic polynomials and symbolic and numeric eigenvalues using both available existing and newly implemented methods in several of the major computer algebra systems (CASs). Table B.1 lists the versions of the computer algebra systems that were used and on what type of computer they were executed. The middle column displays either the version date or the copyright date associated with each system. The various versions of a given computer algebra system are reasonably similar for the most part so that references to specific version numbers will only be provided when the differences are important (especially distinguishing Macsyma 415.69, which is written in Common Lisp, from the earlier versions of Macsyma, which were written in Franz Lisp). All computers listed in the third column of Table B.1 ran some variant of the UNIX operating system.

Macsyma 309.4	1986	Sun 3/160 with 8 Mb memory
Macsyma 309.6	1987	Sun 3/160 with 4 Mb memory
Macsyma 415.69	1989	Sun 3/160 with 8 Mb memory
Maple 4.1	May 1987	Sun 3/160 with 4 Mb memory
Maple 4.3	Mar 1989	Sequent-Symmetry with 8 Mb memory
Mathematica 1.1 (sun3.fpa)	September 17, 1988	Sun 3/160 with 8 Mb memory
Reduce 3.3	15–Jan–88	Sun 3/160 with 4 Mb memory
Scratchpad II	1989	IBM RT/PC with 16 Mb memory

Table B.1 CASs and computers used.

All times recorded in the raw data tables are in seconds. If a number in parentheses follows a timing entry, then this means that the timing was averaged over this number of trials. A number in square brackets is a garbage collection (GC) time from Macsyma, and is provided separately for certain entries in which the GC time is a significant fraction of

the total time (which is the sum of the CPU and GC times). If a greater than sign (>) precedes a time, this indicates that the calculation was interrupted (presumably because it had already run a long time) and so the length of the complete calculation will be bounded from below by the given value. A superscripted letter following an entry designates that a fatal error at this time caused the computation to terminate (the list of error messages will be presented later in the given section). A numerical superscript acts more like an ordinary footnote.

Ordinarily, the output from a calculation was printed, and the times presented will reflect this. For some calculations or series of calculations, the output was suppressed and this is indicated by a dagger (†) for an individual entry or a note at the beginning of a computational series. Some timings also include file loading times (not all packages in a computer algebra system are necessarily placed in core initially). These entries are designated by an asterisk (*). Additional notes are provided before the calculations to which they refer.

N in the tables below will always refer to the size of the matrix (which will always be square and so its dimensions will be $N \times N$). The names of the actual functions used are displayed in a sans serif type style. On occasion, a function name will be followed by parenthesized characters. These will indicate the particular options that were applied in this instance and will be identified in a "Key" somewhere above the table.

Several different complexity measures were used at various times to give an idea of the size of an expression. $\mathcal{N}_{10}$ is defined by equation (3.1). The Macsyma function length counts the number of terms in an expression while complexity computes the total number of operators and atomic operands of an expression. The Maple function length, on the other hand, computes something like the number of bytes taken up by an expression. Finally, the Maple "words used" refers to the number of words of memory that were requested to perform a calculation (some of these may have been from reallocations, however).

In a table, double vertical lines are used to separate distinct computational series, while single vertical lines divide closely related calculation sequences. An example of the latter can be seen in the first table below due to a "very simple numerical diagonal". The third and fourth columns display the times required to compute the characteristic polynomial and then to factor it for a pair of different matrix sizes. Many similar groupings occur throughout the entire appendix. The matrices employed in the timing tests will be generally specified by the section titles and if they are simple enough, described in subsection headers. The more complicated test matrices are produced in Appendix A.

B.1 DIAGONAL MATRICES

B.1.1 Very Simple Numerical Diagonal: $\{1, 1, 1, \ldots\}$

Key: † = output suppressed

	Scratchpad II		
N	eigenvalues	characteristicpol	factor
10	1.8165 (2)	1.3165 (2)	0.333
100	193.6655 (2)	171.732†	27.5

B.1.2 Simple Numerical Diagonal: $\{1, 2, 3, \ldots\}$

	Mathematica
N	Eigenvalues
1	0.05
2	0.2
3	0.333333
4	0.783333
5	0.666667
6	1.41667
7	1.1
8	1.9
9	1.81667
10	3.31667

B.1.3 Simple Symbolic Diagonal: $\{a, b, c, \ldots\}$

	Mathematica
N	Eigenvalues
1	0.0333333
2	0.35
3	1.08333
4	2.81667
5	8.03333
6	29.2833
7	137.233
8	767.9
9	4957.7
10	$>$8500

B.1.4 Sulsky1 Diagonal: $\{2-a,\ 2-a,\ 2-a, \ldots\}$

	Maple				
			Sequent-Symmetry		
N	eigenvals	charpoly	eigenvals	charmat	det(sparse)
100	342.57	248.65	652.65	∼38	22.65

	Mathematica		Reduce		
N	Eigenvalues	charpoly	mateigen	charpoly	charpoly(factor)
100	>8718	1678.22	6.239	>173	54.859

Table B.2

Key: † = output suppressed

	Scratchpad II			
N	eigenvalues	eigenvalues(best)	characteristicpol	factor
3	0.4 (2)	0.367	0.3	—
4	0.95 (2)	0.6	—	—
5	1.183 (2)	0.833	—	—
6	7.8835 (2)	1.667	—	—
7	1.6 (2)	1.2	—	—
8	2.017 (2)	1.467	—	—
9	8.9165 (2)	1.833	—	—
10	7.078 (3)	2.767	2.2	0.766
100	1691.301 (2)	1545.634†	—	—

Table B.3

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
D = diagonal, H = Hermitian, G = general
* = includes one or more file loading times

	Macsyma				
N	charpoly	charpoly(R)	charpoly(FR)	charpoly(RS)	charpoly(FRS)
2	$\sim$0	0.1245 (2)	0.1245 (2)	0.150	0.250
10	11.350	48.391 (2)	5.283 (2)	2.583	3.933
100	5997[a]	>60440[b,c]	2745.483	1538.133	387.416

N	meigenvalues applied to the above characteristic polynomials			
2	0.066	1.358* (2)	1.208* (2)	
10	0.200	10.933 (2)	0.0415 (2)	
100	—	—	0.083	

Table B.4

N	mcharpoly(D)	mcharpoly(H)	mcharpoly(G)
2	1.183*	0.016	0.033
10	0.166	1.416	0.550
100	4.816 [3.150]	8.583 [3.716]	211.083 [4.316]

N	meigenvalues applied to the above characteristic polynomials		
2	0.116	0.116	3.816 [3.733]
10	0.066	0.083	0.150
100	0.100	0.066	0.166

Table B.5

		415.69		
N	ncharpoly	loop	det	det(FR)
2	0.350	0.0665 (2)	0.216	1.300*
10	65.800	0.4495 (2)	12.700	19.750
100	76237[d]	4.533 (2)	15952.350	—

N	meigenvalues applied to the above characteristic polynomials			
2	1.483*		trivial	trivial
10	7.583		trivial	trivial

Table B.6

N	loop	newdet	newdet(F)
2	0.033 (2)	0.016	0.050
10	0.150 (2)	5.983	2.550
20	0.283	>6063	—
25	0.383	0.333[e]	—
100	1.816 (2)	0.016[f]	$\sim$0[f]

N	meigenvalues applied to the above characteristic polynomials		
2		1.366*	0.250
10		7.150	5.783

Table B.7

Fatal errors:

[a] For sbrk from lisp: no space... Goodbye!

[b] ERROR: Could not allocate space in MARKDP.

[c] NOT ENOUGH SPACE FOR ARRAY
Totaltime= ⟨printr:bad lisp data: 0x21c684⟩ msec. so far
GCtime= ⟨printr:bad lisp data: 0x21c684⟩ msec.

[d] Internal bad memory reference, you are advised to (reset).

[e] NOT ENOUGH SPACE FOR ARRAY — *and upon trying to quit:*
|Error:| SORRY, ABSOLUTE PAGE LIMIT HAS BEEN REACHED

— **Macsyma** 415.69:
Unrecoverable error: Can't allocate. Good-bye!.
Signal 6 caught.
The internal memory may be broken.
You should check the signal and exit from Lisp.

[f] Array too big – NEWDET: 100

B.2 TRIDIAGONAL MATRICES

B.2.1 Random 4-Digit Integer Entries

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.

	Macsyma	
N	mcharpoly	$\mathcal{N}$(max coeff)
3	0.033	12
10	0.633	39
25	7.750 [3.116]	93
50	64.466 [39.016]	189
100	496.033 [320.400]	375

Table B.8

B.2.2 FMM Matrices (Diagonal = $\{-1, 4, 4, \ldots, 4, -1\}$, Super/Subdiagonals = 1)

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.

	Macsyma	
N	mcharpoly	factor
2	0.050	0.050
3	0.066	0.216
4	0.050	3.466 [2.966]
5	0.116	0.966
6	0.183	1.000
7	0.250	4.500 [3.033]
8	0.283	2.450
9	0.300	6.116 [3.083]
10	0.416	6.416 [3.016]

Table B.9

B.2.3 Wilkinson Matrices (Diagonal = 2, Super/Subdiagonals = 1)

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
T = tridiagonal, H = Hermitian, G = general
* = includes one or more file loading times

	Maple		Mathematica		
N	charpoly	factor	charpoly	Factor	Eigenvalues
2	0.54	0.41	0.0666667	0.15	0.166667
10	5.19	12.05	0.65	0.783333	1.86667
100	2291.75	14964.00	>102481	—	—

Table B.10

Note: mateigen returned an unfactored characteristic polynomial.

	Reduce			
N	charpoly	factor	charpoly(factor)	mateigen
2	0.170	0.034	0.136	0.119
10	0.561	3.468	21.267	2.261
100	2085.934	896.138	—	—

Table B.11

	Macsyma		
N	mcharpoly	meigenvalues	# of lines
2	∼0	0.166	1
3	0.050	0.350	1
4	0.083	0.683	3
5	0.100	0.933	1
6	0.166	1.550	43
7	0.233	4.766 [3.083]	5
8	0.250	1.900	39
9	0.333	3.083	13
10	0.416	6.383 [3.166]	5 (factored polynomial)

Table B.12

Note: From here on down, meigenvalues used factor rather than solve.

N	charpoly	charpoly(R)	charpoly(FR)	charpoly(RS)	charpoly(FRS)
2	0.016	0.033	0.066	0.133	0.183
10	20.591 (2)	1.983	2.316	1.800	2.850
100	314[a]	3608.250	37.083[b]	380.483	386.416

N	length [complexity] of the above characteristic polynomials				
2	2 [9]	3 [8]	2 [7]	3	2
10	6 [738]	11 [48]	6 [35]	11	6
100	—	101 [498]	—	101	51

N	meigenvalues applied to the above characteristic polynomials			
2	0.216	1.200*	0.200	
10	10.0165 (2)	5.883	6.516	

Table B.13

N	mcharpoly(T)	mcharpoly(H)	mcharpoly(G)
2	2.150*	0.016	0.050
10	0.450	0.500	1.150
100	138.166 [80.116]	136.200 [71.016]	1597.866 [768.183]

N	length [complexity] of the above characteristic polynomials		
2	3 [8]	3 [8]	3 [8]
10	11 [48]	11 [48]	11 [48]
100	101 [498]	101 [498]	101 [498]

N	meigenvalues applied to the above characteristic polynomials		
2	0.200	0.150	0.233
10	2.666	6.300 [3.583]	6.883 [4.133]

Table B.14

N	ncharpoly	loop + newdet	*415.69* loop + det(FR)
2	0.200	0.049	1.366*
10	17.850	2.850	59.883
15	—	95.182	—
20	—	[c]	—
25	—	0.699[d]	—
100	>54201	—	>41490

N	length [complexity] of the above characteristic polynomials		
2	3 [8]	3 [9]	2
10	11 [48]	11 [25]	2
15	—	16 [35]	—

N	meigenvalues applied to the above characteristic polynomials		
2	1.266*	3.266	
10	2.733	2.650	
15	—	11.083	

Table B.15

B.2.4 Sulsky1 Matrices (Diagonal = $2 - a$, Super/Subdiagonals = -1)

Key: † = output suppressed

	Maple		Mathematica		Reduce	
N	charpoly	eigenvals	charpoly	Eigenvalues	charpoly	mateigen
2	0.22	0.32	—	—	—	—
3	0.37	3.43	0.1	0.85	0.255	—
4	0.55	8.23	0.216667†	1.8	0.153†	—
5	0.75	5.87	0.383333†	1.65	0.255†	—
6	1.41	—	0.616667†	6.41667†	0.238†	—
7	1.87	17.28	0.916667†	4.23333	0.323†	—
8	2.44	—	1.35†	9.4†	0.408†	—
9	3.30	30.59	1.88333†	6.75	0.527†	—
10	4.28	48.94	2.53333†	7.48333[a]	0.748†	—
100	>3972[1]	—	—	—	810.747[b]	614.414[b]

[1] User time. 9371 seconds of system time were also used.

Fatal errors:

[a] Eigenvalues::eivfl: Algorithm failed to find eigenvalues.

[b] ***** Heap space low

Table B.16

	Scratchpad II		
N	eigenvalues	characteristicpol	factor
3	4.64175 (4)	0.45 (2)	0.567
4	1.133	0.433†	—
5	1.933	0.667†	—
6	10.3	0.967	—
7	3.5335 (2)	8.0† (2)	—
8	5.733	1.7†	—
9	5.7335 (2)	8.8835† (2)	—
10	13.35 (2)	2.667†	17.134
100	—	[a]	—

Fatal errors:
[a] >> System error:
Not enough storage after GC.

Table B.17

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
T = tridiagonal, H = Hermitian, G = general, C = CRE
* = includes one or more file loading times

	Macsyma			
N	mcharpoly	meigenvalues	# of lines	
2	2.050*	0.217	1	
3	0.150	1.150	1	
4	0.333	6.200	7	
5	0.700	13.733	1	
6	1.216	40.850	43	
7	2.034	121.250	5	
8	3.417	380.950	41	
9	5.316	1247.917	15	
10	8.267	4125.466	16	(factored polynomial)

Table B.18

Note: From here on down, meigenvalues used factor rather than solve.

N	charpoly	charpoly(R)	charpoly(FR)	charpoly(RS)	charpoly(FRS)
2	$\sim$0	0.100	0.066	0.116	0.283
10	22.733	6.800	6.300	4.150	4.783
25	>1935	199.483	40.316	—	—
100	—	10780[a]	3438.733	>620[b]	714.050

N	length [complexity] of the above characteristic polynomials				
2	2 [12]	5 [18]	2 [8]	5	2
10	6 [1035]	21 [310]	6 [40]	21	6
25	—	51 [1724]	13 [103]	—	—
100	—	—	51 [400]	—	51

N	meigenvalues applied to the above characteristic polynomials			
2	1.300	1.233*	1.450*	
10	35.650	18.966	18.233	
25	—	577.350*	377.533	

Table B.19

N	mcharpoly(T)	mcharpoly(H)	mcharpoly(G)
2	2.187* (4)	0.033	0.066
10	8.3705 (4)	13.683	16.450
25	678.066 (2)	665.682	708.833
100	>7681[a,c]	—	—

N	length [complexity] of the above characteristic polynomials		
2	4 [19]	4 [19]	4 [19]
10	35 [597796]	35 [597796]	35 [597796]
25	129 [1]	129	129

N	meigenvalues applied to the above characteristic polynomials		
2	0.22075 (4)	0.366	0.300
10	5685.6745 (2)	5392.500	5368.950

[1] Complexity computation was interrupted after 64233 seconds.

Table B.20

N	mcharpoly(T:C)	mcharpoly(T:CF)	mcharpoly(H:C)	mcharpoly(G:C)
2	2.1053* (3)	2.266*	0.016	0.100
10	5.7995 (2)	191.850	6.233	7.566
25	102.533	26670.416	103.900	161.166
50	1018.150	—	873.366	1206.500
75	5219.533	—	—	—
100	>1071[a,c]	—	—	—

N	length [complexity] of the above characteristic polynomials			
2	5 [18]	5 [18]	5 [18]	5 [18]
10	21 [310]	21 [310]	21 [310]	21 [310]
25	51 [1720]	51 [1720]	51 [1720]	51 [1720]
50	101 [6570]	—	101 [6570]	101 [6570]
75	151 [14545]	—	—	—

N	meigenvalues applied to the above characteristic polynomials			
2	0.283 (3)	0.283	0.250	0.250
10	15.2415 (2)	15.250	15.400	15.583
25	574.033	817.283	575.266	587.766
50	3738.050	—	3839.300	3998.316
75	20301.583	—	—	—

Table B.21

N	ncharpoly	loop + newdet	*415.69* loop + det(FR)
2	1.666	0.082	1.483*
10	588.000	3.499	200.832
25	>16924[d]	—	—
100	—	—	>41328

N	length [complexity] of the above characteristic polynomials		
2	5 [18]	5 [21]	2
10	21 [310]	21 [157]	2

N	meigenvalues applied to the above characteristic polynomials		
2	0.266	0.300	
10	16.450	12.333	

Table B.22

Fatal errors:

[a] For sbrk from lisp: no space... Goodbye!

[b] Unrecoverable error: signal 6 caught (during GBC). — *many of these*

[c] Space request would exceed maximum memory allocation
[Storage space totally exhausted]
Space exhausted when allocating list

[d] Space request would exceed maximum memory allocation

B.2.5 Sulsky2 Matrices (Diagonal = $2 - a_i$, Super/Subdiagonals = -1)

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.

Key: C = CRE, R = RATMX
* = includes one or more file loading times

	Macsyma				
N	charpoly	rat	charpoly(C)	charpoly(R)	ncharpoly
2	0.016	0.016	0.050	0.150	1.733*
3	0.083	0.050	0.150	0.333	1.000
4	0.100	0.116	0.483	0.766	6.066
5	0.216	0.283	3.900	5.066	18.833
6	0.383	0.666	1.966	6.333	64.316
7	0.883	4.816	10.883	13.583	204.566
8	1.966	13.833	23.783	31.600	630.400
9	5.200	35.400	64.416	68.633	1754.566
10	18.333	68.216	113.583	135.116	5439.683

N	mcharpoly	rat	mcharpoly(C)
2	0.083	0.016	0.066
3	0.166	0.133	0.233
4	0.350	0.700	0.650
5	0.766	6.450	4.816
6	1.383	30.083	7.666
7	5.566 [3.133]	112.433	25.666
8	3.866	378.733	79.716
9	6.383	1345.966	193.166
10	13.300 [3.316]	5095.516	—

Table B.23

Fatal errors:
[a] For sbrk from lisp: no space... Goodbye!
[b] Non-number to minus g00044
[c] ERROR: Could not allocate space in MARKDP.
[d] NOT ENOUGH SPACE FOR ARRAY

B.3 SULSKY4 MATRICES

Notes: Characteristic polynomial output was suppressed.
Numerical eigenvalues were computed for nonlinear factors.
All times are total times which may include GC time.

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
G = general, C = CRE, D = HDIVFREE, Dn = HDIVFREE:NOGCDS
H = Hermitian, C = CRE
M = MINHERMPIVOT, Ma = MINHERMPIVOT:ALWAYS
* = includes one or more file loading times

	Macsyma				
N	charpoly	charpoly(R)	charpoly(FR)	charpoly(RS)	charpoly(FRS)
4	0.116	0.233	0.300	3.308 (2)	0.683
9	10.533	5.067	5.000	13.316	14.216
16	54332.884	11.784	10.966	2255.700	2253.316
25	24233[a]	55.000	36.334	[b]	[b]
36	—	149.250	109.817	—	—
49	—	469.700	318.550	—	—
64	—	1669.467	847.466	—	—
81	—	4184.750	2258.600	—	—
100	—	7331.833	4623.100	—	—

N	meigenvalues applied to the above characteristic polynomials			
4	0.450	0.450	0.216	
9	9.883	1.317	0.816	
16	817.617	7.034	5.383	
25	—	22.250	17.867	
36	—	54.366	39.267	
49	—	123.767	91.150	
64	—	291.150	214.750	
81	—	610.700	474.917	
100	—	1361.950	1083.500	

Table B.24

N	hessenberg(H)	hessenberg(H:C)	hessenberg(H:M)	hessenberg(H:Ma)
4	0.100	0.217	0.117 (2)	0.083
9	0.550	4.200	0.5085 (2)	0.467
16	6.533	13.234	6.6745 (2)	7.234
25	33.967	58.817	75.6255 (2)	4221.916
36	783.050	952.883	930.117 (2)	>46234
49	1290.650	1653.267	6638.5585 (2)	—
64	29375.873	32097.924	>86907	—
81	>47016	>47029	—	—

N	mcharpoly applied to the above transformed matrices			
4	0.034	0.150	0.0835 (2)	0.034
9	3.383	0.400	3.308 (2)	3.217
16	0.833	1.533	0.8085 (2)	0.800
25	3.017	8.117	3.125 (2)	4.933
36	30.300	36.034	27.2335 (2)	—
49	59.133	71.116	62.350 (2)	—
64	292.050	345.184	—	—

N	meigenvalues applied to the above characteristic polynomials			
4	0.500	0.400	0.425 (2)	0.417
9	1.100	1.216	1.2085 (2)	1.200
16	8.350	8.167	8.2085 (2)	7.700
25	24.400	28.233	27.8665 (2)	28.367
36	52.583	56.650	56.383 (2)	—
49	158.333	163.767	159.9835 (2)	—
64	336.433	342.449	—	—

Table B.25

N	hessenberg(G)	hessenberg(G:C)	hessenberg(G:D)	hessenberg(G:Dn)
4	0.100	0.200	0.217	0.200
9	3.633	1.316	4.450	4.467
16	3.333	15.433	13.967	58.050
25	19.034	61.067	53.984	>121945
36	68.967	193.800	185.483	—
49	186.950	511.467	459.350	—
64	521.550	1274.350	1076.400	—
81	1461.100	3081.533	—	—
100	3745.000	6983.116	—	—

N	mcharpoly applied to the above transformed matrices			
4	0.084	0.117	0.100	0.084
9	0.217	0.500	0.234	0.284
16	3.800	5.233	0.900	2696.234
25	5.900	10.050	11.417	—
36	17.167	32.867	263.533	—
49	49.950	85.150	2671.884	—
64	183.516	278.033	>4028[c]	—
81	780.667	1182.133	—	—
100	2550.616	3655.583	—	—

N	meigenvalues applied to the above characteristic polynomials			
4	0.433	0.467	0.450	0.400
9	1.200	1.234	1.334	1.334
16	7.634	7.800	8.317	255.584
25	24.900	25.084	29.400	—
36	66.250	68.883	91.600	—
49	126.450	127.400	266.816	—
64	275.950	295.850	—	—
81	694.100	777.834	—	—
100	1378.500	1709.167	—	—

Table B.26

N	ncharpoly	newdet	*415.69* loop + det(R)	ratsimp
4	3.883	0.183	1.883	0.233
9	8.917	7.300	54.083	11.550
16	92.384	483.583	1707.716	12.650
25	644.100	[d]	30098.383	512.000
36	3325.317	—	>25275	—
49	12943.084	—	—	—

N	meigenvalues applied to the above characteristic polynomials		
4	0.383	0.350	
9	1.183	1.150	
16	10.200	8.666	
25	27.867	—	
36	50.433	—	
49	111.300	—	

Table B.27

Fatal errors:

[a] Space request would exceed maximum memory allocation
[Storage space totally exhausted]
Space exhausted when allocating list

[b] Unrecoverable error: signal 6 caught (during GBC). — *many of these*

[c] For sbrk from lisp: no space... Goodbye!

[d] NOT ENOUGH SPACE FOR ARRAY

B.4 UPPER HESSENBERG MATRICES

B.4.1 Random 4-Digit Integer Entries

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.

	Macsyma	
N	mcharpoly	$\mathcal{N}_{10}$(max coeff)
3	0.050	12
10	1.216	38
25	36.900 [18.566]	93
50	433.950 [231.583]	189
100	4967.283 [2152.166]	380

Table B.28

B.5 MESSY MATRICES

B.5.1 Tournier Matrix (5×5)

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.
complexity refers to the complexity of the characteristic polynomial.

Key: c = charpoly, F = RATFAC, R = RATMX
m = mcharpoly, C = CRE, D = HDIVFREE

	Macsyma			
METHOD	hessenberg	charpoly/ mcharpoly	meigenvalues	complexity
c	—	0.366	6.416 [3.283]	403
c(R)	—	6.783	6.516 [3.216]	150
c(FR)	—	16.950	2.800	360
newdet	—	0.816	6.616 [3.216]	73
m	0.266	0.433	55.400 [35.100]	2032
m(D)	1.483	0.950	150.450 [82.383]	13318
m(C)	5.333	6.683	1.766	85
m(CD)	1.750	5.866	9.650 [3.700]	706

B.6 SMALL MATRICES

Notes: Numerical eigenvalues were computed for nonlinear factors (**Macsyma**).
All times are total times which may include GC time (**Macsyma**).
Both characteristic polynomials and eigenvectors were computed by mateigen (**Reduce**).

Key: # = number of lines of output, † = output suppressed
″ = **Macsyma** 415.69, F = RATFAC, R = RATMX, S = SPARSE
c = CRE and not transformed into general representation, C = CRE
H = Hermitian (symmetric), T = tridiagonal, B = symmetric banded

B.6.1 Fox Matrix (4×4)

METHOD	times	#
Macsyma		
charpoly†	0.200	—
charpoly	0.166	9
factor	1.033	2
charpoly(R)†	0.350	—
charpoly(R)	0.300	2
factor	0.566	2
charpoly(R)″	0.450	2
factor″	1.100	2
charpoly(RS)″	0.566	2
factor″	1.233	2
hessenberg	0.183 (2)	—
mcharpoly†	0.183	—
mcharpoly	0.1495 (2)	2
meigenvalues	4.133 [3.133]	—
hessenberg(C)	0.266	—
mcharpoly(c)	0.283	2
mcharpoly(C)	0.233	2
meigenvalues	3.816 [2.966]	—

METHOD	times	#
Maple		
det	0.31 (3)	—
charpoly	0.62 (3)	2
factor	1.13	2
Mathematica		
Det	0.0444 (3)	—
charpoly	0.1944 (3)	2
Factor	0.3	2
Reduce		
det	0.0907 (3)	—
charpoly	0.2323 (3)	2
factor	1.190 (3)	2
mateigen†	0.2777 (3)	—
mateigen	0.391	2
Scratchpad II		
characteristicpol	1.267	2
eigenvalues†	27.1665 (2)	—
eigenvalues	24.733	—
numeric	21.9	—

B.6.2 Wilkinson1 Matrix (5×5)

METHOD	times	#
Macsyma		
charpoly†	0.400	—
charpoly	0.383	6
factor	0.700	2
charpoly(R)†	0.500	—
charpoly(R)	0.500	2
factor	0.550	2
charpoly(R)″	0.650	2
factor″	0.966	2
charpoly(RS)″	0.633	2
factor″	0.916	2
hessenberg	0.1745 (2)	—
mcharpoly†	0.133	—
mcharpoly	0.233	2
meigenvalues	10.350	—
hessenberg(C)	0.300	—
mcharpoly(c)	0.400	2
mcharpoly(C)	0.383	2
meigenvalues	10.583	—
Maple		
charpoly	1.02	2
factor	0.43	2
Mathematica		
charpoly	0.183333	2
Eigenvalues	0.566667	[a]
Reduce		
charpoly	0.306	2
mateigen	0.544	2
Scratchpad II		
characteristicpol	0.6	2
factor	0.333	2
eigenvalues	7.783 (2)	[1]

B.6.3 Moler Matrix (5×5)

METHOD	times	#
Macsyma		
charpoly†	0.516	—
charpoly	0.433	79
factor	4.950	2
charpoly(R)†	0.450	—
charpoly(R)	0.466	2
factor	0.033	2
charpoly(R)″	0.650	2
factor″	0.033	2
charpoly(RS)″	0.916	2
factor″	0.066	2
hessenberg	3.808 [3.1665] (2)	—
mcharpoly†	0.350	—
mcharpoly	0.333	2
meigenvalues	0.016	1
hessenberg(C)	0.766	—
mcharpoly(c)	0.483	2
mcharpoly(C)	0.516	2
meigenvalues	$\sim$0	1
Maple		
charpoly	1.23	2
eigenvals	1.37	1
Mathematica		
charpoly	0.55	2
Eigenvalues	0.616667	1
Reduce		
charpoly	0.442	2
mateigen	0.493	1
Scratchpad II		
characteristicpol	1.8 (2)	2
eigenvalues	1.5	1

B.6.4 Rosser Matrix (8×8)

METHOD	times	#
Macsyma		
charpoly†	14.400	—
charpoly	14.1495 (2)	b
charpoly(R)†	6.033 [3.133]	—
charpoly(R)	2.800	5
factor	5.300 [3.183]	2
charpoly(R)″	2.916	5
factor″	3.266	2
charpoly(RS)″	20.900	5
factor″	3.366	2
hessenberg	7.3915 [3.2245] (2)	—
mcharpoly†	2.516	—
mcharpoly	2.583	5
meigenvalues[2]	7.116 [3.683]	3
meigenvalues	3.250	—
hessenberg(C)	8.516 [3.200]	—
mcharpoly(c)	6.500 [3.300]	5
mcharpoly(C)	6.516 [3.316]	5
meigenvalues	3.133	—
hessenberg(H)	64.4165 (2)	—
mcharpoly†	3.366	—
mcharpoly	3.383	5
meigenvalues	6.516 [3.133]	—

METHOD	times	#
Maple		
charpoly	4.27	5
factor	7.11	2
eigenvals	9.94	4
Mathematica		
charpoly	8.8	8
Factor	1.03333	5
Eigenvalues	10.1833	7
Reduce		
charpoly	6.120	7
factor	2.482	2
solve	2.618	13
mateigen	23.817	6
Scratchpad II		
characteristicpol	29.083 (2)	5
factor	1.167	2
eigenvalues	34.2777 (3)	5
numeric	1.4	—

B.6.5 Hankel Matrix (9×9)

METHOD	times	#
Macsyma		
charpoly†	102.433	—
charpoly	85.150	1433
factor	114.483	2
charpoly(R)†	1.650	—
charpoly(R)	1.700	2
factor	0.550	2
charpoly(R)″	2.000	2
factor″	1.233	2
charpoly(RS)″	50.150	2
factor″	1.283	2
hessenberg	0.9995 (2)	—
mcharpoly†	0.233	—
mcharpoly	0.300	2
meigenvalues	0.966	—
hessenberg(C)	2.500	—
mcharpoly(c)	0.683	2
mcharpoly(C)	0.683	2
meigenvalues	0.833	—
hessenberg(H)	4.0495 [1.583] (2)	—
mcharpoly†	0.316	—
mcharpoly	0.300	2
meigenvalues	4.150 [3.083] (2)	—

METHOD	times	#
Maple		
charpoly	3.93	2
factor	0.58	2
Mathematica		
charpoly	4.85	2
Factor	0.5	2
Eigenvalues†	6.25	—
Eigenvalues	6.36667	—
Reduce		
charpoly	3.162	2
factor	1.088	2
mateigen	7.242	2
Scratchpad II		
characteristicpol	9.93325 (4)	2
factor	2.6585 (2)	2
eigenvalues†	43.767	—
numeric	8.2	—

B.6.6 Wester Matrix (10×10)

METHOD	times	#
Macsyma		
charpoly†	135.450	—
charpoly	135.850	b
charpoly(R)†	3.816	—
charpoly(R)	4.000	5
factor	24.450	5
charpoly(R)″	8.983	5
factor″	19.483	5
charpoly(RS)″	179.483	5
factor″	19.216	5
charpoly(FR)	207.750	328
charpoly(FRS)″	277.433	319
hessenberg	34.766 (2)	—
mcharpoly†	43.583	—
mcharpoly	46.533	5
meigenvalues	29.533	—
hessenberg(C)	40.766	—
mcharpoly(c)	51.066	5
mcharpoly(C)	50.950	8
meigenvalues	29.600	—
Maple		
charpoly	6.98	5
factor	2.04	5
Mathematica		
charpoly	112.883	7
Factor	0.7	7
Reduce		
charpoly	22.474	8
factor	5.678	7
mateigen	63.376	7
Scratchpad II		
characteristicpol	50.4 (2)	5
factor	1.567	5
eigenvalues	65.5	1

B.6.7 Schwarz Matrix (11×11)

METHOD	times	#
Macsyma		
charpoly†	88.933	—
charpoly	91.816	1873
factor	161.416	5
charpoly(R)†	3.016	—
charpoly(R)	3.033	5
factor	24.683	5
charpoly(R)″	3.850	5
factor″	31.633	5
charpoly(RS)″	9.266	5
factor″	33.700	5
charpoly(FR)	9.500	19
charpoly(FRS)″	24.016	17
hessenberg	2.5915	—
mcharpoly†	7.550 [3.350]	—
mcharpoly	7.666 [3.516]	5
meigenvalues	22.233	—
hessenberg(C)	8.933 [3.633]	—
mcharpoly(c)	9.500 [3.633]	5
mcharpoly(C)	9.516 [3.683]	5
meigenvalues	22.750	—
hessenberg(B)	66.491 (2)	—
mcharpoly†	12.466 [3.150]	—
mcharpoly	12.783 [3.366]	5
meigenvalues	22.083	—
Maple		
charpoly	7.57	5
factor	29.67	5
Mathematica		
charpoly	2.05	8
Reduce		
charpoly	1.394	8
mateigen	18.411	7
Scratchpad II		
characteristicpol	14.711 (3)	5
factor	2.333	6
eigenvalues†	30.633	—

B.6.8 JR Matrix (12×12)

METHOD	times	#
Macsyma		
charpoly†	731.300	—
charpoly	754.250	b
charpoly(R)†	4.750	—
charpoly(R)	4.750	5
factor	10.133	5
charpoly(R)″	10.566	5
factor″	7.183	5
charpoly(RS)″	2345.900	5
factor″	7.383	5
charpoly(FR)	4861.567	b
charpoly(FRS)″	2584.283	2201
hessenberg	2.5245 (2)	—
mcharpoly†	1.283	—
mcharpoly	1.166	5
meigenvalues	16.433	—
hessenberg(C)	14.416	—
mcharpoly(c)	6.816 [3.500]	5
mcharpoly(C)	6.900 [3.450]	5
meigenvalues	16.616	—
hessenberg(H)	94.133 (2)	—
mcharpoly†	2.416	—
mcharpoly	2.416	5
meigenvalues	19.283	—
Maple		
charpoly	11.97	5
factor	12.59	8
Mathematica		
charpoly	>146182	—
Reduce		
charpoly	36.448	5
factor	4.658	5
mateigen	12.818	7
Scratchpad II		
characteristicpol	128.7835 (2)	5
factor	0.8	8
eigenvalues	129.0	1

B.6.9 Wilkinson2 Matrix (6×6)

METHOD	times	#
Macsyma		
charpoly†	0.383	—
charpoly	0.400	17
factor	6.100 [3.100]	5
charpoly(R)†	3.933 [3.100]	—
charpoly(R)	0.883	8
factor	2.600	5
charpoly(R)″	1.316	8
factor″	5.583	5
charpoly(RS)″	1.283	8
factor″	5.783	5
hessenberg	1.8245 [1.625]	—
mcharpoly†	0.983	—
mcharpoly	1.000	286
meigenvalues	30.700	5
hessenberg(C)	0.416	—
mcharpoly(c)	1.383	5
mcharpoly(C)	1.400	5
meigenvalues	6.300 [3.666]	5
hessenberg(T)	0.160 (2)	—
mcharpoly†	0.966	—
mcharpoly	0.933	286
meigenvalues	29.900	5
Maple		
charpoly	1.83	8
factor	1.17	5
Mathematica		
charpoly	0.35	11
Factor	3.63333	5
Reduce		
charpoly	0.748	11
factor	5.338	5
mateigen	4.760	10
Scratchpad II		
characteristicpol	38.733	**c**
factor	0.667	**d**

B.6.10 Cullen Matrix (3×3)

METHOD	times	#
Macsyma		
charpoly†	0.066	—
charpoly	0.100	11
factor	1.183	4
charpoly(R)†	0.383	—
charpoly(R)	0.416	4
factor	0.800	4
charpoly(R)″	0.750	5
factor″	6.916	4
charpoly(RS)″	0.666	5
factor″	1.966	4
hessenberg	0.141 (2)	—
mcharpoly†	0.316	—
mcharpoly	0.316	89
meigenvalues	12.833	5
hessenberg(C)	0.200	—
mcharpoly(c)	0.266	5
mcharpoly(C)	0.250	5
meigenvalues	0.816	5
Maple		
charpoly	0.52	2
factor	1.61	2
Mathematica		
charpoly	0.35	5
Reduce		
charpoly	0.374	5
factor	1.955	4
mateigen	0.765	6
Scratchpad II		
characteristicpol	0.717 (2)	5
factor	30.833	e
eigenvalues	6.467	—

B.6.11 Buckley Matrix (3×3)

METHOD	times	#
Macsyma		
charpoly†	0.100	—
charpoly	0.116	11
factor	4.533 [2.950]	11
charpoly(R)†	1.016	—
charpoly(R)	0.966	5
factor	0.783	5
charpoly(R)″	1.916	8
factor″	2.166	5
charpoly(RS)″	1.583	8
factor″	2.233	5
hessenberg	0.116 (2)	—
mcharpoly†	0.200	—
mcharpoly	0.200	54
meigenvalues	17.350	11
hessenberg(C)	0.233	—
mcharpoly(c)	4.500 [3.666]	11
mcharpoly(C)	4.550 [3.733]	11
meigenvalues	1.316	5
Maple		
charpoly	1.70	13
factor	0.82	13
Mathematica		
charpoly	0.2	9
Reduce		
charpoly	0.578	8
mateigen	1.054	10
Scratchpad II		
characteristicpol	—	f
eigenvalues	—	f

B.6.12 SU3 Matrix (8×8)

METHOD	times	#
Macsyma		
charpoly†	10.200 [3.416]	—
charpoly	6.666	1039
factor	>4219	—
charpoly(R)†	74.100 [43.900]	—
charpoly(R)	78.983 [48.550]	59
factor	3091	b
charpoly(R)″	81.633	59
factor″	>3121	—
charpoly(RS)″	23.133	59
charpoly(FR)	166.900	302
charpoly(FRS)″	40.450	296
hessenberg	315.3915 [222.583] (2)	—
mcharpoly†	23.333	—
mcharpoly	>3431	—
hessenberg(C)	>1688	—
Maple		
charpoly	24.32	77
Mathematica		
charpoly	23.2667	86
Reduce		
charpoly	7.905	83
mateigen	>2107	—
Scratchpad II		
characteristicpol†	457.733	—
characteristicpol	536.333	—
factor	31.867	77

1 Only some or none of the eigenvalues were determined.
2 All symbolic eigenvalues were computed.

Fatal errors:
a Eigenvalues::eivfl: Algorithm failed to find eigenvalues.
b For sbrk from lisp: no space... Goodbye!
c Cannot find applicable characteristicpol for given argument.
d Cannot find applicable eigenvalues for given argument.
e Cannot find applicable factor for given argument.
f The domain P QF I does not belong to the category FIELD.

B.6.13 Hessenberg Similarity Forms

This table shows the number of digits of the largest entry for the numerical matrices in this section and for their Hessenberg similarity forms (or tridiagonal for matrices treated as Hermitian [H]).

matrix	N	$\mathcal{N}_{10}$(max entry)	$\mathcal{N}_{10}$(max entry of similarity form)
Fox	4	1	6
Wilkinson1	5	1	1
Moler	5	5	14
Rosser	8	3	39
Rosser (H)	8	3	140
Hankel	9	1	3
Hankel (H)	9	1	14
Wester	10	2	190
Schwarz	11	1	48
Schwarz (H)	11	1	186
JR	12	2	5
JR (H)	12	2	22

Table B.29

B.7 FUKA MATRICES

Key: gen(N) = general $N \times N$, sym(N) = symmetric $N \times N$
1 = METHOD:=bareiss1, 2 = METHOD:=bareiss2, s = METHOD:=sparse
T = matrix transposed

	Maple			
matrix	det	Det(1)	Det(2)	Det(s)
sym(6)	2.33	2.06	1.53	0.68
gen(6)	2.81	2.70	2.14	0.57
gen(8)	10.42	10.02	9.06	1.37
gen(10)	33.56	34.84	29.93	3.73
gen(16)	1583.14	1640.75	1483.64	497.34

matrix	Words used in computing the above determinants			
sym(6)	49884	46151	35667	17781
gen(6)	60405	60219	48243	10543
gen(8)	197896	200757	176368	20722
gen(10)	659443	670759	578256	55877
gen(16)	23109520	23837865	21626963	5962461

matrix	# of lines [length] of the above determinants			
sym(6)	5 [189]	5 [189]	5 [189]	5 [189]
gen(6)	11 [395]	11 [395]	11 [395]	11 [395]
gen(8)	30 [964]	30 [964]	30 [964]	30 [964]
gen(10)	69 [2466]	69 [2466]	69 [2466]	67 [2466]
gen(16)	1480 [38630]	1480 [38630]	1480 [38630]	1480 [38630]

Table B.30

Notes: All results in this section refer to sym(34).
None of the calculations completed.
Max stage is the last stage of the elimination successfully completed.
Accumulated time is the time used through the max stage.

METHOD	max stage	accumulated time	time	words used
det[1]	16[a]	∼3881	5368.27	11077632
Det(1)[2]	13[a]	929.80	1310.18	3661824
Det(s)[1]	13[b]	5.00	∼22555	—
Det(s)[2]	33[a]	7047.55	19625.52	4101766
Det(sT)[1]	33[a]	9.13	34582.73	11223040
Det(sT)[2]	33[a]	7.43	20030.47	3864092

[1] Sun 3/160 with 16 Mb memory
[2] Sun 3/60 with 8 Mb memory

Fatal errors:
[a] System error, ran out of memory
[b] Segmentation fault (from UNIX)

Table B.31

Note: All times are total times which may include GC time.

Key: gen(N) = general $N \times N$, sym(N) = symmetric $N \times N$
F = RATFAC, R = RATMX, S = SPARSE (415.69)
1 = METHOD:BAREISS1, 2 = METHOD:BAREISS2, s = METHOD:SPARSE
r = RAT, T = matrix transposed, C = "better pivot" checking was performed

	Macsyma			
matrix	determinant	determinant(R)	determinant(FR)	determinant(Fr)
sym(6)	0.550	0.966	1.000	35.349
gen(6)	0.533	1.217	1.166	58.816
gen(8)	2.300	3.300	3.400	547.466
gen(10)	22.333	12.933	10.016	5280.399
gen(16)	35939.667	135.166	130.833	>78789

matrix	complexity of the above determinants			
sym(6)	177	86	86	448
gen(6)	173	95	95	514
gen(8)	451	244	244	1577
gen(10)	1173	565	565	4316
gen(16)	20873	8355	8355	—

matrix	ratsimp applied to the above determinants			
sym(6)	0.300	0.166	0.583	0.700
gen(6)	0.366	0.233	0.700	0.850
gen(8)	0.983	0.633	2.150	5.933
gen(10)	5.916	1.483	22.650	14.966
gen(16)	74.283	56.716	448.000	—

matrix	complexity of the above ratsimp'ed determinants			
sym(6)	86	86	91	91
gen(6)	100	100	100	100
gen(8)	242	242	242	242
gen(10)	542	542	542	542
gen(16)	7879	7879	7879	—

Table B.32

matrix	determinant(RS)	determinant(FRS)
sym(6)	1.066	1.250
gen(6)	1.416	1.550
gen(8)	3.166	3.450
gen(10)	11.633	6.883
gen(16)	116.250	117.483

matrix	complexity of the above determinants	
sym(6)	86	86
gen(6)	100	100
gen(8)	232	232
gen(10)	517	517
gen(16)	7370	7370

matrix	ratsimp applied to the above determinants	
sym(6)	0.550	1.500
gen(6)	0.933	2.116
gen(8)	2.533	10.900
gen(10)	5.850	20.816
gen(16)	139.416	640.483

matrix	complexity of the above ratsimp'ed determinants	
sym(6)	86	91
gen(6)	100	100
gen(8)	242	242
gen(10)	542	542
gen(16)	7879	7879

Table B.33

matrix	Det(1)	Det(2)	Det(s)	Det(1r)	Det(2r)	Det(sr)
sym(6)	6.683	6.666	2.700	11.766	11.700	7.566
gen(6)	6.683	6.183	2.733	12.733	12.516	8.000
gen(8)	9.833	12.300	7.716	35.933	39.733	24.916
gen(10)	18.133	16.666	13.366	87.633	91.216	64.766
gen(16)	63.650	61.016	26.133	3015.816	3062.299	2726.883

matrix	complexity of the above determinants					
sym(6)	279	266	219	69	69	69
gen(6)	287	248	212	136	136	136
gen(8)	731	651	548	331	331	331
gen(10)	1829	1626	1408	720	720	720
gen(16)	31017	27684	24692	10411	10411	10411

matrix	ratsimp applied to the above determinants					
sym(6)	0.650	0.616	0.433	0.250	0.283	0.250
gen(6)	1.000	0.883	0.683	0.350	0.400	0.383
gen(8)	9.666	9.916	2.100	0.966	1.000	0.967
gen(10)	30.550	30.850	21.400	2.300	2.316	2.216
gen(16)	2415.266	2308.050	2262.216	40.100	40.183	39.100

matrix	complexity of the above ratsimp'ed determinants					
sym(6)	86	86	86	86	86	86
gen(6)	100	100	100	100	100	100
gen(8)	242	242	242	242	242	242
gen(10)	542	542	542	542	542	542
gen(16)	7879	7879	7879	7879	7879	7879

Table B.34

Notes: All results in this section refer to sym(34).
Max stage is the last stage of the elimination successfully completed.

METHOD	max stage	time	complexity	ratsimp
determinant(RS)	—	>84002	—	—
determinant(FRS)	—	>96006	—	—
Det(s)	33	96.616	—	93269[a]
Det(Cs)	33	7634.316	141859844[1]	—
Det(sT)	33	116.150	263969848[2]	>100086
Det(Crs)	24[b]	4467.200	—	—
Det(rsT)	23[c]	2414.950	—	—
Det(Frs)	18	>148037	—	—
Det(s)[3]	31[d]	78237.435	—	—

Table B.35

[1] The complexity computation alone required 4650.333 seconds!
[2] The complexity computation alone required 8379.133 seconds!
[3] After each stage of the elimination, the final value of the corresponding diagonal element was ratsimp'ed.

Fatal errors:
[a] Attempt to allocate beyond static structures (20).
[Storage space totally exhausted]
[b] Space request would exceed maximum memory allocation
[Storage space totally exhausted]
Space exhausted when allocating list
[c] Space request would exceed maximum memory allocation
[Storage space totally exhausted]
Space exhausted when allocating fixnum
[d] For sbrk from lisp: no space... Goodbye!

Appendix C

Normalized Timings

The tables presented below contain the results of selected timing tests taken from the preceding appendix which have been normalized to a consistent standard. In general, the computational series that were chosen involved matrices that were 10×10 or larger and/or consisted of calculations that used up large amounts of CPU time. The sections are arranged so that the matrices examined exhibit a general trend of increasing complexity. Within each section, all the times are normalized in units whose value in seconds is given at the top of the section. For a given section, the unit chosen corresponds to the time it takes to perform a base calculation. With the exception of the first section, the base calculation has been taken to be the Macsyma computation in which the output was inhibited and the standard determinant routines with RATMX set to true were used. This latter requirement corresponds to the functions determinant and charpoly. If there are matrices of more than one size present, then the computation involving the one with dimensions closest to 10×10 was selected.

Some calculational series show a definite exponential (or greater) trend for the computation times as a function of increasing matrix size. In these cases, tables will be repeated but with the entries replaced by their base 10 logarithms. These secondary sets of tables are labeled with the header "LOG". Other details in the format of the following tables are explained in the introduction to Appendix B.

C.1 Simple Symbolic Diagonal: $\{a, b, c, \ldots\}$

1 unit = 0.0333333 seconds

	Mathematica	
N	Eigenvalues	LOG
1	1.0	0.000
2	10.5	2.351
3	32.5	3.481
4	84.5	4.437
5	241.0	5.485
6	878.5	6.778
7	4117.0	8.323
8	23037.0	10.045
9	148731.1	11.910
10	>255000	>12.449

Table C.1

C.2 Sulsky1 Diagonal: $\{2 - a, 2 - a, 2 - a, \ldots\}$

1 unit = 48.391 seconds

	Maple				
			Sequent-Symmetry		
N	eigenvals	charpoly	eigenvals	charmat	det(sparse)
100	7.079	5.138	13.487	~0.785	0.468

	Mathematica		Reduce		
N	Eigenvalues	charpoly	mateigen	charpoly	charpoly(factor)
100	>180	34.680	0.129	6.256[a]	1.134

Fatal errors:
[a] ***** Heap space low

Table C.2

Key: † = output suppressed

	Scratchpad II			
N	eigenvalues	eigenvalues(best)	characteristicpol	factor
10	0.146 (3)	0.057	0.045	0.016
100	34.951 (2)	31.941†	—	—

Table C.3

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
D = diagonal, H = Hermitian, G = general

	Macsyma				
N	charpoly	charpoly(R)	charpoly(FR)	charpoly(RS)	charpoly(FRS)
10	0.235	1.000 (2)	0.109 (2)	0.053	0.081
100	123.928[a]	>1248[b,c]	56.735	31.786	8.005

N	meigenvalues applied to the above characteristic polynomials			
10	0.004	0.226 (2)	0.001 (2)	
100	—	—	0.002	

Table C.4

N	mcharpoly(D)	mcharpoly(H)	mcharpoly(G)
10	0.003	0.029	0.011
100	0.100 [0.065]	0.177 [0.077]	4.362 [0.089]

N	meigenvalues applied to the above characteristic polynomials		
10	0.001	0.002	0.003
100	0.002	0.001	0.003

Table C.5

		415.69		
N	ncharpoly	loop	det	det(FR)
10	1.360	0.009 (2)	0.262	0.408
100	1575.438[d]	0.094 (2)	329.655	—

N	meigenvalues applied to the above characteristic polynomials			
10	0.157		trivial	trivial

Table C.6

N	loop	newdet	newdet(F)
10	0.003 (2)	0.124	0.053
20	0.006	>125	—
25	0.008	0.007[e]	—
100	0.038 (2)	0.000[f]	∼0[f]

N	meigenvalues applied to the above characteristic polynomials		
10	—	0.148	0.120

Table C.7

Fatal errors:

[a] For sbrk from lisp: no space... Goodbye!

[b] ERROR: Could not allocate space in MARKDP.

[c] NOT ENOUGH SPACE FOR ARRAY
Totaltime= ⟨printr:bad lisp data: 0x21c684⟩ msec. so far
GCtime= ⟨printr:bad lisp data: 0x21c684⟩ msec.

[d] Internal bad memory reference, you are advised to (reset).

[e] NOT ENOUGH SPACE FOR ARRAY — *and upon trying to quit*:
|Error:| SORRY, ABSOLUTE PAGE LIMIT HAS BEEN REACHED

— **Macsyma** 415.69:
Unrecoverable error: Can't allocate. Good-bye!.
Signal 6 caught.
The internal memory may be broken.
You should check the signal and exit from Lisp.

[f] Array too big - NEWDET: 100

C.3 Wilkinson Matrices (Diagonal = 2, Super/Subdiagonals = 1)

1 unit = 1.983 seconds

	Maple		Mathematica		
N	charpoly	factor	charpoly	Factor	Eigenvalues
10	2.617	6.077	0.328	0.395	0.941
100	1155.698	7546.142	>51679	—	—

Table C.8

Note: mateigen returned an unfactored characteristic polynomial.

	Reduce			
N	charpoly	factor	charpoly(factor)	mateigen
10	0.283	1.749	10.725	1.140
100	1051.908	451.910	—	—

Table C.9

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.
meigenvalues used factor.

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
T = tridiagonal, H = Hermitian, G = general

	Macsyma				
N	charpoly	charpoly(R)	charpoly(FR)	charpoly(RS)	charpoly(FRS)
10	10.384 (2)	1.000	1.168	0.908	1.437
100	158.345[a]	1819.592	18.700[b]	191.872	194.864

N	length [complexity] of the above characteristic polynomials				
10	6 [738]	11 [48]	6 [35]	11	6
100	—	101 [498]	—	101	51

N	meigenvalues applied to the above characteristic polynomials			
10	5.051 (2)	2.967	3.286	

Table C.10

N	mcharpoly(T)	mcharpoly(H)	mcharpoly(G)
10	0.227	0.252	0.580
100	69.675 [40.401]	68.684 [35.812]	805.782 [387.384]

N	length [complexity] of the above characteristic polynomials		
10	11 [48]	11 [48]	11 [48]
100	101 [498]	101 [498]	101 [498]

N	meigenvalues applied to the above characteristic polynomials		
10	1.344	3.177 [1.807]	3.471 [2.084]

Table C.11

N	ncharpoly	loop+newdet	*415.69* loop+det(FR)
10	9.002	1.437	30.198
15	—	47.999	—
20	—	[c]	—
25	—	0.352[d]	—
100	>27332	—	>20922

N	length [complexity] of the above characteristic polynomials		
10	11 [48]	11 [25]	2
15	—	16 [35]	—

N	meigenvalues applied to the above characteristic polynomials		
10	1.378	1.336	
15	—	5.589	

Table C.12

Fatal errors:

[a] For sbrk from lisp: no space... Goodbye!

[b] Non-number to minus g00044

[c] ERROR: Could not allocate space in MARKDP.

[d] NOT ENOUGH SPACE FOR ARRAY

C.4 Sulsky1 Matrices (Diagonal = $2 - a$, Super/Subdiagonals = -1)

1 unit = 6.8 seconds

Key: † = output suppressed

	Maple		Mathematica		Reduce	
N	charpoly	eigenvals	charpoly	Eigenvalues	charpoly	mateigen
10	0.629	7.197	0.373†	1.100[a]	0.110†	—
100	>5847[1]	—	—	—	119.228[b]	90.355[b]

[1] User time. 1378.088 units of system time were also used.

Fatal errors:

[a] Eigenvalues::eivfl: Algorithm failed to find eigenvalues.

[b] ***** Heap space low

Table C.13

Key: † = output suppressed

	Scratchpad II		
N	eigenvalues	characteristicpol	factor
10	1.963 (2)	0.392†	2.520
100	—	[a]	—

Fatal errors:

[a] >> System error:
Not enough storage after GC.

Table C.14

Notes: Characteristic polynomial output was suppressed.
All times are total times which may include GC time.
meigenvalues used factor.

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
T = tridiagonal, H = Hermitian, G = general, C = CRE
* = includes one or more file loading times

	Macsyma				
N	charpoly	charpoly(R)	charpoly(FR)	charpoly(RS)	charpoly(FRS)
10	3.343	1.000	0.926	0.610	0.703
25	>284	29.336	5.929	—	—
100	—	1585.294[a]	505.696	>91[b]	105.007

N	length [complexity] of the above characteristic polynomials				
10	6 [1035]	21 [310]	6 [40]	21	6
25	—	51 [1724]	13 [103]	—	—
100	—	—	51 [400]	—	51

N	meigenvalues applied to the above characteristic polynomials			
10	5.243	2.789	2.681	
25	—	84.904*	55.520	

Table C.15

N	mcharpoly(T)	mcharpoly(H)	mcharpoly(G)
10	1.231 (4)	2.012	2.419
25	99.716 (2)	97.894	104.240
100	>1129[a,c]	—	—

N	length [complexity] of the above characteristic polynomials		
10	35 [597796]	35 [597796]	35 [597796]
25	129 [1]	129	129

N	meigenvalues applied to the above characteristic polynomials		
10	836.129 (2)	793.015	789.551

[1] complexity computation was interrupted after 9446.029 units.

Table C.16

N	mcharpoly(T:C)	mcharpoly(T:CF)	mcharpoly(H:C)	mcharpoly(G:C)
10	0.853 (2)	28.213	0.917	1.113
25	15.078	3922.120	15.279	23.701
50	149.728	—	128.436	177.426
75	767.578	—	—	—
100	$>157^{\mathbf{a,c}}$	—	—	—

N	length [complexity] of the above characteristic polynomials			
10	21 [310]	21 [310]	21 [310]	21 [310]
25	51 [1720]	51 [1720]	51 [1720]	51 [1720]
50	101 [6570]	—	101 [6570]	101 [6570]
75	151 [14545]	—	—	—

N	meigenvalues applied to the above characteristic polynomials			
10	2.241 (2)	2.243	2.265	2.292
25	84.417	120.189	84.598	86.436
50	549.713	—	564.603	587.988
75	2985.526	—	—	—

Table C.17

LOG

N	mcharpoly(T:C)	mcharpoly(T:CF)	mcharpoly(H:C)	mcharpoly(G:C)
10	-0.159 (2)	3.340	-0.087	0.107
25	2.713	8.274	2.726	3.166
50	5.009	—	4.855	5.179
75	6.643	—	—	—
100	$>5.056^{\mathbf{a,c}}$	—	—	—

N	meigenvalues applied to the above characteristic polynomials			
10	0.807 (2)	0.808	0.818	0.829
25	4.436	4.789	4.438	4.459
50	6.309	—	6.336	6.377
75	8.002	—	—	—

Table C.18

N	ncharpoly	loop + newdet	*415.69* loop + det(FR)
10	86.471	0.515	29.534
25	>2488[d]	—	—
100	—	—	>6077

N	length [complexity] of the above characteristic polynomials		
10	21 [310]	21 [157]	2

N	meigenvalues applied to the above characteristic polynomials		
10	2.419	1.814	

Table C.19

Fatal errors:

[a] For sbrk from lisp: no space... Goodbye!

[b] Unrecoverable error: signal 6 caught (during GBC). — *many of these*

[c] Space request would exceed maximum memory allocation
[Storage space totally exhausted]
Space exhausted when allocating list

[d] Space request would exceed maximum memory allocation

C.5 Schwarz Matrix (11×11)

1 unit = 3.016 seconds

Notes: Numerical eigenvalues were computed for nonlinear factors (**Macsyma**).
All times are total times which may include GC time (**Macsyma**).
Both characteristic polynomials and eigenvectors are computed by mateigen (**Reduce**).

Key: # = number of lines of output, † = output suppressed
″ = **Macsyma** 415.69, F = RATFAC, R = RATMX, S = SPARSE
c = CRE and not transformed into general representation, C = CRE
B = symmetric banded

METHOD	times	#
Macsyma		
charpoly†	29.487	—
charpoly	30.443	1873
factor	53.520	5
charpoly(R)†	1.000	—
charpoly(R)	1.006	5
factor	8.184	5
charpoly(R)″	1.277	5
factor″	10.488	5
charpoly(RS)″	3.072	5
factor″	11.174	5
charpoly(FR)	3.150	19
charpoly(FRS)″	7.963	17
hessenberg	0.859	—
mcharpoly†	2.503 [1.111]	—
mcharpoly	2.542 [1.166]	5
meigenvalues	7.372	—
hessenberg(C)	2.962 [1.205]	—
mcharpoly(c)	3.150 [1.205]	5
mcharpoly(C)	3.155 [1.221]	5
meigenvalues	7.543	—
hessenberg(B)	22.046 (2)	—
mcharpoly†	4.133 [1.044]	—
mcharpoly	4.238 [1.116]	5
meigenvalues	7.322	—
Maple		
charpoly	2.510	5
factor	9.838	5
Mathematica		
charpoly	0.680	8
Reduce		
charpoly	0.462	8
mateigen	6.104	7
Scratchpad II		
characteristicpol	4.878 (3)	5
factor	0.774	6
eigenvalues†	10.157	—

Table C.20

C.6 Sulsky4 Matrices

1 unit = 5.067 seconds

Notes: Characteristic polynomial output was suppressed.
Numerical eigenvalues were computed for nonlinear factors.
All times are total times which may include GC time.

Key: F = RATFAC, R = RATMX, S = SPARSE (415.69)
G = general, C = CRE, D = HDIVFREE, Dn = HDIVFREE:NOGCDS
H = Hermitian, C = CRE, M = MINHERMPIVOT
Ma = MINHERMPIVOT:ALWAYS

	Macsyma				
N	charpoly	charpoly(R)	charpoly(FR)	charpoly(RS)	charpoly(FRS)
4	0.023	0.046	0.059	0.653 (2)	0.135
9	2.079	1.000	0.987	2.628	2.806
16	10722.890	2.326	2.164	445.175	444.704
25	4782.514[a]	10.855	7.171	[b]	[b]
36	—	29.455	21.673	—	—
49	—	92.698	62.868	—	—
64	—	329.478	167.252	—	—
81	—	825.883	445.747	—	—
100	—	1446.977	912.394	—	—

N	meigenvalues applied to the above characteristic polynomials			
4	0.089	0.089	0.043	
9	1.950	0.260	0.161	
16	161.361	1.388	1.062	
25	—	4.391	3.526	
36	—	10.729	7.750	
49	—	24.426	17.989	
64	—	57.460	42.382	
81	—	120.525	93.727	
100	—	268.788	213.835	

Table C.21

LOG

N	charpoly	charpoly(R)	charpoly(FR)	charpoly(RS)	charpoly(FRS)
4	-3.772	-3.079	-2.830	-0.426 (2)	-2.002
9	0.732	0.000	-0.013	0.966	1.032
16	9.280	0.844	0.772	6.098	6.097
25	8.473[a]	2.385	1.970	[b]	[b]
36	—	3.383	3.076	—	—
49	—	4.529	4.141	—	—
64	—	5.798	5.120	—	—
81	—	6.716	6.100	—	—
100	—	7.277	6.816	—	—

N	meigenvalues applied to the above characteristic polynomials			
4	-2.419	-2.419	-3.147	
9	0.668	-1.347	-1.826	
16	5.084	0.328	0.060	
25	—	1.480	1.260	
36	—	2.373	2.048	
49	—	3.196	2.890	
64	—	4.051	3.747	
81	—	4.792	4.540	
100	—	5.594	5.365	

Table C.22

N	hessenberg(H)	hessenberg(H:C)	hessenberg(H:M)	hessenberg(H:Ma)
4	0.020	0.043	0.023 (2)	0.016
9	0.109	0.829	0.100 (2)	0.092
16	1.289	2.612	1.317 (2)	1.428
25	6.704	11.608	14.925 (2)	833.218
36	154.539	188.057	183.564 (2)	>9124
49	254.717	326.281	1310.156 (2)	—
64	5797.488	6334.700	>17151	—
81	>9278	>9281	—	—

N	mcharpoly applied to the above transformed matrices			
4	0.007	0.030	0.016 (2)	0.007
9	0.668	0.079	0.653 (2)	0.635
16	0.164	0.303	0.160 (2)	0.158
25	0.595	1.602	0.617 (2)	0.974
36	5.980	7.112	5.375 (2)	—
49	11.670	14.035	12.305 (2)	—
64	57.638	68.124	—	—

N	meigenvalues applied to the above characteristic polynomials			
4	0.099	0.079	0.084 (2)	0.082
9	0.217	0.240	0.239 (2)	0.237
16	1.648	1.612	1.620 (2)	1.520
25	4.815	5.572	5.500 (2)	5.598
36	10.378	11.180	11.127 (2)	—
49	31.248	32.320	31.574 (2)	—
64	66.397	67.584	—	—

Table C.23

LOG

N	hessenberg(H)	hessenberg(H:C)	hessenberg(H:M)	hessenberg(H:Ma)
4	−3.912	−3.147	−3.772 (2)	−4.135
9	−2.216	−0.188	−2.303 (2)	−2.386
16	0.254	0.960	0.275 (2)	0.356
25	1.903	2.452	2.703 (2)	6.725
36	5.040	5.237	5.213 (2)	>9.119
49	5.540	5.788	7.177 (2)	—
64	8.665	8.754	>9.750	—
81	>9.135	>9.136	—	—

N	mcharpoly applied to the above transformed matrices			
4	−4.962	−3.507	−4.135 (2)	−4.962
9	−0.403	−2.538	−0.426 (2)	−0.454
16	−1.808	−1.194	−1.833 (2)	−1.845
25	−0.519	0.471	−0.483 (2)	−0.026
36	1.788	1.962	1.682 (2)	—
49	2.457	2.642	2.510 (2)	—
64	4.054	4.221	—	—

N	meigenvalues applied to the above characteristic polynomials			
4	−2.313	−2.538	−2.477 (2)	−2.501
9	−1.528	−1.427	−1.431 (2)	−1.440
16	0.500	0.477	0.482 (2)	0.419
25	1.572	1.718	1.705 (2)	1.722
36	2.340	2.414	2.409 (2)	—
49	3.442	3.476	3.452 (2)	—
64	4.196	4.213	—	—

Table C.24

N	hessenberg(G)	hessenberg(G:C)	hessenberg(G:D)	hessenberg(G:Dn)
4	0.020	0.039	0.043	0.039
9	0.717	0.260	0.878	0.882
16	0.658	3.046	2.756	11.456
25	3.756	12.052	10.654	>24066
36	13.611	38.247	36.606	—
49	36.896	100.941	90.655	—
64	102.931	251.500	212.433	—
81	288.356	608.157	—	—
100	739.096	1378.156	—	—

N	mcharpoly applied to the above transformed matrices			
4	0.017	0.023	0.020	0.017
9	0.043	0.099	0.046	0.056
16	0.750	1.033	0.178	532.116
25	1.164	1.983	2.253	—
36	3.388	6.486	52.010	—
49	9.858	16.805	527.311	—
64	36.218	54.871	>794[c]	—
81	154.069	233.300	—	—
100	503.378	721.449	—	—

N	meigenvalues applied to the above characteristic polynomials			
4	0.085	0.092	0.089	0.079
9	0.237	0.244	0.263	0.263
16	1.507	1.539	1.641	50.441
25	4.914	4.950	5.802	—
36	13.075	13.594	18.078	—
49	24.956	25.143	52.658	—
64	54.460	58.388	—	—
81	136.984	153.510	—	—
100	272.054	337.313	—	—

Table C.25

LOG

N	hessenberg(G)	hessenberg(G:C)	hessenberg(G:D)	hessenberg(G:Dn)
4	−3.912	−3.244	−3.147	−3.244
9	−0.333	−1.347	−0.130	−0.126
16	−0.419	1.113	1.014	2.439
25	1.323	2.489	2.366	>10.089
36	2.611	3.644	3.600	—
49	3.608	4.615	4.507	—
64	4.634	5.527	5.359	—
81	5.664	6.410	—	—
100	6.605	7.229	—	—

N	mcharpoly applied to the above transformed matrices			
4	−4.075	−3.772	−3.912	−4.075
9	−3.147	−2.313	−3.079	−2.882
16	−0.288	0.032	−1.726	6.277
25	0.152	0.685	0.812	—
36	1.220	1.870	3.951	—
49	2.288	2.822	6.267	—
64	3.590	4.005	>6.677[c]	—
81	5.037	5.452	—	—
100	6.221	6.581	—	—

N	meigenvalues applied to the above characteristic polynomials			
4	−2.465	−2.386	−2.419	−2.538
9	−1.440	−1.411	−1.336	−1.336
16	0.410	0.431	0.495	3.921
25	1.592	1.599	1.758	—
36	2.571	2.610	2.895	—
49	3.217	3.225	3.964	—
64	3.997	4.067	—	—
81	4.920	5.034	—	—
100	5.606	5.821	—	—

Table C.26

			415.69	
N	ncharpoly	newdet	loop+det(R)	ratsimp
4	0.766	0.036	0.372	0.046
9	1.760	1.441	10.674	2.279
16	18.232	95.438	337.027	2.497
25	127.117	**d**	5940.080	101.046
36	656.269	—	>4988	—
49	2554.388	—	—	—

N	meigenvalues applied to the above characteristic polynomials		
4	0.076	0.069	
9	0.233	0.227	
16	2.013	1.710	
25	5.500	—	
36	9.953	—	
49	21.966	—	

Table C.27

LOG

			415.69	
N	ncharpoly	newdet	loop+det(R)	ratsimp
4	−0.267	−3.324	−0.989	−3.079
9	0.565	0.365	2.368	0.824
16	2.903	4.558	5.820	0.915
25	4.845	**d**	8.689	4.616
36	6.487	—	>8.515	—
49	7.846	—	—	—

N	meigenvalues applied to the above characteristic polynomials		
4	−2.577	−2.674	
9	−1.457	−1.483	
16	0.700	0.536	
25	1.705	—	
36	2.298	—	
49	3.089	—	

Table C.28

Fatal errors:

a Space request would exceed maximum memory allocation
[Storage space totally exhausted]
Space exhausted when allocating list

b Unrecoverable error: signal 6 caught (during GBC). — *many of these*

c For sbrk from lisp: no space... Goodbye!

d NOT ENOUGH SPACE FOR ARRAY

C.7 Wester Matrix (10×10)

1 unit = 3.816 seconds

Notes: Numerical eigenvalues were computed for nonlinear factors (**Macsyma**).
All times are total times which may include GC time (**Macsyma**).
Both characteristic polynomials and eigenvectors are computed by mateigen (**Reduce**).

Key: # = number of lines of output, † = output suppressed
″ = **Macsyma** 415.69, F = RATFAC, R = RATMX, S = SPARSE
c = CRE and not transformed into general representation, C = CRE
H = Hermitian (symmetric)

METHOD	times	#
Macsyma		
charpoly†	35.495	—
charpoly	35.600	[a]
charpoly(R)†	1.000	—
charpoly(R)	1.048	5
factor	6.407	5
charpoly(R)″	2.354	5
factor″	5.106	5
charpoly(RS)″	47.034	5
factor″	5.036	5
charpoly(FR)	54.442	328
charpoly(FRS)″	72.703	319
hessenberg	9.111 (2)	—
mcharpoly†	11.421	—
mcharpoly	12.194	5
meigenvalues	7.739	—
hessenberg(C)	10.683	—
mcharpoly(c)	13.382	5
mcharpoly(C)	13.352	8
meigenvalues	7.757	—
Maple		
charpoly	1.829	5
factor	0.535	5
Mathematica		
charpoly	29.581	7
Factor	0.183	7
Reduce		
charpoly	5.889	8
factor	1.488	7
mateigen	16.608	7
Scratchpad II		
characteristicpol	13.208 (2)	5
factor	0.411	5
eigenvalues	17.165	[1]

[1] Only some or none of the eigenvalues were determined.

Fatal errors:

[a] For sbrk from lisp: no space... Goodbye!

Table C.29

C.8 JR Matrix (12×12)

1 unit = 4.75 seconds

METHOD	times	#
Macsyma		
charpoly†	153.958	—
charpoly	158.789	a
charpoly(R)†	1.000	—
charpoly(R)	1.000	5
factor	2.133	5
charpoly(R)″	2.224	5
factor″	1.512	5
charpoly(RS)″	493.874	5
factor″	1.554	5
charpoly(FR)	1023.488	a
charpoly(FRS)″	544.060	2201
hessenberg	0.531 (2)	—
mcharpoly†	0.270	—
mcharpoly	0.245	5
meigenvalues	3.460	—
hessenberg(C)	3.035	—
mcharpoly(c)	1.435 [0.737]	5
mcharpoly(C)	1.453 [0.726]	5
meigenvalues	3.498	—
hessenberg(H)	19.817 (2)	—
mcharpoly†	0.509	—
mcharpoly	0.509	5
meigenvalues	4.060	—
Maple		
charpoly	2.520	5
factor	2.651	8
Mathematica		
charpoly	>30775	—
Reduce		
charpoly	7.673	5
factor	0.981	5
mateigen	2.699	7
Scratchpad II		
characteristicpol	27.112 (2)	5
factor	0.168	8
eigenvalues	27.158	1

[1] Only some or none of the eigenvalues were determined.

Fatal errors:

[a] For sbrk from lisp: no space... Goodbye!

Table C.30

C.9 SU3 Matrix (8×8)

1 unit = 74.1 seconds

METHOD	times	#
Macsyma		
charpoly†	0.138 [0.046]	—
charpoly	0.090	1039
factor	>56	—
charpoly(R)†	1.000 [0.592]	—
charpoly(R)	1.066 [0.655]	59
factor	41.714	[a]
charpoly(R)″	1.102	59
factor″	>42	—
charpoly(RS)″	0.312	59
charpoly(FR)	2.252	302
charpoly(FRS)″	0.546	296
hessenberg	4.256 [3.004] (2)	—
mcharpoly†	0.315	—
mcharpoly	>46	—
hessenberg(C)	>22	—
Maple		
charpoly	0.328	77
Mathematica		
charpoly	0.314	86
Reduce		
charpoly	0.106	83
mateigen	>28	—
Scratchpad II		
characteristicpol†	6.177	—
characteristicpol	7.238	—
factor	0.430	77

Fatal errors:

[a] For sbrk from lisp: no space... Goodbye!

Table C.31

C.10 Fuka Matrices

1 unit = 12.933 seconds

Key: gen(N) = general $N \times N$, sym(N) = symmetric $N \times N$
1 = METHOD:=bareiss1, 2 = METHOD:=bareiss2, s = METHOD:=sparse
T = matrix transposed

	Maple			
matrix	det	Det(1)	Det(2)	Det(s)
sym(6)	0.180	0.159	0.118	0.053
gen(6)	0.217	0.209	0.165	0.044
gen(8)	0.806	0.775	0.701	0.106
gen(10)	2.595	2.694	2.314	0.288
gen(16)	122.411	126.865	114.717	38.455

matrix	Words used in computing the above determinants			
sym(6)	49884	46151	35667	17781
gen(6)	60405	60219	48243	10543
gen(8)	197896	200757	176368	20722
gen(10)	659443	670759	578256	55877
gen(16)	23109520	23837865	21626963	5962461

matrix	# of lines [length] of the above determinants			
sym(6)	5 [189]	5 [189]	5 [189]	5 [189]
gen(6)	11 [395]	11 [395]	11 [395]	11 [395]
gen(8)	30 [964]	30 [964]	30 [964]	30 [964]
gen(10)	69 [2466]	69 [2466]	69 [2466]	67 [2466]
gen(16)	1480 [38630]	1480 [38630]	1480 [38630]	1480 [38630]

Table C.32

Notes: All results in this section refer to sym(34).
None of the calculations completed.
Max stage is the last stage of the elimination successfully completed.
Accumulated time is the time used through the max stage.

METHOD	max stage	accumulated time	time	words used
det[1]	16[a]	~300.085	415.083	11077632
Det(1)[2]	13[a]	71.894	101.305	3661824
Det(s)[1]	13[b]	0.387	~1743.988	—
Det(s)[2]	33[a]	544.928	1517.476	4101766
Det(sT)[1]	33[a]	0.705	2673.991	11223040
Det(sT)[2]	33[a]	0.574	1548.788	3864092

[1] Sun 3/160 with 16 Mb memory
[2] Sun 3/60 with 8 Mb memory

Fatal errors:
[a] System error, ran out of memory
[b] Segmentation fault (from UNIX)

Table C.33

Note: All times are total times which may include GC time.

Key: gen(N) = general $N \times N$, sym(N) = symmetric $N \times N$
F = RATFAC, R = RATMX, S = SPARSE (415.69)
1 = METHOD:BAREISS1, 2 = METHOD:BAREISS2, s = METHOD:SPARSE
r = RAT, T = matrix transposed, C = "better pivot" checking was performed

	Macsyma			
matrix	determinant	determinant(R)	determinant(FR)	determinant(Fr)
sym(6)	0.043	0.075	0.077	2.733
gen(6)	0.041	0.094	0.090	4.548
gen(8)	0.178	0.255	0.263	42.330
gen(10)	1.727	1.000	0.774	408.289
gen(16)	2778.912	10.451	10.116	>6092

matrix	complexity of the above determinants			
sym(6)	177	86	86	448
gen(6)	173	95	95	514
gen(8)	451	244	244	1577
gen(10)	1173	565	565	4316
gen(16)	20873	8355	8355	—

matrix	ratsimp applied to the above determinants			
sym(6)	0.023	0.013	0.045	0.054
gen(6)	0.028	0.018	0.054	0.066
gen(8)	0.076	0.049	0.166	0.459
gen(10)	0.457	0.115	1.751	1.157
gen(16)	5.744	4.385	34.640	—

matrix	complexity of the above ratsimp'ed determinants			
sym(6)	86	86	91	91
gen(6)	100	100	100	100
gen(8)	242	242	242	242
gen(10)	542	542	542	542
gen(16)	7879	7879	7879	—

Table C.34

matrix	determinant(RS)	determinant(FRS)
sym(6)	0.082	0.097
gen(6)	0.109	0.120
gen(8)	0.245	0.267
gen(10)	0.899	0.532
gen(16)	8.989	9.084

matrix	complexity of the above determinants	
sym(6)	86	86
gen(6)	100	100
gen(8)	232	232
gen(10)	517	517
gen(16)	7370	7370

matrix	ratsimp applied to the above determinants	
sym(6)	0.043	0.116
gen(6)	0.072	0.164
gen(8)	0.196	0.843
gen(10)	0.452	1.610
gen(16)	10.780	49.523

matrix	complexity of the above ratsimp'ed determinants	
sym(6)	86	91
gen(6)	100	100
gen(8)	242	242
gen(10)	542	542
gen(16)	7879	7879

Table C.35

matrix	Det(1)	Det(2)	Det(s)	Det(1r)	Det(2r)	Det(sr)
sym(6)	0.517	0.515	0.209	0.910	0.904	0.585
gen(6)	0.517	0.478	0.211	0.985	0.968	0.619
gen(8)	0.760	0.951	0.597	2.778	3.072	1.927
gen(10)	1.402	1.289	1.033	6.776	7.053	5.008
gen(16)	4.921	4.718	2.021	233.188	236.782	210.847

matrix	complexity of the above determinants					
sym(6)	279	266	219	69	69	69
gen(6)	287	248	212	136	136	136
gen(8)	731	651	548	331	331	331
gen(10)	1829	1626	1408	720	720	720
gen(16)	31017	27684	24692	10411	10411	10411

matrix	ratsimp applied to the above determinants					
sym(6)	0.050	0.048	0.033	0.019	0.022	0.019
gen(6)	0.077	0.068	0.053	0.027	0.031	0.030
gen(8)	0.747	0.767	0.162	0.075	0.077	0.074
gen(10)	2.362	2.385	1.655	0.178	0.179	0.171
gen(16)	186.752	178.462	174.918	3.101	3.107	3.023

matrix	complexity of the above ratsimp'ed determinants					
sym(6)	86	86	86	86	86	86
gen(6)	100	100	100	100	100	100
gen(8)	242	242	242	242	242	242
gen(10)	542	542	542	542	542	542
gen(16)	7879	7879	7879	7879	7879	7879

Table C.36

Notes: All results in this section refer to sym(34).
Max stage is the last stage of the elimination successfully completed.

METHOD	max stage	time	complexity	ratsimp
determinant(RS)	—	>6495	—	—
determinant(FRS)	—	>7423	—	—
Det(s)	33	7.470	—	7211.706[a]
Det(Cs)	33	590.297	141859844[1]	—
Det(sT)	33	8.981	263969848[2]	>7739
Det(Crs)	24[b]	345.411	—	—
Det(rsT)	23[c]	186.728	—	—
Det(Frs)	18	>11446	—	—
Det(s)[3]	31[d]	6049.442	—	—

Table C.37

[1] The complexity computation alone required 359.571 units!
[2] The complexity computation alone required 647.888 units!
[3] After each stage of the elimination, the final value of the corresponding diagonal element was ratsimp'ed.

Fatal errors:
[a] Attempt to allocate beyond static structures (20).
[Storage space totally exhausted]
[b] Space request would exceed maximum memory allocation
[Storage space totally exhausted]
Space exhausted when allocating list
[c] Space request would exceed maximum memory allocation
[Storage space totally exhausted]
Space exhausted when allocating fixnum
[d] For sbrk from lisp: no space... Goodbye!

Appendix D

Determinant and Eigenmethods in Existing Computer Algebra Systems

In Appendices B and C, raw timing data and normalized times are presented for an extensive series of calculations that were performed to examine the capabilities of various determinant and characteristic polynomial algorithms (and their implementations) in many of the existing general purpose computer algebra systems. The names of the actual functions and options that were used are clearly specified there. In this appendix, brief descriptions of those functions and options which are normally available in the systems under which they are listed are given. The rest of the methods used were developed by the author and are described in Appendices E and F.

The descriptions below are organized by computer algebra system. Along with those functions and options explicitly used in the calculations, additional entries are included to provide a more thorough summary of what software is currently available. The function and global variable names are displayed using a sans serif type style. Function arguments are *italicized* and optional arguments are enclosed in braces ({ }). For consistency, a square matrix argument will be designated by A, its size by n, its characteristic polynomial by $c_A(\lambda)$ and its eigenvalues by $\{\lambda_i\}_{i=1}^{n}$. Finally, the term "Bareiss elimination" refers to fraction-free Gaussian elimination as described in [Bar66, Bar68].

D.1 MACSYMA

See [Bog83, Com88]. All functions and options labeled with a dagger (†) are available only in the newer version of MACSYMA described in [Com88].

RATMX if true, causes matrix elements to be converted into CREs (Canonical Rational Expressions) before performing matrix operations. Otherwise, matrix operations take place using the original form of the elements (normally general representation).

RATFAC if true, invokes a partially factored form for CREs in which expressions are maintained as factored as possible without performing any new, explicit factorizations.

determinant(A) computes $\det A$, selecting an algorithm depending on the settings of various global variables [Wan77b]:

- default — Gentleman-Johnson bottom-up minor expansion [Gen74]. All possible 2×2 minors in the last two columns of A are computed. Then all 3×3 minors in the last three columns are calculated using the values of the 2×2 minors just computed. This process continues until the $n \times n$ minor (i.e., the determinant) is computed;
- RATMX:true — each row of A is multiplied by the LCM of the denominators of the elements in it and then the computation proceeds by one-step Bareiss elimination;
- RATMX:true and SPARSE:true† — conservative minor expansion. A is first transformed into block triangular form by row and column permutations [Wan76]. Starting with the leftmost column of each diagonal block, a top-down analysis is performed next, determining exactly which minors will need to be computed for a bottom-up minor expansion of the block. This results in a lattice-structure diagram of minor interdependence for each block, in which duplicate minors and those with zero multipliers have been eliminated and branches that depend solely on singular 2×2 minors have been pared off [Wan77a]. The minors are then computed in a bottom-up manner to avoid repeated computations. The $\det A$ is simply the product of the determinants of the diagonal blocks.

newdet(A {, n}) computes $\det A$ by first converting all matrix elements into CREs and then performing Gentleman-Johnson bottom-up minor expansion.

det(A)† computes $\det A$ by two-step Bareiss elimination.

charpoly(A, λ) computes $c_A(\lambda)$ from determinant($A - \lambda I$).

ncharpoly(A, λ) computes $c_A(\lambda)$ by combining traces of powers of A.

eigenvalues(A) computes $c_A(\lambda)$ from charpoly(A, λ) and then solves $c_A(\lambda) = 0$ for what roots it can, returning the unique solutions and their multiplicities.

eigenvectors(A) and uniteigenvectors(A) compute the λ_i and corresponding eigenvectors (or unit normalized eigenvectors) of A, the latter by determining a linearly independent basis for the null space of $A - \lambda_i I$.

similaritytransform(A) computes the similarity matrices S and S^{-1} such that $S^{-1}AS$ is diagonal if A can be diagonalized.

HERMITIANMATRIX if true, A is assumed to be Hermitian. This implies that any degenerate eigenvectors initially produced will be orthogonalized by means of the Gram-Schmidt process and also that the similarity matrix S^{-1} will be computed as S^H.

jordanform(A)† computes the Jordan normal form of A.

jordan_simtran(A)† if J:jordanform(A) has been performed, computes the similarity matrix S such that $S^{-1}AS = J$.

jordan_matrix_expt(A, k)† computes A^k by first transforming A into Jordan normal form.

mat_exp(A, t)† computes an exact (if possible) or approximate value of e^{At} by means of inverse Laplace transforms.

factor(*expr* {, $p(x)$}) factors *expr*. A polynomial will be factored over $\mathbf{Z}(x_1, \ldots, x_v)$. If $p(x)$ is present then the polynomial will be factored over the field of rational numbers extended by the element whose minimum polynomial is $p(x)$.

solve(*expr*, x) solves the equation *expr* or *expr* $= 0$ for x.

rat(*expr*) converts *expr* into CRE form which results in a rational simplification of *expr*.

ratsimp(*expr*) rationally simplifies *expr* by first converting it into CRE form and then back into general representation.

D.2 MAPLE

See [Cha88].

det(A {, *sparse*}) computes $\det A$, selecting an algorithm that depends on the nature of the elements in the matrix:

- floating point numbers and $\mathbf{Q}$ — Gaussian elimination with partial pivoting;
- $\mathbf{Z}$ and $\mathbf{Z}_p$ — one-step Bareiss elimination;
- $\mathbf{Z}[x]$ — the problem domain is mapped onto $\mathbf{Z}$ using a single point evaluation (the point is chosen to be a bound on the coefficients of the determinant as derived from Hadamard's inequality). One-step Bareiss elimination is then performed and the solution polynomial is constructed from the integer result by $\mathbf{Z}$-adic expansion about the previously selected evaluation point;
- general expressions — one-step Bareiss elimination is performed until no simple pivot (one that is not a sum of terms) is available, at which point, recursive expansion by cofactors is used to compute the determinant of the remaining submatrix.

If $n \leq 5$ or *sparse* is specified, then $\det A$ will be computed exclusively by recursive expansion by cofactors.

mdet(A, p) computes $\det A \pmod p$ by modular one-step Bareiss elimination when the entries of A are from either $\mathbf{Z}$, $\mathbf{Q}$ or $\mathbf{Q}[x]$.

ihermite(A) and hermite(A, x) compute the Hermite normal (reduced row echelon) form of A by performing elementary row operations. The elements of A should be from either $\mathbf{Z}$ or $\mathbf{Z}[x]$, respectively. The number of nonzero rows in the Hermite normal form is rank A, while the product of the diagonal elements will be $|\det A|$. Thus, this transformation may result in a partial factorization of the determinant. However, in general, this method is not a very efficient way to compute either the rank or the determinant of a matrix.

ismith(A) and smith(A, x) compute the Smith normal form of A by performing elementary row and column operations. The elements of A should be from either $\mathbf{Z}$ or $\mathbf{Z}[x]$, respectively. The product of the diagonal elements of the Smith normal form will be $|\det A|$. In addition, if A is of the form $B - \lambda I$, then the (n,n)-element of the Smith form will be the minimum polynomial of B.

charmat(A, λ) produces the matrix $A - \lambda I$.

charpoly(A, λ) computes $c_A(\lambda)$ from det($A - \lambda I$).

minpoly(A, λ) computes the minimum polynomial of A in terms of λ.

eigenvals(A {, B}{, *implicit*}) computes $c_A(\lambda)$ from det($A - \lambda I$) and then solves $c_A(\lambda) = 0$ for what roots it can. If an $n \times n$ matrix B is supplied, then the eigenvalues for the generalized problem $A\mathbf{x} = \lambda B\mathbf{x}$ will be computed from det($A - \lambda B$). If *implicit* is specified, then only rational λ_i [roots corresponding to linear factors of $c_A(\lambda)$] will be solved for explicitly.

eigenvects(A {, *implicit*}) computes the λ_i and corresponding eigenvectors of A, the latter by determining a linearly independent basis for the null space of $A - \lambda_i I$. Non-calculable λ_i will be expressed in an implicit form and their corresponding eigenvectors will be parameterized by the undetermined eigenvalue. If *implicit* is specified, then only rational λ_i will be solved for explicitly.

exponential(A {, t}) computes e^A (or e^{At}) if all the λ_i can be determined. If A has floating point entries, a numerical solution method will be used.

factor(*expr*) factors the polynomial *expr* over $\mathbf{Z}(x_1, \ldots, x_v)$.

D.3 Mathematica

See [Wol88].

Minors[A, k] produces all the $k \times k$ minors of A.

Det[A] computes $\det A$, selecting an algorithm that depends on the nature of the elements in the matrix:

- floating point numbers — Gaussian elimination;
- integers — modular reduction followed by Gaussian elimination (probably Bareiss);
- symbolic entries — recursive expansion by cofactors.

charpoly$[A,\ \lambda] \equiv$ Det$[A - \lambda I]$. This function is not available in standard Mathematica.

Eigenvalues$[A]$ computes $c_A(\lambda)$ from $\det(A - \lambda I)$ and then solves $c_A(\lambda) = 0$ for what roots it can.

Eigenvectors$[A$ {, ZeroTest→$test$}$]$ computes the eigenvectors of A that correspond to calculable λ_i's by determining a linearly independent basis for the null space of $A - \lambda_i I$. If the *test* is provided, then it will be used to determine when an expression is zero.

Eigensystem$[A]$ combines the results of the above two functions.

MatrixExp$[A]$ computes e^A for diagonalizable A's.

MatrixPower$[A,\ k]$ computes A^k where k may be fractional for diagonalizable A's.

Factor$[expr]$ factors the polynomial $expr$ over $\mathbf{Q}[x_1, \ldots, x_v]$.

D.4 muMATH

See [Sof83].

det(A) computes $\det A$ by recursive expansion by cofactors.

detnum(A) computes $\det A$ by Gaussian elimination.

charpol$(A,\ \lambda$ {, $WANTVEC$}$)$ computes $c_A(\lambda)$ by combining traces of powers of A [Sof86]. If $WANTVEC$ is *true*, then a general expression for the eigenvectors in terms of λ will also be computed.

D.5 REDUCE

See [Hea87].

det(A) computes $\det A$ by recursive expansion by cofactors along the first row.

charpoly$(A,\ \lambda) \equiv$ det$(A - \lambda I)$. This function is not available in standard REDUCE.

mateigen(A) returns the square free factors of $c_A(\lambda)$ and their multiplicities along with general expressions for the corresponding eigenvectors of each in terms of λ. $c_A(\lambda)$ is computed by taking the exterior product of the rows of $A - \lambda I$ which is effectively equivalent to recursive expansion by cofactors [Fla63, pp. 7–10].

FACTOR if on, causes all polynomials to be factored (including those generated in the course of a computation) over $\mathbf{Z}(x_1, \ldots, x_v)$.

solve(*expr*, x) solves the algebraic equation $expr = 0$ for x.

D.6 Scratchpad II

See [Sut88].

determinant(A) computes det A for matrices with elements forming at least a commutative ring, selecting an algorithm that depends on the highest level of algebraic structure the matrix elements form:

- field — Gaussian elimination;
- integral domain — one-step Bareiss elimination;
- otherwise — recursive expansion by cofactors.

minordet(A) computes det A by recursive expansion by cofactors.

characteristicpol(A) computes $c_A(\lambda)$ from determinant($A - \lambda I$).

eigenvalues(A) computes $c_A(\lambda)$ from $\det(A - \lambda I)$ and then solves $c_A(\lambda) = 0$ for what roots it can.

eigenvector(λ, A) finds the eigenvectors of A that correspond to the eigenvalue λ by computing a linearly independent basis for the null space of $A - \lambda I$.

eigenvectors(A) computes all rational λ_i of A and their corresponding eigenvectors. All other λ_i are expressed as implicit roots of their minimal polynomials and symbolically parameterize their corresponding eigenvectors.

eigenSolve(A) performs as above but the λ_i are solved for in terms of radicals if possible.

eigenMatrix(A) computes the similarity matrix S such that SAS^{-1} is diagonal if A can be diagonalized.

orthonormalBasis(A) computes the orthogonal matrix S such that SAS^{-1} is diagonal for A a symmetric matrix.

factor(*expr*) factors *expr* over the appropriate domain. A polynomial will be factored over $\mathbf{Z}(x_1, \ldots, x_v)$.

numeric(*expr*) converts the numerical portions of *expr* into floating point.

Appendix E

The Matrice Package

The Matrice package is a collection of functions and variables for MACSYMA written in Franz Lisp [Fod83] which allowed several of the ideas discussed in the main body of the text to be tried out in practice. In the following sections, an explanation of what a matrice is from a Lisp point of view is presented first, and then descriptions of the MACSYMA level global variables and functions in the package. In the global variable descriptions, default values are given by (default: value). For functions, arguments are in *italics* and optional arguments are enclosed in braces ({ }). Some of these functions were used in the computations presented in Appendices B and C.

E.1 Matrice Implementation in Lisp

A matrice is an atom that has the indicator MATRICE on its property list. The value of this property is a pointer to a one dimensional array. Each element in the array contains a pointer to one of a set of secondary one dimensional arrays. (These secondary arrays correspond, for example, to the rows in a general matrix or the diagonal bands in a banded matrix.) A second indicator found on the property list of the matrice is MATRICE–MATHEMATICS. Its value is one of $REAL or $COMPLEX, and indicates whether the matrix represented by the matrice is to be treated as consisting of purely real or complex elements. A third indicator presenting a property of the matrice is the MATRICE–TYPE. It can have as a value any one of the names listed in the first column of the following table:

$GENERAL	$m \times n$		
$HERMITIAN	$n \times n$		
$GENERAL_BANDED	$n \times n$	$2k-1$ bands	($m \times n$ if 1 band)
$HERMITIAN_BANDED	$n \times n$	$2k-1$ bands	
$UPPER_TRIANGULAR	$m \times n$		
$LOWER_TRIANGULAR	$m \times n$		
$UPPER_HESSENBERG	$n \times n$		
$LOWER_HESSENBERG	$n \times n$		

The data structure of the matrice will be generated according to its MATRICE–TYPE. In general, elements that are always zero (e.g., the lower subtriangle of an upper triangular matrix) or repetitious (e.g., the lower [or upper] subtriangle of a Hermitian matrix) will be omitted from the data structure, while all other elements will be represented. The second column of the table displays the possible dimensions each matrix of the indicated type can take. Here, m and n are positive integers where m need not be equal to n. The corresponding indicator on the matrice property list is called MATRICE–DIMENSIONS, and its value is a list of the row and column dimensions of the associated matrix: $(m\ n)$. The banded matrices have, in addition, the property indicator MATRICE–HALF–BANDWIDTH. This is the number of diagonal bands, including the principal diagonal, needed to cover the nonzero portion of either the upper or lower triangle of the matrix. The total number of bands corresponding to a MATRICE–HALF–BANDWIDTH of k is $2k-1$. The final indicator found on the matrice property list is MATRICE–ELEMENTS. Its value indicates the form of the elements in the matrice: $NUMERICAL (integers, rational numbers, floating point numbers, etc.), $CRE (Canonical Rational Expressions) or $GENERAL_EXPRESSIONS (all other forms). In summary, a matrice has the following indicators and corresponding values on its property list:

MATRICE	an array pointer
MATRICE–MATHEMATICS	$REAL or $COMPLEX
MATRICE–TYPE	one of the types in the previous table
MATRICE–DIMENSIONS	$(m\ n)$
†MATRICE–HALF–BANDWIDTH	k
MATRICE–ELEMENTS	$NUMERICAL, $CRE or $GENERAL_EXPRESSIONS

†This property is only present for banded matrices.

E.2 Global Variables

MATRICE_FUNCTIONS is a list of all the MACSYMA callable functions in the Matrice package, arranged in alphabetical order.

MATRICE_VARIABLES is a list of all the MACSYMA variables in the Matrice package, arranged in alphabetical order.

MATRICES is a MACSYMA list whose elements are all the matrices that have been defined in the current session.

EIGENMETHOD (default: solve) specifies the method meigenvalues will use to determine the roots of a characteristic polynomial. If EIGENMETHOD is the atomic value solve then characteristic polynomial roots will be solved for using the MACSYMA function solve. Otherwise, the value of EIGENMETHOD will be taken to be the name of a factoring

operator (e.g., factor, gfactor) and only those factors that are linear with respect to this operator will have their roots solved for, while higher degree irreducible factors will be consolidated into the (factored) residual polynomial.

EXPRSWELL (default: false) if true, will cause all basic rational arithmetic operations in the Matrice package (additions, subtractions, multiplications, divisions, exponentiations to an integer power, absolute values, explicit GCDs) to act appropriately for an expression swell analysis. This means that a rational number m/n will be treated as representing a rational number consisting of an m-digit numerator and an n-digit denominator, while an integer n will be treated as representing an n-digit integer. When performing a rational arithmetic operation, it will be assumed that all GCDs are one and that all integer arithmetic operations produce worst case behavior, that is, the results contain the greatest number of digits that are possible. To make the mathematics simpler, all rational numbers and integers must be explicitly specified (e.g., 2/3, 1) and so contain no variables. Since the arithmetic on rational numbers is slightly different from that on integers (the former produces an upper bound on the possible rational number expression swell while the latter does the same for integers under integer preserving operations), the two types of numbers are not allowed to be mixed during expression swell analyses—an error will be generated if they are. It is also an error to attempt to perform a non-integer preserving operation (such as a non-exact division) when performing arithmetic on integers under this option. If EXPRSWELL is set to the atomic value rational then an integer n will be treated as representing a rational number consisting of an n-digit numerator and an n-digit denominator during expression swell analysis. The results of expression swell analysis in this mode will give a cheap estimate of the upper bound derived when doing a full rational expression swell analysis, although this number may not always be an upper bound itself.

HDIVFREE [Hessenberg DIVision FREE algorithm] (default: false) if true, will cause hessenberg to use a division free algorithm when performing the similarity reduction of a matrice from general into upper Hessenberg form. If the initial matrice entries are polynomials, perhaps with a common divisor (produced, for example, by applying derat [q.v.] to the original matrice), then the final Hessenberg form will also have entries of this type (as will the similarity matrices). This algorithm may lead to an increase in the speed of the reduction since no GCDs are performed, however, the matrice entries can become quite large. One method of alleviating this problem to some extent is to allow GCDs (and exact divisions) to be taken occasionally. The default action (which can be prevented by setting HDIVFREE to the atomic value nogcds) is to perform some GCDs as part of a process of removing common factors from the entries of the elementary similarity matrices that are formed at each stage of the reduction (which will result in reducing the transformed matrice by the square of these factors).

HESSTRANS (default: false) if true, will allow hessenberg to perform the general Hessenberg reduction on the transpose of its matrice argument if it deems this to be advantageous

for the first stage of the transformation. The results will be transformed back to the original problem after the reduction process is completed. For some problems, setting this option may reduce the computation time and/or produce simpler looking results.

MCHARPOLY_VAR (default: lambda) is the variable in which meigenvalues will express the residual polynomial if this function is called with a matrice as the argument.

MINHERMPIVOT [MINimize HERMitian PIVOT] (default: false) if false, will cause hessenberg to not try to minimize the complexity of the pivots used during the reduction of a Hermitian matrice to tridiagonal form unless a superdiagonal element is zero at the beginning of some stage of the reduction and so a search has to be made in any case. This will minimize the number of similarity permutations that will be applied to the matrice, all of which involve actual physical swapping of matrice elements rather than implicit swapping through indexing arrays as is done when transforming a general matrice into upper Hessenberg form. If MINHERMPIVOT is true then at the beginning of each stage of the reduction, the appropriate superdiagonal element and those elements that will undergo elimination during this stage are scanned and the simplest element found will be chosen to be the pivot. Setting MINHERMPIVOT to the atomic value always will cause this process to occur at the beginning of every single step (there are, in general, several per stage) of the reduction.

MRALTSIMP [Matrice Reduction ALTernate SIMPlifier] (default: false) if not false then this should be the name of a function of two arguments which simplifies its first argument. The second argument will be an atomic descriptor indicating the form of the matrice elements (NUMERICAL, CRE or GENERAL_EXPRESSIONS) which the function may find useful for deciding how to perform its simplification. This function will be applied as an alternate to the default simplification during matrix reduction transformations (e.g., hessenberg). Thus, this allows the user to completely specify what simplifications (if any) should take place during each stage of matrix reduction processes.

MVERBOSE (default: false) if true, will cause comments about the progress of various Matrice package functions (e.g., hessenberg) to be printed as their execution proceeds.

MVERYVERBOSE (default: false) if true, will cause very detailed comments about the progress of various Matrice package functions (e.g., hessenberg) to be printed as their execution proceeds. These comments provide detailed information mainly for debugging purposes. MVERYVERBOSE implies MVERBOSE.

NUMEIGS [NUMerical EIGenvalueS] (default: false) if true, will cause meigenvalues to numerically solve for the roots (using the MACSYMA function allroots) of any univariate residual polynomials left over during the course of computing the exact roots of matrix characteristic polynomials. If NUMEIGS is false then only exact roots will be returned by meigenvalues.

OPCOUNT (initially: false) will keep a running count of the number of basic rational arithmetic operations that have been performed by functions in the Matrice package since OPCOUNT was last initialized by countops (q.v.) up until operation counts are turned off by countops(false). The normal structure of OPCOUNT is a list of two lists. The first sublist displays the operations that are being counted (additions, negations, multiplications, divisions, raising to powers, absolute values, explicit GCDs) while the second shows their corresponding counts.

USEMDIVISOR [USE Matrice DIVISOR] (default: false) if set to the atomic value mcharpoly, during an invocation of the function mcharpoly to compute the characteristic polynomial of a matrice, will result in the automatic substitution of the polynomial variable by its product with the matrice divisor (if there is one). If USEMDIVISOR is set to the atomic value meigenvalues, then during an invocation of the function meigenvalues to compute the eigenvalues of a matrice, will cause the eigenvalues discovered to be automatically divided by the matrice divisor (again, if there is one) and the residual polynomial variable to be automatically replaced by its product with the divisor. In either case, these actions will adjust the results of the respective function applications to properly account for the presence of a matrice divisor (see the description of derat). If USEMDIVISOR is false, mcharpoly and meigenvalues will ignore any divisors of their matrice arguments.

ZEROTEST (default: false) if not false then this should be the name of a function of one argument which returns true if given an expression equivalent to zero and false otherwise. This function will be applied in various situations when it is necessary to test for zeroes (e.g., in order to avoid a zero pivot) and the expression being tested has been determined not to be a simple zero (by zerop). Thus, this allows the user to impose additional zero testing if he feels that the checks for simple zeroes are not sufficient for his particular problem (e.g., dealing with general functions for which expressions equivalent to zero may not be trivially obvious to MACSYMA). Note that any expressions determined to be equivalent to zero will be immediately replaced by zero in order to simplify further calculations.

E.3 Functions

dupe(e, n) creates a list containing n copies of the item e. For example, dupe(e, 3) $\Rightarrow$ [e, e, e].

countops({*flag*}) [COUNT OPerationS] if the *flag* is true or is omitted, will, after initializing the second sublist in OPCOUNT (q.v.) to zero, cause OPCOUNT to keep a running record of the number of basic rational arithmetic operations performed subsequently by functions in the Matrice package. Calling countops with the *flag* false will cause the operation count to be terminated.

The next 10 functions provide a direct MACSYMA level interface to the expression swell and operation count portions of the Matrice package.

add_($term_1$, $term_2$, ...) adds together any number of terms.

mul_($factor_1$, $factor_2$, ...) multiplies together any number of factors.

neg_(x) takes the negative of x.

sub_(x, y) subtracts y from x.

recip_(x) takes the reciprocal of x.

div_(x, y) divides x by y.

ediv_(x, y) [Exactly DIVide] divides x by y where x is known to be an exact multiple of y.

power_(x, y) raises x to the power of y.

abs_(x) takes the absolute value of x.

gcd_(x, y) computes the greatest common divisor of x and y.

ratio(x, y) creates the ratio x/y which will NOT simplify like a rational number to lowest terms.

ndigits(e) counts the number of digits comprising the integer or rational number e. If e is a rational number then ndigits will return the ratio of the number of digits in the numerator to the number of digits in the denominator. If e is a list or a matrix, ndigits will map itself onto the elements of the list or matrix. This function is useful when doing expression swell analyses.

complexity(e) produces a measure of the complexity of the MACSYMA expression e. The result returned is a list of the classification of e followed by its complexity within that classification. The possible classifications are R (rational number or integer), F (floating point number), B (extended precision floating point number [bigfloat]) and E (expression that is not a simple number). The corresponding complexities are respectively, the sum of the absolute value of the numerator plus the denominator of a rational number (integers are considered to have a denominator of one), the absolute value of a regular or extended precision floating point number and the number of operators and atomic operands of a general expression.

matrice(*symbol, options, object*) creates a matrice described by the *options* with its elements taken from the *object* and attaches the newly created matrice to the *symbol*. The *object* may be a matrix, a two-dimensional array, the word zeroes (in which case the matrice will be filled with zeroes), the pseudofunction value(k) (in which case the nonzero portion of the matrice will be filled with the value specified by k), or a list of lists.

In the last case, the (sub)lists will be used to fill the secondary arrays of the matrice directly so that this operation performs the inverse of ice2list. (As a shorthand, a single unlistified expression will abbreviate a sublist of identical copies of this expression.) If the *object* is omitted then matrice will call entermatrice. The *options* consist of zero or more descriptors providing the parameters of the matrice. If a descriptor is a positive integer or a list containing a single positive integer then the number will be taken for the row and column dimension of the matrice. If a descriptor is a list containing two positive integers then the first integer will be taken to be the row dimension and the second the column dimension of the matrice. (Note that if the *object* that is provided to matrice is a matrix or a two-dimensional array then the *object*'s dimensions will be used for the matrice and so do not need to be explicitly specified.) A descriptor may also indicate the mathematics of the matrice (REAL or COMPLEX—default is REAL), the form of its elements (NUMERICAL, CRE [Canonical Rational Expressions] or GENERAL_EXPRESSIONS [the default]) or its type (default is GENERAL). The possible types that may be specified are given in the first column of the following table:

GENERAL	$m \times n$	
HERMITIAN	$n \times n$	
+GENERAL_BANDED	$n \times n^{\dagger}$	
+HERMITIAN_BANDED	$n \times n$	
UPPER_TRIANGULAR	$m \times n$	
LOWER_TRIANGULAR	$m \times n$	
UPPER_HESSENBERG	$n \times n$	
LOWER_HESSENBERG	$n \times n$	
SYMMETRIC	$n \times n$	REAL HERMITIAN
+BANDED	$n \times n^{\dagger}$	GENERAL_BANDED
DIAGONAL	$m \times n$	GENERAL_BANDED(1)
TRIDIAGONAL	$n \times n$	GENERAL_BANDED(2)
HERMITIAN_TRIDIAGONAL	$n \times n$	HERMITIAN_BANDED(2)
+SYMMETRIC_BANDED	$n \times n$	REAL HERMITIAN_BANDED
SYMMETRIC_TRIDIAGONAL	$n \times n$	REAL HERMITIAN_BANDED(2)
TRIANGULAR	$m \times n$	UPPER_TRIANGULAR
HESSENBERG	$n \times n$	UPPER_HESSENBERG

†May be $m \times n$ if the half-bandwidth is 1.

The first 8 entries are the basic matrice types while the remaining entries designate certain special aliases whose equivalents are given in the third column of the table. The second column displays the possible dimensions each matrice of the indicated type can take. Here, m and n are positive integers where m need not be equal to n. The (banded) types preceded by a plus (+) can also be given in the form of pseudo-functions with the half-bandwidth as their argument (for example, GENERAL_BANDED(3) indicates a

type of GENERAL_BANDED with a half-bandwidth of 3). If any necessary descriptors are omitted and cannot be defaulted then their values will be prompted for interactively. matrice returns as its value the description of the newly created matrice produced by matriceinfo.

entermatrice(*symbol, options*) allows the user to interactively enter the elements of a matrice that is described by the *options* (see the description of matrice for more information on the form of the *options*). The matrice created will be attached to the *symbol*. entermatrice will only prompt for unique, nonzero elements (for example, the upper triangle of a Hermitian or upper triangular matrix) and will do this on a row by row basis. Elements can be entered either individually or several at a time in lists. Note, however, that the elements in a list can correspond to locations in only one row of the matrice at a time. The value returned by entermatrice is the description of the created matrice produced by matriceinfo.

transmute(*A, B, options*) converts the matrice A into a form as specified by the *options* and attaches it to the symbol B (see the description of matrice for more information on the form of the *options*). The new matrice B will have the same dimensions as the original matrice A. All other properties of the original matrice, however, may be changed as long as they are kept consistent (a rectangular matrice may not be converted to a type that is only allowed to be square, for example). Any properties not specified will default to the values held by the matrice A, if possible. The effect (and implementation) of this function is very similar (but not identical for certain special cases) to matrice(*B, options,* ice2ix(*A*)). The value returned by transmute is the description of B produced by matriceinfo.

copymatrice(*A, B*) creates a distinct but identical copy of the matrice A attached to the symbol B. The value returned is the description of the copied matrice produced by matriceinfo.

mapmatrice(*A, B, f*) maps the function f onto the non-redundant elements of the matrice A. The result will be stored in the newly created matrice B. The value returned is the description of B produced by matriceinfo.

renamematrice(*A, B*) renames the matrice A to B. The value returned is the description of the matrice produced by matriceinfo.

ice2ix(*matrice*) [matrICE to matrIX] converts the *matrice* into a matrix. The matrix is returned by the function while the *matrice* is left physically unaltered.

ice2list(*matrice*) [matrICE to LIST] converts the *matrice* into a list of lists as appropriate to the *matrice* type. In other words, the arrays in the *matrice* are directly transformed into lists. The list of lists is returned by the function while the *matrice* is left physically unaltered.

diagonal(A) produces a list containing the diagonal elements (in order) of A where A is either a matrix, a two-dimensional array or a matrice.

remmatrice($matrice_1$, $matrice_2$, ...) [REMove MATRICE] removes each of the $matrice_i$ from MACSYMA. This allows the space taken up by these matrices to be reclaimed. remmatrice(all) will remove all the matrices present in the current MACSYMA. remmatrice returns a list of the matrices actually removed as its value.

matricep(e) [MATRICE Predicate] returns true if e is a matrice and false otherwise.

matriceinfo(M) produces a list describing the matrice M or signals an error if its argument is not a matrice. The form of the list returned is

[mathematics, type, [row dimension, column dimension], half bandwidth, form of the elements]

where the next to the last entry (half bandwidth) is present only if the type refers to a banded matrix.

gmat(M, i, j) [Get MATrice element] returns the value of the matrice element M_{ij}. Note that the rows and columns of the matrice are assumed to be numbered starting from one.

pmat(M, i, j, $value$) [Put into MATrice element] stores the given $value$ into the matrice element M_{ij}. Note that the rows and columns of the matrice are assumed to be numbered starting from one.

conjugate(e) returns the complex conjugate of e. All variables in e are assumed to be real. The effect of this function is to substitute -%I for %I throughout e.

zerop(e) [ZERO Predicate] returns true if e is a simple syntactic zero (e.g., 0, rat(0.0), intopois(0.0b0), etc.) and false otherwise.

lcm(x, y) computes a least common multiple of x and y. This is implemented simply as $(x * y)/\mathsf{gcd}(x, y)$ (almost).

mtrace(A) computes the trace of A where A is either a matrix, a two-dimensional array or a matrice. mtrace calls diagonal and does not check whether A is square.

norm(A, p) calculates the p-norm of the matrix or matrice A. p may be either 1 (1-norm), INF or INFINITY (infinity-norm), or EUCLIDEAN, F, FROBENIUS or SCHUR (F-norm $= \sqrt{\mathrm{trace}(A^H A)}$).

identity(n {, A}) creates an $n \times n$ identity matrix/matrice. If a second argument is not provided then identity will return the specified identity matrix as its value. Otherwise, identity will create the specified identity matrice and attach it to the symbol A, returning as its value the description of the created matrice provided by matriceinfo.

perm(n, i, j {, A}) creates the $n \times n$ permutation matrix/matrice P_{ij}. If a fourth argument is not provided then perm will return the specified permutation matrix as its value. Otherwise, perm will create the specified permutation matrice and attach it to the symbol A, returning as its value the description of the created matrice provided by matriceinfo.

companion(p, x {, C}) generates the generalized hypercompanion matrix associated with the polynomial p when p is taken as a function of x. If p is factored, then this matrix will be a direct sum of the hypercompanion matrices associated with each unique factor. The resultant matrix has the property that its characteristic polynomial is p. If a third argument is provided to companion then a matrice will be produced (attached to C) instead of a matrix.

hermitian(A {, B}) takes the Hermitian (complex conjugate transpose) of the matrix or matrice A. If A is a matrix then no second argument should be specified and hermitian will return as its value the matrix that is the Hermitian of A. Otherwise, if A is a matrice then a matrice representing the Hermitian of A will be attached to the symbol B and hermitian will return as its value the description of B produced by matriceinfo.

mtranspose(A {, B}) [Matrix/Matrice TRANSPOSE] takes the transpose of the matrix or matrice A. If A is a matrix then no second argument should be specified and mtranspose will return as its value the matrix that is the transpose of A. Otherwise, if A is a matrice then a matrice representing the transpose of A will be attached to the symbol B and mtranspose will return as its value the description of B produced by matriceinfo.

gcdn(A) returns the greatest common divisor of the elements in the list, matrix or matrice A.

derat(A, B {, $NOGCDS$}) converts the matrice A of rational entries into a matrice of polynomial entries by multiplying each element in A by a common multiple of the denominators of all the matrice entries (call this quantity d). The matrice resulting from this transformation will be attached to the symbol B. d will be the least common multiple unless the optional flag $NOGCDS$ is provided, in which case d will be the product of the denominators of the non-redundant matrice entries. derat will globally associate the divisor d with the matrice B [so that mdivisor(B) will yield d]. Many, but not all functions in the Matrice package will properly deal with matrice divisors. In particular, those functions which produce a non-matrice quantity from a matrice (ice2ix, ice2list, diagonal, mtrace, norm, mcharpoly, meigenvalues) will ignore matrice divisors currently. In these cases, the correct result is obtained by dividing the quantity by the divisor (or the absolute value of the divisor for norms) of the matrice from which the quantity was computed. The exception to this rule occurs when polynomials (characteristic or residual) are calculated, in which case a proper result is obtained by substituting for the polynomial variable, the variable multiplied by the divisor of the matrice from which the polynomial was computed [e.g., subst(mdivisor(A)$*\lambda$, λ, mcharpoly(A, λ))].

In some cases, these corrections can be performed automatically (see USEMDIVISOR). derat will return d as its value.

mdivisor(*matrice*) [Matrice DIVISOR] returns the common divisor of the elements of the *matrice* determined previously by derat (q.v.) or false if no divisor has been determined previously.

mdivmod(*matrice* {, *divisor*}) [Matrice DIVisor MODification] modifies the divisor associated with the *matrice*. If no second argument is provided then the divisor associated with the *matrice* will be eliminated and the deleted divisor (or false if the *matrice* had no divisor) will be returned as mdivmod's value. Otherwise, *divisor* will become the divisor associated with the *matrice*, replacing the old value if one exists. In this case, mdivmod will return as its value the former divisor or true if the *matrice* originally had no divisor associated with it.

hessenberg(A, H {, S {, *SINV*}}) transforms the square matrice A into upper Hessenberg form using similarity transformations. The resultant transformed matrice is attached to the symbol H, and the appropriate transforming matrices are attached to the symbols S and *SINV* if one or both of these is provided as an argument. (Here, $H = SINV\,A\,S$ where *SINV* is the inverse of S.) As a special case, Hermitian matrices are transformed via a mostly symmetry preserving transformation into general tridiagonal form. For this special case, the first argument to hessenberg may also be a list like [A, D] where A is the Hermitian matrice and D is a diagonal matrice with real, positive diagonal elements. In this situation, the similarity transformation will be applied to $D^{-1}A$, corresponding to the eigenproblem $A\mathbf{x} = \lambda D\mathbf{x}$. The value returned by hessenberg is a list of the descriptions of the newly created matrices (in argument order) produced by matriceinfo.

mcharpoly(A, *lambda*) computes the characteristic polynomial of the matrice A in terms of the variable *lambda*. The calculation will take advantage of matrices in certain special forms (diagonal, tridiagonal, triangular, upper Hessenberg).

meigenvalues(A) or meigenvalues(p, *lambda*) [Matrice EIGENVALUES] computes the eigenvalues of the matrice A or of the characteristic polynomial p written in terms of the variable *lambda*. meigenvalues returns as its value a list containing two sublists. The first sublist specifies the eigenvalues determined, while the second lists their corresponding multiplicities. If the characteristic polynomial of A (or p) is not completely factorable then the list of multiplicities will be followed by the residual polynomial that results from removing the solvable factors from the characteristic polynomial. The residual polynomial will be expressed in terms of the variable MCHARPOLY_VAR (q.v.) if meigenvalues is called with a matrice as the argument. meigenvalues defaultly uses solve to find the roots of the characteristic polynomial and so deficiencies in solve can affect the results returned by this function (see also EIGENMETHOD and NUMEIGS).

Appendix F

Maple Implementation of Bareiss Style Algorithms

In this appendix is presented the actual Maple coding for the Det function that was used in some of the timings detailed in Appendices B and C. Det performs a Bareiss style Gaussian elimination to compute the determinant of a matrix. The actual algorithm selected will depend on the setting of the global variable METHOD:

METHOD	Algorithm
bareiss1	1-step fraction-free Bareiss Gaussian elimination
bareiss2	2-step fraction-free Bareiss Gaussian elimination
sparse	1-step fraction-free sparse Gaussian elimination

These algorithms are described in detail in Section 2.1.2.

A second function, linearsolve, which solves the linear system of equations $AX = B$, has also been included since it shares much of its code with Det. Both of these functions use an indexing scheme to record row permutations in the matrix undergoing elimination, rather than physically swapping rows. This stratagem results in lessening the computation time required to perform the reduction and is used throughout the Matrice package (see Appendix E) as well. A MACSYMA version of Det and linearsolve was also created and this Maple version is a direct translation of that code.

F.1 Code

```
METHOD:= ''bareiss1'':

# linearsolve(A, B, 'X') solves the linear system A X = B where A is an
# n x n matrix and X and B are n x 1 matrices.  A and B are assumed to hold
# polynomial entries.  If the METHOD is 'bareiss1' (the default), a single-
# step fraction-free Gaussian elimination method is used to triangularize A
# and then a division minimizing back substitution is performed to determine
```

```
# X.  If the METHOD is 'bareiss2', a two-step fraction-free Gaussian
# elimination method is used to triangularize A.  Otherwise, A will be
# triangularized by a sparse fraction-free Gaussian elimination method.
# linearsolve will return as its value a list containing the matrix of the
# numerators of the solution vectors X and their common denominator, which
# will be the determinant of A.
linearsolve:= proc(A, B, X)
   local n, l, C, j, i;
   if not type(A, 'matrix') or not type(B, 'matrix') or not type(X, 'name')
      then
      ERROR(`*** linearsolve: Incorrect argument types ***`)
   fi;
   n:= linalg[rowdim](A);
   if linalg[coldim](A) <> n then
      ERROR(`*** linearsolve: A is not square ***`)
   fi;
   if linalg[rowdim](B) <> n then
      ERROR(`*** linearsolve: B is incompatible with A ***`)
   fi;
   l:= linalg[coldim](B);
   C:= convert(A, array, 1..n, 1..n+l);
   for j from 1 to l do
      for i from 1 to n do
         C[i, n+j]:= B[i, j]
      od
   od;
   linsolve_det(X, C, n, l)
   end:

# Det(A) computes the determinant of the n x n matrix A which is assumed to
# hold polynomial entries.  If the METHOD is 'bareiss1' (the default), a
# single-step fraction-free Gaussian elimination method is used to
# triangularize A from whence the determinant can just be picked off as the
# lower righthand element.  If the METHOD is 'bareiss2', a two-step
# fraction-free Gaussian elimination method is used to triangularize A.
# Otherwise, A will be triangularized by a sparse fraction-free Gaussian
# elimination method with the determinant being the product of the diagonal
# elements less some excess accumulated factors.
Det:= proc(A)
   local n, C, j, i;
   if not type(A, 'matrix') then
      ERROR(`*** Det: Incorrect argument type ***`)
```

```
   fi;
   n:= linalg[rowdim](A);
   if linalg[coldim](A) <> n then
      ERROR(`*** Det: A is not square ***`)
   fi;
   C:= copy(A);
   linsolve_det(0, C, n, 0)
   end:

# Consolidated function.
linsolve_det:= proc(X, C, n, l)
   local i, indx, t1, sign, d, k, p, ii, t, kk, j, f, t2, kk_1, cof0, cof1,
         cof2, nn;
   if printlevel > 1 then
      lprint(`The method used is`, METHOD)
   fi;
   indx:= array(1..n);
   for i from 1 to n do
      indx[i]:= i
   od;
   # Triangularize A
   t1:= time();
   sign:= 1;
   d:= 1;
   if METHOD <> 'bareiss2' then   # 1-step methods
      for k from 1 to n-1 do
         # Choose the pivot
         p:= k;
         for i from k+1 to n do
            ii:= indx[i];
            if C[ii, k] <> 0 and better_pivot(C[ii, k], C[indx[p], k]) then
               p:= i
            fi
         od;
         if C[indx[p], k] = 0 then
            if l > 0 then
               ERROR(`*** linearsolve: A is singular [1] ***`)
            else   # Det
               RETURN(0)
            fi
         elif p <> k then   # swap
            if printlevel > 1 then
```

```
      lprint(`[1] Swapping rows`, k, `and`, p)
   fi;
   sign:= -sign;
   t:= indx[k];
   indx[k]:= indx[p];
   indx[p]:= t
fi;
# Eliminate
kk:= indx[k];
if METHOD = 'bareiss1' then
   for i from k+1 to n do
      ii:= indx[i];
      if C[ii, k] <> 0 then
         for j from k+1 to n+l do
            #C[ii, j]:= (C[kk, k]*C[ii, j] - C[ii, k]*C[kk, j]) / d
            divide(C[kk, k]*C[ii, j] - C[ii, k]*C[kk, j], d,
                   evaln(C[ii, j]))
         od;
         C[ii, k]:= 0
      else
         #f:= C[kk, k] / d;
         for j from k+1 to n+l do
            #C[ii, j]:= f*C[ii, j]
            divide(C[kk, k]*C[ii, j], d, evaln(C[ii, j]))
         od
      fi
   od;
   d:= C[kk, k]
else   # sparse method
   for i from k+1 to n do
      ii:= indx[i];
      if C[ii, k] <> 0 then
         d:= d*C[kk, k];
         for j from k+1 to n+l do
            C[ii, j]:= C[kk, k]*C[ii, j] - C[ii, k]*C[kk, j]
         od;
         C[ii, k]:= 0
      fi
   od
fi;
t2:= time();
if printlevel > 1 then
```

```
         lprint(`Elimination: k =`, k, `[`, t2 - t1, `sec ]`)
      fi;
      t1:= t2
   od
else   # 2-step method
   for k from 2 by 2 to n do
      # Choose the pivot
      p:= k-1;
      for i from k to n do
         ii:= indx[i];
         if C[ii, k-1] <> 0 and better_pivot(C[ii, k-1], C[indx[p], k-1])
               then
            p:= i
         fi
      od;
      if C[indx[p], k-1] = 0 then
         if l > 0 then
            ERROR(`*** linearsolve: A is singular [2] ***`)
         else   # Det
            RETURN(0)
         fi
      elif p <> k-1 then   # swap
         if printlevel > 1 then
            lprint(`[2] Swapping rows`, k-1, `and`, p)
         fi;
         sign:= -sign;
         t:= indx[k-1];
         indx[k-1]:= indx[p];
         indx[p]:= t
      fi;
      # Compute the constant cofactor
      kk_1:= indx[k-1];
      kk:= indx[k];
      if k < n then
         p:= k-1;
         cof0:= 0;
         for i from k to n while cof0 = 0 do
            p:= p+1;
            ii:= indx[i];
            cof0:= C[kk_1, k-1]*C[ii, k] - C[kk_1, k]*C[ii, k-1]
         od;
         if cof0 = 0 then
```

```
      if l > 0 then
         ERROR(`*** linearsolve: A is singular [3] ***`)
      else   # Det
         RETURN(0)
      fi
   elif p <> k then   # swap
      if printlevel > 1 then
         lprint(`[3] Swapping rows`, k, `and`, p)
      fi;
      sign:= -sign;
      t:= indx[k];
      indx[k]:= indx[p];
      indx[p]:= t;
      kk:= indx[k]
   fi;
   #cof0:= cof0 / d
   divide(cof0, d, 'cof0')
fi;
# Eliminate
for i from k+1 to n do
   ii:= indx[i];
   #cof1:= (C[kk_1, k]*C[ii, k-1] - C[kk_1, k-1]*C[ii, k]) / d;
   #cof2:= (C[kk, k-1]*C[ii, k] - C[kk, k]*C[ii, k-1]) / d;
   divide(C[kk_1, k]*C[ii, k-1] - C[kk_1, k-1]*C[ii, k], d,
          'cof1');
   divide(C[kk, k-1]*C[ii, k] - C[kk, k]*C[ii, k-1], d, 'cof2');
   for j from k+1 to n+1 do
      #C[ii, j]:= (C[ii, j]*cof0 + C[kk, j]*cof1 + C[kk_1, j]*cof2)
      #           / d
      divide(C[ii, j]*cof0 + C[kk, j]*cof1 + C[kk_1, j]*cof2, d,
             evaln(C[ii, j]))
   od;
   C[ii, k-1]:= 0;
   C[ii, k]:= 0
od;
for j from k to n+1 do
   #C[kk, j]:= (C[kk_1, k-1]*C[kk, j] - C[kk_1, j]*C[kk, k-1]) / d
   divide(C[kk_1, k-1]*C[kk, j] - C[kk_1, j]*C[kk, k-1], d,
          evaln(C[kk, j]))
od;
C[kk, k-1]:= 0;
d:= C[kk, k];
```

```
      t2:= time();
      if printlevel > 1 then
         lprint(`Elimination: k =`, k, `[`, t2 - t1, `sec ]`)
      fi;
      t1:= t2
   od
fi;
# Compute the determinant
nn:= indx[n];
if METHOD = 'bareiss1' or METHOD = 'bareiss2' then
   if sign = -1 then
      for j from n to n+l do
         C[nn, j]:= -C[nn, j]
      od
   fi;
   d:= C[nn, n]
else   # sparse method
   d:= 1/d;
   for i from 1 to n do
      ii:= indx[i];
      d:= d*C[ii, i]
   od;
   d:= normal(sign*d);
   if l > 0 and d <> 0 then   # linearsolve
      #f:= C[nn, n]/d;
      divide(C[nn, n], d, 'f');
      C[nn, n]:= d;
      for j from n+1 to n+l do
         #C[nn, j]:= C[nn, j]/f
         divide(C[nn, j], f, evaln(C[nn, j]))
      od
   fi
fi;
if l = 0 then   # Det
   d
else   # linearsolve
   if d = 0 then
      ERROR(`*** linearsolve: A is singular [4] ***`)
   fi;
   # Back substitute
   for i from n-1 by -1 to 1 do
      ii:= indx[i];
```

```
         for j from n+1 to n+l do
            #C[ii, j]:= (d*C[ii, j] - sum('C[ii, k]*C[indx[k], j]',
            #                            'k' = i+1..n)) / C[ii, i]
            divide(d*C[ii, j] - sum('C[ii, k]*C[indx[k], j]', 'k' = i+1..n),
                   C[ii, i], evaln(C[ii, j]))
         od;
         t2:= time();
         if printlevel > 1 then
            lprint(`Back substitution: i =`, i, `[`, t2 - t1, `sec ]`)
         fi;
         t1:= t2
      od;
      X:= array(1..n, 1..l);
      for i from 1 to n do
         ii:= indx[i];
         for j from 1 to l do
            X[i, j]:= C[ii, n+j]
         od
      od;
      [X, d]
   fi
   end:

# Is x a better pivot than y?
better_pivot:= proc(x, y)
   if y = 0 then
      true
   elif type(y, 'integer') then
      if type(x, 'integer') then
         evalb(abs(x) < abs(y))
      else
         false
      fi
   else
      if type(x, 'integer') then
         true
      else
         evalb(length(x) < length(y))
      fi
   fi
   end:
```

References

[Akr88] Alkiviadis G. Akritas, "A New Method for Computing Polynomial Greatest Common Divisors and Polynomial Remainder Sequences", *Numerische Mathematik*, Volume 52, 1988, 119–127.

[Akr89] Alkiviadis G. Akritas, *Elements of Computer Algebra with Applications*, John Wiley & Sons, 1989.

[Bar66] Erwin H. Bareiss, *Multistep Integer-Preserving Gaussian Elimination*, ANL-7213, Argonne National Laboratory, Argonne, Illinois, May 1966.

[Bar68] Erwin H. Bareiss, "Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination", *Mathematics of Computation*, Volume 22, Number 103, July 1968, 565–578.

[Bel70] Richard Bellman, *Introduction to Matrix Analysis*, Second Edition, McGraw-Hill Book Company, 1970.

[Bog83] Richard Bogen, Jeffrey Golden, Michael Genesereth, Richard Pavelle, Michael Wester, Richard Fateman and Alexander Doohovskoy, *MACSYMA Reference Manual*, 2 volumes, The Mathlab Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, Version Ten—January 1983.

[Bra75] Gerald L. Bradley, *A Primer of Linear Algebra*, Prentice-Hall, 1975.

[Cam82] J. A. Campbell and F. Gardin, "Transformation of an Intractable Problem into a Tractable Problem: Evaluation of a Determinant in Several Variables", *Computer Algebra: EUROCAM '82, European Computer Algebra Conference, Marseille, France, April 1982*, Springer-Verlag, 1982, 196–203.

[Cha88] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Michael B. Monagan and Stephen M. Watt, *MAPLE Reference Manual: Fifth Edition*, WATCOM Publications Limited, Waterloo, Ontario, Canada, March 1988.

[Coh78] Daniel I. A. Cohen, *Basic Techniques of Combinatorial Theory*, John Wiley & Sons, 1978.

[Com88] Computer Aided Mathematics Group, *MACSYMA Reference Manual: Version 13*, Symbolics Inc., November 1988.

[Cul72] Charles G. Cullen, *Matrices and Linear Transformations*, Second Edition, Addison-Wesley Publishing Company, 1972.

[Fat79] Richard J. Fateman, "MACSYMA's General Simplifier: Philosophy and Operation", *Proceedings of the 1979 MACSYMA Users' Conference*, Washington, DC, June 1979, 563–582.

[Fla63] Harley Flanders, *Differential Forms with Applications to the Physical Sciences*, Academic Press, 1963.

[Fod83] John K. Foderaro, Keith L. Sklower and Kevin Layer, *The FRANZ LISP Manual*, University of California at Berkeley, Berkeley, California, June 1983.

[For67] George E. Forsythe and Cleve B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967.

[For77] George E. Forsythe, Michael A. Malcolm and Cleve B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1977.

[Fox65] L. Fox, *An Introduction to Numerical Linear Algebra*, Oxford University Press, 1965.

[Gan60] F. R. Gantmacher, *The Theory of Matrices*, 2 volumes, Chelsea Publishing Company, 1960.

[Gar75] Burton S. Garbow and Jack J. Dongarra, *Path Chart and Documentation for the EISPACK Package of Matrix Eigensystem Routines*, TM-250, Argonne National Laboratory, Argonne, Illinois, August 1975.

[Ged82] Keith O. Geddes, *Algebraic Algorithms for Symbolic Computation*, Department of Computer Science, University of Waterloo, Waterloo, Canada, January 1982 (third draft).

[Gen73] W. Morven Gentleman, "Least Squares Computations by Givens Transformations Without Square Roots", *Journal of the Institute of Mathematics and its Applications*, Volume 12, 1973, 329–336.

[Gen74] W. M. Gentleman and S. C. Johnson, "The Evaluation of Determinants by Expansion by Minors and the General Problem of Substitution", *Mathematics of Computation*, Volume 28, Number 126, April 1974, 543–548.

[Gen76] W. M. Gentleman and S. C. Johnson, "Analysis of Algorithms, A Case Study: Determinants of Matrices With Polynomial Entries", *ACM Transactions on Mathematical Software*, Volume 2, Number 3, September 1976, 232–241.

[Gol76] G. H. Golub and J. H. Wilkinson, "Ill-Conditioned Eigensystems and the Computation of the Jordan Canonical Form", *SIAM Review*, Volume 18, Number 4, October 1976, 578–619.

[Gol83] Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, John Hopkins University Press, 1983.

[Gre88] Brent Gregory and Erich Kaltofen, "Analysis of the Binary Complexity of Asymptotically Fast Algorithms for Linear System Solving", *SIGSAM Bulletin*, Volume 22, Number 2, April 1988, 41–49.

[Gro87] Eric Grosse and Cleve Moler, "Underflow Can Hurt", *SIAM News*, Volume 20, Number 6, November 1987, p. 1.

[Ham74] Sven Hammarling, "A Note on Modifications to the Givens Plane Rotation", *Journal of the Institute of Mathematics and its Applications*, Volume 13, 1974, 215–218.

[Hea87] Anthony C. Hearn, *REDUCE User's Manual: Version 3.3*, Rand Publication CP78, The Rand Corporation, Santa Monica, California, July 1987.

[Hen64] Peter Henrici, *Elements of Numerical Analysis*, John Wiley & Sons, 1964.

[Hör86] Lars Hörnfeldt, *STENSOR Reference Manual*, Institute of Theoretical Physics, University of Stockholm, Stockholm, Sweden, July 1986.

[How69] Jo Ann Howell and Robert T. Gregory, "An Algorithm for Solving Linear Algebraic Equations Using Residue Arithmetic I-II", *BIT*, Volume 9, Number 3–4, 1969, 200–224 and 324–337.

[Joh81] Lee W. Johnson and R. Dean Riess, *Introduction to Linear Algebra*, Addison-Wesley Publishing Company, 1981.

[Knu81] Donald E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Second Edition, Addison-Wesley Publishing Company, 1981.

[McC73] Michael T. McClellan, "The Exact Solution of Systems of Linear Equations with Polynomial Coefficients", *Journal of the Association for Computing Machinery*, Volume 20, Number 4, October 1973, 563–588.

[McC77] Michael T. McClellan, "The Exact Solution of Linear Equations With Rational Function Coefficients", *ACM Transactions on Mathematical Software*, Volume 3, Number 1, March 1977, 1–25.

[Mol73] Cleve B. Moler, *Matrix Eigenvalues and Least Squares Computations*, Department of Mathematics, University of New Mexico, February 1973 (draft).

[Mol76] C. B. Moler and C. F. Van Loan, *Nineteen Ways to Compute the Exponential of a Matrix*, TR 76-283, Department of Computer Science, Cornell University, Ithaca, New York, July 1976.

[Mol77] C. B. Moler and J. H. Wilkinson, *Fast Givens Methods for Matrix Eigenvalue Problems*, 1977 (draft).

[Mol80] Cleve Moler, *MATLAB—An Interactive Matrix Laboratory*, Technical Report No. 369, Department of Mathematics and Statistics, University of New Mexico, February 1980.

[Sch73] H. R. Schwarz, *Numerical Analysis of Symmetric Matrices*, Prentice-Hall, 1973.

[Smi81] J. Smit, "A Cancellation Free Algorithm, with Factoring Capabilities, for the Efficient Solution of Large Sparse Sets of Equations", *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, Snowbird, Utah, August 1981, 146–154.

[Sof83] The Soft Warehouse, *Microsoft muMATH Symbolic Mathematics Package*, 1983.

[Sof86] "The muMATHematician: Improved Routines for Determinants, Matrix Inverses, Eigenvalues and Eigenvectors", *Newsletter 15*, Soft Warehouse Inc., Honolulu, Hawaii, September 1986, 2–4.

[Sta70] Harold M. Stark, *An Introduction to Number Theory*, The MIT Press, 1970.

[Ste87] Guy L. Steele Jr., Raphael Finkel, Donald Woods, Geoff Goodfellow and Mark Crispin, "Glossary of Jargon", 1987.

[Sto72] A. P. Stone, "Induced Transformations on Exterior Product Spaces", *Tensor, N.S.*, Volume 23, 1972, 147–150.

[Str88] Gilbert Strang, *Linear Algebra and its Applications*, Third Edition, Harcourt Brace Jovanovich, 1988.

[Str74] A. H. Stroud, *Numerical Quadrature and Solution of Ordinary Differential Equations*, Springer-Verlag, 1974.

[Sut88] Robert S. Sutor (ed.), *The Scratchpad II Computer Algebra System Interactive Environment Users Guide*, Computer Algebra Group, Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, New York, September 1988 (draft version 1.3).

[Tou79] Evelyne Tournier, "An Algebraic Form of a Solution of a System of Linear Differential Equations with Constant Coefficients", *Symbolic and Algebraic Computation: EUROSAM '79, An International Symposium on Symbolic and Algebraic Manipulation, Marseille, France, June 1979*, Springer-Verlag, 1979, 153–163.

[Wan76] Paul S. Wang and Tadatoshi Minamikawa, "Taking Advantage of Zero Entries in the Exact Inverse of Sparse Matrices", *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, Yorktown Heights, New York, August 1976*, 346–350.

[Wan77a] Paul S. Wang, "On the Expansion of Sparse Symbolic Determinants", *Proceedings of the 10th Hawaii International Conference on System Sciences, January 1977.*

[Wan77b] Paul S. Wang, "Matrix Computations in MACSYMA", *Proceedings of the 1977 MACSYMA Users' Conference*, University of California, Berkeley, California, July 1977, 435–446.

[Wes91] Michael Wester, "Expression Swell Analysis of the Computation of Matrix Characteristic Polynomials", *Transactions of the Eighth Army Conference on Applied Mathematics and Computing*, Report No. 91-1, U.S. Army Research Office, February 1991.

[Wil65] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, 1965.

[Wol88] Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley Publishing Company, 1988.