INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

High Precision Numerical Computation, The Determinant Case

by

Colin D Stewart

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1998

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy

5/20/98
Date

Victor Pan
Professor Victor Pan, Chair of Examining Committee

5/21/98
Date

Stanley Habib
Professor Stanley Habib, Executive Officer

Professor Charles Giardina (CUNY)

Professor Xiang ZhiGang (CUNY)

Professor Carl Bredlau (Montclair State University)

The City University of New York

ABSTRACT

High Precision Numerical Computation. The Determinant Case

by

Colin D Stewart

Advisor: Professor Victor Pan

In recent years the problem of accurately determining the sign of a matrix determinant has attracted a great deal of attention. The difficulty in this problem lies with the characteristic presence of truncation or rounding errors in floating point machine arithmetic. Various approaches have been suggested for getting around this problem, notably focusing on alternative numerical algorithms or exact arithmetic methods. We take a fresh approach, really a mix of the previously proposed ideas, featuring a new algorithmic step called "certification of the sign of the determinant" whereby an initial computation may be done in fast floating point using a standard numerical algorithm. If the certification test succeeds, then the result is said to be certified and the sign of the result is guaranteed to be correct. If the test fails then, and only then, need the computation be done using exact, albeit slower, arithmetic. Since the signs tend to be ambiguous only in small regions of the application problem space, the certification method is likely to prove competitively efficient in these applications: In effect the certification method applies the costly power of exact arithmetic selectively.

This dissertation focuses on the key problem areas of the certification method:

1. Finding a reliable method for determining the accuracy of a floating point computation. The 'correct' result is never present, errors must be deduced from the residue of the computation itself. We find that a measure of the proximity of the input to a singular matrix is available in the residue of LU decomposition with full pivoting and that this measure is a reliable source of certification.

2 Finding a parameterizable exact arithmetic alternative. The effectiveness of the certification process would be at least compromised if the exact back-up computation could not use information obtained from the primary calculation. The author's Modulus Vector formulation of classical modular arithmetic, developed in this dissertation, is just such a flexible exact arithmetic alternative.

3 Effectively combining the two major subtasks, certification and modular recomputation, into a workable algorithm.

Much of the research reported in this dissertation, and its findings, are based on numerical experiments.

Acknowlegements

This Dissertation is dedicated to my wife, Sharon. She has suffered the full measure of the distress known to few. PhD widowhood. Through it all, and it has been long, she has supported me with unfailing cheerfulness and understanding. Thanks love.

There can never be sufficient recognition for the services of your thesis advisor. Professor Victor Pan suggested the topic, provided the central idea, and continually enriched the research with his vast knowledge of the literature in algorithm studies. Moreover, by encouraging joint publication, Professor Pan gives his students a priceless scholarly boost, we go forth with Pan Number 0! That's some boost. Here's my chance to acknowledge my debt to Professor Stanley Habib, EO of our Computer Science Program, who has supported my doctoral quest for many years through thick and thin, and there's been some thin, which daunted him not at all. I must thank my colleague, and fellow student, Steve Yu, for his assistance, and for his friendship. The reader will find Steve's contributions in this dissertation. I must also thank the Dissertation Defense review committee, whose interest and contributions are highly valued, but one of them must be singled out for special attention. Professor Charles Giardina got me started at CUNY, so he's to blame for it all!

I am most grateful for the assistance of the research staff at Sparta Public Library. It is rewarding, in many ways, to have such a professional resource at local hand. No book title, no journal citation escaped their grasp. I trust our relationship will continue.

Finally let it be said that the doctoral quest is no small matter. I have benefited immeasurably in my odyssey by the professional counsel of Dr Sondra Tuckfelt, psychiatrist and author. Beyond her reassuring insights into an occasionally terrified psyche, she offered the companionship of understanding my intellectual self.

Table of Contents

Chapter 1 - Introduction 1
1. Errors in Computation: Motivation for this Study 1
2. Certification of Computations 5
3. Organization of the Thesis 9

Chapter 2 - Determinants and Geometry 13
1. Definition and Properties 13
2. Geometry 21
3. Extension to R^n 27
4. Sensitivity to Error 31
5. Size of Determinants 34

Chapter 3 - Floating Point Arithmetic and Algorithms 40
1. Computer Arithmetic 41
2. LU Algorithm 45
3. Estimating the Errors in the LU Algorithm 53
4. Stability of the LU Algorithm, Pivoting 62
5. Computing and Estimating the Inverse Matrix 69

Chapter 4 - Modular Arithmetic 76
1. Classes, Rings of Classes, Arithmetic, Modular Vectors 77
2. Elementary Algorithms 83
3. The Chinese Remainder Theorem 88
4. Projections in the Vector Ring, Knockouts 94
5. LU Decomposition Algorithm using Modular Arithmetic 98

Chapter 5 - Data Generation and Experimental Studies 102
1. Data Generation and Sources 102
2. Data Reduction 106
3. Certifying Determinant Computations 107
4. A Technical Engineering Note 120

Chapter 6 - Summary and Directions for Further Research 123
1. Discussion of our Results 123
2. Extension to other Numerical Algorithms 125
3. Research Topics in Modular Arithmetic 128

References 139

List of Figures

Figure 2.1 - EBM in C++ . . . 17
Figure 2.2 - Convex Set in 2-D . . . 22
Figure 2.3 - Convex Set with Interior Point . . . 25
Figure 2.4 - Resolution Example . . . 26
Figure 2.5 - Size of Determinants . . . 38
Figure 3.1– LU^- Algorithm . . . 52
Figure 3.2– LU^- Algorithm . . . 52
Figure 3.3 - LU Data Flow . . . 55
Figure 3.4 - Typical Error Matrix . . . 61
Figure 3.5–LU^- Algorithm with Full Pivoting . . . 66
Figure 3.6 - Inverse Matrix Algorithm . . . 71
Figure 4.1 - Integer Euclidean Division . . . 85
Figure 4.2 - Euclidean Division in Floating Point . . . 85
Figure 4.3 - Extended GCD Algorithm . . . 88
Figure 4.4 - Chinese Remainder-1 . . . 91
Figure 4.5 - Chinese Remainder-2 . . . 91
Figure 4.6 - Chinese Remainder-3 . . . 92
Figure 4.7 - Modular LU Decomposition . . . 99
Figure 5.1 - Estimated vs Actual Error . . . 113
Figure 5.2 - Estimated vs Actual Error . . . 114
Figure 5.3 - Distance to Singular vs Actual Error . . . 116
Figure 5.4 - Algorithm 0 . . . 120

List of Tables

Table 2.1 -- Permutation Data 14
Table 3.1 -- Operation Counts for LU 53
Table 3.2 - Experiment 1: LU Error 58
Table 3.3 - Experiment 2: LU Error 58
Table 3.4 - Experiment 3: LU Error 60
Table 3.5 - Experiment 4: LU Error 60
Table 3.6 - Operation Count for Full Pivot LU 67
Table 3.7 - Experiment 5: Pivoting Study 68
Table 3.8 - Operation Count for Backsubstitution 72
Table 3.9 - Experiment 6: Estimate of Norm of the Inverse 74
Table 4.1- $\mathbf{M}^3 = \{17, 23, 29\}$ 90
Table 4.2- $\mathbf{M}^2 = \{17, 29\}$ 95
Table 4.3 - Effect of Knock-out 97
Table 5.1 - Reduced Data Record 107
Table 5.2 - Det Error, n=5, $\alpha \leq 1{,}000{,}000$ 111
Table 5.3 - Distance to Singular Matrix, n=5, $\alpha \leq 1{,}000{,}000$ 115
Table 5.4 - Hit inventory, n=5, $\alpha \leq 1{,}000{,}000$ 118
Table 5.5 - Hit inventory, n=7 $\alpha \leq 35{,}000$ 118
Table 5.6 - Hit inventory, n=4 $\alpha \leq 100{,}000{,}000$ 119

Chapter 1

Introduction

Error in computation was a major concern from the very beginning of the computer era. Goldstine [Gold72] reports that an early analysis of error in the gaussian elimination algorithm that was proposed for the ENIAC machine was so pessimistic that for a short time the entire project was thrown in doubt[1]. These errors arise from the fact that arithmetic computations undergo truncation or rounding in any automatic machine which must necessarily use a finite length accumulator, or register, to store intermediate results in a lengthy computation. Although computational errors were quickly shown to be much less catastrophic than Hotelling's original prediction, the problem has continued to be a central concern in all applications of automatic computation.

1. Errors in Computation: Motivation for this Study

Our Thesis concerns computational errors, particularly the effect of errors in determining the sign of the determinant of a matrix. The problem has a long history (see, for instance [Mu], [Ma], [Fox], [P88]) and has recently received major motivation due to its importance in applications to computational geometry. We cite, for instance, its central role in the computation of convex hulls and Voronoi diagrams, and in testing whether a

---

[1] [Gold72] page 290. In 1943 H Hotelling estimated the magnitude of error in an $n \times n$ gauss elimination to be $4^n \cdot \epsilon$. This would be a disaster for n much beyond 15 or 20, say. Fortunately this frightening estimate was soon corrected, see [GoN47], [Tu48].

family of line intervals have a common intersection. In these applications one needs a sign or a singularity test, that is, one needs to determine whether det(A) is less than, greater than, or equal to zero. In most of these applications the dimension of the square matrix, A, is small, rarely larger than 5. In these low dimensional applications the computations are typically done in double-precision. However, there is a class of optimization applications (for examples see [DLa94], [AF92], [FR92]) dealing with high-dimensional polyhedra where the matrices are typically larger, of dimension over several hundred. In these cases at least the input data can be small.

The influence of internal error on the validity of the *sign*(det(A)) test is easily appreciated. If the absolute value of the error is denoted as e_d and the correct value of the determinant is D, then the computed result is $D - e_d < \det(A) < D + e_d$. If det($A$) is anywhere near e_d in magnitude, then the sign of det(A) is in question. Note also that a *zero*(det(A)) test is essentially meaningless in the presence of any error whatever. Happily e_d grows at a lower order than det(A) in dimension of the matrix and input data size so, in general, $e_d < |\det(A)|$ when $|\det(A)|$ is large. And, also fortuitous, $|\det(A)|$ is large *most* of the time. Real trouble comes in the near singular cases, when $|\det(A)|$ is small. Sadly, the latter case is the most interesting in applications.

Computational errors may be appreciated in the more general view by considering the classical problem of solving a linear system $Ax = f$. Here f is the input to a solver and x is the output. The numerical algorithm which does the work never actually solves this

problem, but solves a related problem which can be written $(A - E)\cdot x' = f$. That is, the algorithm takes A and f as the input and returns x' as the output for the problem of solving the linear system A. In fact, the problem is transformed into $A - E$, E being a matrix of (hopefully small) elements arising from the computational rounding errors suffered by the algorithm. Now, a well known relationship connects the quantities here, namely x, x', A, and E: (see for instance [GolVL], Sec 2.5, pp24-28, [ORT], Sec. 2.1.1 et seq., pp32-35)

$$\frac{\| x - x' \|}{\| x \|} \leq \kappa(A)\cdot\frac{\| E \|}{\| A \|}$$

up to smaller order errors. Here the bars denote compatible vector and matrix norms and κ denotes the problem *condition number* of the matrix A. The essence of this formula is that the relative error of the output is equal to the condition number of the problem *multiplied* by the relative computational errors. The effect is that for poorly conditioned problems, that is, for ones where κ is very large, moderate computational errors may easily be inflated to yield useless output.

In both of these expressions of computational mischief the trouble can be traced to a phenomenon which we call *Catastrophic Subtraction*. This misbehavior is a consequence of the finite length of the arithmetic accumulator, which is, of course, a fixed artifact of the computing engine. During computation large operands are truncated, or rounded, to fit into the available size of the arithmetic accumulator. The most significant part (the left hand piece) of the operand is retained, the least significant

part (the right hand part) is lost. An error is committed when this happens - the right-most digit of the retained number is in error by a certain amount. The maximum error of this kind for a fixed machine precision is called the *unit roundoff* and also the *machine epsilon*, which we customarily denote using a Greek lower-case epsilon, ϵ. Normally the operand participates in a chain of operations and this truncation damage is repeated many times in the course of a computation. Thus errors are accumulated and in fact can be compounded in multiplications or divisions. Suppose it happens that two operands, mangled in this way, come together as operands in a subtraction (this is not uncommon in numerical computations), and suppose the two operands are almost equal (which is also rather common). Then the result will consist only of their least significant digits, where the errors live. The result is 'all error' and small but significant 'real' differences will be lost in this way. Under these conditions algorithms can behave unpredictably; and we say they exhibit *instability*.

There are several cures for these ills, suggested or actually used. One obvious one, which has been applied since the beginning of the computer era, is to increase the size of the arithmetic unit. This is expensive. Another one is to increase operand size in software, the *multi-precision* approach. This idea is elaborated in Knuth [Knu], and has been widely used in the exact arithmetic work of [FvnW93], [YapDu], [Yap]. Again, this approach is expensive, this time in the computational cost sense[2]. A significant problem

[2][Yap97] reports a computational slowdown by factors from 40-140, one case reported a factor of 10,000 slowdown, but, he says, "...careful fine-tuning eventually reduces these factors to a small constant factor (less than 10)". Those readers familiar

with this 'BigNum' approach involves the question of how much multi-precision does one need. It is difficult to tailor an algorithm so that the full battery of expensive multi-precision is applied only to computations that need it. A major theme in our thesis grapples with this question and we offer an effective solution to this problem. Other approaches toward exact high precision arithmetic are the so-called algebraic methods. p-adic ([MC], [Yun], [P92] and modular ring arithmetic (see [P97], and especially [BEPP97]). Although these methods may be viewed as a variety of multi-precision arithmetic, they strongly contrast with the 'BigNum' approaches in that they execute computations in *alternate arithmetics*, polynomial rings in the former case and modular rings in the latter. The p-adic work is now somewhat old, but the modular arithmetic work is still very active and is proving to be an effective road to the exact computation goal. We take this path.

2. Certification of Computations

We are now in a position to state the major idea of our Thesis. Rather than abandoning altogether the efficiencies of floating point arithmetic, with its sometimes catastrophic injection of error, we introduce the notion of *certifying*[3] the results of floating point computation. By this we mean that we *compute* an estimate of the error generated in the

---

with software will recognize the great danger lurking just below the surface of these 'BigNumber' packages.

[3]The idea originates with Pan [BPY98].

calculation. This estimate computation becomes part of the algorithm - in most cases the data needed to make that computation is already available in the standard numerical algorithms, the error estimate becomes just an additional output. For example, suppose we compute $\det(A)$ using an efficient floating point algorithm, then if e_d is also computed and output we are in a position to *certify the correctness* of the sign of the determinant by comparing $|\det(A)|$ with e_d. Then, as we saw above, if $e_d < |\det(A)|$ we sure that the *sign* of the computed $\det(A)$ is reliable. Otherwise the computation is not certified to be correct and we use an exact arithmetic method just for this case. Note that costly exact arithmetic is used only when needed. Moreover, the floating point results are still available and can be used to 'parameterize' the exact arithmetic computation, they can be used to bound the 'precision' required, for instance.

Let us state our argument more formally. Let $\mathbf{F}(x)$ and $\mathbf{E}(x)$ be algorithms taking x as input and which use floating point arithmetic. Let $\mathbf{A}(p,x)$ be an algorithm, possibly parameterized by p, taking x as input and computing the same result as $\mathbf{F}(x)$ but using exact arithmetic (possibly multi-precision). $\mathbf{F}(x)$ and $\mathbf{A}(p,x)$ may in fact be the *same* algorithm but with the floating point operations of $\mathbf{F}(x)$ being replaced with the corresponding exact arithmetic operations in $\mathbf{A}(p,x)$. For example, $\mathbf{F}(x)$ might be a $\det(x)$ computation based on LU matrix decomposition in floating point, then $\mathbf{A}(p,x)$ could be the same LU factorization procedure returning $\det(x)$ but with its arithmetic operations replaced by those over a modular ring. Now define a *certification predicate* $\Psi(s,t)$, an algorithm which takes two inputs and returns TRUE or FALSE according to

whether s is *correct with respect to* t. For instance, if $s = \det(x)$ and $t = e_d$, then we have $\Psi(\det(x), e_d) = e_d < \det(x)$. Now we may state our major thesis as

Algorithm 0: *If* $\Psi(\mathbf{F}(x), \mathbf{E}(x))$ *then* $\mathbf{F}(x)$ *else* $\mathbf{A}(\mathbf{F}(x), x)$.

There are several questions that can be asked concerning this idea, the skeptical reader has no doubt already formulated a number of them. We discuss a few:

Can $\mathbf{E}(x)$ *be computed*? -- The notion of calculating errors appears initially to be in the nature of a mathematical oxymoron. Indeed, if the error arising in a computation can be computed then its effects would appear to be easily undone. We seem to propose a new, revolutionary, and somewhat dubious exact arithmetic method. In fact rounding error estimation is a classic topic in numerical analysis ([CdB], [Fox], [FM67], [GolVL], [ORT], and [StB] represent a sampling of the extensive literature on this topic). These results provide upper bounds on the errors to be expected in a computation. However, the sharpest, most informative and accurate of these results provide *a posteriori* bounds, results which depend on the problem and the algorithm. These results must be computed and can be output as side-effects of the computation. Thus we seek the best available computable error bound in the sense of the least conservative, or sharpest estimates. This topic is part of our study.

Will Algorithm 0 improve on computing cost? -- If $\mathbf{E}(x)$ computes the best available upper bound on errors in executing $\mathbf{F}(x)$, in floating point, and if $\mathbf{F}(x)$ and $\mathbf{E}(x)$ together

are faster than the software algorithms of $\mathbf{A}(p,x)$, then clearly Algorithm 0 will be a more efficient scheme for computing a reliable result than using the exact arithmetic of $\mathbf{A}(p,x)$ alone. This point needs study and proof, which it will receive, but it can be seen immediately that only in the worst case, where the precision of $\mathbf{A}(p,x)$ is required in the vast majority of cases, will the claims for Algorithm 0 be doubtful. The software designation for $\mathbf{A}(p,x)$ needs some examination: Floating point arithmetic gets its popularity from the fact that is implemented in hardware, and we are well aware that software emulation of any kind of arithmetic is dismal as regards efficiency. An argument will be made below for considering hardware implementation of modular arithmetic, an event which would undermine the claims for Algorithm 0. We are reminded, however, that $\mathbf{A}(p,x)$ is parameterized, tailored for each problem, and that floating point, for all its speed as well as its susceptibility to error, is not.

Will Algorithm 0 be effective? Yes. But we must point out that its effectiveness for a given class of problems, e.g. computing determinants, depends on $\mathbf{E}(x)$. Also, Algorithm 0 depends on an $\mathbf{A}(p,x)$ being available for the class of problem considered. There are still fundamental algorithms which appear, to the author's knowledge, unavailable in modular arithmetic. This area abounds in topics for further research and we will revisit this topic.

Can $\mathbf{A}(p,x)$ *be efficiently parameterized*? – This is a crucial point for Algorithm 0, obviously. The answer is yes, for modular arithmetic. Exact arithmetic methods,

whether used bare or conditionally under a certification predicate, as in Algorithm 0, must be parameterizable. The issue between the methods turns greatly on how easily and/or efficiently this can be done.

3. Organization of the Thesis

This thesis is organized into six chapters. Chapter 2 is concerned with the relationship between Arithmetic and Geometry. The determinant computation arithmetizes many key geometric concepts, especially the notion of measure, or size, and orientation of geometrical objects. The algebraic model for this arithmetization is the vector space. After defining the determinant computation and describing its fundamental algebraic properties, the discussion turns to a description of geometric concepts in terms of determinants and the role they play in capturing geometry in the vector model. This description is initially set in the visually intuitive space of 2 dimensions, but is quickly generalized to R^n, and we derive a fundamental relationship between arithmetic and geometry, namely that the $|\det(A)|$ computes the *measure* of a geometric object described by a matrix and that the *orientation* of the object is algebraically encoded in $sign(\det(A))$. We also show the important connection between the idea of *resolution* of geometric properties and the *inverse* of the describing matrix. We turn next to the question of the sensitivity of the determinant computation to computational error, a central question of our study: The formulation of Algorithm 0 begins here. We close with a study of the size distribution of determinants. This factor plays a role in the

efficiency of the certification paradigm since, in general, the sign computation for large determinants are certified in floating point computations, only small determinants require exact arithmetic.

Chapter 3 begins with a brief introduction to machine arithmetic, including floating point. We introduce the notion of machine errors as loss of information and formulate a rule for quantifying this loss. We then introduce the LU matrix decomposition We focus on this algorithm because it is the most efficient one available for calculating determinants and it is exclusively algebraic, an important feature since the algorithm may therefore be used in the modular arithmetic domain with no significant change. Moreover, LU is also highly suited to computing the matrix inverse. We then turn to a study of the effects of round off error which arise in the floating point version of the LU class of algorithms. This study features experimental methods and measurements. We find by these methods that, surprisingly and happily, the actual errors are always less than the estimated loss calculations predict. We speculate that the reason lies on the bright side of Catastrophic Subtraction. Since we have found, in chapter 2, that the norm of the inverse is an important source of information for certification, we explore ways of calculating $\|A^{-1}\|$ using the results of LU without having to bear the full computational cost of evaluating A^{-1}. This investigation is also based on laboratory work, and we find that the 1-norm of the inverse matrix can be estimated reliably using the results of LU with full pivoting.

Chapter 4 is a tutorial on Modular Ring arithmetic. The Modular Ring is a finite set isomorphic with the classes of integers modulo an integer p. The arithmetic operations of addition, subtraction, multiplication and limited division are defined for elements of this ring, so that the essentially algebraic numerical algorithms, particularly the LU computation, may be implemented using modular arithmetic. Arithmetic operations in modular arithmetic suffer no roundoff or truncation error, hence modular arithmetic is exact. We introduce the concept of a vector space over modular rings of prime order. The arithmetic operations are defined in this ring of vectors, and this ring has many properties valuable in numerical computation service. We present the Lagrange formulation of the Chinese Remainder Theorem and study its floating point implementation and error characteristics. Finally we describe the Modular implementation of the LU algorithm, and show its effectiveness as the exact alternative computation of determinants. Chapter 4 completes our study and implementation of algorithms needed for the certified computation of determinants, i.e. Algorithm 0.

In chapter 5 we return to the laboratory for an experimental determination of the parameters needed to implement $\Psi(\mathbf{F}(x), \mathbf{E}(x))$. We begin with a description of our experimental methods, which includes an explanation of how we generate experimental data. We need to explore a very large data space containing all integer matrices of dimension less than 10, say. Clearly a severely selective discipline must be employed. We focus on the region in the data space where certification of the sign of the determinant begins to fail, that is, where the error in the computation begins to

approximate the determinant. We find that this occurs when the distance, $\hat{H}$, to the singular matrix is less than 100 times the unit roundoff, where a unit is ϵ, the machine precision. The correctness of this finding is verified by many measurements whose results are tabulated. Thus we conclude: $\Psi(\det_{f.p}(A), \hat{H}) = \hat{H} > 100 \cdot \epsilon$.

Chapter 6 summarizes the thesis and proposes open questions for further research. We discuss the potential for using run-time certification in other numerical algorithms and the use of modular arithmetic for achieving exact arithmetic in these computations. This field holds attractive open topics, and we will highlight several of these. We then comment on some interesting open problems in modular arithmetic. A rational extension of our modulus vector model is proposed which would open this exact arithmetic option to solver applications. Further extensions are necessary to open the door to QR applications. Hardware implementation holds promise to greatly speed up modular arithmetic.

Chapter 2

Determinants and Geometry

In this chapter we seek to 'arithmetize' geometry, that is we explore the connection between geometry questions and computational algebra. We show that the determinant of a matrix plays a central role in this connection. In section 1 we define determinants and discuss the elementary arithmetic properties of determinants. Section 2 explores the relationship between algebra and the geometry of 2-space through examples using the properties of determinants developed in section 1. Section 3 extends this investigation to the general case in R^n. Section 4 studies the sensitivity of determinants to errors arising in computation. Finally, the concern in section 5 has to do with the size of determinants, particularly with respect to the notion of a 'random' determinant.

1. Definition and Properties

Consider the square matrix A, of *dimension* n, with elements $a_{i,j}$, $1 \le i \le n$, $1 \le j \le n$,

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & s_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n} \end{bmatrix}$$

The *determinant* of A, written as $\det(A)$ is a function of this matrix computed as

$$\det(A) = \sum_{i=1}^{n!} \left(\epsilon[\pi_i(n)] \cdot \prod_{j=1}^{n} a_{j,\pi_{ij}} \right) \qquad (2.1)$$

where $\pi_i(n)$ is the i^{th} distinct *permutation* on the indices $<1, 2, \cdots, n>$ and π_{ij} is the j^{th} member of the i^{th} permutation. $\epsilon[\pi_i(n)]$ is the *parity* of the i^{th} permutation, which is +1 or -1 according to whether the number of inversions in the permutation is even or odd. An inversion occurs whenever an index in the permutation list is greater than one which follows. For instance, suppose $\pi_j(6) = <6\ 1\ 4\ 2\ 3\ 5>$, then the number of inversions in this list is 7 and $\epsilon[\pi_j(6)] = -1$.

As an example of the foregoing, consider the $3 \cdot 3$ matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 2 \\ 3 & 3 & 1 \end{bmatrix} \qquad (2.2)$$

with index permutation data displayed in the following table ...

j	$\pi_j(3)$	inversions	$\epsilon[\pi_j(3)]$
1	1 2 3	0	+1
2	1 3 2	1	-1
3	2 3 1	2	+1
4	2 1 3	1	-1
5	3 1 2	2	+1
6	3 2 1	3	-1

Table 2.1 – Permutation Data

Then the computation of this determinant goes as follows:

$$\begin{aligned}\det(A) &= 1\cdot-1\cdot1 - 1\cdot2\cdot3 - 2\cdot2\cdot3 - 2\cdot2\cdot1 - 3\cdot2\cdot3 - 3\cdot-1\cdot3 \\ &= -1 - 6 - 12 - 4 - 18 - 9 \\ &= 28\end{aligned}$$

The cost of computing the determinant by using equation (2.1) is $n!\cdot(n-1)$ multiplications and $(n-1)!$ additions. A more practical use of (2.1) is to verify the following properties of determinants (see, e.g. [Hoh], [CdB])

D.1 *If any column or row of A is a linear combination of any other rows or columns, respectively, then* $\det(A) = 0$.

D.2 *Interchanging any two columns or rows of A changes the sign of* $\det(A)$.

D.3 $\det(A^{t}) = \det(A)$

D.4 *If A and B are any two $n \times n$ matrices,* $\det(A\cdot B) = \det(A)\cdot\det(B)$

Another useful property of the determinant is derived in hopes of improving on the computational cost estimate for equation (2.1). Suppose we seek to factor (2.1) into groups. Since a given element of A occurs in $(n-1)!$ of the terms in equation (2.1) we would gain a computational advantage from such a factoring. Consider grouping all terms containing $a_{n,n}$. $a_{n,n}$ would always appear as the last factor and $\pi_i(n)$would appear in 'natural' order as the permutation $< \cdots, n>$, in the terms of this group. Now factoring out $a_{n,n}$, the group could be written as the single term $a_{n,n}\cdot a^{\cdot}_{n,n}$, where $a^{\cdot}_{n,n}$ is the *cofactor* of $a_{n,n}$ in this new expression of $\det(A)$. That is,

$$a^{\cdot}_{n,n} = \sum_{i=1}^{(n-1)!} \epsilon[\pi_i(n-1)] \prod_{j=1}^{n-1} a_{i,\pi_{i,j}}$$

Comparison with equation (2.1) shows that this cofactor is $\det(A_{n,n})$, where the matrix $A_{n,n}$ is of dimension $(n-1)\cdot(n-1)$ and obtained from A by 'striking-out' the n^{th} row and n^{th} column of A. A matrix obtained in this way by deleting the i^{th} row and j^{th} column of a matrix A is called a *minor* of A and written $A_{i,j}$. Any other element in A, $a_{i,j}$ say, can be chosen as the grouping pivot and can be moved to the $*_{n,n}$ position in A by a sequence $n-i$ row interchanges and $n-j$ column interchanges. Recalling property D.2, concerning the interchange of rows and columns, we can see that the cofactor of $a_{i,j}$, is just

$$a_{i,j}^{\cdot} = (-1)^{i+j}\cdot\det(A_{i,j}) \tag{2.3}$$

With each of these grouping steps we remove $(n-1)!$ terms so that only n elements of A can be chosen as grouping pivots. Any n distinct elements can be chosen but the common practice, and the most useful in applications, is to choose the n elements of a row or column. This way of evaluating determinants by grouping elements along rows or columns is called *Expansion By Minors* (EBM). Note that the row- or column cofactors used in EBM must correspond to the matrix row or column chosen as pivots for the expansion, otherwise the expansion evaluates to zero, as can be verified by an application of D.1. This can be formally expressed in Properties

D.5a EBM *using row i*: $\sum_{j=1}^{n} a_{i,j}\cdot a_{k,j}^{\cdot} = \begin{cases} \det(A), & k = i \\ 0, & k \neq i \end{cases}$

D.5b EBM *using column j*: $\sum_{i=1}^{n} a_{i,j} \cdot a^{*}_{i,k} = \begin{cases} \det(A), & k = j \\ 0, & k \neq j \end{cases}$

For example, we expand matrix (2.2) by pivoting on row 1:

$$\begin{aligned} \det(A) &= a_{1,1} \cdot a^{*}_{1,1} - a_{1,2} \cdot a^{*}_{1,2} - a_{1,3} \cdot a^{*}_{1,3} \\ &= a_{1,1} \cdot \det(A_{1,1}) - a_{1,2} \cdot \det(A_{1,2}) - a_{1,3} \cdot \det(A_{1,3}) \\ &= 1 \cdot (-7) - 2 \cdot (-4) - 3 \cdot (9) \\ &= 28 \end{aligned}$$

The cost of computing the determinant of an $n \times n$ matrix using the EBM process is $n!$ multiplications and $(n-1)!$ additions. This is an improvement, by a factor of $n-1$, over the previous cost estimates. While this complexity is still unsatisfactory for the practical

```
Mod_Vec Steve_Yu(Mod_Mtx x)
/*  EBM computation for determinant of a square matrix */
/*  Nice piece of recursive programming                */
{
     int i, j, k, dim;
     Mod_Vec res;
     res = x(0,0);
     if ((dim = Dim(x)) > 1) {
          Mod_Mtx y(dim-1, x(0,0));
          for (k=0; k<dim; k++) {
               for (i=0; i<dim-1; i++)
                    for (j=0; j<dim-1; j++)
                         y(i,j) = x(i+1,(j<k)?j:j-1);
               if (k%2)
                    res = res - x(0,k)*Steve_Yu(y);
               else
                    res = res + x(0,k)*Steve_Yu(y);
               }
          }
     else
          res = x(0,0);
     return (res);
}
```

Figure 2.1 - EBM in C++

evaluation of large matrices, the EBM algorithm is handy for computing determinants of small determinants, particularly by hand or even by inspection, as the example shows. Moreover, a very neat recursive algorithm for evaluating the determinant of a matrix can be based on the EBM properties. This is shown in Figure 2.1 in the style of the **C** language. This code is based on an original by Steve Yu, the author's colleague and fellow student, hence the eponymous reference. It appears here in a version used in the author's modular arithmetic software.

The *Adjoint* matrix, $A^{\cdot}$, to A is a matrix of the cofactors $a_{i,j}^{\cdot}$ written in transposed form, specifically

$$A^{\cdot} = \begin{bmatrix} a_{1,1}^{\cdot} & a_{2,1}^{\cdot} & a_{3,1}^{\cdot} & \cdots & a_{n,1}^{\cdot} \\ a_{1,2}^{\cdot} & a_{2,2}^{\cdot} & a_{3,2}^{\cdot} & \cdots & a_{n,2}^{\cdot} \\ a_{1,3}^{\cdot} & a_{2,3}^{\cdot} & a_{3,3}^{\cdot} & \cdots & a_{n,3}^{\cdot} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{1,n}^{\cdot} & a_{2,n}^{\cdot} & a_{3,n}^{\cdot} & \cdots & a_{n,n}^{\cdot} \end{bmatrix} \tag{2.4}$$

Computing the matrix product $A \cdot A^{\cdot}$ with the help of Properties D.5, will reveal that

$$A \cdot A^{\cdot} = A^{\cdot} \cdot A = \det(A) \cdot \mathbf{I}$$

where **I** is the identity matrix. Thus we have that

$$A^{-1} = \left(\frac{1}{\det(A)} \right) \cdot A^{\cdot} \tag{2.5}$$

The matrix A^{-1} is called the *inverse* to A since $A \cdot A^{-1} = A^{-1} \cdot A = \mathbf{I}$. A matrix whose

determinant is zero is said to be a *singular* matrix, and it is clear from (2.5) that a matrix has an inverse if and only if it is *non*-singular. We can use equation (2.5) to formulate a method for solving a system of linear equations. Let y be a column vector of dimension n whose elements are known. We *solve* the system of equations of the form $\sum_j a_{i,j} \cdot x_j = y_i$ by finding the vector x which satisfies these equations. It is clear that the system expressed in this in this way is equivalent to the *matrix* equation $A \cdot x = y$, and the solution will be given by the matrix equation $x = A^{-1} \cdot y$. From this and (2.5) we have immediately the well-known fact that a system has a solution if and only if the system matrix is non-singular. If the latter condition holds, then substituting from (2.5) in this equation we obtain, as the first row of the matrix into vector multiplication, for instance

$$\frac{1}{\det(A)} \left\{ y_1 \cdot a_{1,1}^{\cdot} - y_2 \cdot a_{2,1}^{\cdot} - \cdots y_n \cdot a_{n,1}^{\cdot} \right\}$$

We can recognize the bracketed expression, with the help of D.5b, as the column EBM of a matrix obtained from A by replacing its first column with the vector y. We also recognize this expression as the element x_1 of the solution vector. If we adopt the notation[4] $A_{k,y}$ as the matrix obtained from A by replacing its k^{th} column with the vector y, then we have property

D.6 *Cramer's Rule*: $x_k = \dfrac{\det(A_{k,y})}{\det(A)}$.

---

[4]We use the notation A_k to denote the k^{th} *column* of matrix A, and trust that no confusion with our previous notation for the *minor*, $A_{i,j}$, will arise.

We now consider the determinant of the sum of two square matrices of the same dimension. Let A and B be two matrices of dimension n, then we have

D.7 *Determinant of the Sum*:

$$\det(A+B) = \det(A) + \det(B) + \sum_{k=1}^{n-1} \sum_{\binom{n}{k}} \det(A_{i_1}\ B_{i_2}\ \cdots\ B_{i_k}\ \cdots\ A_{i_n})$$

The expression inside the double summation above is the determinant of the matrix A where k of its columns have been replaced by the corresponding columns of B. For example if n=10 and k=3, say, then the expression will contain determinant computations, among many others, for $\binom{10}{3} = 120$ matrices in each of which three of the columns of A have been replaced by the corresponding columns of B. Thus the expression will have

$$\det([A_1\ A_2\ B_3\ A_4\ B_5\ A_6\ A_7\ A_8\ B_9\ A_{10}])$$

as one of its terms. D.7 can be proved by induction on the dimension. We first compute the case n=2 directly,

$$\begin{aligned}
\det(A+B) &= \det\left(\begin{bmatrix} a_{1,1}+b_{1,1} & a_{1,2}+b_{1,2} \\ a_{2,1}+b_{2,1} & a_{2,2}+b_{2,2} \end{bmatrix}\right) \\
&= (a_{1,1}+b_{1,1})\cdot(a_{2,2}+b_{2,2}) - (a_{2,1}+b_{2,1})\cdot(a_{1,2}+b_{1,2}) \\
&= \det\left(\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}\right) + \det\left(\begin{bmatrix} b_{1,1} & a_{1,2} \\ b_{2,1} & a_{2,2} \end{bmatrix}\right) + \det\left(\begin{bmatrix} a_{1,1} & b_{1,2} \\ a_{2,1} & b_{2,2} \end{bmatrix}\right) + \det\left(\begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}\right) \\
&= \det([A_1\ A_2]) + \det([B_1\ A_2]) + \det([A_1\ B_2]) + \det([B_1\ B_2])
\end{aligned}$$

Now as the induction hypothesis assume that D.7 holds for dimension n-1, we can apply EBM in column 1, D.5b, to compute the case for dimension n.

$$\det(A-B) = \sum_{i=1}^{n} (a_{i,1} - b_{i,1}) \cdot (a_{i,1} - b_{i,1})^{\cdot}$$

$$= \sum_{i=1}^{n} (a_{i,1} - b_{i,1}) \cdot \left(a_{i,1}^{\cdot} - b_{i,1}^{\cdot} - \sum_{k} \sum_{\binom{n-1}{k}} a_{i,j_1}^{\cdot} B_{1} \ldots B_{j_k} \right)$$

We carry out the indicated product in the summation on the right hand side and get, all told, six terms which are all applications of D.5b. The first of these terms is $\det(A)$, the second is $\det(B)$ with the first column of B replaced by A_1, the third term is a sum of determinants of matrices containing a mixture of columns from A and B but whose first column is A_1. Similarly, the fourth term is $\det(A)$ with the first column replaced by B_1, the fifth term is $\det(B)$, and the last term is the sum of determinants whose matrices contain the mixture of columns but whose first column is B_1. Thus we obtain D.7.

As mentioned above, these properties of determinants *do* represent computational procedures, or algorithms. While they are not practical for the 'industrial strength' computations required for general computational geometry applications, we may apply them immediately to make the connection between the geometric domain of these applications and the algebraic domain of more practical algorithms.

2. Geometry

We introduce geometrical concepts using Figure 2.2. A *convex set* in R^n is a set of points

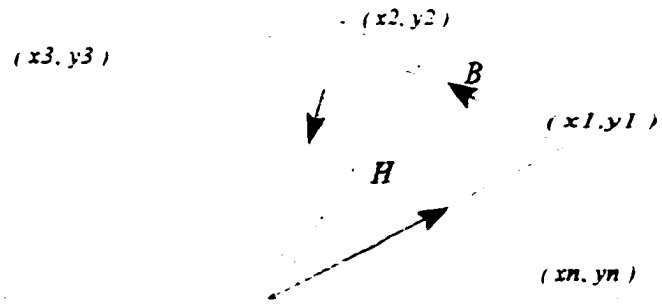


Figure 2.2 - Convex Set in 2-D

such that a line segment lies wholly in the set whose end points lie in the set. A *convex hull* is the *closure*, or boundary points, of a convex set. The polygon in Figure 2.2 depicts the convex hull of the set it encloses. The convex hull is specified by the points $v_i = (x_i, y_i)$, $1 \le i \le n$ which are oriented in the counter clock-wise sense in Figure 2.2.

The set of points, $\{v_i\}$, specify the convex hull, the convex hull itself is the union of the line segments (subsets in the convex hull) joining the v_i. Ordering of the finite set $\{v_i\}$, as in the counter clockwise orientation, is not required for this specification. However, orientation, the sequence of the points in the geometric sense, is crucial for applying computational techniques to solving geometrical problems. In these applications questions about geometrical orientation become questions about algebraic signs.

We remark that *convexity*, the *convex set*, the *convex hull*, *measure* (length, area, volume), and *orientation* are all *geometric* concepts. This contrasts with the essentially *algebraic* ideas of the previous section concerning determinants, ideas which are expressed in arithmetic domains.

It is easy to verify that the area of the shaded triangle in Figure 2.2, with a vertex at the origin and whose opposite side is one of the line segments of the convex hull, is equal to

$$\frac{1}{2}\cdot\det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$$

Moreover, we know, by D.2, that the sign of this expression is determined by the *order* of the vectors in the matrix. The area of the triangle is also given by the expression $\frac{1}{2}\cdot H\cdot B$, where H is the height and B is the length of the base of the triangle, namely the measure of the convex hull segment $\overline{v_1, v_2}$. This length is computed as

$$B = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{2.6}$$

and H will then be computed as

$$H = \frac{\det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}}{B} \tag{2.7}$$

The shaded triangle in Figure 2.2 was used to provide the link between the geometrical notion of orientation and the algebraic notion of signs of determinants. We may expand

on this connection by observing that if the determinant computation be extended to a summation of determinants over the point-pairs of the $\{v_i\}$, then the result would be the geometric area of the whole convex set. Specifically, we have

$$\text{Area} = \frac{1}{2}\left\{\sum_{i=1}^{n-1} \det([v_i, v_{i+1}]) + \det([v_n, v_1])\right\} \tag{2.8}$$

the result would be positive, so long as the $\{v_i\}$ are consistently *oriented in the counter clockwise sense*.

We explore this topic further in Figure 2.3 by inserting a new, variable, point (ξ, η), which can move around R^2. In this picture the new point is in the convex set. We can apply the same argument as used above to compute the area of the darkly shaded triangle, namely, we compute

$$\det\begin{pmatrix} x_1-\xi & x_2-\xi \\ y_1-\eta & y_2-\eta \end{pmatrix} = \det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} - \det\begin{pmatrix} x_1 & \xi \\ y_1 & \eta \end{pmatrix} + \det\begin{pmatrix} \xi & x_2 \\ \eta & y_2 \end{pmatrix} \tag{2.9}$$

This expression can be verified by a straight forward application of the basic definition of the determinant, (2.1), but it follows also as an application of D.7, where the matrix B is set to $-\begin{bmatrix} \xi & \xi \\ \eta & \eta \end{bmatrix}$. Note that this expansion follows the orientation of the smaller heavily shaded triangle in Figure 2.3, which is made to be consistent with the counter clockwise orientation of the convex hull. Since B has the same length as in the triangle of Figure 2.2, the height H of the heavily shaded triangle of Figure 2.3 will be computed as

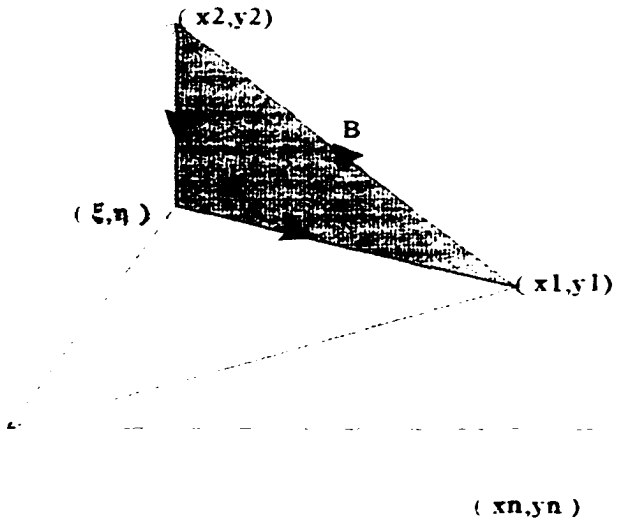


Figure 2.3 - Convex Set with Interior Point

$$H = \frac{\det\begin{pmatrix} x_1-\xi & x_2-\xi \\ y_1-\eta & y_2-\eta \end{pmatrix}}{B} \qquad (2.10)$$

H is the minimum distance from point $(\xi,\eta)^{\cdot}$ to the line defined by $\overline{v_1,v_2}$. Since the point $(\xi,\eta)^{\cdot}$ is variable and can move around R^2, H will vary as $(\xi,\eta)^{\cdot}$ varies. In particular this point may be moved into the convex hull segment $\overline{v_1,v_2}$ itself, in which case $(\xi,\eta)^{\cdot}$ becomes a linear combination of $(x_1,y_1)^{\cdot}$ and $(x_2,y_2)^{\cdot}$. That is, (2.9) is singular in the convex hull. Thus H measures the distance to the singular matrix or to the convex hull.

We shall apply (2.10) to the example shown in Figure 2.4. This shows a fragment of a

convex region in R^2 with $(\xi,\eta)^t$ set to points respectively inside and outside the region. Let $(x_1,y_1)^t = (23,9)^t$ and $(x_2,y_2)^t = (6,20)^t$. The two variable points are marked *Inside* at $(10,17)^t$ and *Outside* at $(15,15)^t$, respectively. We find that B is 20.2484 to an accuracy of 10^{-4}. Then applying (2.10), with the point $(\xi,\eta)^t$ set to *Inside*, we compute H to be 0.3457, small and positive. Setting $(\xi,\eta)^t$ to *Outside* we obtain, $H = -0.6914$, small and negative.

We remark that the difference in signs reflects a geometric relationship between the convex hull segment $\overline{v_1,v_2}$ and the points *Inside* and *Outside* which is just visibly resolved in Figure 2.4 but is clearly discriminated in the numerical results. Had the precision available for the computational work not been sufficient then the computational resolution of this relationship would have failed. This neatly describes

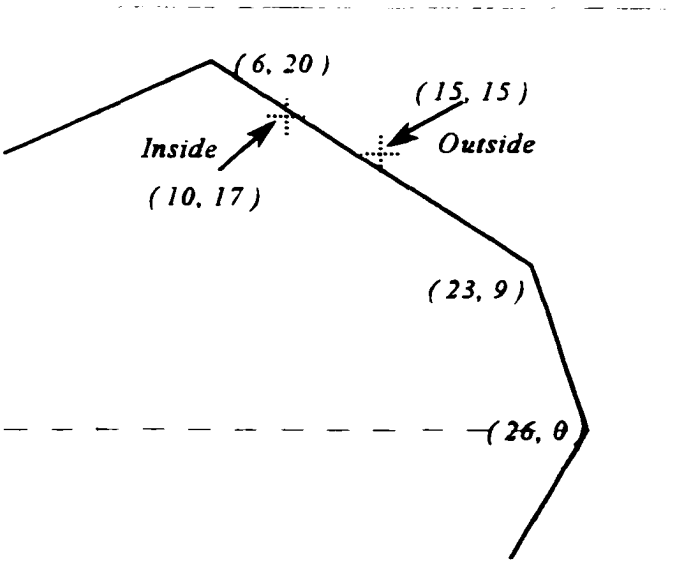


Figure 2.4 - Resolution Example

the fundamental question being examined in this thesis.

3. Extension to R^n

We now turn to the task of arithmetizing geometric concepts in the algebraic domain R^n. We define points:

$$A_i = (a_{1,i}, a_{2,i}, a_{3,i}, \cdots, a_{n,i})^t \in R^n$$

Let the set $\{A_1, A_2, \cdots, A_n\}$ be a *linearly independent* set of points in R^n. Then the determinant of the matrix $A = [A_1 \ A_2 \ \cdots \ A_n]$ is non-singular. Moreover, the points $\{A_1, A_2, \cdots, A_n\}$ specify a *hyperplane* in R^n, which is an element of R^{n-1}. Let $\hat{h} \in R^n$ be the unit *normal* vector to this hyperplane with *direction cosines* $\{\hat{h}_1, \hat{h}_2, \cdots, \hat{h}_n\}$. Then the hyperplane is the subspace

$$span(\{A_i - A_j\}),\ 1 \le j \ne i \le n,\ i \text{ fixed}$$

such that $\hat{h}$ intersects the hyperplane at a distance H from the origin. We have $A_i \cdot \hat{h} = H$, for $1 \le i \le n$.

The geometric situation in R^n is that the points $\{A_1, A_2, \cdots, A_n\}$ define a polytope of dimension $n-1$ in this hyperplane which has some *measure*, and with an *orientation* specified by the order in which these points are listed. That is, in the geometric domain the set $\{A_1, A_2, \cdots, A_n\}$ is *ordered*. The set $\{A_1, A_2, \cdots, A_n\} \cup \mathbf{0}$, where $\mathbf{0}$ denote the zero vector or origin, defines a polytope of dimension n. The orientation of this polytope will depend on where $\mathbf{0}$ is inserted in list $<A_1, A_2, \cdots, A_n>$. The measure of this latter polytope is $|\det(A)|$. *Measure*, in the R^n context, captures the geometric notion of size, i.e.,

length, area, or volume in the R^1, R^2, or R^3 domains, respectively. We note that the measure $\det(A)$ will be the product of the measure of the base polytope lying in the hyperplane and the distance H of this hyperplane from the origin. Note also that the sign of this determinant will that of H in this factorized geometric expression for the determinant. We now derive the computation of the base polytope measure.

The vertices A_j of the of the polytope each lie in the hyperplane so each satisfies the R^n equation of the hyperplane:

$$c_1 \cdot a_{1,k} + c_2 \cdot a_{2,k} + \cdots + c_n \cdot a_{n,k} = C, \quad 0 \le k \le n \tag{2.11}$$

where the c_i's and C are numbers. Since all the A_j satisfy (2.11), it can be written in matrix form $A \cdot c = \bar{C}$, where $c = (c_1, c_2, \cdots, c_n)$ and $\bar{C}$ is the vector $(C, C, \cdots, C)$. Using D.6 (Cramer's Rule) yields the solution

$$c_j = \frac{\det(A_j \leftarrow c)}{\det(A)} = \frac{C \cdot \sum_{i=1}^{n} a_{j,i}}{\det(A)}$$

Hence, (2.11) can be written

$$\sum_{i=1}^{n} \left\{ \sum_{j=1}^{n} a_{i,j} \right\} \cdot a_{i,k} = \hat{A} \cdot A_k = \det(A) \tag{2.12}$$

$\hat{A}$ is a vector whose component, $\hat{A}_i$, is the *sum* of the elements in the i^{th} *column* of the adjoint matrix A^*. But the i^{th} column of the adjoint contains the cofactors of a determinant computed as *projections* along the i^{th} coordinate axis. Hence the sum of the

column is a determinant, the measure of the base polytope, projected onto $\mathbf{i}$. So this determinant is the *vector* $\hat{A} \in \mathrm{R}^n$, and $\hat{A}_i$ is its i^{th} component. The measure of the base polytope is

$$\left(\sum_{i=1}^{n} \hat{A}_i^2 \right)^{\frac{1}{2}} = \|\hat{A}\|_2$$

and $\hat{A}_i = \|\hat{A}\|_2 \cdot \hat{h}_i$. Thus we have

D.8 *Geometric Factorization* $\det(A) = \|\hat{A}\|_2 \cdot H$

It is time for an example to illuminate the gathering darkness: Consider the $4 \cdot 4$ non singular matrix

$$A = \begin{bmatrix} 4 & 3 & -1 & 7 \\ -5 & 8 & 6 & -2 \\ 1 & 9 & 4 & 3 \\ -3 & 10 & -3 & 1 \end{bmatrix}$$

with determinant -1461. The four column vectors in this matrix specify the vertices of a 3-dimensional polytope which lies in a hyperplane of the 4-dimensional vector space. This polytope may be considered a facet in the hull of a convex set in R^4. Other (at least one other) points could be defined, which in turn would define the additional facets comprising the convex hull of the set. (2.12) and D.8 allow us to compute some facts about this polytope.

The adjoint to this matrix is the transpose of a matrix constructed of its cofactors:

$$A^{*} = \begin{bmatrix} 364 & 451 & -560 & 34 \\ 143 & 125 & -220 & -91 \\ -49 & -145 & -37 & 164 \\ -485 & -332 & 409 & 43 \end{bmatrix}$$

The polytope has a measure, actually its volume in 3-space, which is the value of a 3-dimensional determinant and which, in our 4-dimensional view, is a vector. The import of (2.12) is that the sum of the elements in the i^{th} column of the adjoint is the i-component of this vector. Then we see that the measure of the 3-dimensional polytope is just the Euclidean norm of the column sums. Hence we compute the measure of this facet in the convex hull to be 446.6475. From D.8 we find that the origin lies a perpendicular distance of -3.2710 from the this facet of the convex hull. The sign is determined by the ordering of the points specifying this facet and corresponds to the order of the vectors in A. We remark that equation (2.6) shows the computation of $\|\hat{A}\|_2$ in R^2. Equation (2.7), and the present example, illustrates the most useful application of D.8, namely computing H when $\det(A)$ is known. The sign of the determinant is the sign of H, of course.

In view of the fact that $\hat{A}_i$ is the sum of elements in the i^{th} column of the adjoint we see that $\hat{A}_i \le \|A^{*}\|_1$. Recalling 2.5, we have that

$$|\hat{A}| \le \sqrt{n}\|A^{*}\|_1 = \sqrt{n}\|A^{-1}\|_1 \cdot \det(A)$$

So we have an interesting corollary to D.8

D.8a *Floor on Height* $H > \dfrac{1}{\sqrt{n}\,\|A^{-1}\|_1}$.

This corollary is particularly interesting in the light it sheds on the relationship between geometric resolution and computational error. The proximity of the test point to the convex hull (which represents the singular matrix) is fixed by the norm of the inverse matrix. To obtain this knowledge the inverse must be computed, a computation that will give $\det(A)$ as a byproduct. Errors will be made in this computation, thus the floor of D.8a will have error associated with its computation and this error will provide the maximum resolution available for judging the proximity of the convex hull. If the error is too great then the sign of H will be suspect. We shall have more to say on this in the next section. We remark also that D.8 says that an estimate of the resolution requires the computation of the inverse, a computation that exceeds the requirements for $\det(A)$ by a considerable margin, so alternative methods for certifying H are worth seeking.

4. Sensitivity to Error

In this section we study the effect of computational error on the evaluation of determinants. For this study we will model *internal* computational error as a perturbation in *external* data. That is, suppose that $\det(A)$ is the result of evaluating the determinant of the matrix A by some computational procedure in an error-free arithmetic environment. Then $\det(A + E)$ is the computation, by the same procedure, of the

matrix A perturbed by the addition of an *error* matrix E of small elements $\{ e_{i,j} \mid 1 \le i,j \le n \}$. We may then argue that the effect of this perturbation on the computation of det(A) is identical to that of replacing the exact arithmetic by an environment in which computational errors occur. Then

$$e_d = | \det(A-E) - \det(A) | \tag{2.13}$$

measures the effect of these internal computational errors.

The $e_{i,j}$ are small signed numbers whose precise value is unknown which we take in our model to be generated by round-off or truncation in the computing machinery. So we can think of them as 'random' numbers. Since we are interested in their overall effect on the determinant computation, and would be satisfied with a useful bound on their effect, we can bound them in (2.13) by replacing each one with $e = \max_{i,j} \{ | e_{i,j} | \}$. This appears to be a reasonable, and safe, summary of the detailed occurrence of rounding errors in the determinant computation. The E matrix is just filled with e's and by applying D.7 to evaluate the determinant of the sum $A-E$, (2.13) simplifies to

$$\det(A+E) \le \det(A) + e \cdot \sum_{i,j} a_{i,j}^{\cdot} \tag{2.14}$$

That is, the effect of error on the computation of det(A)is bounded by a term consisting of the sum of the cofactors in A multiplied by the bound on the errors in internal arithmetic. Thus we have

$$e_d \le e \cdot \left| \sum_{i,j} a_{i,j}^{\cdot} \right| \tag{2.15}$$

This bound on e_d may be expressed in a more interesting way by recalling the relationship (2.5) between the Adjoint and the Inverse matrices:

$$e_d \le e \cdot \det(A) \cdot \Sigma \tag{2.16}$$

where Σ is the absolute value of the sum of the elements in the inverse matrix A^{-1}, for which we can put $0 \le \Sigma \le n \cdot \|A^{-1}\|_1$. Clearly neither upper or lower bound for Σ is a reasonable estimate of its value in practical cases. Experience suggests we consider the *granularity* of the elements of the inverse for this latter task, that is, we consider an estimate of the mean absolute value of these elements. The number turns out to be a good estimate of Σ, and, supposing we have $\|A^{-1}\|_1$ or a good estimate available, this granularity in A^{-1} can be approximated well with $\frac{\|A^{-1}\|_1}{n}$. Thus (2.16) can be written as an approximation

$$e_d \approx e \cdot \det(A) \cdot \frac{\|A^{-1}\|_1}{n}$$

In chapter 3, section 5, we demonstrate that an good estimate of $\|A^{-1}\|_1$ is available as a byproduct of the LU matrix factorization algorithm and exploit this fact to obtain the granularity estimate in the laboratory work of chapter 5. If we put $\hat{e}_d$ as a *relative* version of e_d, then we have

$$\hat{e}_d = e \cdot \frac{\|A^{-1}\|_1}{n} \tag{2.17}$$

We will occasionally refer to the coefficient multiplying the arithmetic error, e, in these expressions as the *error gain* factor associated with a computation.

5. Size of Determinants

In this section we study the size of determinants. More precisely we are concerned about establishing bounds on $\det(A)$ as functions of the norm of A, the dimension of A, or the magnitude of the elements of A. These bounds are of interest because they are properties that can be evaluated prior to the actual computation of $\det(A)$. We first discuss the so called *Hadamard* bound, $\|A\|^n$, n being the dimension of A. We then turn to the *Max Measure* computation, $2^{n-1}\cdot\alpha^n$, in which α is the element of A having the greatest absolute value. We will then discuss these bounds as they relate to the size of determinants as they may be encountered in applications.

The Hadamard bound may be proven by induction on matrix dimension. We use the 1–norm, that is $\|A\|_1 = \max_j \sum_i |a_{i,j}|$. We chose this norm here because it is readily computed from the input data, matrix A. The case of dimension one is exceedingly trivial to show: $|\det(A)| = |a| \le \|A\|$, where a is the single element of A. The case of dimension 2 is more interesting:

$$|\det(A)| \le \begin{vmatrix} \frac{\|A\|}{2} & -\frac{\|A\|}{2} \\ \frac{\|A\|}{2} & \frac{\|A\|}{2} \end{vmatrix} = \frac{\|A\|^2}{2} < \|A\|^2$$

Now let $|\det(A)| \le \|A\|^k$ for all square matrices of dimension $k<n$, we show that this implies that Hadamard holds for the case of dimension equal to n. We apply D.5b to expand $|\det(A)|$ using column 1:

$$\det(A) = \sum_{i=1}^{n} a_{1,i} \cdot a_{1,i}$$

$$\leq \sum_{i=1}^{n} \frac{|A_1|}{n} \cdot |A|^{n-1} \leq |A|^{n-1} \sum_{i=1}^{n} \frac{|A_1|}{n}$$

$$\leq |A|^{n-1} \cdot |A_1| \leq |A|^n$$

where $|A|$ is the maximum of the norm of the minors in A. Clearly equality holds in Hadamard only in the case of dimension one. In practice the Hadamard bound proves to be more than ample except in special cases, particularly in sparse matrices.

Max Measure may be proved similarly with induction on matrix dimension. The case of dimension one is also trivial, in fact in this case $\alpha = |a_{1,1}|$, and $|\det(A)| = \alpha$. A study of the case $n=3$ is illuminating. Consider the $3 \cdot 3$ matrix

$$\begin{bmatrix} \alpha & -\alpha & x \\ \alpha & \alpha & -\alpha \\ \alpha & \alpha & \alpha \end{bmatrix}$$

which is obtained from any 3×3 matrix by replacing every element except the upper right with $\pm\alpha$ as shown, where α is the absolute value of the element having the largest absolute value The determinant of this matrix is $4\alpha^3$. It is easy to see that this value is the largest determinant that can be achieved from any 3×3 matrix with an element no larger than α in absolute magnitude, given the above rules of construction. By these rules $|x| \leq \alpha$. Moreover, replacing α with any smaller value in any element in this matrix will result is a smaller determinant. For instance, setting the $*_{2,1}$ element to

$y \leq \alpha$ results in the determinant $3\alpha^3 - (y-x)\alpha^2 + xy\alpha < 4\alpha^3$.

So we assert that the $n \times n$ matrix

$$\begin{bmatrix} \alpha & -\alpha & x & x & \cdots & x \\ \alpha & \alpha & -\alpha & x & \cdots & x \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \alpha & \alpha & \alpha & \cdots & -\alpha & x \\ \alpha & \alpha & \alpha & \cdots & \alpha & -\alpha \\ \alpha & \alpha & \alpha & \cdots & \alpha & \alpha \end{bmatrix}$$

obtained from any $n \times n$ matrix whose elements in absolute magnitude do not exceed α and constructed following the rules outlined above has the largest attainable determinant in absolute value of all $n \times n$ matrices. This value is $2^{n-1}\alpha^n$. We adopt as an induction hypothesis that this assertion holds for all matrices of dimension less than n, and in particular we have that the maximum determinant for the $n-1 \times n-1$ matrix is $2^{n-2}\alpha^{n-1}$. We construct the $n \times n$ case by adjoining a column of n α's on the left and the row $(\alpha, -\alpha, x, \ldots, x)$ on the top of the $n-1 \times n-1$ matrix. It is then trivial to verify that the hypothesis holds for $n \times n$ matrices. It is harder to verify that the determinant is indeed the maximum possible. Consider expanding this determinant by the top row, using D.5a. The x's in this row all have a zero cofactor since the first two columns of the Minor are identical, they are linearly dependent and so, by D.1, their determinant vanishes. Thus the value of x has no bearing on the determinant. Of course, by the rules of construction the x's cannot exceed α. Replacing either of the α's in the top row with some $|y| < \alpha$, clearly results in a smaller determinant. We now turn to the effect on the $n \times n$

determinant of modifying the α's in the left-most column. Only these additional elements must be considered since only this column was adjoined to the $n-1 \times n-1$ case. But settling the question of the top row also serves to answer questions about these left column elements. Indeed, expanding by the minors of the top row involve cofactors in which these left-hand column elements figure. But these cofactors are $n-1 \times n-1$ determinants, which by hypothesis are the maximum attainable. Hence we have the Max Measure result: $\det(A) \leq 2^{n-1}\alpha^n$.

Max Measure is sharp, that is equality holds in every dimension. In fact we have just constructed maximum determinants to demonstrate the Max Measure. In general Max Measure is less generous than Hadamard, and so it is a tighter bound than Hadamard. It is easy to find matrices, however, where this relationship is reversed. Moreover, both bounds prove to be overly tolerant in practical applications. The Hadamard bound was used to estimate the Error Gain factor in Proposition 4.1 of [PYS], our equation (2.14). The generosity of this estimate dominated the results of the numerical experiments reported in that paper. Max Measure would not have provided much tighter results.

Nevertheless, while *a posteriori* computed estimates, such as those involving the norm of inverse matrices, are much more realistic, the *a priori* Hadamard or Max Measure determinant size estimates do have the advantage of being immediately available and can be used to estimate critical parameters for algorithms, for instance, the number and size of moduli needed in modular arithmetic computations. But note that in certified

computations these modular parameters based on *a priori* estimates, such as Hadamard or Max Measure, prove to be excessive. In the certified paradigm parameters based on *a posteriori* estimates are available and much more efficient estimators.

It is interesting to see how the *a priori* Hadamard or Max Measure bound compares to the size of actual determinants. The latter is a hard population to capture, but Figure 2.5 shows results derived by computing the determinants of a number of 'random' matrices of dimension 5. A random matrix is one populated by random integers drawn from a uniform distribution over the interval $[-a, a]$. Determinant sizes are markedly skewed toward the large values (Figure 2.5 somewhat exaggerates this feature because the horizontal co-ordinate axis is logarithmic, the MAX ABS DET number is $2^4 \cdot 30000^5$). In

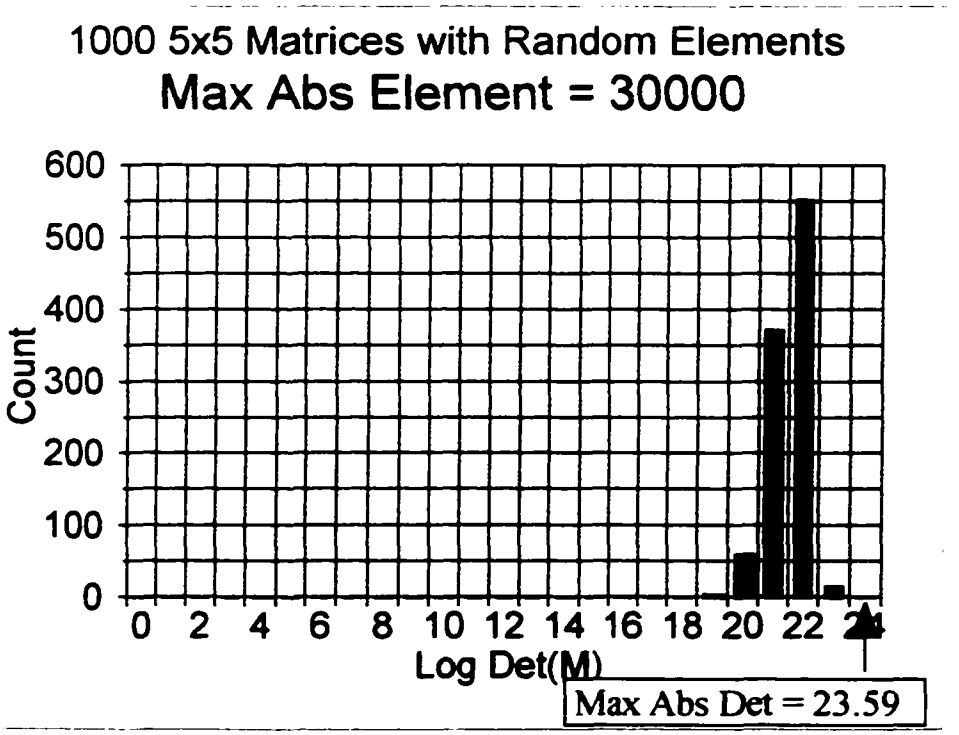


Figure 2.5 - Size of Determinants

any case, the facts illustrated in Figure 2.5 are easily understood since small determinants occur only when test points are near convex hulls, and points near the convex hull of sets are a vanishingly small minority of all points. Figure 2.5 makes this very clear since 'random' matrices correspond to a Monte-Carlo sampling of points in a vector space.

The facts illustrated in Figure 2.5 can be usefully correlated with our remarks above concerning the size of moduli required in exact arithmetic computations. We see that most determinants are large, which might imply the need for large moduli (see chapter 4), except that, in general, the signs computed for large determinants in floating point arithmetic are certifiable, that is, they are trustworthy. Thus no exact arithmetic is needed. Only the small determinants yield questionable results under floating point and require the exact arithmetic recalculation. But in these cases only small moduli are needed.

Chapter 3

Floating Point Arithmetic and Algorithms

We study the LU matrix factoring algorithm: It provides as good a numerical tool as can be found for computing determinants and can be immediately used, without (much) modification, for the same task in the modular arithmetic domain. But this chapter concerns the floating point case with its ever present companion, computational error. LU is an old, important, and much visited, topic in the history of machine computation and many outstanding computer scientists have contributed, see [Tu48], [FM67], [GolVL], [ORT], [StB], and [High]. In section 1 we take a brief, but sufficient, look at machine arithmetic, including floating point, from a symbolic point of view. Section 2 is detailed study of the LU algorithm. In section 3 we arm ourselves with a theory concerning the effects of floating point arithmetic error on the performance of the algorithm and then examine the actual case by using empirical methods; i.e. we measure things. It is well known that the quality of the LU algorithm can be considerably improved by using a 'pivoting' strategy, that is by resequencing the order in which its input is consumed. This is the topic of section 4: We find that pivoting is crucial to our application of the algorithm. We show in section 5 how by-products of the LU factorization may be used to compute an excellent estimate of the norm of the inverse of the input matrix. Interestingly, this valuable additional service has not been a featured topic in the literature on the LU algorithm, but it is quite useful to us.

1. Computer Arithmetic

Computer arithmetic is an old topic and has been discussed by many authors, see for instance [CdB], [FM67], [GolVL], [ORT], [StB], [High]. Our purpose in reviewing the topic here is to identify the differences between *exact* arithmetic and *floating point* arithmetic and to justify our treatment of error in algorithm analysis and design.

Consider the polynomial expression

$$s\cdot\left\{d_{w-1}\cdot b^{w-1} + d_{w-2}\cdot b^{w-2} + \cdots + d_1\cdot b + d_0\right\} \tag{3.1}$$

where w, b, and the d_i's are small natural numbers. The sign, s, is the number +1, or -1. Let $0 \le d_i < b$. Then when s, b, w, and the d_i's are given, (3.1) evaluates to a signed number. A number is written as an *ordered list* of s and the d_i's: $<s, d_{w-1}, d_{w-2}, \cdots, d_1, d_0>$. This list is the data structure used in machine arithmetic, and machine arithmetic manipulates these lists. The latter algorithms are embedded in the machine's Arithmetic Processing Unit. There are many variants of this *positional system* of writing and manipulating numbers used in actual computers, but these essentials are common to all.

This system is, in fact, identical to the paper-and-pencil arithmetic we learn in school. There are two major differences, however, between the latter and computer arithmetic. w is finite and fixed in machine versions of the scheme and the arithmetic algorithms are executed with much greater speed and, supposedly, greater accuracy in computers. This

comment may seem unnecessary here, but computers *do* generate errors, in copious quantity, and this phenomenon is one of our major concerns. These errors occur just because w is finite in machines.

If the machine representation of a number can be correctly specified by the coefficients of (3.1), $<s,d_{w-1},d_{w-2},\cdots,d_1,d_0>$, then we say the number is *writable*. As an example of (3.1), let w=6, b=10, and s be the sign tag, + or -. Then we could write the numbers 123 and -58, say. Moreover, we could write their sum, 65, or their difference, 181, or their product, -7134. We could not write the quotient $123 \div -58$, -2.120689655..., since there is no room in the format for all the digits required nor is there provision for the decimal point. On the other hand the Euclidean (modular) division, $-7 \equiv 123 \bmod -58$, could be accommodated easily. But, many writable numbers would generate an unwritable product when multiplied by 123. In fact, this number format is unusable in general numeric application. On the other hand (3.1) supports exact arithmetic when operands and results are guaranteed to be writable. For example, if operands were never permitted to exceed 999 in absolute value then our six-digit format would support error-free computations with the operations $+, -, \cdot$, mod, and the arithmetic comparison predicates. Multi-precision arithmetic models are based on (3.1), see [Knu]. Pan, [P97] has recently proposed a high-precision exact arithmetic model using (3.1) with b being a sizeable fraction of w and where numbers are written as *vectors* whose components are the exact coefficients of (3.1), i.e. lists $<s,d_{w-1},d_{w-2},\cdots,d_1,d_0>$. The modular ring arithmetic described in chapter 4 of this thesis may be implemented under (3.1).

A classical accommodation to the requirements of numerical computation extends (3.1) to the *floating point* format by appending an additional polynomial. A pair of numbers are written: the *mantissa*

$$s_m \cdot \left\{ m_1 \cdot b^{-1} + m_2 \cdot b^{-2} + \cdots + m_{w-1} \cdot b^{-(w-1)} + m_w \cdot b^{-w} \right\} \tag{3.2}$$

and the exponent

$$s_e \cdot \left\{ e^{p-1} \cdot b^{p-1} + \cdots + e_1 \cdot b + e_0 \right\} \tag{3.3}$$

In (3.2) and (3.3) b, p, w, the m_i's, and the e_i's are small natural numbers as in (3.1). The mantissa (3.2) is considered a *normalized fraction* m, i.e. $\frac{1}{b} \le m < 1$. Thus *leading zeros* never appear in the mantissa, and all the m_i's are considered to be significant. The exponent is used as a *scale factor*. Thus the real number 123 would be written <+123000,-03> in a floating point format where b=10, w=6, and p=2. In this same format the number $0.001230\overline{0}$ is written <+123000,-03>, and the number 123456789.321... would be written <+123456,+09>. Note that in this latter example information beyond the sixth digit is lost. Note also that in this format numbers larger than 10^{99} or smaller than 10^{-99} cannot be written.

The floating point format has proven very successful in numerical applications. Indeed, It has been the essential tool in numeric work and has been considered such from the beginning of the computer era. But since it is encumbered with limits on the magnitude and precision of numbers that can be written it does not fully and accurately represent the real algebraic domain. These limits are rather capacious in modern machinery, to be

sure. Machines used in serious numerical work use a p rarely greater than 4, but a w of up to 27 is available, as double precision, in some of these machines (e.g. Cray, CDC6600, CDC7600, CDC Cybers). The IEEE standard provides a w of 16, which is routinely available as double precision in desktop microprocessors and work-station computers. We remark that the arithmetic hardware uses a w exceeding these parameters by some margin, but it will be the errors induced by *storing*, in memory, of intermediate or final results that determines the performance of algorithms.

All information beyond the w^{th} digit in an operand is lost when it is stored. The relative error of this operation is thus bounded by b^{1-w}. This bound is called the *machine precision* and is denoted ϵ. Let x be the real value of the operand, then the bound on the error, that is, the maximum amount of information that will be lost in the storage operation, is just $x \cdot b^{1-w}$. Then, if a computation involves K steps with storage (and subsequent retrieval) of intermediate operands x_i, we can put

$$e \leq K \cdot x \cdot \epsilon \tag{3.4}$$

as the cumulative error bound on the process, where $x = \max_i x_i$.

2. LU Algorithm

The *LU matrix decomposition algorithm* is a procedure for factoring a matrix A into two matrices L and U such that $A = L \cdot U$ The matrix L is lower *unit triangular* and the matrix U is *upper triangular* That is we get the following schematic decomposition:

$$\begin{bmatrix} a & a & a & a & a & a \\ a & a & a & a & a & a \\ a & a & a & a & a & a \\ a & a & a & a & a & a \\ a & a & a & a & a & a \\ a & a & a & a & a & a \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ l & 1 & 0 & 0 & 0 & 0 \\ l & l & 1 & 0 & 0 & 0 \\ l & l & l & 1 & 0 & 0 \\ l & l & l & l & 1 & 0 \\ l & l & l & l & l & 1 \end{bmatrix} \cdot \begin{bmatrix} u & u & u & u & u & u \\ 0 & u & u & u & u & u \\ 0 & 0 & u & u & u & u \\ 0 & 0 & 0 & u & u & u \\ 0 & 0 & 0 & 0 & u & u \\ 0 & 0 & 0 & 0 & 0 & u \end{bmatrix}$$

In the above scheme the a's, l's, and u's represent numbers not necessarily zero. Although it can be used in problems involving rectangular matrices the LU algorithm is usually associated with applications involving square, dense, matrices. It is considered the choice numerical computing tool for these problems.

The LU algorithm is based on the Gaussian Elimination process for solving a system of equations which in turn is based on the fact that the addition or subtraction of a multiple of one equation in the system from another will not change the solution of the system. We prefer to present the algorithm in terms of matrix operations (see, e.g. [GolVL], [FM67]). Given the matrix to be factored

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}$$

we left multiply this matrix by a matrix derived from computations on elements appearing in column 1 to get

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ -\frac{a_{2,1}}{a_{1,1}} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ -\frac{a_{n,1}}{a_{1,1}} & 0 & \cdots & 1 \end{bmatrix} \cdot \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ 0 & a_{2,2}-\frac{a_{1,2}\cdot a_{2,1}}{a_{1,1}} & \cdots & a_{2,n}-\frac{a_{1,n}\cdot a_{2,1}}{a_{1,1}} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n,2}-\frac{a_{1,2}\cdot a_{n,1}}{a_{1,1}} & \cdots & a_{n,n}-\frac{a_{1,n}\cdot a_{n,1}}{a_{1,1}} \end{bmatrix} \quad (3.5)$$

It can be seen that the left hand matrix was derived just so the first column of the product matrix has the form $(a_{1,1},0,\cdots,0)$. Moreover, this left hand matrix has the form

$$L^{(1)} = \mathbf{I} - [G_1 \quad \mathbf{0} \quad \cdots \quad \mathbf{0}]$$

where $\mathbf{I}$ is the identity matrix, $\mathbf{0}$ is the zero vector and the vector G_1 is the *gaussian spike*

$$\left(0, \frac{a_{2,1}}{a_{1,1}}, \frac{a_{3,1}}{a_{1,1}}, \cdots, \frac{a_{n,1}}{a_{1,1}}\right)$$

Observe that the top row in (3.5) has not been changed by this computation. (3.5) may be written as $L^{(1)} \cdot A = A^{(1)}$, where the right hand matrix is

$$\begin{bmatrix} u_{1,1}^{(1)} & u_{1,2}^{(1)} & \cdots & u_{1,n}^{(1)} \\ 0 & u_{2,2}^{(1)} & \cdots & u_{2,n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & u_{n,2}^{(1)} & \cdots & u_{n,n}^{(1)} \end{bmatrix}$$

This procedure may be continued, marching down the diagonal, so that the k^{th} step would be

$$L^{(k)} \cdot L^{(k-1)} \cdot \ldots \cdot L^{(1)} \cdot A = L^{(k)} \cdot U^{(k-1)} = U^{(k)}$$

$L^{(k)}$ is the matrix $\mathbf{I} - \begin{bmatrix} \mathbf{0} & \cdots & \mathbf{0} & G_k & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix}$ and its product into $U^{(k-1)}$ is

$$U^{(k)} = \begin{bmatrix} u_{1,1}^{(k-1)} & u_{1,2}^{(k-1)} & u_{1,3}^{(k-1)} & \cdots & u_{1,k}^{(k-1)} & u_{1,k+1}^{(k-1)} & \cdots & u_{1,n}^{(k-1)} \\ 0 & u_{2,2}^{(k-1)} & u_{2,3}^{(k-1)} & \cdots & u_{2,k}^{(k-1)} & u_{2,k+1}^{(k-1)} & \cdots & u_{2,n}^{(k-1)} \\ 0 & 0 & u_{3,3}^{(k-1)} & \cdots & u_{3,k}^{(k-1)} & u_{3,k+1}^{(k-1)} & \cdots & u_{3,n}^{(k-1)} \\ \vdots & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & u_{k,k}^{(k-1)} & u_{k,k+1}^{(k-1)} & \cdots & u_{k,n}^{(k-1)} \\ 0 & 0 & 0 & \cdots & 0 & u_{k+1,k+1}^{(k-1)} - u_{k,k+1}^{(k-1)} \cdot g_{k+1,k} & \cdots & u_{k+1,n}^{(k-1)} - u_{k,n}^{(k-1)} \cdot g_{k+1,k} \\ \vdots & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 0 & u_{n,k+1}^{(k-1)} - u_{k,k+1}^{(k-1)} \cdot g_{n,k} & \cdots & u_{nn}^{(k-1)} - u_{k,n}^{(k-1)} \cdot g_{n,k} \end{bmatrix} \tag{3.6}$$

The elements $g_{i,k}$ in the gaussian spike G_k are

$$g_{i,k} = \begin{cases} 0, & i \le k \\ \dfrac{u_{i,k}^{(k-1)}}{u_{k,k}^{(k-1)}}, & i > k \end{cases} \tag{3.7}$$

Thus the effect of this step in the computation is to zero column k below the diagonal and

modify the elements of the $(n-k)\times(n-k)$ lower diagonal submatrix according to the expression

$$u_{i,j}^{(k-1)} - u_{k,j}^{(k-1)} \cdot \left(\frac{u_{i,k}^{(k-1)}}{u_{k,k}^{(k-1)}} \right) \tag{3.8}$$

This procedure stops at step $n-1$. The entire decomposition procedure may be written

$$L^{(n-1)} \cdot L^{(n-2)} \cdot \ldots \cdot L^{(2)} \cdot L^{(1)} \cdot A = U^{(n-1)} = U$$

and so if we put

$$L^{-1} = L^{(n-1)} \cdot L^{(n-2)} \cdot \ldots \cdot L^{(2)} \cdot L^{(1)} \tag{3.9}$$

we have that $A = L \cdot U$. Clearly U is upper triangular and L^{-1} is lower unit triangular since it is a product of unit lower triangular matrices. Moreover, L is unit lower since it is the inverse of a unit lower matrix. Hence we have accomplished the desired decomposition of A. From (3.9) we obtain

$$L = L^{(1)-1} \cdot L^{(2)-1} \cdot \ldots \cdot L^{(n-2)-1} \cdot L^{(n-1)-1} \tag{3.10}$$

We must evaluate (3.10) and it is often desirable to evaluate (3.9) also. Indeed in the many important applications where the LU algorithm is used to solve systems of equations L^{-1} is required instead of L. In these cases we want to find the vector x such that $A \cdot x = y$ where vector y is the given input data. Using LU, we put $L \cdot U \cdot x = y$, which can be rewritten as $U \cdot x = L^{-1} \cdot y = y'$, so y' is computed as a matrix-vector multiplication where L^{-1} is required. Since U is upper triangular, x can then be evaluated by means of the Backsubstitution algorithm (see section 5). Moreover, we will need L^{-1} since it is the path to computing A^{-1}, for we can write

$A \cdot A^{-1} = L \cdot U \cdot A^{-1} = \mathbf{I}$. This says that $U \cdot A_i^{-1} = L_i^{-1}$, or, in words, that back substituting the i^{th} column of L^{-1} gives us the i^{th} column of A^{-1}.

To compute L and L^{-1} it is helpful to have some facts about the gaussian spike matrices. In particular we need their products

$$\begin{bmatrix} \mathbf{0} & \cdots & \mathbf{0} & G_i & \mathbf{0} & \cdots \mathbf{0} \end{bmatrix} \times \begin{bmatrix} \mathbf{0} & \cdots & \mathbf{0} & G_j & \mathbf{0} & \cdots \mathbf{0} \end{bmatrix} \tag{3.11}$$

in cases where $i<j$, $i=j$, and $i>j$. It is quite easy to discover that this product is the *zero* matrix in the cases where $i \leq j$. The case $i>j$ is more challenging but after some tinkering with direct evaluation we find that the product above is

$$\begin{bmatrix} \mathbf{0} & \cdots & \mathbf{0} & \dot{G}_j & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix}$$

where $\dot{G}_j$ is a spike, in column j, whose elements are

$$\dot{g}_{k,j} = \begin{cases} = 0, & k<i-1 \\ = g_{k,j} \cdot g_{i,j}, & k>i-1 \end{cases} \tag{3.12}$$

(3.12) says that the new gaussian matrix contains a single spike in the j^{th} column, $\dot{G}_j$, consisting of the spike G_i whose elements are all multiplied by the i^{th} element of the spike G_j.

With these results it is easy to show that $L^{(k)-1}$ is the matrix

$$\mathbf{I} - \begin{bmatrix} \mathbf{0} & \cdots & \mathbf{0} & G_k & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix}$$

since $L^{(k)} \cdot L^{(k)-1} = \mathbf{I}$. Note that the only difference between $L^{(k)}$ and its inverse $L^{(k)-1}$ is

just that the spikes are subtracted from the unit matrix in the former and added in the latter. We can also show without much trouble that

$$L = L^{(1)^{-1}} \cdot L^{(2)^{-1}} \cdot \ldots \cdot L^{(n-2)^{-1}} \cdot L^{(n-1)^{-1}} = \mathbf{I} + \begin{bmatrix} G_1 & G_2 & \cdots & G_{n-1} \end{bmatrix}$$

which says in words that L is just the unit lower triangular matrix whose columns below the diagonal are the gaussian spikes used in the LU decomposition. Computing L^{-1} is more involved. Define $\hat{L}^{(k)}$ as the matrix product $L^{(k)} \cdot \ldots \cdot L^{(2)} \cdot L^{(1)}$, the partial product in (3.9). That is, $L^{-1} = L^{(n-1)} \cdot \ldots \cdot L^{(k-1)} \cdot \hat{L}^{(k)}$. Let $\hat{l}_{i,j}$, $i>1$, $j<k$ be the elements of $\hat{L}^{(k-1)}$, and $g_{i,k}$, $i>k$ be the elements in the gaussian spike in $L^{(k)}$. Then we have

$$\hat{L}^{(k)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \hat{l}_{2,1} & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \hat{l}_{3,1} & \hat{l}_{3,2} & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & & & \vdots & \vdots & \vdots & \vdots & \vdots \\ \hat{l}_{k,1} & \hat{l}_{k,2} & \cdots & \hat{l}_{k,k-1} & 1 & 0 & 0 & \cdots & 0 \\ \hat{l}_{k+1,1} - \hat{l}_{k,1} \cdot g_{k+1,k} & \hat{l}_{k+1,2} - \hat{l}_{k,2} \cdot g_{k+1,k} & \cdots & \hat{l}_{k+1,k-1} - \hat{l}_{k,k-1} \cdot g_{k+1,k} & g_{k+1,k} & 1 & 0 & \cdots & 0 \\ \hat{l}_{k+2,1} - \hat{l}_{k,1} \cdot g_{k+2,k} & \hat{l}_{k+2,2} - \hat{l}_{k,2} \cdot g_{k+2,k} & \cdots & \hat{l}_{k+2,k-1} - \hat{l}_{k,k-1} \cdot g_{k+2,k} & g_{k+2,k} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \hat{l}_{n,1} - \hat{l}_{k,1} \cdot g_{n,k} & \hat{l}_{n,2} - \hat{l}_{k,2} \cdot g_{n,k} & \cdots & \hat{l}_{n,k-1} - \hat{l}_{k,k-1} \cdot g_{n,k} & g_{n,k} & 0 & 0 & \cdots & 1 \end{bmatrix}$$

This result is the product $L^{(k)} \cdot \hat{L}_{k-1}$ and can be verified by applying (3.5), where the left hand factor is a sum of spike matrices, and by applying (3.12). The effect of multiplying $\hat{L}_{k-1}$ on the left by $L^{(k)}$ is to add the gaussian spike G_k and otherwise leave all the other elements of $\hat{L}_{k-1}$ unchanged except those in the lower left $n-k \cdot k-1$ submatrix. These

will undergo a modification according to the expression

$$\hat{l}_{i,j} = \hat{l}_{i,j} - \hat{l}_{k,j} \cdot \left(\frac{u_{i,k}^{(k-1)}}{u_{k,k}^{(k-1)}} \right), \quad k<i\le n,\ 1\le j<k$$

which is obtained by comparing with (3.8).

We have two versions of the LU algorithm, one, which we denote as LU', which computes L in addition to U, and another version, LU, which computes L^{-1} in place of L. Both L and L^{-1} are lower unit triangular matrices and a study of the algorithm shows that these matrices may replace the lower triangle of A as the algorithm proceeds. Hence the input matrix may be overwritten by the algorithm to produce an output matrix of the form

$$\begin{bmatrix} d & u & u & u & u & u \\ l & d & u & u & u & u \\ l & l & d & u & u & u \\ l & l & l & d & u & u \\ l & l & l & l & d & u \\ l & l & l & l & l & d \end{bmatrix}$$

The *d's* represent the *diagonal* elements $u_{i,i}$. The *l's* represent the proper lower triangular body of L or L^{-1}, where the diagonal 1's are understood. The *u's* are the proper upper triangular body of U. We present the two versions of LU as **C** code fragments, LU' in Figure 3.1, and LU in Figure 3.2. The difference between these two codes is hard to spot, being just the insertion of the if (k != j) ...

```
/*    Algorithm LU'      */
{
  Matrix A[n][n];
  int    i,j,k;

  for (i=1; i<n; i++)
   for (j=i+1; j<n; j++) {
     A[j][i] = (A[j][i]/A[i][i]);
     for (k=i+1; k<n; k++)
          A[j][k] = A[j][k] - A[i][k]*A[j][i];
   }
}
```

Figure 3.1-- LU' Algorithm

statement and the change in the `for(k=...` statement in the LU code. While the textual difference is slight, the operation count of the two algorithms differ more significantly. Table 3.1 displays this computation cost data as a function of input matrix dimension. The cost of the *if* statement in LU is buried in the $O(n^2)$ componentof the Compare count. Note that the entries in the table record asymptotic complexity measures and the difference between the two versions of LU are not that drastic in matrices of small dimension. Indeed, for small values of n the $O(n^2)$ terms

```
/*  Algorithm LU      */
{
   Matrix A[n][n];
   int    i,j,k;

   for (i=1; i<n; i++)
     for (j=i+1; j<n; j++) {
       A[j][i] = (A[j][i]/A[i][i]);
       for (k=i+1; k<n; k++)
          if (k != i)
            A[j][k] = A[j][k] - A[i][k]*A[j][i];
     }
}
```

Figure 3.2-- LU' Algorithm

Algorithm	LU^*	LU^-
Addition	$\frac{1}{3}\cdot n^3+O(n^2)$	$\frac{1}{2}\cdot n^3+O(n^2)$
Multiplication	$\frac{1}{3}\cdot n^3+O(n^2)$	$\frac{1}{2}\cdot n^3+O(n^2)$
Division	$\frac{1}{2}\cdot n^2+O(n)$	$\frac{1}{2}\cdot n^2+O(n)$
Compare	$\frac{1}{3}\cdot n^3+O(n^2)$	$\frac{1}{2}\cdot n^3-O(n^2)$

Table 3.1 – Operation Counts for LU

play a significant role in the actual operation counts. Nevertheless, LU is clearly more costly to run than LU^*. This fact will figure significantly in the design of the algorithm we use for evaluating the certification predicate of Algorithm 0.

3. Estimating the Errors in the LU Algorithm

Errors are generated in the floating point implementation of the LU Algorithms. That is, the factors L and U obtained by these algorithms for an input matrix A will not, in fact, be the factors of A. Nevertheless they will be lower unit and upper triangular matrices and may be multiplied to obtain a resulting matrix A'. An important question is: What is the relation between the input matrix A and the product matrix A'? Clearly a difference matrix $E = A - A'$ will contain essential information about the errors generated in the execution of the LU algorithms.

Recall the model used in chapter 2 for the discussion of the effect of errors in the computation of determinants. There the internally generated errors were 'accounted' for by replacing the exact input A by a 'nearby' input $A+E$, where E is an error matrix. In that discussion we let e denote an upper bound on the magnitude of the elements of this error matrix. This number was found to be an important factor in estimating the sensitivity of the determinant computation to truncation and round-off errors. We will show that this bound is

$$e \le (3n-2)\cdot a \cdot \epsilon \qquad (3.13)$$

where n is the dimension of the input matrix A, a is the magnitude of the largest internal operand encountered in the process of computing the LU decomposition, and ϵ is the floating point machine precision. (See also, e.g. [GolVL], [FM67], [PYS]).

We can prove (3.13) by first recalling (3.4), the general expression for evaluating floating point errors. Clearly a, in (3.13), substitutes directly for the quantity x in the general expression. Our real task is to justify the value $3n-2$ used in (3.13) to substitute for K, the count of operand stores in the general expression (3.4). This value can be established by studying the data-flow character of the LU algorithm with the help of Figure 3.3. Referring to the preceding description of the LU algorithm, particularly equations (3.6), (3.7), and (3.8), we see that the diagonal element $u_{k,k}^{(k-1)}$ will be the *pivot* element in step k of the algorithm. That is, the $u_{k,k}$ of step $k-1$ will appear as the denominator in expressions (3.7) and (3.8) which implement step k. These expressions also appear in

the assignment statements of the inner loops of the coded versions of the algorithm in Figures 3.1 and 3.2. Figure 3.3 depicts this portion of the computation as a data-flow diagram where $i>k$ indexes a typical row, and $j>k$ a typical column, in the matrix of step k. Element $u_{i,j}$ is an element being updated in this step. The number C is the count of data storage operations that have occurred in the computation of $u_{k,k}^{(k-1)}$ up to the completion of step $k-1$.

A study of the data-flow diagram reveals that one more data storage operation will occur on the computation of (3.6) in step k, and that an additional one will occur upon the update of a typical matrix element, $u_{i,j}$, $i>k$, $j>k$. Hence, the data storage count will be $C-1$ for all elements $u_{i,k}$, $i>k$ in column k, and this value will *not* change thereafter in subsequent steps of the algorithm. Moreover, the data storage count will be $C-2$ for all

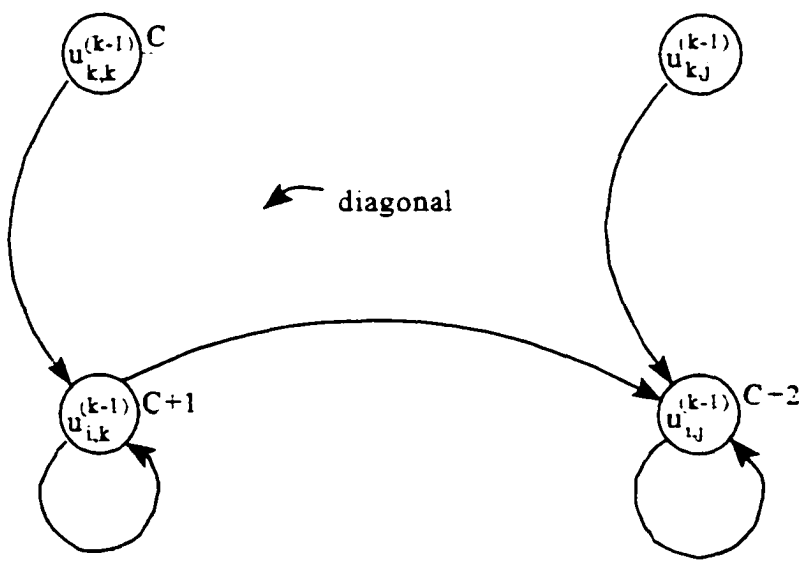


Figure 3.3 - LU Data Flow

elements $u_{i,j}$, $i>k$, $j>k$ at the completion of the k^{th} step. However, this count may increase in subsequent steps. Since the storage count starts at zero, we can see that, at the completion of step k, the storage operation count for elements in column $j \le k$ below the diagonal will be $2j-1$. The storage count for elements in row $i \le k$, including the diagonal element, will be $2i-2$. The count for all remaining elements, at the completion of the k^{th} step, will be $2k-2$. At the completion of the algorithm, the n^{th} step only element $u_{n,n}$ falls in this latter category and its storage count will be $2n-2$. So that we can conclude that the bound on the magnitude of error in the elements of the LU decomposition is $(2n-2)\cdot a^{*}\cdot\epsilon$. The matrix components of this decomposition, namely L and U, must be multiplied to get A'. Since the elements of this latter matrix are inner products of n-dimensional vectors, these elements will accumulate an additional n storage operations during their computation. Hence we have the result (3.13).

We now examine the relationship between the *actual* errors encountered in computing the LU decomposition and (3.13) by means of computational experiments. We are concerned, in particular, about the quality of that bound, that is, how well it estimates actual computational error. We make use of the following notation scheme: Let e_{act} denote the actual error as captured from the result of a computation, and let e_{est} denote the estimated quantity as computed from (3.13). We use the superscript * to denote the maximum of a quantity, thus a^{*} denotes the absolute value of the largest operand occurring in a computation. The symbol ϵ denotes the machine precision, which in our work is $2^{-53} = 1.11 \cdot 10^{-16}$. We use the symbol α to denote the upper bound on the

absolute magnitude of a matrix element and $M^{\cdot}$ to denote the *Max Measure*[5] of a matrix, i.e. $M^{\cdot} = \frac{(2\alpha)^n}{2}$. Also we use the symbol ρ to denote the logarithm of the absolute magnitude of the determinant of a matrix relative to the logarithm of its Max Measure, that is $\rho = \frac{\log|\det|}{\log M^{\cdot}}$. Thus for any matrix, $0 \le \rho \le 1$. We have seen, in section 5 of chapter 2, that ρ tends to be large, $= 0.9$, for matrices drawn randomly from uniform distributions Chapter 5 is devoted to a full description of our experimental methods, sources of data, and additional results.

The experiments consist of applying the LU algorithm to obtain the factors L and U of each input matrix A. We capture the quantities $\det(A)$, α, and a from this computation and calculate ρ, the log relative magnitude of $\det(A)$. We obtain $A' = L \cdot U$ by matrix multiplication and compute the matrix $E = A - A'$. We capture e_{act} as the element of E with the largest magnitude. Equation (3.13) is used to compute e_{est} using these values, and the quantity $e_{rel} = \frac{e_{act}}{e_{est}}$ is calculated. Note that if $e_{rel} < 1$ then (3.13) correctly bounds the measured error in the LU decomposition. These data are then reduced and tabulated as follows: The mean value of ρ for each set of input matrices is computed and tabulated as $\bar{\rho}$. The maximum value of e_{act} in each set is tabulated as $\hat{e}_{act}$. The corresponding value of e_{est} is tabulated as $\hat{e}_{est}$. The mean value of e_{rel} over the input set is computed and tabulated as $\overline{e_{rel}}$, and the maximum value of e_{rel} is tabulated as $\hat{e}_{rel}$. We also record, and denote as k_0, the number of matrices in each input set such that $e_{act} = 0$.

---

[5]See chapter 2, section 5

$\alpha \leq 10000$, random matrices						
dim	$\bar{\rho}$	$\bar{e}_{act}$	$\bar{e}_{est}$	$\overline{e_{rel}}$	$\bar{e}_{rel}$	k_0
2	0.910	9.09495e-13	1.13376e-11	0.0129	0.2547	37
3	0.900	7.27596e-12	5.77002e-11	0.0374	0.1753	21
4	0.906	7.27596e-12	6.70730e-11	0.0480	0.1992	8
5	0.915	1.45519e-11	1.85089e-10	0.0424	0.1400	4
6	0.919	2.91038e-11	3.11491e-09	0.0376	0.1068	1
7	0.928	2.91038e-11	6.07090e-10	0.0310	0.1295	0
8	0.933	4.65661e-10	2.78332e-08	0.0245	0.0812	0
9	0.936	3.72529e-09	1.89062e-07	0.0236	0.0880	0
10	0.943	1.86265e-09	5.04295e-08	0.0211	0.0570	0

Table 3.2 - Experiment 1: LU Error

The first experiment uses sets of 40 matrices each in which the matrices are all of the same dimension with elements drawn randomly from a uniform distribution of integers in the range [-10000, 10000]. The results are displayed in Table 3.2. A second experiment uses a similar input data organization where matrices are populated from a

$\alpha \leq 1{,}000{,}000$, random matrices						
dim	$\bar{\rho}$	$\bar{e}_{act}$	$\bar{e}_{est}$	$\overline{e_{rel}}$	$\bar{e}_{rel}$	k_0
2	0.936	2.32831e-10	1.17632e-09	0.0290	0.3164	33
3	0.943	4.65661e-10	3.00543e-09	0.0485	0.1785	18
4	0.934	9.31323e-10	4.49312e-08	0.0435	0.1369	9
5	0.942	1.86265e-09	1.63470e-08	0.0398	0.1257	2
6	0.946	3.72529e-09	6.49853e-08	0.0367	0.1127	0
7	0.951	7.45058e-09	1.21834e-07	0.0257	0.0635	0
8	0.953	7.45058e-09	2.50705e-07	0.0205	0.0647	0
9	0.960	3.72529e-09	1.02377e-07	0.0171	0.0528	0
10	0.960	7.45058e-09	2.86612e-07	0.0196	0.0676	0

Table 3.3 - Experiment 2: LU Error

uniform distribution of integers over the range $[-10^6, 10^6]$, i.e. $\alpha < 1{,}000{,}000$. These results are found in Table 3.3.

These two experiments verify the soundness of the error bound (3.13) for the case of matrices populated with random integers. In fact they demonstrate that e_{act} is consistently and safely bounded by the (3.13) expression, usually much less. Moreover, the results of these experiments demonstrate the expected behavior of error with respect to matrix dimension and matrix element size. But, as is to be expected from essentially unsmoothed random data, the results are not monotonic. Of interest is the number of computations in which errors are zero, the k_0 columns, which in these first two experiments appears to be a feature of small dimension. We will return to study this rather surprising result.

The next two experiments focus on the relationship between computational error and determinant size. Data for these experiments consist of sets of 32 5-dimensional matrices of known determinant. The method used to generate matrices with known determinant is explained in chapter 5, here we just remark here that these matrices are also generated from integers drawn randomly from uniform distributions but that certain parameters figuring in their construction must be fixed to guarantee the desired determinant. The procedures used tabulate the previous results are also used for these experiments but dimension, of course, no longer figures as a tabulated variable. Also, ρ is tabulated as a *nominal* value, denoted ρ_0, for each data set. Table 3.4 shows the

$\alpha \leq 10000$, fixed determinant, dim = 5					
ρ_0	e_{act}	e_{est}	$\overline{e_{rel}}$	e_{rel}	k_0
0.050	7.27596e-12	1.36547e-10	0.0286	0.1027	7
0.100	9.09495e-13	4.26037e-11	0.0292	0.1041	2
0.200	3.63798e-12	4.92011e-11	0.0269	0.0934	2
0.300	3.63798e-12	4.73033e-11	0.0248	0.1083	4
0.400	1.81899e-12	2.01653e-10	0.0173	0.0694	4
0.500	2.91038e-11	2.13949e-10	0.0381	0.1592	4
0.600	3.63798e-12	8.13410e-11	0.0211	0.1024	8
0.700	2.32831e-10	1.80472e-09	0.0157	0.1290	8
0.800	9.31323e-10	1.76179e-08	0.0184	0.1161	5

Table 3.4 - Experiment 3: LU Error

results of experiment 3 where the randomly selected elements in the matrices are drawn from a distribution of integers over the interval $[-10000, 10000]$. Table 3.5 shows results for experiment 4 where the corresponding interval is $[-10^6, 10^6]$.

$\alpha \leq 1,000,000$, fixed determinant, dim = 5					
ρ_0	e_{act}	e_{est}	$\overline{e_{rel}}$	e_{rel}	k_0
0.050	4.65661e-10	5.63836e-09	0.0247	0.0826	4
0.100	4.65661e-10	5.75832e-09	0.0261	0.0877	5
0.200	4.65661e-10	6.90543e-09	0.0285	0.1254	4
0.300	1.86265e-09	5.61698e-08	0.0265	0.0877	8
0.400	4.65661e-10	5.22269e-09	0.0260	0.0892	6
0.500	3.72529e-09	1.63889e-07	0.0338	0.1313	2
0.600	1.49012e-08	2.83606e-07	0.0183	0.0820	4
0.700	1.49012e-08	7.65807e-07	0.0076	0.1224	4
0.800	2.98023e-08	1.73659e-05	0.0045	0.0713	9

Table 3.5 - Experiment 4: LU Error

These results are compatible with those of experiments 1 and 2. There appears to be some correlation of error with determinant size in the results of experiments 3 and 4 which is not a feature of (3,13). But this finding may be connected with certain aspects of the algorithm for generating matrices with known determinant, namely that the *randomly* selected elements used to populate the matrices, under the constraint of fixed determinant, tend to grow in magnitude with determinant size. This artifact of the experimental data will be studied in more detail in chapter 5.

Nevertheless, the results of the present experiments confirm the correctness of (3.13): *Equation* (3.13) *bounds the actual floating point errors generated by the LU algorithm.*

We return now to the number of zero errors enumerated in the k_0 columns of the tables. The results of experiments 3 and 4 confirms that this is a phenomenon not necessarily associated with matrices of small dimension. These are unexpected findings. We emphasize that these errors are *zero*, not just numbers smaller then ϵ! A close look verifies that, in fact, in all cases studied, a non-zero entry is a minority in the population of the matrix E, the result of the matrix subtraction $A - L \cdot U$. Figure 3.4 displays a typical error matrix associated with a computation done in assembling the data for Table 3.5. We write this matrix in terms of ϵ to highlight the fact that the entries in the typical E matrix turn out to be precisely multiples of the machine precision, i.e., single bit errors, and to show the range

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2048\cdot\epsilon & 0 \\ 0 & 0 & 2048\cdot\epsilon & 2048\cdot\epsilon & 0 \\ 0 & 0 & 0 & 0 & 64\cdot\epsilon \end{bmatrix}$$

Figure 3.4 - Typical Error Matrix

of magnitude typical of these entries. This example, selected at random from the several hundred generated in an experimental run, aligns with the correponding observation that the majority of actual errors recorded in the experimental data are of the single-bit kind, i.e. of the form 2^{-k}, $k<53$.

The prevalence of zero entries in the error matrices invites the conjecture that some process of *random cancellation of error*, a variant of *catastrophic subtraction*, is more active in floating point computations, at least in the LU algorithms, than intuition would suggest. The most probable outcome of this process is zero error, much less probable is the single-bit form we commonly see in the error matrices. Less likely still are multi-bit errors, which do occur but very infrequently. A corollary to this conjecture is just what we actually see in the results of our experiments, namely *that the bound estimate*, (3.13), *comfortably overstates the actual error by a nearly constant factor.*

4. Stability of the LU Algorithm, Pivoting

In the description of the LU process, equations (3.7) and (3.8), the diagonal element $u_{k,k}^{k-1}$ appears as a denominator in the computation of the gaussian spike of L_k. We call this denominator element a *pivot* element in the LU process. It is clear that if this pivot is small then the entries in the spike will be large and this result will appear subsequently as a large multipliers in equation (3.8) of the computation. The effect is that small entries in the diagonal of the input matrix will induce large internal operands in the LU process and this in turn inflates the errors produced in the process. The

ultimate result is to compromise the stability of the algorithm (see, e.g., [GolVL] sections 4.3, 4.4, pp62 et. seq.).

In chapter 1 we equated instability in algorithms with unpredictable behavior. We can be more precise: Suppose that an algorithm f yields output y from an input x, i.e., $f(x) = y$. The output of f when given a nearby input $x + \delta$ (where δ is 'small' in some sense) is $y + \Delta$. Then we say that f is unstable (at x) if Δ is unbounded. This phenomenon occurs in the LU algorithm whenever $u_{k,k}^{(k-1)}$ is zero. Moreover a 'small' change consisting of interchanging rows or columns of the input matrix will, for non-singular matrices, avoid the zero pivot. We can argue that row or column interchange is 'small' since their effect is just to change the sign of the determinant of the matrix. Thus LU can exhibit instability according to the definition. We do not know the value of the pivot ahead of time except for the $a_{1,1}$ element in the input matrix. Moreover, the diagonal elements are updated as the algorithm progresses. Hence the LU instability phenomenon is unpredictable.

This classical instability in the LU algorithm is just an extreme expression of the general relationship between pivot size, internal operand, and computational error. Evidently pre-conditioning the input matrix using row and column interchanges in such a way as to favor large pivots will eliminate unstable behavior. But this will also favor small internal operands and thus we can suppose the preconditioning process will also act to reduce computational errors in the LU algorithm.

A *permutation matrix* is a square matrix whose columns are vectors containing all zeros except for a one in some position in the column and such that the one is in a distinct position in each column. The matrix

$$P = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

is a permutation matrix of dimension 5. Premultiplying a 5×5 matrix A by P will interchange the rows of A and post multiplying will interchange the columns of A. Our objective, then, is to find permutation matrices P and P' such that we obtain optimally large pivot elements when we apply the LU decomposition algorithm to the permuted matrix PAP'. We remark that the determinant of a permutation matrix is ± 1, the exact determination of which is easy. Hence we have that $\det(PAP') = \pm\det(A)$.

It is clear, unfortunately, that obtaining the optimal permutation matrices P and P' for the scheme outlined above is, at best, a complex task requiring trial applications of LU. However, a modification of the scheme which computes a nearly optimal P and P' may be proposed with the help of the following observations. Consider the situation described in (3.6): Step k of the LU algorithm is about to be executed and element $u_{k-1,k-1}^{(k)}$ will be used as the pivot denominator in (3.7) and (3.8). But any element in the $(n-k)\times(n-k)$ lower diagonal submatrix may be selected as the pivot by interchanging rows and columns, in particular the element of greatest magnitude may be selected.

While this interchange will not affect the computations already done in the previous $k-1$ steps of the algorithm, it will result in the LU algorithm yielding the decomposition of $A' = P_k A P_k'$, a different matrix than A. In principle the permutation matrices P_k and P_k' may be output at this stage and saved. We use the permutation matrices recovered at each stage to construct the modified matrix:

$$A' = P_1 \cdot P_2 \cdot \cdots \cdot P_{n-1} \cdot A \cdot P'_{n-1} \cdot \cdots \cdot P'_2 \cdot P'_1 = P \cdot A \cdot P'$$

and the decomposition obtained will be that of the transformed matrix A'. In view of the criterion for selecting the permutations at each stage, this modification of LU will favor the selection of large pivots.

While the sketch just described is not yet a practical algorithm, it is not difficult to put it into a workable form by replacing the permutation matrices with the equivalent permutation *lists* introduced in chapter 2 as coefficients of (2.1), the definition of determinants. We recall the π-notation introduced in chapter 2 where $\pi_k(n)$ denotes the k^{th} distinct permutation of the list of numbers <1,2,...,n> and we wrote $\pi_{k,j}$ to denote the number in the j^{th} position in the permuted list $\pi_k(n)$. We may relate a permutation matrix to this π-notation by letting $\pi_P(n)$ denote the π-notation equivalent to P and defining the element $p_{i,j}$ of P as

$$p_{i,j} = \begin{cases} 1, & \text{if } \pi_{P,j} = i \\ 0, & \text{otherwise} \end{cases} \qquad (3.14)$$

For example, the $5 \cdot 5$ permutation matrix P given above is written in π-notation as

<2.4.1.3.5>. Note also that det(P) is just $\epsilon[\pi_p(n)]$ where we use the parity function introduced in chapter 2. Thus for the example above we have det(P) = $\epsilon[\pi_p(5)]$ = -1.

Figure 3.5 displays this *Full Pivoting* modification of the LU algorithm of Figure 3.1 where the int vectors row and col are the permutation lists corresponding to P and P' respectively. The functions init(n, n) and xch(n, i, j), respectively, initialize the list n to <1,2,...,n> and exchange the elements of the list in positions i and j. A key feature of this implementation is that all addressing of the input matrix is done *indirectly* through these permutation lists. A fortuitous side effect of this method of implementing permutations is that row and col may be incorporated in the class definition of the Matrix object so that subsequent operations on this matrix, as in

```
/*   Algorithm LU'      */
/*    Full Pivoting */

Matrix A[n][n];
int row[n], col[n];
int   i,j,k;
double u, v;

init row, n);
init col, n);
for (i=1; i<n; i++) {
  v = abs(A[row[i]][col[i]]);
  for (j=i; j<n; j++)
     for (k=i; k<n; k++)
      if (v < (u= abs(A[row[j]][col[k]])) {jj=j; kk=k; v=u};
    xch(row,i,jj);
     xch(col,i,kk);
    for (j=i+1; j<n; j++) {
      A[row[j]][col[i]]-=A[row[j]][col[i]]/A[row[i]][col[i]];
     for (k=i+1; k<n; k++)
       A[row[j]][col[k]]+=A[row[i]][col[k]]*A[row[j]][col[i]];
     }
  }
}
```

Figure 3.5--LU' Algorithm with Full Pivoting

Add	$\frac{2}{3}n^3 + O(n^2)$
Multiply	$\frac{1}{3}n^3 + O(n^2)$
Divide	$\frac{1}{2}n^2 + O(n)$
Compare	$\frac{2}{3}n^3 + O(n^2)$
F. P. Compare	$\frac{2}{3}n^3 + O(n^2)$

Table 3.6 - Operation Count for Full Pivot LU

later algorithms taking this object as input, may address the permuted matrix A by using *indirect* addressing through the lists `row` and `col`. Moreover, the determinant of the original input matrix is easily recovered from the Full Pivoting output as

$$\det(A) = \epsilon[\mathrm{row}(n)] \cdot \epsilon[\mathrm{col}(n)] \cdot \det(A')$$

where $\det(A')$ is the product of the diagonals of the permuted matrix obtained by indirect addressing through `row` and `col`.

Table 3.6 displays the computational cost of the Full Pivoting modification. This table should be compared to Table 3.1, operation counts for the basic LU algorithms. A new operation count is introduced: The *Floating Point Compares* account for the search for maximum pivot at each step, one count is manifest in the comparison `if(tmp < abs(...` and one is implicit in the `abs(...` operation. This latter operation also accounts for one more addition count in the inner loops of the algorithm. The accounting in Table 3.6 does not record the cost of the `init(m,n)` and

`xch(π, i, j)` function calls. These are small: `init` is called once and `xch` is called *n* times in the course of executing the algorithm on an n-dimensional matrix.

Computational experiment 5 is designed to examine the effectiveness of full pivoting using the data of experiment 2, described above, as input to algorithm LU' in both the standard and full pivoting form. As in experiment 2, we capture and tabulate e_{act}, the maximum error magnitude for each set of input data. We also capture α, the absolute value of the largest element, and a', the maximum absolute value of all the operands arising the computation, for each matrix input. We then compute the operand swell ratio $t = \frac{a'}{\alpha}$ for each matrix and tabulate the mean value of t, denoted $\bar{t}$, over each data set. These results are shown in Table 3.7.

dim	no pivoting		full pivoting	
	e_{act}	$\bar{t}$	e_{act}	$\bar{t}$
2	2.32831e-10	2.03	5.82077e-11	1.02
3	4.65661e-10	4.76	1.16415e-10	1.12
4	9.31323e-10	6.46	1.16415e-10	1.19
5	1.86265e-09	9.31	2.32831e-10	1.32
6	3.72529e-09	14.73	2.32831e-10	1.44
7	7.45058e-09	29.19	2.32831e-10	1.52
8	7.45058e-09	55.18	2.32831e-10	1.54
9	3.72529e-09	86.70	3.49246e-10	1.67
10	7.45058e-09	66.18	4.65661e-10	1.75
25	2.38419e-07	572.21	9.31323e-10	2.70

Table 3.7 - Experiment 5: Pivoting Study

These results clearly demonstrate the effect on operand growth that we expect from full pivoting. Moreover, the verification of our expectations concerning its effect on computational error is striking. We remark that the study of t, under the alternative name 'growth factor' in the literature, has been the topic of much activity in recent years, see [High, 1996, section 9.3 pp 177 et.seq.]. Practical experience dating from the earliest days of automatic computation has consistently shown a growth factor far smaller than the theoretical estimates would suggest. Turing's early work [Tu48, 1948] was a response, for instance, to Fox's success with LU factorization against the daunting odds published by Hotelling in 1943. Wilkinson (1961, cited in [High], op. cit.) calculated a bound $O(n^{\frac{1}{2}\cdot\frac{\log n}{4}})$ on t for full pivoting, but Wilkinson also showed that this bound was unattainable. In fact the worst cases for full pivoting found so far has $t = O(n)$, which is about what we see in Table 3.7. All the recent work seems to be a search for the 'worst case' or an effort to find counter examples to the established theory. Our results are consistent with experience. We note especially that the test matrices we use are not special but do run over the full range of determinant sizes possible within the constraints of dimension and maximum element size.

5. Computing and Estimating the Inverse Matrix

Recall (2.17), which estimates the relative error in the determinant:

$$\hat{e}_d = e^{\cdot\frac{A^{-1}}{n}} \qquad (2.17)$$

We have previously established an upper bound on e in this equation, namely

$$e \le (3n-2)\cdot a^{-1}\cdot\epsilon \tag{3.13}$$

Hence the key task remaining toward obtaining a bound on determinant error is computing or estimating the norm of the inverse matrix. We review here our previous remarks on the need for obtaining L^{-1} and the role that the Backsubstitution algorithm plays in computing A^{-1}. We have that $A\cdot A^{-1} = \mathbf{I}$. But since $A = L\cdot U$, we get that $U\cdot A^{-1} = L^{-1}$, or that the columns of L^{-1} are the columns of A^{-1} premultiplied by the matrix U. This says that A_j^{-1}, the j^{th} column of A^{-1} is the solution to the matrix equation $U\cdot A_j^{-1} = L_j^{-1}$. Recall also that the LU algorithm of Figure 3.2 yields a matrix containing U as the upper triangle and L^{-1} as the lower proper triangle, where the 1's in the diagonal are implied. That is, LU produces the matrix output

$$\begin{bmatrix} d_1 & u_{1,2} & u_{1,3} & \cdots & u_{1,n} \\ l_{2,1} & d_2 & u_{2,3} & \cdots & u_{2,n} \\ l_{3,1} & l_{3,2} & d_3 & \cdots & u_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & l_{n,3} & \cdots & d_n \end{bmatrix}$$

where the $l_{i,j}$ are the elements of L_j^{-1} and the d_i are the diagonal, or pivot, elements of U, that is, the $u_{i,i}$.

Now, since U is upper triangular we may use the Backsubstitution algorithm to compute A^{-1}. This algorithm is well-known, see for instance [GolVL], algorithm 4.1-2, page 53.

```
Matrix A, B
int i, j, k, n=dim(A);
element ONE=1.0, ZERO=0.0;

for (k=0; k<n; k++)
     for (j=n-1; j>=0; j--) {
        B(j,k)=(k<j)?A(j,k):((k==j)?ONE:ZERO);
        for (i=j+1; i<n; i++) B(j,k) -=B(i,k)*A(j,i);
        B(j,k) /= A(j,j);
     }
```

Figure 3.6 - Inverse Matrix Algorithm

It works recursively from the 'bottom' up, that is, it computes the elements $a_{i,j}$ of A_j^{-1} in the order $\{a_{n,j}, a_{n-1,j}, \cdots, a_{1,j}\}$. The Backsubstitution algorithm can be written

$$a_{i,j} = \frac{\lambda_{i,j} - \sum_{k=i+1}^{n} a_{k,j} \cdot u_{i,k}}{d_i}, \quad \text{for } \lambda_{i,j} = \begin{cases} l_{i,j} & i > j \\ 1 & i = j \\ 0 & i < j \end{cases} \tag{3.15}$$

Thus, if $j < n-1$ for instance, $a_{n,j} = \frac{l_{n,j}}{d_n}$ and $a_{n-1,j} = \frac{l_{n-1,j} - a_{n,j} \cdot u_{n-1,n}}{d_{n-1}}$. Also note that, in particular, $a_{n,n} = \frac{1}{d_n}$.

(3.15) is implemented in Figure 3.6, using the C language style. `Matrix A` is the input matrix assumed to be result of applying algorithm LU, possibly with full pivoting. When this algorithm terminates, `Matrix B` contains the inverse matrix. Addressing is indirect to accommodate the possibility of pivoting in the LU algorithm. As we remarked in the previous section, the permutation lists which record the row- and column permutations are implicit in the class definition of `Matrix` in this code. Indirect addressing through the *class member* permutation lists is done by using the *class method* `*(i, j)`, where `*` represents a matrix object. The norm of the inverse, $\|B\|_1$, which is

Addition	$\frac{1}{2}n^3 + O(n^2)$
Multiplication	$\frac{1}{2}n^3 + O(n^2)$
Division	n^2
Compare	$\frac{1}{2}n^3 + O(n^2)$

Table 3.8 - Operation Count for Backsubstitution

required in (2.17), is obtained from the output of the algorithm by selecting the maximum of the column vector one-norms of B.

Operation counts for the Backsubstitution algorithm are given in Table 3.8. This computation adds considerably to the over all cost of certifying the sign of the determinant. However, as (2.17) requires only the norm of the inverse there is considerable motivation for seeking a reliable estimate of this quantity in an effort to avoid the cost of computing an inverse matrix. An examination of (3.15) suggests such an estimator provided we are willing to bear the cost of full pivoting in computing LU The calculation of an element of the inverse matrix. $a_{i,j}$ will be dominated by a small diagonal d_i. In fact all elements in the j^{th} column of the inverse above the i^{th} row will be similarly dominated owing to the recursive nature of (3.15). If the full pivoting discipline is applied during the LU factoring process, the diagonals, d_i, tend to be maximized, with the exception of d_n, which is not subject to pivoting. Moreover, the determinant computed from the results of the factoring algorithm is just the product of the diagonals. Hence, if the determinant is small, relative to the MaxMeasure, say, of the matrix, then d_n must necessarily be small and its reciprocal will dominate the

elements of the inverse matrix.. Since it is precisely the small determinants, in the above mentioned sense, whose computational error must be examined carefully for certification, then it is clear that the n^{th} diagonal, or rather its reciprocal, appears as a potential estimator for the norm of the inverse matrix, $\|A^{-1}\|_1$.

This conjecture is examined in experiment 6. We use the data of experiment 4. Recall that this data consists of sets of input matrices of dimension 5 and such that the absolute value of the matrix elements is less than 10^6. Each set consists of matrices with fixed determinant. We use sets consisting of 32 matrices for the present experiments. We compute the LU factorization algorithm (Figure 3.2), with and without pivoting for each matrix. We compute the inverse matrix and capture the norm of the inverse matrix, $\|A^{-1}\|_1$ for each input matrix. We also capture the value of the diagonal element in the factorization with the *smallest* absolute value. Note that this is nearly always, but need not necessarily always be, element $u_{5,5}$. We compute a number c as the *ratio* of the norm of the inverse to the reciprocal of the magnitude of this smallest diagonal. Note, then, that the norm of the inverse may be calculated by taking the product of c into the reciprocal magnitude of the smallest diagonal element in each case.

We present the results of these experiments in Table 3.9 In the table each set of data is denoted by the nominal value of ρ, tabulated as ρ_0. The norm of the inverse is tabulated as an average value over each set, denoted as $\overline{\|A^{-1}\|}$. The value of c averaged over each set is tabulated as $\bar{c}$. Three times the standard deviation of c over each set is also

ρ_0	Full Pivoting			No Pivoting		
	$\overline{\|A^{-1}\|}$	$\bar{c}$	$3\sigma(c)$	$\overline{\|A^{-1}\|}$	$\bar{c}$	$3\sigma(c)$
0.05	3.09865e+16	1.070	0.765	4.27962e+17	2.539e+12	3.969e+13
0.10	1.12452e+16	1.056	0.494	1.68192e+16	3.072e+11	2.802e-12
0.15	1.71192e+15	1.090	0.544	7.18865e+15	1.138e+11	1.080e+12
0.20	8.55628e+13	1.152	0.760	8.67681e+13	2.271e+10	2.458e+11
0.25	5.77671e+12	1.232	0.881	6.19371e+12	2.045e+09	2.146e-10
0.30	2.01740e+11	1.409	1.222	1.93923e+11	2.513e+08	2.350e+09
0.35	7.12823e+09	1.930	2.004	7.13764e+09	7.277e+07	5.762e-08
0.40	6.27072e-08	2.442	1.748	6.27090e+08	7.524e-07	6.199e-08
0.45	9.51118e+07	3.131	1.991	9.51118e+07	3.797e-07	3.222e-08
0.50	2.63940e+07	2.052	0.785	2.63940e+07	2.843e+07	2.057e-08
0.55	4.13505e-05	1.294	0.345	4.13505e+05	4.280e-06	5.603e-07
0.60	1.93105e+04	1.077	0.106	1.93105e+04	4.192e-05	4.501e-06
0.65	3.73536e-02	1.070	0.423	3.73542e-02	4.660e-04	6.017e-05
0.70	7.09246e-00	1.142	0.528	7.09246e-00	2.437e-03	2.614e-04
0.75	2.33738e-01	1.241	0.889	2.38044e-01	3.810e-01	2.817e-02
0.80	7.22593e-03	1.449	1.168	7.14426e-03	1.235e-01	1.010e-02

Table 3.9 - Experiment 6: Estimate of Norm of the Inverse

calculated as a reasonable measure of the *dispersion* of the coefficient c within each set. This quantity is tabulated as $3\sigma(c)$. These results clearly confirm our conjecture that the minimum absolute value of the diagonal in the LU decomposition of the input matrix, under full pivoting, serves as an estimator of the norm of the inverse matrix. Although there appears to be a some relationship between the minimum diagonal and inverse norm in the non-pivoting algorithm, it exhibits considerable dispersion in this case and appears to reflect an unreliable dependence on detailed features of the input matrices. There is no question, however, that the pivoting algorithm yields an estimator of manifest value:

The reciprocal of the minimum diagonal is within a small constant factor and within very tight limits equal to the norm of the inverse and this result holds uniformly over the full range of determinant sizes. Moreover, the less expensive LU' algorithm can be used in practice instead of LU' since there is no need for the L^{-1} data.

Chapter 4

Modular Arithmetic

In this chapter we study the arithmetic of Modular Rings. Although this topic is rooted in the classical algebraic tradition, and indeed has truly ancient origins, it has been re-examined by a number of authors over the last few years as a modern topic in Computer Algebra. See for instance [AKR], [DST], and [GCL]. [AKR] is an especially interesting reference for its lengthier treatment of Modular Arithmetic and its interesting historical notes. Moreover, its exposition of the topic is consistent with our own slant toward practical application. In spite of the practical bias we begin the chapter with sections devoted to mathematical (or more precisely, algebraic) theory, a task necessary to secure the foundation for the numerical algorithms of the later sections. In section 1 we introduce the Ring of Modular Classes and its arithmetic and continue with an extension of this theory to the space of Modular Vectors and its arithmetic. In section 2 we discuss the basic algorithms in the ring, and in section 3 the celebrated and fundamental Chinese Remainder Theorem. In section 4 we introduce the concept of projections in the Modulus Vector space, a property of Modulus Vectors which has recently found application and which is an important element in the modular arithmetic formulation of the LU decomposition algorithm. This latter is, of course, our major interest here, and is the subject of section 5.

1. Classes, Rings of Classes, Arithmetic, Modular Vectors

Let $u,p,r,k \in \mathbf{Z}$. Then the set $\{u | u = k \cdot p + r, \forall k\}$ is an *equivalence class* of the integers. We write $r = u \bmod p$, or $u \equiv r(p)$ and say that u is equivalent to r *modulo* p. If in addition we have another integer $y \equiv r(p)$, then we may say that y is equivalent to u *modulo* p, i.e. things equivalent *modulo* p to the same thing are equivalent *modulo* p to each other. The 'thing' u and y are equivalent to, namely r, is a number $0 \le r < p$. We *name*, or denote the class to which u and y belong with the number r. Thus the numbers 74 and -28 are equivalent modulo 17 since $74 = 3 \cdot 17 + 6$ and $-28 = -2 \cdot 17 + 6$, and the class, *modulo* 17, containing 74 and -28 is 6. The latter is one of the classes in $\mathbf{M}_{17}$, namely $\{0, 1, 2, \ldots, 16\}$. To summarize: The mapping $\equiv_p$ induces a partition on the integers onto the set of classes $\mathbf{M}_p$,

$$\equiv_p : \mathbf{Z} \to \mathbf{M}_p \tag{4.1}$$

such that $u = k \cdot p + r \equiv r(p)$ where $u,p,r,k \in \mathbf{Z}$. $\mathbf{M}_p = p$ and the elements of $\mathbf{M}_p$ are denoted by natural numbers r such that $0 \le r < p$. (We will allow r to be a signed number shortly.) We remark that the *class name*, the natural number r, is identical to the special member of the class $\{u \mid u = k \cdot p + r\}$ where $k = 0$. We shall often refer to this special member as the *principal* element or member of the class.

Suppose $u \equiv r(p)$ and $y \equiv q(p)$. Then since $u = k_u \cdot p + r$ and $y = k_y \cdot p + q$, addition yields

$$u + y = (k_u + k_y) \cdot p + (r + q)$$

But $r + q \equiv t(p)$, for some $0 \le t < p$. Hence we have

$$u + y \equiv (u+y)(p) = u(p) + y(p) = r + q \equiv t(p)$$

This result defines an arithmetic homomorphism mapping addition from **Z** onto $\mathbf{M}_p$. We may similarly define an arithmetic homomorphism mapping subtraction from **Z** onto $\mathbf{M}_p$. For instance let $p = 26$. Thus $317 \equiv 5(26)$ and $-76 \equiv 2(26)$. We have $317 - 76 = 241 \equiv 7(26)$ and $317 + 76 = 393 \equiv 3(26)$. Moreover,

$$\begin{aligned} u \cdot y &= (k_u \cdot p + r) \cdot (k_y \cdot p + q) \\ &= (k_u \cdot k_y + k_u \cdot q - k_y \cdot r) \cdot p - p \cdot q \end{aligned}$$

But $p \cdot q \equiv s(p)$ for some $0 \le s < p$ so we have $u \cdot y \equiv s(p)$. For example, using the data above, $317 \cdot (-76) = -24092 \equiv 10(26)$. So there is another arithmetic homomorphism mapping multiplication from **Z** onto $\mathbf{M}_p$. We can summarize these results by writing the homomorphism $\square = \{+, -, \cdot\}$

$$\square_p : \mathbf{Z} \square \mathbf{Z} \to \mathbf{M}_p \square \mathbf{M}_p \tag{4.2}$$

This establishes $\mathbf{M}_p$ as an algebraic *Ring of Classes*. We call $\mathbf{M}_p$ the *Modular Ring*.

The homomorphism of (4.2) supports the following interpretation: Any integer in class $r \in \mathbf{M}_p$ combined under $\square_p$ with any integer of class $s \in \mathbf{M}_p$ yields an integer in class $r \square s \in \mathbf{M}_p$. The rule of composition, $\square_p$, is just

$$(r \square s) \bmod p \tag{4.3}$$

where r and s are integers and integer arithmetic is used.

Division may be added to $\square$ under this interpretation, but doing so requires some special considerations. Division is defined over the integers only when the divisor is a factor in the dividend. But we may define division in $\mathbf{M}_p$, in terms of multiplication, by searching for an element $s^{-1} \in \mathbf{M}_p$ such that division, $r \div s$, is expressed as a multiplication, $r \times s^{-1}$. Such an element is called the *inverse to* s and if it exists we must have $s \times s^{-1} \equiv 1(p)$. Certainly if $p>0$ then $1 \in \mathbf{M}_p$, so we must have

$$t \cdot p + s \cdot s^{-1} = 1 \qquad (4.4)$$

where t is an integer. Thus s^{-1}, if it exists, is a solution, along with t, to the last equation above. A solution does exist so long as p and s are *relatively prime* integers. (See e.g. [IR]) For example, we set $p = 26$ and $s = 5$. These numbers are relatively prime, and so s^{-1} exists in $\mathbf{M}_{26}$ and is equal to 21, which can be found by using the extended GCD algorithm (see section 2, below).

With this result we can extend $\square$ to include division under the interpretation given above: Any element in class r divided by an element in class s is equivalent (belongs to the same class) as the element in class r multiplied by any element in class s^{-1}. Thus for example, given that $114 \equiv 10(26)$ and $317 \equiv 5(26)$, we have $114 \div 317 \equiv (114 \cdot x)(26)$, where x is any integer such that $x \equiv 21(26)$. Let us choose $x = -447$. Then $114 \cdot -447 = -50958 \equiv 2(26)$.

The stipulation above that s^{-1} exists in $\mathbf{M}_p$ if, and only if, s and p are relatively prime

implies that division by s in $\mathbf{M}_p$ is not defined if s divides p. Another way of saying this is that there are no classes in $\mathbf{M}_p$ corresponding to the inverse of the factors of p and so the division by elements of class s is *undefined*. Note that this corresponds to the case of division by zero in ordinary arithmetic. Thus $\mathbf{M}_p$ remains a ring when p is composite. On the other hand, if p is prime then an inverse exists to all $s \neq 0$ in $\mathbf{M}_p$ so $\mathbf{M}_p$ is a *field*, the *finite field* of order p, $\mathbf{F}_p$.

Since $\mathbf{M}_p$ is odd when p is prime, it is convenient to define and use negative elements of $\mathbf{M}_p$. Then the arithmetic of $\square$ of $\mathbf{M}_p$, p prime, is just that of $\mathbf{F}_p$ with elements labeled $-\frac{p-1}{2}, \cdots, 0, \cdots, \frac{p-1}{2}$. The computation of (4.3) is modified to correspond to this usage

$$t = (r \square s) \bmod p$$
$$\text{if} \left(t > \frac{p-1}{2} \right) \text{ then } t = t - p \tag{4.3a}$$

Under these modifications, namely restricting $\mathbf{M}_p$ to prime fields and introducing a formally sound definition of negative numbers, some numerical algorithms may be implemented in Modular Arithmetic. We discuss one of them in section 5, below.

The elements of $\mathbf{M}_p$ are in one to one correspondence with the principal element in each class and we see from the previous discussion that the arithmetic of classes under $\mathbf{M}_p$ is identical to that of the finite fields of prime order whose elements are these principal elements. Hence these elements are integers and the arithmetic operations of the field

are implemented as integer arithmetic which in current machines is done in the integer format using the Arithmetic Processing Unit. For details of this topic the reader should consult section 1 of chapter 3, and in particular the discussion of equation (3.1). The key feature is the finite precision available in Machine Arithmetic. This imposes a significant limitation on the size of modular rings available in practice. Hence we introduce the notion of modulus vector, a vector of classes.

Consider a family of rings, $\mathbf{M}_{p_1}, \mathbf{M}_{p_2}, \cdots, \mathbf{M}_{p_n}$ where the $\{p_i\}$ are prime. Let x be an integer. Then we have $x \equiv r_i(p_i)$, for $1 \le i \le n$. In words, x is simultaneously a member of all the classes $r_i \in \mathbf{M}_{p_i}$, $1 \le i \le n$. This joint membership may be denoted by the list $<r_1, r_2, \cdots, r_n>$ which we shall call the *modulus vector* representation of x. Each component of this vector denotes the equivalence class of the corresponding ring in which x resides. We write $\mathbf{M}^n$ to denote the intersection set $\{\mathbf{M}_{p_1} \cap \mathbf{M}_{p_2} \cap \cdots \cap \mathbf{M}_{p_n}\}$. Then $M = |\mathbf{M}^n| = \Pi_i p_i$ and a modulus vector denotes a class in $\mathbf{M}^n$. Any $x \in \mathbf{Z}$ is a member of some class in $\mathbf{M}^n$: $x \equiv x_0(M)$ where x_0 is the principal element in the class $<r_1, r_2, \cdots, r_n>$. Clearly we have $r_i = x_0 \bmod p_i$. The inverse computation, finding the x_0, the principal element, when the r_i are given is the subject of the Chinese Remainder Theorem, for which see the next section of this chapter. But looking ahead, we can compute

$$x_0 = (r_1 \cdot q_1 + r_2 \cdot q_2 + \cdots + r_n \cdot q_n) \bmod M \tag{4.5}$$

where the numbers q_i depend only on the structure parameters p_i. Compare (4.5) with

(3.1) which concerns the evaluation of an integer given an ordered list of digits.

The q_i have the interesting property that

$$q_i \equiv \begin{cases} 1(p_k), & k = i \\ 0(p_k), & k \neq i \end{cases} \tag{4.6}$$

This implies that the q_i may be considered the *unit* modulus vectors, $\hat{\boldsymbol{i}}$, (e.g. $\hat{\mathbf{1}} = <1,0,\cdots,0>$ etc.) in a vector space over $\mathbf{F}_{p_i}$ of modulus vectors. This immediately gives us the arithmetic homomorphism $\pm: \mathbf{Z} \to \mathbf{M}^n$ for addition and subtraction: if $x, y \in \mathbf{Z}$, $x \equiv <r_1, r_2, \cdots, r_n>(M)$ and $y \equiv <s_1, s_2, \cdots, s_n>(M)$ then $x \pm y \equiv <r_1 \pm s_1, r_2 \pm s_2, \cdots, r_n \pm s_n>(M)$. The operations on the components of the modulus vectors are those of the corresponding fields $\mathbf{F}_{p_i}$. Moreover, the unit modulus vectors have the further multiplicative property that

$$\hat{\boldsymbol{i}} \cdot \hat{\boldsymbol{j}} = \begin{cases} \hat{\boldsymbol{i}}, & j = i \\ \mathbf{0}, & j \neq i \end{cases} \tag{4.7}$$

This gives us the multiplication homomorphism $\cdot: \mathbf{Z} \to \mathbf{M}^n$: $x \cdot y \equiv <r_1 \cdot s_1, r_2 \cdot s_2, \cdots, r_n \cdot s_n>(M)$. This behaves exactly as the inner, or 'dot', product of vector spaces. The effect of all this is that arithmetic in the modulus vector ring corresponds to that in the scalar modulus ring, the arithmetic on vectors being carried out component-wise according to each component field. Since the components are elements of finite fields, division of vectors is defined through the multiplicative homomorphism of (4.7).

Two additional and significant remarks are due on the arithmetic of modulus vectors. First, since a vector may have zero components, division by such vectors will yield undefined components in the quotient vector. The vector ring will have many of these non-invertible elements. They require special treatment in applications and we will have more to say on the subject later in this chapter. The second observation highlights the fact that the arithmetic operations are component-wise and independent of one another. This feature supports a highly attractive opportunity for computational parallelism.

We will close this section with a few examples of modulus vector arithmetic. Consider the vector ring $\mathbf{M}^3 = \{\mathbf{F}_{17}, \mathbf{F}_{23}, \mathbf{F}_{29}\}$. Then $M = 11339$. The integer 2356 is written as the vector <-7, 10, 7> in this ring and -1613 is written as <2, -3, 11>. The difference of these two numbers 2356-(-1613) or $\langle (-7-2)_{17}, (10-(-3))_{23}, (7-11)_{29} \rangle = \langle 8, -10, -4 \rangle$. Their product, 2356·-1613 = -1663(11339) is represented as the vector <3, -7, -10>. The number 460 is represented as the vector <1, 0, -4>, and so the quotient of 2356÷460 is <-7, *, -9>, where the asterisk indicates an undefined component.

2. Elementary Algorithms

Clearly the modular arithmetic operations we have introduced must be executable on existing machines. Concerning Euclidean Division, i.e.(4.3), it is a fact that given any two integers x and y there are two other integers p and r such that $y = p \cdot x - r$, but we can ask under what conditions can these numbers actually be found using any computing

tool. On a machine using integer arithmetic, i.e (3.1), and assuming that x and y are writable, then p and r can be computed exactly by repeated subtraction using the algorithm of Figure 4.1, where it is expressed in the style of the C language. The process complexity of this algorithm is $O(y)$, or approximately p. On machines with floating point arithmetic, i.e. (3.2) and (3.3), we can compute p and r by using the algorithm of Figure 4.2. The process complexity of Euclidean Division with the floating point algorithm is O(1). Both x and y must be writable within the floating point mantissa, that is, the number of digits in the inputs to the algorithm must not exceed the w of (3.2). On many machines the integer type is designed to fit well within the width of doubles, so this requirement is normally met automatically. Note that the code of Figure 4.2 is designed to be compatible with that of Figure 4.1; it takes integer arguments and returns integer results. Thus floating point truncation errors will not be a problem.

We remark that the Figure 4.2 code is almost never seen explicitly since its arithmetic operations are always implemented as 'built-in' functions in higher level programming languages. We should also remark that had we implemented the Figure 4.2 algorithm to accept floating point arguments and return floating point results the problem of floating point errors would be a greater concern, but clearly if the integer parts of the input arguments were less than b^{w-1} in absolute value then the algorithm could be implemented so that the results would be safe from floating point errors. However, our aim is to work exclusively with 'small' elements in this chapter on modular arithmetic.

```
void DIV (int y, int x, int *p, int *r)
{
     *p = 0;
     *r = abs(y);
     if (y == 0) {*r = 0; return;}
     if (x == 0) {return;}
     while (*r >= 0) {
          *r -= abs(x);
          *p += 1;
          }
     if (y > 0) {
          *r -= abs(x);
          *p -= 1;
          if (x < 0) *p = -(*p);
          }
     else {
          *r = -(*r);
          if (x > 0) *p = -(*p);
          }
     return;
}
```

Figure 4.1 - Integer Euclidean Division

and so we just avoid this issue. Finally, note that the programs in Figures 4.1 and 4.2 are designed to return positive remainders; if negative remainders are desired then these codes could be supplemented with the use of (4.3a).

```
void DIVf (int y, int x, int *p, int *r)
{
     double s;
     if (y==0) {*p=0; *r=0; return;}
     if (x==0) {*p=0; *r=y; return;}
     s = floor((double)y/(double)x);
     if (s>=0.0) {
          if (y>=0) {*p = (int)s;}
               else {*p = (int)(s - 1.0);}
     } else {
          if (y<0) {*p = (int)s;}
               else {*p = (int)(s - 1.0);}
     }
     *r = y - (*p)*x;
}
```

Figure 4.2 - Euclidean Division in Floating Point

Clearly the floating point version is the less costly division algorithm and so would be used wherever floating point is available. We remark that the computational efficiency of Euclidean division is important for our work since it is the core arithmetic operation in the extended greatest common divisor algorithm.

This algorithm implements division of modulus vectors. The algorithm solves the equation $p \cdot y - q \cdot x = d$, where x and y are given integers and p and r are integers to be found, as is d, the greatest common divisor of x and y. That solutions exist is a fact of number theory; our task is to discover an effective method of actually finding them. A suitable procedure turns out to be the following elaboration of the classical gcd algorithm. Let $\{r_i\}$, and $\{s_i\}$ denote sequences of integers, and set $r_1 = y$, and $r_2 = x$ Then we compute the recursive system of Euclidean divisions:

$$\begin{aligned} r_0 &= s_1 r_1 - r_2 \\ r_1 &= s_2 r_2 - r_3 \\ &\vdots \\ r_{k-1} &= s_k r_k - r_{k+1} \\ &\vdots \\ r_{n-1} &= s_n r_n - r_{n+1} \\ r_n &= s_{n+1} r_{n+1} - 0 \end{aligned} \tag{4.8}$$

Recursion stops when the remainder at any step, in particular at step $n+1$, equals zero. Clearly this must happen since the $\{r_i\}$ is a monotonically decreasing sequence of strictly positive integers. Hence $r_{n+1} \mid r_n$, and recursively we have $r_{n+1} \mid r_i \ \forall\, 1 \le i \le n$. In particular $r_{n+1} \mid r_1$ and $r_{n+1} \mid r_0$. Thus r_{n+1} is a common divisor of x and y. Moreover, no other common divisor could be smaller, since $r_{i+1} < r_i$. Hence $r_{n+1} = d$ is the gcd of

x and y. This much is just the 'standard' Euclidean gcd algorithm. But if we set $p_n = 1$ and $q_n = -s_n$, then we can put $d = p_n r_{n-1} + q_n r_n$ as the penultimate line in (4.8). Now, suppose we can climb the recursive ladder in (4.8) so that at the $k+1^{th}$ step we have

$$d = p_{k-1} r_k + q_{k-1} r_{k-1}$$

So if we replace r_{k-1} in this expression with the next rung up the ladder, the k^{th} step in (4.8), we would get

$$\begin{aligned} d &= p_{k-1} r_k + q_{k-1}(r_{k+1} - s_k r_k) \\ &= q_{k-1} r_{k+1} + (p_{k-1} - q_{k-1} s_k) r_k \end{aligned}$$

But by setting

$$\begin{aligned} p_k &= q_{k-1}, \quad \text{and} \\ q_k &= p_{k-1} - s_k q_{k-1}, \end{aligned}$$

we have

$$d = p_k r_{k+1} - q_k r_k$$

as the next rung. Ascending to the top we find $d = p_1 r_0 - q_1 r_1 = p_1 y - q_1 x = py - qx$. Thus we have computed the d, q, and p and have solved the equation.

An implementation of this procedure is given below in Figure 4.3. The function `DIV(y, x, &p, &q)` is an implementation of Euclidean division. The process compexity of the extended gcd is $O(n)$, that is, linear in the depth of the recursion in (4.8). This is usually not very great, depths of 5 or more being rare. To see why this algorithm figures so prominently in our work consider the case where y is a prime. Then

```
int E_GCD(int y, int x, int *p, int *q)
{
      int r, s, t, u;
      DIV(y, x, &s, &r);
      if ( r == 0 ) {
            *p = 0;
            *q = 1;
            return x;
            }
      else {
            r = E_GCD(x, r, &t, &u);
            *p = u;
            *q = t - s*u;
            return r;
            }

}
```

Figure 4.3 - Extended GCD Algorithm

$d = 1$, and we have $p \cdot y - q \cdot x = 1$. But this is the equivalence $q \cdot x \equiv 1(y)$, i.e. $q \equiv x^{-1}$ in the ring $\mathbf{M}_y$. Since we implement division in the ring by using multiplication of the inverse, it is clear that the extended gcd algorithm is a key component of the arithmetic of modular rings and modulus vectors.

3. The Chinese Remainder Theorem

The Chinese Remainder Theorem establishes the connection between integer numbers and their Euclidean remainders modulo a set. Equation (4.5) expresses this as an evaluation of the principal element corresponding to a modulus vector, which we repeat here (recall that a modulus vector is an ordered list of moduli)

$$x_0 = (r_1 \cdot q_1 + r_2 \cdot q_2 + \cdots + r_n \cdot q_n) \bmod M \tag{4.5}$$

where $<r_1, r_2, \cdots, r_n>$ is a modulus vector, $M = |\mathbf{M}^n|$, and the set $\{q_1, q_2, \cdots, q_n\}$ is to be determined. $\mathbf{M}^n = \{\mathbf{F}_{p_1}, \mathbf{F}_{p_2}, \cdots, \mathbf{F}_{p_n},\}$ is the modulus vector ring. The $\{\mathbf{F}_{p_i}\}$ are the finite basis fields of the ring, which are of prime order p_i. Recall that $M = \Pi_i p_i$. x_0 is the principal element of the class in $\mathbf{M}^n$ denoted by the given vector $<r_1, r_2, \cdots, r_n>$. if $x \equiv <r_1, r_2, \cdots, r_n>(M)$, then $x = k \cdot M + x_0$. Hence $x \equiv x_0 \equiv r_i(p_i)$.

Consider the numbers $\left\{m_i = \frac{M}{p_i}\right\}$. Then p_j divides m_i for $1 \le j \ne i \le n$ so that $m_i \equiv 0(p_i)$. On the other hand p_i and m_i are mutually prime so that the gcd of p_i and m_i is 1. Hence we can write the equation $s \cdot m_i + t \cdot p_i = 1$, which implies that $s \cdot m_i \equiv 1(p_i)$. This fact can be written as the modular equivalence $m_i \equiv s^{-1}(p_i)$.

Let us now compute an integer, y, with the following sum

$$y = s_1 \cdot m_1 \cdot r_1 + s_2 \cdot m_2 \cdot r_2 + \cdots + s_n \cdot m_n \cdot r_n$$

where the $\{s_i\}$ are computed such that $s_i \cdot m_i \equiv 1(p_i)$ as above using the egcm algorithm. Then we have

$$s_i \cdot m_i \equiv \begin{cases} 1(p_k) & k = i \\ 0(p_k) & k \ne i \end{cases}$$

That is, the $s_i \cdot m_i$ are the principal elements of classes corresponding to the unit vectors in $\mathbf{M}^n$. Thus we will get $y \equiv r_i(p_i)$ which implies that $y \equiv <r_1, r_2, \cdots, r_n>(M)$. This in turn implies that y is a member of the class in $\mathbf{M}^n$ denoted by the vector $<r_1, r_2, \cdots, r_n>$ whose principal element is $x_0 = y \bmod M$. If we put $q_i = s_i \cdot m_i$ we have equation (4.5).

We display the example $\mathbf{M}^n = \{\mathbf{F}_{17}, \mathbf{F}_{23}, \mathbf{F}_{29}\}$ in Table 4.1. Using this table with input vector $< -7, 10, 7 >$ yields the calculation

$$\begin{aligned} -7 \cdot -2668 &= 18676 \\ 10 \cdot 3451 &= +34310 \\ 7 \cdot -782 &= -5474 \\ &\text{----------} \\ &47712 \end{aligned}$$

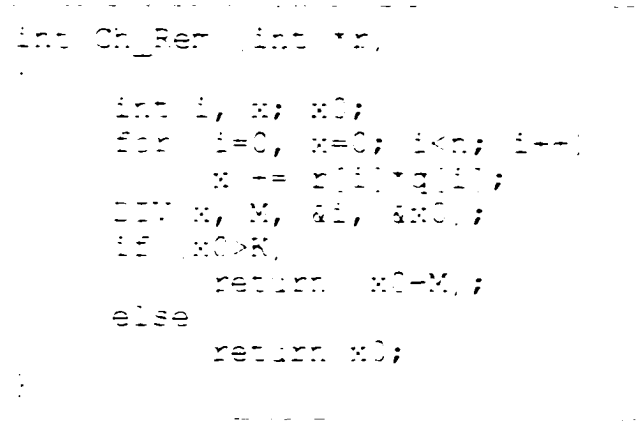
i	p_i	m_i	s_i	q_i
1	17	667	-4	-2668
2	23	493	7	3451
3	29	391	-2	-782

Table 4.1- $\mathbf{M}^3 = \{17, 23, 29\}$

And we have $47712 \equiv 2356 (11339)$ from Euclidean division. Recall that we computed the modulus vector $<-7, 10, 7>$ from 2356 at the end of section 1.

Several algorithms may be derived from (4.5) designed to meet various computational objectives. One such is the desirability of supporting *negative* principal elements of rings. Principal members, computed by (4.5) for instance, are the result of Euclidean division and we have, so far, required these results to be positive integers. Recall that we have already studied the support for negative elements in modular rings, see the discusion surrounding equation (4.3a) in section 1. Negative principal members are supported in like manner.. We require the principal element, x_0, to lie in the range $-\frac{M-1}{2} < x_0 \leq \frac{M-1}{2}$, so that when the result of computing (4.5) is greater than $\frac{M-1}{2}$ then M is subtracted yielding a negative result.

An implementation of (4.5) in the **C** language idiom is given in Figure 4.4. The modulus vector r is passed into the function as an argument. An external table, such as Table 4.1, supplies the array q[] as well as the parameters of the ring $\mathbf{M}^n$, n, M, and $K = \frac{M-1}{2}$.



```
int Ch_Rem (int *r)
{
     int i, x; x0;
     for (i=0, x=0; i<n; i++)
          x += r[i]*q[i];
     DIV x, M, &i, &x0);
     if (x0>K)
          return x0-M;
     else
          return x0;
}
```

Figure 4.4 - Chinese Remainder-1

Note that the variable 'i' serves merely as a dummy argument in the DIV call. A minor variation of this algorithm, shown in Figure 4.5, takes as its argument the vector ring product $<r_1,\cdots,r_n> \cdot <s_1,\cdots,s_n>$ where $<s_i>$ is the s_i column of the ring parameter table, considered as a modulus vector. This variant is useful in applications where it is desirable to maintain intermediate operands in modulus vector form to the greatest extent possible, delaying data conversion to integer form for output display only. The process complexity of both algorithms is O(n), neither one has a compelling cost advantage. Both feature exact arithmetic.

```
int Ch_Rem (int *r)
{
     int i, x; x0;
     for (I=0, x=0; i<n; i++) {
          x += r[i]*m[i];
          if (x<K) x += M;
          if (x>K) x -= M;
          }
     return x;
}
```

Figure 4.5 - Chinese Remainder-2

A version of (4.5) using floating point arithmetic is given in Figure 4.6. This algorithm is based on the following real domain modification of (4.5)

$$y = M\cdot\left\{\left[\frac{s_1}{p_1}\cdot r_1 - \frac{s_2}{p_2}\cdot r_2 - \cdots - \frac{s_n}{p_n}\cdot r_n\right] \bmod 0.5\right\} \tag{4.5a}$$

where $-\frac{1}{2} \le \frac{s_i}{p_i} \le -\frac{1}{2}$, and the quantity within the braces is the inner bracketed sum *modulo* 0.5, so it lies in the range [-0.5, 0.5]. The result, y, is the *floating point* product of M with the remainder in braces. This computation has the advantage of floating point speed coupled with the fact that floating point machine precision is greater than the integer precision. The argument to this variant is the modulus vector product $<r_1,\cdots,r_n> \cdot <s_1,\cdots,s_n>$ as in Chinese Remainder-2. The process complexity of this variant is also O(n). However floating point roundoff errors are present. For (4.5a) the absolute value of this error will be bounded by

$$M\cdot(3\cdot n-1)\cdot\epsilon \tag{4.9}$$

in view of the floating point error formula (3.4) (cf. BEPP97]). This bound is

```
double Ch_Rem (int *r)
{
     int i;
     double x;
     for (i=0, x=0.0; i<n; i++)
          x+=((double)r[i]/(double)p[i]) {
          if (x>0.5) x -= 1.0;
          if (x<-0.5) x += 1.0;
     {
     return (double)(M*x);
}
```

Figure 4.6 - Chinese Remainder-3

$$M\cdot(2n-1)\cdot\epsilon$$

for the algorithm in Figure 4.6, since the the numerators, r_i, have been precomputed in this version.

We offer a digression here, illustrating modulus vector arithmetic in $\mathbf{M}^3 = \{\mathbf{F}_{17}, \mathbf{F}_{23}, \mathbf{F}_{29}\}$ by way of an application which prepares the argument for Chinese Remainder-2 or -3. Let P_i be the modulus vector corresponding to p_i, that is, the i^{th} component of this vector is just $p_i \bmod p_i$. The i^{th} component of P_i is zero, of course. Then in $\mathbf{M}^3$ we have

$$P_1 = <\ 0, -6, -12>$$
$$P_2 = <\ 6,\ 0,\ -6>$$
$$P_3 = <-5,\ 6,\ \ 0>$$

Now we compute the vector products $Q_i = \Pi_{j\neq i} P_j$. Only the i^{th} component of Q_i is non-zero...

$$Q_1 = <4,\ 0,\ 0>$$
$$Q_2 = <0, 10,\ 0>$$
$$Q_3 = <0,\ 0, 14>$$

Q_i is, in fact, the modulus vector corresponding to m_i and we could as well have computed Q_i directly by observing that the j^{th} component of Q_i is just $m_i \bmod p_j$. But the m_i are large numbers and we have expressed our interest in avoiding computing with large numbers and our present desire is to show by this example how modular arithmetic makes this possible. In any case the components $q_{i,j}$ of the vector Q_i are

$$q_{i,j} = \begin{cases} q_{i,j} & \text{for } i = j \\ 0 & \text{otherwise} \end{cases}$$

and the vector Q_i is $q_{i,i}\cdot\hat{i}$, cf. (4.6). Now, forming the vector sum $S = \Sigma_i Q_i$ we can observe that the components of the vector inverse S^{-1} are just the s_i needed in tables, like Table 4.1, used in the Chinese Remainder computation. Indeed, in $\mathbf{M}^3$

$$S^{-1} = < -4, 7, -2 >$$

Thus we enter the Chinese Remainder functions with the argument $\frac{x}{S}$.

Our project to avoid using large numbers comes to a problem, of course, within these programs since we must do multiplications in them using the operands m_i in the case of Chinese Remainder 2 or M in the case of Chinese Remainder 3. There, unless further refinements are added concerning the representation of large numbers, we confront the limitations of machine operand size. But the Chinese Remainder computations are *output* functions processing the tail end of other computations which can be done exactly to arbitrary precision in modular arithmetic using only small numbers throughout.

4. Projections in the Vector Ring, Knockouts

Let $x \equiv x_0(M)$ where $M = |\mathbf{M}^n|$ as in the preceding section. Then we can say the following: x belongs to the class in $\mathbf{M}^n$ containing the principal element x_0. This class is denoted by the modulus vector $<r_1,\cdots,r_{i-1},r_i,r_{i-1},\cdots,r_n>$ where $r_i = x_0 \bmod p_i$. Given x we can compute the $\{r_i\}$. On the other hand given $<r_1,\cdots,r_{i-1},r_i,r_{i-1},\cdots,r_n>$ we can

compute x_0 using (4.5) or one of the algorithms Chinese Remainder-*. We have that $x_0 \le \frac{M-1}{2}$.[6] But suppose that x_0 is less than $\frac{m_i - 1}{2}$, where we recall that $m_i = \frac{M}{p_i}$. Then we can write $x \equiv x_0 (m_i)$ since $m_i = \Pi_{j \neq i} p_j$. We have, in fact, that $x = k \cdot p_i \cdot m_i + x_0$. This suggests that we may view x as a member of a class in a *new* vector ring $\mathbf{M}_i^{n-1}$ where $\mathbf{M}_i^{n-1} = m_i$ and whose principal element is still x_0. This class in $\mathbf{M}_i^{n-1}$ is denoted by the modulus vector $<r_1, \cdots, r_{i-1}, -, r_{i+1}, \cdots, r_n>$, the same as the old vector in $\mathbf{M}^n$ but with the i^{th} component 'knocked out'. In words, $\mathbf{M}_i^{n-1}$ is the subring of $\mathbf{M}^n$ projected along the component $\mathbf{M}_{p_i}$ containing all the classes of $\mathbf{M}^n$ such that $x_0 \le \frac{m_i - 1}{2}$.

Clearly, given x_0, the components of the shortened vector are obtained, as usual, by computing $x_0 \bmod p_j$, $j \neq i$. Also we can compute x_0 from the knock-out vector $<r_1, \cdots, r_{i-1}, -, r_{i+1}, \cdots, r_n>$ by using (4.5). However, the $\{q_j\}$ used in this computation must be recomputed -- a new table is constructed reflecting the status of $\mathbf{M}_i^{n-1}$ as a projection.

i	p_i	m_i'	s_i'	q_i'
1	17	29	-7	-203
2	29	17	12	204

Table 4.2- $\mathbf{M}^2 = \{17, 29\}$

We put $m_j' = \frac{m_j}{p_i}$ for $j \neq i$ and then the $\{s_j\}$ must be recalculated so that $s_j' \cdot m_j' \equiv 1 (p_j)$. Table 4.2 shows the necessary modifications for the projection $\mathbf{M}_2^2 = \{\mathbf{F}_{17}, \mathbf{F}_{29}\}$ of the ring $\mathbf{M}^3$ of Table 4.1. This corresponds to knocking out the component $\mathbf{F}_{23}$. We note that $|\mathbf{M}_2^2| = m_2 = 493$.

---

[6]Recall that M, m_i, etc, are products of prime numbers and hence odd unless one of the primes is 2. Generally 2 is not used and we can consider it "inconvenient".

Consider the example $x_0 = -213$. This corresponds to the modulus vector $<8, -6, -10>$ in $\mathbf{M}^3$. In this case $x_0 < \frac{m_2 - 1}{2}$. So we can knock out the second component of the modulus vector to obtain $<8, -10>$ as the projection onto $\mathbf{M}_2^2$. Now we can evaluate this shortened modulus vector using (4.5) or (4.6) in conjunction with the data of table 4.2 to check that $x_0 = -213$. As another example we use the principal member 2356 in $\mathbf{M}^3$ which as we have seen above corresponds to the modulus vector $<-7, 10, 7>$. In this case, however, $x_0 > \frac{m_2 - 1}{2}$, so we must expect to loose something in the projection. So knocking out the second component and recalculating we find that the projected vector $<-7, 7>$ turns out to correspond to the principal member -109 in $\mathbf{M}_2^2$. However, 2356 *is* a member of the class whose principal member is -109, since $2356 = 5 \cdot 493 - 109$.

We briefly continue the digression of the previous section on modular arithmetic by observing that the entries in the s_i' column in the knock-out Chinese Remainder tables (as Table 4.2) may be obtained as the modulus vector product of S^{-1} with the modulus vector P_k, where k indexes the component to be knocked out. Thus, in the present example, we need $P_2 \cdot S^{-1}$ which we can verify by doing the calculation

$$<6, 0, -6> \times <-4, 7, -2> = <-7, 0, 12>$$

Knock-out finds application in two important areas. First, knock-out may be used to remove undefined components in a modulus vector during the Chinese Remainder evaluation of its integer principal element. If the absolute value of the latter number is

known to be bounded and this bound is less than the m_i corresponding to an undefined component in the vector then the principal element can be recovered using the knock-out reduction method described above. This is exactly case in our application where we first attempt to compute a determinant using floating point numerical methods and resort to modular arithmetic only when the error in this computation turns out to be too large. Here the result, even with errors, bounds the absolute value expected and can be used to define the modular vector space used in the subsequent exact determination. Prudence suggests making allowance in this definition to provide for the possibility of undefined components being generated in the modular LU factorization, see the next section for a discussion of that topic. A second application of knock-out is used to reduce the floating point errors arising in the Chinese Remainder evaluation (4.6). A deliberate knock-out has the effect of reducing the modulus, M, in that expression, and a glance at the error estimate expression (4.9) will confirm that the resulting effect will be to reduce the expected error in the evaluation. Of course, effective use of this technique also requires some prior knowledge of the magnitude of the result. This application is illustrated in Table 4.3 where a modulus vector in $\mathbf{M}^5 = \{ \mathbf{F}_{1009}, \mathbf{F}_{1013}, \mathbf{F}_{1019}, \mathbf{F}_{1021}, \mathbf{F}_{1031} \}$ whose principal element is 379 is evaluated while successively knocking out components. The column labeled 'n' counts the number of components knocked out.

n	Value
0	379.0185313112108
1	378.9998777556075
2	378.9999999388447
3	379.0000000000029
4	379.0000000000000

Table 4.3 - Effect of Knock-out

5. LU Decomposition Algorithm using Modular Arithmetic

We said in earlier remarks that many numerical algorithms can be implemented in Modular arithmetic. Many others cannot, of course. (See chapter 6 for attractive research initiatives along these lines.) but one numerical algorithm that can be nicely converted to modular arithmetic is the LU matrix decomposition algorithm. Figure 4.7 shows the modular version in the C language style. Little needs to be said about this; it is virtually identical to the LU algorithm shown in Figure 3.1, the only change is the use of modulus vectors (the Mod_Mtx declarations) as operands in the implementation, and, of course, the modulus vector arithmetic operations. This is really just an example of 'polymorphism' in object-oriented programming parlance.

The comment concerning pivoting highlights the single unique problem that can arise in this application of modular arithmetic. As we saw above divisions by modulus vectors containing zero components will result in the generation of undefined components in quotients. The LU algorithm features divisions in the computation of the pivot

```
Mod_Mtx LU (Mod_Mtx x)
/*  LU decomposition  */
/*  returns the decomposition of x */
/*  input matrix, x, is preserved */
/*  11 Nov 96:  pivoting preprocessing not
implemented */
{
     int i, j, k, dim = Dim(x);
     Mod_Mtx y(x);
     for (k=0; k<dim; k++) {
          for (i=k+1; i<dim; i++) {
               y(i,k) = -(y(i,k)/y(k,k));
               for (j=k+1; j<dim; j++)
                    y(i,j) = y(i,j) + y(i,k)*y(k,j);
          }
     }
     return y;
}
```

Figure 4.7 - Modular LU Decomposition

elements. This topic was much discussed in chapter 3, and there the problem was seen as division by zero, or even 'small' operands. The consequence of this event was to introduce instability in the floating point results. This problem is present but is manifested differently in the modular version. In effect, instability is a 'crisp' phenomenon here, the algorithm either generates undefined components, or it does not. If an undefined component is generated in a pivot element, then all results in the right hand square below this pivot will contain the same undefineds. Note that a pivoting scheme, patterned after that used in the floating point algorithms (see e.g. Figure 3.5) and designed to avoid the generation of undefined components, could not avoid any such components being generated in $u_{n,n}$. Moreover, the consequences would be the same in this latter case as those deriving from any undefineds generated earlier in the factoring

process. For this reason the extra cost of a pivoting procedure does not appear justified. Hence the comment in the code.

However, the generation of undefineds by no means spells disaster in the modular arithmetic case, as it does in floating point domain. Indeed, there are strategies available which can make the occurrence of input matrix elements having zero components unlikely and to soften the effect if they do occur. Since a zero component implies that a principal ring element is a multiple of the degree of a basis field in the vector ring, using basis fields of fairly large degree would reduce the probability of that happening. And increasing the number of basis fields in the ring (increasing the dimension of the vector ring) would increase the likelihood of successfully applying knock-out to eliminate the consequences of undefineds in the results. Note especially that both of these strategies are available prior to and independent of the execution of the modular LU algorithm.

Let the modular LU algorithm operate on elements in $\mathbf{M}^m$. Then the process complexity of the algorithm in this domain is $\frac{m}{3}\cdot n^3 + O(n^2)$ (see Table 3.1), that is m times the cost of the floating point version. Of course the modular operations are being counted in this expression and their additional cost can significantly effect the execution time of the modular case. The factor m is an important number in applying the strategies of the paragraph above. But recall that the arithmetic operations in the ring are component-wise in the vector and thus may be done, at least potentially, in parallel. This would eliminate the factor m, bringing the process cost down to the machine floating point

case. We discuss this attractive possibility again in chapter 6.

Finally we offer an example of LU decomposition in modular arithmetic. The matrix below is taken from one of the data sets used in the experiments of chapter 5.

$$\begin{bmatrix} -26085 & -114752 & -24 & 60672 & 5080 \\ 117533 & 145857 & 884 & -171619 & -146386 \\ -75942 & -216371 & -1288 & 177628 & 211880 \\ 235 & 672 & 4 & -551 & -658 \\ 178309 & -258918 & -1532 & 99849 & 249854 \end{bmatrix}$$

The determinant computed from this matrix by floating point LU and the subsequent multiplication of the diagonal elements of U is 75649.28268318021. However the distance of this matrix from a singular matrix is estimated to be ≈ 4.666, considerably less than the required certification threshold of $\bar{\mathbf{H}} > 100$. (See below, chapter 5. for an explanation and all the details...) Thus we conclude that there is sufficient error in this floating point result that a recalculation in modular arithmetic is necessary to guarantee reporting the correct sign for this determinant. Clearly the correct value for the determinant cannot exceed $\approx 159{,}000$, say, so we choose to re-evaluate the determinant using LU and diagonal multiplication in the modular vector ring $\mathbf{M}^2 = \{\mathbf{F}_{1009}, \mathbf{F}_{1013}\}$. $M = 1{,}022{,}117$ in this case. The recomputed determinant, recovered from the modular result, is 1279.999999999944. The correct value, known in this case, is ± 1280. The determinant is certified to be positive!

Chapter 5

Data Generation and Experimental Studies

This chapter is devoted to finding an effective certification predicate, $\Psi(*,*)$, with the aid of computational experiments. We describe our experimental methods including details on generating data for the experiments. We present our experimental results. We conclude that a practical predicate can be based on an estimate of the proximity of a matrix to a singular matrix and that this estimate can be reliably computed from the results of the LU factorization of the matrix. This predicate is just the boolean expression $\hat{H} > \bar{\mathbf{H}}$, where $\hat{H}$ is the estimated distance to singularity and $\bar{\mathbf{H}}$ is a threshold value which we establish experimentally to be ≈ 100 in units of machine precision.

1. Data Generation and Sources

Recall that our study concerns a data space consisting of square dense matrices filled with integer elements. We will generate matrices for numerical experiments by the simple method of populating them with elements drawn randomly from a uniform distribution of integers over a range $[-\alpha,\alpha]$. We have discussed the properties of matrices of this type in Chaper 2 where we observed that they tend to have 'large' determinants. More precisely we argued that the probability of a matrix with a 'small' determinant occuring in this random generating process is exceedingly small. We justified this observation experimentally in Figure 2.5 and with the additional geometric argument that 'small' determinants correspond to points in an embedding n-dimensional

space lying 'close' to convex hulls, a vanishingly small subspace. An inspection of the algebraic definition, (2.1), will lead to a similar conclusion on probabilistic grounds: The definition is a sum of products of uniformly distributed random variables and the resulting distribution, a sum of joint distributions, corresponds exactly to the contents of Figure 2.5.

Thus our data space is hree dimensional, spanned by n, the dimension of matrix A, its determinant, det(A), and α, the maximum absolute value of the elements of A. Sample matrices constructed by just filling them with integers from a random distribution restricts our study to a limited and uninteresting region of this data space. The really interesting regions are just those pertaining to the 'small' determinants, at many values of α and n, and thus not accessible by this simple data generating scheme. We need a general method of generating matrices of known, small, determinants.

Such a method is available, suggested first by the author's colleague Steve Yu. Consider two square matrices of, respectively, lower triangular and upper unit triangular form

$$S_l = \begin{bmatrix} d_1 & 0 & 0 & \cdots & 0 \\ l_{2,1} & d_2 & 0 & \cdots & 0 \\ l_{3,1} & l_{3,2} & d_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & l_{n,3} & \cdots & d_n \end{bmatrix} \quad \text{and} \quad S_r = \begin{bmatrix} 1 & r_{1,2} & r_{1,3} & \cdots & r_{1,n} \\ 0 & 1 & r_{2,3} & \cdots & r_{2,n} \\ \vdots & 0 & 1 & \cdots & r_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \tag{5.1}$$

Then their product $S = S_l \cdot S_r$ will be a dense matrix such that $\det(S) = \prod_i d_i$. That is,

$\det(S)$ will be known and independent of the $\{l_i\}$ and the $\{r_i\}$. These latter may be simply drawn from a distribution. Additional regions of the data space may be sampled by permuting the rows and columns of the matrix S. Further expansion of the reachable space may be obtained by observing that the $\{d_i\}$ are arbitrary so long as their product is the desired determinant.

This method of generating data imposes some constraints on our experimental designs, however. As mentioned above one of the dimensions of the data space is α, the absolute value of the largest element in a matrix, and this is one of the parameters we wish to control in the experimental data. On the other hand the primary objective in using (5.1) to generate experimental data is to control the magnitude of the determinants in these sets. These two simultaneous requirements impose constraints on the range of the uniform distributions available in the random selection of the $\{l_i\}$ and $\{r_i\}$, namely

$$q \le \frac{\alpha}{d_1}, \quad \text{and} \quad q \le \sqrt{\frac{\alpha - d_n}{n-1}} \tag{5.2}$$

where the variables $\{l_i\}$ and $\{r_i\}$ are to be chosen randomly from a uniform distribution over the range $[-q, q]$. We must choose q according to the smaller of the constraints in (5.2) otherwise the elements in the matrix S will exceed α. Note also that $\det(S) = \prod_i d_i$ and $1 \le d_i \le \alpha$ are fundamental constraints in (5.1).

Matrices generated from (5.1), under the constraints (5.2), will display some structural

artifacts arising from these constraints. Matrices having 'large' determinants will tend to have small off-diagonal elements. This may cause trouble in the subsequent analysis computations and will also effectively restrict the reachable portion of the data space. On the other hand, since these domains correspond to 'large' determinants, we are not especially interested in their properties in any case. However, the matrices with 'small' determinants will also display artifacts arising from (5.2). Since the variable elements of S are products of the factors $|l_i|$ and $|r_i|$ drawn randomly from a uniform distribution, the resulting distribution of these variable elements tends more to a χ^2 type. As a consequence matrices of the 'small' determinant class will favor somewhat smaller maximum elements than would be expected in a uniform distribution and this effectively moves the intended test point in data space slightly toward the 'large' determinant regions. Another artifact associated with 'small' determinant matrices is the enhanced possibility of computing zero determinants for non-singular matrices. Since this is a product of catastrophic cancellation, the full pivoting LU algorithm is particularly exposed to this phenomenon. But, since this is just a specialized example of certification failure, we accept it in our data sets. Nevertheless, bogus singularities require special handling in data reduction computations.

We design experiments according to the following considerations. Given a choice of matrix dimension and maximum matrix element we can select a determinant value such that $1 \le \det(S) \le \text{MaxMeasure}$. We concentrate on the 'small' determinant region of the data space and focus in particular on the transition from 'large' to 'small' determinants

where certification begins to fail. We select a series of fixed determinant values spanning an interesting region of the data space for a study.

We define determinant values according to a 'log relative' measure. That is

$$\rho = \frac{\log \det(S)}{\log M} \tag{5.3}$$

where M is the Max Measure, introduced in chapter 2, for matrices corresponding to the chosen parameters n, and α:

$$M = \frac{(2\cdot\alpha)^n}{2} \tag{5.4}$$

Recall that M is the largest determinant that can be obtained from an $n \cdot n$ matrix with integer elements no larger than α in absolute value. Note that $0 \le \rho \le 1$. This definition is particularly useful in comparing determinant values between data sets of different matrix dimensions and element magnitudes.

2. Data Reduction

An experiment consists of a set of matrices with fixed values of $n$and α generated by the randomizing method described above and containing subsets of matrices with fixed determinant. Note, then, that ρ is the only variable parameter in an experiment and an experiment captures a one-dimensional view of the multi-dimensional data space.

Data Item	Description
det(A)	Value output to 16 decimal precision
α	max abs value of elements in A
$\|A\|_1$	1-norm of input matrix
a^+	max abs value of internal operands in LU algorithm
Max LU	max abs value of elements in L U factors
Min LU	min abs value of elements in L U factors
Min Diag	min abs value of diagonal elements in U
Max ErrLU	max abs value of elements in matrix $A - L \cdot U$
Max_OP Inv	max abs value of internal operand in A^{-1} comp'n
$\|A^{-1}\|_1$	1-norm of inverse
Max ErrInv	max abs value of elements in matrix $\mathbf{I} - A \cdot A^{-1}$

Table 5.1 - Reduced Data Record

For each matrix, A, in the data set we compute the LU factorization with full pivoting as an option. We compute A^{-1} for the matrix and then compute $\|A\|_1$ and $\|A^{-1}\|_1$. We capture and output the results of these computations, according to Table 5.1, as a single record. Hence the reduced data for the experiment consists of data sets corresponding to the original raw data sets. This data is used in subsequent visualizations. The tables of chapter 3 are typical, using further arithmetic processes on the reduced data of Table 5.1 to emphasize interesting or useful relations between data items. For example, the operand swell ratio displayed in Table 3.7, computed as $\frac{a^+}{\alpha}$, highlights this aspect of the LU factorization algorithm and gives a clear picture of full pivoting.

3. Certifying Determinant Computations

In this section we finally get down to work on the central concern of this thesis: Finding

the *certification predicate* $\Psi(s,t)$. We recall from way back in chapter 1 that this predicate allows us to judge whether the sign of a computed determinant is to be accepted or whether additional computation using higher precision arithmetic is required to correctly evaluate it. The previous chapters of this thesis have suggested two approaches toward making this test. one involves simply estimating the error in the computation and another involves determining whether the input matrix lies too close to a singular matrix for the sign to be resolved. In the latter approach the judgement is based on knowing the computational resolution available for safely determining 'how close'.

To be more precise, let D be the correct value of det(A) and let D' be the value of the determinant computed via the LU decomposition of A, namely $D' = \Pi_i u_{ii}$. Then $e_d = D - D'$ will be the error arising from the computation. If $e_d < D$ then the sign of D' will be correct. We may put the same remarks in relative form by defining the relative error $\hat{e}_d = \frac{D - D'}{D}$ and certification of sign follows if $\hat{e}_d < 1$, i.e. $\Psi(D', \hat{e}_d) = \hat{e}_d < 1$. In practice we do not know D. However, an estimate of $\hat{e}_d$ can be calculated from the numerical debris left over from the LU decomposition of the input matrix and computation of D'. We will examine the properties of this error estimate below. The situation can also be viewed geometrically as follows. $D = 0$ is uniquely the case for singular matrices. Now suppose D is non-zero with some sign and D' turns out to have the opposite sign. Then we can say that the non-singular input matrix, A, was too 'close' to the singular case for the computation to resolve the sign of the

determinant, the 'resolving power' of the computation being limited by floating point error. Thus we are concerned about the relationship between an estimated distance, $\hat{H}$, of the matrix from the singular case and computational error. In this view $\Psi(I, \hat{H}) = \hat{H} > \bar{\mathbf{H}}$, where $\bar{\mathbf{H}}$ is a 'resolution threshold' and is a function of the computing machinery and algorithm in use and is independent of the input A. We have seen in chapter 2 an expression for estimating the resolving power (namely D.8a) and we will also examine the properties of this means of certification below.

We turn now to an experimental study of the estimated error approach to certification. We recall (2.17)

$$\hat{e}_d = e \cdot \frac{\|A^{-1}\|_1}{n} \tag{2.17}$$

Where e is the bound on the errors accumulated in the factors L and U in the factorization process and which we have shown to be bounded by $(2n - 2) \cdot a \cdot \epsilon$ in chapter 3, section 3. Recall that in this expression a is the magnitude of the largest operand occurring in the LU algorithm and that ϵ is the machine precision. Recall also, from chapter 2, that the expression $\frac{\|A^{-1}\|_1}{n}$ represents the *estimated granularity* of the elements of the inverse matrix and thus estimates the sum of the elements of A^{-1} (see the discussion surrounding (2.17) for the details). If the LU factors are obtained using the full pivoting version of the LU algorithms, then we have shown via numerical experiments in chapter 3 that: a) $\|A^{-1}\|_1 = \frac{1}{d}$, where d is the absolute value of the diagonal element in the factor U of least magnitude, this approximation being very good,

and b) to an equally high degree of approximation $a = \alpha$, where α is the largest element, in absolute value, in the matrix A. See Table 3.7 and Table 3.9 for a substantiation of these facts. This information is readily available from the results of the LU computation and so can serve as input to the certification predicate $\Psi(U, \hat{e}_d)$.

We need experimental data that includes a view of the transition from certifiable to non-certified results. Hence we select a data set of $5 \cdot 5$ matrices with $\alpha < 1.000.000$ whose subsets consist of 33 sample matrices each where the nominal ρ values range from 0.05 to 0.60. Referring to Table 5.1, we capture the items det(A), a (from the discussion above α, or even $Max\ U$ would do as well), as well as $MinDiag$ for each matrix computation. For reduction we compute $\hat{e}_{est} = \frac{a \cdot \epsilon}{MinDiag}$, i.e. the estimated relative error, for each matrix, and since we also know the value and the sign of the determinant for each matrix, we compute $\hat{e}_{act}$, the actual relative error occurring in the computation. These results are tabulated in Figure 5.2 as follows. The column labeled ρ_0 lists the nominal value of ρ for each subset. The column labeled $\hat{e}_{act}$ displays the *maximum actual* relative error, $\hat{e}_d$, occurring in the data set corresponding to ρ_0, and $\hat{e}_{est}$ displays the *estimated* relative error which corresponds to the matrix returning this maximum actual error. The columns labeled $[\hat{e}_{est}$, $]\hat{e}_{est}[$, and $\hat{e}_{est}]$ contain, respectively, the minimum, median, and maximum values of the estimated errors in each set of data. Finally, the column labeled *C fail* counts the number of computed determinants in each data set whose signs were incorrect.

$\rho_{\cdot}$	$\hat{e}_{act}$	$\hat{e}_{est}$	$[\hat{e}_{est}$	$]\hat{e}_{est}[$	$\hat{e}_{est}]$	C_fail
0.05	8.835e-04	1.057e-05	4.751e-03	3.679e-05	2.493e-07	21
0.10	3.333e-02	6.962e-04	4.795e-02	1.790e-05	1.588e-07	11
0.15	2.760e-02	2.659e-04	6.458e-01	3.218e-04	8.538e-05	10
0.20	1.611e-00	2.957e-04	1.491e+00	1.216e-03	3.814e-04	1
0.25	1.383e-01	4.543e-03	3.808e-02	4.252e-01	4.543e-03	0
0.30	8.435e-02	1.639e-02	8.418e-04	1.202e-00	1.639e+02	0
0.35	8.284e-03	4.811e-00	2.130e-05	4.617e-02	4.811e-00	0
0.40	7.695e-05	3.508e-02	9.164e-06	4.941e-03	1.652e-01	0
0.45	7.772e-06	7.927e-03	2.669e-06	9.728e-04	7.927e-03	0
0.50	7.628e-06	6.277e-03	2.389e-06	6.390e-04	7.274e-03	0
0.55	4.485e-08	1.200e-04	1.074e-08	9.849e-06	2.374e-04	0
0.60	1.737e-09	1.139e-05	6.100e-09	7.742e-07	1.139e-05	0

Table 5.2 - Det Error. n=5. $\alpha \leq 1{,}000{,}000$

Using the *C-fail* column as a guide, we can see the onset of certification failure in the $\rho_{\cdot} = 0.20$ data. One failure occurs in this set and this can be seen clearly in the entries for $\hat{e}_{act}$ and $\hat{e}_{est}$. This $\hat{e}_{act}$ entry is certainly the one giving rise to the sole certification failure in the subset; we have, in fact, $\Psi(D^{-1}, \hat{e}_{est}) = \hat{e}_{est} > 1 = 1$ here. Note, however, that $\hat{e}_{est}$ is not the maximum *estimated* error for the subset, there is at least one matrix in the subset giving rise to an even bigger estimated determinant error. We must always keep in mind that although $\hat{e}_d > 1$ indicates *certification* failure this does not necessarily imply *sign* error: A determinant error may be large without causing a sign error. But in this case it turns out that any computations returning estimated errors larger than $\hat{e}_{est}$ do not correspond to sign faults. There is also considerable dispersion in the results for both the actual and estimated errors throughout the experimental data.

As a rule for nominal values of ρ from 0.25 through 0.6, the data sets showing no sign faults. $\hat{e}_{est}$ is the maximum for each set, just as we would expect. Determinant errors are all small and diminish for each increase in determinant magnitude. This, at least, confirms experimentally what we have already claimed, namely that large determinant errors are associated with small determinants.

$\hat{e}_{est}$, regarded as a computed estimate of $\hat{e}_{act}$, is obviously generous. This is the persistent finding of [PYS] and it has been argued, [BPY98], in reference to those findings, that the phenomenon may be a result of the failure to take into account cancellation in the evaluation of the error gain term in (2.15). In [PYS] the Hadamard estimate for determinant magnitude was used for this purpose, and so the conjecture is reasonable. However, we have shown in this thesis that this error gain term is, in fact, the sum of the elements of the inverse matrix, and we have accounted for cancellation effects in this sum by emphasizing the *granularity* of the inverse matrix. This has allowed us to use the 1-norm of the inverse as the error gain factor, a value that can be obtained with high accuracy from the LU factorization,. (Note that [BPY89] suggests other methods for obtaining data about the inverse matrix.) Nevertheless $\hat{e}_{est}$ remains persistently generous.

However, in chapter 3 we found aanother source for this persistent overestimate, namely that the *actual* errors occurring in the LU factorization process reflect the effects of the random cancellation of errors, or so we there conjecture. But this overestimate finds its

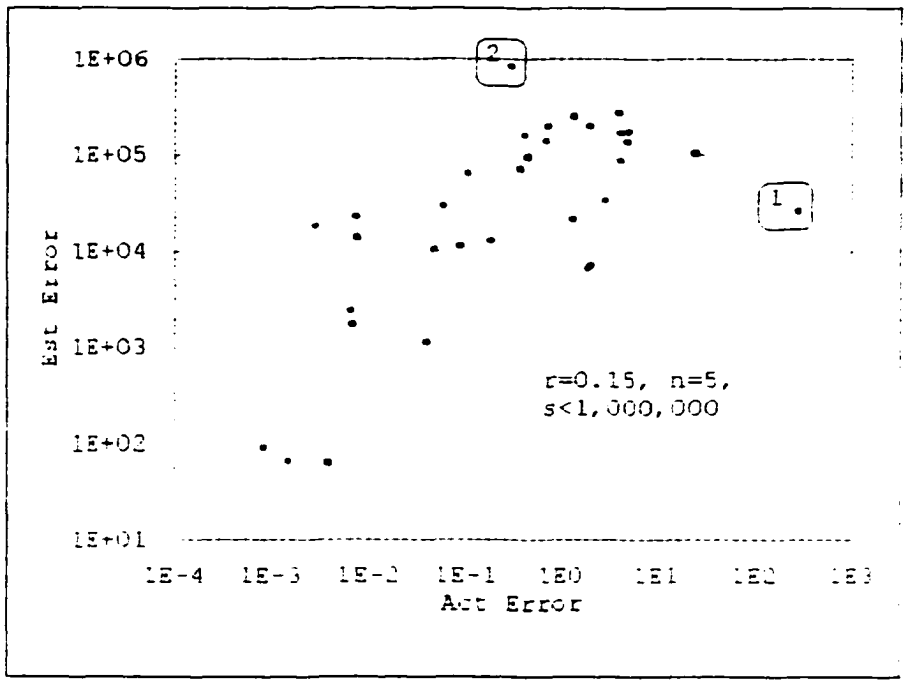


Figure 5.1 - Estimated vs Actual Error

way into the present computation of $\hat{e}_{est}$ and accounts for nearly all of the error overestimate we find here. The remaining divergence between $\hat{e}_{act}$ and its estimate $\hat{e}_{est}$ appear to be in the nature of dispersion.

We view this dispersion in Figure 5.1. This is a dot-plot of $\hat{e}_{est}$ versus $\hat{e}_{act}$ from the $\rho_{ij}=0.15$ data. The point labeled '(1)' corresponds to the $\hat{e}_{act}$ data in the $\rho_{ij}=0.15$ row in Table 5.2 and the point labeled '(2)' corresponds to the $\hat{e}_{est}$] entry, the maximum value of $\hat{e}_{est}$. Note that this latter point should be certified, that is the *actual* error in the computation of the determinant for this point is less than one. Also we find that there are 13 points in this set which are *not* certified and checking the C-fail column in Table 5.2 shows that of these 13 just 10, in fact, have the wrong sign. Moreover, 25 points in the set, including '(2)', would be caught in a certification test based on a threshold value

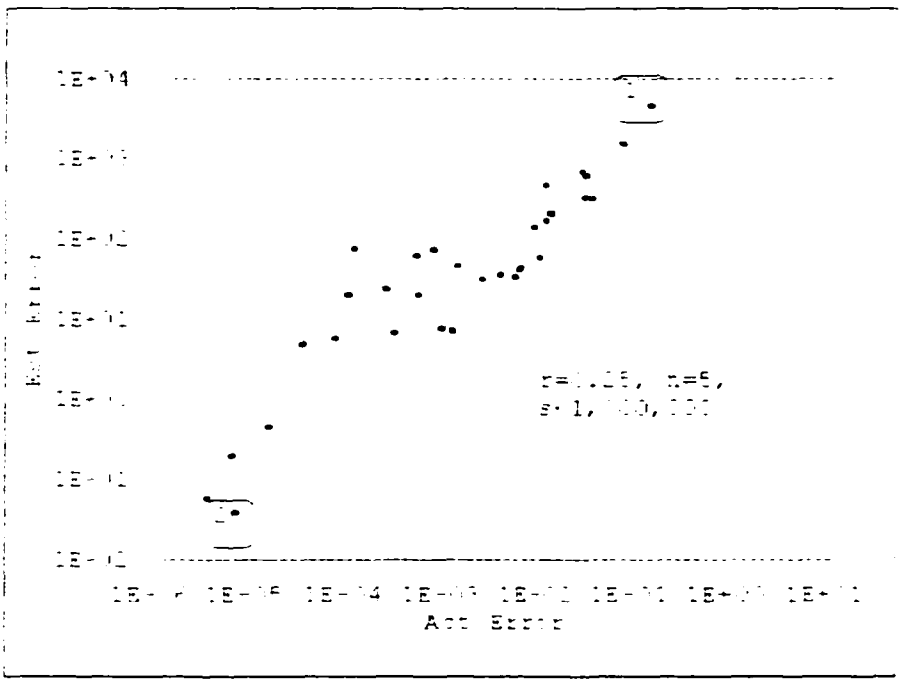


Figure 5.2 - Estimated vs Actual Error

for $\hat{e}_{est}$ set so as to catch just the non-certifieds, and that this threshold would have to be $=5\cdot 10^{3}$, a large number. Figure 5.2 is a similar plot for the data subset of $\rho_0 = 0.25$. This plot shows a more reassuring correlation between actual and estimated errors. There are no non-certified computations in this data set. The point labeled '(1)' in this plot corresponds to the $\hat{e}_{act}$ item on the $\rho_0 = 0.25$ line of Table 5.2 as well as the $\hat{e}_{est}$] item so that the maximum actual error corresponds to the maximum estimated error in this case. The point '(2)' marks the [$\hat{e}_{est}$ item in the data set and although it represents the determinant returning the smallest estimated error in the set, this determinant is *not* the one with the least actual error. The threshold value required for a certification predicate in the $\rho_0 = 0.15$ case, above, would miss all the data points save '(1)' in the $\rho_0 = 0.25$ case, which is also reassuring.

But it is not clear that a threshold value, $\bar{e}_d$, defining a certification predicate $\Psi(I, \dot{e}_{est}) = \dot{e}_{est} \leq \bar{e}_d$, which would work under all matrix input conditions. This motivates an experimental study of the error resolution method of certification.

For this experiment we need only capture the *MinDiag* data item in each capture record. For reduction we compute a number $\dot{H} = \frac{MinDiag}{\epsilon}$ which will be the distance of the input matrix to the singular matrix measured in units of the machine precision. Table 5.3 is a tabulation of these results in a format similar to Table 5.2: The column labeled $\dot{e}_{act}$ displays, as before, the largest actual error in the determinant computations and the $\dot{H}$ column displays the estimated distance to singularity for that matrix. The columns labeled $[\dot{H}$, $]\dot{H}[$, and $\dot{H}]$ display the minimum, median, and maximum values of

$\rho_{..}$	$\dot{e}_{act}$	$\dot{H}$	$[\dot{H}$	$]\dot{H}[$	$\dot{H}]$	C-fail
0.05	8.835e-04	4.001e+00	1.563e-02	1.202e+00	9.081e+01	21
0.10	3.333e-02	1.000e+01	5.079e-02	2.558e+00	4.909e-02	11
0.15	2.760e-02	1.600e+01	7.814e-01	1.289e+01	3.718e+03	10
0.20	1.611e+00	1.966e+01	1.966e+01	3.673e+02	1.630e-05	1
0.25	1.383e-01	9.602e+01	9.602e+01	8.987e+03	5.975e+06	0
0.30	8.435e-02	2.753e+03	2.753e+03	2.988e+05	2.412e+08	0
0.35	8.284e-03	1.002e+05	1.002e+05	1.010e+07	9.279e+09	0
0.40	7.695e-05	1.098e+07	2.964e+06	1.030e+08	4.602e+10	0
0.45	7.772e-06	8.228e+07	5.923e+07	7.585e+08	1.677e+11	0
0.50	7.628e-06	1.688e+08	1.215e+08	1.654e+09	3.439e+11	0
0.55	4.485e-08	1.055e+10	4.696e+09	6.574e+10	6.493e+13	0
0.60	1.737e-09	8.635e+10	8.635e+10	1.262e+12	2.101e+14	0

Table 5.3 - Distance to Singular Matrix, n=5, $\alpha \leq 1{,}000{,}000$

estimated distance in the data set.

An initial inspection of Table 5.3 reveals the not unexpected observation that the minimum, median, and maximum data play reversed roles as compared to those in Table 5.2:. it is easy to see that certification begins to fail when $\hat{H}$ is around 20 units of machine precision. The dispersion picture for this data is shown in Figure 5.3, again for the $\rho_{ij} = 0.15$ case, and which appears immediately as a flipped version of Figure 5.1. The point labeled '(1)' again picks out the $\hat{e}_{act}$ item for this data set. The points labeled '(H=72)' picks out the certification failures having the greatest distance from singularity. So there are items in this set failing certification whose distance to singularity exceeds the 20 units cited above. The most instructive aspect of Figure 5.3, however, is entirely visual: The dispersion displayed in Figure 5.3 is nearly identical to that displayed in

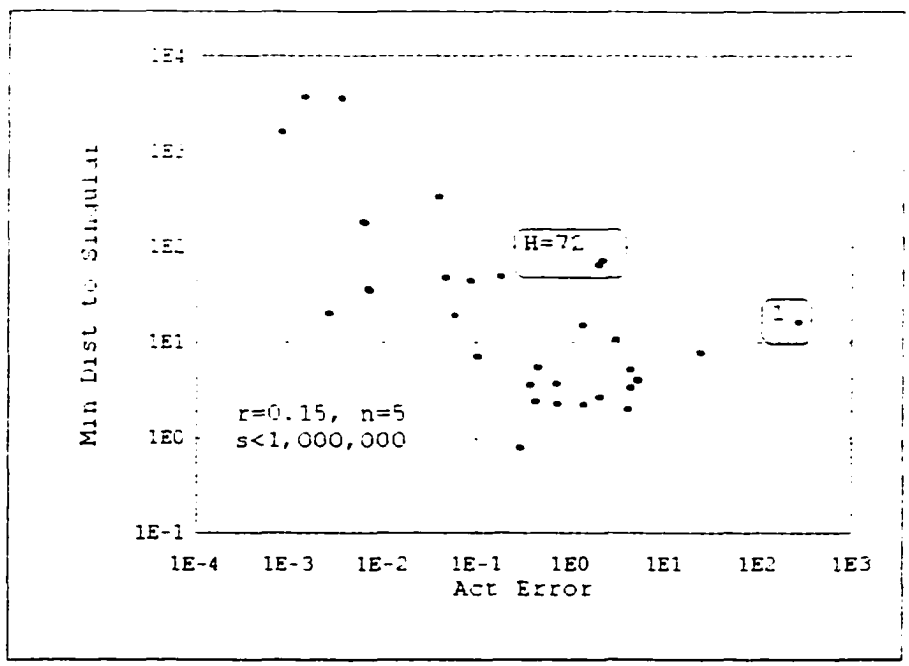


Figure 5.3 - Distance to Singular vs Actual Error

Figure 5.1. Figure 5.3 displays results computed from estimates of $\|A^{-1}\|_1$ only, all other error components of the determinant computation having been factored out of the data, yet the dispersion remains, apparently unchanged. Thus, the dispersion in the data derives from the inverse estimates, and we may conjecture from the inverse itself, since our estimates are demonstrably quite good. What does *not* remain is the persistent overestimate of LU error. This does much to validate the argument that random cancellation effects cause the LU algorithm to work much better than the classical *a priori* error estimates would suggest.

The distance resolution measure is attractive as certification predicate because there is likely to be a machine dependent threshold for $\hat{H}$ which would be a value independent of the data. But we just saw a resolution threshold of at least 72 machine precision units needed to capture all 10 certification failures in the $\rho_0 = 0.15$ data. This suggests that dispersion effects in the data may have a strong role to play in setting a safe resolution threshold value. To study this threshold question more carefully we construct another view of the experimental results in Table 5.4. This displays a 'Certification Hit Count Inventory' for the data where given a resolution threshold value, $\bar{\mathbf{H}}$, we count as a 'hit' the condition $\hat{H} \le \bar{\mathbf{H}}$, where $\hat{H}$ is the estimated distance to the singular matrix. We count as a 'certification hit' a hit which has the wrong sign. In Table 5.4 the columns labeled '#!' tally the total number of hits for the given threshold and the columns labeled '±!' count the certification hits in that number. A study of this table indicates that the dispersion problem occurs just in the $\rho_0 = 0.15$ data. Since the #! columns count the

ρ_0	C-fail	$\bar{H}$ =20		$\bar{H}$ =40		$\bar{H}$ =60		$\bar{H}$ =80		$\bar{H}$ =100	
		#!	±!	#!	±!	#!	±!	#!	±!	#!	±!
0.05	21	32	21	32	21	32	21	32	21	33	21
0.1	11	30	11	30	11	31	11	31	11	31	11
0.15	10	19	8	22	8	25	8	27	10	27	10
0.2	1	1	1	4	1	7	1	7	1	8	1
0.25	0	0	0	0	0	0	0	0	0	1	0
0.3	0	0	0	0	0	0	0	0	0	0	0

Table 5.4 - Hit inventory, n=5, $\alpha \leq 1{,}000{,}000$

number of recalculations needed to certify the signs of determinants it is clear that a higher threshold implies a more costly certification process. Moreover, too high a threshold can lead to unnecessary recalculations, as can be seen in the $\rho_0 = 0.25$ data. We can ask, is the threshold $\bar{H} = 80$, required in the present data just for the $\rho_0 = 0.15$ case, an unfortunate accident of the particular experimental data? To answer this question we present Table 5.5, a hit table from an experiment with 40 $7 \cdot 7$ matrices for which $\alpha \leq 35{,}000$. Certification failure begins in the $\rho_0 = .20$ region as in the previous

ρ_0	C-fail	$\bar{H}$ =20		$\bar{H}$ =40		$\bar{H}$ =60		$\bar{H}$ =80		$\bar{H}$ =100	
		#!	±!	#!	±!	#!	±!	#!	±!	#!	±!
0.05	18	36	18	39	18	39	18	39	18	39	18
0.1	9	28	9	30	9	31	9	31	9	31	9
0.15	5	21	5	23	5	24	5	26	5	27	5
0.2	1	5	1	9	1	12	1	12	1	13	1
0.25	0	0	0	0	0	0	0	0	0	2	0
0.3	0	0	0	0	0	0	0	0	0	0	0

Table 5.5 - Hit inventory, n=7 $\alpha \leq 35{,}000$

experiment but in this case a threshold of $\bar{\mathbf{H}} = 20$ suffices to catch all certification hits. These results show that large resolution thresholds are not always necessary. Of course, this may be a fortunate aspect of computations with matrices of larger dimension. We answer to this possibility in Table 5.6 which is a hit table obtained from another experiment using $4 \cdot 4$ matrices whose elements are such that $\alpha \leq 100{,}000{,}000$. Again the data sets in the experiment contain 40 matrices of this type. Onset of certification failure occurs in the $\rho_0 = 0.25$ data here. The results show clearly that the higher threshold values, $\bar{\mathbf{H}} = 100$, are needed in the general case. We also learn that this is a requirement just of the regions of the data space where onset of certification failure occurs, so it would be helpful to be able to distinguish matrices with merely 'small' determinants from those with truly 'tiny' determinants.

Nevertheless, we have achieved our goal. With the aid of these experiments we have established that an effective certification predicate can be stated: $\Psi(D, \hat{H}) = \hat{H} \leq 100$. Algorithm 0 is then just as displayed in Figure 5.4.

ρ_0	C-fail	$\bar{\mathbf{H}} = 20$		$\bar{\mathbf{H}} = 40$		$\bar{\mathbf{H}} = 60$		$\bar{\mathbf{H}} = 80$		$\bar{\mathbf{H}} = 100$	
		#!	±!	#!	±!	#!	±!	#!	±!	#!	±!
0.1	17	40	17	40	17	40	17	40	17	40	17
0.15	16	30	15	35	15	37	16	38	16	39	16
0.2	4	6	0	13	1	18	2	24	3	27	4
0.25	1	0	0	0	0	0	0	1	1	1	1
0.3	0	0	0	0	0	0	0	0	0	0	0
0.35	0	0	0	0	0	0	0	0	0	0	0

Table 5.6 - Hit inventory, n=4 $\alpha \leq 100{,}000{,}000$

```
Input: Matrix A, Output: sign of det(A)

     Compute the LU decomposition of A, using floating
       point arithmetic with full pivoting
     Compute det(A) = Π_i^n u_{i,i}
     Compute Ĥ = u_{n,n} / ε

     If Ĥ ≤ H̄ then
          return the sign of det(A)
     Else
          Recompute the LU decomposition using Modular
             Arithmetic with modulus > 2·det A)
          Compute the Modulus Vector det(A) = Π_i^n u_{i,i}
          Return the sign of the Chinese Remainder
             evaluation of det(A)
```

Figure 5.4 - Algorithm 0

4. A Technical Engineering Note

We introduced the 'log relative' measure, ρ, of determinant magnitude in (5.3) and used it extensively in the display of experimental data in the previous section. It has the advantage of expressing the value of a determinant in terms of its basic parameters, namely the dimension of a matrix and the largest absolute value of the elements in that matrix. Its usefulness appears manifest in the results of the last section where we found that the value $\rho < 0.20$ implied something significant about the certification properties of the determinant. Its utility can be extended in application to engineering considerations, that is in design of software tools that use the determinant computation and certification methods described in this thesis.

Max Measure was first introduced in chapter 2 and appears in this chapter as (5.4). It is the absolute value of the largest determinant that can arise from a matrix whose element of largest absolute magnitude is α. We note its logarithm

$$n\log\alpha - (n-1)\log 2 \tag{5.5}$$

where n is the dimension of the matrix.

The absolute value of the determinant of the matrix is just the product of the absolute values of the diagonal elements in the matrix U obtained from the LU factorization. (Similar remarks would hold in the case where other matrix factorizing algorithms were used, e.g. QR). We note its logarithm

$$\log\ \det(A) = \sum_i^n \log\ u_{i,i} \tag{5.6a}$$

When full pivoting is used the leading diagonals will be large, and will approximate α as we have seen in chapter 3. Thus

$$\log\ \det(A) = (n-1)\log\alpha + \log\ u_{n,n} \tag{5.6b}$$

Also under full pivoting we have found (see chapter 3) that the absolute value of the diagonal term $u_{n,n}$ approximates closely the reciprocal of the 1-norm of the inverse matrix. We have

$$\log|\det(A)| = (n-1)\log\alpha + \log\frac{1}{\|A^{-1}\|_1} \tag{5.6c}$$

But we have found in our experiments with determinant computations that sign

certification approaches failure just when the norm of the inverse matrix approaches the width of the floating point mantissa in our machine, that is ϵ [7]. We thus have

$$\log \det(A) = (n-1)\log\alpha - \log\epsilon \qquad (5.6d)$$

at the onset of certification failure. ϵ is a machine dependent quantity, for our machine it is $1.11 \cdot 10^{-16}$ (IEEE 53-bit mantissa). Hence, for our case, we have found that certification failures are to be expected when

$$\log \det(A) = (n-1)\log\alpha - 16 \qquad (5.6e)$$

As an example from our data set n=5, $\alpha \approx 1{,}000{,}000$, Tables 5.3 and 5.4, a typical matrix in this set had an actual $\alpha \approx 600{,}000$ (a consequence of the artifacts of data generation mentioned in section 1). Combining the expressions (5.5) and (5.6e) and plugging in these number we obtain a log relative measure for the estimated onset of certification failure of 0.229, which compares favorably with the experimental result.

---

[7]Buttressing this observation with a theoretical explanation is beyond the scope of our thesis but represents an interesting project. We also note that this event corresponds to a very rapid increase of the errors in the computed inverse matrix.

Chapter 6

Summary and Directions for Further Research

The results of this thesis demonstrate the potential for using run-time certification in numerical algorithms and modular arithmetic for achieving exact arithmetic in these computations. In this chapter we review these results. We examine several questions concerning extensions of our notion of certification to other high precision applications. This topic is rich in open problems, and we will highlight several of these. In section 1 we comment on the present results. Section 2 suggests extensions to other numerical computations; we examine solution of systems of equations and application to orthogonalization methods such as QR, SVD, and their friends. This discussion will prompt the contents of section 3: Open questions and research topics in applications of Modular arithmetic.

1. Discussion of our Results

We have successfully demonstrated the use of certification predicates in numerical algorithms. While the idea of conditioning the sequence of instructions in automata on the current state of the computation is hardly novel, including in this state the *quality* of intermediate results has not been commonplace. If we include in the conditioned sequence of instructions the choice of alternative arithmetics, then we are looking at something truly new. An unsympathetic response might be that fixing things up by recalculating a result using alternative arithmetic looks a lot like back-tracking, an

invitation to inefficiency, particularly for the numerical problem domain. So, although new, certification is a bad idea; perhaps it works in the sign of determinant computation application but it should not be pursued. Well, maybe so, but ultimately the question of whether it is a good idea or a bad one can only be answered by further research, trying it out in some other application domains. We suggest a few of these below. In any case if the computational requirement includes exact results the only alternative is the currently available exact arithmetic methods. Their efficiencies are well known.

We based our certification predicate on information we can obtain about the norm of the inverse matrix, information that we can get from the results of LU factorization. We chose this tool because it appeared to be the most reliable indicator of the quality of the determinant result. It works. The competition was the classical estimate of error in the LU factors. In our study of this estimate we were pleased to find that the actual errors arising from the factorization were comfortably less than the estimate would lead us to expect. The classical literature on floating point computational error, [GolVL, ORT, FM67, etc], has recently been augmented with the encyclopedic work of Higham [High], sufficient to indicate that the field still has research potential. It turns out that our experience is in line with tradition in the field, namely that research advances are prompted by the discovery that things work better in practice than they ought to according to theory. See, for instance Turing [TU48], and recall the Hotelling story in the footnote in chapter 1. So it would do well to re-examine the topic of error analysis as we undertake an extension of certification to broader numerical applications than just

the computation of the signs of determinants.

Our results certainly confirm the value of using modular arithmetic in a practical exact arithmetic application. The exact evaluation of determinants is a critical requirement in a surprising number of computational geometry problems. This we have accomplished, using modular arithmetic, so our work is an important step in this regard. We have shown, moreover, that the LU matrix decomposition algorithm is easily implemented in modular arithmetic and that modular arithmetic works well in the present application. The use of the certification procedure, particularly its precomputation in floating point, insures that the structure of the modular vector space is chosen to be optimal *prior* to the determinant computation so the iterative refinement process of [BEPP97] is unnecessary. This should be regarded as an important feature of our experience. Our success with this application of modular arithmetic prompts further research. We note, however, that the apparent computational charms of modular arithmetic have been enumerated before. [AKR], for instance, as far back as 1989, was an enthusiast. We suggest some avenues for additional research below, along with a few warning about known speed bumps.

2. Extension to other Numerical Algorithms

The obvious one is solution of a system of equations. LU (with full pivoting) is very stable and has been used for years as a workhorse solver. Indeed, Golub says that gaussian elimination is "...the algorithm of choice when [the matrix] is square, dense, and unstructured.". Hence we focus here on square, dense, and unstructured problems.

The applications are many and diverse. Our first task is to formulate the certification criterion, although for small matrices the determinant criterion we have used already would probably do. That is, the input matrix too close to singularity would be a signal that the system is poorly conditioned and that exact arithmetic is called for. To complete the Little Solver algorithm we need only add a back-substitution algorithm in modulus arithmetic to what we already have. The author has done a little experimenting with solving small systems in this way, and the idea seems effective and easily completed..

The domain of larger square dense systems, $25 \leq n \leq 500$, presents more formidable problems. The author's past experience with computing magnetic fields in non-linear magnetostatic media [Stew82] gives ample assurance (at least to the author) that effective application of exact arithmetic methods in the domain of moderately large dense systems would be a major contribution to the world of numerical computing. An outstanding feature in problems of this size is the annoying interference of ill-condition phenomena: Results of computation are curiously and intermittently wrong, inappropriate, or just "off", here and there in the solution space. The result undermines the usefulness of an application. It is well to recall at this point that system condition just *amplifies* the effect of computational error in floating point arithmetic, but exact arithmetic applied to the trouble spots would just have the effect of generating nothing to amplify.

In this Big Solver domain where problem sizes exceed 30, say, the certification algorithm, the trouble spot finder, must be considerably more involved than our present methods for certifying determinants. The task appears to be one of estimating the condition number of a system and doing this for a moderately large system is well-known to be a difficult problem. Again the majority of the work in this field has been analytical and *a priori*. Efforts to find a new condition number *computation* algorithm represents a worthwhile and challenging research topic.

There is, however, an interesting problem that arises in modular arithmetic system solving. We have seen that the finite number of elements in the Modular Ring do triple service as numbers; they 'stand' for a numerical *value* as well as the negative (additive inverse) of another value *and* the reciprocal (multiplicative inverse) of yet another (invertible) value. We have not had to concern ourselves with the latter ambiguity in the present work since determinants have always been integers and it is easy to distinguish positive and negative results. There are two ways of dealing with this problem, one is to scale the problem so that output could be expected in integer form only (not always feasible), and the other is to use *rational* modular arithmetic. The latter is the better technique: We visit this topic again in section 3.

We have not touched on orthogonalization methods such as Gram-Schmit, Housholder, Givens, etc. We have steered clear of these methods because they all require the computation of square roots, and this remains a major problem in modular arithmetic,

see further discussion below. Householder QR was tested in [PYS] and gave results comparable to full pivoting LU in respect to determinant errors so it is a good candidate for determinant computation. Floating point errors arising in QR factoring algorithms lead to the failure of the factor Q to be an exactly orthogonal matrix - hence computing with exact arithmetic would likely be very useful in applications of these algorithms. We comment that the Clarkson algorithm [CLAR] is of this class, it is a variant of the Gram-Schmit algorithm which introduces scale changes 'on the fly' in an effort to keep the computation away from the null space of the problem. The reason for trouble at higher dimension is likely that its 'orthogonal deficit' tests fail due to floating point round-off error, or otherwise put, these deficit tests could serve as a certification predicate for transition to exact arithmetic. Clarkson's orthogonality deficit tests may well be used for this purpose in other QR algorithms. Also, QR, and its iterative variant, the Conjugate Gradient method, have many applications in the Big dense square matrix Solver world we discussed above, so there are plenty of good reasons to extend the certification technique to the QR realm.

3. Research Topics in Modular Arithmetic

The preceding discussion identified a number of questions related to the practical application of modular arithmetic and these are properly classed as interesting open research topics. The success of our work, and the recently reported work of [BEPP97] should provide ample motivation for their pursuit. We outline them in this section:

Rational Arithmetic -- We mentioned a problem with using modular arithmetic in system solving. Elements of the ring do multiple duty as reciprocals. For example consider the modular vector space $\mathbf{M}^2 = \{\mathbf{F}_{17}, \mathbf{F}_{19}\}^8$. The number 23 is written {6, 4} in this space, and its reciprocal, $\frac{1}{23}$, appears as <3, 5>; that is <6, 4>·<3, 5> = 1. But <3, 5> also denotes the principal member -14. Now 23·(-14)=-322, which is also equivalent to 1 mod(323). In the present determinant computation application this representation ambiguity has not been a problem but fractional results will arise normally in the solving application. We can extend the Modulus Vector to include numerator and denominator in its components. Then the elementary operations become the fractional arithmetic ones, i.e

$$\frac{a}{b} \pm \frac{c}{d} = \frac{a \cdot d \pm c \cdot b}{b \cdot d}$$
$$\frac{a}{b} * \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$$
$$\frac{a}{b} \div \frac{c}{d} = \frac{a \cdot d}{b \cdot c}$$

These extra operations will add to the cost of the modular arithmetic, of course. But it will not be necessary to include the reduction to simpler terms that normally accompanies fractional arithmetic. Converting a rational modular number would require two Chinese Remainder computations and this step could be combined with a GCD application to effect simplification of fractions. On the other hand this latter step could convert directly to floating point, obviating an explicit simplification step.

---

[8] See chapter 4 for definitions and details about notation.

Table look-up for speed -- Without question the major stumbling block in the way of wider application of modular arithmetic to exact numerical computation is the cost of the arithmetic. In the present work, and to the author's best knowledge, in all modular arithmetic work, the elementary modular operations are implemented in computer native floating point. Thus the subroutines implementing the operations represent considerable interpretive overhead. This makes exact arithmetic expensive, and, in fact motivates our interest in certification since we wish to resort to exact arithmetic only under pressure of extreme need. The ring domain of the operations is, however, of finite cardinality; that is, the ring arithmetic can be specified completely in terms of tables. An operation is just a table look-up, binary operations such as ADD being an access to a 2-dimensional array, where the operands are the array indices and which returns the modular result of the ADD. The cost of an operation amounts to that of memory fetch. The speed up potential of implementing modular arithmetic in this way is very attractive. We remark that table look-up methods for speed up have a history of application in embedded software for navigation and missile guidance which have a severe real-time requirement.

Unfortunately the operation tables are not fixed. In fact the modular ring structure may be different for each problem, at the very least its structure is a parameter of the application. Thus, the ring operation tables must be created and recreated often. Building ring tables becomes part of the arithmetic overhead in the table look-up paradigm, and the complexity of the table building algorithm becomes a concern. Included in this concern are the attendant memory management questions. The ring

structure requires mutually prime moduli, i.e., the tables for each modulus must be present and cannot be overlapped or combined with the others. Moreover, although the arithmetic is commutative, the table for each operation must be supplied in full since otherwise the computational overhead required to insure correct addressing of the table works to defeat the initial computational speed motivation for using tables. Large moduli imply large tables. Thus the use of table look-up tends to shift computational cost concerns from process complexity to space complexity. We contend that the trade off is more than acceptable, but considerable research, especially in the areas of software design and testing, is required before our enthusiasm can be justified rigorously.

The Square Root Problem -- This problem arises as a collision between the properties of modular rings and the needs of certain numerical algorithms, notably the QR class and their use of square roots. In these algorithms the *inner product* of vectors is a significant and ubiquitous sub-computation. That is, given two vectors, $\boldsymbol{a} = \langle a_1, \cdots, a_n \rangle$ and $\boldsymbol{b} = \langle b_1, \cdots, b_n \rangle$, we often need their inner product

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1 \cdot b_1 + a_2 \cdot b_2 + \cdots + a_n \cdot b_n \qquad (6.1)$$

When (6.1) vanishes the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ are said to be *orthogonal*.

A simple application of (6.1) in a linear algebra problem would be in orthogonalizing a set of linearly independent vectors. Suppose we wish to orthogonalize the linearly independent set $\{x_1, x_2, \cdots, x_n\}$. Then we would construct a new set, $\{v_i\}$, from the

$\{x_i\}$ such that $v_i \cdot v_j = 0,\ i \neq j$. We do this construction by initializing $v_1 = x_1$ and proceeding to construct v_k recursively by subtracting an appropriate linear combination of the previously computed $\{v_i\},\ i<k$ from x_k:

$$v_k = x_k - \frac{x_k \cdot v_1}{v_1 \cdot v_1} \cdot v_1 - \frac{x_k \cdot v_2}{v_2 \cdot v_2} \cdot v_2 - \cdots - \frac{x_k \cdot v_{k-1}}{v_{k-1} \cdot v_{k-1}} \cdot v_{k-1} \qquad (6.2)$$

Since $v_i \cdot v_j = 0,\ i \neq j,\ i,j<k$ by the previous construction, it is easy to see from (6.2) that $v_k \cdot v_j = 0,\ j<k$. Such constructions are typical in QR algorithms. It is customary to *normalize* v_k by dividing its elements by $\sqrt{v_k \cdot v_k}$ after its computation, then the denominators effectively disappear in 9.2. This reduces the complexity of the algorithm at the expense of a single inner-product evaluation and extraction of square root at each step, clearly a net gain.

The aforementioned collision begins with the inner product of vectors with themselves in the denominators of (6.2) and continues with the computation of square roots. Inner products of vectors with themselves yield sums of squares, as can be seen from (6.1). In the real domain, the square of a number is non-negative. The sum of squares is therefore non-negative. In fact the expression is that of the 2-norm: Zero if and only if the vector is null. *None of these properties hold in the modular arithmetic domain.* If the modular basis is chosen from prime fields then at least we can say that the square of a number is zero only if the number is zero. But squares may be negative. Thus inner products may be zero for non-null as well as orthogonal vectors.

In the real domain only positive numbers have square roots. Similarly the square root of half of the elements in a modular domain are undefined. This is not difficult to see: Squares of the elements occupy the main diagonal in the multiplication table of the domain, which is a finite, square, array. If a number appears on this diagonal, then the corresponding row or column index (they are the same, of course) is just the square root of the number. But a number, if it appears, will appear twice on the diagonal, once for its root and once more for the modular compliment of the root. Hence only half the elements of the domain appear on the diagonal, the roots of the others are necessarily undefined.

However, this problem can be avoided. If the square root of -1 is undefined in all the moduli chosen for the modulus vector representation, then the *absolute value* of all the elements in of a modulus appear in the diagonal of its multiplication table. That is, each element appears once in the diagonal either as itself or as its complement. This implies that only one of the two solutions to $y = \sqrt{x}$ are defined in modular ring arithmetic. Thus, computing a square root may change the sign of the term in which it appears. So we are fortunate in that only minor modifications are needed in existing algorithms on this score. The moduli in which $\sqrt{-1}$ is defined are just those whose order is one greater than a perfect square. We have consistently considered only moduli of prime order, so this restriction rules out only those primes one greater than perfect squares, a rather small number of them.

Computing the square root of a number in a modular ring, however remains no small problem. Since the diagonal of the multiplication table consists of the squares of numbers, a practical algorithm for finding roots lies in inverting the diagonal list. If the number is found on the diagonal, its row index is the root. If it is not found, the root is undefined. This is a practical method, especially if the arithmetic is done by table look-up. If this is not the case then a square root table must be computed and stored for the square root function. The search for an alternative square root algorithm is an open research problem.

Some additional comments are due at this point concerning the problem of those non-null vectors whose inner product with themselves turns out to be zero. In any vector space each vector is a member of a family of scaled versions of themselves, that is, a vector $y = \alpha \cdot x$, where α is a number and x is any non-null vector, is a member of a set $\{x\}$ consisting of the scalar multiples of x. In spaces over the reals $\{x\}$ is infinite (uncountable, in fact) whereas over modular rings it is finite and of cardinality M. We distinguish a member of this set as the *unit vector*, $\bar{x}$.

There may be additional elements whose inner product vanishes; these elements will be members of vector families of cardinality M all of whose members have vanishing inner products. These elements cannot be treated as vectors. One might say they take on the role of the non-invertible numbers whose reciprocals are undefined. Just as in the LU algorithm where pivoting is used to avoid division by non-invertibles, so matrix and

vector algorithms in modular arithmetic must take precautions against their use.

The development of the matrix algorithms using orthogonalization methods and based on modular arithmetic represents an attractive topic inviting further research on the computational properties of vector spaces over modular rings.

Machine Implementation - Parallelism -- Modular arithmetic possesses the potential for a kind of parallel processing: The arithmetic operations figuring in each modulus of a ring element are independent of those figuring in the others, that is, the ring elements are arithmetically independent and their arithmetic operations can be done simultaneously. This feature was commented on in chapter 4. The potential, at least, of parallelization in Modulus arithmetic has been noted elsewhere recently [see, e.g. [BEPP97], Introduction]. We focus on this topic here and note that this fact can be exploited to speed up computations.

So long as modular arithmetic remains an interpretive software implementation, however, the benefits of its parallelization will remain a potential only. Serious consideration must be given implementing modulus arithmetic in hardware. There is very good reason to do so for we have seen that it can be applied to a wide range of 'standard' numerical computations and is an excellent platform for doing exact arithmetic. It is easy to see that when coupled with the table look-up algorithm we discussed above, a hardware implementation of modular arithmetic would easily rival the

floating point in terms of speed. Moreover, the implementation of modular arithmetic would far less costly than floating point, at least in its basic form.

The most practical implementation for hardware modular arithmetic would be at the board level. It would be installed in existing desktop machines in much the same way as video boards are now. The minimum board logic would need not do much more than the memory management task that we foresaw in the table look-up discussion above. Two megabytes of mod arithmetic accelerator memory, for example, would be sufficient to hold the tables needed for an eight component modulus vector whose moduli are less than 256. This would accommodate a maximum modulus M of around 10^{19}. The table look-up memory would be present on the board and would therefore need not consume a portion of the machine's main memory. The memory requirements increase as the square of the component moduli, so the moduli selection would involve trade-off between size of modulus vector and size of M for a given available mod arithmetic memory, which would be a fixed amount for a given installation.

The on-board logic may support the rational arithmetic capability we also discussed earlier in this section.

The Programming Application Interface (API) for the Modular Arithmetic Unit (MAU) accelerator board would follow the standard model used with other microprocessor components of this type, such as the current SVGA video interface. An API which

supports a full modulus vector structure at the hardware level could then take full advantage of the parallelism inherent in the modular arithmetic paradigm. This API would support the definition of fixed modulus vector structure by constructing the operation tables in the MAU and support hardware execution of arithmetic operations on modulus vectors. Thus once initialized for operation with modulus vector objects suitably defined in software, arithmetic operations on these objects would appear in software as the normal (overloaded) operations.

A highly desirable elaboration of the basic hardware would also permit the Chinese Remainder conversion of modulus vector operands to integer. This would considerably complicate the hardware implementation, but would greatly simplify the software application.

The design of a Modular Arithmetic Unit is clearly an applied research effort. There are risks associated with the effort as a commercial undertaking. First, modular arithmetic cannot substitute for floating point in some applications and therefore could not completely supplant the Floating Point Arithmetic hardware. Second, while the underlying logic required is certainly comparable to that found in many commercially available plug-in board products (the previously mentioned SVGA graphics accelerators, for example), its API would necessarily present a different computing environment to higher level software than the current floating point technology. The recent commentary by [Yap] on the "floating point culture" pervading most current numerical work suggests

at least an initial resistance to this paradigm shift. We have just seen that algorithms will require a lot of review in the light of the exact arithmetic model, some applications would need redesign, new applications would become feasible. We just comment that it would be a valuable contribution, and therefore a worthy research goal, to provide practical hardware support for exact computation, thus easing the transition to this new computing paradigm and probably encouraging it.

References

[ABDPY] F. Avnaim, J.-D. Boissonnat, O. Devvillers, F. Preparata, M. Yvinec, Evaluation of a New Method to Compute Signs of Determinants. *In Proc. 11th Annual ACM Symposium on Computatioanl Geometry*, pp C16-C17, 1995.

[AKR] A. Akritas, Elements of Computer Algebra, with Applications, Wiley, New York, 1989

[AF92] D. Avis, K. Fukuda, A Pivoting Algorithm for Convex Hulls and Vertex Enumeration of Arrangements and Polyhedra, *Discrete Computational Geometery*, v8, pp295-313.

[BEPP97] H. Brönnimann, I. Z. Emiris, V. Y. Pan, S. Pion, Computing Exact Geometric Predicates Using Modular Arithmetic with Single Precision, *Proc. 13th Annual Symposium on Computational Geometry*, ACM Press, New York, 1997.

[BY97] H. Brönnimann, M. Yvinec, Efficient Exact Evaluation of signs of Determinants, Research Report 3140, INRIA Sophia-Antipolis, 1997.

[BP] D. Bini, V. Y. Pan, *Polynomial and Matrix Computations, Vol 1: Funadmental Algorithms*, Birkhaeuser, Boston,1994.

[BPY98] D. Bini, V. Y. Pan, Y. Yu, *Certifying Numerical Computation of the Sign of Matrix Determinants*, (submitted), 1998

[CdB] S. Conte, C. deBoor, *Elementary Numerical Analysis: An Algorithmic Approach*, McGraw Hill, New York, 1980.

[CLAR] K. Clarkson, Safe and Effective Determinant Evaluation, *Proc 33rd Annual IEEE Symposium on the Foundations of Computing*, pp 387-395, 1992.

[DLa94] M. Deza, M. Laurent, Applications of Cut Polyhedra, *Journal of Computational and Applied Mathematics*, I, v55 pp191-216, II, v55 pp217-247, 1994.

[DST] J. H. Davenport, Y. Siret, E. Tournier, *Computer Algebra: Sysytems and Algorithms for Algebraic Computation*, Academic Press, New York, 1988.

[Fox] L. Fox, *An Introduction to Numerical Linear Algebra*, Clarendon, 1964

[FR94] K. Fukuda, V. Rosta, Combinatorial Face Enumeration in Convex Polytopes, *Computational Geometery, Theory and Applications*, v4 pp191-198, 1994.

[FM67] G. Forsythe and C. Moler, *Computer Solution of Linear Alghebraic Systems*, Prentice Hall, Englewood Cliffs, NJ, 1967

[FVnW93] S. Fortune, C. J. Van Wyk, Efficient Exact Arithmetic for Computational Geometry, *Proc. 9th Annual ACM Symposium on Computational Geometry*, pp 163-172, 1993.

[GCL] K. O. Geddes, S. R. Czapor, G. Labahn, *Algorithms for Computer Algebra*, Kluwer Academic Publishers, Boston, 1992

[Gold72] H. Goldstine, *The Computer from Pascal to von Neumann*, Princeton Univ Press, Princeton NJ, 1972.

[GoN47] H. Goldstine, J. v. Neumann, Numerical Inverting of Matrices of High Order, *Bulletin of the Americal Mathematical Society*, v53 pp1021-1099 1947.

[GolVL] G. Golub and G. Van Loan, *Matrix Computations*, Johns Hopkins Press, Baltimore, 1983.

[Knu] D. Knuth, *The Art of Computer Programming, v2 Seminumerical Algorithms*, Addison Wesley, Reading Mass, 1977 and 1997.

[High] N. Higham, *Accuracy and Stability in Numerical Algorithms*, SIAM, Philadelphia, 1996

[Hoh] F. Hohn, *Elementary Matrix Algebra*, Macmillan, New York, 1964.

[IR] K. Ireland, M. Rosen, *A Classical Introduction to Modern Number Theory, 2ed*, Springer, New York, 1990

[Ma55] R. Macmillan, A New Method for the Numerical Evaluation of Determinants, *Journal of the Royal Aeronautical Society*, v59 pp772-, 1955

[MC] R Moenck, J. Carter. Approximate Algorithms to Derive Exact Solutions of Systems of Linear Equations, Proc EUROSAM, Lecture Notes in Computer Science, v72, pp 65-73, 1979.

[Mu] T. Muir, *The Theory of Determinants in Historical Order of Development*, (1693-1900, two volumes) Dover, New York, 1960, and *Contributions to the History of Determinants*, (1900-1920), Blakie and Sons, London, 1950.

[ORT] J. Ortega, *Numerical Analysis, A Second Course*, SIAM, Philadelphia, 1990.

[P88] V. Y. Pan, Computing the Determinant and the Characteristic Polynomial of a Matrix via Solving Linear Systems of Equations, *Information Processing Letters*, v28, pp71-75, 1988.

[P97] V. Y. Pan, Algebraic Approaches to Lower Precision Computation of the Signs of Determinants, Manuscript 1997.

[PYS] V. Y. Pan, Y. Yu, C. Stewart, Algebraic and Numerical Techniques for the Computation of Matrix Determinants, *Computers and Mathematics (with Applications)* v34, no. 1, pp43-70, 1997.

[StB] J. Stoer, R. Bulirsch, *Introduction to Numerical Analysis*, Springer, New York, 1993.

[Stew82] C. Stewart, MDP: Experience with a Two-Dimensional Magnetostatic Program, *IEEE Transactions on Magnetics*, vMAG-2, no. 2, pp644-649, 1982.

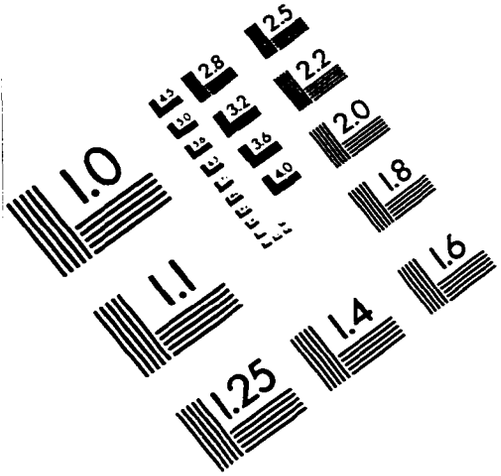
[Tu48] A. Turing, Rounding-Off Errors in Matrix Processes, *Quarterly Journal of Mechanics and Applied Mathematics*, v1 pp287-308, 1948

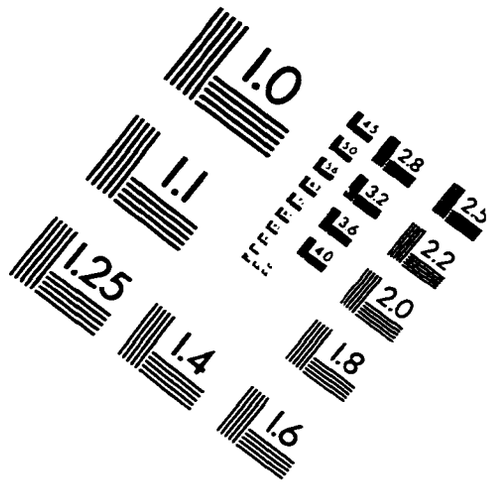
[YapDu] C. Yap, T. Dubé, The Exact Computation Pardigm, in Du and Hwang, eds *Computing in Euclidean Geometry*, World Scientific Press, 1995.

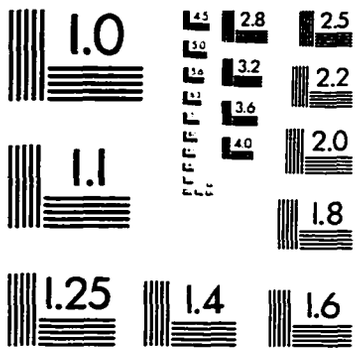
[Yap] C.Yap, Towards Exact Geometric Computation, *Computational Geometry, Theory and Applications*, v7 pp3-23, 1997

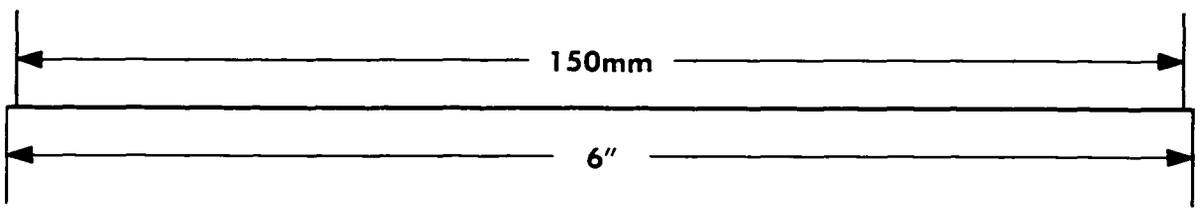
[Yun] D. Y. Yun, Algebraic Algorithms using p-adic Constructions, in *Analytic Computational Complexity*, Academic Press, New York, 1975.

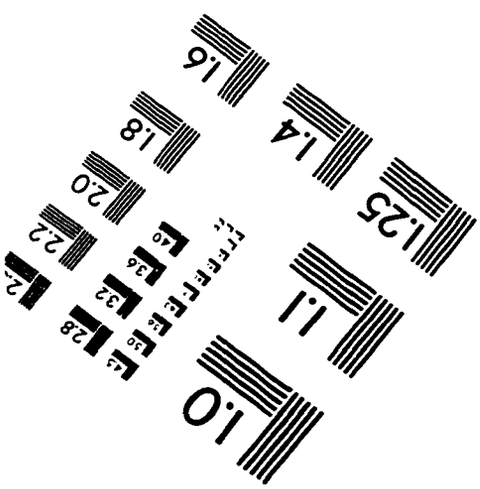
IMAGE EVALUATION TEST TARGET (QA-3)

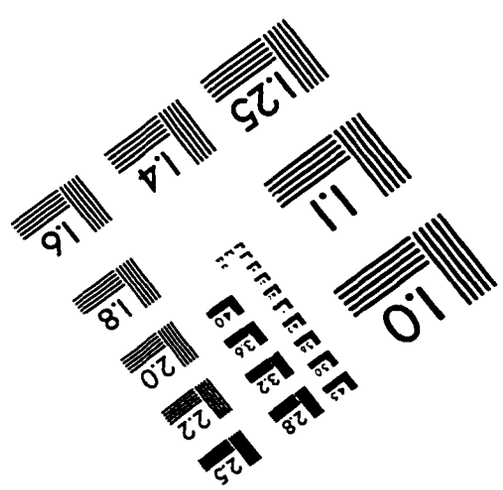












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989