

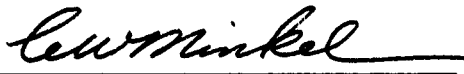
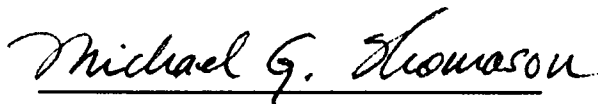
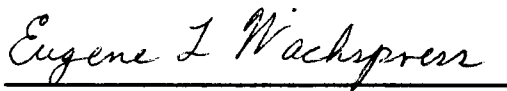
To the Graduate Council:

I am submitting herewith a thesis written by Deborah Susan Crowell entitled "Error-free Computation Using Multiple-Modulus Residue Arithmetic." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.



D.W. Straight, Major Professor

We have read this thesis
and recommend its acceptance:



Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Deborah Susan Crowell

Date April 5, 1986

ERROR-FREE COMPUTATION USING MULTIPLE-MODULUS RESIDUE ARITHMETIC

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Deborah Susan Crowell

June 1986

DEDICATION

This work is dedicated in memory of William Whitaker Smith, Jr. and Robert Todd Gregory.

ACKNOWLEDGMENTS

The author wishes to thank all those who assisted in the research and preparation of this thesis. I am especially indebted to Robert Todd Gregory for exposing me to the method of error-free computation presented. Furthermore, his work in mathematics and computer science is an invaluable source of information drawn upon in this thesis. I would like to particularly thank David Straight, Michael Thomason, Eugene Wachspress and Doyt Perry for serving on my thesis committee. I am grateful to Melissa Wilson and Donald Lampard for proof reading the thesis and Bobby Bodenheimer for his technical assistance.

I extend special thanks to my parents for their emotional and financial support throughout my educational career and especially my grandfather for making graduate school financially feasible.

ABSTRACT

Multiple-modulus residue arithmetic (MMRA) is a method of performing certain arithmetic computations ($+$, $-$, $*$, $/$) on the set of order- N Farey fractions (F_N) in an environment totally free of machine roundoff error. MMRA provides a method of representing elements in the set F_N on unique ordered n -tuples of integers, called standard residue representations with respect to the base vector. Arithmetic operations on elements in F_N are replaced by corresponding operations on the standard residue representations. These computations are performed upon the ordered n -tuples in componentwise fashion with respect to the base vector using residue arithmetic. If the resultant standard residue representation corresponds to an element in F_N , the answer is recovered totally free of machine roundoff error.

The theory and implementation of multiple-modulus residue arithmetic is discussed. Then specific aspects of multiple-modulus residue arithmetic are compared and contrasted to single-modulus residue arithmetic, concluding with recommendations for further research.

TABLE OF CONTENTS

CHAPTER	PAGE
I INTRODUCTION	1
1.1 Motivation for Error-Free Computation	1
1.2 Error-Free Computation Using Residue Arithmetic	3
II THEORETICAL BACKGROUND OF RESIDUE ARITH-	
METIC	6
2.1 Single-modulus Residue Arithmetic	6
2.2 Multiple-modulus Residue Arithmetic	13
2.3 Mapping Standard Residue Representations onto Integers . .	18
2.4 Single-modulus Residue Arithmetic with Rational Numbers .	21
2.5 The Forward Mapping and the Inverse Mapping	24
2.6 Multiple-Modulus Residue Arithmetic with Rational Numbers	29
III RESULTS AND SOFTWARE DESCRIPTIONS	40
3.1 The Hilbert Matrix	40
3.2 Software Descriptions	41
IV CONCLUSIONS AND RECOMMENDATIONS	53
4.1 Multiple-Modulus Residue Arithmetic vs Single-Modulus Residue Arithmetic	53
4.2 Conclusion	58
BIBLIOGRAPHY	59
APPENDIX	62
VITA	106

GLOSSARY OF NOTATION

$ a _m$	the least non-negative residue of a modulo m
$ \frac{a}{b} _m$	$ a * b^{-1} _m$
$ a _\beta$	$[a _{m_1}, a _{m_2}, \dots, a _{m_n}]$
$/a/_\beta$	$[/a/_m, /a/_m, \dots, /a/_m]$
$ \frac{a}{b} _\beta$	$[\frac{a}{b} _{m_1}^*, \frac{a}{b} _{m_2}^*, \dots, \frac{a}{b} _{m_n}^*]$
$a^{-1}(m)$ or a^{-1}	the multiplicative inverse of a modulo m
a_β^{-1}	$[a^{-1}(m_1), a^{-1}(m_2), \dots, a^{-1}(m_n)]$
$< a >_\beta$	$< d_0, d_1, \dots, d_{n-1} >$, mixed-radix digits
β	$[m_1, m_2, \dots, m_n]$, base vector
F_N	the set of order- N Farey fractions
$\gcd(a, b)$	the greatest common division of a and b
I	the set of integers
I_β	$\{ s _\beta : s \in I\}$
I_m	$\{0, 1, 2, \dots, m-1\}$
$\hat{I}_m$	$\{ \frac{a}{b} _m : \frac{a}{b} \in F_N\}$
$\tilde{I}_M$	$\{ \frac{a}{b} _M : \frac{a}{b} \in F_N\}$
Q	the set of rational numbers
$\hat{Q}$	$\{\frac{a}{b} : \gcd(b, m) = 1\}$
$\tilde{Q}$	$\{\frac{a}{b} : \gcd(b, m_i) = 1, \quad \text{for } i = 1, 2, 3 \dots n\}$
S_β	$\{/s/_\beta : s \in I\}$
S_m	$\{\frac{-(m-1)}{2}, \dots, -2, -1, 0, 1, 2, \dots, \frac{m-1}{2}\}$
T_β	$\{ \frac{a}{b} _\beta : \frac{a}{b} \in Q\}$
$\tilde{T}_\beta$	$\{\frac{a}{b} : \frac{a}{b} \in F_N\}$

CHAPTER I

INTRODUCTION

1.1 Motivation for Error-Free Computation

Binary computers, using a radix 2 number system, are capable of representing only a finite set of rational numbers. Thus, the continuum of rational numbers is approximated by a set of discrete computer representable numbers, called floating point numbers.

The set of floating point numbers represented in a computer are rational numbers of the form $\frac{p}{q}$ where q is a power of two; therefore, most of the familiar rational numbers such as $\frac{1}{3}$, $\frac{4}{5}$, $\frac{2}{7}$ and $\frac{1}{10}$ are excluded from this set. If a rational number is not in the set of floating point numbers, the binary representation of the rational number is adjusted to the word size of the host computer, delivering an approximation of the rational number and introducing machine roundoff error. Roundoff error also results from computations performed on the floating point numbers. For example if x and y are two adjacent floating point numbers (usually differing by the rightmost bit in the mantissa), then $z = \frac{(x+y)}{2}$ is not an element in the set of floating point numbers and z will be valued either x or y in binary computers.¹

For most applications the problem of rounding error is not considered excessively serious. However, a certain class of mathematical problems, termed ill-conditioned, can not be solved with a computer using commonly implemented methods of arithmetic. These problems are highly sensitive to perturbations in the data, and can not tolerate any rounding error without

¹Gregory and Krishnamurthy[1984], page 2

seriously distorting the answer. For example consider the problem

$$A = \begin{bmatrix} -73 & 78 & 24 \\ 92 & 66 & 25 \\ -80 & 37 & 10 \end{bmatrix} \quad (1.1)$$

$$E = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 10^{-2} \end{bmatrix} \quad (1.2)$$

$$\det(A) = 1,$$

where as

$$\det(A + E) = -118.94.^2$$

The difficulties incurred when approximating arithmetic on the rational numbers provide a strong incentive for identifying a number system which can be used for performing error-free computation. Since computers can perform some operations exactly if the operands are integers, the motivation is to find an arithmetic system which utilizes integer arithmetic as a means of eliminating machine roundoff error.

²Gregory and Karney[1978], page 50.

1.2 Error-Free Computation Using Residue Arithmetic

John Smyre implemented a form of residue arithmetic called single-modulus residue arithmetic³ (SMRA). This mathematical system allows a set of order- N Farey fractions

$$F_N = \left\{ \frac{a}{b} \in \hat{Q} : 0 \leq |a| \leq N \text{ and } 0 < b \leq N \right\}$$

to be represented on the set

$$I_m = \{0, 1, 2, \dots, m-1\},$$

where m and N satisfy the equation

$$2N^2 + 1 \leq m.$$

Computations upon the set of order- N Farey fractions are replaced by corresponding mathematical operations on the set I_m . If the answer is an element in F_N , the result is recovered exactly.

John Smyre's study demonstrates (1) that SMRA enormously improves the numerical accuracy of the answer over hardware floating point arithmetic and (2) that SMRA is much slower than using rational arithmetic for solving the same problems. Multiple-modulus residue arithmetic (MMRA) approaches several of the problems in SMRA from a new vantage point.

A main contributor to the slow speed of SMRA comes from having to perform all residue computations on extended precision integers represented in software, rather than using the integer operations available on the host computer. This results directly from the magnitude of the modulus (determining the set F_N representable). Thus, the larger the set of order- N Farey fractions representable, the slower SMRA computations are to perform.

³Smyre[1983]

Multiple-modulus residue arithmetic provides a method for representing the set of order-N Farey fractions with respect to a base vector of prime moduli on unique ordered n-tuples

$$|F_N|_\beta = [|F_N|_{m_1}, |F_N|_{m_2}, \dots, |F_N|_{m_n}]$$

where the base vector is

$$\beta = [m_1, m_2, \dots, m_n].$$

The magnitude of N is determined by

$$2N^2 + 1 \leq M$$

where

$$M = \prod_{i=1}^n m_i.$$

Hence, the order-N Farey fraction is represented as an ordered n-tuple of integers, called the standard residue digits with respect to the base vector. The base vector is composed of prime moduli represented in the integer word size of the host computer. The arithmetic operations (+, -, ·, /) are performed on each standard residue digit (s_i) reduced with respect to the corresponding modulus in the base vector (m_i). Thus, all intermediate residue arithmetic computations can be performed on the standard residue digits (with respect to the base vector) utilizing existing computer arithmetic for integers which may decrease execution time. Furthermore, the extended precision integer represented in software is only used twice, once for mapping elements in F_N onto their standard residue representation and then for recovering the order-N Farey fraction from the standard residue digits.

Hence, one can examine arithmetic computations on rational numbers in an environment totally free of machine roundoff error. Computations can be

performed upon these ordered n -tuples using residue arithmetic, and if the resultant standard residue representation corresponds to an order- N Farey fraction, the answer is recovered exactly. The task before us has three well-defined goals:

1. to develop the necessary software for implementing multiple-modulus residue arithmetic;
2. to utilize this extended precision software, solving numerical problems free of machine roundoff error;
3. to compare and contrast specific aspects of multiple-modulus residue arithmetic to single-modulus residue arithmetic, concluding with recommendations for further research.

In Chapter II, the underlying theory of multiple-modulus residue arithmetic is outlined. The numerical algorithms presented are illustrated and theoretical weaknesses demonstrated.

In Chapter III, the computational results of multiple-modulus residue arithmetic are presented and the MMRA software routines are described.

In Chapter IV, specific aspects of multiple-modulus residue arithmetic are compared and contrasted to single-modulus residue arithmetic, concluding with a summary of this research effort and recommendations for further study.

CHAPTER II

THEORETICAL BACKGROUND OF RESIDUE ARITHMETIC

The background of single-modulus residue arithmetic is presented along with the more recently developed principles of multiple-modulus residue arithmetic. The proofs of theorems are omitted here since they are readily accessible in the literature (see especially [8],[12],[14]).

It is well known that computers can perform some arithmetic operations exactly if the operands are integers. So there is motivation to perform exact arithmetic on the finite set of integers $I_m = \{0, 1, 2, \dots, m - 1\}$ as a means of avoiding machine roundoff error. Performing integer arithmetic and reducing the results modulo m , is using an arithmetic system called single-modulus residue arithmetic, where $m > 1$ is called the modulus of the system.¹

2.1 Single-modulus Residue Arithmetic

Let a, b and m be integers with $m > 1$. If m divides $b - a$ evenly, then b "is congruent to" a modulo m , written in the following manner.

$$b \equiv a(m). \tag{2.1}$$

If m does not divide $b - a$ evenly, then b "is not congruent to" a modulo m written as

$$b \not\equiv a(m). \tag{2.2}$$

¹Throughout this book I will represent the set of integers.

Theorem 2.1.1 *Let a, b and $m \in I$ with $m > 1$. The following statements are equivalent*

- (i) $a \equiv b(m)$
- (ii) $b \equiv a(m)$
- (iii) $(a - b) \equiv 0(m)$.

Theorem 2.1.2 *Let a, b and $m \in I$ with $m > 1$. If*

$$a \equiv b(m)$$

and

$$b \equiv c(m)$$

then

$$a \equiv c(m).$$

Theorem 2.1.3 *Let a, b, c, d, x, y and $m \in I$, with $m > 1$. If*

$$a \equiv b(m)$$

and

$$c \equiv d(m)$$

then

$$ax + cy \equiv bx + dy(m).$$

Theorem 2.1.4 *Let a, b, c, d and $m \in I$, with $m > 1$. If*

$$a \equiv b(m)$$

and

$$c \equiv d(m)$$

then

$$ac \equiv bd(m).$$

In single-modulus residue arithmetic $b \in I$ can be represented by an integer $r \in I_m$. The integer r is called the least non-negative residue of b modulo m , and define the mapping between b and r as

Definition 2.1.5 $|\cdot|_m = I \rightarrow I_m$ given by

$$|b|_m = r$$

iff

$$0 \leq r < m$$

and

$$b \equiv r(m).$$

With this mapping one can represent I as the union of m disjoint subsets $R_0, R_1, R_2, \dots, R_{m-1}$, called residue classes where

$$R_k = \{b \in I : |b|_m = k\}.$$

Example: Let $m = 11$ and $b = 26$, then

$$4 = |26|_{11}$$

or

$$26 \equiv 4(11)$$

and $26 \in R_4$. In this example 4 is congruent to 26 modulo 11 and 4 is the least non-negative residue of 26 modulo 11.

Theorem 2.1.6 Let a, b and $m \in I$ with $m > 1$. Then

- (i) $|a|_m$ is unique
- (ii) $|a|_m = |b|_m$ iff $a \equiv b(m)$
- (iii) $|km|_m = 0$ for all $k \in I$.

Theorem 2.1.7 Let a, b and $m \in I$, with $m > 1$. Then

$$\begin{aligned}
 |a + b|_m &= ||a|_m + |b|_m|_m \\
 &= |a + |b|_m|_m \\
 &= ||a|_m + b|_m. \\
 |ab|_m &= ||a|_m |b|_m|_m \\
 &= |a|b|_m|_m \\
 &= ||a|_m b|_m.
 \end{aligned}$$

Theorem 2.1.7 shows how addition and multiplication modulo m are performed. It may be observed that the order of reductions has no effect on the residue result.

Example: Let $x = a + b$ where $a = 15$, $b = 12$ and $m = 11$.

$$\begin{aligned}
 |x|_{11} &= |15 + 12|_{11} \\
 &= ||15|_{11} + |12|_{11}|_{11} \\
 &= |4 + 1|_{11} \\
 &= |15 + 1|_{11} \\
 &= |4 + 12|_{11} \\
 &= 5.
 \end{aligned}$$

Example: Let $x = a \cdot b$ where $a = 15$, $b = 12$ and $m = 11$.

$$\begin{aligned}
 |x|_{11} &= |15 \cdot 12|_{11} \\
 &= ||15|_{11} \cdot |12|_{11}|_{11} \\
 &= |4 \cdot 1|_{11} \\
 &= |15 \cdot 1|_{11} \\
 &= |4 \cdot 12|_{11} \\
 &= 4.
 \end{aligned}$$

Theorem 2.1.8 *The system $(I_m, +, \cdot)$ where $+$ and $\cdot$ represent addition modulo m and multiplication modulo m respectively, is a finite commutative ring with identity.*

The additive inverse of a modulo m is

$$\bar{a} = |-a|_m$$

where

$$\bar{a} = m - a.$$

Subtraction on I_m is defined as

$$|a - b|_m = |a + \bar{b}|_m \quad (2.3)$$

where

$$|b + \bar{b}|_m = 0.$$

Theorem 2.1.9 *The finite commutative ring $(I_m, +, \cdot)$ is a finite field iff m is a prime number.*

Furthermore, the finite field $(I_m, +, \cdot)$ is isomorphic to the Galois field $GF(m)$ and every non-zero element of I_m has a multiplicative inverse.

Definition 2.1.10 *If $\gcd(b, m) = 1$ and b is a non-zero element of I_m . Then there exists a unique integer $c \in I_m$ which satisfies*

$$|bc|_m = |cb|_m = 1,$$

and c is called the multiplicative inverse of b modulo m written as

$$c = b^{-1}(m) \text{ or } b^{-1}$$

when m is understood.

It is observed that if m is a composite number then a multiplicative inverse will not exist for all the elements of I_m .

Example: Let $m = 12$ then

$$I_{12} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

and multiplicative inverses exist for

$$\{1, 5, 7, 11\}.$$

However, if $m = 13$ (a prime number) then

$$I_{13} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

and multiplicative inverses exist for all elements of I_{13} .

Division modulo m can be defined using Definition 2.1.10.

Definition 2.1.11

$$\left| \frac{a}{b} \right|_m = |a \cdot b^{-1}|_m \text{ where } \gcd(b, m) = 1.$$

Example: Let $x = \frac{a}{b}$ where $a = 5$, $b = 6$ and $m = 13$.

$$\begin{aligned} \left| \frac{5}{6} \right|_{13} &= |5 \cdot 6^{-1}(13)|_{13} \\ &= |5 \cdot 11|_{13} \\ &= |55|_{13} \\ &= 3. \end{aligned}$$

So far only positive residues have been discussed. However, negative numbers are frequently encountered, so we introduce the concept of symmetric residues. The term symmetric residues is mnemonic since if m is odd, the resulting set is symmetric about the origin.

This set is given by

$$S_m = \left\{ \frac{-(m-1)}{2}, \frac{-(m-2)}{2}, \dots, -2, -1, 0, 1, 2, \dots, \frac{m-2}{2}, \frac{m-1}{2} \right\}$$

and each integer $b \in I$ can be mapped onto an element $s \in S$ by

Definition 2.1.12

$/ \cdot / : I \rightarrow S_m$ is defined by writing

$$/b/_m = s$$

iff

$$b \equiv s(m)$$

and

$$\frac{-m}{2} < s < \frac{m}{2}.$$

The integer $/b/_m$ is called the symmetric residue of b modulo m . The symmetric residue digits are found by

$$/a/_m = \begin{cases} |a|_m & \text{if } 0 \leq |a|_m < \frac{m}{2} \\ |a|_m - m & \text{otherwise.} \end{cases}$$

The standard residue digits are found by

$$|a|_m = \begin{cases} /a/_m & \text{if } 0 \leq /a/_m < \frac{m}{2} \\ /a/_m - m & \text{otherwise.} \end{cases}$$

If m is a prime number the system $(S_m, +, \cdot)$ is isomorphic to the system $(I_m, +, \cdot)$. Therefore, the data describing the problem can be mapped from $(S_m, +, \cdot)$ onto $(I_m, +, \cdot)$. Then the arithmetic computations are performed on the positive residue digits and the results are mapped back onto the set $(S_m, +, \cdot)$. Thus, freeing oneself from having to maintain signs during intermediate computations.

Example: Let $x = a - \frac{b}{c}$ where $a = -5, b = 4, c = 2$ and $m = 103$.

$$\begin{aligned}
 |x|_{103} &= ||-5|_{103} - |4|_{103}|9^{-1}(103)|_{103}|_{103} \\
 &= |98 - |4 \cdot 52|_{103}|_{103} \\
 &= |98 - 2|_{103} \\
 &= |98 + \bar{2}|_{103} \\
 &= |98 + 101|_{103} \\
 &= 96
 \end{aligned}$$

and $|96|_{103} = -7$, which is the correct result.

However, what if the correct result lies outside the set S_m ? Then the residue answer will be "congruent" to the mathematical answer. This is an instance of the condition called *pseudo-overflow*.²

2.2 Multiple-modulus Residue Arithmetic

The only known method for avoiding *pseudo-overflow* is to increase the set of integers represented in the system $(I_m, +, \cdot)$. There are two possibilities for expanding the set I_m . First, one could use a very large prime modulus extending the size of the system $(I_m, +, \cdot)$ linearly. Second, one could let m represent the product of several moduli increasing the set I_m exponentially. Since it can be shown that using several moduli is equivalent to using one modulus, the multiple-modulus residue arithmetic approach is pursued as a means of avoiding *pseudo-overflow*.

Definition 2.2.1 *The ordered n -tuple of moduli called the base vector (β) is given by*

$$\beta = [m_1, m_2, \dots, m_n]$$

²T.M. Rao first used the term *pseudo-overflow*.

whose components are the moduli $m_1, m_2, \dots, m_n$, and the $\gcd(m_i, m_j) = 1$ for all $i \neq j$.

Definition 2.2.2 Each integer s is represented by the unique ordered n -tuple of residues

$$|s|_\beta = [|s|_{m_1}, |s|_{m_2}, \dots, |s|_{m_n}]$$

called the standard residue representation of s with respect to the base vector β . The residue elements $|s|_{m_i}$ are called the standard residue digits of s with respect to β .

Example: Let $x = 85$ and $\beta = [7, 11, 13]$.

$$\begin{aligned} |x|_\beta &= |85|_\beta \\ &= [|85|_7, |85|_{11}, |85|_{13}] \\ &= [1, 8, 7]. \end{aligned}$$

Let M be the product of the moduli composing the base vector

$$M = \prod_{i=1}^n m_i. \quad (2.4)$$

In multiple-modulus residue arithmetic every $a \in I$ can be represented by an element $r \in I_M$. So if

$$a \equiv r(M)$$

and

$$0 \leq r < M$$

then r belongs to the set of least non-negative residues

$$I_M = \{0, 1, 2, \dots, M - 1\}.$$

Definition 2.2.3 *The standard residue number system for the base vector β is the M member set of standard residue representations*

$$I_\beta = \{ |s|_\beta : s \in I \}.$$

Furthermore, the elements in I_β are in one-to-one correspondence with the elements of I_M . The system $(I_M, +, \cdot)$ where $+$ denotes addition modulo m and $\cdot$ denotes multiplication modulo m is a finite commutative ring. The operations $\boxed{+}$ and $\boxed{\cdot}$ are defined on I_β such that $(I_\beta, \boxed{+}, \boxed{\cdot})$ constitutes a finite commutative ring isomorphic to the system $(I_M, +, \cdot)$.

Symmetric residues are introduced in the system $(I_\beta, \boxed{+}, \boxed{\cdot})$ exist if β only contains odd moduli. The corresponding symmetric residue number system for β is given by

$$/s/_\beta = [/s/_m1, /s/_m2, \dots, /s/_mn] \quad (2.5)$$

where the residue elements $/s/_mi$ are the symmetric residue digits for the base vector β . The symmetric residue number system is defined as the M -member set of symmetric residue representations

$$S_\beta = \{ /s/_\beta : s \in I \}. \quad (2.6)$$

Every integer s is represented by a unique ordered n -tuple $/s/_\beta$ and has a one-to-one correspondence in the set

$$S_M = \left\{ \frac{-(M-1)}{2}, \frac{-(M-2)}{2}, \dots, -2, -1, 0, 1, 2, \dots, \frac{M-2}{2}, \frac{M-1}{2} \right\}.$$

The systems $(I_\beta, \boxed{+}, \boxed{\cdot})$ and $(I_M, +, \cdot)$ correspond to the symmetric finite commutative rings $(S_\beta, \boxed{+}, \boxed{\cdot})$ and $(S_M, +, \cdot)$. Furthermore, the first two rings are isomorphic to each other and to the latter two rings respectively. Hence, multiple-modulus residue arithmetic using the base vector β

is equivalent to single-modulus residue arithmetic using M as the modulus.³

Arithmetic in the system $(I_\beta, \boxed{+}, \boxed{\cdot})$ can now be defined.

Arithmetic in I_β

Theorem 2.2.4 *Let $a, b \in I$. The standard residue representation of the integer $|a \pm b|_\beta$ with respect to the base vector is the ordered n -tuple*

$$|a \pm b|_\beta = [z_1, z_2, \dots, z_n]$$

where

$$z_i = ||a|_{m_i} \pm |b|_{m_i}|_{m_i} \quad i = 1, 2, \dots, n.$$

Theorem 2.2.5 *Let $a, b \in I$. The standard residue representation of the integer ab with respect to the base vector is the ordered n -tuple*

$$|ab|_{m_i} = [w_1, w_2, \dots, w_n]$$

where

$$|w|_{m_i} = ||a|_{m_i}|b|_{m_i}|_{m_i} \quad i = 1, 2, \dots, n.$$

Arithmetic on the standard residue representations is carried out in componentwise fashion, reducing each resultant residue digit with respect to the corresponding modulus.

Example: Let $\beta = [7, 11, 13]$ we find the standard residue representations

$$|85|_\beta = [1, 8, 7]$$

$$|19|_\beta = [5, 8, 6].$$

Thus

$$\begin{aligned} |85|_\beta + |19|_\beta &= [|1 + 5|_7, |8 + 8|_{11}, |7 + 6|_{13}] \\ &= [6, 5, 0]. \end{aligned}$$

³Gregory and Krishnamurthy[1984], page 13.

Similarly,

$$\begin{aligned} |85|_{\beta} - |19|_{\beta} &= [|1 - 5|_7, |8 - 8|_{11}, |7 - 6|_{13}] \\ &= [3, 0, 1], \end{aligned}$$

and

$$\begin{aligned} |85|_{\beta} \cdot |19|_{\beta} &= [|1 \cdot 5|_7, |8 \cdot 8|_{11}, |7 \cdot 6|_{13}] \\ &= [5, 9, 3]. \end{aligned}$$

The results are checked:

$$\begin{aligned} |104|_{\beta} &= [6, 5, 0], \checkmark \\ |66|_{\beta} &= [3, 0, 1], \checkmark \\ |1615|_{\beta} &= [5, 9, 3], \checkmark \end{aligned}$$

Notice that 1615 is not in I_{1001} ; this is an example of *pseudo-overflow* occurring in an intermediate result and

$$614 = |1615|_{1001}.$$

The form of *pseudo-overflow* that results when the mathematical is "congruent" to the residue answer is undetectable, unless the correct answer is already known in advance.

The multiplicative inverse on the system $(I_{\beta}, \boxed{+}, \boxed{\cdot})$ requires that $a^{-1}(M)$ exist, where M is the product of the moduli; so for $a^{-1}(\beta)$ to exist, $a^{-1}(m_i)$ must exist for $i = 1, 2, \dots, n$.

Definition 2.2.6 Let a be an integer and let $a^{-1}(m_1), a^{-1}(m_2), \dots, a^{-1}(m_n)$ exist. Then the ordered n -tuple

$$a^{-1}(\beta) = [a^{-1}(m_1), a^{-1}(m_2), \dots, a^{-1}(m_n)]$$

is called the standard residue representation of the multiplicative inverse of a with respect to the base vector β .

Theorem 2.2.7 *Let a and b be integers and let $b^{-1}(\beta)$ exist. Then*

$$\left| \frac{a}{b} \right|_{\beta} = [c_1, c_2, \dots, c_n],$$

where

$$c_i = ||a|_{m_i} b^{-1}(m_i)|_{m_i} \quad i = 1, 2, \dots, n.$$

Example: Let $\beta = [7, 11, 13]$. If

$$|19|_{\beta} = [5, 8, 6]$$

and

$$12_{\beta}^{-1} = [3, 1, 12]$$

then

$$\begin{aligned} \left| \frac{19}{12} \right|_{\beta} &= [|5 \cdot 3|_7, |8 \cdot 1|_{11}, |6 \cdot 12|_{13}] \\ &= [1, 8, 7]. \end{aligned}$$

Note that M is by definition a composite number and the system $(I_M, +, \cdot)$ is a finite commutative ring. Consequently, one should expect some difficulty because $a^{-1}(\beta)$ will not exist for every $a \in I$. This problem is examined in section 2.6.

2.3 Mapping Standard Residue Representations onto Integers

Identifying the standard residue digits has been demonstrated. However, as yet no method of recovering the integer value from the standard residue representation has been shown. The method presented uses the *mixed-radix* number representation for each integer. The ordered n -tuple of moduli where

$$m_1 < m_2 < \dots < m_n \tag{2.7}$$

and call $m_1, m_2, \dots, m_n$ the radices, and M is the product of the radices

$$M = \prod_{i=1}^n m_i.$$

It is well known that every integer s in the range

$$0 \leq s < M$$

can be uniquely expressed in the form

$$s = d_0 + d_1(m_1) + d_2(m_1 \cdot m_2) + \dots + d_{n-1}(m_1 \cdot m_2 \dots \cdot m_{n-1}). \quad (2.8)$$

The integers $d_0, d_1, d_2, \dots, d_{n-1}$ are called the *standard mixed-radix digits* satisfying

$$0 \leq d_i < m_{i+1} \quad i = 0, 1, 2, \dots, n-1.$$

The mixed-radix digit sequence is denoted as

$$\langle s \rangle_\beta = \langle d_0, d_1, d_2, \dots, d_{n-1} \rangle$$

Hence, given a set of *mixed-radix digits* and the corresponding radices, the integer value represented can be determined.

Example: Given $\langle s \rangle_\beta = \langle 6, 6, 0 \rangle$, find s .

$$\begin{aligned} s &= 6 + 6 \cdot 7 + 0 \cdot 7 \cdot 11 \\ &= 48. \end{aligned}$$

Definition 2.3.1 The *standard mixed-radix system* for β is the set of digit sequences $\langle s \rangle_\beta$ for integers in the range

$$0 \leq s < M.$$

Now one can ask, given a set of standard residue digits how are the mixed-radix digits computed? The method presented for identifying the mixed-radix digits was developed by R.T. Gregory. This method can be solved only by utilizing residue arithmetic. The algorithm for locating the mixed-radix digits $\langle s \rangle$ given the standard residue representation $|s|_\beta$ is shown. To isolate the first mixed-radix digit (d_0) let $s = t_1$ (from 2.8).

$$\begin{aligned} t_1 &= s \\ &= d_0 + m_1[d_1 + d_2(m_2) + \dots + d_{n-1}(m_2 \cdot m_3 \dots \cdot m_{n-1})] \\ &= d_0 + m_1 t_2. \end{aligned} \quad (2.9)$$

The result is reduced (see Theorem 2.1.6)

$$\begin{aligned} |t_1|_{m_1} &= |d_0 + m_1 t_2|_{m_1} \\ &= d_0. \end{aligned} \quad (2.10)$$

In essence the first digit is found with no computation. The value t_2 is used for isolating the second mixed-radix digit: $t_2 = \frac{s-d_0}{m_1}$.

$$\begin{aligned} t_2 &= d_1 + d_2(m_2) + d_3(m_2 \cdot m_3) + \dots + d_{n-1}(m_2 \cdot m_3 \dots \cdot m_{n-1}) \\ &= d_1 + m_2[d_2 + d_3(m_3) + \dots + d_{n-1}(m_3 \cdot m_4 \dots \cdot m_{n-1})] \end{aligned}$$

The result is reduced (see Theorem 2.1.6)

$$\begin{aligned} |t_2|_{m_2} &= |d_1 + m_2 t_3|_{m_2} \\ &= d_1. \end{aligned}$$

This formula may be expressed recursively as follows

$$\begin{cases} t_{i+1} = \frac{t_i - d_{i-1}}{m_i} \\ d_i = |t_{i+1}|_{m_{i+1}} \end{cases} \quad i = 1, 2, \dots, n-1. \quad (2.11)$$

Given $|s|_\beta$ one can successively compute the mixed-radix digits using residue arithmetic.

Example: Let $M = 1001$, $\beta = [7, 11, 13]$ and $s = 675$.

$$|675|_\beta = [3, 4, 12]$$

β	<u>7 11 13</u>
$ s _\beta = t_1 _\beta$	3 4 12
$ d_0 _\beta$	<u>— 3 3 3</u>
$ t_1 - d_0 _\beta$	0 1 9
$m_1^{-1}(\beta_1)$	<u>× 8 2</u>
$ t_2 _{\beta_1} = \frac{(t_1 - d_0)}{m_1} _{\beta_1}$	8 5
$ d_1 _{\beta_1}$	<u>— 8 8</u>
$ t_2 - d_1 _{\beta_1}$	0 10
$m_2^{-1}(\beta_2)$	<u>× 6</u>
$ t_3 _{\beta_2} = (t_2 - d_1)_{m_2} _{\beta_2}$	8

Example: Solving the mixed-radix digits given $\langle s \rangle = \langle 3, 8, 8 \rangle$

$$\begin{aligned} s &= 3 + 8(7) + 8(7 \cdot 11) \\ &= 675. \checkmark \end{aligned}$$

2.4 Single-modulus Residue Arithmetic with Rational Numbers

If the greatest common divisor of m and b is 1, then $b^{-1}(m)$ exists (see Definition 2.1.10). Hence, one is capable of mapping a fraction⁴ onto an element in the set I_m . The set of rational numbers which can be mapped

⁴Since $\frac{ka}{kb} = \frac{a}{b}$ all rational numbers $\frac{a}{b}$ are assumed to be in their reduced form and the $\gcd(a, b) = 1$.

onto images in I_m is given by

$$\hat{Q} = \left\{ \frac{a}{b} : \gcd(b, m) = 1 \right\}.$$

Definition 2.4.1 If $x = \frac{a}{b}$, and if $\gcd(b, m) = 1$ so $b^{-1}(m)$ exists

$$\begin{aligned} |x|_m &= \left| \frac{a}{b} \right|_m \\ &= |a \cdot b^{-1}|_m \end{aligned}$$

and describe the mapping as $|\cdot|_m : \hat{Q} \rightarrow I_m$.

It is observed that the mapping $(\hat{Q} \rightarrow I_m)$ is defined for a subset of the rational numbers. Hence, every $k \in I_m$ is the image of an infinite number of elements in $\hat{Q}$. This infinite set of rational elements which map onto the image Q_k is described as

$$Q_k = \left\{ \frac{a}{b} \in \hat{Q} : \left| \frac{a}{b} \right|_m = k \right\}$$

where

$$\hat{Q} = \bigcup_{k=0}^{m-1} Q_k.$$

The disjoint subsets $Q_0, Q_1, \dots, Q_n$ are called the generalized residue classes modulo m .

The following theorem provides the basis for computations in the set I_m .

Theorem 2.4.2 The mapping $|\cdot|_m : \hat{Q} \rightarrow I_m$ is a homomorphism with respect to addition and multiplication.

Arithmetic operations performed in the ring $(\hat{Q}, +, \cdot)$ correspond to those same arithmetic operations performed in the ring $(I_m, +, \cdot)$. However, an isomorphic relationship between $(\hat{Q}, +, \cdot)$ and $(I_m, +, \cdot)$ does not exist because there is not a one-to-one mapping between $\hat{Q}$ and I_m . A subset of $\hat{Q}$ is identified which has an isomorphic relationship to certain elements I_m .

The mapping $|\cdot| : \hat{Q} \rightarrow I_m$ is one-to-one but not onto. However, a unique element in some of the generalized residue classes $(\hat{Q}_k)$ for which a one-to-one and onto relationship exists can be identified.

Theorem 2.4.3 *Let N be the largest integer satisfying the inequality*

$$2N^2 + 1 \leq m,$$

and let the generalized residue class Q_k contain the order- N Farey fraction $x = \frac{a}{b}$; then x is the only order- N Farey fraction in Q_k .

Definition 2.4.4 *The finite subset of $\hat{Q}$*

$$F_N = \left\{ \frac{a}{b} \in \hat{Q} : \gcd(a, b) = 1 \text{ and } 0 \leq |a| \leq N, 0 < b \leq N \right\},$$

where $N > 0$ is an integer, and F_N is called the set of order- N Farey fractions.

A one-to-one and onto relationship between certain elements of $\hat{Q}$ and certain images in I_m is established by letting

$$\hat{I}_m = \left\{ \left| \frac{a}{b} \right|_m : \frac{a}{b} \in F_N \right\} \quad (2.12)$$

denote the set of images of the order- N Farey fractions.

Theorem 2.4.5 *The mapping $|\cdot|_m : F_n \rightarrow \hat{I}_m$ is one-to-one and onto and thus, has an inverse.*

Example: Let $x = \frac{1}{4} - \frac{3}{4}$ and $m = 37$.

$$\begin{aligned}
 |x|_{37} &= \left| \frac{1}{4} - \frac{3}{4} \right|_{37} \\
 &= |1 \cdot 28 - |3 \cdot 28|_{37}|_{37} \\
 &= |28 + \overline{10}|_{37} \\
 &= |28 + 27|_{37} \\
 &= |55|_{37} \\
 &= 18.
 \end{aligned}$$

The mapping is $|18|_{37} \rightarrow \frac{-1}{2}$ is shown later.

One notices that the intermediate residue answers may not represent elements in F_N ; however, this has no fatal effect on the answer, as long as the solution is an order- N Farey fraction.

2.5 The Forward Mapping and the Inverse Mapping

Gregory and Kornerup[1983] developed a method for performing the mapping $F_N \rightarrow \hat{I}_m$ based upon the Extended Euclidian Algorithm.

An Extended Euclidian Algorithm

Let (A, B) and (C, D) be integer pairs where $A > C > 0$ and the matrix

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

is called the "seed matrix" where the original integer pairs $((A, B)$ and $(C, D))$ generate a finite sequence of additional integer pairs. At each point in the computation the previous two integer pairs $((a, b)$ and $(c, d))$ respectively are used for computing the next integer pairs (x, y) .

Step 1. Initialize the seed matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

Step 2. Compute the integer quotient

$$q = \frac{a}{c}.$$

Step 3. Compute the next integer pair in the sequence by

$$x = a - qc,$$

$$y = b - qd.$$

Step 4. Update the seed matrix

$$(a, b) \leftarrow (c, d)$$

$$(c, d) \leftarrow (x, y).$$

Step 5. If $x \neq 0$ go to Step 2.

Step 6. The $\gcd(A, C) = a$.

First this algorithm is used for computing the greatest common divisor and then other uses of the Extended Euclidian Algorithm in residue arithmetic are demonstrated.

Example: Compute the $\gcd(129, 30)$.

	129	0
	30	1
4	9	-4
3	3	13
3	0	-43

and the $\gcd(129, 30) = 3$.

The Forward Mapping

It is observed that the Extended Euclidian Algorithm can be used for directly computing $b^{-1}(m)$. To find $\left|\frac{a}{b}\right|_m$ a two step process is utilized. First, find $b^{-1}(m)$ beginning with the seed matrix

$$\begin{bmatrix} m & 0 \\ b & 1 \end{bmatrix}.$$

Then multiply $|a \cdot b^{-1}(m)|_m$. However, these steps can be combined into one step using the seed matrix

$$\begin{bmatrix} m & 0 \\ b & a \end{bmatrix}.$$

Example: Map $\left|\frac{3}{14}\right| \rightarrow 1001$ using the two-step method.

	1001	0	
	14	1	
71	7	-71	←
2	0	143	

$$|-71|_{1001} = 930$$

$$|930 \cdot 3|_{1001} = 788.$$

Example: Map $\left|\frac{3}{14}\right| \rightarrow 1001$ using the one-step method.

	1001	0	
	14	3	
71	7	-213	←
2	0	429	

$$|-213|_{1001} = 788.\sqrt{}$$

Hence, the Extended Euclidian Algorithm is useful for mapping $F_N \rightarrow I_m$.

The Inverse Mapping

The algorithm presented was developed independently by Kornerup and Krishnamurthy. The inverse mapping $|\cdot| : I_m \rightarrow F_N$ also utilizes the Extended Euclidian Algorithm. From Theorem 2.4.3 one knows a generalized residue class (Q_k) contains at most one order- N Farey fraction. The following theorem guarantees that if Q_k contains an element in F_N , the fraction will be generated using the Extended Euclidian Algorithm.

Theorem 2.5.1 (*Kornerup*) *If $\frac{r}{s}$ is an order- N Farey fraction satisfying*

$$\left| \frac{r}{s} \right|_m = |k|_m = k$$

where $k \in \hat{I}_m$ and

$$0 < r \leq N$$

$$0 < |s| \leq N,$$

then there exists an integer i such that

$$(r, s) = (a_i, b_i)$$

where $\{(a_j, b_j)\}$ for $j = 1, 2, \dots, n+1$, is the sequence of pairs generated by the Extended Euclidian Algorithm (from pages 25-26) beginning with the seed matrix

$$\begin{bmatrix} m & 0 \\ k & 1 \end{bmatrix}.$$

Example: Let $m = 19, k = 7$ and $N = 3$.

	19	0	
	7	1	
2	5	-2	
1	2	3	←
2	1	-8	
2		0	19

The set of "congruent" fractions is

$$\left\{ 7, \frac{-5}{2}, \frac{2}{3}, \frac{-1}{8} \right\}.$$

Observe that $\frac{2}{3}$ is the only order-3 Farey fraction generated.

Recall (from page 25) if $x = \frac{1}{4} - \frac{3}{4}$ and $m = 37$ then $|x|_{37} = 18$. Now the mapping $|18|_{37} \rightarrow \frac{-1}{2}$ is demonstrated.

	37	0	
	18	1	
2	1	-2	←
18		0	37

What if the resultant residue does not represent an order- N Farey fraction? This is another form of *pseudo-overflow* and is always fatal at this point causing the inverse mapping to fail. This particular form of *pseudo-overflow* is easily detected; in comparison to the form of *pseudo-overflow*

which occurs when the mathematical answer is “congruent” to an order- N Farey fraction causing undetectably fatal effects.

Example: Let $m = 1001$, $b = 43$ and $N = 22$.

	1001	0
	43	1
23	12	-23
3	7	70
1	5	-93
1	2	163
2	1	-419
2	0	1001

Notice that there is no order-22 Farey fraction generated and the detectable form of *pseudo-overflow* has occurred. Hence, one is reminded of the importance in selecting the modulus (or moduli) of the system carefully; so, the data and results are order- N Farey fractions.

2.6 Multiple-Modulus Residue Arithmetic with Rational Numbers

Recall that MMRA utilizes a base vector of moduli

$$\beta = [m_1, m_2, \dots, m_n]$$

where

$$M = \prod_{i=1}^n m_i.$$

In this section it is assumed that β is composed of prime moduli and that N is the largest integer satisfying

$$2N^2 + 1 \leq M.$$

Recall single-modulus residue arithmetic is equivalent to multiple-modulus residue arithmetic and Theorem 2.4.2 establishes the homomorphism between the systems $(\hat{Q}, +, \cdot)$ and $(I_m, +, \cdot)$. An analogous homomorphism is established between a similar commutative ring with identity $(\tilde{Q}, +, \cdot)$ and the commutative ring $(I_M, +, \cdot)$.

To map $x = \frac{a}{b}$ onto an integer in I_M one must compute $b^{-1}(M)$

$$\left| \frac{a}{b} \right|_M = |ab^{-1}|_M.$$

Now $b^{-1}(M)$ exists only if $\gcd(b, m_i) = 1$, for $i = 1, 2, \dots, n$. This shows that all rational numbers can be mapped onto an element in I_M except for those rational numbers whose denominator is divisible by at least one modulus.

The set of rational numbers with images in I_M is

$$\tilde{Q} = \left\{ \frac{a}{b} : \gcd(b, m_i) = 1, \quad \text{for } i = 1, 2, \dots, n \right\}, \quad (2.13)$$

and describe the mapping as $|\cdot|_M : \tilde{Q} \rightarrow I_M$. Every $j \in I_M$ represents an infinite set of rational numbers in $\tilde{Q}$ called a generalized residue class

$$Q_j = \left\{ \frac{a}{b} \in \tilde{Q} : \left| \frac{a}{b} \right|_M = j \right\}$$

where

$$\tilde{Q} = \bigcup_{j=0}^{M-1} Q_j.$$

The following theorem provides the basis for computations in I_M .

Theorem 2.6.1 *The mapping $|\cdot|_M : \tilde{Q} \rightarrow I_M$ is a homomorphism with respect to addition and multiplication.*

Because M is the least common multiple of the moduli composing the base vector, if the generalized residue class Q_j contains the order- N Farey fraction $x = \frac{a}{b}$, then x is the only order- N Farey fraction in that generalized residue class (see Theorem 2.4.3). There are (by definition) fewer order- N Farey fractions than M so one can observe that every generalized residue class will not contain an order- N Farey fraction. The set of images of order- N Farey fractions are denoted by

$$\tilde{I}_M = \left\{ \left| \frac{a}{b} \right|_M : \frac{a}{b} \in F_N \right\},$$

and the mapping $|\cdot|_M : F_N \rightarrow \tilde{I}_M$ is one-to-one and onto, thus has an inverse.

Example: Let $\beta = [5, 7]$ so that $M = 35$ and $N = 4$ the mapping is shown in the following table.

$$|\cdot|_{35} : F_4 \rightarrow \hat{I}_{35}$$

0	0		
1	1	-1	34
2	2	-2	33
3	3	-3	32
4	4	-4	31
.	.	.	.
$\frac{-3}{4}$	8	$\frac{3}{4}$	27
$\frac{1}{4}$	9	$\frac{-1}{4}$	26
.	.	.	.
$\frac{-2}{3}$	11	$\frac{2}{3}$	24
$\frac{1}{3}$	12	$\frac{-1}{3}$	23
$\frac{4}{3}$	13	$\frac{-4}{3}$	22
.	.	.	.
$\frac{-3}{2}$	16	$\frac{3}{2}$	19
$\frac{-1}{2}$	17	$\frac{1}{2}$	18

Example: Solve $x = \frac{2}{3} - \frac{1}{3}$ on the two modulus system with $\beta = [7, 11]$. In the preceeding table $\left|\frac{2}{3}\right|_{35} \rightarrow 24$ and $\left|\frac{-1}{3}\right|_{35} \rightarrow 23$.

$$\begin{aligned} x &= |24 + 23|_{35} \\ &= 12 \end{aligned}$$

and $12 \rightarrow \frac{1}{3}$, which is the correct mathematical answer.

A Two Modulus System⁵

The system $(\tilde{Q}, +, \cdot)$ is homomorphic to the system $(I_M, +, \cdot)$ and the condition met for mapping $\tilde{Q} \rightarrow I_M$ is the $\gcd(b, M) = 1$ where M is the product of the moduli. One may ask, can $m_1, m_2, \dots, m_n$ be selected such that $m_i > N$ for $i = 1, 2, \dots, n$? For a two modulus system we can always select two prime numbers of greater magnitude than N . However, for a system with more than two moduli this is not possible.

Theorem 2.6.2 *Let $M = m_1 m_2$, where m_1 and m_2 are successive primes and let $N > 0$ be the largest integer satisfying the inequality*

$$2N^2 + 1 \leq M.$$

Then m_1 and m_2 are both greater than N .

Theorem 2.6.3 *Let $m_1 < m_2 < \dots < m_n$ be primes and let M be their product. Choose $N > 0$ to be the largest integer satisfying the inequality*

$$2N^2 + 1 \leq M.$$

If $N > 2$, then $m_1 \leq N$.

⁵Hardy and Wright[1960]

Multiple-modulus Residue Arithmetic with More Than Two Moduli

Because using more than two moduli results in some $m_i < N$ for $i = 1, 2, \dots, n$, difficulty arises both in determining the standard residue digits for $\frac{a}{b} \in Q$ when $\gcd(b, m_i) \neq 1$ for $i = 1, 2, \dots, n$, and also in performing residue arithmetic on these standard residue digits. David Matula and R.T. Gregory posit the method presented below for performing residue arithmetic and recovering the rational answer.

Consider the set of images of rational numbers

$$T_\beta = \left\{ \left| \frac{a}{b} \right|_\beta : \frac{a}{b} \in Q \right\},$$

and the subsets T_β consisting of the set of images of order- N Farey fractions

$$\tilde{T}_\beta = \left\{ \left| \frac{a}{b} \right|_\beta : \frac{a}{b} \in F_N \right\},$$

with N as the largest integer satisfying $2N^2 + 1 \leq M$. The mapping

$|\cdot|_\beta : F_N \rightarrow \tilde{T}_\beta$ is onto and will show this mapping is one-to-one by exhibiting the inverse mapping $\tilde{T}_\beta \rightarrow F_N$.

Definition 2.6.4 For any rational number $\frac{a}{b}$

$$\left| \frac{a}{b} \right|_\beta = [c_1, c_2, \dots, c_n]$$

where for $i = 1, 2, \dots, n$,

$$c_i = \left| \frac{a}{b} \right|_{m_i}^*$$

with

$$\left| \frac{a}{b} \right|_{m_i}^* = \begin{cases} \left| \frac{a}{b} \right|_{m_i} & \text{if } b^{-1}(m_i) \text{ exists} \\ \infty & \text{otherwise.} \end{cases}$$

The standard residue representation of a rational number in multiple-modulus residue arithmetic is

$$\left| \frac{a}{b} \right|_{\beta} = \left[\left| \frac{a}{b} \right|_{m_1}^*, \left| \frac{a}{b} \right|_{m_2}^*, \dots, \left| \frac{a}{b} \right|_{m_n}^* \right].$$

Example: Let $\beta = [2, 3, 7]$. So that $M = 42$ and $N = 4$. The mapping $|\cdot|_{\beta} = F_N \rightarrow \tilde{T}_{\beta}$ is exhibited in the following table⁶.

$\frac{a}{b}$	$\left \frac{a}{b} \right _2^*$	$\left \frac{a}{b} \right _3^*$	$\left \frac{a}{b} \right _7^*$	$\frac{a}{b}$	$\left \frac{a}{b} \right _2^*$	$\left \frac{a}{b} \right _3^*$	$\left \frac{a}{b} \right _7^*$
0	0	0	0				
1	1	1	1	-1	1	2	6
2	0	2	2	-2	0	1	5
$\frac{1}{2}$	∞	2	4	$\frac{-1}{2}$	∞	1	3
3	1	0	3	-3	1	0	4
$\frac{1}{3}$	1	∞	5	$\frac{-1}{3}$	1	∞	2
4	0	1	4	-4	0	2	3
$\frac{1}{4}$	∞	1	2	$\frac{-1}{4}$	∞	2	5
$\frac{2}{3}$	0	∞	3	$\frac{-2}{3}$	0	∞	4
$\frac{3}{2}$	∞	0	5	$\frac{-3}{2}$	∞	0	2
$\frac{3}{4}$	∞	0	6	$\frac{-3}{4}$	∞	0	1
$\frac{4}{3}$	0	∞	6	$\frac{-4}{3}$	0	∞	1

The following definitions are needed to show this mapping $\tilde{T}_{\beta} \rightarrow F_N$ is one-to-one.

Definition 2.6.5

$$M^* = \prod_{i=1}^n m_i,$$

⁶[5]Class Notes

If no such element exists then $M^* = 1$.

Definition 2.6.6

$$M_0 = \prod_{j=1}^n m_j,$$

where m_j is a modulus for which

$$\left| \frac{a}{b} \right|_{m_j}^* = 0.$$

If no such element exists then $M_0 = 1$.

Definition 2.6.7

$$M_1 = \prod_{k=1}^n m_k,$$

where m_k is a modulus for which

$$0 \leq \left| \frac{a}{b} \right|_{m_k}^* \leq m_k - 1.$$

If no such element exists then $M_1 = 1$.

Definition 2.6.8 If $I_{M1} = \{0, 1, \dots, M_1 - 1\}$ is defined $q \in I_{M1}$ to be the unique integer which has the property

$$|q|_{M1} = c_l,$$

for each c_l different from 0 or ∞ .

The n-tuple $|q|_{M1}$ is the standard residue representation of q with respect to $M1$. The integer value represented by q is found by using the recovery method which first isolates the mixed-radix digits $\langle q \rangle$ and then solves the mixed-radix digits (see section 2.3).

The n-tuple $|q|_{M_1}$ is the standard residue representation of q with respect to M_1 . The integer value represented by q is found by using the recovery method which first isolates the mixed-radix digits $\langle q \rangle$ and then solves the mixed-radix digits (see section 2.3).

Definition 2.6.9

$$q' = |qM^*M_0^{-1}(M_1)|_{M_1}$$

The following theorem is fundamental and gives Matula's algorithm for the mapping $|\cdot| : \tilde{T} \rightarrow F_N$.

Theorem 2.6.10 *Given $|\frac{a}{b}|_\beta \in \tilde{T}_\beta$, the unique element $\frac{a}{b} \in F_N$ is found by using the inverse mapping (from section 2.5) with the seed matrix*

$$\begin{bmatrix} M_0M_1 & 0 \\ M_0q' & M^* \end{bmatrix}.$$

Example: Let $\beta = [11, 13, 17]$, $M = 2431$, $N = 34$, and $|\frac{a}{b}|_\beta = [6, 5, 11]$. What is $\frac{a}{b}$ in F_N ?

$$\begin{aligned} (i) \quad M^* &= 1 \\ (ii) \quad M_0 &= 1 \\ (iii) \quad M_1 &= 2431 \\ (iv) \quad q' &= |qM^*M_0^{-1}(M_1)|_{M_1} \\ &= |2085 \cdot 1|_{2431} \\ &= 9 \end{aligned}$$

The element in F_N is recovered beginning with the seed matrix presented in

Theorem 2.9.8 where $M_0M_1 = 2431$, $M_0q' = 2085$ and $M^* = 1$.

	2431	0	
	2085	1	
1	346	-1	
6	9	7	←
38	4	-267	
2	1	541	
4	0	2431	

It is observed that $\frac{9}{7}$ is the only order-34 Farey fraction recovered.

Arithmetic in I_β

The arithmetic operations are performed in componentwise fashion on the standard residue digits with respect to the base vector. However, a way of determining the resultant standard residue digit if either of the components is ∞ must be established.

Case I

Let $x = \frac{a}{b} + \frac{c}{d}$ in $\tilde{T}_\beta$ where the modulus m_i divides d , but does not divide b or c and have

$$\left| \frac{a}{b} \right|_{m_i}^* \neq \infty \text{ and } \left| \frac{c}{d} \right|_{m_i}^* = \infty.$$

Applying the common denominator method of arithmetic yields

$$x = \frac{ad + bc}{db}.$$

It is observed that m_i divides the denominator but not the numerator. Hence, an addition rule is established if $\frac{a}{b} \neq \infty$ then

$$\left| \frac{a}{b} \right|_{m_i}^* + \infty = \infty.$$

Case II

Let $x = \frac{a}{b} \times \frac{c}{d}$ in $\tilde{T}$ where the modulus m_i divides d but does not divide a or c . Again the common denominator form of arithmetic is applied yielding

$$x = \frac{ac}{bd}.$$

It is observed that m_i divides the denominator but does not divide the numerator. Hence, the following rule for multiplication is established if $\left|\frac{a}{b}\right|_{m_i}^* \neq \infty$ then

$$\left|\frac{a}{b}\right|_{m_i}^* \times \infty = \infty.$$

Case III

Suppose we want to find the sum $x = \frac{a}{b} + \frac{c}{d}$ where m_i divides b and d but not a and c then

$$\left|\frac{a}{b}\right|_{m_i}^* = \infty = \left|\frac{c}{d}\right|_{m_i}^*.$$

In this case we must factor out m_i from the addends

$$\left|\frac{a}{b'm_i}\right|_{m_i}^* + \left|\frac{c}{d'm_i}\right|_{m_i}^*.$$

Now we can perform addition on the values

$$x = \left|\frac{a}{b'}\right|_{m_i}^* + \left|\frac{c}{d'}\right|_{m_i}^*,$$

then if $|x|_{m_i} \neq 0$, the sum should be ∞ , otherwise $|x|_{m_i}$ should be a residue digit in the set $I_{m_i} = \{1, 2, \dots, m_i - 1\}$.

Suppose we want to find the product $x = \frac{a}{b} \times \frac{c}{d}$ where m_i divides b and c but not a and d then

$$\left|\frac{a}{b}\right|_{m_i}^* = \infty \text{ and } \left|\frac{c}{d}\right|_{m_i}^* = 0.$$

In this case we must factor out m_i from the multiplicands

$$\left|\frac{a}{b'm_i}\right|_{m_i}^* \times \left|\frac{c'm_i}{d}\right|_{m_i}^*.$$

Now we can perform multiplication on the values

$$x = \left| \frac{a}{b'} \right|_{m_i}^* \times \left| \frac{c'}{d} \right|_{m_i}^*,$$

$|x|_{m_i}$ should be a residue digit in the set $I_{m_i} = \{1, 2, \dots, m_i - 1\}$.

Because of the difficulty encountered determining the resultant residue digit in Case III, David Matula suggests utilizing a value denoted ω . The ω symbol represents a standard residue digit which is omitted from the recovery process (along with the corresponding modulus). The following arithmetic tables are utilized in MMRA.

Multiple-modulus Residue Arithmetic Tables

$+(-)$	0	x	∞	ω
0	0	x	∞	ω
y	y	$ y + x _m$	∞	ω
∞	∞	∞	ω	ω
ω	ω	ω	ω	ω

$*(/)$	0	x	∞	ω
0	0	0	ω	ω
y	0	$ y * x _m$	∞	ω
∞	ω	∞	∞	ω
ω	ω	ω	ω	ω

Hence, when an ω value occurs that standard residue digit and the corresponding modulus are eliminated from further processing.

CHAPTER III

RESULTS AND SOFTWARE DESCRIPTIONS

The multiple-modulus residue arithmetic algorithms described in the previous chapter were implemented in C on a Vax 11/780 computer. The main program and subroutines are listed in Appendix A. These algorithms are applied for calculating the inverse elements of the Hilbert Matrix.

3.1 The Hilbert Matrix

Computing the inverse elements of a Hilbert Matrix is interesting for two main reasons. First, this problem is widely known as an inherently ill-conditioned problem, highly sensitive to perturbations in the data. Second, the elements comprising the Hilbert Matrix are simple fractions. The implemented MMRA computer routines are tested by computing the inverse elements of an $N \times N$ Hilbert matrix where $N = 3$, $N = 4$ and $N = 5$. For example, each element a_{ij} of the Hilbert Matrix is given by

$$a_{ij} = \frac{1}{(i+j-1)}$$

and the 3×3 Hilbert Matrix is

$$H_3 = \begin{bmatrix} \frac{1}{1} & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}. \quad (3.1)$$

The inverse elements of the Hilbert Matrix are given by

$$a_{ij}^{-1} = \frac{(-1)^{(I+J)}(N+I-1)!(N+J-1)!}{(I+J-1)[(I-1)!(J-1)!]^2((N-I)!(N-J)!)}.$$

The computed inverse values of the 3×3 Hilbert Matrix are

$$H_3^{-1} = \begin{bmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{bmatrix}. \quad (3.2)$$

The computed inverse values of the 4×4 Hilbert Matrix are

$$H_4^{-1} = \begin{bmatrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{bmatrix}. \quad (3.3)$$

The computed inverse values of the 5×5 Hilbert Matrix are

$$H_5^{-1} = \begin{bmatrix} 25 & -300 & 1050 & -1400 & 630 \\ -300 & 4800 & -18900 & 26880 & -12600 \\ 1050 & -18900 & 79380 & -117600 & 56700 \\ -1400 & 26880 & -117600 & 179200 & -88200 \\ 630 & -12600 & 56700 & -88200 & 44100 \end{bmatrix}. \quad (3.4)$$

3.2 Software Descriptions

An overview of how the computer programs interact is presented and then the routines are described. First, the main program initializes the base vector of moduli. Since all the fractions are generated in the program, the numerators and denominators are moved into an extended precision fraction. The forward mapping routine calculates the standard residue representation of $\frac{a}{b}|_{\beta}$. Then the integer arithmetic operations $(+, *)$ are performed on the standard residue representation in componentwise fashion, reducing each residue digit with respect to the corresponding modulus in the base vector.

Then the order- N Farey fraction is recovered from the resultant standard residue representation. There are two main programs which compute the inverse elements of the Hilbert Matrix. One program solves the analytical formula presented. The second program performs Gaussian elimination with backward substitution solving for the identity matrix. A description of all the data structures and subroutines follow in alphabetical order.

Data Structures

ARRAY_	This particular data structure serves four main purposes: (1) Representation of an extended precision number, (2) Representation of standard residue digits, (3) Representation of the product of the moduli, (4) Representation of the N value in the order- N Farey fractions.
BASE_VECTOR	This data structure contains the base vector.
FRACTION	This is used for representing an extended precision fraction.
NUMBER_LENGTH	This is the magnitude of the allowed extended precision.

Multiple-modulus Residue Arithmetic Routine Descriptions

<u>a_b_beta</u>	This routine maps the reduced Order- n Farey fraction onto the set of standard-residue digits
-----------------	---

$$\left| \frac{a}{b} \right|_{\beta} = [c_1, c_2, \dots, c_n], \text{ for } i = 1, 2, \dots, n,$$

where

$$c_i = \begin{cases} \left| \frac{a}{b} \right|_{m_i} & \text{if } b^{-1}m_i \text{ exists} \\ \infty & \text{otherwise.} \end{cases}$$

ARGUMENTS: **rational:** An extended precision Order-N Farey fraction.

RESULTS: **base:** The base vector of prime moduli.
answer: The standard residue representation of the order-N Farey fraction.

NOTE: Negative one denotes ∞ for $\left| \frac{a}{b} \right|_{\beta}$ when the $\gcd(b, m_i) \neq 1$ for $i = 1, 2, \dots, n$.

add_array This routine adds two extended precision integers.

ARGUMENTS: **a.1:** An extended precision integer.

a.2: An extended precision integer.

RESULTS: **answer:** The extended precision result.

add_standard_residues This routine adds two standard residue representations in componentwise fashion with respect to the base vector.

ARGUMENTS: **a.1:** The standard residue representation of the first addend.

a.2: The standard residue representation of the second addend.

base: The base vector of moduli.

RESULTS:	answer: The standard residue representation of the result.								
<u>big_m</u>	This routine computes the product of the moduli comprising the base vector.								
ARGUMENTS:	base: The base vector of moduli.								
RESULTS:	The product of the moduli.								
FATAL ERRORS:	If the product of the moduli exceeds the storage allowed for the extended precision representation of M , a message is printed indicating the source of the problem and program execution is terminated.								
<u>compare</u>	This routine compares the magnitude of two extended precision decimal numbers, the returned value reflects the outcome of the comparison.								
ARGUMENTS:	a1: An extended precision integer. a2: An extended precision integer.								
RETURNED VALUE:	comparison returned								
	<table border="0"> <tr> <td>decision</td> <td>values</td> </tr> <tr> <td>a1 = a2</td> <td>0</td> </tr> <tr> <td>a1 < a2</td> <td>-1</td> </tr> <tr> <td>a1 > a2</td> <td>1</td> </tr> </table>	decision	values	a1 = a2	0	a1 < a2	-1	a1 > a2	1
decision	values								
a1 = a2	0								
a1 < a2	-1								
a1 > a2	1								
<u>convert</u>	This routine recovers the mixed radix digits from the standard residue digits with respect to the base vector.								
ARGUMENTS:	t_minus.d: The array containing the standard residue digits.								
	base: The base vector of moduli.								
RESULTS:	s_digit_set: The recovered mixed radix digit set.								

<u>decimal_value</u>	This function computes the decimal value represented by an extended precision number.
ARGUMENTS:	value: An extended precision number.
RETURNED VALUE:	The decimal value represented in the extended precision array.
FATAL ERRORS:	If the decimal number is too large (ten digits are used) an error message indicating the source of the problem is printed and program execution terminates.
<u>digit_array</u>	This routine copies a decimal number (from a particular position in the array passed) into an extended precision array for further processing.
ARGUMENTS	digit_set: This array contains integers. posn: This indicates the location in the "digit_set" of the integer one wishes to copy.
RESULT:	answer: This is the extended precision copy of the selected integer.
<u>divide_array:</u>	This routine performs the operation $ \frac{a}{b} _p$. If $b^{-1}(m_i)$ for $i = 1, 2, \dots, n$, does not exist the fraction is denoted as negative one in the standard residue vector.
ARGUMENTS:	numerator_: The numerator of standard residue digits. denominator_: The denominator of standard residue digits. base: The base vector of moduli. posn: The starting position in the standard residue digits.
RESULTS:	quotient: The standard residue quotient.

farey_fraction_array This routine utilizes the Extended Euclidian Algorithm, solving for the order-N Farey fraction. The seed matrix is

$$\begin{bmatrix} modulus_ & 0 \\ denominator_ & numerator_ \end{bmatrix}.$$

ARGUMENTS: denominator_: An extended precision number used in the seed matrix.
 numerator_: An extended precision number used in the seed matrix.
 modulus_: An extended precision number used in the seed matrix.

RESULTS: answer: The extended precision fraction returned.

NOTE: If no Order-N farey fraction is recovered a message indicating *pseudo - overflow* has occurred is printed.

forward_map_array This routine is used for computing the inverse of a fraction with respect to the modulus utilizing the seed matrix

$$\begin{bmatrix} modulus_ & 0 \\ denominator_ & numerator_ \end{bmatrix}.$$

ARGUMENTS: denominator_: An extended precision number used in the seed matrix.
 numerator_: An extended precision number used in the seed matrix.

modulus.: An extended precision number used in the seed matrix.

RESULTS:

answer: This routine computes

$denominator^{-1}(modulus) \times numerator.$

forward_map_integer

This routine utilizes the Extended Euclidian algorithm to map a reduced integer onto the set

$$I_m = 0, 1, 2, \dots, m - 1,$$

where m is the modulus of the system using the seed matrix

$$\begin{bmatrix} modulus & 0 \\ value & 1 \end{bmatrix}.$$

ARGUMENTS:

value: An integer in the seed matrix.

RESULTS

modulus: The modulus in the seed matrix.

answer: The inverse of value with respect to the modulus.

gcd_array

This routine finds the greatest common divisor between two extended precision integers.

ARGUMENTS:

value.: An extended precision integer.

modulus.: An extended precision integer.

RESULTS:

answer: The extended precision greatest common divisor.

get_moduli

This routine reads the prime moduli into the base vector.

ARGUMENTS:

base: The base vector initialize to prime moduli.

RESULTS:

The initialized base vector.

M_0 This routine scans the standard residue digits. If a digit is zero then the corresponding modulus becomes a component in the product vector returned. If no digit is zero the product returned is one.

ARGUMENTS: base: The base vector of moduli.

residue: The standard residue digit array.

RESULT: m_0_vector: The product of the moduli whose corresponding standard residue digit is zero. If no digit is zero the product returned is one.

M_1 This routine scans the standard residue digits. If a digit is greater than zero then the corresponding modulus becomes a component in the product vector returned. If no digit is greater than zero the product returned is one.

ARGUMENTS: base: The base vector of moduli.

residue: The standard residue digit array.

RESULT: m_1_vector: The product of the moduli whose corresponding standard residue digit is greater than zero. If no digit is greater than zero, the product returned is one.

m_star This routine scans the standard residue digits. If a digit is negative one (symbolically ' ∞ ') the corresponding modulus becomes a component in the product vector returned. If no digit is negative one then the product returned is one.

ARGUMENTS: base: The base vector of moduli.

residue: The standard residue digit array.

RESULTS:

m_star_vector: The product of the moduli whose corresponding standard residue digit is negative one. If no digit is negative one, one is the value of the returned array.

mod_array

This routine reduces the value with respect to the modulus the number of reductions and returns an extended precision array with residue given by

$$answer \equiv value(modulus).$$

ARGUMENTS:

*value_*: The extended precision number which is reduced.

*modulus_*: The extended precision modulus of the system.

RESULTS:

*answer_*: The residue value passed to the calling routine.

ret_val: The number of reductions required on the extended precision array.

multiply_array

This routine multiplies two extended precision integers.

ARGUMENTS:

a1: An extended precision multiplicand.

a2: An extended precision multiplicand.

RESULTS

answer: This array contains the extended precision product.

multiply_standard_residues

This routine multiplies two standard residue representations in componentwise fashion with respect to the base vector.

ARGUMENTS: a.1: The standard residue representation of the first multiplicand.

a.2: The standard residue representation of the second multiplicand.

base: The base vector of moduli.

RESULTS: answer: This array contains the standard residue representation of the product.

print_ This routine prints an extended precision integer.

ARGUMENTS: vector: This is an extended precision array.

print_base_vector This routine prints the BASE_VECTOR data structure.

ARGUMENTS: base: This the base vector data structure.

print_fraction This routine prints an extended precision fraction.

ARGUMENTS: rational: An extended precision fraction.

q_value This routine establishes a subset of the standard residue digits, $|s|m_j = c_k$, for each $c_k > 0$ where $j = 1, 2, \dots, n$. The new set of digits and their corresponding moduli are solved for their mixed-radix digits. The mixed-radix digits are solved, resulting in the q value.

ARGUMENTS: sum.: The array containing the standard residue digits.

base: The base vector of moduli.

RESULT: q_value: the value of the solved mixed-radix digits.

<u>recover</u>	This routine recovers the Order-N Farey fraction from the standard residue representation of the result.
ARGUMENTS:	base: The base vector of moduli.
	sum: The standard residue digit array.
RESULT:	rational: The order-N Farey fraction recovered.
<u>reduce_integer</u>	This routine reduces a value with respect to a modulus.
ARGUMENTS:	value: The integer which is reduced.
	modulus: The modulus of the reduction.
RETURNED VALUE	The residue of the reduction.
<u>set_up_digit_array</u>	This routine duplicates an integer value in an extended precision array.
ARGUMENTS:	value: The integer which is copied into an extended precision array.
RESULTS:	answer: The extended precision integer represented in an array.
<u>sub_array</u>	This routine performs the extended precision operation ($a_1 - a_2$).
ARGUMENTS:	a_1: An extended precision subtrahend.
	a_2: An extended precision subtrahend.
RESULT:	answer: The extended precision result of ($a_1 - a_2$).
<u>sum_array</u>	This routine computes the value represented in the mixed-radix digit set with respect to the base vector

$$\langle s \rangle_\beta = \langle d_0, d_1, d_2, \dots, d_{n-1} \rangle$$

and

$$s = d_0 + d_1(m_1) + d_2(m_1m_2) + \dots + d_{n-1}(m_1m_2\dots m_{n-1}).$$

ARGUMENTS: d_set: The mixed-radix digits.
base: The base vector of moduli.
RESULT: answer: The extended precision integer represented in
the mixed-radix digit set.

CHAPTER IV

CONCLUSIONS AND RECOMMENDATIONS

4.1 Multiple-Modulus Residue Arithmetic vs Single-Modulus Residue Arithmetic

There are four specific aspects of MMRA which are compared and contrasted to SMRA.

1. If the data describing the problem and the solution to the problem are order- N Farey fractions one can use SMRA or MMRA for computing the results totally free of machine roundoff error. However, if the result is not an order- N Farey fraction *pseudo-overflow* occurs. Unfortunately, *pseudo-overflow* when recovering the answer is always fatal, and undetectable if the mathematical answer is "congruent" to an order- N Farey fraction. Therefore MMRA has no advantage over SMRA in this particular aspect.

2. The use of prime numbers is no longer restricted to Mersenne Prime numbers which allow easy computation of the order- N Farey fraction. The study conducted¹ on single-modulus residue arithmetic uses very large Mersenne prime numbers as a method for avoiding *pseudo-overflow*. Mersenne prime numbers allow relatively easy computation of the integer N satisfying

$$2N^2 + 1 \leq M$$

where M is the modulus of the system. Mersenne prime numbers are of the form

$$p = 2^a - 1$$

¹Smyre[1983]

where p and a are primes. Substituting this value into the equation $2N^2 + 1 \leq M$ yields

$$2N^2 + 1 \leq 2^a - 1$$

$$2N^2 \leq 2^a - 2$$

$$N^2 \leq 2^{a-1} - 1$$

$$N^2 < 2^{a-1}$$

$$N < 2^{\frac{a-1}{2}}.$$

Hence, one can see Mersenne primes allow easy initialization of the value N .

However, in MMRA we compute N as an extended precision integer. The formula is

$$k_i = \sqrt{(m_i - 1)/2} \quad \text{for } i = 1, 2, \dots, n$$

where k_i is an integer in the product

$$N = \prod_{i=1}^n k_i.$$

Thus, the value used as N in this implementation is somewhat less than the square root of M where

$$M = \prod_{i=1}^n m_n \quad \text{for } i = 1, 2, \dots, n.$$

Yet, there is extra flexibility in selecting prime numbers.

3. One notices that the elements in the standard residue representation are independent. This results from the integer arithmetic operations being performed in componentwise fashion with respect to the base vector. Suppose there were K parallel processors available. Then ideally one could

utilize a base vector with K moduli and perform the residue arithmetic computations on all K members of the standard residue representation simultaneously. Thus, a parallel processing implementation may speed up MMRA significantly over SMRA utilizing extended precision integers.

4. Single-modulus residue arithmetic will always be preferable to MMRA if the host computer is capable of representing the values in the system I_m and the modulus of the system in ordinary word lengths. Consequently, the speed of SMRA would be more expedient than MMRA for performing arithmetic computations in an environment free of machine roundoff error.

5. Single-modulus residue arithmetic needs only one operand for representing an element in the system I_m . However, multiple-modulus residue arithmetic uses n operands (where n is the number of moduli in the base vector) for representing the same element in the system I_β .

Disadvantages of Multiple-modulus Residue Arithmetic

1. In MMRA (as in SMRA) the recovered order- N Farey fraction is not guaranteed the correct mathematical solution. This results directly from *pseudo-overflow* which occurs when the order- N Farey fraction is "congruent" to the mathematical solution or the residue result does not correspond to an order- N Farey fraction.

2. The implemented multiple-modulus residue routines require enormous amounts of CPU time. Although this is not a time study, the execution time requirements are clearly higher than similar floating point computations.

3. One can not estimate the magnitude or sign of a rational number represented in a standard residue representation. Therefore, the only option is mapping the intermediate results onto their corresponding fractions in F_N and performing the comparison.

However, *pseudo-overflow* may occur during the inverse mapping of the intermediate result, causing the entire computation to fail.

4. When encountering an undefined value in the standard residue digits, we discard the digit and the corresponding modulus from the base vector. Consequently, the set F_N is reduced. Thus, increasing the likelihood of *pseudo-overflow*.

Research Suggestions

1. Clearly the most disturbing problem in MMRA is *pseudo-overflow*. This system of arithmetic will have a much higher acceptance if this particular problem is resolved (at least to the point where it is always detectable).

2. Currently, one is unable to determine the sign and magnitude of the rational equivalent of standard residue representation without performing the inverse mapping. This detracts from the MMRA and SMRA systems, since performing the inverse mapping on intermediate results may cause the entire computation to fail. Hence, some form of magnitude estimation should be examined.

3. The problem of ω values in the standard residue representations should be investigated. Robert Gregory and David Matula developed the method for performing the arithmetic operations $(+, -, \cdot, /)$ utilizing ω . Carl Gregory (Robert Gregory's son) and David Matula also developed a method for performing the arithmetic operations $(+, -, \cdot, /)$ which eliminates the ω value from the standard residue digits. However, this method detracts from the parallel processing aspects of MMRA. (Furthermore, as of this writing a homomorphism had not been established between Gregory and Matula's system and $\tilde{T}_\beta$.)

4. Experimental research defining a "function of usefulness" for determining when MMRA or SMRA is the more expedient form of residue arithmetic will be useful. It is expected that when executing a computationally intensive program the arithmetic computations on the standard residue digits will be much quicker than the same arithmetic computations on the extended precision integer in SMRA. With (extensive) research an exact function correlating execution time and the number of arithmetic computations could be demonstrated.

5. This is a new method for stabilizing numerical algorithms, thus numerical analysis will have a new approach for studying algorithms. Furthermore, existing numerical analysis techniques can be applied to the data of MMRA problems which may minimize the possibility of undefined values occurring.

Software Enhancements

The implemented routines are extremely slow. For improved execution time the routines which perform the mappings $|\cdot| : F_N \rightarrow |\frac{a}{b}|_\beta$ and $|\cdot| : |\frac{a}{b}| \rightarrow F_N$ should be (1) implemented in hardware or (2) implemented in microcode or (3) implemented in assembly language. (Of the three, the hardware implementation gives the best performance improvement and assembly language the least performance improvement.

2. If the data are rational numbers with the numerator and denominator representable in the integer word size of the host computer, then a specific routine can be dedicated for computing the $|\frac{a}{b}|_\beta$ which utilizes only integer arithmetic. The currently implemented routine serving this purpose is designed for mapping extended precision fractions into the system.

3. The main program performing residue arithmetic is considerably longer than a similar program using floating point arithmetic. For example to execute the string of commands

$$\left| \frac{a}{b} \right|_{\beta} \times \left| \frac{c}{d} \right|_{\beta} + \left| \frac{e}{f} \right|_{\beta}^2$$

requires

```
multiply_standard_residues(|a/b|_beta, |c/d|_beta, temp_1)
multiply_standard_residues(|e/f|_beta, |e/f|_beta, temp_2)
add_standard_residues(temp_1, temp_2, result).
```

Clearly a complex assignment could be handled more conveniently if the implementation called for statements similar to ordinary arithmetic symbols.

4.2 Conclusion

A method of residue arithmetic is presented which performs computations exactly if the data describing the problem and the solutions to the problem are order-N Farey fractions. Multiple-modulus residue arithmetic is less restrictive in the choice of prime moduli and has the potential to out-perform (in terms of execution time) single-modulus residue arithmetic because of the ability to utilize existing integer arithmetic on the host computer.

However, the underlying theory of MMRA has some gaps which detract from the system. For example, the problem of *pseudo-overflow* will restrain any wide application of MMRA for calculating unknown variables. Consequently, this method is acceptable for solving all problems when the magnitude of the solution is known, but, this method is unacceptable for solving problems when the magnitude of the solution is unknown.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] Andrews, G.E. [1971], Number Theory, W.B. Saunders Co., Philadelphia, Pa.
- [2] Burden, R.L, Faires, J.D., and Reynolds, A.C. [1978], Numerical Analysis, Prindle, Weber and Schmidt, Boston, Mass.
- [3] Ferguson and Matula[1985], "Rationally Based Arithmetic," Warren E. Ferguson, Jr. and David W. Matula, Proceedings 7th Symposium on Computer Arithmetic, IEEE Computer Society Press, Silver Spring, M.D.
- [4] Gregory, R.T. [1980], Error-Free Computation, Robert E. Krieger Pub. Co., Melbourne, Florida.
- [5] Gregory, R.T. [1980], Error-Free Computations: Methods and Applications, class notes, Special Topics in Computer Science, Winter Quarter, 1984.
- [6] Gregory, R.T. [1981], "ERROR-FREE COMPUTATION WITH RATIONAL NUMBERS," *BIT*, 21, 194-202.
- [7] Gregory, R.T. and Karney, D.L. [1978], A Collection of Matrices for Testing Computational Algorithms, Robert E. Krieger Pub. Co., Melbourne, Florida.
- [8] Gregory, R.T. and Krishnamurthy E.V. [1984], Methods and applications of Error-Free Computaiton, Springer-Verlag, New York Inc., New York, New York.
- [9] Hardy, G.H. and Wright, E.W. [1960], An Introduction to the Theory of Numbers, (4th ed.) Clarendon Press, Oxford.
- [10] Howell, J.A. and Gregory, R.T. [1970], "SOLVING LINEAR EQUATIONS USING RESIDUE ARITHMETIC - ALGORITHM II", *BIT*, 10, 23-27.
- [11] Hwang, S.H. [1981], COMPUTATION IN A FINITE FIELD USING RATIONAL OPERANDS, Master's Thesis, Department of Computer Science, University of Tennessee, Knoxville, Tenn.
- [12] Kornerup, P. and Gregory R.T. [1982], "MAPPING INTEGERS AND HENSEL CODES ONTO FAREY FRACTIONS", *BIT*, 23, 9-20.

- [13] Smyre, J.S. [1983], EXACT COMPUTATION USING EXTENDED-PRECISION SINGLE-MODULUS RESIDUE ARITHMETIC, Master's Thesis. Department of Computer Science, University of Tennessee, Knoxville, Tenn.
- [14] Szabo, N.S. and Tanaka, R.I. [1967], Residue Arithmetic and its Applications to Computer Technology, McGraw-Hill, New York.
- [15] Taylor, J.T. [1984], "Residue Arithmetic: A Tutorial with Examples", Computer, 18, 50-62.
- [16] Young, D.M. and Gregory, R.T. [1973], A Survey of Numerical Mathematics, Vol. II, Addison-Wesley, Reading, Massachusetts.

APPENDIX

C PROGRAMS

```
#include <stdio.h>

/*
    This is the header file called "hilbert.h".

    These are the data types utilized in the multiple-modulus residue
    arithmetic routines.
*/

#define NUMBER_LENGTH 100

typedef struct array {
    int sign;
    int array_[100];    /* NUMBER_LENGTH */ } ARRAY_;

typedef struct base_ {
    int num_moduli;
    ARRAY_ value_[100]; /* NUMBER_LENGTH */
    int decimal_[100];  /* NUMBER_LENGTH */ } BASE_VECTOR;

typedef struct fraction {
    int sign[2];
    int array_[2][100];
} FRACTION;

typedef int FRACTION_ARRAY[2][100];
typedef int FRACTION_INT[2];
```

```

#include <stdio.h>
#include "hilbert.h"

main()
/* This routine performs Gaussian Elimination on a (N x N)
   matrix. This routine is tested on a 4 x 4 Hilbert Matrix.
   */
{
    ARRAY_ matrix[4][4],
        solution[4][4];

    BASE_VECTOR base, element;

    ARRAY_ srd, num_1, num_2, den_1, den_2, n_value, t_1, t_2, t_3,
        result, n_1, d_1;

    int i, j, k, l, tmp, row, col, place;

    FRACTION rational, f_1, f_2;

    /* n = 4;          The Hilbert Matrix size */

    get_moduli(&base);
    printf("\nThe base vector:");
    print_base_vector(&base);

    for (i = 1; i <= 4; i++)
        for (j = 1; j <= 4; j++) {
            /* set up denominator */

            rational.array_[1][0] = 1;
            rational.array_[1][1] = i + j - 1;
            rational.sign[1] = 1;

            rational.array_[0][0] = 1;
            rational.array_[0][1] = 1;
            rational.sign[0] = 1;

            a_b_beta(&rational, &base, &element);

            for (k = 0; k <= element.num_moduli; k++)
                matrix[i - 1][j - 1].array_[k + 1] = element.decimal_[k];

            matrix[i - 1][j - 1].array_[0] = element.num_moduli;
            matrix[i - 1][j - 1].sign = 1;

        } /* i, j */

    /* Initialize the solution matrix as the identity matrix */
    for (col = 0; col < 4; col++)

```

```

for (row = 0; row < 4; row++)
  if (row == col) {
    solution[row][col].array_[0] = 3;
    solution[row][col].array_[1] = 1;
    solution[row][col].array_[2] = 1;
    solution[row][col].array_[3] = 1;
    solution[row][col].sign = 1;
  } else {
    solution[row][col].array_[0] = 3;
    solution[row][col].array_[1] = 0;
    solution[row][col].array_[2] = 0;
    solution[row][col].array_[3] = 0;
    solution[row][col].sign = 1;
  }

/*      Forward elimination of Gaussian Elimination      */
for (col = 0; col < 4; col++)
  for (row = col; row < 4; row++)
    if (row == col) { /* t_1 is the multiplicative inverse */
      place = row; /* of the diagonal elements. */

      for (k = 1; k <= matrix[row][col].array_[0]; k++) {
        tmp = matrix[row][col].array_[k];
        forward_map_integer(tmp, base.decimal_[k - 1], &1);
        t_1.array_[k] = 1;
      }
      t_1.sign = 1;
      t_1.array_[0] = matrix[row][col].array_[0];

      for (k = 0; k < 4; k++) {
        for (l = 0; l <= matrix[row][k].array_[0]; l++)
          srd.array_[l] = matrix[row][k].array_[l];
        srd.sign = matrix[row][k].sign;

        multiply_standard_residues(&t_1, &srd, &base, &result);

        for (l = 0; l <= result.array_[0]; l++)
          matrix[row][k].array_[l] = result.array_[l];
        matrix[row][k].sign = result.sign;

        for (l = 0; l <= solution[row][k].array_[0]; l++)
          srd.array_[l] = solution[row][k].array_[l];
        srd.sign = solution[row][k].sign;

        multiply_standard_residues(&t_1, &srd, &base, &result);

        for (l = 0; l <= result.array_[0]; l++)
          solution[row][k].array_[l] = result.array_[l];
        solution[row][k].sign = result.sign;
      } /* k loop */
    } else {

```

```

for (k = 1; k <= matrix[row][col].array_[0]; k++) {
    tmp = base.decimal_[k - 1] -
        matrix[row][col].array_[k];
    t_1.array_[k] = reduce_integer(tmp,
        base.decimal_[k - 1]);
}
t_1.sign = 1;
t_1.array_[0] = matrix[row][col].array_[0];

/* t_1 is the additive inverse of the matrix element */
for (k = 0; k < 4; k++) {
    for (l = 0; l <= matrix[place][k].array_[0]; l++)
        srd.array_[l] = matrix[place][k].array_[l];
    srd.sign = matrix[place][k].sign;

    multiply_standard_residues(&t_1, &srd, &base, &t_2);

    for (l = 0; l <= matrix[row][k].array_[0]; l++)
        srd.array_[l] = matrix[row][k].array_[l];
    srd.sign = matrix[row][k].sign;

    add_standard_residues(&t_2, &srd, &base, &t_3);

    for (l = 0; l <= t_3.array_[0]; l++)
        matrix[row][k].array_[l] = t_3.array_[l];
    matrix[row][k].sign = 1;

    /* duplicate the mathematics on the solution matrix */
    for (l = 0; l <= solution[place][k].array_[0]; l++)
        srd.array_[l] = solution[place][k].array_[l];
    srd.sign = solution[place][k].sign;

    multiply_standard_residues(&t_1, &srd, &base, &t_2);

    for (l = 0; l <= solution[row][k].array_[0]; l++)
        srd.array_[l] = solution[row][k].array_[l];
    srd.sign = solution[row][k].sign;

    add_standard_residues(&t_2, &srd, &base, &t_3);

    for (l = 0; l <= t_3.array_[0]; l++)
        solution[row][k].array_[l] = t_3.array_[l];
    solution[row][k].sign = 1;
} /* End k loop */
} /* End else */

```

```

/*      Backward Substitution in Gaussian Elimination      */
for (col = 3; col >= 0; col--)
  for (row = col; row >= 0; row--)
    if (row == col)
      place = col;
    else {
      /* Get the additive inverse. */
      for (k = 1; k <= matrix[row][col].array_[0]; k++) {
        tmp = base.decimal_[k - 1] -
              matrix[row][col].array_[k];
        t_1.array_[k] = reduce_integer(tmp,
                                       base.decimal_[k - 1]);
      }
      t_1.array_[0] = base.num_moduli;
      t_1.sign = 1;

      for (k = 0; k < 4; k++) {
        for (l = 0; l <= matrix[place][k].array_[0]; l++)
          srd.array_[l] = matrix[place][k].array_[l];
        srd.sign = matrix[place][k].sign;

        multiply_standard_residues(&t_1, &srd, &base, &t_2);

        for (l = 0; l <= matrix[row][k].array_[0]; l++)
          srd.array_[l] = matrix[row][k].array_[l];
        srd.sign = matrix[row][k].sign;

        add_standard_residues(&t_2, &srd, &base, &t_3);

        for (l = 0; l <= t_3.array_[0]; l++)
          matrix[row][k].array_[l] = t_3.array_[l];
        matrix[row][k].sign = 1;

        /* duplicate the mathematics on the solution matrix */
        for (l = 0; l <= solution[place][k].array_[0]; l++)
          srd.array_[l] = solution[place][k].array_[l];
        srd.sign = solution[place][k].sign;

        multiply_standard_residues(&t_1, &srd, &base, &t_2);

        for (l = 0; l <= solution[row][k].array_[0]; l++)
          srd.array_[l] = solution[row][k].array_[l];
        srd.sign = solution[row][k].sign;

        add_standard_residues(&t_2, &srd, &base, &t_3);

        for (l = 0; l <= t_3.array_[0]; l++)
          solution[row][k].array_[l] = t_3.array_[l];
        solution[row][k].sign = 1;
      }
    } /* End k loop */
  } /* End else */

```

```

printf("\f Here is the final matrix\n");
for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++) {
        for (k = 0; k < NUMBER_LENGTH; k++)
            srd.array_[k] = matrix[i][j].array_[k];
        srd.sign = 1;
        printf("\nThe standard residue digits for {%d, %d}:\n",
            i, j);
        print_(&srd);
        order_n(&base, &srd, &n_value);
        recover(&srd, &base, &rational, &n_value);
        printf("\nThe recovered fraction:\n");
        print_fraction(&rational);
    }

printf("\f Here is the final solution matrix\n");
for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++) {
        for (k = 0; k < NUMBER_LENGTH; k++)
            srd.array_[k] = solution[i][j].array_[k];

        srd.sign = 1;
        printf("\nThe standard residue digits for {%d, %d}:\n",
            i, j);
        print_(&srd);
        order_n(&base, &srd, &n_value);
        recover(&srd, &base, &rational, &n_value);
        printf("\nThe recovered fraction:\n");
        print_fraction(&rational);
    }
}

```

```

a_b_beta(rational, base, answer)
/*
    This routine "reduces" the rational number to lowest terms.
    Then this number is "forward mapped" w.r.t. the base_vector.
*/

FRACTION
    *rational;

BASE_VECTOR
    *base,
    *answer;

{
    int i, place, numerator_sign, denominator_sign;

    ARRAY_ numerator, denominator, temp_numerator, temp_denominator,
        new_numerator, new_denominator, common_factor, zero, one,
        negative_one;

    /* Set up arrays */
    for (i = 0; i < NUMBER_LENGTH; i++) {
        numerator.array_[i] = rational->array_[0][i];
        denominator.array_[i] = rational->array_[1][i];
    }
    if (rational->sign[0] == rational->sign[1]) {
        numerator_sign = 1;
        denominator_sign = 1;
    } else {
        numerator_sign = -1;
        denominator_sign = 1;
    }

    numerator.sign = numerator_sign;
    denominator.sign = denominator_sign;

    new_numerator.array_[0] = 0;
    new_denominator.array_[0] = 0;
    zero.array_[0] = 0;
    zero.sign = 1;
    one.array_[0] = 1;
    one.array_[1] = 1;
    one.sign = 1;
    negative_one.array_[0] = 1;
    negative_one.array_[1] = 1;
    negative_one.sign = -1;

    gcd_array(&denominator, &numerator, &common_factor);

    i = compare(&common_factor, &one);

    if (compare(&common_factor, &one) > 0) {
        if (compare(&numerator, &zero) > 0)
            while (compare(&numerator, &zero) > 0) {

```

```

    sub_array(&numerator, &common_factor, &temp_numerator);

    for (i = 0; i <= temp_numerator.array_[0]; i++)
        numerator.array_[i] = temp_numerator.array_[i];
    numerator.sign = temp_numerator.sign;

    add_array(&one, &new_numerator, &temp_numerator);

    for (i = 0; i <= temp_numerator.array_[0]; i++)
        new_numerator.array_[i] = temp_numerator.array_[i];
    new_numerator.sign = temp_numerator.sign;
}
else
    while (compare(&numerator, &negative_one) < 0) {
        add_array(&numerator, &common_factor, &temp_numerator);

        for (i = 0; i <= temp_numerator.array_[0]; i++)
            numerator.array_[i] = temp_numerator.array_[i];
        numerator.sign = temp_numerator.sign;

        sub_array(&new_numerator, &one, &temp_numerator);

        for (i = 0; i <= temp_numerator.array_[0]; i++)
            new_numerator.array_[i] = temp_numerator.array_[i];
        new_numerator.sign = temp_numerator.sign;
    }

    while (compare(&denominator, &zero) > 0) {
        sub_array(&denominator, &common_factor, &temp_denominator);

        for (i = 0; i <= temp_denominator.array_[0]; i++)
            denominator.array_[i] = temp_denominator.array_[i];
        denominator.sign = temp_denominator.sign;

        add_array(&one, &new_denominator, &temp_denominator);
        for (i = 0; i <= temp_denominator.array_[0]; i++)
            new_denominator.array_[i] = temp_denominator.array_[i];
        new_denominator.sign = temp_denominator.sign;
    }

    for (i = 0; i <= new_numerator.array_[0]; i++)
        numerator.array_[i] = new_numerator.array_[i];
    numerator.sign = new_numerator.sign;

    for (i = 0; i <= new_denominator.array_[0]; i++)
        denominator.array_[i] = new_denominator.array_[i];
    denominator.sign = new_denominator.sign;
}
numerator.sign = numerator_sign;
denominator.sign = denominator_sign;

place = 0;
divide_array(&numerator, &denominator, answer, base, place);
} /* end a_b_beta */

```

```

#include <stdio.h>
#include "hilbert.h"

add_array(a_1,a_2,answer)
/*
  This routine adds two arrays positionally; the result is in answer.
*/

  ARRAY_
    *a_1,
    *a_2,
    *answer;
  {
    ARRAY_
      a1,
      a2;

    int
      i, j, cary, l, a1_sign, a2_sign;

    for (i = 1; i < NUMBER_LENGTH; i++)
      answer->array_[i] = 0;

    answer->array_[0] = a_1->array_[0] + a_2->array_[0];

    a1_sign = a_1->sign;
    a2_sign = a_2->sign;
    a_1->sign = 1;
    a_2->sign = 1;

    if (compare(a_1, a_2) >= 0)
      for (i = 0; i <= a_1->array_[0]; i++) /* a_1 >= a_2 */
        {
          a1.array_[i] = a_1->array_[i];
          a2.array_[i] = a_2->array_[i];
        }
      a1.sign = a1_sign;
      a2.sign = a2_sign;
    } else {
      for (i = 0; i <= a_2->array_[0]; i++)
        {
          a1.array_[i] = a_2->array_[i];
          a2.array_[i] = a_1->array_[i];
        }
      a1.sign = a2_sign;
      a2.sign = a1_sign;
    }

    a_1->sign = a1_sign;
    a_2->sign = a2_sign;

    /* add arrays with like signs */

    if (a1.sign == a2.sign)
      for (i = 1; i <= a1.array_[0]; i++)
        if (a2.array_[0] < i)
          answer->array_[i] = a1.array_[i];
        else
          answer->array_[i] = a1.array_[i] + a2.array_[i];
    /* reduce */

```

```

    for (i = 1; i <= answer->array_[0]; i++)
        if (answer->array_[i] >= 10) {
            cary = answer->array_[i] / 10;
            answer->array_[i] = answer->array_[i] - (10 * cary);
            answer->array_[i + 1] = answer->array_[i + 1] + cary;
        }
    } else {
        /* add arrays with unlike signs */

        for (i = 1; i <= a1.array_[0]; i++)
            if (a2.array_[0] < i)
                answer->array_[i] = a1.array_[i];
            else
                answer->array_[i] = a1.array_[i] - a2.array_[i];

        /* reduce */

        for (i = 1; i <= answer->array_[0]; i++)
            if (answer->array_[i] < 0) {
                answer->array_[i] = answer->array_[i] + 10;
                answer->array_[i + 1] = answer->array_[i + 1] - 1;
            }
    }

    for (i = answer->array_[0]; i > 0; i--)
        if (answer->array_[i] != 0)
            break;

    answer->array_[0] = i;

    if (answer->array_[0] == 0)
        answer->sign = 1;
    else
        answer->sign = a1.sign;
} /* end add */

```

```

#include <stdio.h>
#include "hilbert.h"

add_standard_residues(a_1, a_2, base, answer)
/*
    This routine subtracts two standard residue representations in
    componentwise fashion, reducing each result with respect to the
    modulus.
*/
    BASE_VECTOR
    *base;

    ARRAY_
    *a_1, *a_2, *answer;

    {
    int i, temp;

    answer->sign = 1;

    for (i = 1; i <= base->num_moduli; i++) {
        if (a_1->array_[i] == -2 || a_2->array_[i] == -2) {
            answer->array_[i] = -2;
            continue;
        }

        if (a_1->array_[i] == -1 || a_2->array_[i] == -1) {
            if (a_1->array_[i] == -1)
                if (a_2->array_[i] > -1)
                    answer->array_[i] = -1;

            if (a_2->array_[i] == -1)
                if (a_1->array_[i] > -1)
                    answer->array_[i] = -1;
                else
                    answer->array_[i] = -2;
            continue;
        }

        temp = a_1->array_[i] + a_2->array_[i];
        answer->array_[i] = reduce_integer(temp, base->decimal_[i-1]);
    }

    answer->array_[0] = base->num_moduli;
    }

```

```

#include <stdio.h>
#include "hilbert.h"

big_m(base)
/*
    Find the product of the moduli.

    BASE_VECTOR
    *base;

    {
    ARRAY_
    temp_1, temp_2, temp_answer;
    int
    i, j;

    for ( i = 0; i < base->num_moduli; i++) {
        if ( i == 0) { /* multiply initially by 1 */
            temp_2.array_[0] = 1;
            temp_2.array_[1] = 1;
            temp_2.sign = 1;
            temp_1.sign = 1;
        }
        for ( j = 0; j <= base->value_[i].array_[0]; j++)
            temp_1.array_[j] = base->value_[i].array_[j];
        multiply_array(&temp_1, &temp_2, &temp_answer);
        for ( j = 0; j <= temp_answer.array_[0]; j++)
            temp_2.array_[j] = temp_answer.array_[j];
        if (temp_2.array_[0] >= 65 && i < (base->num_moduli - 1)) {
            printf("\nFATAL ERROR: insufficient array size.");
            exit(1);
        }
    } /* end for */
} /* end program */

```

```

#include "hilbert.h"

compare(a1,a2)
/*
  This routine compares two integer arrays.

  Return values :  0 - a1 = a2
                  1 - a1 > a2
                 -1 - a1 < a2
*/
  ARRAY_
  *a1,
  *a2;
  {
    int
      i, j;

    /*
      Compare the longest array to the shortest, testing the signs
      for exact magnitude comparison.

      step 1: length(a1) > length(a2)
    */

    if (a1->sign != a2->sign)
      if (a1->sign == 1)
        return(1);
      else
        return(-1);

    if ((a1->array_[0] == 0) && (a2->array_[0] == 0))
      return(0);

    if (a1->array_[0] > a2->array_[0])
      for ( i = a1->array_[0]; i > 0; i--)
        if ((a1->array_[i] > 0) && (i > a2->array_[0]))
          if (a1->sign == 1)
            return(1);
          else
            return(-1);
        else if ((i <= a2->array_[0]) && (a1->array_[i] > a2->array_[i]))
          if (a1->sign == 1)
            return(1);
          else
            return(-1);
        else if ((i <= a2->array_[0]) &&
          (a1->array_[i] < a2->array_[i]))
          if (a2->sign == 1)
            return(-1);
          else
            return(1);

    /*

```

```

    /* step2: length(a1) <= length(a2)
    if (a1->array_[0] <= a2->array_[0])
    for (i = a2->array_[0]; i > 0; i--)
        if ((a2->array_[i] > 0) && (i > a1->array_[0]))
            if (a2->sign == 1)
                return(-1);
            else
                return(1);

        else if ((i <= a1->array_[0]) && (a2->array_[i] > a1->array_[i]))
            if (a2->sign == 1)
                return(-1);
            else
                return(1);
        else if ((i <= a1->array_[0]) &&
            (a1->array_[i] > a2->array_[i]))
            if (a1->sign == 1)
                return(1);
            else
                return(-1);

    /*
    step 3:
    The numbers in the array are identical, check the signs to determine
    equality.
    */

    if (a1->sign == a2->sign)
        return(0);
    else if (a1->sign == 1)
        return(1);
    else
        return(-1);
} /* end compare routine */

```

```

#include "hilbert.h"

convert(t_minus_d, base, s_d_set)
/*
    This routine recovers the s_digit_set from the mixed radix
    digit set, passed in t_minus_d.
*/

    BASE_VECTOR
        *t_minus_d,
        *base,
        *s_d_set;

    {
        BASE_VECTOR
            t_m_d;

        ARRAY_
            t_beta, m_inverse, temp_t_m_d, d_beta, quotient, d,
            last_modulus, remainder, product, modulus, temp_t, one, dummy;

        int i, k, place;

/*
        Initialize t_m_d to t_minus_d. This protects the integrity of the
        passed array.
*/

        one.array_[0] = 1;
        one.array_[1] = 1;
        one.sign = 1;

        if (base->num_moduli > 0) {
            d.sign = t_minus_d->value_[0].sign;
            s_d_set->value_[0].sign = 1;
            s_d_set->num_moduli = base->num_moduli;

            for (k = 0; k <= t_minus_d->value_[0].array_[0]; k++) {
                d.array_[k] = t_minus_d->value_[0].array_[k];
                s_d_set->value_[0].array_[k] = t_minus_d->value_[0].array_[k];
            }
        } else {
            for (k = 0; k <= t_minus_d->value_[0].array_[0]; k++)
                s_d_set->value_[0].array_[k] = 0;

            s_d_set->value_[0].sign = 1;
            return(0);
        }

        for (i = 0; i < base->num_moduli; i++) {
            for (k = 0; k <= t_minus_d->value_[i].array_[0]; k++)
                t_m_d.value_[i].array_[k] = t_minus_d->value_[i].array_[k];
            t_m_d.value_[i].sign = t_minus_d->value_[i].sign;
            t_m_d.num_moduli = t_minus_d->num_moduli;
        }
    }

```

```

for (i = 1; i < base->num_moduli; i++) {
    for (k = 0; k <= base->value_[i - 1].array_[0]; k++)
        last_modulus.array_[k] = base->value_[i - 1].array_[k];
    last_modulus.sign = 1;

    for (place = 1; place < base->num_moduli; place++) {
        for (k = 0; k <= base->value_[place].array_[0]; k++)
            modulus.array_[k] = base->value_[place].array_[k];
        modulus.sign = 1;

        for (k = 0; k <= t_m_d.value_[place].array_[0]; k++)
            t_beta.array_[k] = t_m_d.value_[place].array_[k];
        t_beta.sign = t_m_d.value_[place].sign;

        sub_array(&t_beta, &d_, &temp_t);
        mod_array(&temp_t, &modulus, &remainder, &dummy);
        forward_map_array(&last_modulus, &one, &modulus, &quotient);
        multiply_array(&remainder, &quotient, &product);
        mod_array(&product, &modulus, &remainder, &dummy);

        for (k = 0; k <= remainder.array_[0]; k++)
            t_m_d.value_[place].array_[k] = remainder.array_[k];

        t_m_d.value_[place].sign = remainder.sign;
    }
    /* end for */

    for (k = 0; k <= t_m_d.value_[i].array_[0]; k++) {
        s_d_set->value_[i].array_[k] = t_m_d.value_[i].array_[k];
        d_.array_[k] = t_m_d.value_[i].array_[k];
    }
    /* end for k = 0 */
    s_d_set->value_[i].sign = t_m_d.value_[i].sign;
    d_.sign = t_m_d.value_[i].sign;
}
/* end for i = 1 */

}
/* end program */

```

```

#include "hilbert.h"

decimal_value(value)

/*
Find the decimal value in an array, where the least significant digit
is in the 1st position, and the most significant digit is in the last
used position of the array. The exact position of the last significant
digit is in the .array_[0]-th position.
*/
ARRAY_
*value;

{
    int
        i,
        decimal_val,
        tens;

    if (value->array_[0] >= 10) {
        printf("\nFatal error in decimal_.");
        printf("\nThe array you wanted to represent as an integer is:\n");
        print_(value);
        exit(1);
    }

    tens = 1;
    decimal_val = 0;

    for (i = 1; i <= value->array_[0]; i++) {
        decimal_val = value->array_[i] * tens + decimal_val;
        tens = tens * 10;
    }

    return(decimal_val);
}

```

```

#include "hilbert.h"
digit_array(digit_set, posn, answer)
/*
  This routine sets up the answer array with the integer value stored at the
  digit_set_[posn].
*/

  ARRAY_
    *digit_set,
    *answer;

  int
    posn;

  {
    int
      temp, value, i, digit;

    i = 0;
    value = digit_set->array_[posn];
    while (value > 0) {
      temp = value / 10;
      answer->array_[++i] = value - (temp * 10);
      value = (value - answer->array_[i]) / 10;
    }
    answer->sign = digit_set->sign;
    answer->array_[0] = i;
  } /* end program */

```

```

#include "hilbert.h"

farey_fraction_array(denominator_, numerator_, modulus_, answer_, n_value_)
/*
    This routine forward maps the rational number onto an integer in the
    range 0,1,2,..., (modulus-1).
*/

ARRAY_
*denominator_,
*numerator_,
*modulus_,
*n_value_;

FRACTION
*answer_;

{
    int i, multiplicand, j, k, numerator_sign;

    ARRAY_
        temp_modulus, denominator, numerator, last_answer, temp_denominator,
        temp_numerator, answer, modulus, multiplicand_array, temp,
        result, last_result, zero, ret_val;

    /* initialize */

    for (i = 0; i < NUMBER_LENGTH; i++) {
        modulus.array_[i] = modulus_>array_[i];
        temp_modulus.array_[i] = modulus_>array_[i];
        denominator.array_[i] = denominator_>array_[i];
        numerator.array_[i] = numerator_>array_[i];
        last_answer.array_[i] = 0;
        answer_>array_[i][1] = numerator_>array_[i];
        answer_>array_[0][i] = denominator_>array_[i];
    }

    answer_>sign[0] = 1;
    answer_>sign[1] = 1;
    numerator.sign = numerator_>sign;
    denominator.sign = denominator_>sign;

    temp_modulus.sign = 1;
    last_answer.sign = 1;

    modulus.sign = modulus_>sign;

    if (compare(n_value_, &denominator) != -1)
        if (compare(n_value_, &numerator) != -1) {
            for (i = 0; i < NUMBER_LENGTH; i++) {
                answer_>array_[0][i] = denominator.array_[i];
                answer_>array_[1][i] = numerator.array_[i];
                answer_>sign[0] = numerator.sign;
                answer_>sign[1] = denominator.sign;
            }
        }
}

```

```

    } return(1);
}

zero.sign = 1;
zero.array_[0] = 0;

/*      page 3 of gcd      */

mod_array(&denominator, &temp_modulus, &temp_denominator, &ret_val);
if (compare(&ret_val, &zero) != 0) {
    for (k = 0; k <= temp_denominator.array_[0]; k++)
        denominator.array_[k] = temp_denominator.array_[k];
    denominator.sign = temp_denominator.sign;
}

mod_array(&numerator, &temp_modulus, &temp_numerator, &ret_val);
if (compare(&ret_val, &zero) != 0) {
    for (k = 0; k <= temp_numerator.array_[0]; k++)
        numerator.array_[k] = temp_numerator.array_[k];
    numerator.sign = temp_numerator.sign;
}

sub_array(&denominator, &numerator, &result);
if (result.array_[0] == 0)
    return(0);

while (denominator.array_[0] > 0) {
    mod_array(&temp_modulus, &denominator, &temp_denominator,
        &multiplicand_array);
    if (last_answer.array_[0] == 0) {
        for (i = 0; i <= numerator.array_[0]; i++)
            last_answer.array_[i] = numerator.array_[i];
        multiply_array(&multiplicand_array, &last_answer, &answer);
        numerator.sign = answer.sign * -1;
        for (i = 0; i <= answer.array_[0]; i++)
            numerator.array_[i] = answer.array_[i];
    } else {
        for (i = 0; i <= numerator.array_[0]; i++)
            temp.array_[i] = numerator.array_[i];
        temp.sign = numerator.sign;
        multiply_array(&numerator, &multiplicand_array, &answer);
        answer.sign = answer.sign * -1;
        add_array(&last_answer, &answer, &numerator);
        for (i = 0; i <= temp.array_[0]; i++)
            last_answer.array_[i] = temp.array_[i];
    }
}

```

```

    last_answer.sign = temp.sign;
}

/* common code with gcd */
for (i = 0; i < NUMBER_LENGTH; i++) {
    temp_modulus.array_[i] = denominator.array_[i];
    denominator.array_[i] = temp_denominator.array_[i];
}
temp_modulus.sign = denominator.sign;
denominator.sign = temp_denominator.sign;

for (i = 0; i <= result.array_[0]; i++)
    last_result.array_[i] = result.array_[i];

last_result.sign = 1;
result.sign = 1;

numerator_sign = numerator.sign;

if (compare(n_value_, &denominator) != -1) {
    if (compare(n_value_, &numerator) != -1) {
        for (i = 0; i < NUMBER_LENGTH; i++) {
            answer->array_[0][i] = denominator.array_[i];
            answer->array_[1][i] = numerator.array_[i];
            answer->sign[0] = numerator_sign;
            answer->sign[1] = 1;
        }
        return(1);
    }
} else
    numerator.sign = numerator_sign;
}
/* end while */
/* end inverse map array */

```

```

#include "hilbert.h"

forward_map_array(denominator_, numerator_, modulus_, answer_)
/*
    This routine forward maps the rational number onto an integer in the
    range 0,1,2,...,(modulus-1).
*/

ARRAY_
*denominator_,
*numerator_,
*modulus_,
*answer_;

{
    int i, multiplicand, j, k;

    ARRAY_
    temp_modulus, denominator, numerator, last_answer, temp_denominator,
    temp_numerator, answer, modulus, multiplicand_array, temp, zero,
    ret_val;

    /* initialize */
    for (i = 0; i < NUMBER_LENGTH; i++) {
        modulus.array_[i] = modulus_>array_[i];
        temp_modulus.array_[i] = modulus_>array_[i];
        denominator.array_[i] = denominator_>array_[i];
        numerator.array_[i] = numerator_>array_[i];
        last_answer.array_[i] = 0;
    }
    zero.array_[0] = 0;          zero.sign = 1;

    numerator.sign = numerator_>sign;
    denominator.sign = denominator_>sign;
    temp_modulus.sign = 1;
    last_answer.sign = 1;
    answer_>sign = 1;
    answer_>array_[0] = 0;
    modulus.sign = modulus_>sign;

    mod_array(&denominator, &temp_modulus, &temp_denominator, &ret_val);

    if (compare(&ret_val, &zero) != 0) {
        for (k = 0; k <= temp_denominator.array_[0]; k++)
            denominator.array_[k] = temp_denominator.array_[k];
        denominator.sign = temp_denominator.sign;
    }

    mod_array(&numerator, &temp_modulus, &temp_numerator, &ret_val);

    if (compare(&ret_val, &zero) != 0) {
        for (k = 0; k <= temp_numerator.array_[0]; k++)
            numerator.array_[k] = temp_numerator.array_[k];
    }
}

```

```

    numerator.sign = temp_numerator.sign;
}
while (denominator.array_[0] > 0) {
    mod_array(&temp_modulus, &denominator,
        &temp_denominator, &multiplicand_array);
    if (last_answer.array_[0] == 0) {
        for (i = 0; i <= numerator.array_[0]; i++)
            last_answer.array_[i] = numerator.array_[i];
        multiply_array(&multiplicand_array, &last_answer, &answer);
        numerator.sign = answer.sign * -1;
        for (i = 0; i <= answer.array_[0]; i++)
            numerator.array_[i] = answer.array_[i];
    } else {
        for (i = 0; i <= numerator.array_[0]; i++)
            temp.array_[i] = numerator.array_[i];
        temp.sign = numerator.sign;
        multiply_array(&numerator, &multiplicand_array, &answer);
        answer.sign = answer.sign * -1;
        add_array(&last_answer, &answer, &numerator);
        for (i = 0; i <= temp.array_[0]; i++)
            last_answer.array_[i] = temp.array_[i];
        last_answer.sign = temp.sign;
    }
    for (i = 0; i < NUMBER_LENGTH; i++) {
        temp_modulus.array_[i] = denominator.array_[i];
        denominator.array_[i] = temp_denominator.array_[i];
    }
    temp_modulus.sign = denominator.sign;
    denominator.sign = temp_denominator.sign;
}
/* end while */
mod_array(&last_answer, modulus_, answer_, &ret_val);
}
/* end inverse map array */
*/

```

```

#include <stdio.h>
#include "hilbert.h"

forward_map_integer(value, modulus, answer)
/*
    This maps the rational number ( 1 /denominator) onto
    a member of the set 0,1,2,...,(modulus-1), using the
    extended Euclidian algorithm.
*/
int
value,
modulus,
*answer;

{
    int i, numerator, denominator, temp,
        multiplicand, last_answer, return_answer, original_modulus;

    original_modulus = modulus;
    last_answer = 0;

    if (value == 0 || value == 1) {
        *answer = value;
        return(value);
    }

    if (value > 0) {
        denominator = value;
        numerator = 1;
    } else {
        denominator = value * -1;
        numerator = -1;
    }

    if (denominator != original_modulus)
        denominator = reduce_integer(denominator, original_modulus);

    while (denominator != 0) {
        multiplicand = modulus/denominator;

        if (last_answer == 0) {
            last_answer = numerator;
            numerator = (multiplicand * last_answer) * -1;
        } else {
            temp = numerator;
            numerator = (numerator * multiplicand * -1) + last_answer;
            last_answer = temp;
        }
        temp = denominator;
        denominator = modulus - (multiplicand * denominator);
        modulus = temp;
    }
    /* end while */
    return_answer = reduce_integer(last_answer, original_modulus);
    *answer = return_answer;

    return(return_answer);
}
/* end forward map integers */

```

```

#include "hilbert.h"

gcd_array(value_, modulus_, answer_)
/*
    This routine forward maps the rational number onto an integer in the
    range 0,1,2,...,(modulus-1).
*/
ARRAY_
    *value_,
    *modulus_,
    *answer_;

{
    int i, k, multiplicand, value_sign, modulus_sign;

    ARRAY_
        temp_modulus, denominator, numerator, last_answer, temp_denominator,
        answer, dummy, modulus, multiplicand_array, temp, common_factor;

    /* initialize */

    value_sign = value_>sign;          modulus_sign = modulus_>sign;
    value_>sign = 1;                    modulus_>sign = 1;

    answer_>sign = 1;                  answer_>array_[0] = 1;
    answer_>array_[1] = 1;

    if (((value_>array_[0] == 1) && (value_>array_[1] == 1)) ||
        ((modulus_>array_[0] == 1) && (modulus_>array_[1] == 1))) {
        value_>sign = value_sign;      modulus_>sign = modulus_sign;
        return(1);
    }

    if (compare(value_, modulus_) <= 0)
        for (i = 0; i < NUMBER_LENGTH; i++) {
            temp_modulus.array_[i] = modulus_>array_[i];
            denominator.array_[i] = value_>array_[i];
            last_answer.array_[i] = 0;
        }
    else
        for (i = 0; i < NUMBER_LENGTH; i++) {
            temp_modulus.array_[i] = value_>array_[i];
            denominator.array_[i] = modulus_>array_[i];
            last_answer.array_[i] = 0;
        }

    numerator.sign = 1;                numerator.array_[0] = 1;
    numerator.array_[1] = 1;
    denominator.sign = 1;
    temp_modulus.sign = 1;
    last_answer.sign = 1;
    answer_>sign = 1;

```

```

answer->array_[0] = 0;
/* begin common code */
while (denominator.array_[0] > 0) {
mod_array(&temp_modulus, &denominator,
        &temp_denominator, &multiplicand_array);
if (last_answer.array_[0] == 0) {
    for (k = 0; k <= numerator.array_[0]; k++)
        last_answer.array_[k] = numerator.array_[k];
    last_answer.sign = numerator.sign;
    multiply_array(&multiplicand_array, &last_answer, &answer);
    numerator.sign = answer.sign * -1;
    for (k = 0; k <= answer.array_[0]; k++)
        numerator.array_[k] = answer.array_[k];
} else {
    for (k = 0; k <= numerator.array_[0]; k++)
        temp.array_[k] = numerator.array_[k];

    temp.sign = numerator.sign;
    multiply_array(&numerator, &multiplicand_array, &answer);
    answer.sign = answer.sign * -1;
    add_array(&last_answer, &answer, &numerator);

    for (k = 0; k <= temp.array_[0]; k++)
        last_answer.array_[k] = temp.array_[k];
    last_answer.sign = temp.sign;
}
/* end common code */
    for (i = 0; i < NUMBER_LENGTH; i++) {
        temp_modulus.array_[i] = denominator.array_[i];
        denominator.array_[i] = temp_denominator.array_[i];
    }
    temp_modulus.sign = denominator.sign;
    denominator.sign = temp_denominator.sign;
}
/* end while */

for (i = 0; i <= temp_modulus.array_[0]; i++)
    answer->array_[i] = temp_modulus.array_[i];
answer->sign = temp_modulus.sign;
value->sign = value_sign;    modulus->sign = modulus_sign;
/* end inverse map array */
}

```

```

#include "hilbert.h"
#define charval(n) ((n) - '0')

get_moduli(base)

/*
  Read the moduli into the base_vector and store character by character in
  b_vector, and total integer value in base_vector.
*/

BASE_VECTOR
*base;

{
    FILE
    *data_in;

    ARRAY_
    temp,
    decimal;

    int i, j, k,
    data;

    data_in = fopen("moduli.dat", "r");

    /* get the number of moduli */
    j = 0;
    decimal.sign = 1;
    for (;;) {
        while (((data = getc(data_in)) == ' ') || (data == '\n') ||
              (data == '\t'))
            ;
        if (data == '$')
            break;
        else
            temp.array_[++j] = charval(data);
    }

    temp.sign = 1;
    temp.array_[0] = j;

    for (k = temp.array_[0]; k > 0; k--) {
        decimal.array_[temp.array_[0] - k + 1] = temp.array_[k];
    }
    decimal.array_[0] = temp.array_[0];
    base->num_moduli = decimal_value(&decimal);

    /* now get the base vector */
    for (i = 0; i < base->num_moduli; i++) {
        j = 0;

```

```

for (;;) {
    while (((data = getc(data_in)) == ' ') || (data == '\n') ||
           (data == '\t'))
        ;
    if (data == '$')
        break;
    else
        temp.array_[++j] = charval(data);
}

temp.array_[0] = j;
temp.sign = 1;

for (k = temp.array_[0]; k > 0; k--) {
    decimal.array_[temp.array_[0] - k + 1] = temp.array_[k];
    base->value_[1].array_[temp.array_[0] - k + 1] = temp.array_[k];
}
base->value_[1].array_[0] = temp.array_[0];
base->value_[1].sign = 1;

decimal.array_[0] = temp.array_[0];
print_(&decimal);
base->decimal_[1] = decimal_value(&decimal);
printf("\n base->decimal_[%d] = %d", 1, base->decimal_[1]);

/* zero the remaining positions in the array */
for (k = (temp.array_[0] + 1); k < NUMBER_LENGTH; k++)
    base->value_[1].array_[k] = 0;
} /* end read base vector */
} /* end program */

```

```

#include "hilbert.h"

M_0(base, residue, m_0_vector)

/*
    This routine finds the product of the base vector moduli which
    correspond to zero.
*/
BASE_VECTOR
*base;

ARRAY_
*m_0_vector,
*residue;
{
    int
        temp,
        digit,
        i, j;

    ARRAY_
        a1, a2, answer;

    a2.sign = 1;
    a2.array_[0] = 1;
    a2.array_[1] = 1;
    a1.sign = 1;

    answer.array_[0] = 1;
    answer.array_[1] = 1;
    answer.sign = 1;

    /*
        remember: the base vector values begin in position 0, and
        the residue vector values begin in position 1.
    */
    for (i = 0; i < base->num_moduli; i++)
        if (residue->array_[i + 1] == 0) {
            for (j = 0; j <= base->value_[i].array_[0]; j++)
                a1.array_[j] = base->value_[i].array_[j];

            multiply_array(&a1, &a2, &answer);

            for (j = 0; j <= answer.array_[0]; j++)
                a2.array_[j] = answer.array_[j];
        }
        /* end if */
    for (i = 0; i <= answer.array_[0]; i++)
        m_0_vector->array_[i] = answer.array_[i];

    m_0_vector->sign = answer.sign;
}
/* end m_0 */

```

```

M_l(base, residue, m_l_vector)
/*
    This routine finds the product of the base vector moduli which
    correspond to zero.
*/
BASE_VECTOR
    *base;

ARRAY_
    *m_l_vector,
    *residue;

{
    int
        temp,
        digit,
        i, j;

    ARRAY_
        a1, a2, answer;

    a2.sign = 1;
    a2.array_[0] = 1;
    a2.array_[1] = 1;
    a1.sign = 1;

    answer.array_[0] = 1;
    answer.array_[1] = 1;
    answer.sign = 1;

    /*
        remenber: the base vector values begin in position 0, and
        the residue vector values begin in position 1.
    */
    for (i = 0; i < base->num_moduli; i++)
        if ((residue->array_[i+1] != 0) && (residue->array_[i+1] != -1)) {
            for (j = 0; j <= base->value_[i].array_[0]; j++)
                a1.array_[j] = base->value_[i].array_[j];

            multiply_array(&a1, &a2, &answer);

            for (j = 0; j <= answer.array_[0]; j++)
                a2.array_[j] = answer.array_[j];

        }
        /* end if */

    for (i = 0; i <= answer.array_[0]; i++)
        m_l_vector->array_[i] = answer.array_[i];

    m_l_vector->sign = answer.sign;
    /* end m_0 */
}

```

```

#include "hilbert.h"

multiply_array(a1,a2,answer)
/*
  This routine multiplies two arrays together, storing the result in answer.
  */
  ARRAY_
    *a1,
    *a2,
    *answer;
  {
    int
      i, j,
      cary;

    /* initialize answer array */

    if (a1->array_[0] == 0)
      a1->sign = 1;
    if (a2->array_[0] == 0)
      a2->sign = 1;

    for (i = 0; i < NUMBER_LENGTH; i++)
      answer->array_[i] = 0;

    /* multiply numerator */

    answer->array_[0] = a1->array_[0] + a2->array_[0];

    if (a1->array_[0] > a2->array_[0])
      for (i = 1; i <= a1->array_[0]; i++)
        for (j = 1; j <= a2->array_[0]; j++)
          answer->array_[i + j - 1] =
            a2->array_[j] * a1->array_[i] +
            answer->array_[i + j - 1];
    else
      for (i = 1; i <= a2->array_[0]; i++)
        for (j = 1; j <= a1->array_[0]; j++)
          answer->array_[i + j - 1] =
            a1->array_[j] * a2->array_[i] +
            answer->array_[i + j - 1];

    /* reduce */

    for (i = 1; i <= answer->array_[0]; i++) {
      if (answer->array_[i] >= 10) {
        cary = answer->array_[i] / 10;
        answer->array_[i] = answer->array_[i] - (10 * cary);
        if (i < answer->array_[0])
          answer->array_[i + 1] = answer->array_[i + 1] + cary;
        else {
          answer->array_[0] = answer->array_[0] + 1;
          answer->array_[1] = cary;
        }
      }
      cary = 0;
    }

    if (a1->sign == a2->sign)
      answer->sign = 1;
    else
      answer->sign = -1;

    for (i = answer->array_[0]; i > 0; i--)
      if (answer->array_[i] != 0)
        break;

    answer->array_[0] = i;
  }

```

```

#include "hilbert.h"

mod_array(value_, modulus_, answer_, return_value)

/*
   This returns an integer which is the # of times the value is reduced by
   the modulus. The remainder is stored in answer.
*/

ARRAY_
*value_,
*modulus_,
*answer_,
*return_value;

{
  int
  i;

  ARRAY_
  temp_value,
  temp_modulus,
  answer,
  dummy,
  zero, one, j, k,
  k_last, j_last;

  temp_value.sign = value_>sign;
  temp_modulus.sign = modulus_>sign;

  j.array_[0] = 0;      j.sign = 1;
  j_last.array_[0] = 0; j_last.sign = 1;
  k.array_[0] = 0;      k.sign = 1;
  k_last.array_[0] = 0; k_last.sign = 1;
  one.array_[0] = 1;    one.array_[1] = 1;
  one.sign = 1;

  for (i = 0; i < NUMBER_LENGTH; i++) {
    temp_value.array_[i] = value_>array_[i];
    temp_modulus.array_[i] = modulus_>array_[i];
    answer.array_[i] = value_>array_[i];
    answer_>array_[i] = 0;
    zero.array_[i] = 0;
  }

  answer.sign = value_>sign;

  if ((temp_value.array_[0] > 0) && (temp_modulus.array_[0] > 0)) {
    while (compare(&temp_value, &temp_modulus) >= 0) {
      sub_array(&temp_value, &temp_modulus, &answer);
      for (i = 0; i <= answer.array_[0]; i++)
        temp_value.array_[i] = answer.array_[i];
    }
  }
}

```

```

    temp_value.sign = answer.sign;
    add_array(&one, &j_last, &j);
    for (i = 0; i <= j.array_[0]; i++)
        j_last.array_[i] = j.array_[i];
    j_last.sign = j.sign;
} /* end while */

while (compare(&temp_value, &zero) == -1) {
    add_array(&temp_value, &temp_modulus, &answer);
    for (i = 0; i <= answer.array_[0]; i++)
        temp_value.array_[i] = answer.array_[i];

    temp_value.sign = answer.sign;
    add_array(&one, &k_last, &k);
    for (i = 0; i <= k.array_[0]; i++)
        k_last.array_[i] = k.array_[i];
    k_last.sign = k.sign;
} /* end while */
} else
    return(-1);

for (i = 0; i < NUMBER_LENGTH; i++)
    answer->array_[i] = answer.array_[i];

answer->sign = answer.sign;
if ((j.array_[0] > 0) || (k.array_[0] > 0))
    sub_array(&j, &k, return_value);
return_value->sign = 1;
} /* end mod_array */

```

```

#include "hilbert.h"

multiply_standard_residues(a_1,a_2,base,answer)
/*
  This routine multiplies two standard residue representations of
  rational numbers. The multiplication is performed in componentwise
  fashion, each result is reduced with respect to the corresponding
  modulus.
*/
    BASE_VECTOR
    *base;

    ARRAY_
    *a_1,
    *a_2,
    *answer;
    {
    int
    i, temp;

    answer->sign = 1;

    for (i = 1; i <= base->num_moduli; i++) {
        if (a_1->array_[i] == -2 || a_2->array_[i] == -2) {
            answer->array_[i] = -2;
            continue;
        }

        if ((a_1->array_[i] == -1 && a_2->array_[i] == 0) ||
            (a_1->array_[i] == 0 && a_2->array_[i] == -1)) {
            answer->array_[i] = -2;
            continue;
        }

        if (a_1->array_[i] == -1 || a_2->array_[i] == -1) {
            answer->array_[i] = -1;
            continue;
        }

        temp = a_1->array_[i] * a_2->array_[i];
        answer->array_[i] = reduce_integer(temp, base->decimal_[i-1]);
    }

    answer->array_[0] = base->num_moduli;
    }

```

```

#include "hilbert.h"
print_(vector)
/*
*/
    This routine prints arrays of values.

    ARRAY_
    {
        *vector;
    }
    int
        i,
        j;

    printf("\n");

    if (vector->sign == 1)
        printf("+");
    else
        printf("-");

    for (i = vector->array_[0]; i > 0; i--) {
        printf(" %d ", vector->array_[i]);
        j++;
        if (j == 10) {
            j = 0;
            printf("\n ");
        }
    }
    /* for
    /* print
    */
    */
    */
}

```

```

#include "hilbert.h"

print_base_vector(base)
/*
  This routine prints the base vector.
*/
BASE_VECTOR
*base;

{
    int i, j, k;
    printf("\n%d moduli compose the base vector.\n", base->num_moduli);
    for (i = 0; i < base->num_moduli; i++) {
        k = 0;
        printf("\n");
        if (base->value_[i].sign == 1)
            printf(" + ");
        else
            printf(" - ");

        for (j = base->value_[i].array_[0]; j > 0; j--) {
            printf(" %d", base->value_[i].array_[j]);
            k++;
            if (k == 10) {
                printf("\n  ");
                k = 0;
            } else
                printf(" ");
        }
        printf("\nThe decimal value of the preceeding modulus = %d\n",
            base->decimal_[i]);
    }
}

```

```

#include "hilbert.h"

print_fraction(rational)
/*
    This program prints fractions.
    FRACTION
    *rational;
    {
        int i, j;
        j = 0;
        printf("\nNumerator: \n");
        if (rational->sign[0] == 1)
            printf("+");
        else
            printf("-");
        for (i = rational->array_0[0]; i > 0; i--) {
            printf("%d", rational->array_0[i]);
            j++;
            if (j == 25){
                j = 0;
                printf("\n ");
            }
        }
        j = 0;
        printf("\nDenominator:\n");
        if (rational->sign[1] == 1)
            printf("+");
        else
            printf("-");
        for (i = rational->array_1[0]; i > 0; i--) {
            printf("%d", rational->array_1[i]);
            j++;
            if (j == 25){
                j = 0;
                printf("\n ");
            }
        }
    }
    /* end program */

```

```

#include "hilbert.h"
q_value (sum, base, q_value)
/*
    This routine recovers the value from the standard residue digits.

    BASE_VECTOR
    *base;

    ARRAY_
    *q_value,
    *sum;

    {
    int i, k, place;

    BASE_VECTOR
    c_j, base_j, s_d_set;

    ARRAY_
    q,
    answer;

    place = 0;

    for (i = 0; i < base->num_moduli; i++)
        if ((sum->array_[i + 1] != 0) && (sum->array_[i + 1] != -1)) {
            digit_array(sum, (i + 1), &answer);

            for (k = 0; k <= answer.array_[0]; k++)
                c_j.value_[place].array_[k] = answer.array_[k];

            c_j.value_[place].sign = 1;

            for (k = 0; k <= base->value_[i].array_[0]; k++)
                base_j.value_[place].array_[k] = base->value_[i].array_[k];

            place++;
        }
        /* end if */

    base_j.num_moduli = place;

    if (place > 0)
    {
        convert(&c_j, &base_j, &s_d_set);

        sum_array(&s_d_set, &base_j, &q);

        for (i = 0; i <= q.array_[0]; i++)
            q_value->array_[i] = q.array_[i];

        q_value->sign = q.sign;
    } else {
        q_value->array_[0] = 1;
        q_value->array_[1] = 1;
        q_value->sign = 1;
    }

    /* end program */

```

```

#include <stdio.h>
reduce_integer(value, modulus)
/*
    This routine finds the least non-negative residue of 'value'
    w.r.t. 'modulus'.
*/
int
    value, modulus;
{
    int quotient, remainder;

    if (value == 0)
        return(0);
    while (value > 0)
        value = value - modulus;
    while (value < 0)
        value = value + modulus;
    return(value);
} /* end reduce_integer */

```

```

#include "hilbert.h"

set_up_digit_array(value, answer)
/*
    Set up the answer array (positionally) reflecting the integer
    value passed.
*/
    ARRAY_
        *answer;

    int
        value;

    {
    int
        temp_value,
        i, posn;

    posn = 0;

    if (value > 0)
        answer->sign = 1;
    else {
        answer->sign = -1;
        value = value * -1;
    }

    while (value > 0) {
        temp_value = (value/10) * 10;
        for ( i = 0; i < 10; i++)
            if (value == (temp_value + i))
                break;

        value = temp_value/10;
        answer->array_[++posn] = i;
    }
    answer->array_[0] = posn;
    }
    /* end set_up_digit_array */

```

```

#include "hilbert.h"

sub_array(a_1, a_2, answer)
/*
  This routine subtracts two arrays positionally; the result is in answer.
  */

  ARRAY_
    *a_1,
    *a_2,
    *answer;
  {
    ARRAY_
      al,
      a2;

    int
      i, j, cary, sign;

    if (a_1->array_[0] == 0)
      a_1->sign = 1;
    if (a_2->array_[0] == 0)
      a_2->sign = 1;

    al.sign = 1;
    a2.sign = 1;
    for (i = 0; i < NUMBER_LENGTH; i++) {
      al.array_[i] = a_1->array_[i];
      a2.array_[i] = a_2->array_[i];
    }

    if ((sign = compare(&al, &a2)) < 0) {
      for (i = 0; i < NUMBER_LENGTH; i++) {
        al.array_[i] = a_2->array_[i];
        a2.array_[i] = a_1->array_[i];
      }
      al.sign = a_2->sign;
      a2.sign = a_1->sign;
    } else {
      al.sign = a_1->sign;
      a2.sign = a_2->sign;
    }

    answer->sign = al.sign * sign;

    /* subtract like sign numbers */

    if (al.sign == a2.sign) {
      answer->array_[0] = al.array_[0];
      for (i = 1; i <= al.array_[0]; i++)
        if (a2.array_[0] < i)
          answer->array_[i] = al.array_[i];
        else

```

```

        answer->array_[i] = a1.array_[i] - a2.array_[i];
/* reduce */
    for (i = 1; i < answer->array_[0]; i++) {
        if (answer->array_[i] < 0) {
            answer->array_[i] = answer->array_[i] + 10;
            answer->array_[i + 1] = answer->array_[i + 1] - 1;
        }
    }
    for (i = answer->array_[0]; i > 0; i--)
        if (answer->array_[i] != 0)
            break;
    answer->array_[0] = 1;

    if (answer->array_[0] == 0)
        answer->sign = 1;
    return(1);
} /* end subtract like signs */

/* Subtract arrays with like signs */
if (a1.sign != a2.sign) {
    answer->array_[0] = a1.array_[0];
    for (i = 1; i <= a1.array_[0]; i++)
        if (a2.array_[0] < i)
            answer->array_[i] = a1.array_[i];
        else
            answer->array_[i] = a1.array_[i] + a2.array_[i];
/* reduce */
    for (i = 1; i < answer->array_[0]; i++) {
        if (answer->array_[i] >= 10) {
            cary = answer->array_[i]/10;
            answer->array_[i] = answer->array_[i] - (10 * cary);
            if (i < answer->array_[0])
                answer->array_[i + 1] = answer->array_[i + 1] + cary;
            else {
                answer->array_[0] = answer->array_[0] + 1;
                answer->array_[i] = cary;
            }
        }
        cary = 0;
    }
} /* reduce - unlike signs */
/* end subtract unlike signs */
/* end if */
for (i = answer->array_[0]; i > 0; i--)
    if (answer->array_[i] != 0)
        break;

answer->array_[0] = 1;
if (answer->array_[0] == 0)
    answer->sign = 1;
} /* end program

```

```

#include "hilbert.h"

sum_array(d_set, base, answer)
/*
  This routine computes the value of the digit set w.r.t. the base vector.
*/

    ARRAY_
        *answer;

    BASE_VECTOR
        *base,
        *d_set;

    {
        int i, posn;

        ARRAY_
            s_value, multiplicand, d_value, digit_set,
            temp_modulus, temp_answer;

        for (i = 0; i <= d_set->value_[0].array_[0]; i++)
            s_value.array_[i] = d_set->value_[0].array_[i];

        multiplicand.array_[0] = 1;
        multiplicand.array_[1] = 1;
        multiplicand.sign = 1;
        temp_modulus.sign = 1;
        digit_set.sign = 1;
        s_value.sign = 1;

        for (posn = 1; posn < base->num_moduli; posn++) {
            for (i = 0; i <= base->value_[posn - 1].array_[0]; i++)
                temp_modulus.array_[i] = base->value_[posn - 1].array_[i];
            temp_modulus.sign = 1;

            multiply_array(&multiplicand, &temp_modulus, &temp_answer);

            for (i = 0; i <= temp_answer.array_[0]; i++)
                multiplicand.array_[i] = temp_answer.array_[i];
            multiplicand.sign = 1;

            for (i = 0; i <= d_set->value_[posn].array_[0]; i++)
                d_value.array_[i] = d_set->value_[posn].array_[i];
            d_value.sign = d_set->value_[posn].sign;

            multiply_array(&multiplicand, &d_value, &temp_answer);
            add_array(&s_value, &temp_answer, &d_value);

            for (i = 0; i <= d_value.array_[0]; i++)
                s_value.array_[i] = d_value.array_[i];
            s_value.sign = d_value.sign;
        }

        answer->sign = s_value.sign;
        for (i = 0; i <= s_value.array_[0]; i++)
            answer->array_[i] = s_value.array_[i];
    }

```

VITA

Deborah Susan Crowell was born on November 19, 1958. She lived in Asheville, North Carolina for her first ten years before moving to Lake Tox-away, North Carolina. After graduating from Rabun-Gap Nacoochee High School, Rabun Gap, Georgia in 1977, she attended undergraduate school at the University of North Carolina, Asheville and received her Bachelor of Arts degree in Psychology. She attended the University of Tennessee, Knoxville and finalized her Master of Science Degree majoring in Computer Science with this thesis in June 1986.