TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

EXACT MATRIX COMPUTATION BY MULTIPLE P-ADIC ARITHMETIC

by
Xin Kai Li

A Dissertation
Presented to the Faculty of the Graduate School
of Towson University
in Partial Fulfillment of the Requirements for the Degree of
DOCTOR OF SCIENCE
Department of Computer and Information Sciences

TOWSON UNIVERSITY
Towson, Maryland, 21252
December 2015

DISSERTATION APPROVAL PAGE

This is to certify that the dissertation prepared by **Xinkai Li**, entitled **"Exact Matrix Computation by Multiple P-adic Arithmetic"**, has been approved by this committee as satisfactorily completing the dissertation requirements for the degree of **Doctor of Science** in Information Technology.

Dr. Chao Lu Chair, Dissertation Committee	12/1/2015 Date
Dr. Matthias Zimand Member, Dissertation Committee	12/1/2015 Date
Dr. Alex Wijesinha Member, Dissertation Committee	12/1/15 Date
Dr. Ramesh Karne Member, Dissertation Committee	12/1/2015 Date
Dr. Chao Lu Chair, Department of COSC	12/1/2015 Date
Dr. Ramesh Karne Doctoral Program Director	12-1-2015 Date
Dr. Janet DeLany Dean of Graduate Studies	12-3-15 Date

Dedicated

To

My Family

Table of Contents

Page

Abstract

Exact Matrix Computation by Multiple P-adic Arithmetic

Xinkai Li

Most of the algorithms are assumed to use exact computation. But in practice, the machine floating point arithmetic is implemented on these algorithms which causes many problems. The existing method is to use a link list representing arbitrary size of integer or decimal numbers, which is extremely time consuming for a larger size matrix calculation.

This dissertation research is to focus on finding an effective way to do exact large matrix calculation. We built a finite *P*-adic number system and found a method to detect overflow. Based on this method and Dixon – Krishnamurthy's theory we established Dixon – Krishnamurthy algorithm to implement finite *P*-adic number system on linear and nonlinear matrix calculation. Dixon – Krishnamurthy algorithm transfers the classic symbolic calculation into integer calculation, significantly improving the calculation efficiency. Furthermore, based on the multiple modulus rational system and finite *P*-adic number system, we constructed a Multiple *P*-adic Data Type. The multiple *P*-adic Data Type can easily transform the finite *P*-adic calculation process into parallel calculation process without modification on math algorithms. With enough independent CPU resources, the calculation time is significantly decreased.

We developed a computational library based on Multiple *P*-adic Data Type and the object oriented program using C/C++. Computational algorithms have been developed using the

data type for the calculation of matrix inverse, Lower Hessenberg form transformation, Wilkinson form transformation, Frobenius form transformation, post processing from all the transformations, reflexive general matrix inverse, Moore-Penrose inverse, calculation of Laplace's method for FTA (Fundamental Theorem of Algebra), Bézoutian formulation of the Resultant and etc. Furthermore, based on the properties of the Multiple P-adic Data Structure, we have developed an efficient proactive self – defense algorithm, which can detect and recover compromised computational data.

Acknowledgement

I would like to give my sincere thanks to all of the people who have helped me during my graduate studies at Towson University.

First and foremost, I would like to thank my advisor, Dr. Chao Lu, for his guidance in my research. He is a wonderful mentor and has always been supportive, inspiring and patient with me. I am also thankful for his encouragement during my difficult times. Without him, completing my degree would have been impossible.

I would also like to thank Dr. Jon Sjogren for his support on the research project, without him our research would be impossible. His insightful and inspiring comments through emails and during meetings proved helpful.

I would also like to thank Jun Tao, Mu Zhao and Yanfeng Zhu for their generous help during my research. I learned a lot from them in both my academic studies and life.

I would also thank Dr. Wei Yu and Dr. Ge Han for their support and helpful discussions. Along with Hanlin Zhang, Linqiang Ge, Guobin Xu, Zhijiang Chen, and Ying Zheng for their friendship and helpful discussion and comments. Many thanks also to all other members of the Department of Computer Science for their friendship.

My appreciation also goes to my dissertation committee members: Dr. Ramesh Karne, Dr. Alex Wijesinha, and Dr. Marius Zimand for their time and suggestions to improve my dissertation.

Last but not least, I would like to thank my family, Xin Li, Jonathan, Peter, Yi Song, Fangzhou Li, my parents and my in-laws for their love, support, and understanding. My love to them is beyond words, time, and distance.

Xinkai Li

Towson Maryland

November, 2015

List of Tables

Page

List of Figures

Page

Chapter 1

Introduction

Most of the algorithms are assumed to use exact computation. But in practice, the machine floating-point arithmetic is implemented on these algorithms causing many problems, usually called “robustness issues” [1]. For example, when applying Gaussian elimination, equal or not-equal zero will be determined for judging singular or nonsingular situation. Floating point arithmetic can only give a precision range which depends on the number of bits for the designated data format. This will hide some potential problem. When the determined element is beyond the accuracy range, an incorrect determination will be made. The truncation error of the floating point will be accumulated which can cause troubles. See the following example taken from [2] for the calculation of e^A,

$$A = \begin{bmatrix} -182 & 91.5 \\ -244 & 123 \end{bmatrix}.$$

Using double-precision floating point data type, taking the series method,

$$e^A = 1 + \frac{A}{1!} + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots.$$

Only up to 175 iteration terms can be implemented as the following:

$$e^A \cong \begin{bmatrix} 2.35836e+009 & -1.18517e+009 \\ 3.16045e+009 & -1.59221e+009 \end{bmatrix}.$$

If the iteration is larger than 175, the result will be meaningless as,

$$\begin{bmatrix} -1.\#IND & -1.\#IND \\ -1.\#IND & -1.\#IND \end{bmatrix}.$$

Actually, there are other ways to calculate e^A,

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -60 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}^{-1}$$

and

$$e^A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} e^1 & 0 \\ 0 & e^{60} \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}^{-1}$$

The result with iteration 175 or more will be:

$$e^A \cong \begin{bmatrix} -5.4366 & 4.0774 \\ -10.8731 & 8.1548 \end{bmatrix}$$

If we use symbolic number system as we did for the Exact Scientific Computational Library (ESCL), with 175 iteration the rational result will be:

$$e^A \cong \begin{bmatrix} \frac{-499031\ldots}{917917\ldots} & \frac{827387\ldots}{826155\ldots} \\ \frac{-336858\ldots}{309808\ldots} & \frac{442180\ldots}{542230\ldots} \end{bmatrix}$$ (Full size in appendix A)

In decimal number representation will be (the same as the above):

$$e^A \cong \begin{bmatrix} -5.4366 & 4.0774 \\ -10.8731 & 8.1548 \end{bmatrix}$$

Furthermore, some algorithms have zero tolerance for errors, such as the Moore-Penrose Inverse [3], A^+ means the Moore-Penrose Inverse of A

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, A_\varepsilon = \begin{bmatrix} 1 & 1 \\ 1 & 1+\varepsilon \end{bmatrix}$$

$$A^{+} = \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, A_{\varepsilon}^{+} = \begin{bmatrix} 1+\frac{1}{\varepsilon} & -\frac{1}{\varepsilon} \\ -\frac{1}{\varepsilon} & \frac{1}{\varepsilon} \end{bmatrix}$$

In that algorithm $\lim_{\varepsilon \to 0} A_{\varepsilon}^{+} \nrightarrow A^{+}$.

There are a number of ways for doing exact computing, which are fraction number system, finite *P*-adic number system (Hensel code), multiple-modulus rational systems, multiple *P*-adic rational system, and etc.

1.1. Fraction Number System

Fraction number system is to use link list to represent arbitrary size of integers and based on that to build fraction number to realize rational number computation.

The arbitrary integers (symbolic) will be established as,

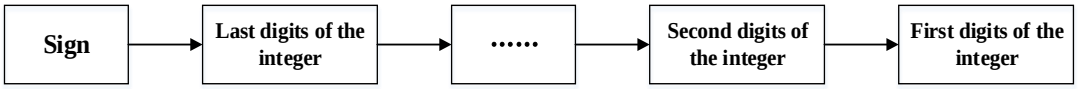


Figure 1.1 Arbitrary Integer Structure

The integer can be as large as possible and the only limitation is the size of the memory. The fraction number can be represented with two arbitrary integers, numerator and denominator. The fraction number system calculation can be implemented as,

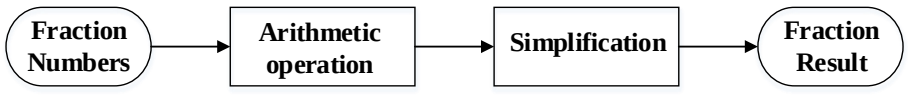


Figure 1.2 Fraction Number Calculation Flow Chart

Simplification is an importation process of the fraction number calculation. After an arithmetic operation, the greatest common divisor (GCD) should be found and divided by numerator and denominator. The complexity of this simplification process can grow very fast. The two importation elements affecting the calculation are the calculation efficiency of the basic arithmetic operation between two link lists and the efficiency of the simplification. The first element is affected by the architecture of the computer CPU. The second element is affected by algorithm convergence effectiveness. Extended Euclidean Algorithm is a good choice to find the GCD [4]. NTL library [5] supplied arbitrary size integers and based on that we established our fraction number system.

The fraction number system is not hard for software development. There will be no overflow problem, if there is enough memory. For small size of calculation, it will have a very good performance. But this data is not effective enough for large calculations. At first, the link list is not an effective data structure, and during the calculation process there are too many functional calls. Also the simplification process cost is too much. When the calculation size is large and the intermediate rational number will be too complex, then the cost on calculation is unacceptable. Meanwhile the size of the intermediate rational number is unpredictable, the memory (heap) overflow problem could possibly occur during the calculation process.

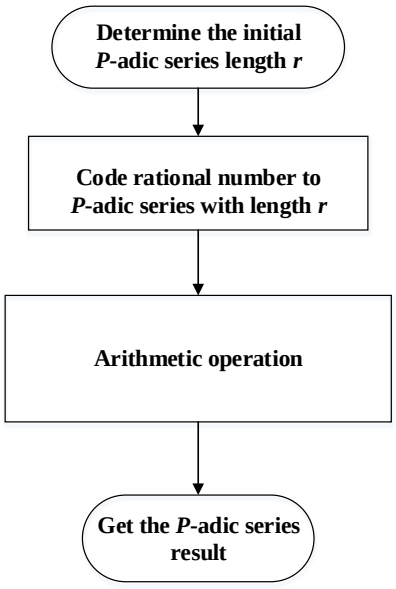
1.2. Finite P-adic Number System [6]

All rational numbers can be uniquely represented as [7],

$$Q(P) = a_{-m}P^{-m} + \cdots + a_{-1}P^{-1} + a_0 + a_1P^1 + \cdots + a_nP^n + \cdots, a_i \in [0, P-1] = \sum_{i=-m}^{\infty} a_iP^i,$$

$$a_i \in [0, P-1],\ P \text{ is a prime number.}$$

Based on this theorem, all the rational numbers can be transformed to a sequence of finite integers. With Krishnamurthy's theory [7, 8], finite length of integer sequence can be used to represent rational numbers and do all the arithmetic operations. Based on this theory, we established the *P*-aidc number system. The original calculation process is as the following:



In this process, according to the input rational numbers and the calculation algorithm, the *P*-adic sequence length *r* is determined. Then rational numbers are transformed to *P*-adic sequences and *P*-adic arithmetic is implemented to get the *P*-adic sequence results. An important issue is how to identify the initial value of *P*-adic sequence *r*. The time complex for *P*-adic arithmetic is $O(r^2)$. The other issue is how to check Hensel code overflow.

Figure 1.3 P-adic Calculation Process

Hensel code of the *P*–adic expansion can realize error free computation for rational numbers [8]. Hensel (1908) introduced the *P*–adic arithmetic [9]. Bachman (1964) gave a computational algorithm for transforming a rational number into *P*–adic expansion [10]. Young and Gregory (1973) suggested a residue arithmetic procedure for error free computing [11]. Krishnamurthy, Rao and Subramanian (1974) defined the finite length *P*-adic arithmetic named Hensel code [8]. Dixon [12] and Miola [13] (1982) both found separately an efficient computational algorithm to reconstruct rational numbers from *P*–adic expansion, based on extended Euclid's algorithm. For this algorithm, Kornerup and Gregory (1983) gave a more complete explanation about Farey fractions theory [14]. Koc (2002) pointed out four research directions on Hensel code. One of them is the detection of overflow and underflow [15]. Hensel code can be used to develop a new word-based computer data structure [16]. But for Hensel code, if the word length is not sufficient for the rational number it represents, the transformed rational number will be meaningless. One of the aspects of this problem is to predict the required word length. Rao (1976) [17] and Dixon (1982) [12] both gave ways to predict the length of the *P*-adic expansion for specific matrix calculation. We have developed a method [6] for Hensel code overflow detection. The method can realize overflow detection by the prime *P* and the Hensel code itself. Using this method, a few digits of the Hensel code will be sacrificed.

Finite *P*-adic number system can be easily implemented on 32 or 64 bits CPU architecture machines. The sequence length is determined before the calculation. Array data structure can be used to build the class of the *P*-adic number system, which is more effective than the link list data structure comparing the fraction number system. During the calculation process, fewer function calls and most of the operations are integer

operations which can decrease the calculation time cost. While for finite *P*-adic number system, if the *P*-adic sequence length is not sufficient, the Hensel code overflow problem possibly occurs and the calculation results will be useless. And for multiplication and division operations of *P*-adic arithmetic, the time complex is $O(n^2)$.

1.3. Multiple-modulus Rational System [8]

Multiple-modulus rational system is to extend residue number system (RNS) to rational number field. The main idea is to represent a rational number $\propto$ with integer set $r\sim\{r_1, r_2, \cdots, r_s\}$, which come from $\propto$ moduli prime base set $\{p_1, p_2, \cdots p_s\}$. The calculation arithmetic is based on RNS arithmetic. Multiple-modulus rational system can be implemented with parallel computation. For each prime p_i, the calculation process is independent.

RNS was introduced by Garner in 1959 [18]. In 1967, Newman gave a way of solving linear equation $Ax = b$ by Chinese Remainder Theorem [19]. This looks as a try to extend RNS to rational number field. Meanwhile, Jo Ann Howell and Robert T. Gregory also made a progress on using RNS to solve linear algebraic equations in 1969 [20, 21]. Rao, Subramanian and Krishnamurthy introduced the way to solve Moore-Penrose inverse by residue arithmetic [17].

For each prime channel, the calculation is independent from the others. Parallel computation will be easily implemented on the computation process. The time cost will be significantly decreasing when operating on a huge calculation. Overflow problem (as Hensel code overflow problem) is still an issue. However, when the overflow problem occurs, the results can be kept at the intermediate stages. These results will be reused for

generating the final results. When using the multiple-modulus rational system, a lot of prime numbers will be chosen on the base set. But if the intermediate rational number's denominator is the dividend of any prime in the prime set, the channel with that prime will be terminated. And it is hard to compare with zero or one.

1.4. Multiple P-adic Rational System [22]

Multiple *P*-adic rational system is the combination of finite *P*-adic arithmetic (Hensel code) and multiple-modulus rational system. A rational number $\propto$ will be represented by the following structure,

$$\propto: \underbrace{\boxed{Integer\ a_{00}}\boxed{Integer\ a_{01}}\cdots\boxed{Integer\ a_{0k}}}_{module\ results\ of\ p_0}\cdots\underbrace{\boxed{Integer\ a_{n0}}\boxed{Integer\ a_{n1}}\cdots\boxed{Integer\ a_{nk}}}_{module\ results\ of\ p_n}$$

$p \sim \{p_1, p_2, \cdots p_n\}$ is the prime base set.

$a_i \sim \{a_{i1}, a_{i2}, \cdots, a_{ik}\}$ is the finite p_i-adic sequence.

Figure 1.4 Multiple P-adic Data Structure

Each finite p_i-adic sequence is an independent calculation process with *P*-adic arithmetic.

John F. Morrison introduced the concept of multiple *P*-adic algorithm and named it with parallel *P*-adic computation in 1988 [23]. In 1993, Carla Limongelli and Hans Wolfgang Loidl gave the arithmetic on parallel *P*-adic algorithm [24]. Meanwhile, Koc had published a paper about parallel *P*-adic algorithm for linear system [25]. According to the multiple *P*-adic rational system, we developed a multiple *P*-adic data structure which can realize parallel rational calculation.

There are many other algorithms that can realize rational computation, such as slash number system, which was established from the finite continue fraction expansion of fraction number. My research is to focus on *P*-adic direction.

Multiple *P*-adic rational system have the advantages of *P*-adic number system and RNS. It can realize parallel computation at each prime channel and can compare with zero or one.

1.5. Overview of this Dissertation Research

The goal of this dissertation is to find an effective way to do exact large matrix calculation. During the research process, we first built a fraction number system which cost too much time for large matrix calculation. And then we try to establish a periodic P-adic number system, while the length of the period of multiplication of two period P-adic sequence is too large to be acceptable during the practical implementation. Then we built a finite P-adic number system and found a method to detect overflow situations. Based on this method and Dixon-Krishnamurthy's theory we established Dixon-Krishnamurthy algorithm to implement finite P-adic number system on linear and nonlinear matrix calculation. However, the complexity of P-adic number system is $O(l^2)$ with the P-adic sequence length l. During the practical implementation, even a small size of matrix operation need a long length of P-adic sequence. The computation cost is not as effective as we expected. So we constructed multiple P-adic rational system based on the multiple-modulus rational system and finite P-adic number system. During this process, we found the extended Chinese Remainder Theorem and gave an effective way to do overflow detection. The multiple P-adic rational system can easily transform finite P-adic calculation process into parallel calculation process without modification on math

algorithm. With enough independent CPU resources, the calculation time cost is significantly decreased. Furthermore, based on the property of the Multiple P-adic Data Structure, we have developed an efficient proactive self – defense algorithm.

1.5.1. Periodicity of the P-adic Expansion

It is known that a real number is rational if and only if its decimal expansion is periodic. Similarly, a P-adic number is rational if and only if its P-adic expansion is periodic. So we represent a rational number $\propto$ with the form

$$\alpha =. A_0 \cdots A_s \overline{a_0 \cdots a_{n-1}}$$

$\overline{a_0 \cdots a_{n-1}}$ is the periodic part.

If the periods of two entry P-adic sequences are *m* and *n*, for addition/subtraction operation, the maximum length of periodic part is $LCM(n, m)$; for multiplication operation, the maximum length of periodic part is $LCM(m, n) \times (P^{GCD(m,n)} - 1)$. The mathematical proof process details will be displayed in Chapter 2 section2.3.

1.5.2. Hensel Code Overflow Detection

Hensel code can be used to develop a new word-based computer data structure [16]. But for Hensel code, if the word length is not sufficient for the rational number it represents, the transformed rational number will be meaningless. One of the aspects of this problem is to predict the required word length. Rao (1976) [17] and Dixon (1982) [12] both gave ways to predict the length of the *P*-adic expansion for specific matrix calculation. But for some cases, it is hard to predict properly when the matrix calculation process is overly complex. Hensel code is one kind of code, in which a finite *P*–adic expansion is mapped

to the Farey Rationals [14, 31]. A Hensel code expansion can be identified as to whether it is overflow, just by the prime *p* and the Hensel code itself. We have developed an easy method for detecting the Hensel code overflow, resulting in some Hensel code digits being sacrificed.

1.5.3. Multiple P-adic Data Type

During the past few years, we have been working on *P*-adic theory and its implementation. Based on the Chinese Remainder Theorem and Hensel code, a new data type has been established to realize a rational calculation called Multiple *P*-adic Data Type [12, 16, 17]. With this data type, all rational number operations are converted to integer calculations, and the fast integer multiplication of modern computer architectures can be fully used. This data type can be significantly effective in the parallel and cloud computing environment due to its independent computation at each node during the calculation process. Furthermore, the existing C++ programs can be converted to run with this data type with no (or minimal) changes to the source code. We have developed a computational library based on the data type and the object oriented program using C/C++. Computational algorithms have been developed using the data type for the calculation of matrix inverse, Lower Hessenberg form transformation, Wilkinson form transformation, Frobenius form transformation, post processing from all the transformations, reflexive general matrix inverse, Moore-Penrose inverse, e^{At}clculation, Laplace's method for FTA (Fundamental Theorem of Algebra), Bézoutian formulation of the Resultant, and etc.

1.5.4. Overflow Detection for Multiple P-adic Data Type

P-adic Arithmetic (Hensel code) can be used for rational number computation to avoid rounding (truncation) errors. But during the implementation process, it usually needs an enormous length of the *P*-adic sequence digits to maintain the accuracy of the results. That will obviously decrease the efficiency of the calculation. Multiple *P*-adic data type [22] can realize parallel calculation among multiple CPU cores and each calculation process is independent. This can significantly decrease the calculation time, when there are sufficient CPU cores. The Multiple *P*-adic data type is based on the Chinese remainder theorem and Hensel code. In 1981, John F. Morrison [23] introduced the concept of Multiple *P*-adic algorithm, which is based on the multiple-modulus arithmetic proposed by D. Matula and C. Gregory [28]. In 1993, Carla Limongelli and Hans Wofgang Loidl [4] gave the description of arithmetic for multiple *P*-adic sequences. Koc [25] had published a series of papers about Parallel *P*-adic algorithm for linear systems. The Multiple *P*-adic data type has the same overflow problem with Hensel code. If the total length of the Multiple *P*-adic sequence (a more accurate description is the product of p_i^r, p_i means the prime base and r means the p_i-adic sequence length) is not sufficient enough, the converted rational number will be meaningless. This situation is called overflow problem. We have developed an overflow detection method to check whether the overflow problem happened during the single *P*-adic (Hensel code) sequence calculation process [6]. The method can identify whether the overflow situation happened just by the prime *P* and the sequence digits themselves. But with this method some digits should be sacrificed as identification parts. This method can still be used on Multiple *P*-adic overflow detection. However according to the property of the Multiple *P*-adic data

type, we have improved the previous method. And with this improved method we will sacrifice fewer digits as verification parts and meanwhile greatly decrease the error rates for detecting results.

1.5.5. Proactive Self – defense Algorithm

To protect the distributed high performance computing (HPC) systems from attacks, we need to consider two defense levels: data and system. At the data level, we develop pre-defined data self-correction methods to detect and recover compromised computational data. At the system level, we implement monitoring and detection tools to discover exploitable vulnerabilities proactively and to make the HPC systems robust against cyber-attacks. An efficient proactive self-defense algorithm was developed based on multiple *P*-adic data structure, in which ADU (Application Data Unit) [39] attacks occurred, and how to enable the computation process to detect data being compromised and still deliver the correct results. The operation process implemented with multiple *P*-adic data type can be separated into several parallel sub-processes. Each sub-process can be allocated in different nodes of a HPC system and each sub-process is operated independently by computing nodes. If some sub-processes have been compromised and the data is distorted, algorithms [35] can be leveraged to identify the abnormality. Further, if the number of sub-processes with errors does not pass a given threshold, the sub-processes with errors can be identified and the correct results can still be obtained [36]. If this data type is implemented on huge integer or rational number computing, the operation time cost will be efficient. Both linear and non-linear calculation processes can be applied with multiple *P*-adic data type instances. The algorithm for data self-correction comes from the redundant residue number system (RRNS) [34]. Using the

multiple *P*-adic data type, we define a prime set$p\sim\{p_1, p_2, \cdots, p_k\}$. With k primes, we can make sure to avoid the overflow situation [38]. Nonetheless, during the implementation, we set $p\sim\{p_1, p_2, \cdots, p_k, p_{k+1}, \cdots p_n\}$, where the $\{p_{k+1}, \cdots p_n\}$ part is the redundant portion. According to Mandelbaum's theory [36, 40, 41], if $\frac{n-k}{2}$ or less sub-processes are changed/compromised due to attacks, we can identify the compromised sub-processes and still get the correct results. The main idea is to compare the decoded values from all the combinations of C_n^k among the nodes. The details will be given in section 5.3 as well as examples to explain how the algorithm works.

Chapter 2

P-adic Arithmetic

2.1. Overview

Usually we are familiar with the 10-base number system, but in our everyday life, we also use 60- base (seconds & hours) and 12- base (dozen of eggs) number systems. For computer science professionals, we get used to the binary number system. Numbers can be represented by different formats. P-adic is to represent rational number by the prime base system with integer sequence. For example, if we choose 3 as the prime base *P*,

$$7 = P^0 + 2P^1 + 0P^2 + 0P^3 + \cdots$$

$$75 = 0P^0 + P^1 + 2P^2 + 2P^3 + 0P^4 \cdots$$

It is not hard to transform a positive integer into P-adic sequence, but how to represent a negative number and how to represent a fraction number? The algorithm has been given by Bachman [10] as follows,

For $\alpha = \frac{a}{b}P^n, a, b, n \in \mathbb{Z}, b \neq 0, GCD(a,b), GCD(a,P), GCD(b,P) = 1, i = 0$
Step 1. $\alpha \bmod P = a_i$
Step 2. $\alpha = (\alpha - a_i)/P$, $i = i + 1$, go to Step 1 to get a_i
Continue Step 1 and Step 2, to get P-adic sequence
Finally, $\alpha = P^n \cdot \sum_{i=0}^{\infty} a_i P^i = \sum_{i=n}^{\infty} a_{i-n} P^i$

Figure 2.1 Bachman Algorithm

Tips: $a/b \bmod P, b \neq 1$ is calculated by this: find $c \cdot b \bmod P \equiv 1$, the answer equals to $c \cdot a \bmod P$.

The *P*-adic sequence for $\frac{a}{b}P^n$ will have the form as the following [15]:

$$\begin{aligned} a_n a_{n+1} \cdots a_{-2} a_{-1} . a_0 a_1 a_2 \cdots \quad & for\ n < 0 \\ . a_0 a_1 a_2 \cdots \quad & for\ n = 0 \\ .000 a_0 a_1 a_2 \cdots \quad & for\ n > 0 \end{aligned}$$

Conventionally, we write *P*-adic sequence as,

$a_{i-n} a_{i-n+1} \ldots$ point = n
point means the position of a_0.

Here is the process for conversing $\frac{1}{5}$ to 3-adic sequences as given in Table 2.1,

	Input fraction	**Module**	**Fraction For next Loop**
Loop 1	1/5	1/5 mod 3 = 2	(1/5 – 2) / 3 = -3/5
Loop 2	-3/5	-3/5 mod 3 = 0	(-3/5 – 0) / 3 = -1/5
Loop 3	-1/5	-1/5 mod 3 = 1	(-1/5 – 1) / 3 = -2/5
Loop 4	-2/5	-2/5 mod 3 = 2	(-2/5 – 2) / 3 = -4/5
Loop 5	-4/5	-4/5 mod 3 = 1	(-4/5 – 1) / 3 = -3/5
……			

Table 2.1: P-adic Sequence Coding Process

All rational numbers can be represented as,

$$Q(P) = a_{-m}P^{-m} + \cdots + a_{-1}P^{-1} + a_0 + a_1P^1 + \cdots + a_nP^n + \cdots,\ a_i \in [0, P-1]$$

Can be written as,

$$\alpha = \sum_{i=-m}^{\infty} a_i P^i$$

$a_i \in [0, P-1];\ m, n \in \mathbb{z};\ P$ is a prime number.

Then note $Q(P) = a_{-m} \cdots a_0 \cdots$ point = $-m$

2.2. P-adic Arithmetic [7] [15]

The P-adic arithmetic is quite similar to the decimal arithmetic. They both need to carry digits from low to high. In the decimal number system, such as 365, the lower digits are written from right to left. While for P-adic number system, such as $\frac{1}{6} = .1404040$ ... where $P = 5$, the lower digits are written from left to right. The calculation process will have some difference.

2.2.1. Addition/Subtraction

The addition of *P*-adic numbers is similar to the binary numeral addition. The difference is modulo *P*.

The addition process is calculating from left to right. Here is an example of computing $1/6 + 1/2 = 2/3$ for P = 5:

$$\frac{1}{6} = .140404040\cdots \qquad \frac{1}{2} = .32222222...$$

In the addition operation process, the position of the point should be kept in alignment.

$$\begin{array}{l} .140404040\cdots \\ \underline{.322222222\cdots} \\ .413131313\cdots \end{array}$$

We can check that

$$\frac{2}{3} = .413131313...$$

Subtraction is also an addition process. First we use recursive ways to get the opposite sequences subtracted then do an addition:

$$\alpha - \beta = \alpha + (-\beta).$$

2.2.2. Multiplication/ Division

The multiplication of P-adic numbers is also similar to binary multiplication. The difference is also that *P*-adic multiplication is calculating from left to right.

The point of the multiplication result equals to point1 + point2. (point1 and point2 means the value of point for multiplicand/dividend P-adic sequence and multiplicator/divisor P-adic sequence)

Here is an example of $\frac{2}{3} \times \frac{1}{6} = \frac{1}{9}$, where P = 5:

$$\frac{2}{3} = .413131313\cdots \qquad \frac{1}{6} = .140404040\cdots$$

The multiplication operations can be showed following,

$$
\begin{array}{r}
.4131313131313\cdots \\
\times\ .1404040404040\cdots \\
\hline
4131313131313\cdots \\
123131313131\cdots \\
00000000000\cdots \\
1231313131\cdots \\
000000000\cdots \\
12313131\cdots \\
0000000\cdots \\
123131\cdots \\
00000\cdots \\
1231\cdots \\
000\cdots \\
12\cdots \\
+\qquad 0\cdots \\
\hline
.4201243201243\cdots
\end{array}
$$

(This example came from Koc[15].)

Check the P-adic expansion of 1/9, they are the same,

$$\frac{1}{9} = .4201243201243.....$$

Division can also be done by multiplication process. First we use recursive method to get the inverse of dividend then do a multiplication.

The point for division equals to point1 - point2.

P-adic sequence is infinite. It is not possible to use this theory directly in computers. Computer architecture is a finite system. The way of using P-adic arithmetic is to find a way of using finite P-adic sequence to represent fraction numbers. There are two ways for doing that: periodicity of the P-adic sequence and finite P-adic arithmetic (Hensel code).

Chapter 3

Implementation with P-adic Arithmetic

3.1 Periodicity of the P-adic Expansion

It is known that a real number is rational if and only if its decimal expansion is periodic. Similarly, a *P-adic* number is rational if and only if its *P-adic* expansion is periodic. Consequently, since we are primarily interested in the *P-adic* expansions of rational numbers, we will be dealing only with *P-adic* expansions which are periodic. The expansion eventually repeats to the right. That is, if α is a rational number, then it has a repeating pattern of $a_i P^j$ in its *P-adic* expansion, i.e., it is of the form

$$\alpha = .A_0 \cdots A_s \overline{a_0 \cdots a_{n-1}}$$

with periodic length *n*. It can give the sufficient number of digits for exact computation [26]. Let us observe what happens after the arithmetic operations of two *P-adic* sequences, and discuss the periodicity of a resulted *P-adic* sequence from arithmetic operation in *P-adic* field.

From the Table 2.1, the values on loop 1 and loop 5 of **Fraction for next loop** are both -3/5. This means there will have the period circle. The periodicity of 1/5 on 3-adic is 4: $.2\overline{0121}$.

As described in Koc[15], the series

$$1+p+p^2+p^3+\cdots$$

converges to $\frac{1}{1-p}$ in the *p*-adic norm. Now, as an example, consider the power series expansion

$$\alpha = 2+3p+p^2+3p^3+p^4+3p^5+p^6+\cdots$$

$$= 2+(3p+p^2)(1+p^2+p^4+\cdots)$$

Since $1+p^2+p^4+\cdots$ converges to $\frac{1}{(1-p^2)}$, we have

$$\alpha = 2+\frac{3p+p^2}{1-p^2}.$$

Shan gave the decoding formula in [26] [27].

For a P-adic sequence $.A_1 \ldots A_s\overline{a_1 \ldots a_n}$, the decoding process is,

$$\alpha = A_1 \times P^0 + A_2 \times P^1 + \cdots + A_s \times P^{s-1} + (a_1 \times P^0 + a_2 \times P^1 + \cdots + a_n \times P^{n-1}) \times \frac{P^s}{1-P^n}$$

P is the prime for the *P-adic* expansion.

For example:

$$.2\overline{0121} = 2\times 3^0 + \left(0\times 3^0 + 1\times 3^1 + 2\times 3^2 + 1\times 3^3\right)\times \frac{3}{1-3^4} = \frac{1}{5}.$$

The property of periodicity of P-adic sequence makes it possible to use a finite sequence to represent a fraction number. But there are still issues for implementing the P-adic arithmetic in practice. We need to find an algorithm to determine the period length for the result of the P-adic arithmetic operations. Based on the algorithm, we can choose how long the entry P-adic sequence length should be for the calculation process. Through Shan's [26, 27] theory, we get the theorem as the following:

If the periods of two entry P-adic sequences are *m* and *n*, for addition/subtraction operation, the maximum length of periodic part is $LCM(n,m)$; for multiplication operation, the maximum length of periodic part is $LCM(m,n) \times (P^{GCD(m,n)} - 1)$.

The proof process is as the following:

Addition/Subtraction [27]

Assume that we have two *P-adic* sequences ($s < t$)

$$a =. A_0 \cdots A_s \overline{a_0 \cdots a_n}$$

$$b =. B_0 \cdots B_t \overline{b_0 \cdots b_m}$$

Line up these two sequences and set $a_{t-s+1} = c_1$, we get

$$\begin{array}{lll} A_1A_2...A_sa_1...a_{t-s} & c_1... & c_nc_{n+1}...c_{2n}... \\ B_1 \quad\quad ...B_t & b_1... & b_mb_{m+1}...b_{2m}... \end{array}$$

Let's consider the right side of the vertical stroke line. Suppose two integer *x, y* satisfy the condition that $x \neq y$ and $c_x + b_x = c_y + b_y$.

$\because$ c_i and b_j can be any numbers

∴ In the worst case, we must make sure that

$$x = k_1 \times n + i = k_2 \times m + j$$

$$y = k_3 \times n + i = k_4 \times m + j$$

$$\Rightarrow \quad x - y = (k_1 - k_3) \times n = (k_2 - k_4) \times m$$

This means *x* - *y* must be exactly divisible by *n* and *m*, the smallest integer which satisfies this requirement is the least common multiple of *n* and *m*.

Conclusion: In addition/subtraction, the maximum length of periodic part is: $LCM(n, m)$.

Multiplication

$$\begin{aligned} a \times b &= .A_1 \dots A_s \overline{a_1 \dots a_n} \times .B_1 \dots B_t \overline{b_1 \dots b_m} \\ &= .A_1 \dots A_s \overline{a_1 \dots a_n} \times .B_1 \dots B_t + .A_1 \dots A_s \overline{a_1 \dots a_n} \times .\overset{t}{0 \dots 0} \overline{b_1 \dots b_m} \\ &= \underbrace{.A_1 \dots A_s \overline{a_1 \dots a_n} \times .B_1 \dots B_t}_{1} + \underbrace{.A_1 \dots A_s \times .\overset{t}{0 \dots 0} \overline{b_1 \dots b_m}}_{2} + \underbrace{.\overset{s}{0 \dots 0} \overline{a_1 \dots a_n} \times .\overset{t}{0 \dots 0} \overline{b_1 \dots b_m}}_{3} \end{aligned}$$

It's obviously that the result of *Part 1* and *Part 2* has a periodic part of *n* and *m* digits respectively. According to the previous conclusion we get from addition, the resulted *P-adic* sequence for (*Part 1 + Part 2)* has a periodic part of $LCM(n, m)$ digits.

Part 3:

$$\begin{aligned} & .\overset{s}{0 \dots 0} \overline{a_1 \dots a_n} \times .\overset{t}{0 \dots 0} \overline{b_1 \dots b_m} \\ &= (.a_1 \dots a_n) \times (1 + p^n + p^{2n} + \dots + p^{\infty}) \times (.b_1 \dots b_m) \times (1 + p^m + p^{2m} + \dots + p^{\infty}) \times p^{s+t} \\ &= \underbrace{(.a_1 \dots a_n) \times (.b_1 \dots b_m) \times p^{s+t}}_{4} \times \underbrace{(1 + p^n + p^{2n} + \dots + p^{\infty}) \times (1 + p^m + p^{2m} + \dots + p^{\infty})}_{5} \end{aligned}$$

Part 4 is a constant and *Part 5* is the one we need to focus on.

$$(1+\underset{n-1}{p^{n}}+\underset{n-1}{p^{2n}}+\ldots+p^{\infty})\times(1+\underset{m-1}{p^{m}}+\underset{m-1}{p^{2m}}+\ldots+p^{\infty})$$
$$=.10\ldots010\ldots0\ldots\times.10\ldots010\ldots0\ldots$$

To determine the periodic length of multiplication is to find the period of *Part 5*, it also means to find the period of the following result:

For easy understanding, here is a specific example of *m=6, n=4*:

$$\begin{array}{rl} & .\underbrace{10000}_{m}\ldots10000\ldots. \\ \times & .\underbrace{10000}_{n}\ldots10000\ldots. \end{array}$$

$$\underset{n=4}{1000}\,001000001000001000001000001000001000001000$$
$$\underset{n=4}{1000}\,001000001000001000001000001000001000001$$
$$\underset{n=4}{1000}\,001000001000001000001000001000000100$$
$$\underset{n=4}{1000}\,001000001000001000001000001000$$
$$\underset{n=4}{1000}\,001000001000001000001000001$$
$$\underset{n=4}{1000}\,001000001000001000000100$$
$$\underset{n=4}{1000}\,001000001000001000$$
$$\underset{n=4}{1000}\,001000001000001$$
$$\underset{n=4}{1000}\,001000001000$$
$$\underset{n=4}{1000}\,001000001$$
$$\underset{n=4}{1000}\,00100$$
$$\underset{n=4}{1000}\,0$$
$$1$$
$$\ldots$$

We can view the result by the number of blocks as the following:

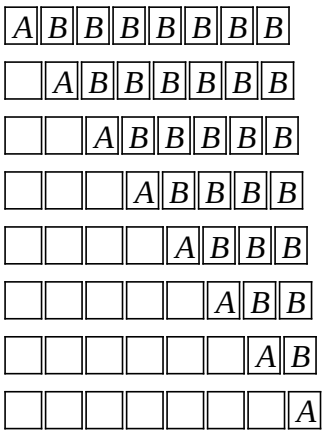
$\overbrace{\quad}^{LCM(6,4)}$	$\overbrace{\quad}^{LCM(6,4)}$			
100000100000 10000010 1000	100000100000 000010000010 001000001000	100000100000 000010000010 001000001000	100000100000 000010000010 001000001000	100000100000 000010000010 001000001000
	100000100000 10000010 1000	100000100000 000010000010 001000001000	100000100000 000010000010 001000001000	100000100000 000010000010 001000001000
		100000100000 10000010 1000	100000100000 000010000010 001000001000	100000100000 000010000010 001000001000
			100000100000 10000010 1000	100000100000 000010000010 001000001000
				100000100000 10000010 1000

From the graph, there are only two kinds of number blocks, which are:

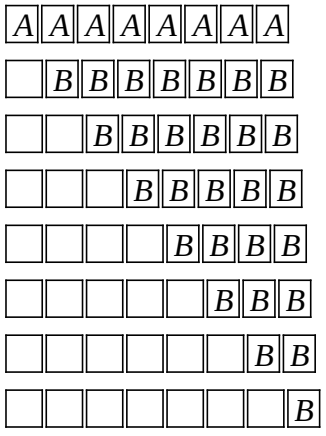
$$\boxed{A} = \overbrace{\begin{matrix}100000100000\\10000010\\1000\end{matrix}}^{LCM(6,4)}$$

$$\boxed{B} = \overbrace{\begin{matrix}100000100000\\000010000010\\001000001000\end{matrix}}^{LCM(6,4)}$$

The heights of $\boxed{A}$ and $\boxed{B}$ are $\frac{LCM(m,n)}{n} = 3$. The result can be shown by $\boxed{A}$ and $\boxed{B}$ as the following:



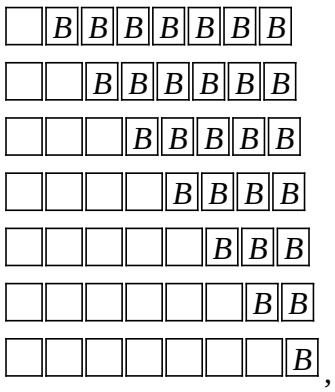
We can move the position of $\boxed{A}$ for better analysis, while not change the final answer as the following:



If we want to define the periodic length of *Part 5*, we can get that from determining the periodic length of two parts:

Part A is $\boxed{A}\boxed{A}\boxed{A}\boxed{A}\boxed{A}\boxed{A}\boxed{A}\boxed{A}$, whose periodic length is the length of $\boxed{A}$, equaling to $LCM(m,n)$.

Part B is



whose periodic length will be $LCM(m,n) \times (P^{GCD(m,n)} - 1)$.

Thus, the total periodic length will be:

$$LCM(m,n) \times (P^{GCD(m,n)} - 1).$$

Proof:

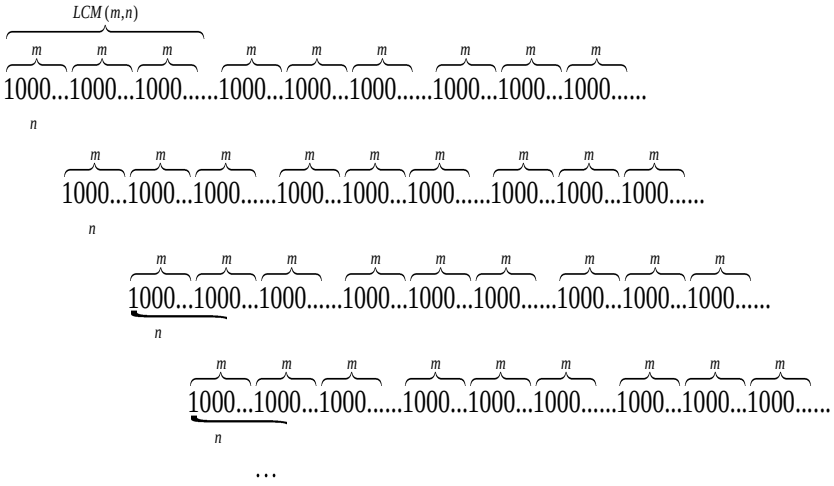
Assume $m \geq n$, $m, n \in N$

For the two *P-adic* sequences:

$$m: \quad \underbrace{10000...}_{\text{m}}10000....$$
$$n: \quad \underbrace{10000...}_{\text{n}}10000....$$

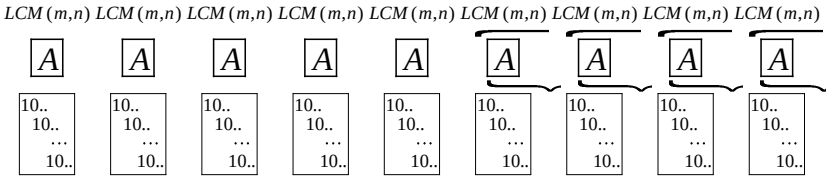
The multiplication is following:

$$\begin{array}{rl} & \underbrace{10000...}_{\text{m}}10000.... \\ \times & \underbrace{10000...}_{\text{n}}10000.... \\ \hline \end{array}$$

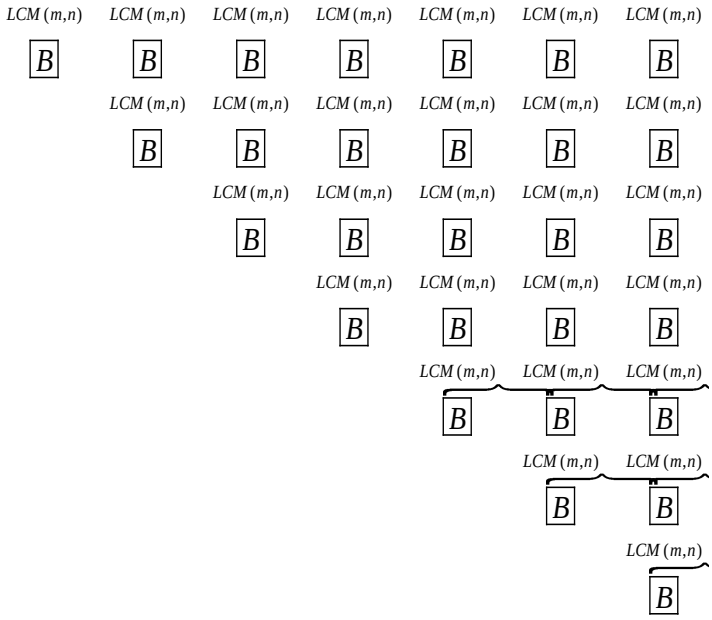


Following the above example, we can transform the result into number blocks,

Part A:



Part B:



$$\boxed{B} = \left.\begin{bmatrix} \overbrace{1..0..1..}^{LCM(m,n)} \\ 0..0..1.. \\ \cdots \\ 0..0..1.. \end{bmatrix}\right\} \frac{LCM(m,n)}{n}$$

For the *Part B*, after the sum, it will become a natural number sequence for the basic elements $\boxed{B}$, as the following:

$$\boxed{B} \quad 2\boxed{B} \quad 3\boxed{B} \quad 4\boxed{B} \quad \cdots$$

Theorem 3.1.1. For a natural number sequence, if it is transformed into a *P-adic* field, and the carry is from left to right, the resulting sequence will be periodic and the periodic length is $P - 1$.

A specific example in *5-adic* field:

Natural number sequence N	S_0	S_1	S_2	S_3
1	*1*			
2	*2*			
3	*3*			
4	*4*			
5		*0*		*5*
6		*2*	*4*	*0*
7		*3*	*4*	*0*
8		*4*	*4*	*0*
9		*0*	*4*	*5*
10		*2*	*8*	*0*
11		*3*	*8*	*0*
12		*4*	*8*	*0*
…		*…*	*…*	*…*
4n+1		*0*	*4(n-1)*	*5*
4n+2		*2*	*4n*	*0*
4n+3		*3*	*4n*	*0*
4n+4		*4*	*4n*	*0*
…		*…*	*…*	*…*

By generalizing this on any *P-adic* field, the table will be the following:

Natural number sequence *N*	S_0	S_1	S_2	S_3
1	*1*			
2	*2*			
3	*…*			
…	*P-1*			
P		*0*		*P*
P+1		*2*	*P-1*	*0*
P+2		*3*	*P-1*	*0*
…		*…*	*…*	*…*
2P-2		*P-1*	*P-1*	*0*
2P-1		*0*	*P-1*	*P*
2P		*2*	*2(P-1)*	*0*
2P+1		*3*	*2(P-1)*	*0*
…		*…*	*…*	*…*
nP-n		*P-1*	*(n-1)(P-1)*	*0*
nP-n+1		*0*	*(n-1)(P-1)*	*P*
nP-n+2		*2*	*n(P-1)*	*0*
nP-n+3		*3*	*n(P-1)*	*0*
…		*…*	*…*	*…*
nP-n+P-1		*P-1*	*n(P-1)*	*0*
…		*…*	*…*	*…*

From the above table, it is shown that the natural number $N = S_0 + S_1 + S_2 + S_3$. S_0 is limited. It is not hard to find that after *P-adic* transform, S_1 will have the period of *P-1*, and $S_2 + S_3$ will become 0 sequences. Thus, the period of the natural sequence will be *P – 1*.

Theorem 3.1.2. For a natural number sequence of $\boxed{x}$, which is also built by number sequences with digit length l and within periodic length r , if it is transformed into a P-adic field, and the carry is from left to right, the resulting sequence will be periodic and the periodic length is $l \times (P^r - 1)$.

Let us show this by an example: Convert the sequence $11112222333344445555666 6$ ... into 3-adic field:

Here: $l = 4, r = 1, P = 3$

$$l \times (P^r - 1) = 4 \times 2 = 8$$

We can verify this by converting the number $11112222333344445555666 6$ ... into 3-adic sequence: $.11112222\overline{01112222}$, which has the periodic length 8.

Another example, covert the sequence 1010202030304040 ... into 2-adic field:

Here: $l = 4, r = 2, P = 2$

$$l \times (P^r - 1) = 4 \times 2 = 12$$

Verification: covert the number 1010202030304040 ... into 2-adic sequence $.\overline{1010010111110}$, which has the periodic length 12.

The proof of Theorem 2 will be similar with the proof of Theorem 1.

From Theorem 2, if the period within $\boxed{B}$ is C_B, we can get the period of *Part B* directly as $LCM(m, n) \times (P^{C_B} - 1)$.

Next step is to find the C_B:

The elements $\boxed{B}$ can be divided into less length structures $\overbrace{\boxed{x_{ij}}}^{GCD(m,n)}$, where $\left(1 \le i \le \frac{LCM(m,n)}{GCD(m,n)}, 1 \le j \le \frac{LCM(m,n)}{n}\right)$, which is the largest number sequence that can be divided into the basic elements of the $\boxed{B}$.

$$\boxed{B} = \boxed{\overbrace{1..0..1..}^{LCM(m,n)} \; \overbrace{0..1..1..}^{LCM(m,n)} \; \dots \; \overbrace{0..0..1..}^{LCM(m,n)}} = \boxed{\begin{matrix} \overbrace{1..}^{GCD(m,n)} & \overbrace{0..}^{GCD(m,n)} & \overbrace{1..}^{GCD(m,n)} \\ \overbrace{0..}^{GCD(m,n)} & \overbrace{1..}^{GCD(m,n)} & \overbrace{0..}^{GCD(m,n)} \\ \dots & & \\ \underbrace{0..}_{GCD(m,n)} & \underbrace{0..}_{GCD(m,n)} & \underbrace{1..}_{GCD(m,n)} \end{matrix}} \Rightarrow \boxed{\begin{matrix} \boxed{x_{11}}.. & \boxed{x_{21}}.. & \boxed{x_{\frac{LCM(m,n)}{GCD(m,n)}1}}.. \\ \boxed{x_{12}}.. & \boxed{x_{22}}.. & \boxed{x_{\frac{LCM(m,n)}{GCD(m,n)}2}}.. \\ \dots & & \\ \boxed{x_{1\frac{LCM(m,n)}{n}}}.. & \boxed{x_{2\frac{LCM(m,n)}{n}}}.. & \boxed{x_{\frac{LCM(m,n)}{GCD(m,n)}\frac{LCM(m,n)}{n}}}.. \end{matrix}}$$

The reason for choosing $\overbrace{\boxed{x_{ij}}}^{GCD(m,n)}$ as basic structure of $\boxed{B}$ is the following:

The height of the elements $\boxed{B}$ is $\frac{LCM(m,n)}{n}$, within element $\boxed{B}$, for each column $GCD(m,n) \times \frac{LCM(m,n)}{n} = m$, which means $\sum_{j=1}^{\frac{LCM(m,n)}{n}} length(x_{ij}) = m$, where m is the period length of the first *P-adic* sequence. For the same *i*, $\overbrace{\boxed{x_{ij}}}^{GCD(m,n)}$ will be the different parts from dividing same period of first *P-adic* sequence.

It can be shown that:

$$Sum(\boxed{x_{i1}} + \boxed{x_{i2}} + \cdots + \boxed{x_{i\frac{LCM(m,n)}{n}}}) = Sum(\boxed{x_{j1}} + \boxed{x_{j2}} + \cdots + \boxed{x_{j\frac{LCM(m,n)}{n}}}), i \neq j$$

From above, the period within elements $\boxed{B}$ is $GCD(m,n)$.

Then, $C_B = GCD(m,n)$, and the period of *Part B* is $LCM(m,n) \times (P^{GCD(m,n)} - 1)$.

The total period of (*Part A + Part B)* is:

$$M\big(\ LCM(m,n),\ LCM(m,n) \times \big(P^{GCD(m,n)} - 1\big)\ \big) = LCM(m,n) \times \big(P^{GCD(m,n)} - 1\big)\,.$$

∵ The length of (*Part 1 + Part 2)* is $LCM(m,n)$, *Part 4* is a constant and *Part 5* is

$$LCM(m,n) \times \big(P^{GCD(m,n)} - 1\big).$$

∴ The total length of (*Part 1 + Part 2 + Part 4 + Part 5)* is

$$LCM\big(\ LCM(m,n),\ LCM(m,n) \times \big(P^{GCD(m,n)} - 1\big)\ \big) = LCM(m,n) \times \big(P^{GCD(m,n)} - 1\big).$$

⇨ The total period for multiplication is

$$LCM(m,n) \times \big(P^{GCD(m,n)} - 1\big).$$

The length of periodicity for multiplication is too large. For 32-bits, the prime *P* is chosen as 46337 and 64-bits as 2147483647. By the formula of $LCM(m,n) \times \big(P^{GCD(m,n)} - 1\big)$, it is impossible to set the sequence length.

3.2. Finite P-adic Number System (Hensel Code)

Hensel code was introduced by Krishnamurthy [8][32]. The idea is to use finite P-adic sequence to represent the fraction number. According to the theory:

For a rational number $\frac{b}{a}$, $GCD(a,b) = 1$, if P-adic sequence length r satisfying $\max(a,b) \leq \sqrt{\frac{P^r - 1}{2}}$, first r length of the P-adic sequence of this rational number can be

uniquely used to represent this rational number. And the P-adic sequence can be conversed back to the original rational number.

The Krishnamurthy's idea is to reflect the finite P-adic sequence in Farey rationals [32]. The following quotation part is coming from [17] and it will give great help on understanding Krishnamurthy's conversion algorithm.

The notation$|x|_p$, $|X|_p$ will be used to denote the residue of integer x and matrix X with respect to positive integer p, via

$$\left|x\right|_p = x \bmod p$$
$$\left|X\right|_p = X \bmod p$$

Hence if $0 < a < p$, then there exists a unique integer b, $0 < b < p$ such that

$$ab \equiv 1 \bmod p$$

The integer b is called the inverse of a modulo p and is denoted by $|a|_p^{-1}$. This permits us to have a unique representation for integers in the range $\frac{-(p-1)}{2}$ to $\frac{(p-1)}{2}$.

The following function *value* maps the residue of a, $0 \le a < p$, to the corresponding positive or negative integer

$$value(a) = \begin{cases} a & \text{if } a \le \dfrac{(p-1)}{2} \\ -(p-a) & \text{otherwise} \end{cases}$$

$Bound\,(Ax = b)$ means that during the calculation of $Ax = b$ the largest denominator or numerator will show out.

In the *P*-adic representation $a_1 a_2 a_3 \cdots a_r e$, *e* means the position of radix point.

$$I(m) = \sum_{i=0}^{r} a_i p^i$$

r means the length of the P-adic expansion.

A rational number $\frac{a}{b}$ $(0 \le b < p, 0 < a < p)$ can therefore be represented in the form $\left| b a_{p^{-1}} \right|_p \left(\left| b a_{p^{-1}} \right|_p = \frac{b}{a} \bmod p \right)$, if ka is known, then the rational number can be converted back by:

$$\frac{1}{ka} \left| kaba_{p^{-1}} \right|_p \qquad |ka|, |kb| \le p$$ " [17]

The Krishnamurthy's algorithm is as the following [7]:

Step1. Change all the rational numbers into *P*-adic series and record m_1, which is the LCM of all the denominators. Make sure the series length *r* satisfying $\sqrt{\frac{p^r - 1}{2}} \ge Bound(Ax = b)$.

Step2. Using P-adic & Hensel codes arithmetic to get the solution of *Ax = b*, record m_2 which is the product of *P*-adic expansions and used as divisor during the calculation process.

Step3. $k = m_1 m_2$, make sure $m_1 m_2 \cdot \max(numerators, denominators) \le p^r$. Then we can use the way mentioned before to convert P-adic series into rational number.

Step4. Convert the output entries to fraction numbers using:

$$\alpha = p^{e}\frac{value\left(I\left(m_c \cdot m_\alpha\right)\bmod p^{r}\right)}{value\left(I\left(m_c\right)\right)}$$

In 1982, Dixon [12] introduced an algorithm to reflect the finite P-adic sequence in Continued Fraction. The algorithm is based on Euclidean algorithm. Miola also mentioned it [13]. Koc did a summary [25]. For a finite *p*-adic sequence $.a_1a_2\cdots a_i$,

Euclidean Algorithm

$x = \sum_{i=0}^{r} a_i p^i$

$u_{-1} = p^r$, $u_0 = x$

$v_{-1} = 0$, $v_0 = 1$

Begin $i = 0$

While $u_i < p^{r/2}$

$q_i = \lfloor u_{i-1}/u_i \rfloor$

$u_{i+1} = u_{i-1} - q_i u_i$

$v_{i+1} = v_{i-1} + q_i v_i$

End

Rational Number $\alpha = \frac{(-1)^i u_i}{v_i}$

Figure 3.1 Euclidean Algorithm

There is also a condition given from Dixon. Define $\frac{a}{b}$, $GCD(a,b) = 1$ and $\delta = max(a,b)$, if δ satisfies $\delta \le \lambda\sqrt{p^r}$ ($\lambda = 0.618\cdots$ is a root of $\lambda^2 + \lambda - 1 = 0$), r means the *p*-adic sequence length, we can use the decoding algorithm to get the rational number back.

For finite P-adic number system, Krishnamurthy's theory supplied a good idea to separate the data structure from the math algorithm. While Dixon's theory gave a better idea to convert finite P-adic sequence back to rational number. We combined these theories and supply an algorithm called Dixon-Krishnamurthy Algorithm (D-K algorithm).

3.3. Dixon-Krishnamurthy Algorithm [16]

Dixon- Krishnamurthy algorithm (D- K algorithm) is based on Dixon and Krishnamurthy theory. D- K algorithm includes conversion between rational number and *P*-adic sequence, length *r* prediction and overflow detection. D- K algorithm is a data structure algorithm. Other math algorithm can directly use this algorithm, with little changing.

3.3.1. Algorithm Implementation Process

In computer representations, there are different types of integers, such as short, int, long, and double. Each type has its own range. For example, the range of short is $[-32768, 32767]$. If there is a number larger than 32767, during the computing process, errors will come. The solution is using other type with larger range, such as long. The range of long is $[-2147483648, 2147483647]$. Similarly, there are data types for P-adic number setting. And the rage set is decided by both the selected prime and number sequence length *r*. It is proved that the Hensel code is unique, as long as the absolute value of the numerator or denominator does not exceed $\sqrt{(p^r - 1)/2}$. For example, when$p = 2$ & $r = 5$, $\sqrt{\frac{(p^r-1)}{2}} = 3.93$, the rage of number can be used is

$$\begin{bmatrix} -\frac{3}{1} & -\frac{2}{1} & \frac{1}{1} & \frac{2}{1} & \frac{3}{1} \\ -\frac{3}{2} & -\frac{1}{2} & 0 & \frac{1}{2} & \frac{3}{2} \\ -\frac{2}{3} & -\frac{1}{3} & 0 & \frac{1}{3} & \frac{2}{3} \end{bmatrix}.$$

If the number out of this range is needed, the p & r setting should be changed.

Step 1. Predict Expansion Length r.

As Dixon's theory: $\delta \leq \lambda\sqrt{p^r}$, δ means the largest integer among denominators and numerators would show out during the calculation process, *(*$\lambda = 0.618\cdots$ is a root of $\lambda^2 + \lambda - 1 = 0$*),*

$$r = \frac{2\log(\delta / \lambda)}{\log(p)}$$

When meeting some matrix computation systems, which can predict the bound, this formula supplies a way to predict the length of the *P*-adic sequence.

For example, to predict the *r* for calculating inverse for *n* by *n* matrix *A*, by Hadamard's inequality, as given in [19].Assume *m* is the largest integer among the numerators and denominators in matrix *A*, the largest integer δ during the calculation process will be less than the product of the Euclidean lengths of its row vectors. $\delta \leq n^{\frac{n}{2}}m^n$,

$$\text{so } r = \frac{2\log(n^{\frac{n}{2}}m^n/\lambda)}{\log p}.$$

For the prediction issue, we will discuss more on the following chapters.

Step 2. Convert All Rational Numbers into P-adic Expansions

After defining P & r, all the rational numbers will be converted into Hensel codes by P & r. Use *P*-adic arithmetic to do calculation. Then the Hensel code result will be given.

Step 3. Result Verification and Convert back into Rational Numbers

Theorem 3.3.1 Using D- K Algorithm, for matrix calculations, if there are only addition, subtraction, multiplication or division operations, the P-adic sequence result by using length r ($r \geq 2$), will be equal to the first r sequence of the P-adic sequence result by using length $r + k$ ($k \geq 0$), only if during calculation process, no P-adic sequence is used as dividend which has the form of all zero in sequence (convert to rational number will has form of p^r/x. (p is the Prime used by P-adic sequence, $x \in \mathbb{Z}, x \neq 0, x \neq 0 \bmod p$)).

The proof process will be described on the overflow detection section.

The result verification is based on the following simple idea: for a rational number $a/$b, if the length R is enough for P-adic sequence, then the conversation from P-adic sequence R and R+1 should be the same rational number. (This verification cannot make sure the accuracy exactly, but for practical implementation it is good enough).

The Euclidean algorithm will be used to convert *P*-adic expansions back into rational numbers.

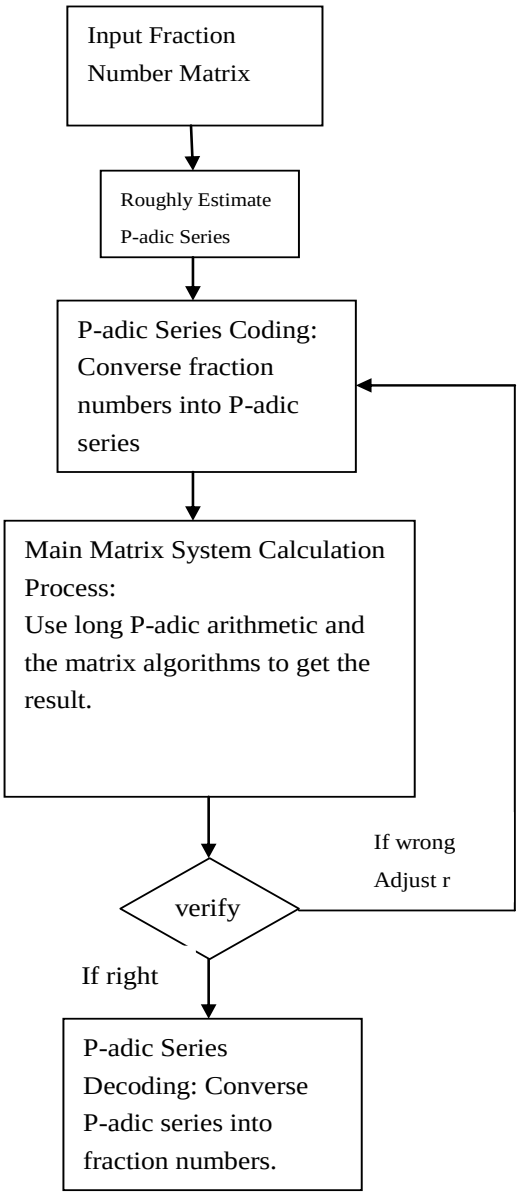


Figure 3.2 Dixon – Krishnamurthy Algorithm Overview Flowchart

3.3.2. P-adic Arithmetic Using Long- digit Method

When D- K algorithm is implemented on computer, the computer resource is limited.

The prime P and length r is also limited. We have proved that: if we keep the prime *P* of the *P*-adic series satisfying $P^2 < m \ \& \ r < m + 2 - P$ (m means the largest integer of kind of computer data type, *r* means the length of the P-adic expansions). We can keep the arithmetic of addition, subtraction, multiplication and division under the long integer variables, which will greatly improve the efficiency at the practice level.

For the 32-bit computer architecture, for *int* data type the *P* is chosen as 46337, *r* < 2147437310. For the 64-bit computer architecture, for *int* data type the P is chosen 2147483647, *r* < 9223372036854775807.

The proof process is given in Appendix B.

3.3.3. Predict P-adic Expansion r for Complex Matrix System

If a matrix calculation system is too complex to predict the bound integer, we will do another calculation process for D- K algorithm. We will try to make a relationship between data size and P-adic expansion length *r*. The process will be changed to as Figure 3.3.

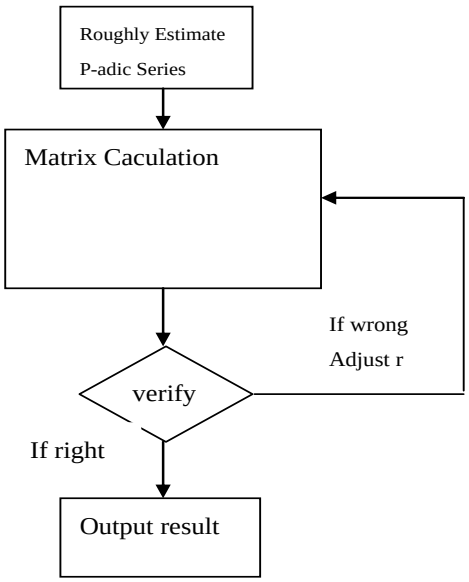


Figure 3.3 D– K Algorithm Improved Flow Chart

Actually, we need to choose a bigger *r* as the matrix size gets bigger, the choice of *r* can be determined by running experiments. The following is an example for calculating 1000 Moore-Penrose inverse, at the different length and to get the accuracy, where we select:

$$P = 2147483647.$$

The range of the denominator or numerator on the input matrix is random from -50 to 50.

Matrix size	$r = 40$, the accuracy	$r = 45$, the accuracy	$r = 50$, the accuracy
10	100%	100%	100%
15	100%	100%	100%
20	1.20%	84.30%	100%

Table 3.1 D – K Algorithm Accuracy Comparing Flow Chart

3.4. Hensel Code Overflow Detection

Definition: Let β be a rational number and $a_{-n}, a_{-n+1}, \cdots, a_{-1}, a_0, \cdots, a_k, \cdots$ be its *P*-adic expansion. Then the finite segment $a_{-n}, a_{-n+1}, \cdots, a_{-1}, a_0, \cdots, a_k$, where $r = n + k + 1$ is called the Hensel code of β and is denoted by $H(p, r, \beta) = (a_{-n}a_{-n+1} \cdots a_{-1}a_0 \cdots a_k, i)$[7]. Where *p* is the prime, *r* is the length of *P*-adic sequence, and *i* is the position of the *dot*.

Some examples:

$$H(7,4,-1/3) = (.2222,0)$$

$$H(7,4,1/21) = (.5444,-1)$$

If *r* and *p* are fixed, any rational number $\frac{b}{a}, GCD(a,b) = 1$, will have a unique Hensel code representation, if it satisfies $0 < |a,b| \leq \sqrt{(p^r - 1)/2}$ [8].

Hensel code overflow*:* For Hensel code $H\left(p, r, \frac{\mathrm{a}}{\mathrm{b}}\right), \mathrm{GCD(a,b)} = 1$, when it satisfies $|a,b| > \sqrt{(p^r - 1)/2}$, the rational number, which the Hensel code represents, cannot be uniquely recovered by the inverse transformation. In this situation, we call it Hensel code overflow.

Notation*:* $Decoding(x,i) and\ Decoding(X,i)$ will be used to donate decoding Hensel code *x* and Hensel code matrix *X* into rational number and rational number matrix by first *i* digits.

For example, we take prime $p = 7$,

$$x = (.363000,0)$$

$$Decoding(x,4) = -2/25$$

$$Decoding(x,6) = 192$$

$$X = \begin{matrix} (.1000000,0) & (.4333333,0) & (.5444444,0) \\ (.4333333,0) & (.5444444,0) & (.2515151,0) \\ (.5444444,0) & (.2515151,0) & (.3145214,0) \end{matrix}$$

$$Decoding(X,4) = \begin{matrix} 9 & -36 & 30 \\ -36 & -2/25 & 3/40 \\ 30 & 3/40 & -3/40 \end{matrix}$$

$$Decoding(X,7) = \begin{matrix} 9 & -36 & 30 \\ -36 & 192 & 180 \\ 30 & 180 & 180 \end{matrix}$$

3.4.1 Overflow Detection Method

We give a method to detect the Hensel code overflow problem, just by using the prime *p* and Hensel code itself. In this method, each Hensel code should have a verification part *k*. This part will be sacrificed on Hensel code overflow detection. For Hensel code *x* with *P*-adic sequence length $i + k$, if $Decoding(x, i) \neq Decoding(x, i+k)$, then Hensel code overflow happened.

For example of Hensel code *x*, where

$$x = (.a_0 a_1 \cdots a_i a_{i+1} \cdots a_{i+k}, 0)$$

will be treated as

$$x = (a_0 a_1 \cdots a_i \underbrace{a_i \cdots a_{i+k}}_{verification\ part}, 0).$$

Overflow happened, if:

$$Decoding(x, i) \neq Decoding(x, i + k).$$

Overflow did not happen, if:

$$Decoding(x, i) = Decoding(x, i + k).$$

For example: By taking prime p = 7, Hensel codes

x = (.64121144213620046103, 0),

y = (.1033421534, 0).

We take the last 3 digits as verification part,

Decoding(x, 17) = 1/33333,

Decoding(x, 20) = 1/33333.

But, Decoding(y, 7) = -45/53,

which is not equal to

Decoding(y, 10) = 908/1545.

By our method, we take x as 1/33333. And for y we experienced an overflow problem.

PROOF

The proof of the method will contain two parts:

Part 1. Overflow happens:$Decoding(x,i) \neq Decoding(x,i+k)$.

Before the proof of part 1, let us introduce theorem 3.4.1.1.

Theorem 3.4.1.1. For any rational number $\frac{a}{b}(GCD(a,b)=1)$, given any prime *P*, the *P*-adic sequence$a_0 a_1 \cdots a_k \cdots$ of the rational number satisfies that

$$\lim_{i\to\infty} Decoding\big((a_0 a_1 \cdots a_k \cdots,0),i\big) = \frac{a}{b}$$

For example, take the prime *p=7*, and a rational number 1/143, its Hensel code $x =$ (.51035550652456560502,0)

Decoding(x, 2) = -1/4

Decoding(x, 3) = 12

Decoding(x, 4) = 17/30

Decoding(x, 5) = 118/67

Decoding(x, 6) = 1/143

Decoding(x, 7) = 1/143

Decoding(x, 8) = 1/143

…

Proof: It is well known that any rational number $\frac{a}{b}\,(GCD(a,b)=1)$ will have unique Hensel code representation, if it satisfies $0<|a,b|\le\sqrt{(p^r-1)/2}$. The theorem 1 can be directly proved from the above statement. Thus part 1 of theorem 1 is proved.

Part 2. Overflow doesn't happen: $Decoding(x,i)=Decoding(x,i+k)$

This part cannot be proved. The statement, strictly speaking, should be that there is a high likelihood, say 99.999999% or more, (experiment shows that when prime *P* is large enough, or *k* is long enough, it can be 100%), that overflow doesn't happen, when *Decoding (x, i) = Decoding (x, i + k).*

In order to prove ***Part 2***, we make the following guess.

Guess: For any rational number $\frac{a}{b}\,(GCD(a,b)=1)$, given any prime *p*, with the condition $|a,b|>\sqrt{(p^r-1)/2}$, the *P*-adic sequence $a_0a_1\cdots a_r\cdots a_{r+k}\cdots$ of the rational number satisfies:

a. With the increase of prime *p*, the probability of

   $Decoding\big((a_0a_1\cdots a_r\cdots a_{r+k}\cdots,0),r\big)=$

   $Decoding((a_0a_1\cdots a_r\cdots a_{r+k}\cdots,0),r+k)$ decreases.

b. If prime *p* is fixed, then with the increase of *k*, the probability

   of $Decoding\big((a_0a_1\cdots a_r\cdots a_{r+k}\cdots,0),r\big)=$

   $Decoding\big((a_0a_1\cdots a_r\cdots a_{r+k}\cdots,0),r+k\big)$ decreases.

We cannot prove the guess, but we designed experiments to show the property.

Experiment 3.4.1.1, each time, the prime p is fixed. We randomly take 1000 rational numbers, $\frac{a}{b}(GCD(a,b)=1), |a,b| \leq 10^{100}$. When $|a,b| > \sqrt{(p^r-1)/2}$, we compare whether *Decoding(H(P, i+k, a/b), i) ≠ Decoding(H(p, i+k, a/b), i+k), i+k ≤ r*. If it happens, which means that it is a possible mistake in our judgment, and we record it as one error. The following diagrams show that when k = 1 and k = 3, the primes versus the percentage of errors.

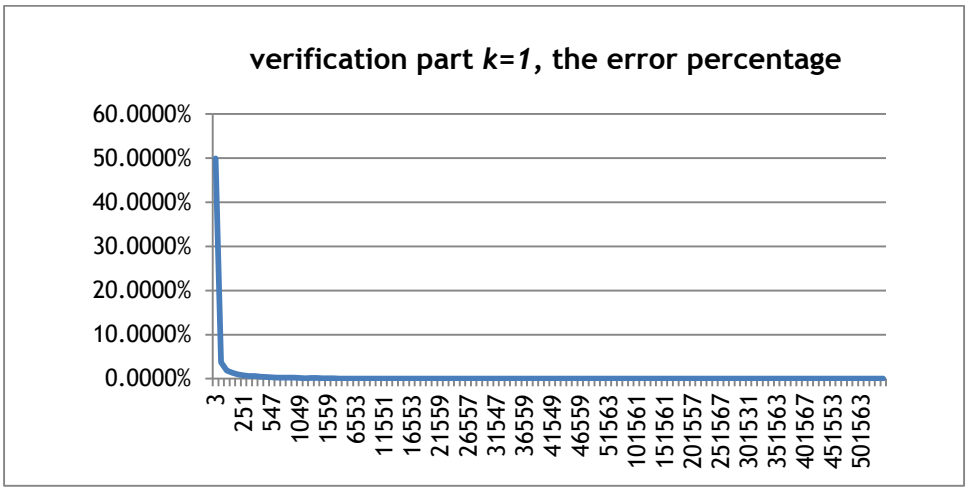


Figure 3.4 k = 1, the primes versus the percentage of errors; Vertical axis: Error percentage; Horizontal axis: Prime value

When verification part *k = 1*, and prime p > 100,000, the percentage of errors goes down to 0.

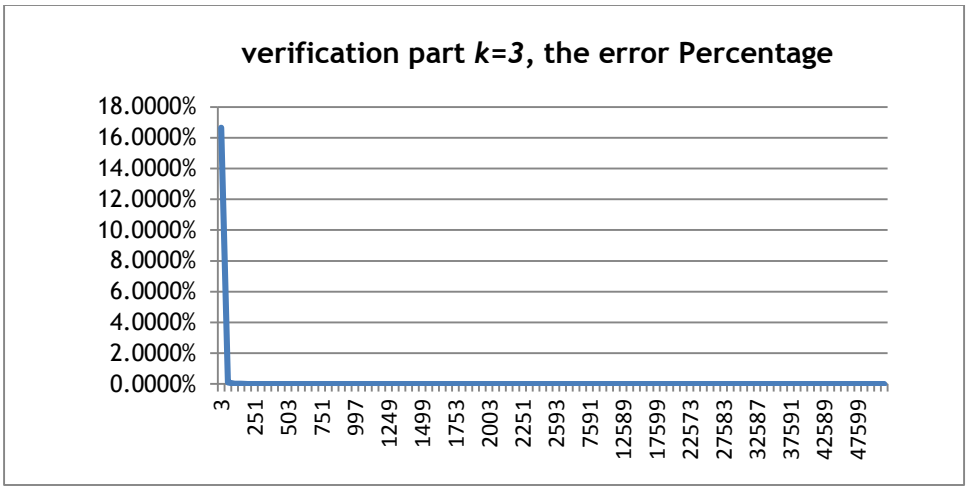


Figure 3.5 k = 3, the primes versus the percentage of errors; Vertical axis: error percentage; Horizontal axis: prime value

When verification part $k = 3$, the percentage of errors goes down to 0 much earlier. When prime p=600 or larger, the percentage of errors is close to 0.

We also did experiments with prime p =2147483647 fixed, and choose random rational numbers $\frac{a}{b}(GCD(a,b)=1), |a,b| \leq 10^{3000}$. For large rational number (a/b) with $|a,b| \leq 10^{3000}$, we choose up to 700 elements for P-adic sequence, which translate to around 700 comparison tests for each rational number, and we tested more than 2000 very large random rational numbers. The percentage of errors is 0 with verification part $k=1$ for all the tests.

Experiment 3.4.1.2, in this experiment, the prime p is fixed, we try to compute the error rate versus verification part k. Each time, we randomly take 1000 rational numbers $\frac{a}{b}(GCD(a,b)=1), |a,b| \leq 10^{100}$. When $|a,b| > \sqrt{(p^r-1)/2}$, we compare whether *Decoding(H(P, i+k, a/b), i) ≠ Decoding(H(P, i+k, a/b), i+k), i+k ≤ r*. If it happens, which means that it is a possible mistake for our judgment, and we record it as one error.

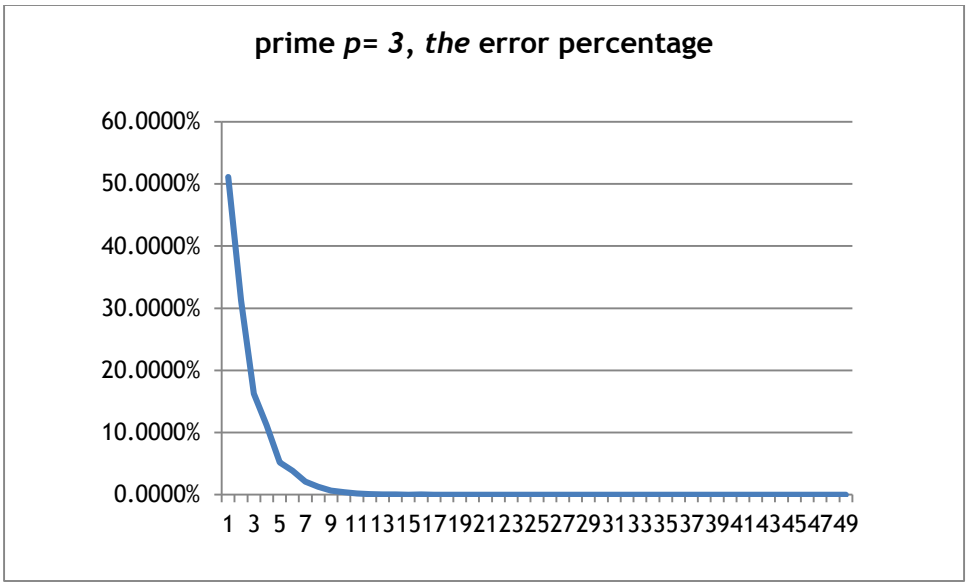


Figure 1.6 Fix prime p =3, the verification part k versus the percentage of errors; Vertical axis: error percentage; Horizontal axis: the value of k

When the prime is p =3, as verification part k >18, the percentage of errors goes to 0.

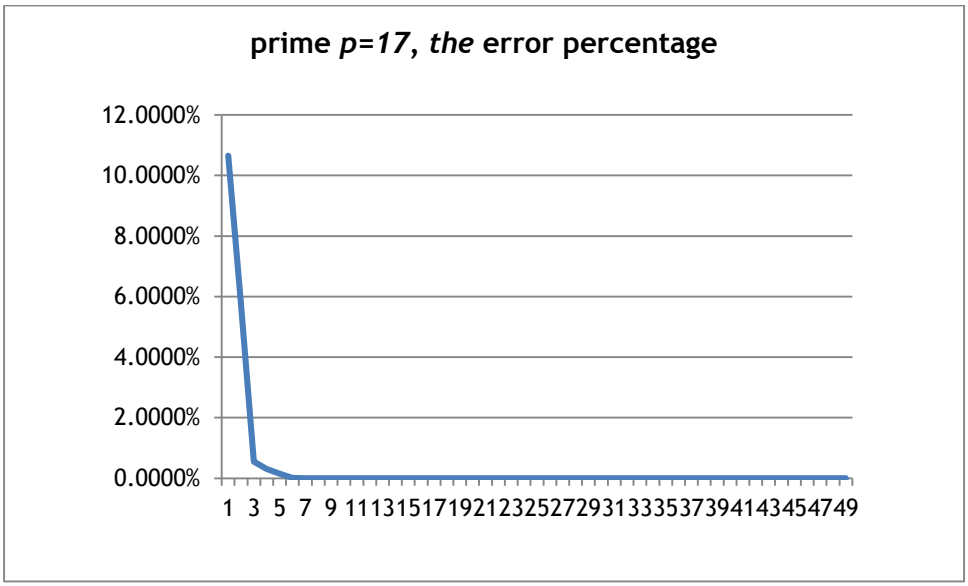


Figure 3.7 Fix prime p = 17, the verification part k versus the percentage of errors; Vertical axis: error percentage; Horizontal axis: the value of k

When we take the prime p =17, as verification pare $k > 6$, the percentage of errors goes to 0.

3.4.2. Practical Consideration

Our goal is to use Hensel code arithmetic on exact matrix calculation. We have developed an algorithm called the Dixon-Krishnamurthy algorithm [14, 16] (D–K algorithm). This algorithm can do any rational matrix computation using Hensel code arithmetic. Here is an example of using D–K algorithm and Hensel code overflowing detection method for matrix inverse calculation.

$$x = \begin{matrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{matrix}$$

We take the prime $p=7$, we choose the length of Hensel code $r = i + k$, ($i = 6$, $k = 1$)

$$x \xrightarrow{\text{Hensel code}} \begin{bmatrix} (.1000000,0) & (.4333333,0) & (.5444444,0) \\ (.4333333,0) & (.5444444,0) & (.2515151,0) \\ (.5444444,0) & (.2515151,0) & (.3145214,0) \end{bmatrix}$$

$$\xrightarrow{\text{Inverse}} \begin{bmatrix} (.2100000,0) & (.6166666,0) & (.2400000,0) \\ (.6166666,0) & (.3630000,0) & (.2236666,0) \\ (.2400000,0) & (.2236666,0) & (.5430000,0) \end{bmatrix}$$

$$\xrightarrow{\text{Overflow detection}} Decoding\left(Inv(s),6\right) = Decoding\left(Inv(s),7\right)$$

No Hensel code overflow

$$\xrightarrow{\text{Decoding}} \begin{bmatrix} 9 & -36 & 30 \\ -36 & 192 & 180 \\ 30 & 180 & 180 \end{bmatrix}$$

Theorem 3.4.2.1 Use *P*-adic arithmetic for rational matrix calculation, choose r ($r \geq 1$) as *P*-adic sequence length to get the result R_1; for the same calculation process, choose $r + k$ ($k \geq 0$) as *P*-adic sequence length to get the result R_2; the first r elements on each *P*-adic sequence of R_2 are equal to the corresponding r elements of the *P*-adic sequence of R_1, if during both the calculation processes, no *P*-adic sequences with elements of all zeros are used as dividend.

Since the *P*-adic arithmetic [7] calculates from left to right, by the *P*-adic arithmetic rules, this theorem can be easily proved.

From **Theorem 3.4.2.1**, there is an interesting property when we use D–K algorithm with this method.

Property: If the length r is not long enough for a matrix calculation, we cannot get all the correct answers in the whole matrix, but we can identify which elements in the matrix are correct.

Taking the inverse of Hilbert Matrix [5x5] as an example, if the prime $p = 751$, verification part $k = 3$, and length $r = 6$, using Gaussian elimination to get the results as the following:

25	−300	1050	−1400	630
−300	4800	−18900	−16291/15757	−12600
1050	−18900	6929/5336	−4196/14407	−15751/7470
−1400	−16591/15757	−4196/14407	−3147/14407	12947/7091
630	−12600	−15751/7470	−3147/14407	15749/9605

We do not know whether overflow has happened during the calculation process. By using our method, we can identify the correct elements in the matrix as the following:

25	–300	1050	–1400	630
–300	4800	–18900	*error*	–12600
1050	–18900	*error*	*error*	*error*
–1400	*error*	*error*	*error*	*error*
630	–12600	*error*	*error*	*error*

For a Hilbert matrix H_{ij}, we can directly get the matrix inverse [14]:

$$\alpha_{ij} = (-1)^{i+j}(i+j-1)\binom{n+i-1}{n-j}$$

$$\binom{n+j-1}{n-i}\binom{i+j-2}{i-1}^2$$

From the above formula, the right answer for the inverse of Hilbert Matrix [5x5] should be the following:

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

Comparing the matrix inverse results, we can find that the matrix inverse has an overflow problem, but correct entries of the resulting matrix can be identified.

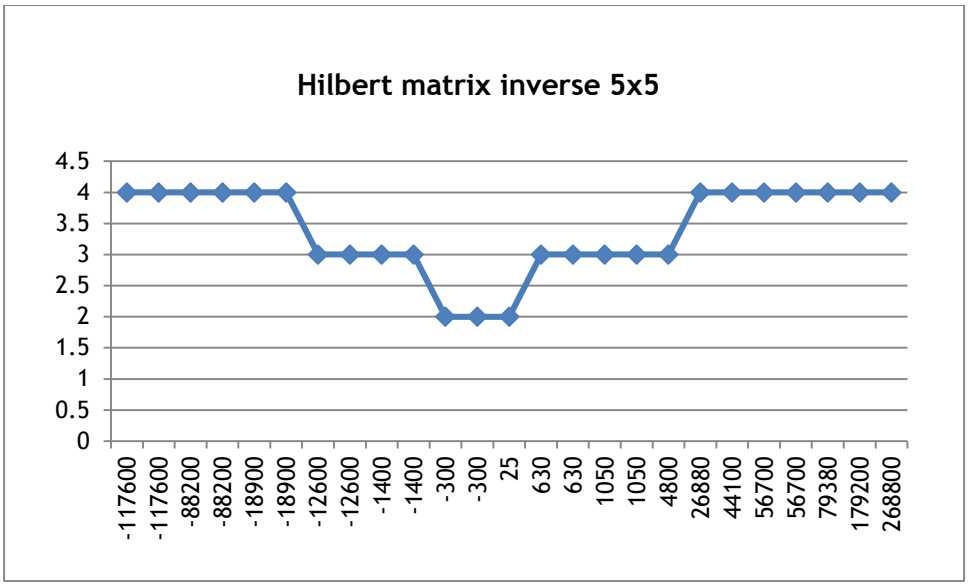


Figure 3.8 Hilbert matrix inverse 5 x 5; Vertical axis: value of $r - k$; Horizontal axis: value of Hilbert matrix inverse 5 x 5, ordered from small to large

In Figure 3.8, the prime $p = 751$, the actual length is *r-k*, which *r* is the total length used for calculation, and *k* is the verification part length, *(r-k)* means the effective length needed for the Hensel code conversion. The example in this diagram is the exact representation of the rational numbers from the right answers of the inverse of Hilbert Matrix [5x5].

In this diagram, the horizontal axis shows the exact rational number results of the inverse of Hilbert matrix [5x5], and the vertical axis shows the length *r-k* (*k* is the verification part), which satisfies $|a, b| \leq \sqrt{(p^{r-k} - 1)/2}$, from rational number result $\frac{a}{b}(GCD(a, b) = 1)$. The diagram shows that the longer length *r-k*, the more right answers we can identify. If the length *r-k>4*, then we will get all the answers correctly.

Experiment 3.4.2.1. In our following experiments, prime *p*=2147483647 is fixed (which is the largest prime we can use on *64*-bits computer architecture) and verification part *k*=1. We generate random square rational matrix 30x30, each element $\frac{a}{b}(GCD(a,b)=1)$

satisfies $|a,b| \leq 200$. We do matrix inverse on the generated matrix with both the NTL-RationalNumber package and the *P*–adic arithmetic package. The RationalNumber package is based on NTL library [14], which can dynamically represent integers of any size. The NTL-RationalNumber package can realize exactly rational number calculation too. We compare the results from both packages. If one element from the *P*–adic package passes Hensel code overflow detection, but it is not equal to the element from the NTL-RationalNumber package, an error happened. We also use the NTL-RationalNumber package for comparison to test the correctness of our *P*–adic arithmetic.

We have taken 200 random matrices and did 960000 comparisons. The error rate is 0. We have developed a software package [10, 15] to do matrix calculation by D-K algorithm and the Hensel code overflow detection method. This software has been well tested on matrix calculations so far, and the performance is good.

3.5. Compare Rational Number System with Finite P-adic Number System

If we choose a large prime p = 2147483647 and sequence length r = 4 for *P*-adic sequence, then *P*-adic sequences will need 4 units to represent each rational number regardless of the size of the number. When the size of the rational number is large, it does the same amount of operations. When the size of the number is relatively small, Rational

Number System will need fewer units to represent the number, and then do fewer operations. Here are two extreme examples:

a) If the number is 9 (one digit only), *Rational Number System* represents it with one unit, while *P*-adic sequences will need 4 units.

b) If the number is 123456789123456789, *Rational Number System* needs 18 units, while *P*-adic sequences still only need 4 units. If we do calculation uses 123456789123456789, *Rational Number System* should do operations with 18 units, while *P*-adic sequences only need 4 units.

Rational Number System carries out exact calculation by rational number arithmetic, which means *Rational Number System* represent numerator and denominator separately. The calculations for *addition* and *multiplication* are as the following:

Notation: *GCD (a, b)* means Greatest Common Divisor of *a* and *b*.

Addition process:

$$\frac{a_1}{b_1}+\frac{a_2}{b_2}=\frac{a_1b_2+a_2b_1}{b_1b_2}=\frac{\left(a_1b_2+a_2b_1\right)/GCD\left(\left(a_1b_2+a_2b_1\right),b_1b_2\right)}{b_1b_2/GCD\left(\left(a_1b_2+a_2b_1\right),b_1b_2\right)}.$$

Multiplication processes:

$$\frac{a_1}{b_1}\times\frac{a_2}{b_2}=\frac{a_1a_2}{b_1b_2}=\frac{a_1a_2/GCD\left(a_1a_2,b_1b_2\right)}{b_1b_2/GCD\left(a_1a_2,b_1b_2\right)}.$$

During the rational number arithmetic process, the *GCD (numerator, denominator)* must be found. This process costs a lot of time. The Euclidean algorithm is used to find *GCD* in our computation, and we find that it is the most efficient algorithm.

The following charts are the experimental results of testing the calculation time for both *Rational Number System* and *Finite P-adic Number System*. We did 100,000 times of operations for rational numbers from 99/98 to 9999…9/99999…98.

For the *P*-adic sequence, we choose prime P = 2147483647, length r = 30, then $\sqrt{\frac{P^r-1}{2}} \approx 6.73 \times 10^{139}$, which means that the denominators and numerators must be smaller than 6.73×10^{139}. During the multiplication calculation process, we have to make sure that all numbers are smaller than $\sqrt{6.73 \times 10^{139}}$, that translates to 10^{69}. In the following charts, we calculated up to 10's power of 60, which guarantees the arithmetic process not overflowing the range of *P* –adic sequence. The horizontal axis represents the time (in seconds) for both data type. The vertical axis represents the number of digits of denominators and/or numerators.

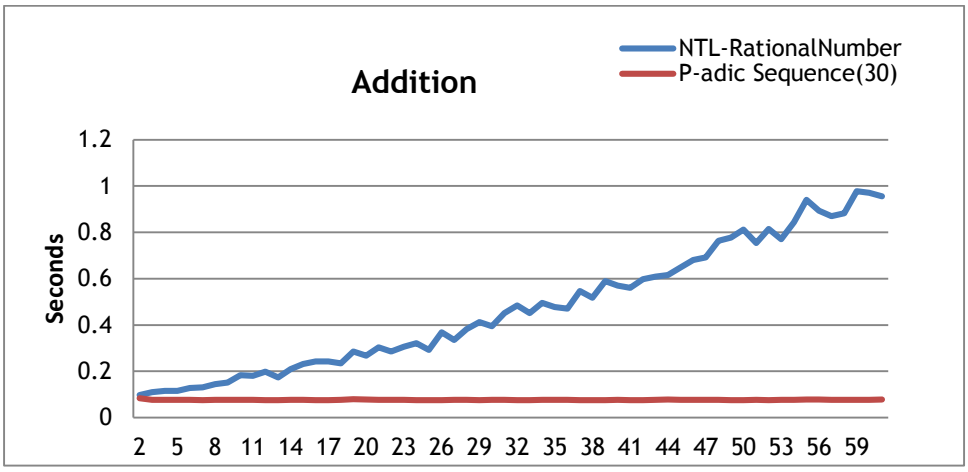


Figure 3.9 Efficiency Comparison for Additions

For additions, *P*-adic method is always faster than NTL method for all sizes due to no GCD calculation is needed during the computational process.

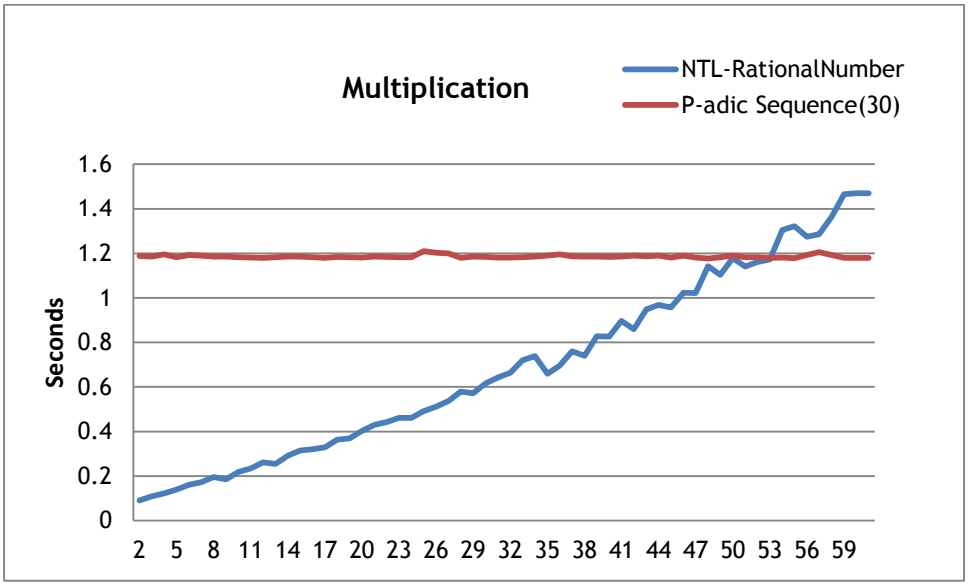


Figure 3.5.2. Efficiency Comparison for Multiplications

For multiplications, NTL method is faster for small sizes less than 50, *P*-adic method is faster for large sizes (> 50 digits) due to the symbolic representation of all digits for NTL method, the GCD calculation gets really slow for rational numbers with more digits during the computational process.

Chapter 4

Multiple P-adic Data Type

We have been developing *P*-adic Exact Scientific Computational Library (ESCL) for rational matrix operations. Based on Krishnamurthy [7, 17] and Dixon [12] theories, we have established a finite *P*-adic sequence calculation system [6, 16, 27 and 42]. But there is a problem that for certain complex matrix operations, even with small matrix sizes, the new method requires a long *P*-adic sequence to guarantee against overflow [6]. The longer the *P*-adic sequences are, the longer the calculation will take, and computational efficiency becomes an issue. One solution to this problem is to adopt parallel computing. It is difficult to realize parallel computation directly in *P*-adic arithmetic due to its data structure. If we combine the multiple modulus rational systems [33] and the *P*-adic arithmetic, then parallel computation can be realized, which was called multiple *P*-adic arithmetic by Morrison [23]. A similar idea was also mentioned by Limongelli, Loidl [24] and Koc [25]. This chapter will be focused on parallel implementation of multiple *P*-adic arithmetic applied to rational matrices using *P*-adic exact computation. Overflow detection will also be addressed. Finally, comparison tests and experimental results will be presented.

4.1. Extended Chinese Remainder Theorem

Recalling the Theorem (Chinese remainder theorem) [17, 19], if $r \sim \{r_1, r_2, \cdots, r_s\}$ is the residue representation of an integer r with respect to moduli $\{p_1, p_2, \cdots p_s\}$, where, $GCD(p_i, p_j) = 1$ for i ≠ j, define $p = \prod_{i=1}^{s} p_i$ and p_i' by $\frac{p}{p_i} p_i' \equiv 1 \, mod \, p_i$, then the solution of the system is given by

$$r \equiv \sum_{i=1}^{s} \frac{p}{p_i} p_i' r_i \mod p$$

If the given condition is $|r| < \frac{1}{2} p$, the value of r can be identified by:

$$r = \begin{cases} r & \text{if } r \leq (p-1)/2 \\ -(p-r) & \text{otherwise} \end{cases}$$

For example:

$$r \equiv 2 \mod 3$$

$$r \equiv 3 \mod 4$$

$$r \equiv 4 \mod 5$$

According to the Chinese remainder theorem,

$$p = 60$$

$$p_1' = 2$$

$$p_2' = 3$$

$$p_3' = 3$$

$$r \equiv 59 \, mod \, 60$$

If given condition $|r| < \frac{1}{2}p$,

$$r = -1$$

4.1.1. Extended Chinese Remainder Theorem to Rational Numbers [33]

The Chinese remainder theorem deals with integers. It shows how to transform a large integer into a sequence of small integers. There is also a way to transform a fractional number with a large numerator and/or denominator into a sequence of small integers. This method has been named as multiple module number systems [33], which we like to call it the extended Chinese remainder theorem.

a. How to calculate rational module

For a rational number $\frac{b}{a}$ with $GCD(a, b) = 1$, the calculation of $\frac{b}{a}\ mod\ p$ ($p \geq 0, p \in$ Z) is defined as

$$r = ba' \bmod p\ (aa' \bmod p \equiv 1)$$

b. How to decode from the extended Chinese remainder theorem [11]

If $r \sim \{r_1, r_2, \cdots, r_s\}$ is the residue representation of a rational number r with respect to moduli $\{p_1, p_2, \cdots p_s\}$ where $GCD(p_i, p_j) = 1$ for $i \neq j$, then the decoding algorithm is given as in Figure 4.1.

Decoding algorithm

Step 1: Chinese remainder theorem

$p = \prod_{i=1}^{s} p_i$

For $i = 1$ to s

Using extended Euclidean algorithm to find p_i' by $\frac{p}{p_i} p_i' \equiv 1 \; mod \; p_i$

End

$\bar{r} = \sum_{i=1}^{s} \frac{p}{p_i} p_i' r_i \; mod \; p$

Step 2: Euclidean algorithm

$u_{-1} = p, u_0 = \bar{r}$

$v_{-1} = 0, v_0 = 1$

$i = -1$

While $u_i < \sqrt{p}$

$q_i = \lfloor u_{i-1}/u_i \rfloor$

$u_{i+1} = u_{i-1} - q_i u_i$

$v_{i+1} = v_{i-1} + q_i v_i$

$i++$

End

Rational solution:

$r = ((-1)^i u_i / v_i)$

Figure 4.1 Extended Chinese Remainder Theorem

c. How to identify the bound of the representation of a fraction number from the extended Chinese remainder theorem

Define$r = \frac{a}{b}$, $GCD(a, b) = 1$and $\delta = max(a, b)$, according to Dixon's theory that if δ satisfies $\delta \leq \lambda\sqrt{p}$ *(*$\lambda = 0.618\cdots$ is a root of $\lambda^2 + \lambda - 1 = 0$*)*, we can use the decoding algorithm to get the rational number back.

For example, we choose $r = \frac{1}{7}$ and $p_1 = 3, p_2 = 4, p_3 = 5$ to check the decoding process:

$$r \equiv 1 \bmod 3$$

$$r \equiv 3 \bmod 4$$

$$r \equiv 3 \bmod 5$$

Step 1:

Using the Chinese remainder theorem, we get,

$$p = 60, \overline{r} = 43$$

Step 2:

By the Euclidean algorithm, we get,

$$u_{-1} = 60, v_{-1} = 0$$

$$u_0 = 43, v_0 = 1$$

$$u_1 = 17, v_1 = 1$$

$$u_2 = 9, v_2 = 3$$

$$u_3 = 8, v_3 = 4$$

$$u_4 = 1, v_4 = 7$$

The rational solution is,

$$r = \frac{1}{7}$$

4.1.2. Implementation of the Extended Chinese Remainder Theorem with P-adic Arithmetic

By the nature of the extended Chinese remainder theorem, it can be implemented on parallel computers. The idea can be demonstrated as follows:

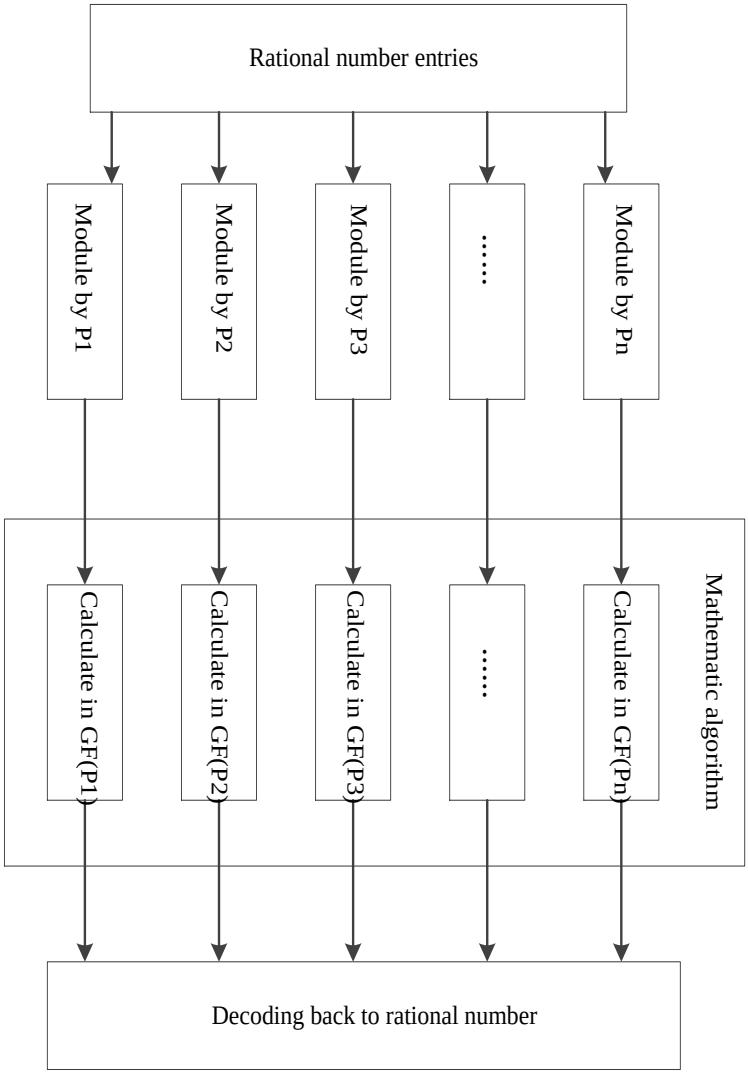


Figure 4.2 Extend CRT Parallel Implementation Chart

But in practice, there is a disadvantage of direct application. For a rational number $\frac{b}{a}$ and a prime p, if a and p are not relatively prime, we cannot get the result of $\frac{b}{a} \bmod p$. The way to solve this problem is to combine Hensel code calculation systems with the extended Chinese remainder theorem.

4.1.3. Combining P-adic Arithmetic with the Extended Chinese Remainder Theorem

P-adic arithmetic can be combined with the extended Chinese remainder theorem to do exact computing. It was called multiple *P*-adic algorithm [23]. In each $GF(p_i)$ we can use finite *P*-adic sequence to do calculation, the flow chart is the following:

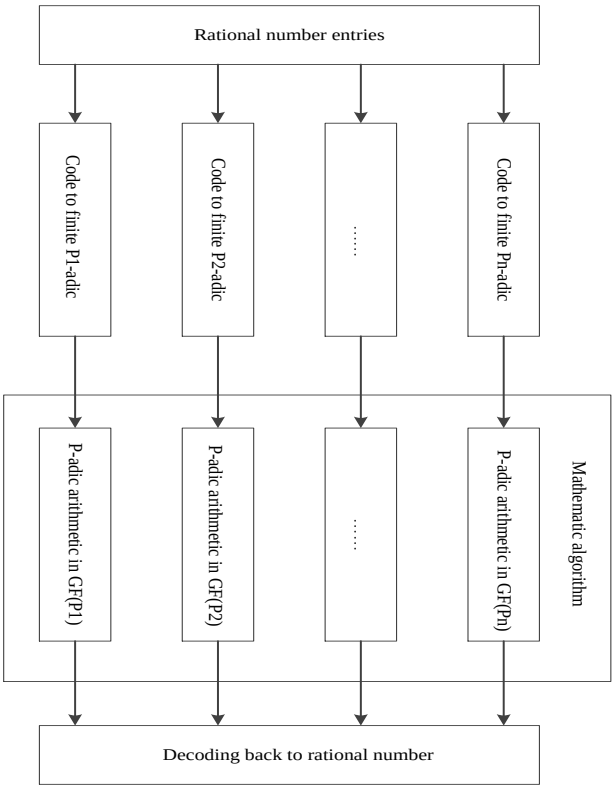


Figure 4.3 Extended CRT combined with P-adic arithmetic for parallel implementation

The decoding process:

If $x \sim \{x_1, x_2, \cdots x_s\}$, x_i is the Hensel code *P*-adic sequence with $x_i \sim \{a_{i0}, a_{i1}, \cdots, a_{in}; \text{point position}\}$ respect to prime set $\{p_1, p_2, \cdots, p_s\}$.

The residue representation $r \sim \{r_1, r_2, \cdots, r_s\}$ can be given as:

$$r_i = p^{point\ position} \sum_{j=0}^{n} a_{ij} p^j$$

where $p \sim \{p_1^n, p_2^n, \cdots, p_s^n\}$.

For example, if we choose the prime set as {2147483647, 2147483629, 2147483587} (the largest prime numbers smaller than square root of 2 to 64 power, 64-bit CPU architecture, $p_i \leq 2147483647$), we wish to obtain the reflexive general inverse of matrix *A*, given in the following example. For each *GF*(p) calculation, we choose the *P*-adic length as 2. The computation process is the following:

The entry rational matrix,

$$A = \begin{bmatrix} 1 & 2 \\ 1/3 & 1/4 \\ 5 & 6 \end{bmatrix}$$

After modulo operations by p_1, p_2, p_3, we have the following *P*-adic matrices,

$p_1 = 2147483647$,

$$\mathrm{A_{P_1}} = \begin{bmatrix} .1,0 & .2,0 \\ .1431655765,\ 1431655764 & .536870912, 1610612735 \\ .5,0 & .6,0 \end{bmatrix}$$

$p_2 = 2147483629$

$$\mathrm{A}_{\mathrm{P}_2} = \begin{bmatrix} .1,0 & .2,0 \\ .1431655753,1431655752 & .1610612722,1610612721 \\ .5,0 & .6,0 \end{bmatrix}$$

$p_3 = 2147483587$

$$\mathrm{A}_{\mathrm{P}_3} = \begin{bmatrix} .1,0 & .2,0 \\ .1431655725,1431655724 & .536870897,1610612690 \\ .5,0 & .6,0 \end{bmatrix}$$

Parallel calculation of each p_i under *P*-adic arithmetic to get the reflexive general inverse, the results:

$p_1 = 2147483647$

$$g - Inverse(A)_{P_1}$$

$$= \begin{bmatrix} .1717986917,858993458 & .1288490193,1717986917 & .0,0 \\ .1288490189,1717986917 & .429496727,1288490188 & .0,0 \end{bmatrix}$$

$p_2 = 2147483629$

$$g - Inverse(A)_{P_2} = \begin{bmatrix} .858993451,1288490177 & .1717986908,429496725 & .0,0 \\ .1717986904,429496725 & .1288490175,858993451 & .0,0 \end{bmatrix}$$

$p_3 = 2147483587$

$$g - Inverse(A)_{P_3}$$

$$= \begin{bmatrix} .1717986869,858993434 & .1288490157,1717986869 & .0,0 \\ .1288490153,1717986869 & .429496715,1288490152 & .0,0 \end{bmatrix}$$

Decoding from the extended Chinese remainder theorem is the following:

$$g - Inverse(A) = \begin{bmatrix} -3/5 & 24/5 & 0 \\ 4/5 & -12/5 & 0 \end{bmatrix}.$$

4.1.4. Practical Considerations for the Implementation of Multiple P-adic Algorithm

4.1.4.1. Advantages of multiple modulus arithmetic

There are three advantages of multiple *P*-adic algorithm as stated below.

a. Avoid the denominator problem

For rational number $\frac{b}{a}$ and prime p, if a and p are not relatively prime, we cannot calculate $\frac{b}{a} \bmod p$. Because p is a prime, if a and p are not relatively prime, $a = xp^y, x, y \in N$. We can still get the finite *P*-adic sequence of $\frac{b}{a}$, just the point position will be equal to y.

b. Increase the representation range

With $\{p_1, p_2, \cdots, p_s\}$, $p = \prod_{i=1}^{s} p_i$, for multiple module arithmetic, the bound for the representation of denominator and/or numerator will be $\lambda\sqrt{p}$ ($\lambda = 0.618\cdots$ is a root of $\lambda^2 + \lambda - 1 = 0$). While for multiple *P*-adic algorithm with each *P*-adic length is r, the bound will be $\lambda\sqrt{p'}$, $p' = \prod_{i=1}^{s} p_i^r$.

c. Parallel data structure

One of the important issues of finite *P*-adic arithmetic is to choose the *P*-adic sequence length r. If the initial r is not long enough, Hensel code overflow will happen [6]. The *P*-

adic sequence length r needs to be increased and the calculated results have to be discarded. On the other hand, for the multiple *P*-adic algorithm, when overflow happens, the calculated results can be kept. One should merely choose another prime p_i to continue the calculation, then combine the previously calculated results to convert back to the rational number by the extended Chinese remainder theorem.

By the "natural" structure of the extended Chinese remainder theorem, multiple *P*-adic arithmetic can be realized through parallel computation.

4.1.4.2. Choosing a prime

How to choose the prime set $\{p_1, p_2, \cdots, p_s\}$? According to the theory, for a fixed s value, the larger p_i you choose, the larger the bound that will result. But for computer architectures with 32 bit or 64 bit CPUs, when using the existing integer classes, the largest p_i should be chosen with respect to 46337 or 2147483647 to assure overflow protection [16]. This means that for a 32-bit CPU architecture, $p_i \leq 46337$, while for a 64-bit CPU architecture, $p_i \leq 2147483647$.

4.1.4.3. Parallel programming

The modern computer architecture utilizes multiple cores in the CPU. The parallel tasking design can significantly improve the efficiency of any computation. The multiple *P*-adic arithmetic has the natural property to realize parallel computation. The programming design can be described by the flowing flow chart:

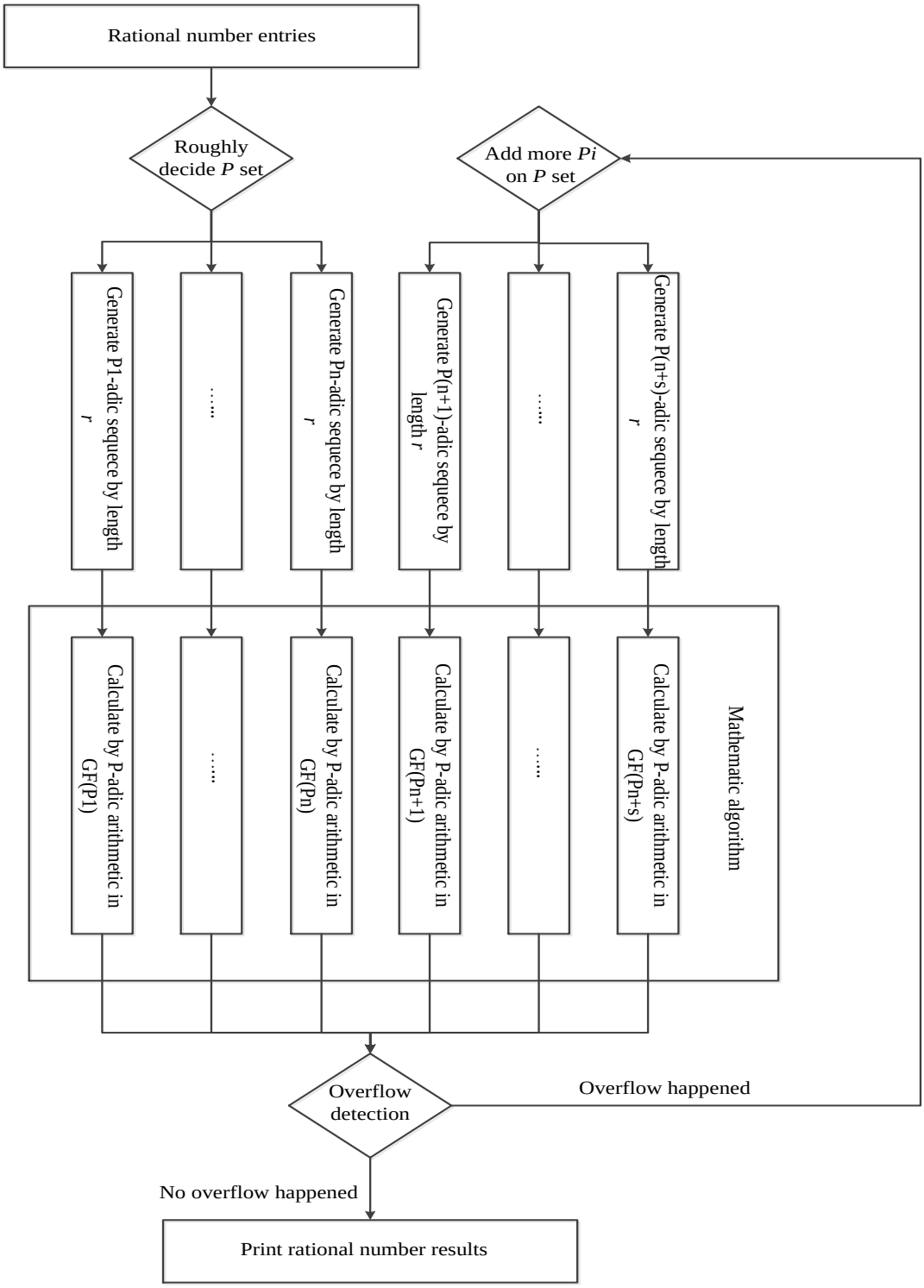


Figure 4.4. Multiple P-adic Arithmetic Implementation Flow Chart

The number of tasks, which will be the same as s from $\{p_1, p_2, \cdots, p_s\}$, can be chosen with respect to the number of CPU cores to improve the efficiency.

4.2. The Main Properties of Multiple P-Adic Data Type

4.2.1. Error-free Computing in Rational Number Field

Each rational number is represented by a finite sequence of integers. The integers' values are the module results of prime numbers, which can be chosen by developers and also depend on the CPU architectures. The arithmetic calculation process is in a rational number field, thus there will be no truncation error. The arithmetic is transformed to integer arithmetic and module operations.

The structure of the data type can be explained as the following:

$$\frac{b}{a} : \underbrace{\boxed{Integer\ 00}\boxed{Integer\ 01}\cdots\boxed{Integer\ 0k}}_{module\ results\ of\ P_0} \cdots \underbrace{\boxed{Integer\ n0}\boxed{Integer\ n1}\cdots\boxed{Integer\ nk}}_{module\ results\ of\ P_n}$$

For example, form prime number set $[257, 251, 241]$, then,

$$\frac{1}{10234567} : \underbrace{\boxed{179}\boxed{235}}_{module\ results\ of\ 257} \quad \underbrace{\boxed{6}\boxed{193}}_{module\ results\ of\ 251} \quad \underbrace{\boxed{229}\boxed{114}}_{module\ results\ of\ 241}$$

The integer sequence for $\frac{1}{10234567}$ is:

$$179, 235;\ 6, 193;\ 229, 114.$$

The size of the prime number set and the length for the integer sequence can be self-defined. The size of the prime number set can affect the efficiency of parallel computing. The detailed explanation will be introduced in the natural parallel ability section. Because

the arithmetic operation is in a rational number field as integers, there will be no truncation error. For example of the calculation of $\frac{1}{2}+\frac{1}{3}$ with the prime set[257,251,241],

$\frac{1}{2}$: 129,128; 126,125; 121, 120

$\frac{1}{3}$: 86, 171; 84,167; 161, 160

$$\begin{array}{r} 129 \quad 128 \quad 126 \quad 125 \quad 121 \quad 120 \\ +\ \underline{86 \quad 171 \quad 84 \quad 167 \quad 161 \quad 160} \\ 215 \quad 42 \quad 210 \quad 41 \quad 41 \quad 40 \end{array}$$

The sequence: 215,42; 210,41; 41,40 is transformed to $\frac{5}{6}$

4.2.2. Integer Calculations Taking Full Use of Computer Architecture

Usually, the rational calculation is using arbitrary length integers to represent the numerator and denominator. For example, we use one character (1 byte) to represent each digit of an integer and then link the data structure to realize the arbitrary length of the integer. For example, when calculating $1234567+7654321$

$$\begin{array}{r} 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \\ +\ \underline{7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1} \\ 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \quad 8 \end{array}$$

Including the carry-out operations, there will be 16 possible character operations. While using the Multiple *P*-adic Data Type, and choosing [46337] as the prime set, the calculation process will be

$$\begin{array}{r} 29805 \quad 26 \quad 0 \\ +\ \underline{8716 \quad 165 \quad 0} \\ 38521 \quad 191 \quad 0 \end{array}$$

There are only 3 additions and 3 module operations.

For rational operation, the numerator and denominator will cost more due to the reasons shown below:

Addition process:

$$\frac{a_1}{b_1} + \frac{a_2}{b_2} = \frac{a_1 b_2 + a_2 b_1}{b_1 b_2}$$

$$= \frac{(a_1 b_2 + a_2 b_1)/GCD((a_1 b_2 + a_2 b_1), b_1 b_2)}{b_1 b_2 / GCD((a_1 b_2 + a_2 b_1), b_1 b_2)}$$

Multiplication process:

$$\frac{a_1}{b_1} \times \frac{a_2}{b_2} = \frac{a_1 a_2}{b_1 b_2} = \frac{a_1 a_2 / GCD(a_1 a_2, b_1 b_2)}{b_1 b_2 / GCD(a_1 a_2, b_1 b_2)}$$

During these calculation processes, $GCD(numerator, denominator)$ must be found and extra calculation steps will be needed. However, for Multiple P-adic Data Type there will be no difference between fractions and integers.

The Multiple *P*-adic Data Type can be easily implemented on 32 and 64-bit platforms. The only difference is to choose the right prime number set. On the 32-bit platform, the maximum prime is $P = 46337$, the largest prime numbers smaller or equal to 46337 can be used, while on the 64-bit platform, the maximum prime will be $P = 2147483647$, the largest prime numbers smaller or equal to 2147483647 can be used.

4.2.3. Natural Parallel Structure Taking Full Use of Multi-core System

According to the features of the Multiple *P*-adic Data Type, parallel computing can be implemented on any algorithm with basic arithmetic operations and do not depend on the specific algorithm. The parallel structure depends on the size of the chosen prime set. For example, we calculate $(\frac{1}{17}-1)\times\frac{1}{2}$ with prime set$[257, 251, 241]$,

$$\frac{1}{17}: 121,60; 192,14; 156,212$$

$$1: 1,0; 1,0; 1,0$$

$$\frac{1}{2}: 129,128; 126,125; 121,120$$

$$\underbrace{\begin{array}{|l|}\hline 257\ Line\ Process \\ \begin{array}{rr} & 121\quad 60 \\ - & \underline{1\quad 0} \\ & 120\quad 60 \\ \times & \underline{129\quad 128} \\ & 60\quad 30 \end{array} \\ \hline\end{array}\quad \begin{array}{|l|}\hline 251\ Line\ Process \\ \begin{array}{rr} & 192\quad 14 \\ - & \underline{1\quad 0} \\ & 191\quad 14 \\ \times & \underline{126\quad 125} \\ & 221\quad 132 \end{array} \\ \hline\end{array}\quad \begin{array}{|l|}\hline 241\ Line\ Process \\ \begin{array}{rr} & 156\quad 212 \\ - & \underline{1\quad 0} \\ & 155\quad 212 \\ \times & \underline{121\quad 120} \\ & 198\quad 226 \end{array} \\ \hline\end{array}}_{3\ Separated\ Process}$$

After the 3 separated computing processes, we can get the final result: $-\frac{8}{17}: 60,30; 221,132; 198,226$.

Most of the linear processes can directly use this data type to realize parallel computing without modification at the mathematical algorithm level. We have implemented this data type to calculate matrix inverse, Moore-Penrose inverse (General Inverse) and e^{At}.

4.2.4. Easy for Task Allocation in Cloud Environment

Using the Multiple *P*-adic Data Type, the total work load is homogeneously allocated into small parts which is as many as the size of the prime set. It will be easier for making task allocations in a cloud environment. Furthermore in the symbolic (numerator-denominator) rational number calculation process, as the size of the arbitrary length number grows, the memory cost will increase quickly, while the Multiple *P*-adic Data Type will not have that kind of problem. The memory cost for each key will not grow during the calculation process. It is easy to estimate the memory cost before the calculation.

4.2.5. Practical Considerations in a Cloud Environment

Using Multiple *P*-adic Data Type, each module is independent of others, so that each can be computed on a different cluster node and can be done not necessarily at the same time. At each cluster node, a parallel algorithm can also be implemented during the matrix calculation process on multi CPU cores. If the matrix size is too large, the block algorithm can be freely implemented in the calculation process. The efficient formula for calculation time is the following:

$$\begin{aligned} &Total\ Calculation\ Time\ Cost \\ &\quad = Basic\ Integer\ Calculation\ Time\ Cost \\ &\quad \times \left\lceil \frac{Calculation\ Complexity}{Number\ of\ CPU\ Used} \right\rceil \end{aligned}$$

Basic Integer Calculation Time means the time cost on one node calculation by one module prime (64bits integer type or 32bits integer type). Calculation Complexity means

the necessary prime set length with no overflow happening. Of course, the total time cost must also include the communication cost between nodes and the host at the beginning and the end of each node calculation.

The implementation process is the following:

1) Analyze the work load. According to the matrix size, the complexity of the number and the matrix calculation algorithms, the size of prime set *s* and the length of the *P*-adic sequence *r* will be decided. For a specific matrix transform, there will be a specific algorithm chosen or created for the work load evaluation. For example, to calculate $Ax = b$, Hadamard's inequality will be used: $2max(n^{\frac{n}{2}}M(A)^n, n(n-1)^{\frac{n-1}{2}}M(A)^{n-1}M(b)) \leq \lambda\sqrt{\prod_{i=1}^{s} p_i^r}$, where $\lambda = 0.618\cdots$ is a root of $\lambda^2 + \lambda - 1 = 0$, $M(X)$ means the largest value of denominator or numerator among elements in matrix *X*, *s* means the number of cluster node assigned, and *r* usually can represent the calculation efficiency. The smaller *r* means less calculation time and less memory usage for a cluster node, while the smaller *r* requires larger *s*, which means to assign more cluster nodes.

2) Work load *separation.* The original matrix data and a specific prime from a prime set will be sent to different cluster nodes. In each node, the original matrix elements will be module by the prime and generate *P*-adic sequence with length equals to *r*. Then the matrix transformation will be calculated. During this process, parallel and block algorithms can be freely implemented.

3) Generate the *final result.* All the temporary results will be collected from various nodes in the cloud by the master (host). The final rational result will be generated on the host machine. The Hensel code overflow detection will be used for verification. If

overflow does not happen, we get the final result. Otherwise, keep the temporary results and choose a different prime set, then go to step *1)*.

A. Compare with the MATLAB Symbolic Toolbox

This new data type can significantly shorten the calculation time by using more CPU-cores. We have compared the calculation time for matrix inverses with the symbolic toolbox in MATLAB, the experimental results are given in Figure 4.2.1. The computer used is the Intel(R) Core(TM) i7-2600 CPU @ 3.40 GHz for the experiment. This CPU has 8 CPU cores.

The following results (Figure 4.2.1) show the calculation time of the inverse of $Hilbert\ matrix \times Hilbert\ matrix$.

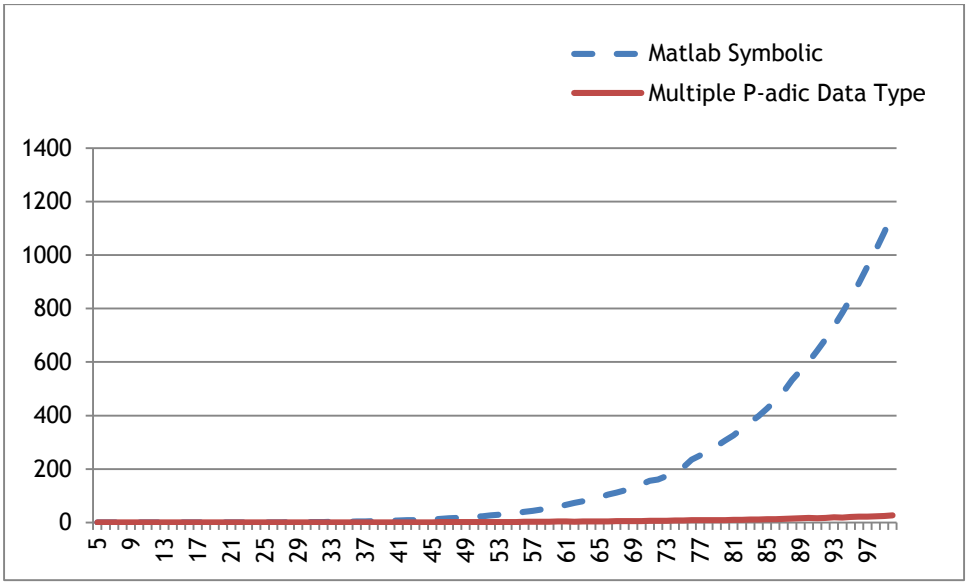


Figure 4.5 Vertical axis: Calculation time (seconds); Horizontal axis: Matrix size (N x N)

The MATLAB code is following:

```
for n = 5:100
        A = hilb(n);
        B = sym(A);
        B = B * B;
        n
        tic
        inv(B);
        t1(n) = toc;
        toc
end
```

During the calculation process, Matlab uses only one core of CPU to do the calculation, while our data type takes full use of all the 8 cores.

B. Compare with the Mathematic Symbolic Toolbox

We have compared the calculation time for Moore-Penrose inverses with symbolic toolbox in Mathematica 8, the experimental results are given in Figure 4.2.2 and Figure 4.2.3.

Intel(R) Core(TM) i7-2600 CPU @ 3.40 GHz is used to do the experiment. This CPU has 8 CPU cores. The following results (Fig. 3) show the calculation time of the inverse of $Hilbert\ matrix \times Hilbert\ matrix$.

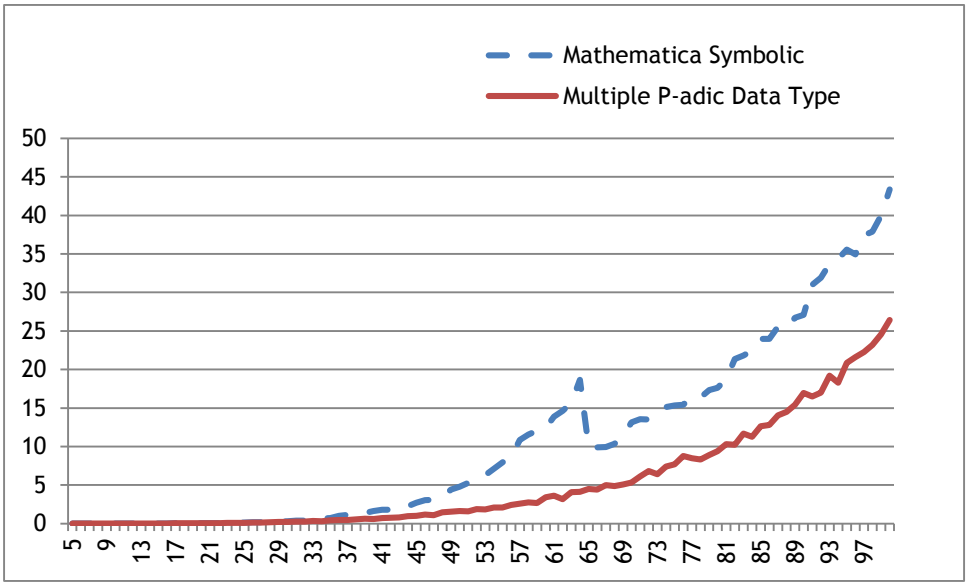


Figure 4.6 Vertical axis: calculation time (seconds); Horizontal axis: matrix size (N x N)

The Mathematica code is following:

```
Array[f,100];
For[i=5,i<105,i++,s =
HilbertMatrix[i]*HilbertMatrix[i];f[i-4] =
Timing[Inverse[s];][[1]];Print[f[i-4]]]
```

The following results (Figure 4.2.3) show the calculation time of the Moore-Penrose inverse of $Hilbert\ matrix \times Hilbert\ matrix$.

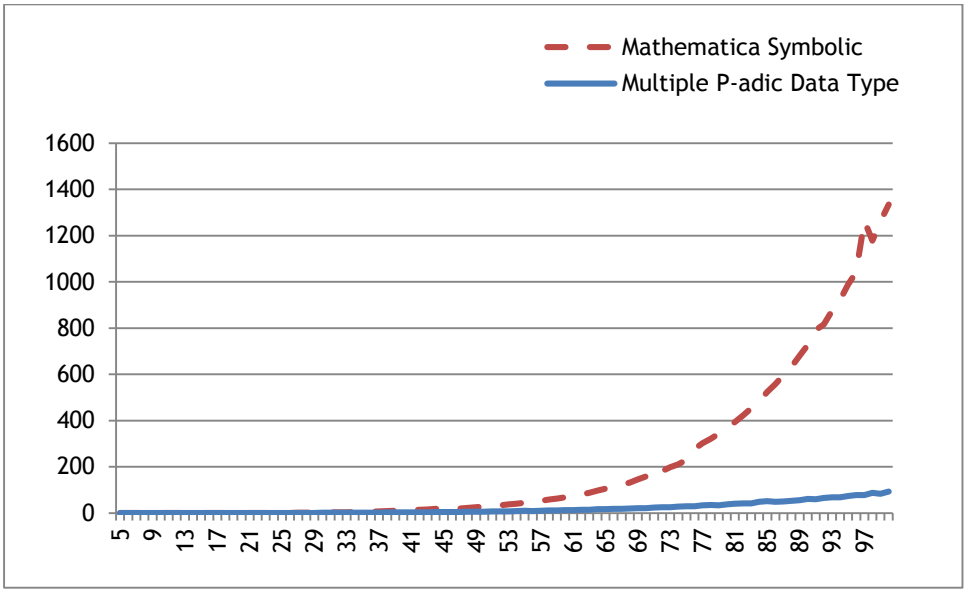


Figure 4.7 Vertical axis: calculation time (seconds); Horizontal axis: matrix size

The Mathematica code is following:

```
Array[f,100];
For[i=5,i<105,i++,s =
HilbertMatrix[i]*HilbertMatrix[i];f[i-4] =
Timing[PseudoInverse[s];][[1]];Print[f[i-4]]]
```

During the calculation process, Mathematica uses 4 cores of the CPU, while our data type takes full use of all the 8 cores of the CPU. If the input matrix is more complex, the advantage of this new data type is more obvious. This observation can be shown by

comparing Figure 4.6 and Figure 4.7 Calculating the Moore-Penrose Inverse is more complex than general matrix inverse.

4.3. Overflow Detection for Multiple P-adic Data Type

***Multiple P-adic data type overflow*:** for a rational number $\frac{a}{b}$, when it satisfies $|a, b| > \lambda\sqrt{p}$ *(*$\lambda = 0.618\cdots$ is a root of $\lambda^2 + \lambda - 1 = 0$*)*, the rational number which the Multiple *P*-adic data represents, cannot be uniquely recovered by the inverse transformation. In this situation, the overflow problem happens.

Notation: *Decoding(x,* p_i*, k)* will be used to donate the decoding of Multiple *P*-adic data sequence *x* into a rational number and the last *k* digits of p_i-adic sequence will be used as identification digits and these digits will not be used on the decoding process. *Decoding(x)* means to decode the full size of Multiple *P*-adic data to rational number with no verification part. *X* will be used to the donate matrix of Multiple P-adic data.

For example, we take a prime set [257, 251, 241]

$$x = (131;\ \ 239;\ \ 133)$$

$$Decoding(x) = 1/1234$$

$$Decoding(x, 241, 1) = -209/122$$

$$Decoding(x, 251, 1) = -50/237$$

$$Decoding(x, 257, 1) = -49/25$$

$$X = \begin{bmatrix} (1;\ \ 1;\ \ 1) & (150;\ \ 21;\ \ 221) \\ (140;\ \ 100;\ \ 145) & (131;\ \ 239;\ \ 133) \end{bmatrix}$$

$$Decoding(X) = \begin{bmatrix} 1 & 1/12 \\ 1/123 & 1/1234 \end{bmatrix}$$

$$Decoding(X, 241, 1) = \begin{bmatrix} 1 & 1/12 \\ 1/123 & -209/122 \end{bmatrix}$$

$$Decoding(X, 251, 1) = \begin{bmatrix} 1 & 1/12 \\ 1/123 & -50/237 \end{bmatrix}$$

$$Decoding(X, 257, 1) = \begin{bmatrix} 1 & 1/12 \\ 1/123 & -49/25 \end{bmatrix}$$

No matter the Hensel code or the Multiple *P*-adic data type, if the range of the calculation results cannot be predicted, it is hard to avoid an overflow problem completely. Also there is no way to guarantee 100 percent of finding out whether the overflow problem is happening, it simply depends on the prime set and data sequences. The method given in the *Hensel code overflow detection* or the *Multiple P-adic overflow detection* can only give extremely high likelihood of detection, say 99.999999% or more.

The reason for the overflow problem happening is a possibility of a rational number having the same digits on the first part of the sequence with another different rational number. For example, the rational number $\frac{a}{b}$, the data sequence is $(a_1, a_2, \cdots a_k, \cdots a_s)$ while for rational number $\frac{d}{c}$, the data sequence is $(a_1, a_2, \cdots a_k, b_1 \cdots b_{s-k})$. From the bound condition, the sufficient digits for representing $\frac{a}{b}$ is s and the sufficient digits for representing $\frac{d}{c}$ is $(k-1)$ $where$ $((k-1) < s)$. If you choose k digits as a calculation sequence length and 1 digits as verification part, you may identify $\frac{d}{c}$ as the resulting rational number, but can also possible be $\frac{a}{b}$.

4.3.1. Overflow Detection Method

The rate of making mistakes can be decreased by choosing a larger prime for the prime set or increasing the length of the verification part. For Multiple P-adic data type, we can also increase the verification times to decrease the rate of making mistakes. For prime set p_i of Multiple P-adic data type, with the same length of verification part, each time chooses a different set of p_i-adic sequences to supply verification parts. The decoding results will be possible different when the overflow situation happened.

Based on the property above, the new method to detect the Multiple P-adic data type overflow detection problem is similar to the Hensel code overflow detection, only using a prime p_i and data sequences. In this method, we randomly pick up a prime p_i and choose *k* digits of the p_i-adic as the verification part. For Multiple P-adic data *x*, if *Decoding(x,* p_i *, j)* $\neq$ *Decoding(x)* $(0 \leq i \leq s, 0 \leq j \leq k)$, then Multiple P-adic data overflow happened.

For example of Multiple P-adic data *x*, where

$$x = (a_{00}, a_{01} \cdots ; a_{10}, a_{11} \cdots ; \cdots ; a_{s0}, a_{s1} \cdots)$$

Will be treated as

$$x = (a_{00}, a_{01} \cdots ; a_{10}, a_{11} \cdots ; \cdots ;$$

$$a_{i0}, a_{i1} \cdots \underbrace{a_{i(k-j)} \cdots a_{ik}}_{verification\ part} ; \cdots ; a_{s0}, a_{s1} \cdots)$$

Overflow happened, if:

Decoding(x, p_i, j)≠ Decoding(x)

Overflow did not happen, if:

Decoding(x, p_i, j)= Decoding(x)

For example, by taking prime set [257, 251, 241]

$$x = (150;\ \ 21;\ \ 221)$$

$$y = (91;\ \ 173;\ \ 85)$$

For x,

Decoding(x) = 1/12

Decoding(x, 241, 1) = 1/12

Decoding(x, 251, 1) = 1/12

Decoding(x, 257, 1) = 1/12

For y,

Decoding(y) = 14/3

Decoding(y, 241, 1) = -209/9

Decoding(y, 251, 1) = -43/8

Decoding(y, 257, 1) = 14/3

By this method, we take *x* as 1/12. For *y* we experienced an overflow problem.

Experiment 4.3.1. Each time, the prime set $\{p_1, \cdots p_s\}$ is fixed, p_i is a continuous series of prime numbers. For each prime set, we randomly generate rational number$\frac{a}{b}(GCD(a,b)=1), |a,b| \leq 10^{30}$. The size of prime set is 3. When *Decoding(x, p_i, 1)= Decoding(x)*, but *Decoding(x)*$\neq \frac{a}{b}$, it is recorded as one error.

In Table 4.3.1, the $\boldsymbol{P_0}$ column represents the value of the first prime in prime set. The **Exp Nums** column are the numbers of generating random fractions which caused 3000 errors. **Total Errors** mean the amount of errors during the **Exp Nums** of experiments. The digits ***i*** column is the total number of experiments in which the *ith* decoding process generated errors. After finishing all the decoding processes in the experiment. ***i*** equals to 1, which means that each experiment in all the decoding processes generate error results, which will make the overflow detection method fail.

Take first row as example, $\boldsymbol{P_0} = 46337$ means the prime set is {46337, 46327, 46309} which is a decreasing continuous series of prime numbers. **Exp Nums** = 33576299 means after generating 33576299 random samples, there are 3000 of them occurring overflow problem. **3** = 2305 means there are 2305 samples from the 3000 overflow samples have the property following:

Decoding(samples, 46337, 1) ≠ Decoding(samples, 46327, 1) ≠ Decoding(samples, 46309, 1) but one of them equals to *Decoding(samples)*

2 = 54 means there are 54 samples from the 3000 overflow samples have the property following:

Two of the Decoding(samples, p_i, 1) equals to *Decoding(samples)*, but not equal to the third one.

1 = 641 means there are 641 samples from the 3000 overflow samples have the property following:

Decoding(samples, 46337, 1) = Decoding(samples, 46327, 1) = Decoding(samples, 46309, 1) = Decoding(samples)

P_0	Exp Nums	3	2	1	Total Errors
46337	33576299	2305	54	641	3000
41011	29918912	2285	55	660	3000
35603	26366287	2266	59	675	3000
30493	22655982	2233	66	701	3000
25309	18367716	2227	66	707	3000
20287	14477520	2278	60	662	3000
15331	11435596	2305	60	635	3000
10597	7929831	2289	62	649	3000
6073	4320387	2272	84	644	3000
1907	1347864	2306	59	635	3000

Table 4.1. Prime Set Length equals to 3

From Table 4.1, with the value of the prime set increases, the number of experiments decreases, which means that the error rate increases. The error rate means the possibility of the detection method fails. Using Table 1, we generate the following graph shown in Fig. 4.8. Horizontal axis is the error rate. The vertical axis means the first prime P_0 of the

prime set. Comparing once means during the comparing process (*Decoding(x, p_i, 1)= Decoding(x)*), *i* randomly chooses one value. Comparing twice means during the comparing process, *i* randomly choose the value twice. Full comparing means during the comparing process, *i* chooses all the possible values.

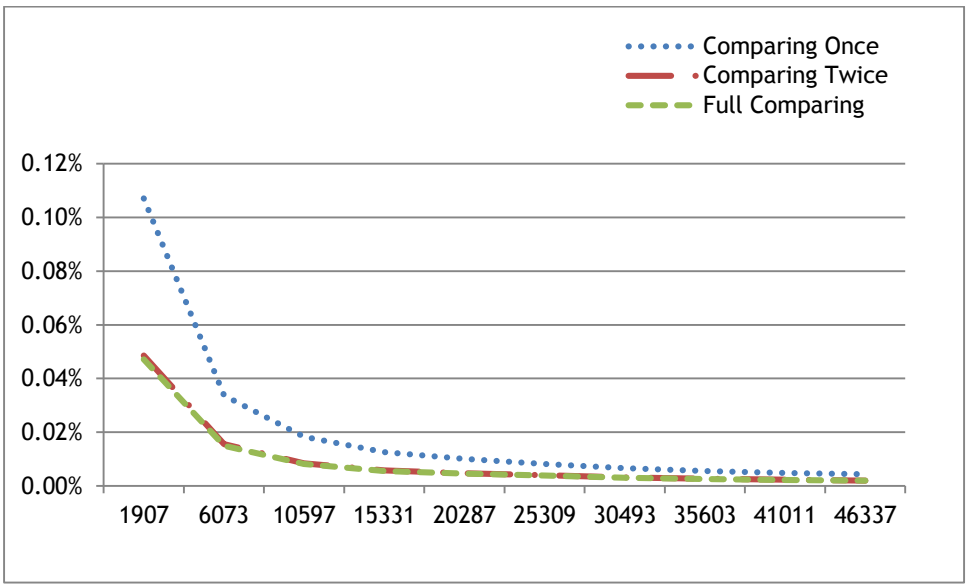


Figure 4.8 Error Percentage for Three Comparing Method; Vertical axis: error percentage; Horizontal axis: the first prime of prime sett

From the graph, we can find the error rate decreasing with the comparing times increasing. A large prime set value also can decrease the error rate. The Comparing Twice line is almost overlapping the Full Comparing line. During the practical implementation process, comparing twice is a good choice to balance the error rate and efficiency.

4.3.2. Practical Consideration

We compare the new method to the old method to find a more accurate ways to detect the overflows. During the practical implementation, the following ways also should be considered,

a. *Improve the value of the prime set.* The largest prime for 32-bits and 64-bits are 46337 and 2147483647. When choose prime close to 2147483247, even verification part is 1, the mistake rate is significantly low.

b. *Increasing the length of the verification part.* When choose prime close to 46337, the verification part is better to use 5 or more.

c. *Increasing the size of the prime set.* On this way, increase the *P*-adic sequences for each prime in prime set also works. Even we cannot exactly predict the ranges of the final results, a general prediction should be done for choosing the property size of a prime set and digits of the *P*-adic sequence.

d. *Randomly choose two different prime for Comparing Twice.* When the size of prime set is small, Full Comparing can be applied which is the best way to avoid detection mistakes. While consider the balance for efficiency, Comparing Twice is a better choice.

When an overflow problem happens, the results can be kept as temporary results. Another prime set which includes prime with different values from the primes in original prime set will be chosen and do the calculation again. The new results will be combined with temporary results to be detected. If the combined results pass the detection, the

results will be recorded as final results, otherwise repeat the last 2 steps. Further research will be done concerning how to choose a new prime set. The implementation process will be taken as Figure 4.1.3.

For example, choose prime set [17, 13, 11] and *P*-adic sequence is 2 to do matrix inverse for

$$x = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{bmatrix}$$

The calculation process is following:

$$\mathrm{x} \xrightarrow{Multiple\ P-adic\ code}$$

$$\begin{bmatrix} (1,0;1,0;1,0) & (9,8;7,6;6,5) & (6,11;9,8;4,7) & (13,12;10,9;3,8) \\ (9,8;7,6;6,5) & (6,11;9,8;4,7) & (13,12;10,9;3,8) & (7,3;8,2;9,8) \\ (6,11;9,8;4,7) & (13,12;10,9;3,8) & (7,3;8,2;9,8) & (3,14;11,10;2,9) \\ (13,12;10,9;3,8) & (7,3;8,2;9,8) & (3,14;11,10;2,9) & (5,7;2,11;8,4) \end{bmatrix}$$

$$\xrightarrow{Inverse} y$$

$$\begin{bmatrix} (16,0;3,1;5,1) & (16,9;10,3;1,0) & (2,14;6,5;9,10) & (13,8;3,2;3,9) \\ (16,9;10,3;1,0) & (10,2;4,1;1,10) & (3,11;4,0;6,7) & (14,13;3,12;8,9) \\ (2,14;6,5;9,10) & (3,11;4,0;6,7) & (3,7;6,4;1,6) & (16,7;12,1;2,3) \\ (13,8;3,2;3,9) & (14,13;3,12;8,9) & (16,7;12,1;2,3) & (12,11;5,7;6,1) \end{bmatrix}$$

We use prime 11 and $k = 2$ as verification part, the verification results is following:

$$\begin{bmatrix} Pass & Pass & Failed & Pass \\ Pass & Failed & Failed & Failed \\ Failed & Failed & Failed & Failed \\ Pass & Failed & Failed & Failed \end{bmatrix}$$

So, we choose additional prime set[23,19]

$$x \xrightarrow{Multiple\ P-adic\ code}$$

$$\begin{bmatrix} (1,0;1,0) & (12,11;10,9) & (8,15;13,12) & (6,17;5,14) \\ (12,11;10,9) & (8,15;13,12) & (6,17;5,14) & (14,4;4,15) \\ (8,15;13,12) & (6,17;5,14) & (14,4;4,15) & (4,19;16,15) \\ (6,17;5,14) & (14,4;4,15) & (4,19;16,15) & (10,16;11,13) \end{bmatrix}$$

$$\xrightarrow{Inverse} y$$

$$\begin{bmatrix} (16,0;16,0) & (18,17;13,12) & (10,10;12,12) & (21,16;12,11) \\ (18,17;13,12) & (4,6;3,6) & (14,20;17,9) & (1,4;8,12) \\ (10,10;12,12) & (14,20;17,9) & (17,5;1,18) & (9,1;18,6) \\ (21,16;12,11) & (1,4;8,12) & (9,1;18,6) & (17,6;7,14) \end{bmatrix}$$

We combine the temporary and additional results to:

$$y_{11} = (16,0;\ \ 16,0;\ \ 16,0;\ \ 3,1;\ \ 5,1)$$

$$y_{12} = (18,17;\ \ 13,12;\ \ 16,9;\ \ 10,3;\ \ 1,0)$$

$$y_{13} = (10,10;\ \ 12,12;\ \ 2,14;\ \ 6,5;\ \ 9,10)$$

$$y_{14} = (21,16;\ \ 12,11;\ \ 13,8;\ \ 3,2;\ \ 3,9)$$

$$y_{21} = (18,17;\ \ 13,12;\ \ 16,9;\ \ 10,3;\ \ 1,0)$$

$$y_{22} = (4,6;\ \ 3,6;\ \ 10,2;\ \ 4,1;\ \ 1,10)$$

$$y_{23} = (14,20;\ \ 17,9;\ \ 3,11;\ \ 4,0;\ \ 6,7)$$

$$y_{24} = (1,4;\ \ 8,12;\ \ 14,13;\ \ 3,12;\ \ 8,9)$$

$$y_{31} = (10,10;\ \ 12,12;\ \ 2,14;\ \ 6,5;\ \ 9,10)$$

$$y_{32} = (14{,}20;\ \ 17{,}9;\ \ 3{,}11;\ \ 4{,}0;\ \ 6{,}7)$$

$$y_{33} = (17{,}5;\ \ 1{,}18;\ \ 3{,}7;\ \ 6{,}4;\ \ 1{,}6)$$

$$y_{34} = (9{,}1;\ \ 18{,}6;\ \ 16{,}7;\ \ 12{,}1;\ \ 2{,}3)$$

$$y_{41} = (21{,}16;\ \ 12{,}11;\ \ 13{,}8;\ \ 3{,}2;\ \ 3{,}9)$$

$$y_{42} = (1{,}4;\ \ 8{,}12;\ \ 14{,}13;\ \ 3{,}12;\ \ 8{,}9)$$

$$y_{43} = (9{,}1;\ \ 18{,}6;\ \ 16{,}7;\ \ 12{,}1;\ \ 2{,}3)$$

$$y_{44} = (17{,}6;\ \ 7{,}14;\ \ 12{,}11;\ \ 5{,}7;\ \ 6{,}1)$$

The prime set for combined results is [23, 19, 17, 13, 11] using prime 11 and $k = 2$ as a verification part, the verification results are:

$$\begin{bmatrix} Pass & Pass & Pass & Pass \\ Pass & Pass & Pass & Pass \\ Pass & Pass & Pass & Pass \\ Pass & Pass & Pass & Pass \end{bmatrix}$$

And the final results is

$$\begin{bmatrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{bmatrix}$$

Chapter 5

Implementation

5.1. Mathematics Background

5.1.1. Moore – Penrose Inverse

This algorithm is based on the Hermite theory [7], it is expressed as,

$$A^{+} = A^{t}(AA^{t}AA^{t})_{R}^{-}AA^{t}$$

A^{+}means the Moore-Penrose inverse of A (of order $m \times n$). M_{R}^{-} of $M = AA^{t}AA^{t}$ means the reflexive *g-inverse* of M.

5.1.2. Polynomial Method to Calculate e^{At}

5.1.2.1 Definition of e^{At}

$$e^{At} = I + tA + \frac{t^{2}A^{2}}{2!} + \cdots$$

According to the definition, if $A = W^{-1}CW$

$$e^{At} = W^{-1}IW + W^{-1}tCW + \frac{W^{-1}t^{2}C^{2}W}{2} + \cdots = W^{-1}e^{tC}W$$

5.1.2.2 Companion Matrix [43]

The companion matrix of the polynomial

$$c(z) = z^n - \sum_{k=0}^{n-1} c_k z^k$$

is defined as

$$C = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_0 & c_1 & c_2 & \cdots & c_{n-1} \end{bmatrix}$$

5.1.2.3. Poor-Man Method [11, 44]

Sjogren described the "Poor-Man" method in his Matlab code [44] based on Danilevskii algorithm (1937) in the book of Gregory and Young [11], which can compute the Frobenius form of a matrix over a field *M*. For any matrix $A \in M^{n\times n}$ there exists an invertible *W* over *M*, such that

$$W^{-1}CW = G = \begin{bmatrix} C_{g_1} & & & \\ & C_{g_2} & & \\ & & \ddots & \\ & & & C_{g_i} \end{bmatrix} \in M^{n\times n}$$

G is the Frobenius canonical form of *A*, also called the rational canonical form. Each diagonal block is the companion matrix with form as,

$$C_f = \begin{bmatrix} 0 & & & & -f_0 \\ 1 & 0 & & & -f_1 \\ & \ddots & \ddots & & \vdots \\ & & 1 & 0 & -f_{n-2} \\ & & & 1 & -f_{n-1} \end{bmatrix}$$

Algorithm description [11, 45]:

1. Transform a $n \times n$ matrix *A* into Lower Hessenberg form *H*, and get the transforming matrix *T*,

$$T^{-1}AT = H$$

2. Convert the lower Hessenberg matrix *H* to Frobenius form according to the formula of Wilkinson[4],

$$C^{-1}WC = F$$

3. Form a diagonal matrix *D* that is supposed to transform the matrix so that the sub-diagonal consists of *1s*,

$$D^{-1}FD = G$$

After the three transforming steps, we get the Frobenius canonical form *G*, invertible matrix *W* and its inverse matrixW^{-1}, for which$W^{-1} = D^{-1}C^{-1}T^{-1}$, *W=TCD***.** In some cases, this transformation does not result in a "block companion" matrix. There are some non-diagonal elements in G outside the blocks.

5.1.2.4. Polynomial Method

As in Mastascusa [46], Polynomial method is based on the Cayley-Hamilton theorem. This method will cost less in calculation, when the degree of series approximation of e^{Ct} is much higher than the rank of the matrix *C* [46].

Moler and Loan gave a short description of the method as the following [2]:

1. Find the characteristic polynomial of matrix *A*

$$c(z) = \det(zI - A) = z^n - \sum_{k=0}^{n-1} c_k z^k$$

2. According to the Cayley-Hamilton theorem $c(A) = 0$, hence

$$A^n = c_0 I + c_1 A + \cdots + c_{n-1} A^{n-1}$$

And it follows that any power of *A* can be expressed in terms of $I, A, \cdots A^{n-1}$:

$$A^k = \sum_{j=0}^{n-1} \beta_{kj} A^j$$

3. The e^{At} can be implied as the following:

$$e^{tA} = \sum_{k=0}^{\infty} \frac{t^k A^k}{k!} = \sum_{k=0}^{\infty} \frac{t^k}{k!} \left[\sum_{j=0}^{n-1} \beta_{kj} A^j \right] = \sum_{j=0}^{n-1} \left[\sum_{k=0}^{\infty} \beta_{kj} \frac{t^k}{k!} \right] A^j = \sum_{j=0}^{n-1} \alpha_j(t) A^j$$

5.2. Implementation of Multiple P-adic Arithmetic on Matrix Calculation

Our experiments were carried out on a typical laptop with Intel Core i5-2500 CPU as a parallel environment. The CPU has 4 cores for parallel processing.

Experiment 5.2.1. We generated random matrices with size from 3 by 3 to 40 by 40, each element $\frac{a}{b}$ satisfies $|a, b| \leq 20$. For Multiple *P*-adic arithmetic algorithm, $s = 12$ for $p \sim \{p_1, p_2, \cdots, p_s\}$ and for each p the sequence length is 5. While for *P*-adic arithmetic, the sequence is 60. For each matrix size, we generated 30 simples. Both algorithms are used to calculate the Moore-Penrose inverse. We use NTL [5] to represent larger integers for the experiments. The speed up is defined as:

$$\text{Speed-up Rate} = \frac{P-adic}{Multiple\ P-adic}$$

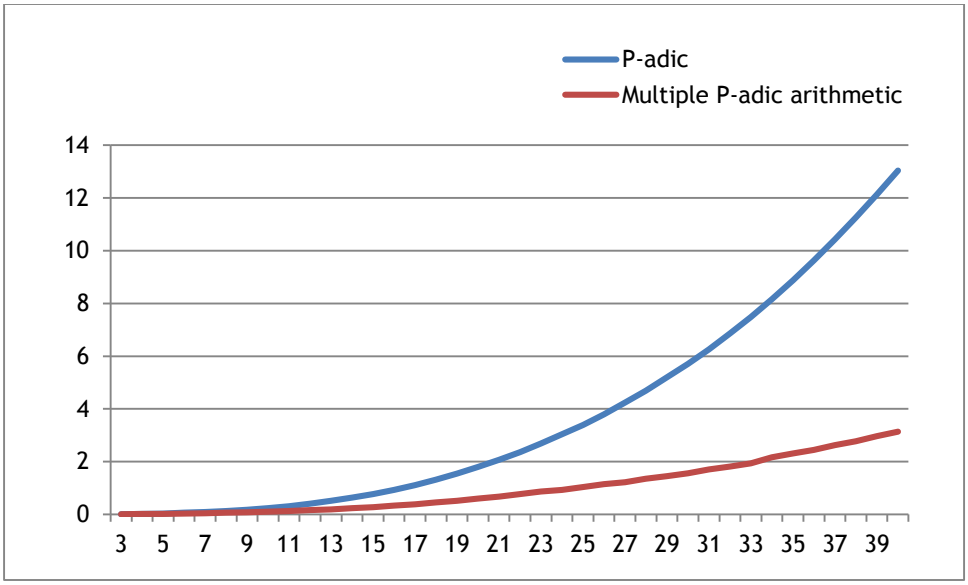


Figure 5.1 Moore-Penrose Inverse; Vertical axis: the average implementation time in second; Horizontal axis: the matrix size

Experiment 5.2.2. We generated random matrices with size from 3 by 3 to 40 by 40, each element $\frac{a}{b}$ satisfies $|a, b| \leq 20$. For Multiple *P*-adic arithmetic algorithm, $s = 12$ for $p \sim \{p_1, p_2, \cdots, p_s\}$ and for each p the sequence length is 5. While for *P*-adic arithmetic, the sequence is 60. For each matrix size, we generated 30 simples. Both algorithms are used to calculate $e^{At}, t = 1$ with 100 iterations.

From the above two experiments, we can find that on the 4 cores CPU (Intel Core i5-2500), the multiple *P*-adic arithmetic algorithm will speed up about 2 to 4 times based on the matrix sizes compared with that of direct *P*-adic arithmetic.

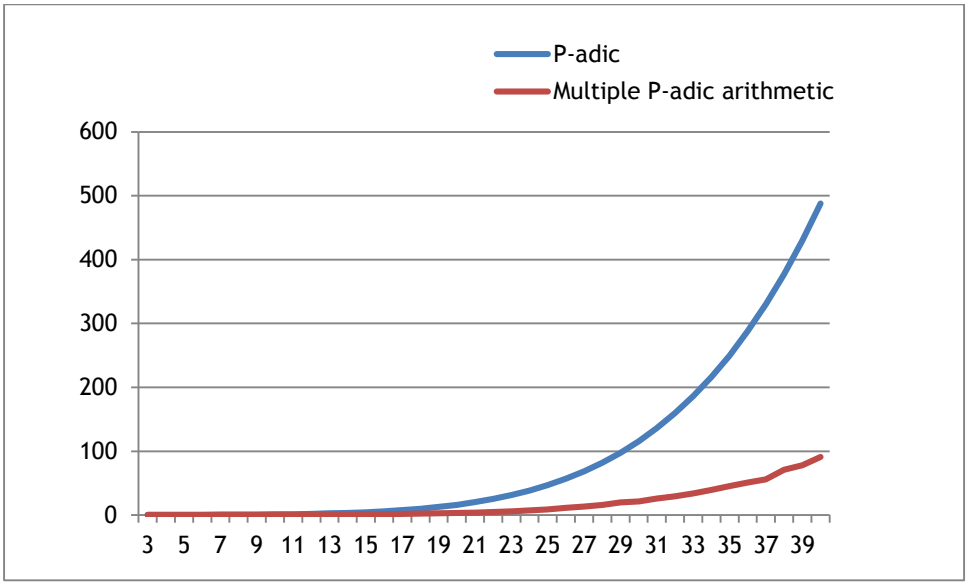


Figure 5.2 Polynomial method to calculate $\mathbf{e^{At}}$; Vertical axis: the average implementation time in second; Horizontal axis: the matrix size

Experiment 5.2.4. We generated random matrices with size from 3 by 3 to 40 by 40, each element $\frac{a}{b}$ satisfies $|a, b| \leq 20$. For the multiple *P*-adic arithmetic algorithm, $s \sim \{4, 5, 8, 12\}$ for $p \sim \{p_1, p_2, \cdots, p_s\}$ and for each p the sequence length is 5. While for *P*-adic arithmetic, the sequence is $\{20, 25, 40, 60\}$. For each matrix size, we generated 30 simples. Both algorithms are used to calculate the Moore-Penrose inverse.

We get the average of speed up rate ($\frac{P-adic}{Multiple\ P-adic}$) for each size *s* as shown in Figures 5.3 and 5.4.

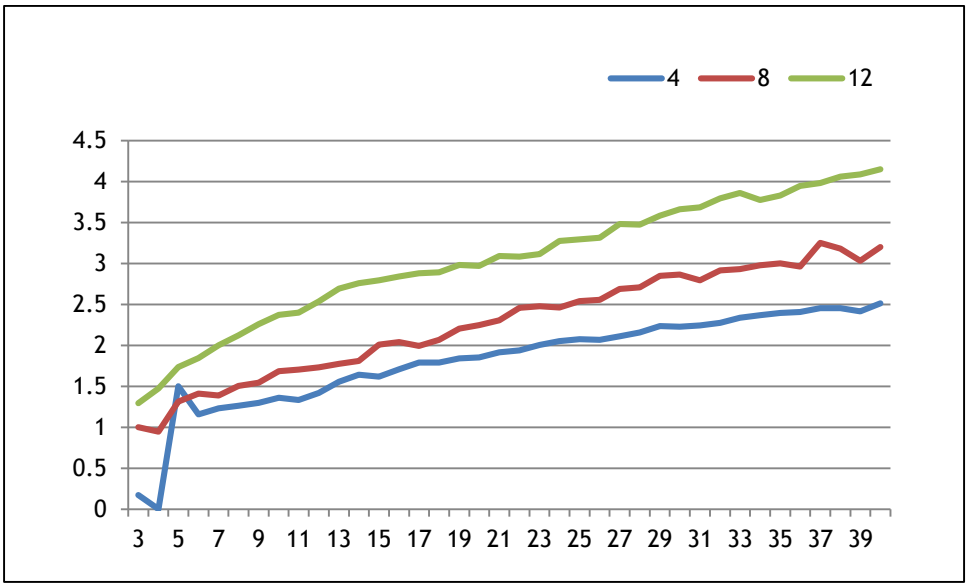


Figure 5.3 Speed up rate for s equal to 4, 8 and 12; vertical axis: speed up rate value; horizontal axis: the matrix size

From Figure 5.2.4, we can see that with the increase of the integer sequence length for multiple *P*-adic and *P*-adic sequences, we will have more advantage of the multiple *P*-adic arithmetic. The reason is that as the length increase, the time complexity for *P*-adic arithmetic is $O(n^2)$, while for Multiple *P*-adic arithmetic is $O(n)$.

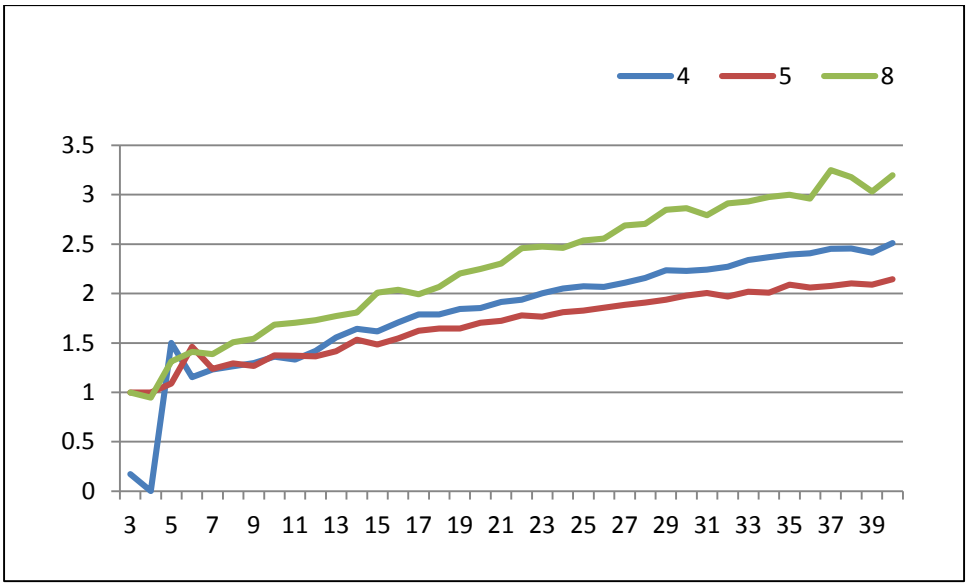


Figure 5.4 Speed up rate for s equal to 4, 5 and 8; Vertical axis: speed up rate value; Horizontal axis: the matrix size

The CPU architecture can be an important part of the speeding up. From Figure 5.2.4 and Figure 5.2.5, we can see that if the length is a multiple of the number of CPU cores, the speed up is outstanding; while when the length is not a divisible number by CPU cores, such as 5 for a CPU with four cores, the speed-up will be poor. Also, as the matrix sizes grow, the speed-up factor becomes even more significant.

5.3 Using Multiple *P*-adic Data Type in the Security Field

The operation process implemented with the multiple *P*-adic data type can be separated into several parallel sub-processes. Each sub-process can be allocated in different nodes of the cloud system and each sub-process is operated independently. If some sub-processes have been compromised and the data are distorted, specific algorithms [34, 36,

40, 41] can be used to identify the abnormality. Furthermore, if the number of sub-processes with errors is not passing the threshold, the sub-processes with errors can be identified and the corrected value can be obtained [36]. And if the data type is implemented on huge integer operations or rational number calculations, the operation time cost will be significantly decreased. Both linear and non-linear calculation process can be applied with the multiple *P*-adic data type. The data type naturally has encryption property. And if the calculation process is linear, we can use an additional encryption key to encrypt it.

5.3.1. Data Self-correction Property

The algorithm for data self-correction property is coming from redundant residue number system (RRNS) [34]. Using multiple *P*-adic data type, we should define a prime set $p\sim\{p_1, p_2, \cdots p_k\}$. With the number of *k* primes, we can make sure to avoid the overflow situation [38]. But during the implementation, we set$p\sim\{p_1, p_2, \cdots p_k, p_{k+1}, \dots, p_n\}$. The $\{p_{k+1}, \dots, p_n\}$ part is the redundant part. According to Mandelbaum's theory [36], if $\frac{n-k}{2}$ or less sub-processes are changed, we can identify the compromised sub-processes and get the correct results. The main idea is to compare the decoded values from combination C_n^k among the nodes. For example, we use prime set $\{46337, 46327, 46309, 46307\}$. The redundant number *k=2*. We take 4 nodes $\{n_0, n_1, n_2, n_3\}$ of the cloud to do operations. One of the results is$19861/23831$. If there are no changes, the correct integers kept in the node should be {2443, 36085, 7970, 5634}. Assume the node n_0 is compromised by a hacker and the value is distorted from 2443 to 23201.

Node Values		Decoded Results
$\{n_0, n_1\} = \{46337,46327\}$	$\Leftrightarrow$	$-43262/3659$
$\{n_0, n_2\} = \{46337,46309\}$	$\Leftrightarrow$	$11973/37438$
$\{n_0, n_3\} = \{46337,46307\}$	$\Leftrightarrow$	$-26632/25557$
$\{n_1, n_2\} = \{46327,46309\}$	$\Leftrightarrow$	$19861/23831$
$\{n_1, n_3\} = \{46327,46307\}$	$\Leftrightarrow$	$19861/23831$
$\{n_2, n_3\} = \{46309,46307\}$	$\Leftrightarrow$	$19861/23831$

Through the above table, the correct decoded result is $19861/23831$ and node n_0 can be identified as abnormal.

The data self-correction process can be descripted as the following:

Step 1: Compare the decoded value of $\left\{n_0, n_1, \cdots, n_{\left\lceil \frac{n+k}{2} \right\rceil}\right\}$ and $\left\{n_{\left\lfloor \frac{n+k}{2} \right\rfloor}, \cdots, n_n\right\}$. If the values are equal, which means there is no error among the nodes, return the value. Otherwise, continue to Step 2.

*Step 2:*Get the decoded values from combination C_n^k among the nodes$v \sim \left\{v_0, v_1, \cdots, v_{C_n^k}\right\}$. Sort *v* and pick up the duplicate elements with their nodes indexes. If the number of the duplicate elements is less than$C_{\left\lceil \frac{n+k}{2} \right\rceil}^k$, which means there are more than $\frac{n-k}{2}$ nodes with errors and we cannot get the correct result, return with NULL. Otherwise, the result of the duplicate elements is the correct value and the nodes with the duplicate elements are the normal nodes, return them.

5.3.2. Linear Calculation Encryption Property

The multiple *P*-adic data type has natural encryption property, because the sub-processes only have the moduli values. For example, if the prime set is

$\{46337, 46327, 46309, 46307\}$ and the *P*-adic sequence length is 2, $1/463377$ will be represented as: $(33098, 5673; 11690, 1505; 26785, 12620; 6486, 1607)$. If only 1 node is compromised, the partial data is useless. However, the number 1 will be represented as $1, 0; 1, 0; 1, 0; 1, 0$. It is easy to get the original value just from 1 node. In this situation, we can choose a random fraction number *f* to multiple with the original number and after a linear operation, we can use *f* to decode the original fraction number.

5.3.3. Implementation of the Algorithm on HPC

We did an experiment to prove this method works in practice. During the experiment, we generated rational numbers which can be represented with length of 15 multiple *P*-adic data structure and the prime set are adjacent primes with the largest prime 46337. The redundant length was chosen $k = 5$. With the numbers of errors from 1 to 5. In each situation, we generated 1000 random rational numbers and tried to identify the modified digits. The experimental result is the following:

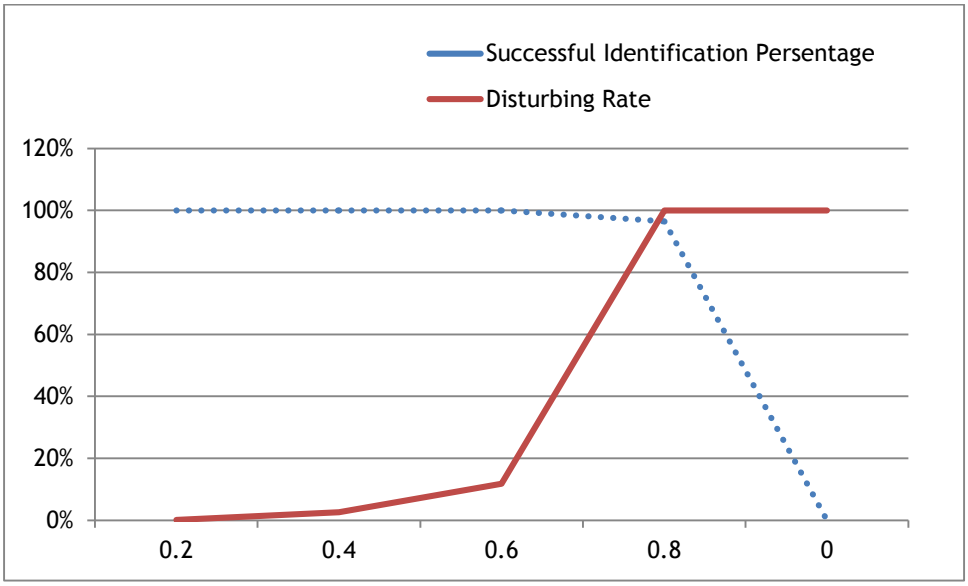


Figure 5.5 Implementation Results; Horizontal axis: (Number of Error Digits)/k

Successful Identification Percentage means among the 1000 experiment samples, the percentage of samples which have been successfully identified for the error digits and restored to the correct rational numbers. Disturbing Rate means, in the worst case situation, the number of disturbing groups divided by the number of correct groups. For example, if we want to identify the correct digits from the digits of a multiple *P*-adic data, we should do all the possible combinationsC_n^k of the digits to decode them to the rational number. Assume the number of compromised digits is *e*, there will be the number of C_{n-e}^kcombinations decoded results with the same value. Besides the correct digits having the same decoded results, some of the combinations with the compromised digits also will have the same decoded results, which will be called disturbing elements. Usually the number of disturbing elements is far more less than C_{n-e}^k. While with the increase of *e*, the number of correct groups C_{n-e}^kwill decrease, which will make it harder to identify the correct groups versus the disturbing groups. As we mentioned above, when the number of compromised digits is larger than $\frac{n-k}{2}$, it will be almost impossible to identify the correct or compromised digits. The number of disturbing groups is depended on the length of the Multiple *P*-adic Data Type.

A Method to Improve the Efficiency

In general, if you want to identify the correct digits, all the combination C_n^k of digits should be decoded to rational numbers. With the increase of the *n*, the computational complexity will be high. For example, if $n = 25, k = 15$ then $C_{25}^{15} = 3{,}268{,}760$, it will take too much time to decode the groups. We now give a better method that can significantly decrease the calculation load.

Step 1: Randomly select $m = k + e + 1$ digits among *n* digits of the multiple *P*-adic data, where *e* means the number of nodes being possibly compromised. If there is no prediction, *e* can be taken as 1.

Step 2: Get the decoded values from the combinations C_m^k among the nodes$v \sim \left\{v_0, v_1, \cdots, v_{C_m^k}\right\}$. Sort *v* and pick up the duplicate elements with their nodes index. If the number of the duplicate elements is less than C_{k+1}^k, which means the number of correct digits is less than *k*, then repeat step 2 with *m = m + 1*. Otherwise, the result from the duplicate elements is the correct value, and the nodes with the duplicate elements are the normal nodes, record them and the correct rational number.

Step 3: Replace the first digit from the correct group with one from the remaining digits and decode to get the rational number. If the rational number is the same, then that node is normal, take it to the normal nodes collection, otherwise it is incorrect. Go through the remaining digits and identify the correct and incorrect digits.

During step 2, if the disturbing rate is high, say possibly 2 or more groups which cannot be identified as the correct groups, then in this situation we can increase the value of *m* and repeat step 2. Or we can keep all of them to go through step 3. The correct digits are the group with the most digits.

The following experiments have been designed to compare the new method with the old version.

Experiment 5.3.1. During this experiment, we generated a rational number which can be represented with the length of 2 multiple *P*-adic data structure, and the prime set are the

adjacent primes with the largest prime to be 46337. The number of errors is fixed. The redundant length is *k* varies from4 to 33.In each situation, we generate 100 random rational numbers and try to identify the compromised digits using both old and new methods. The experimental results are given in Figures 5.6 and 5.7.

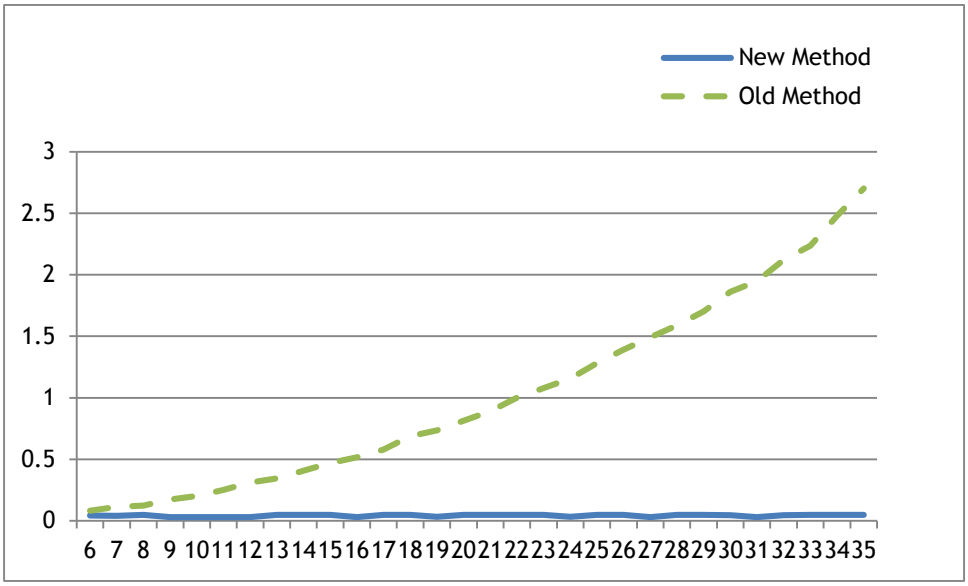


Figure 5.6 Experiment 1Implementation Results; Horizontal axis: Length of Multiple *P*-adic; Vertical axis: 100 operations cost time in second

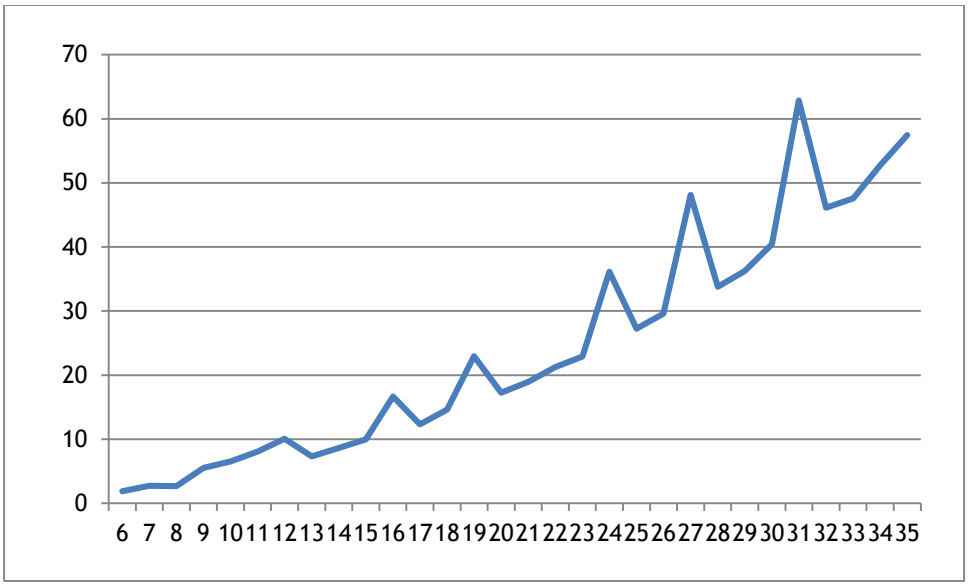


Figure 5.7 Experiment 1Implementation Results; Horizontal axis: Length of Multiple *P*-adic; Vertical axis: $\frac{Old\ Method\ Time\ Cost}{\text{New Method Time Cost}}$

From Figure 5.7, we can find that the time cost on the new method is significantly shorter than that of the old method. The reason is that it usually only needs 1 to 2 iterations for the new method to identify the compromised digits, and the steps cost would be like $\boldsymbol{C_{k+1}^{k} + C_{m-k-1}^{1}}$ for 1 iteration and $\boldsymbol{C_{k+2}^{k} + C_{m-k-2}^{1}}$ for 2 iteration, while the old method is $\boldsymbol{C_{m}^{k}}$. In this experiment,$\boldsymbol{k = 2}$, the time complexity for the new vs old versions would be $\boldsymbol{O(n)}$ vs $\boldsymbol{O(n^2)}$. Actually, with the increase of *k*, the old method's time complexity will increase to $(\boldsymbol{n^k})$, and new method's iteration steps will also increase but it still has a significant advantage.

Experiment 5.3.2. During this experiment, we generated a rational number which can be represented with the length of 6 multiple *P*-adic data structure and the prime set are adjacent primes with the largest prime to be 46337. The redundant length is *k = 10*. With the number of errors from 1 to 5. In each situation, we generated 100 random rational

numbers and tried to identify the compromised digits using both the old and new methods. The experimental results are given in Figures5.8 and 5.9.

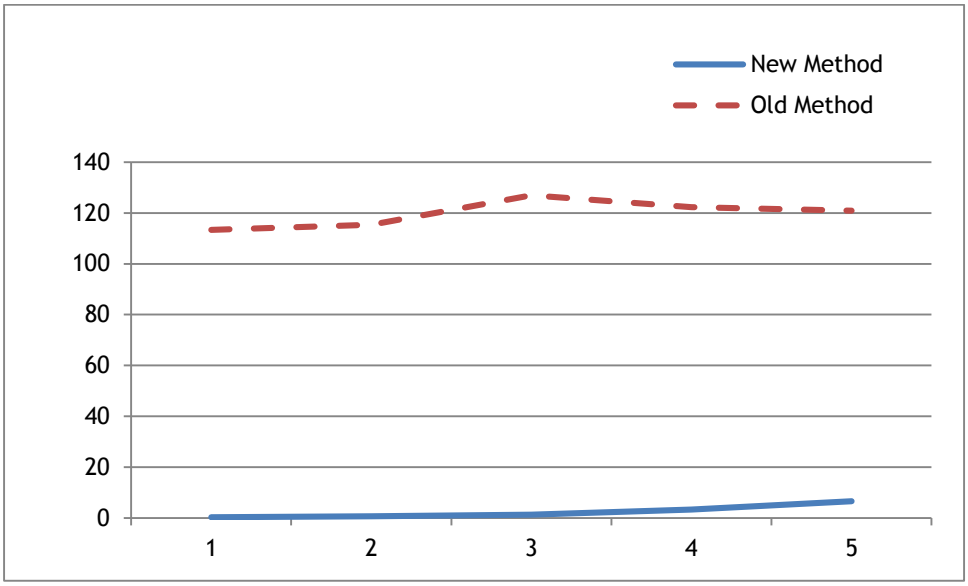


Figure 5.8 Experiment 2 Implementation Results; Horizontal axis: Length of Error Digits; Vertical axis: 100 operations cost time in second

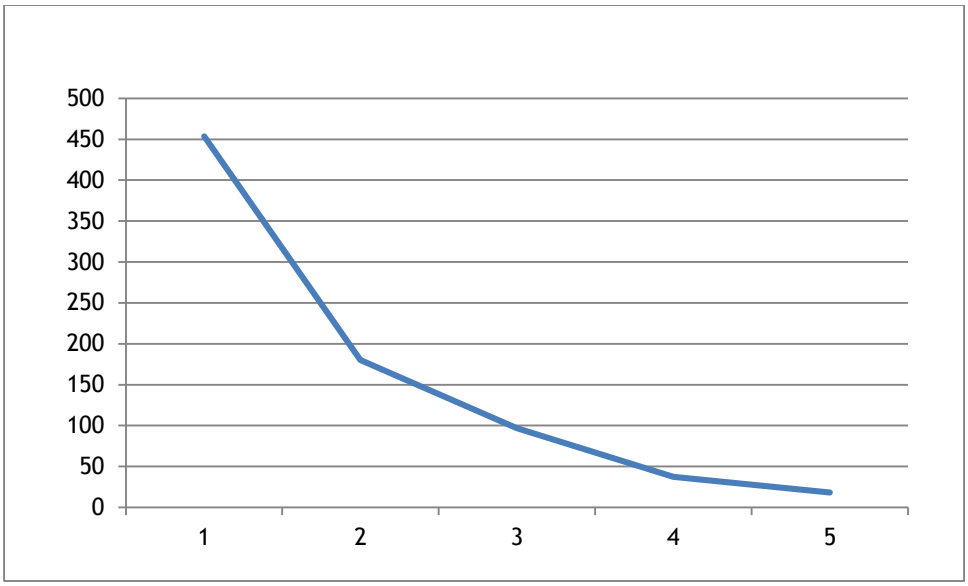


Figure 5.9 Experiment 2 Implementation Results; Horizontal axis: Length of Error Digits; Vertical axis: $\frac{Old\ Method\ Time\ Cost}{\text{New Method Time Cost}}$

From Figure 5.8, we can find that when *k* is large, the new method's advantage is more significant. The reason for Figure 5.9 is that when the number of error digits increases, the new method needs more and more iteration steps to identify the compromised digits.

References

[1] C. Yap, “Towards exact geometric computation”, Computational Geometry 7 (1997) 3-23.

[2] C. Moler and C. V. Loan, “Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later, Society for Industrial and Applied Mathematics”, Vol. 45, No. 1, pp. 3–000, 2003.

[3] J. A. Sjogren, notes.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7.

[5] V. Shoup, http://www.shoup.net/ntl/.

[6] X. Li, C. Lu and J. A. Sjogren, “A Method for Hensel Code Overflow Detection”, ACM SIGAPP Applied Computing Review, Vol. 12, Issue 1, p. 6-11, 2012.

[7] E. V. Krishnamurthy, “Matrix Processors Using P-adic Arithmetic for Exact Linear Computations”, IEEE Transactions on computers, Vol. C-26, No, 7, 1977.

[8] E. V. Krishnamurthy, T. M. Rao and K. Subramanian. “Finite Segment P–adic Number Systems with Applications to Exact Computation”, Proc. Indian Acad. Sci., 81A(2): 57-79, 1975.

[9] K. Hensel. *Theorie der Algebraischen Zahlen, Teubner*, Leipzig-Stuttgart, 1908.

[10] G. Bachman. *Introduction to P–adic Numbers and Valuation Theory*, Academic Press, New York, NY, 1964.

[11] D. M. Young and R. T. Gregory. "A Survey of Numerical Mathematics", Addison Wesley. Reading, Mass, 2 (1973).

[12] J. Dixon. "Exact Solution of Linear Equations Using P-adic Expansions", Numerische Mathematik 40, 137-141 (1982) Springer- Verlag.

[13] M. Miola. "The conversion of Hensel Codes to Their Rational Equivalents: or How to Solve the Gregory's Open Problem", ACM Sigsam bulletin, vol. 16, Issue 4, November 1982.

[14] P. Kornerup, R. T. Gregory. "Mapping Integers and Hensel Codes onto Farey Fractions", BIT 23, 9-20, 1983.

[15] C. K. Koc. "A Tutorial on P-adic Arithmetic", Technical Report, April 2002, Electrical & Computer Engineering, Oregon State University Corvallis, Oregon 97331.

[16] X. Li, M. Zhao and C. Lu. "Efficient Algorithms and Implementation for Error-free Computation Using P-adic", CSNI2011, JeJu, Korea, May 23-24, 2011.

[17] T. M. Rao, K. Subramanian and E. V. Krishnamurthy. "Residue Arithmetic Algorithms for Exact Computation of g-Inverses of Matrices", SIAM J. NUMER. ANAL. Vol. 13, No. 2, April 1976.

[18] G. L. Harvey, "The Residue Number System, Electronic Computers", IRE Transactions, V. EC-8, Issue. 2, June 1959.

[19] M. Newman, "Solving Equations Exactly", Mathematics and Mathematical Physics, Vol. 71B, No. 4, Oct-Dec 1967.

[20] J. A. Howell, “Solving systems of linear algebraic equations using residue arithmetic”, University of Texas at Austin, Computation Center, B0007ERILI, 1967.

[21] J. A. Howell and R. T. Gregory, “An Algorithm for Solving Linear Algebraic Equations Using Residue Arithmetic I”, Bit 9, 1969.

[22] C. Lu and X. Li, “An Introduction of Multiple P-adic Data Type and Its Parallel Implementation”, ICIS 2014.

[23] J. Morrison, “Parallel P-adic computation”, Information Processing Letters, Vol. 28, Issue 3, 1988.

[24] C. Limongelli and H. W. Loidl, “Rational Number Arithmetic by Parallel P-adic Algorithms”, Springer Verlag, editor, Proc. Of Second International Conference of the Austrian Center for Parallel Computation (ACPC), Vol. 734 of LNCS, 1993.

[25] C. K. Koc, “Parallel P-adic Method for Solving Linear Systems of Equations”, Parallel Computing, Vol 23, Issue 13, 1997.

[26] L. Shan, “Extension of P-adic Exact Scientific Computational Library (ESCL) to Compute the Exponential of a Rational Matrix”, M.S. thesis, Department of Computer & Information Science, Towson University, August 2007.

[27] C. Lu, X. Li and L. Shan, “Periodicity of the P-adic Expansion after Arithmetic Operations in P-adic Field”, Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference, 2012.

[28] R. T. Gregory. “The Use of Finite-segment *P*-adic Arithmetic for Exact Computation”, 18(3):282-300, 1978.

[29] R. T. Gregory and E. V. Krishnamurthy. “Methods and Applications of Error-Free Computation”. Springer, Berlin, Germany, 1984.

[30] A. Miola. “Algebraic approach to p-adic conversion of rational numbers. Information Processing Letters, 18(3):167-171, 30 March 1984.

[31] E. V. Krishnamurthy, “On the Conversion of Hensel Codes to Farey Rationals”, IEEE Transactions on Computers, Vol. C-32, No. 4, April 1983.

[32] H. Haramoto, M. Matsumoto, “A P-adic algorithm for computing the inverse of integer matrices”, Journal of Computational Applied Mathematics 225, 320-322 (2009).

[33] P. Kornerup and D. W. Matula, “Finite Precision Number Systems and Arithmetic”, Cambridge University Press, 2010.

[34] L. Yang and L. Hanzo, “Redundant Residue Number System Based Error Correction Codes”, Vehicular Technology Conference, IEEE VTS 54th, 2001.

[35] V. T. Goh and M. U. Siddiqi, “Multiple Error Detection and Correction Based on Redundant Residue Number Systems”, IEEE Transactions on Communications, Vol. 56, No. 3, March 2008.

[36] D. M. Mandelbaum, “Error Correction in Residue Arithmetic, IEEE Transactions on Computers”, Vol. c-21, No. 6, June 1972.

[37] O. Goldreich, D. Ron and M. Sudan, “Chinese Remaindering with Errors”, IEEE Transactions on Information Theory, Vol. 46, no. 7, July 2000.

[38] Xinkai Li, Chao Lu and Jon A. Sjogren, “Overflow Detection In Multiple P-adic Parallel Implementation”, RACS 2014.

[39] J. Du, W. Wei, X. Gu and T. Yu, "Towards Secure Dataflow Processing in Open Distributed Systems". Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing, 2009.

[40] D. M. Mandelbaum, "On a class of arithmetic codes and a decoding algorithm". IEEE Transactions on Information Theory, 22(1): 85-88, 1976. DOI=http://doi.acm.org/10.1109 /TIT.1976.1055504.

[41] D. M. Mandelbaum, "Further results on decoding arithmetic residue codes", IEEE Transactions on Information Theory, 24(5): 643-644, 1978.

[42] X. Li, M. Zhao, C. Lu and J. A. Sjogren, "Implementation of the Polynomial Method to Calculate e^{At} Using *P*-adic", Proceedings of the 2012 ACM Research in Applied Computation Symposium, 2012.

[43] B. Louis, "The Companion Matrix and Its Properties, The American Mathematical Monthly", Vol. 71, No. 6, pp. 629-634.

[44] J. A. Sjogren, Matlab code of the "Poor-Man" method.

[45] C. Lu, "Exact Computation of Generalized Inverse of Matrices with P-adic Exact Scientific Computational Library", Computer & Information Sciences, Towson University.

[46] E. J. Mastascusa, "A method of calculating e^{At}based on the Cayley-Hamilton theorem", Proc. IEEE, 57 (1969), pp. 1328-1329.

Appendix A

Full size of matrix

$$\begin{bmatrix} e_{00}^{A} & e_{01}^{A} \\ e_{10}^{A} & e_{11}^{A} \end{bmatrix}$$

$$e_{00}^{A} = \frac{\begin{array}{l} -4990319103277459371586782403429421124607383 57 \\ 6881921987519180225314410067784161624259457527 \\ 6026307920636823586372102345126805085844059078 \\ 4074542147651536093110756252220146919882101022 \\ 6061401097577072077485046894606805994907411701 \\ 2717289612880116209684578641320599811663054433 \\ 44414302147782848136159122084137871 46167 \end{array}}{\begin{array}{l} 9179179519067235333094775009324116369068786440 \\ 2291807740735072529720980104108840079387214968 \\ 6551682253701813182588071780421932625503562484 \\ 1616097423515172013613580871946770453324249466 \\ 3316241453549686813846861745670315523362009127 \\ 4830152650903361847890546666188872707840409600 \\ 00000000000000000000000000000000000000 \end{array}}$$

$$e_{01}^{A} = \frac{\begin{array}{l}336858576301722586681674054274061813793404090611055148610332192922658278476269206951130370751851153023420317914960372494647322845576117863988046054797564824792638875135793766456090167242662300180694029522825452979241605845488276031780733355473181790251971156218526524653642223827387801682545197233499079932161043782738780168254519723349907993216104378261556622754181591985011258793804510894439740482708507684288109393080819399059859282680871495804276150321687274981038203733241245290822743981058433250556617845717274654759649407874800941352493408522117607560684277624026532099825038651325691190656297311252540433441279180800000000000000000000000000000000000000000}{826155662275418159198501125879380451089443974048270850768428810939308081939905985928268087149580427615032168727498103820373324124529082743981058433250556617845717274654759649407874800941352493408522117607560684277624026532099825038651325691190656297311252540433441279180800000000000000000000000000000000000000000}$$

$$e_{10}^{A} = \frac{\begin{array}{l}-336858576301722586681674054274061813793404090611055148610332192922658278476269206951130370751851153023420317914960372494647322845576117863988046054797564824792638875135793766456090167242662300180694029522825452979241605845488276031780733355473181790251971156218526524653642223827387801682545197233499079932161043782738780168254519723349907993216104373098083733532818096994379222047676691585414902681015690381608041022405307274647447231005326810926603556370632728117889326399965466984058528992896912468958731692143977995534868527953050353007185028195794102835256604109009949537434389494247134196496111491719702662540479692800000000000000000000000000000000000000000}{309808373353281809699437922204767669158541490268101569038160804102240530727464744723100532681092660355637063272811788932639996546698405852899289691246895873169214397799553486852795305035300718502819579410283525660410900994953743438949424713419649611149171970266254047969280000000000000000000000000000000000000000}$$

$$e_{11}^{A} = \frac{\begin{array}{l}
44218018720066974680267106959109347977796055 23 \\
43819353287716113400829436459777726010669585 85 \\
82193744884045646328523622586375939883946599 27 \\
03529948836000974101045308627463501740079581 66 \\
68610916783168123302929708207769994523974292 42 \\
47852522648973926780052191841791825599351784 22 \\
92278626625923970826726398213879873711
\end{array}}{\begin{array}{l}
54223001378456242473977573894741615682262873 48 \\
65858779903076377812697772766887662916211396 43 \\
93132300232837715002629959884348969095801634 92 \\
75369351868865995017126638062132821243265608 66 \\
18745730105356799744243065164297222500872627 75 \\
72836344773268804164754284104069290257285120 00 \\
00000000000000000000000000000000000000000
\end{array}}$$

Appendix B

Proof for P-adic Arithmetic Using Long- Integer method

1. Introduction

By combining with the algorithm of Dr. E. V. Krishnamurthy[1] and John. D. Dixon[2], we have developed an algorithm on error free matrix calculation called "D-K Algorithm". Furthermore, we improve it to a more efficient way using long-integer method which fits the computer integer operation rules. To ensure correctness during the operation process, we should predict the range of prime p and P-adic series length r first. In this report, we focus on finding range of p and r in the basic arithmetic under long-integer method.

2. Proof of Basic Arithmetic Operations

In order to obtain the range limitation of p and r in basic arithmetic operations, we should go deep into the process.

Firstly, let's assume $a_1a_2a_3\cdots a_r$ and $b_1b_2b_3\cdots b_r$ are two P-adic series under p as the prime number and r as the length of the series.

For all the arithmetic proofs, we assume:

a) For P-adic expansion $a_1a_2a_3\cdots a_r$, each $a_i \in [0, p-1]$ under prime p

b) Since the limitation of algorithm, p and r should be less than the maximum of long integer (+2147483647); we denote $\boldsymbol{m}$.

We will prove the following operations: Addition, Subtraction, Multiplication and Division.

2.1 Addition

According to the operation rule of P-adic Addition:

$$\begin{array}{llllll} & a_1 & a_2 & a_3 & \cdots\cdots & a_r \\ + & b_1 & b_2 & b_3 & \cdots\cdots & b_r \\ \hline & d_1 & d_2 & d_3 & \cdots\cdots\ldots & d_r \end{array}$$

……

Because $d_i = \left\| a_i + b_i \right\|_p$, we can get $d_1 d_2 d_3 \cdots d_r$ which is also a P-adic series. During the process the largest integer possible shows out is $a_i + b_i$.

Since $a_i, b_i \in [0, p-1],\ \therefore a_i + b_i < 2p < m\ \therefore\ \boldsymbol{p < \frac{m}{2}}$

2.2 Subtraction

Subtraction is similar to P-adic addition but add one more step:

$$\begin{array}{llllll} & a_1 & a_2 & a_3 & \cdots\cdots & a_r \\ - & b_1 & b_2 & b_3 & \cdots\cdots & b_r \\ \hline & d_1 & d_2 & d_3 & \cdots\cdots\ldots & d_r \end{array} \quad \Rightarrow \quad \begin{array}{llllll} & a_1 & a_2 & a_3 & \cdots\cdots & a_r \\ + & c_1 & c_2 & c_3 & \cdots\cdots & c_r \\ \hline & d_1 & d_2 & d_3 & \cdots\cdots\ldots & d_r \end{array}$$

…… ……

When we deal with subtraction, we transfer the subtrahend to a positive P-adic series. If so, we can handle the problem like addition. Because we follow the rule of modulo arithmetic in P-adic field, using $\left\| \boldsymbol{b_i + c_i + t_i} \right\|_{\boldsymbol{p}} = 0$ ($\boldsymbol{t_i}$ is carry number from $\boldsymbol{b_{i-1}}$+$\boldsymbol{c_{i-1}}$, so it only can be 0 or 1. In this way, we can easily transfer the series $-b_1 b_2 b_3 \cdots b_r$ to $+c_1 c_2 c_3 \cdots c_r$ which is also a P-adic series and satisfies with the condition that $\left\| -b_1 b_2 b_3 \cdots b_r \right\|_p = \left\| +c_1 c_2 c_3 \cdots c_r \right\|_p$. Then, we do calculation as follows:

$$
\begin{array}{cccccc}
 & a_1 & a_2 & a_3 & \cdots\cdots\cdots & a_r \\
+ & c_1 & c_2 & c_3 & \cdots\cdots\cdots & c_r \\
\hline
 & d_1 & d_2 & d_3 & \cdots\cdots\cdots & d_r \\
\end{array}
$$

$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$

So whatever $a_i + b_i$ or $c_i + b_i$ will satisfies $p < \frac{m}{2}$

2.3 Multiplication

According to the operation rule of P-adic multiplication, we can split the multiplication operation into two steps showed as follows: the first step is to get new temporary series consisting of d_{ij}, and the next step is to sum up all these temporary series.

$$
\begin{array}{cccccc}
 & a_1 & a_2 & a_3 & \cdots\cdots\cdots & a_r \\
\times & b_1 & b_2 & b_3 & \cdots\cdots\cdots & b_r \\
\hline
 & d_{11} & d_{12} & d_{13} & \cdots\cdots\cdots & d_{1r} \\
 & & d_{21} & d_{22} & \cdots\cdots\cdots & d_{2r} \\
 & & & d_{31} & \cdots\cdots\cdots & d_{3r} \\
 & & & & \cdots\cdots\cdots\cdots\cdots & \\
+ & & & & & d_{r1} \\
\hline
\end{array}
$$

$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$

Now, let’s go to the first step to compute $d_{11}d_{12}d_{13}\cdots d_{1r}$ as an example. There is two parts when computing d_{ij}.

Part (1) is the simple multiplication series as $a_1b_1\ a_2b_1\ a_3b_1\cdots a_jb_i\cdots a_rb_1$ $(i, j \le r)$ and part (2) is the carry number as c_{ij}, which is carry number for a_jb_i.

$$
\underline{digit:\quad 1 \quad 2 \quad 3 \quad \ldots \quad r}\rightarrow
$$

$$\begin{array}{llllll} & a_1b_1 & a_2b_1 & a_3b_1 & \cdots & a_r b_1 \quad \rightarrow (1) \\ + & c_{11} & c_{12} & c_{13} & \cdots & c_{1r} \quad \rightarrow (2) \\ \hline & d_{11} & d_{12} & d_{13} & \cdots & d_{1r} \end{array}$$

Before estimating d_{ij}, we should first figure out the range of the carry number c_{ij}. Knowing the modulus operation is used in P-adic field, we can proof:

$\because a_i, b_i \in [0, p-1] \qquad \therefore a_j b_i < p^2 \ (i, j \le r)$

$c_{11} = 0; \quad c_{12} = \text{int}[(a_1b_1 + c_{11})/p] < p;$

$c_{13} < (a_2b_1 + c_{12})/p < (p^2 + p)/p < p + 1;$

$c_{14} < (a_3b_1 + c_{13})/p < (p^2 + p + 1)/p < p + 1 + \frac{1}{p};$

……

then $c_{1r} < (a_{r-1}b_1 + c_{1r-1})/p < p + 1 + \frac{1}{p} + \frac{1}{p^2} + \cdots + \frac{1}{p^{r-3}}$

$$\lim_{r\to\infty} c_{1r} = \lim_{r\to\infty} p + 1 + \frac{1}{p} + \frac{1}{p^2} + \cdots + \frac{1}{p^{r-3}} = p + \frac{1}{1 - 1/p}$$

If we assume m= 2147483647 (maximum of long integer), and then we approximately have

$$\lim_{p\to m}(p + \frac{1}{1-1/p}) = p + 1;$$

$$\therefore \lim_{p\to m}\lim_{r\to\infty} c_{1r} = p + 1;$$

$$\therefore \lim_{r\to\infty}\lim_{p\to m} d_{1r} = \lim_{r\to\infty}\lim_{p\to m}(c_{1r} + a_r b_1) \le (p-1)^2 + p + 1 < p^2$$

Since d_{1r} can be the largest number of the calculation, so if we set $\boldsymbol{p^2 < m}$, then the error of data overflow won't occur during the first step. Here, one thing should be clear that d_{ij} can only be greater than *p* during the calculation process, but each d_{ij} of

$d_{11}d_{12}d_{13}\cdots d_{1r}$ must meet the condition: $d_{ij} \in [0, p-1]$ after carry, because it is also a P-adic series under p as prime.

After completing the first step, now we start the second step to sum up all temporary series.

1	2	3	……	r −digit
d_{11}	d_{12}	d_{13}	……	d_{1r}
	d_{21}	d_{22}	……	d_{2r}
		d_{31}	……	d_{3r}
			……	
				d_{r1}
+				
……				

1	2	3	……	r −digit
d_{11}	d_{12}	d_{13}	……	d_{1r}
	d_{21}	d_{22}	……	d_{2r}
		d_{31}	……	d_{3r}
			……	
				d_{r1}
+ e_1	e_2	e_3	……	e_r
……				

Above graph shows addition of r P-adic series, we can still use the same method before adding e_i as the i-th carry number. So we rewrite column addition involving e_i.

And now we can proof that:

For each e_i, the carry number of column i, reflects the effect from column 1 to column (i-1). Take column 4 for example, e_4 can be affected by column 1 to column 3. For column 1, it has no effect on e_4 since d_{11} can't be more than p. For column 2, it has effect no greater than 1/p because $(d_{12} + d_{21})/p < 2p/p = 2$, which means it can be 0 or 1. If it carries 1 to column 3, so this 1 will carry to column 4 as only 1/p. For column 3, it affects no greater than 2 because $(d_{13} + d_{22} + d_{31})/p < 3p/p = 3$, which means it can be 0, 1 or 2. So, the number carry to column 4 is no greater than

2. Thus, the maximum value of e_4 could be $2 + 1/p$.

By analogy, we have $e_5 = 3 + 2/p + 1/p^2$;

And $e_r = (r-2) + (r-3)/p + (r-4)/p^2 + \cdots + 1/p^{r-3}$ ..... (1)

$p \cdot e_r = (r-2) \cdot p + (r-3) + (r-4)/p + \cdots + 1/p^{r-4}$ ..... (2)

$(1) - (2)$, we have $(1-p) \cdot e_r = -(r-2) \cdot p + 1 + 1/p + \cdots + 1/p^{r-4} + 1/p^{r-3}$

We assume m= 2147483647 (maximum of long digit), and then we have

$\lim_{p \to m} e_r = \lim_{p \to m} \frac{-(r-2) \cdot p + 1 + 1/p + \cdots + 1/p^{r-4} + 1/p^{r-3}}{1-p}$, because both numerator and denominator are polynomial and continuous except $p = 1 \, or \, 0$. So,

$$\lim_{p \to m} e_r = \lim_{p \to m} \frac{-(r-2) \cdot p}{1-p} + \lim_{p \to m} \frac{1 + 1/p + \cdots + 1/p^{r-4} + 1/p^{r-3}}{1-p}$$

$$= (r-2) + \lim_{p \to m} \frac{p^r - 1}{p^{r-1} \cdot (p-1)^2} = r - 2 + 0$$

$$= r - 2$$

Now, let's see the last addition of column addition $d_{r1} + e_r$. Since d_{r1} is a P-aidc number, the error of data overflow won't occur if we set $p + (r-2) < \mathrm{m}$ which is same as $\boldsymbol{r < m + 2 - p}$.

Thus, to sum up the points which we have just indicated, $\boldsymbol{p^2 < m}$ and $\boldsymbol{r < m + 2 - p}$ are the two conditions we cannot violate during multiplication process.

2.4 Division

Basically, we deal with division by transfer it to multiplication like:

$$\begin{array}{llllll} & a_1 & a_2 & a_3 & \cdots\cdots & a_r \\ \div & c_1 & c_2 & c_3 & \cdots\cdots & c_r \\ \hline & m_1 & m_2 & m_3 & \cdots\cdots & m_r \\ \cdots\cdots & & & & & \end{array} \quad \Rightarrow \quad \begin{array}{llllll} & a_1 & a_2 & a_3 & \cdots\cdots & a_r \\ \times & b_1 & b_2 & b_3 & \cdots\cdots & b_r \\ \hline & m_1 & m_2 & m_3 & \cdots\cdots & m_r \\ \cdots\cdots & & & & & \end{array}$$

So we need add one more step that find $b_1b_2b_3\cdots b_r$ which is $c_1c_2c_3\cdots c_r^{-1}$. To do that, we consider the following. If so, we say $b_1b_2b_3\cdots b_r = c_1c_2c_3\cdots c_r^{-1}$.

$$\begin{array}{llllll} & b_1 & b_2 & b_3 & \cdots\cdots & b_r \\ \times & c_1 & c_2 & c_3 & \cdots\cdots & c_r \\ \hline & d_{11} & d_{12} & d_{13} & \cdots\cdots & d_{1r} \\ & & d_{21} & d_{22} & \cdots\cdots & d_{2r} \\ & & & d_{31} & \cdots\cdots & d_{3r} \\ & & & & \cdots\cdots & \\ + & & & & & d_{r1} \\ \hline & 1 & 0 & 0 & \cdots\cdots & 0 \end{array}$$

Actually, to perform the division, we do multiplication twice, one for finding the divisor's inverse and another for computing the final result. Since these two multiplication is independent to each other, the range limitation of *p* and r would not change, which still be $\boldsymbol{p}^2 < \boldsymbol{m}$ and $\boldsymbol{r} < \boldsymbol{m} + 2 - \boldsymbol{p}$.

3. Reference

[1]. Krishnamurthy, E. V. Matrix Processors Using P-adic Arithmetic for Exact Linear Computations, IEEE Transactions on Computers, vol. C-26, No. 7, July 1977.

[2]. Dixon, J. "Exact Solution of Linear Equations Using P-adic Expansions", Numerische Mathematik 40, 137-141 (1982) Springer- Verlag.

CURRICULUM VITA

NAME: Xinkai Li

[redacted]

PROGRAM OF STUDY: Information Technology

DEGREE AND DATE TO BE CONFERRED: Doctor of Science, 2015

Towson University	2011	Information Technology	D. Sc.	2015
Towson University	2009	Applied & Industrial Math	M. S.	2011
University of Baltimore	2007	University of Baltimore	M. S.	2008
Jinan University	2003	Applied Math	B. S.	2007

PROFESSIONAL PUBLICATIONS:

- **X. Li**, C. Lu, "Proactive Self-defense Algorithm for Large Matrix Calculation Using Multiple *P*-adic Data Type", submitted to RACS 2015, in reviewing.
- **X. Li**, C. Lu and J. A. Sjogren, "Overflow Detection In Multiple *P*-adic Parallel Implementation", RACS 2014.
- C. Lu and **X. Li**, "An introduction of Multiple *P*-adic Data Type and Its Parallel Implementation", ICIS 2014.
- **X. Li**, C. Lu and J. A. Sjogren, "Parallel Implementation of Exact Matrix Computation Using Multiple *P*-adic Arithmetic", SNPD 2013 (also on International Journal of Networked and Distributed Computing-Atlantis Press, 1(3), August, 2013).

- J. A. Sjogren, **X. Li**, M. Zhao, and C. Lu, "Computable Implementation of "Fundamental Theorem of Algebra", International Journal of Pure and Applied Mathematics, 2013, ISSN 1311-8080.
- **X. Li**, M. Zhao, C. Lu and J. A. Sjogren, "Implementation of the Polynomial Method to Calculate e^At Using *P*-adic", RACS, 2012.
- **X. Li**, C. Lu and J. A. Sjogren, "A Method for Hesel Code Overflow Detection", Applied Computing Review, Vol. 12 No. 1, pp. 6-11, 2012.
- C. Lu and **X. Li**, "Periodicity of the *P*-adic Expansion after Arithmetic Operations in *P*-adic Field, ICIS", China, 2012.
- **X. Li**, M. Zhao and C. Lu, "Efficient Algorithms and Implementation for Error-free Computation Using *P*-adic", CSNI2011, Korea, 2011.

PROFESSIONAL POSITION HELD:

Lynchval Systems Worldwide, Inc.

13921 Park Center Rd Suite 100

Herndon VA 20171