SOME HIGHLY ACCURATE BASIC LINEAR ALGEBRA SUBROUTINES

SOME HIGHLY ACCURATE BASIC LINEAR ALGEBRA SUBROUTINES

BY

YUHANG ZHAO, B.Sc.

A THESIS

SUBMITTED TO THE SCHOOL OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

© Copyright by YUHANG ZHAO, September 2010

All Rights Reserved

Master of Science (2010)
(Computing and Software)

McMaster University
Hamilton, Ontario, Canada

TITLE:	SOME HIGHLY ACCURATE BASIC LINEAR ALGEBRA SUBROUTINES
AUTHOR:	YUHANG ZHAO B.Sc., (Computer Science) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Sanzheng Qiao
NUMBER OF PAGES:	ix, 54

Abstract

In the computation of the sum of many floating-point numbers x_i $(i = 1, 2, ...n-1, n)$, the method $S = ((...((x_1+x_2)+x_3)+...+x_{n-1})+x_n)$ is called the Ordinary Recursive Summation (ORS) algorithm. Since significant digits might be discarded when the partial sums are rounded, the results are rarely correct. In 1969, Knuth [1] proposed a simple algorithm AddTwo for transforming a pair of floating-point numbers (a, b) into a new pair (x, y) with non-overlapping mantissas and satisfying $x = fl(a + b)$ and $a + b = x + y$, regardless of the magnitude of a and b, where x is the floating-point sum of a and b, while y is the roundoff error. We call an algorithm with such property an *error-free* transformation. Such transformations are at the center of the interest of this thesis. Extending the principle of AddTwo to n summands is called *distillation* by Kahan. Since then, many distillation algorithms have appeared to improve the accuracy of summation. Among them, there are three accurate summation algorithms SumK, iFastSum and HybridSum, which are particularly appropriate for ill-conditioned data, where ORS fails due to the accumulation of rounding error and severe cancellation. In this thesis, we present the accurate summation algorithms with their properties, and then apply them to improve the accuracy of the LAPACK subroutines DDOT and DGEMV.

Acknowledgements

First I would like to express my deep appreciation to my supervisor, Dr. Sanzheng Qiao who gave me the great opportunity to study and do research here. I wish to express my sincere gratitude for his strong support, constant encourage, and instructive guidance for my research and thesis. His detailed comments, enlightening suggestions, and complete corrections help ameliorate this thesis greatly. I have learned much from him in both academic research and non-academic fields.

Next I want to thank Dr. Wolfram Kahl. I greatly appreciate his comprehensive review for this thesis and his valuable suggestions and comments. I would like to thank Dr. Alan Wassyng for being an examiner of thesis defense committee, and also his helps through my studies here. I also want to thank Dr. Christopher Anand for his suggestions and help.

I am grateful to my dearest parents. I would never accomplish any goal in my life without their selfless love and support in many years.

Finally, I give my special thanks to my wife Jinfang Xu for her support and understanding during my studies.

Contents

Abstract iii

Acknowledgements iv

1 Introduction 1

2 Summation Algorithms 8

2.1 Kahan's Algorithm 8

2.1.1 Algorithm Description 9

2.1.2 Properties 11

2.2 SumK 12

2.2.1 Algorithm Description 13

2.2.2 Properties 14

2.3 iFastSum 15
2.3.1 Algorithm Description 17
2.3.2 Properties 19
2.4 HybridSum 20
2.4.1 Algorithm Description 22
2.4.2 Properties 23
3 Numerical Experiments **25**
3.1 Test Data 25
3.2 Test Results 26
3.3 Accuracy 32
3.4 Running Time 33
3.5 Space Complexity 35
4 Application to LAPACK subroutines **36**
4.1 Vector Dot Product 36
4.2 Vector-Matrix and Matrix-Matrix multiplications 45

List of Tables

3.1 Test results (returned sums) of the algorithms on the extremely ill-conditioned data 27

3.2 Test results (relative errors) of the algorithms on well-conditioned data 28

3.3 Test results (relative errors) of the algorithms on random numbers including signs 28

3.4 Test results (relative errors) of the algorithms on Anderson's Ill-Conditioned Data 29

3.5 The exact sums of the four testing data sets with different δ values . 29

3.6 Test timing of the algorithms on the extremely ill-conditioned data . 30

3.7 Test timing of the algorithms on the well-conditioned data 31

3.8 Test timing of the algorithms on the random numbers including signs 31

3.9 Test results of the algorithms on the Anderson's Ill-Conditioned Data 31

3.10 The least K and Running Time for SumK to produce the correct results on the extremely ill-conditioned data 32

4.1 Measured computing time for different dot product algorithms with Ddot_LAPACK normed to 1 44

List of Figures

1.1 Algorithm TwoSum 5

2.1 Recovering the rounding error 10

3.1 Comparison of Timing Results of iFastSum, HybridSum and Sum3 . . 34

4.1 Test Results for Ddot_Lapack 48

4.2 Test Results for Ddot_ORS 48

4.3 Test Results for Ddot_Kahan 49

4.4 Test Results for Ddot_Sum3 (Condition Number $\leq 10^{30}$) 49

4.5 Test Results for Ddot_Sum3 50

4.6 Test Results for Ddot_iFastSum 50

4.7 Test Results for Ddot_HybridSum 51

4.8 Test Results for Ddot_LAPACK (Condition Number $\leq 10^{20}$) 51

4.9 Test Results for Ddot_ORS (Condition Number $\leq 10^{20}$) 52

4.10 Test Results for Ddot_Kahan (Condition Number $\leq 10^{20}$) 52

Chapter 1

Introduction

In this thesis, we study the problem of accurate summation and vector dot product of large vectors of floating-point numbers. Accurate summation and vector dot product have applications in many areas of numerical analysis, e.g., in accurate matrix multiplication, in iterative refinement of the solution of $Ax - b$, in the problem of inversion of extremely ill-conditioned matrices [2]. Since vector dot product can be transformed into summations, we put the main effort on accurate summations.

Summation is the fundamental task in numerical computation, Higham devoted a chapter for the problem of summation in [3]. In the standard computer systems, floating-point numbers are stored with limited digits. In the addition of two floating-point numbers, the computer arithmetic first aligns the two summands with the same exponent, and then computes the sum and rounds it to fit in the limited digits. Some digits might be discarded during the rounding, which is called the rounding error.

In the computation of the sum of many floating-point numbers x_i $(i = 1, 2, ...n-1, n)$, the method $S = ((...((x_1 + x_2) + x_3) + ... + x_{n-1}) + x_n)$ is called the Ordinary Recursive Summation (ORS) algorithm. Since significant digits might be discarded during the floating-point additions, the results are rarely correct due to cancellation. The condition number

$$R = \frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|}$$

of the set of floating-point numbers x_i is used to measure the severity of cancellation.

In the following, we use $fl(\cdot)$ where "$\cdot$" $\in \{ +, -, *, \backslash \}$ to denote the floating-point arithmetic. Let $y = fl(x)$ denote the assignment that assigns to y the floating-point number closest to x. Assume $a \leq x < b$, where a and b are two adjacent floating-point numbers. We say that y is *correctly rounded* if y is rounded to the nearest of a or b determined by the value of x. If x is the mid-point of a and b, y should be rounded to a or b deterministically. We say that y is *faithfully rounded* if y is rounded to a only when $x = a$, and either a or b otherwise. In the following, we assume that the ordinary computer arithmetic $fl(\cdot)$ is correctly rounded.

Let β denote the base of the floating-point number, t denote the length of the mantissa, and l denote the length of exponent where both mantissa and exponent are represented in base β. We use the IEEE754 double precision standard where $\beta = 2$, $t = 53$ including an implicit bit and $l = 11$. We assume a double floating-point number is stored in the computer by 64 bits and the bit operations are allowed.

We denote the relative rounding error unit of the given format by *eps* which is also

called *Machine Epsilon* [3]. According to the IEEE754 standard, floating point operations satisfy $fl(a \cdot b) = (a \cdot b)(1 + \epsilon)$ where "$\cdot$" $\in \{ +, - \}$ and $|\epsilon| \leq eps$. In mathematical terms, the fl-notation implies $|fl(a \cdot b) - a \cdot b| \leq eps|a \cdot b|$. The relative rounding error unit gives an upper bound on the relative error due to rounding in floating point arithmetic. In IEEE754 the quantity $eps = \frac{1}{2}\beta^{1-t} = 2^{-53}$. We use Ulp to denote Unit in the Last Place [3]. It is the gap between the floating-point numbers nearest a given real number. More generally, it is the absolute value of the distance between the two floating-point numbers which are closest to a given number. It is used as a measure of accuracy in numeric calculations. One Ulp of the floating-point number $y = \pm\beta^e \times d_1.d_2d_3...d_t$ is Ulp(y) $= \beta^e \times .00...01 = \beta^{e-t}$. Throughout the thesis we assume no overflow occurs, but allow underflow.

Some algorithms diminish rounding errors by sorting the input data, which incurs the extra time $O(n\ log\ n)$ for adding n floating-point numbers. Some algorithms use long accumulators for summations to minimize the rounding errors. Both approaches slow down the programs performance. There is another approach to improve the accuracy of the floating-point summations by *compensated summation*. This approach estimates the rounding error from the floating-point summation and recycles it at the succeeding summations.

In 1969, Knuth [1] proposed a simple algorithm AddTwo for transforming a pair of floating-point numbers (a, b) into a new pair (x, y) with non-overlapping mantissas and satisfying $x = fl(a + b)$ and $a + b = x + y$, regardless of the magnitude of a and b, where x is the floating-point sum of a and b, while y is the roundoff error. We call

an algorithm with such property an *error-free* transformation. Such transformations are at the center of the interest of this thesis. Other *error-free* transformations like TwoProduct, Split and VecSum will be presented when they are applied in the later chapters.

ALGORITHM $(x, y) = \text{AddTwo}(a, b)$

(1) $x = fl(a + b)$

(2) $z = fl(x - a)$

(3) $y = fl((a - (x - z)) + (b - z))$

Let's briefly describe how the algorithm works, the working mechanism is illustrated in Figure 1.1. Assume $a > b > 0$ and a overlaps b. The other cases like $a < b$, or a does not overlap b can be treated in the similar way. First we compute $x = fl(a+b)$, which is the sum of a and b using the ordinary computer arithmetic. Then we estimate the quantity z by $z = fl(x-a)$, which satisfies $fl(a+b) = fl(a+z)$. In Step (3) we compute the rounding error y by $y = fl((a - (x - z)) + (b - z))$. In this example, we assume $a > b > 0$ and a overlaps b, therefore $fl(a - (x - z)) = 0$ and $fl(b - z)$ is the estimated rounding error.

Algorithm AddTwo plays a fundamental role in the highly accurate algorithms like SumK [7], iFastSum and HybridSum [6], since they all apply the algorithm instead of the ordinary computer $fl(+)$ arithmetic so that all of the rounded errors are preserved. No significant digits are discarded during their summations.

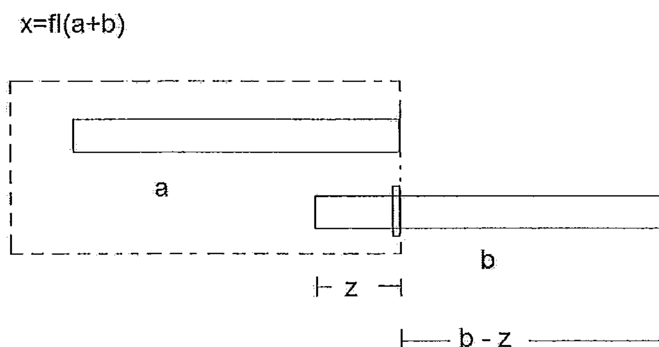

Figure 1.1: Algorithm TwoSum

Another compensated summation is the following algorithm by Dekker[4].

ALGORITHM $[x, y] = \text{FastTwoSum}(a, b)$

$$x = fl(a + b)$$

$$y = fl((a - x) + b)$$

Dekker showed in 1971 that the result is exact if the input is ordered by magnitude, that is $x + y = a + b$ given $|a| \geq |b|$. However, branches involved in the sorting of a and b may increase the computing time due to the lack of compiler optimization. We will see in Chapter 2 that Kahan applied this idea in his summation algorithm. Kahan's algorithm performs well with most data sets and gives more accurate results than ORS. This method is recommended as an efficient algorithm for general data.

Extending the principle of AddTwo to n summands is called *distillation* by Kahan [5], because we can think of it as a process to seperate the significant parts of each

summand and then add that part into the sum. It is an iterative refinement using AddTwo to create a new array of floating-point numbers with mutually non-overlapping mantissas, but whose sum is exactly equal to the sum of the original array.

We will see in Chapter 2 that Rump applied the idea of distillation in his accurate array summation algorithm SumK [7], which has an integer parameter K representing the limit of the numbers of the iterative refinements. For sufficiently large K, the algorithm is guaranteed to return the correct result. However, it is not easy to know the proper value of K beforehand, and always using a large value of K will cause overhead.

Recently, Y. K. Zhu proposed two highly accurate array summation algorithms iFastSum and HybridSum [6]. The idea of distillation was also applied in iFastSum with a clever method to control the loops of iterative refinements on the array. HybridSum uses a large number of "buckets" to store the partial sums, and its running time is independent of the condition number.

This thesis has two purposes. The first is to present comparisons of the above-mentioned four accurate summation algorithms. Those algorithms are applicable to all sets of data but are particular appropriate for ill-conditioned data where ORS fails due to the accumulation of rounding error and severe cancellation. The second purpose is to apply those summation algorithms to DDOT and DGEMV, which are two fundamental subroutines of LAPACK [10]. The first 'D' in the function name suggests that it works in the double precision. LAPACK is a Linear Algebra PACKage with a rich set of subroutines concerning the following algebra problems: linear equations,

linear least squares problems, eigenvalue problems and singular value problems. In the following we use the C-LAPACK instead of the Fortran 77 implementation. BLAS is a set of subroutines that implements low level functions for vectors and matrices operations used in LAPACK. BLAS is divided into three levels.

Level 1: vector/vector operations;

Level 2: matrix/vector operations;

Level 3: matrix/matrix operations.

LAPACK routines perform their computations with the facilities offered by BLAS routines.

This thesis is organized as follows. In Chapter 2 we will present the four accurate summation algorithms as well as their properties. In Chapter 3, we will compare the summation algorithms in accuracy, execution time and space complexity. Numerical results are also included to support the comparisons. In the last part, the summation algorithms are applied to improve the performance of some LAPACK subroutines, and we also supply the numerical results.

Chapter 2

Summation Algorithms

In this chapter, we will present the four recent summation algorithms. We also analyze how and why the summation algorithms work.

2.1 Kahan's Algorithm

Kahan's algorithm is a compensated summation algorithm. In 1951 Gill noticed that in computing the sum of two numbers, the rounding error could be estimated by subtracting one summand from the sum. In 1970 Kahan used the similar idea to derive a compensated summation method to compute the sum of a floating-point number array [3].

ALGORITHM: $s \leftarrow \text{Kahan}(x, n)$

Input: x, the array of the given floating-point summands

n, the length of the array

Output: s, the sum of the array

1. $s \leftarrow 0$; $e \leftarrow 0$;
2. for $i \leftarrow 1$ to n
   (a) $temp \leftarrow s$;
   (b) $y \leftarrow x_i + e$;
   (c) $s \leftarrow temp + y$;
   (d) $e \leftarrow (temp - s) + y$;
3. END

2.1.1 Algorithm Description

Figure 2.1 gives an intuitive explanation of Compensated Summation. Assume a and b are two floating point numbers with $|a| \geq |b|$. Let's denote $s = a + b$ and $\hat{s} = fl(a + b)$. The figure shows that if we evaluate $e = -fl(fl(fl(a + b) - a) - b)$ with floating-point arithmetic $fl(\cdot)$ in the order indicated by the parentheses, then the computed e will be a good estimate of the error $s - \hat{s}$. For rounded floating-point arithmetic in base 2, we have $s = \hat{s} + e$, that is, the computed e represents the error exactly. This result is proved by Knuth in [1].

The main idea of Kahan's algorithm is to keep the rounding error of the current summation and feed it back into the next summation. It employs a correction term

a	a_1	a_2		
$+\ b$		b_1	b_2	
$= \hat{s}$	a_1	$a_2 + b_1$		
$-\ a$	a_1	a_2		
$= \hat{s} - a$		b_1	0	
$-\ b$		b_1	b_2	
$= (\hat{s} - a) - b$			$-b_2$	0
$e = -((\hat{s} - a) - b) = (a - \hat{s}) + b$			b_2	0

Figure 2.1: Recovering the rounding error

on every step of a recursive summation to diminish the rounding errors. Specifically, a correction term is computed immediately after each partial sum is calculated, and in the next loop it is added into the next summand x_i before that summand is added to the partial sum.

Step 1 first initializes the variable s which is used to store the partial sum of the floating-point array x, and then it initializes the correction term e. Step 2 is the main loop of the algorithm. Step 2(a) first records the partial sum s of the previous loop in the variable temp. In Step 2(b) the correction term e computed in the previous loop is added to the current summand x_i to compute the corrected summand y. Step 2(c) adds the corrected summand y to the previous partial sum to compute the current partial sum. Step 2(d) estimates the rounding error generated by step 2(c) and stores it in e as the correction term to be applied in the next loop.

2.1.2 Properties

- The algorithm has two weaknesses. First, the correction term e computed in Step2(d) may not be the exact rounding error, because the correction term e is calculated based on the assumption that $|a| \geq |b|$. (see Figure 2.1). If the condition $|a| \geq |b|$ is not satisfied, the correction term e is always computed to zero. In other words, the computation of the current partial sum shares the same error estimated with the ordinary addition operation in computer. Second, the local error generated by the addition $y = x_i + e$ in Step2(b) is discarded.

- Comparing with the sum $s = \sum_{i=1}^{n} x_i$ computed by ORS, Kahan's algorithm improves the error bound of the sum significantly. Knuth showed that the sum s computed by Kahan algorithm satisfies [1]

$$s = \sum_{i=1}^{n}(1 + \delta_i) \cdot x_i, \qquad where|\delta_i| \leq 2\varepsilon + O(n\varepsilon^2).$$

- If the condition number of the floating-point number array x is very large, that is if $\sum_{i=1}^{n} |x_i| \gg |\sum_{i=1}^{n} x_i|$, Kahan algorithm is not guaranteed to yield a small relative error.

2.2 SumK

SumK extends the idea of the error-free transformation for two floating-point numbers to floating-point number array. The working mechanism of SumK is to iterate $K-1$ applications of the array transformation to produce a result as if it is computed in K-fold working precision. We say the result is "as if computed in K-fold working precision", because it shares the same error estimated with first computing the sum of the array by K-fold working precision and then rounding the result back into the working precision, i.e., the result s' should satisfy $|s' - \sum_{i=1}^{n} x_i| \leq eps|\sum_{i=1}^{n} x_i| + (\varphi \cdot eps)^K \sum_{i=1}^{n} |x_i|$ with a constant φ. For example, assume the working precision is single precision in IEEE754 floating-point standard, and $K = 2$ which means the 2-fold working precision, the result is almost as accurate as computing the array sum in double precision and then rounding back to single precision.

SumK improves Kahan's algorithm in two aspects. First, the error-free transformation AddTwo in SumK for two floating-point numbers a and b does not require $|a| \geq |b|$. Second, SumK could iterate array transformation for more than one time to produce a more accurate result.

ALGORITHM: $s \leftarrow \text{SumK}(x, n, K)$

Input: x, the array of the given floating-point summands

n, the length of the array

K, it performs K-1 transformation iterations on x

Output: s, the correctly rounded sum of the array

1. for $k \leftarrow 1$ to $K-1$ // loops of error-free array transformations

2. $x \leftarrow \text{VecSum}(x)$;

3. $s \leftarrow 0$;

4. for $i \leftarrow 1$ to n // sum up the floating-point numbers in x

5. $s \leftarrow s + x_i$;

6. END

FUNCTION: $x' \leftarrow \text{VecSum}(x, n)$

Input: x, the array of the given floating-point summands

n, the length of the array

Output: x', the transformed array

1. for $i \leftarrow 2$ to n

2. $(x_i, x_{i-1}) \leftarrow \text{AddTwo}(x_i, x_{i-1})$;

3. END

2.2.1 Algorithm Description

Denote x' as the output array of Function VecSum. The array x is transformed by function VecSum without changing the sum, such that $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} x'_i$, and the last summand of the array is replaced by $fl(\sum_{i=1}^{n} x_i)$. Denote $s' = fl(\sum_{i=1}^{n} x'_i)$, $s = \sum_{i=1}^{n} x_i$ and $S = \sum_{i=1}^{n} |x'_i|$, we have

$$|s' - s| \leq eps|s| + \gamma_{n-1}^2 S, \quad where \ \gamma_n = \frac{n \cdot eps}{1 - n \cdot eps} \quad and \ n \cdot eps \ll 1$$

The proof was given by Rump in [7]. The inequality implies that the sum of the transformed array shares the same error estimated as if computed in doubled working precision and rounded back to working precision.

SumK applies VecSum on x for $K-1$ times, and then sums up the array by using standard floating-point addition arithmetic. After applying the transformation on the array x for $K-1$ times, the sum of the array satisfies for $K \geq 3$ [7],

$$|res - s| \leq (eps + 3\gamma_{n-1}^2)|s| + \gamma_{2n-2}^K S$$

Note that, the second term $\gamma_{2n-2}^K S$ reflects that the result is computed in K-fold precision, since $\gamma_{2n-2} \approx (2n-2)eps$ with $n \cdot eps \ll 1$. The term $3\gamma_{n-1}^2$ is negligible compared to eps. So the first term is approximated to $eps|s|$ which reflects that the result is rounded back into the working precision.

2.2.2 Properties

• SumK requires only ordinary addition and subtraction arithmetic in computer. It does not require extra working precision. Access to mantissa or exponent is not needed.

- $K-1$ is the number of applications of VecSum on the array x. K must be increased either with the number of summands n, or the condition number. For large enough K, the algorithm is guaranteed to produce a correct result. The author of SumK suggests a value of $K = 3$ for practical purposes. [7]

2.3 iFastSum

iFastSum is a typical iterative refinement algorithm. It applies AddTwo operations instead of the standard addition operations in computer. The local errors generated by addition operations are kept in the array instead of being discarded. The input array x acts as the storage for local errors generated by each refinement. The algorithm iterates refinements until the sum of errors is small enough. Then it performs a careful check on the errors and returns the correctly rounded sum.

ALGORITHM: $s \leftarrow \text{iFastSum}(x, n)$

Input: x, the array of the given floating-point summands

n, the length of the array

Output: s, the correctly rounded sum of the array

Global: r_c, indicates if a recursive call of iFastSum occurs, initially 0

1. $s \leftarrow 0$; $loop \leftarrow 1$; // *loop* counts the number of loops
2. for $i \leftarrow 1$ to n // accumulate first approximation

   (1) $(s, x_i) \leftarrow \text{AddTwo}(s, x_i)$;

3. loop forever // main loop
   (1) $count \leftarrow 1$; $s_t \leftarrow 0$; $loop \leftarrow loop + 1$; $s_m \leftarrow 0$;
      // $count$ points to the next position in x to store the local error
      // s_t is the temporary sum
   (2) for $i \leftarrow 1$ to n
      (a) $(s_t, x_{count}) \leftarrow \text{AddTwo}(s_t, x_i)$; // x_{count}: local error
      (b) if $x_{count} \neq 0$, then
         (i) $count \leftarrow count + 1$;
         (ii) $s_m \leftarrow \max(s_m, |s_t|)$;
   (3) $e_m \leftarrow (count - 1) \cdot \text{HalfUlp}(s_m)$;
   // each local error $\leq \text{HalfUlp}(s_m)$
   // e_m is the weak upper bound on magnitude of the sum of the errors
   (4) $(s, s_t) \leftarrow \text{AddTwo}(s, s_t)$; $s_t \leftarrow x_{count}$; $n \leftarrow count$;
   (5) if $e_m = 0$ or $e_m < \text{HalfUlp}(s)$, then
      (a) if $r_c > 0$, then return s; // return s if it is a recursive call
      (b) $(w_1, e_1) \leftarrow \text{AddTwo}(s_t, e_m)$;
      (c) $(w_2, e_2) \leftarrow \text{AddTwo}(s_t, -e_m)$;
      (d) if (for $j = 1, 2$) $\text{fl}(w_j + s) \neq s$ or $\text{Round3}(s, w_j, e_j) \neq s$, then
         (i) $r_c \leftarrow 1$; $s_1 \leftarrow \text{iFastSum}(x, n)$; // first recursive call
         (ii) $(s, s_1) \leftarrow \text{AddTwo}(s, s_1)$;
         (iii) $s_2 \leftarrow \text{iFastSum}(x, n)$; $r_c \leftarrow 0$; // second recursive call
         (iv) $s \leftarrow \text{Round3}(s, s_1, s_2)$;
      (e) return s;
4. END

FUNCTION: $R \leftarrow \text{Round3}(s_0, s_1, s_2)$

Input: s_0, s_1, s_2, the three floating-point numbers,

where $\text{fl}(s_0+s_1) = s_0$ and $\text{fl}(s_1+s_2) = s_1$

Output: r, correctly rounded $s_0+s_1+s_2$

1. if s_1 has the form of 1.0×2^e and $\text{Sign}(s_1) = \text{Sign}(s_2)$, then

   return $\text{fl}(1.1\times s_1+s_0)$; // magnify s_1 and add it to s_0

2. return s_0; // Sign(s) returns 1 if $x > 0$, 0 if $x = 0$, and -1 otherwise
3. END

2.3.1 Algorithm Description

In the following, we present a brief description of the working mechanism of iFastSum. More detailed descriptions and the proof of the algorithm correctness can be found in [6].

In Step 2, we first compute the sum of the floating-point numbers in the original array by accumulating x_i in s. We compute the sum of x_i by applying AddTwo operations instead of the standard floating-point addition operations in computer. The computed sum of the array is stored in the global sum s. All the local addition errors generated by AddTwo operations are put back into the original array x. Those local errors become summands in the array for later operations. Denoting the array after

step 2 as x', Step 2 is actually an error-free transformation on the original array since $\sum_{i=0}^{n} x_i = \sum_{i=0}^{n} x'_i + s$.

Step 3 is the main distillation loop. It repeats refinement until the global error bound e_m is small enough. The proof of the termination of loops can be found in [6].

In Step 3(2), it performs a refinement on the array x and computes the sum s_t of the array. During the traverse of array x from x_1 to x_n, it records the largest absolute value of s_t in s_m, which will be used in Step 3(3) to estimate the sum of local errors left in the array. After the refinement is done, in the worst case, every element left in the array is at most $\frac{1}{2}Ulp$ of s_m since the property of AddTwo suggests that in (a', b') = AddTwo(a, b), if $b < \frac{1}{2}Ulp(a)$ then $a' = a$ and $b' = b$. Note that it is different from Step 2, only nonzero errors are redistributed back into the array.

Step 3(3) estimates a weak upper bound e_m for the sum of the current array x, because after Step 3(2) all the errors left in array are at most $\frac{1}{2}ulp(s_m)$ due to the property of AddTwo operation. Subfunction HalfUlp(n) returns n'=$\frac{1}{2}ulp(n)$ if n' is representable by a floating-point number, otherwise it returns 0.

In Step 3(4), the temporary sum computed in Step 3(2) is added to the global sum s. The error of this operation is appended to the array. There is no out-of-boundary problem for the array x. Note that when $i = 1$ in Step 3(2a), x_{count} (where $count$ = 1) is zero because s_t is initialized to 0 in Step 3(1). Since count points to next position in array, we always have $count < n$ after Step 3(2) is executed. Therefore

the new length of the array x is less than n. So s_t can be appended into the array safely without out-of-boundary problem. The new length n of the array is updated for next refinement.

In Step 3(5), it first checks whether the error bound e_m computed in step 3(3) is small enough.
Case1: if the estimated sum of the array is not small enough, it will repeat the refinement.
Case2: if the estimated sum of the array is small enough, a careful check is performed to ensure the sum is correctly rounded. If the local error s_t and the estimated sum of error array e_m can not affect the global sum s, that is, the condition 3(5)(d) is not satisfied, then s is returned as the exact sum. Otherwise the exact sum will be represented by three floating point numbers with non-overlapping mantissa. Since the local errors are left in the array, two recursive calls of iFastSum are executed to compute the non-overlapping numbers. Function Round3 is used to compute the correctly rounded sum of three floating-point numbers with non-overlapping mantissa.

2.3.2 Properties

- iFastSum returns the correctly rounded sum of a floating-point numbers array.
- The accuracy of the sum is independent of condition number and the number of summands.
- iFastSum requires constant storage and its space complexity is $O(1)$.

- It doesn't require extra precision accumulators.

2.4 HybridSum

The main idea of HybridSum is as follows. It first creates large numbers of "buckets" and each bucket acts as an accumulator for summing up the floating-point numbers of a particular exponent. Each summand in the input floating-point number array x is "split" into two numbers of which each has half as many nonzero mantissa digits as the original summand. Therefore the accumulator can act as an extra-precision accumulator for the split numbers. After all of the summands are added to the corresponding accumulators, iFastSum is applied to sum up those accumulators. It is proved that if the number of summands entering an accumulator is less than a certain value, then no significant digits are discarded [6]. The limit number is N=$\beta^{\lfloor t/2 \rfloor}$ with $\beta = 2$ and $t = 53$ according to the IEEE754 standard.

ALGORITHM: $s \leftarrow \mathrm{HybridSum}(x, n)$

Input: x, the array of the given floating-point summands

n, the length of the array

Output: s, the correctly rounded sum of the array

Constant: $N = \beta^{\lfloor t/2 \rfloor}$

1. initialize two arrays, a_1 and a_2, each with β^l floating-point numbers

2. set all the numbers in a_1 to be zero

3. $i \leftarrow 1$; // i records the next summand in x we will process

4. $m \leftarrow \min(n, N)$;

// m records how many summands a_1 will process in each loop

5. loop forever // if $n > N$, $\lceil \frac{n-N}{N-\beta^l} \rceil + 1$ iterations; otherwise 1 iteration

   (1) for $k \leftarrow 1$ to m // add summands in x with a_1

      (a) $\{z_1, z_2\} \leftarrow \text{Split}(x_i)$; $i \leftarrow i + 1$;

      (b) $j \leftarrow \exp(z_1) + \beta^{l-1}$; $a_{1j} \leftarrow \text{fl}(a_{1j} + z_1)$;

      (c) $j \leftarrow \exp(z_2) + \beta^{l-1}$; $a_{1j} \leftarrow \text{fl}(a_{1j} + z_2)$;

         // add two split parts by the corresponding accumulator a_{1j}.

         // exp() returns the exponent of a floating-point number,

         // assume that the minimum exponent is $1 - \beta^{l-1}$.

   (2) $n \leftarrow n - m$; // n records the number of unprocessed summands in x

   (3) if $n = 0$, then go to Step6; // the ending condition

   (4) set all the numbers in a_2 to be zero;

   (5) for $k \leftarrow 1$ to β^l // add partial sums in a_1 with a_2

      (a) $\{z_1, z_2\} \leftarrow \text{Split}(a_{1k})$;

      (b) $j \leftarrow \exp(z_1) + \beta^{l-1}$; $a_{2j} \leftarrow \text{fl}(a_{2j} + z_1)$;

      (c) $j \leftarrow \exp(z_2) + \beta^{l-1}$; $a_{2j} \leftarrow \text{fl}(a_{2j} + z_2)$;

   (6) swap two array points, a_1 and a_2;

   (7) $m \leftarrow \min(n, N - \beta^l)$;

   // the newly initialized a_1 has already processed β^l summands

6. return iFastSum(a_1, β^l);

7. END

FUNCTION: $\{z_1, z_2\} \leftarrow \text{Split}(z)$

Input: z, the given floating-point number

Output: z_1 and z_2, split z into two floating-point numbers, each with half of z's mantissa.

1. $z_1 \leftarrow z$;
2. set the last $\lfloor (t+1)/2 \rfloor$ digits of the mantissa of z_1 be zero;
3. $z_2 \leftarrow z - z_1$;
4. return $\{z_1, z_2\}$;
5. END

2.4.1 Algorithm Description

There are β^l many distinct exponents in the working precision. Therefore β^l many accumulators need to be created. In Step 1, two arrays of accumulators are pre-allocated, where the size of each array is β^l. Step 2 initializes the accumulators in the array a_1 to zero and Step 3 initializes the variable i which is used to record the position of the next summand in the array x.

HybridSum does not limit the number of summands to enter each accumulator. Instead, it limits the total number of the summands entering the accumulator array in one loop. Therefore each accumulator is guaranteed to process less than N summands in one loop. If $n > N$, $\lceil \frac{n-N}{N-\beta^l} \rceil + 1$ loops are iterated for processing all summands in

the array x. In Step 4 the variable m is set to record how many summands for a_1 to process in a loop.

In Step 5(1), the m summands in array x are split and added to the corresponding accumulators. Step 5(2) updates the number of unprocessed summands in the array x. In Step 5(3), if all summands in the array x have been processed, it breaks the loop.

From Step 5(4) to Step 5(6), the numbers in a_1 are split again and distributed to the corresponding accumulator in the array a_2. The array a_2 is used to store the intermediate sums accumulated in a_1 in one loop. Then it swaps a_1 with a_2, and a_2 will be the accumulators array for the next iteration. In Step 5(7), since there are already β^l numbers in a_1, m is updated and it goes to next loop.

In Step 6, iFastSum is used to sum up all accumulators and return the correctly rounded sum.

2.4.2 Properties

- HybridSum returns the correctly rounded sum of a floating-point number array.

- HybridSum sums up numbers of a particular exponent by the corresponding accumulator, and afterwards sums up the accumulators by iFastSum. Therefore only

one traverse of the input array is required and the cost is $O(n)$. Thus the time complexity of HybridSum is $O(n)$ plus the cost for iFastSum to sum up the accumulators. When n is large, the cost for iFastSum to sum up the accumulators can be ignored. Therefore the time complexity of Hybridsum is $O(n)$ and it is independent of the condition number.

- Another advantage of HybridSum is its space complexity. No matter how many summands in the input array x, HybridSum only allocates 2 arrays with size β^l. Therefore the space complexity is $O(\beta^l)$.

Chapter 3

Numerical Experiments

In this chapter, we compare the performances of the following algorithms: ORS (Ordinary Recursive Summation), Kahan's algorithm, SumK ($K = 3$), iFastSum and HybridSum. We use the standard IEEE754 double precision arithmetic where $\beta = 2$, $l = 11$ and $t = 52$. All the algorithms are implemented in ANSI C and compiled by GCC 4.1.2. The programs are run on a server with four AMD 2.53GHz dual-core Processors and 8GB physical memory. The operating system of the server is X86 64bit Redhat Linux.

3.1 Test Data

The algorithms are tested on four kinds of data with size n=10,000,000. A parameter δ is used to denote the maximum exponent difference of the generated random

numbers. Different values of δ from 8 to 1800 are considered.

(1) Data Set No.1 is an extremely Ill-Conditioned data for which the exact sum equals zero. They are generated as the following: After a random number a is generated, it is put into the first half part of the data set. Then the number with the opposite sign $-a$ is put into the other half part of the data set.

(2) Data Set No.2 is a well-conditioned data where the random numbers are all positive. The condition number of the data computed by $R = \frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|}$ is 1.

(3) Data Set No.3 is the random numbers including signs.

(4) Data Set No.4 contains the Anderson's Ill-Conditioned data. They are generated by first generating random numbers including the signs, then subtracting the mean of the data from each number. This method was proposed by I. J. Anderson in [8]. The condition number of the transformed data is more than 10^9.

3.2 Test Results

In the test using Data Set No.1 (extremely ill-conditioned data) where the exact sum of the data eaquls zero, we list the exact results returned by the algorithms in Table 3.1. The Size of test data is 10,000,000. From Table 3.1 we observed that, ORS and Kahan's algorithm never return a correct sum. We denote SumK ($K = 3$) as Sum3 in the following. When δ is greater than 64, Sum3 fails to return the correct sums. However, iFastSum and HybridSum always return the correct sums. Therefore, iFastSum and HybridSum are both reliable algorithms to return the correctly rounded sum of the testing data. Thus in the tests using Data Set No.2, No.3 and No.4, we use

Data Set No.1: Extremely Ill-Conditioned Data						
δ	R	ORS	Kahan	Sum3	iFastSum	HybridSum
8	1.0723e+17	5.5723e-10	4.8850e-14	0.000e+00	0.0000e+00	0.0000e+00
32	8.1573e+16	-7.5298e-07	-2.1270e-11	0.000e+00	0.0000e+00	0.0000e+00
64	5.4110e+16	-3.7190e-02	-4.9950e-06	0.000e+00	0.0000e+00	0.0000e+00
128	9.8596e+15	4.3985e+08	-5.3024e+04	-5.7260e-19	0.0000e+00	0.0000e+00
256	7.1814e+16	5.5912e+26	-7.0789e+21	7.8125e+00	0.0000e+00	0.0000e+00
512	2.0588e+16	3.3298e+65	7.7133e+61	3.5092e+38	0.0000e+00	0.0000e+00
1024	3.2686e+16	-1.2231e+142	1.3489e+139	-3.7324e+114	0.0000e+00	0.0000e+00
1800	2.7906e+17	1.7946e+258	7.8230e+255	-2.4137e+231	0.0000e+00	0.0000e+00

Table 3.1: Test results (returned sums) of the algorithms on the extremely ill-conditioned data

the results returned by HybridSum as the correct results, and list the relative errors of the results returned by all algorithms. The Size of test data is 10,000,000. The results are listed in Table 3.2, Table 3.3 and Table 3.4. We also list the exact sums of the data sets using the results returned by HybridSum in Table 3.5. As we know, the IEEE754 double precision is about 16 decimal digits. Due to the limit of space, we can not print all the 16 decimal digits of the results. Thus listing the relative errors of the results is more illustrative for comparison.

From Table 3.2 to Table 3.4 we observed that in the testing of Data Set No.2 - the well-conditioned random data with $R = 1$, ORS never returns the correctly rounded sums. The relative errors of the results returned by ORS have the magnitudes around 10^{-13}. In our test using Data Set No.3, the numbers are random including the signs. In this case, ORS never returns the reliable results. Kahan's algorithm fails to return the correctly rounded sums in the case of δ is 256 and 512. However, the relative errors of Kahan's results have the magnitude about 10^{-16}, which is within the machine

Data Set No.2: Positive Random Data						
δ	R	ORS	Kahan	Sum3	iFastSum	HybridSum
8	1.0000e+00	2.3678e-15	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
32	1.0000e+00	4.5310e-14	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
64	1.0000e+00	6.0914e-12	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
128	1.0000e+00	3.1221e-12	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
256	1.0000e+00	1.4773e-12	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
512	1.0000e+00	7.5036e-13	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
1024	1.0000e+00	4.0242e-13	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
1800	1.0000e+00	3.7735e-13	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00

Table 3.2: Test results (relative errors) of the algorithms on well-conditioned data

Data Set No.3: Random including sign						
δ	R	ORS	Kahan	Sum3	iFastSum	HybridSum
8	1.6974e+03	7.3953e-14	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
32	9.3762e+02	4.9218e-14	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
64	4.7283e+02	2.1110e-14	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
128	3.3738e+02	3.8770e-14	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
256	2.3632e+02	1.3738e-15	1.1449e-16	0.0000e+00	0.0000e+00	0.0000e+00
512	1.8378e+02	2.3644e-14	1.7778e-16	0.0000e+00	0.0000e+00	0.0000e+00
1024	1.2234e+02	1.5986e-14	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00
1800	1.6478e+02	1.5228e-14	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00

Table 3.3: Test results (relative errors) of the algorithms on random numbers including signs

precision. In our test using Data Set No. 4 - the Anderson's Ill-Conditioned data with the condition numbers greater than 10^9, only Sum3, iFastSum and HybridSum always return the correctly rounded sums.

Next we present the timing results. From Table 3.6 to Table 3.9 we list the time cost by each algorithm to return the results. We display the absolute value of the time in the unit of millisecond, and also the ratio in parenthesis with the time of HybridSum

Data Set No.4: Anderson's Ill-Conditioned Data						
δ	R	ORS	Kahan	Sum3	iFastSum	HybridSum
8	3.0678e+16	1.1219e-01	2.0883e-04	0.0000e+00	0.0000e+00	0.0000e+00
32	1.5902e+16	1.6960e-01	6.8941e-04	0.0000e+00	0.0000e+00	0.0000e+00
64	4.4333e+13	1.1584e+03	7.5702e-03	0.0000e+00	0.0000e+00	0.0000e+00
128	1.5655e+13	6.0016e+02	5.1350e-03	0.0000e+00	0.0000e+00	0.0000e+00
256	1.5962e+13	1.2296e+03	6.6594e-03	0.0000e+00	0.0000e+00	0.0000e+00
512	8.1574e+12	1.1522e+03	9.5511e-04	0.0000e+00	0.0000e+00	0.0000e+00
1024	5.0964e+12	2.2227e+03	4.3694e-03	0.0000e+00	0.0000e+00	0.0000e+00
1800	1.7800e+12	4.3102e+03	1.6377e-03	0.0000e+00	0.0000e+00	0.0000e+00

Table 3.4: Test results (relative errors) of the algorithms on Anderson's Ill-Conditioned Data

The exact sums of the four testing data sets				
δ	Data Set No.1	Data Set No.2	Data Set No.3	Data Set No.4
8	0.0000e+00	5.9786e+07	3.5222e+04	2.1951e-09
32	0.0000e+00	6.1460e+10	6.5548e+07	4.6570e-06
64	0.0000e+00	2.0125e+15	4.2560e+12	-3.9283e-02
128	0.0000e+00	4.3251e+24	1.2820e+22	-4.6077e+08
256	0.0000e+00	3.9925e+43	1.6895e+41	-2.0405e+27
512	0.0000e+00	6.8042e+81	3.7024e+79	7.2833e+65
1024	0.0000e+00	3.9660e+158	3.2419e+156	3.5276e+142
1800	0.0000e+00	4.9906e+275	3.0290e+273	6.5462e+259

Table 3.5: The exact sums of the four testing data sets with different δ values

Data Set No.1: Extremely Ill-Conditioned Data						
δ	R	ORS	Kahan	Sum3	iFastSum	HybridSum
8	1.0723e+17	0.06 (0.10)	0.17 (0.27)	0.91 (1.47)	0.76 (1.23)	0.62 (1.00)
32	8.1573e+16	0.06 (0.10)	0.17 (0.27)	0.91 (1.47)	0.77 (1.24)	0.62 (1.00)
64	5.4110e+16	0.06 (0.10)	0.17 (0.27)	0.91 (1.47)	1.00 (1.61)	0.62 (1.00)
128	9.8596e+15	0.06 (0.10)	0.17 (0.27)	0.92 (1.48)	1.37 (2.21)	0.62 (1.00)
256	7.1814e+16	0.06 (0.10)	0.17 (0.27)	0.90 (1.46)	2.17 (3.50)	0.62 (1.00)
512	2.0588e+16	0.06 (0.10)	0.17 (0.27)	0.91 (1.47)	3.70 (5.97)	0.62 (1.00)
1024	3.2686e+16	0.05 (0.08)	0.17 (0.27)	0.91 (1.47)	6.51 (10.50)	0.62 (1.00)
1800	2.7906e+17	0.05 (0.08)	0.17 (0.27)	0.91 (1.47)	10.74 (17.32)	0.62 (1.00)

Table 3.6: Test timing of the algorithms on the extremely ill-conditioned data

is normed to 1. From Table 3.6 to Table 3.9 we observed that when the size of data are the same, the running time of ORS, Kahan's algorithm and HybridSum are independent of δ. The running time of Sum3 is also independent of δ. However, when the extremely ill-conditioned data is used, Sum3 starts to return the incorrectly rounded sum when $\delta > 64$. In order to return the correctly rounded sum by SumK, the parameter K of SumK which represents the number of iterations of the refinement on the data has to be increased. Table 3.10 lists the least value of K and the running time for SumK to return the correctly rounded sum of the extremely ill-conditioned data, starting from $\delta > 64$. From Table 3.6 we also observed that when the extremely ill-conditioned data is used, the running time of iFastSum is linear to δ. Comparing Table 3.6 with Table 3.10, SumK and iFastSum both generate the same correctly rounded results when extremely ill-conditioned data is used, but iFastSum is faster than SumK.

Data Set No.2: Positive Random Data						
δ	R	ORS	Kahan	Sum3	iFastSum	HybridSum
8	1.0000e+00	0.06 (0.10)	0.18 (0.29)	0.93 (1.50)	0.79 (1.27)	0.62 (1.00)
32	1.0000e+00	0.07 (0.11)	0.17 (0.27)	0.93 (1.50)	0.81 (1.31)	0.62 (1.00)
64	1.0000e+00	0.06 (0.10)	0.18 (0.29)	0.93 (1.50)	0.90 (1.45)	0.62 (1.00)
128	1.0000e+00	0.06 (0.10)	0.17 (0.27)	0.92 (1.48)	0.92 (1.48)	0.62 (1.00)
256	1.0000e+00	0.07 (0.11)	0.17 (0.27)	0.94 (1.49)	0.91 (1.44)	0.63 (1.00)
512	1.0000e+00	0.06 (0.10)	0.17 (0.27)	0.94 (1.52)	0.91 (1.47)	0.62 (1.00)
1024	1.0000e+00	0.06 (0.10)	0.17 (0.27)	0.92 (1.52)	0.91 (1.47)	0.62 (1.00)
1800	1.0000e+00	0.06 (0.10)	0.17 (0.27)	0.93 (1.50)	0.90 (1.45)	0.62 (1.00)

Table 3.7: Test timing of the algorithms on the well-conditioned data

Data Set No.3: Random including sign						
δ	R	ORS	Kahan	Sum3	iFastSum	HybridSum
8	1.6974e+03	0.06 (0.10)	0.17 (0.27)	0.93 (1.50)	0.80 (1.29)	0.62 (1.00)
32	9.3762e+02	0.06 (0.10)	0.17 (0.27)	0.93 (1.48)	0.79 (1.25)	0.63 (1.00)
64	4.7283e+02	0.06 (0.10)	0.17 (0.27)	0.93 (1.50)	0.88 (1.42)	0.62 (1.00)
128	3.3738e+02	0.07 (0.11)	0.17 (0.27)	0.94 (1.49)	0.90 (1.43)	0.63 (1.00)
256	2.3632e+02	0.07 (0.11)	0.17 (0.27)	0.97 (1.52)	0.92 (1.44)	0.64 (1.00)
512	1.8378e+02	0.06 (0.10)	0.17 (0.27)	0.93 (1.50)	0.90 (1.45)	0.62 (1.00)
1024	1.2234e+02	0.06 (0.10)	0.17 (0.27)	0.94 (1.52)	0.92 (1.48)	0.62 (1.00)
1800	1.6478e+02	0.06 (0.09)	0.17 (0.27)	0.93 (1.45)	0.90 (1.41)	0.64 (1.00)

Table 3.8: Test timing of the algorithms on the random numbers including signs

Data Set No.4: Anderson's Ill-Conditioned Data						
δ	R	ORS	Kahan	Sum3	iFastSum	HybridSum
8	3.0678e+16	0.06 (0.10)	0.17 (0.27)	0.93 (1.50)	0.79 (1.27)	0.62 (1.00)
32	1.5902e+16	0.05 (0.08)	0.17 (0.27)	0.93 (1.48)	0.80 (1.27)	0.63 (1.00)
64	4.4333e+13	0.05 (0.08)	0.18 (0.28)	0.92 (1.44)	0.80 (1.25)	0.64 (1.00)
128	1.5655e+13	0.06 (0.10)	0.17 (0.27)	0.93 (1.48)	0.80 (1.27)	0.63 (1.00)
256	1.5962e+13	0.06 (0.10)	0.17 (0.27)	0.93 (1.50)	0.79 (1.27)	0.62 (1.00)
512	8.1574e+12	0.06 (0.10)	0.17 (0.27)	0.93 (1.48)	0.80 (1.27)	0.63 (1.00)
1024	5.0964e+12	0.06 (0.10)	0.17 (0.27)	0.92 (1.46)	0.80 (1.27)	0.63 (1.00)
1800	1.7800e+12	0.06 (0.10)	0.16 (0.27)	0.93 (1.48)	0.79 (1.25)	0.63 (1.00)

Table 3.9: Test results of the algorithms on the Anderson's Ill-Conditioned Data

Test SumK on Extremely Ill-Conditioned Data			
δ	K	Time	Sum
128	5	1.79	0.000e+00
256	7	2.66	0.000e+00
512	13	5.30	0.000e+00
1024	24	10.04	0.000e+00
1800	40	17.09	0.000e+00

Table 3.10: The least K and Running Time for SumK to produce the correct results on the extremely ill-conditioned data

3.3 Accuracy

From the test results we observed that, ORS never returns a correctly rounded sum for any kind of testing data. Kahan's algorithm fails to return reliable results when the data is badly ill-conditioned. SumK (K=3) fails to return the correctly rounded sum when the extremely ill-conditioned data is used and $\delta > 64$. In this case, in order to return the correctly rounded sum by SumK, the parameter K of SumK which represents the number of iterations of the refinement on the data has to be increased. The running time of SumK is also increased. However, the exact value of the parameter K is not easy to be known before SumK is executed. Thus when the data is extremely ill-conditioned, iFastSum and HybridSum are recommended to be used to return reliable results. All of ORS, Kahan's Algorithm, SumK, iFastSum and HybridSum return correctly rounded results when the data is denormalized numbers.

Knuth's AddTwo and Dekker's FastTwoSum are both valid in the presence of underflow since in the algorithms they only used additions and subtractions, and we

know that addition and subtraction are exact when the denormalized number is included in the system. Therefore, all of the summation algorithms we presented by now are valid in the presence of underflow since they only apply addition and subtraction on the data.

3.4 Running Time

In the timing comparisons, we only consider the algorithms returning the correctly rounded sums. The running time of Kahan's algorithm is independent of δ. When the condition number of the data is less than 100, Kahan's algorithm returns the correct result and its timing performance outperforms Sum3, iFastSum and HybridSum.

When more badly ill-conditioned data is used, ORS and Kahan's algorithm never return reliable results. In order to return the correctly rounded sum, the value of K as well as the running time of SumK have to be increased. From Table 3.6 and Table 3.10 we observed that iFastSum is faster than SumK although they both return the same results.

Note that for Data Set No. 4 — Anderson's Ill-Conditioned data, δ listed in the table is the original δ of the data before Anderson transformation is performed. The real δ of the data after Anderson transformation is about 10^{15} no matter what the original δ is. Thus the running time of iFastSum for Data Set No.4 is independent to δ.

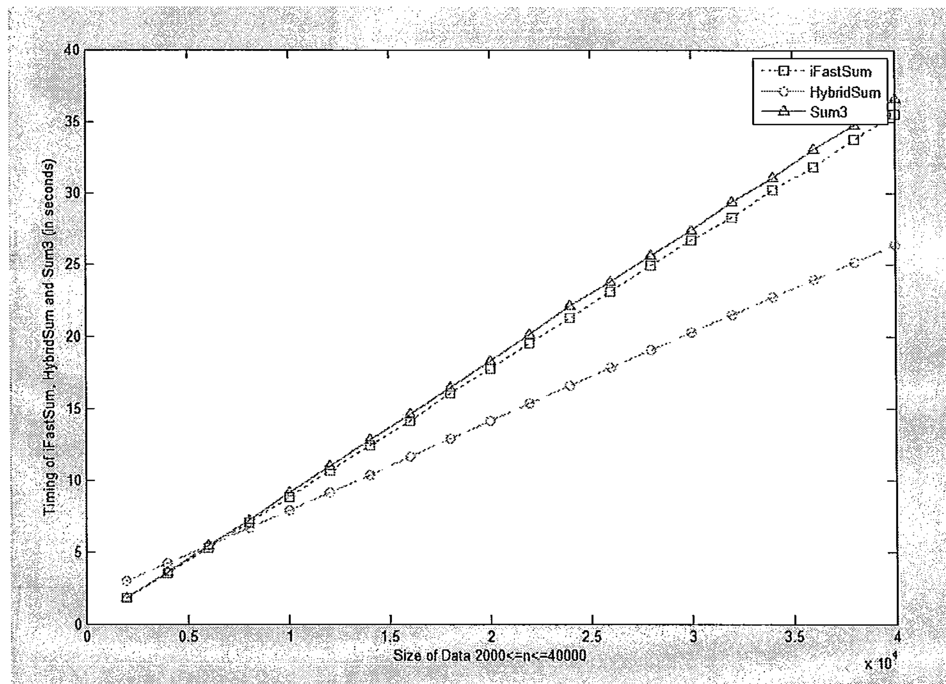

Figure 3.1: Comparison of Timing Results of iFastSum, HybridSum and Sum3

From the results we observed that HybridSum always return reliable results independent of condition number and the value of δ. The author of iFastSum and HybridSum suggests that when the size of data $n < 20,000$, iFastSum is recommended; when the size of data $n > 20,000$, HybridSum is recommened [6]. We also compare the timing performance of iFastSum, HybridSum and Sum3 on different sizes of data and the result is represented in Figure 3.1. We use the random numbers including signs with the difference of exponent $\delta = 1800$, which is the most general situation. The vector length $2000 \leq n \leq 40000$ and the number of samples is 10000. From Figure 3 we observe that the timing costs of iFastSum, HybridSum and Sum3 increase linearly

with the increasing of the data size. When $n < 8000$, iFastSum is faster than HybridSum; when $n \geq 8000$, HybridSum is faster than iFastSum. Sum3 is always a little slower than iFastSum. However, when the extremely ill-conditioned data is used, the parameter K of SumK needs to be increased and as the consequence it is much slower than iFastSum. Therefore for pratical uses, we suggest using iFastSum when the data size is less than 8000, and HybridSum when the data size is larger than 8000.

3.5 Space Complexity

Assume the input data for the algorithms can not be changed (immutable). ORS and Kahan's algorithm require no extra space because they only read data from the array. SumK and iFastSum need $O(n)$ space because they will modify the input array. HybridSum requires $O(\beta^l)$ space (β^l is 2048 when using the standard IEEE754 double precision) for the accumulators. When iFastSum is called by HybridSum to sum up the accumulators, the size of the input array for iFastSum is β^l. Thus the space for HybridSum is constant. Therefore if the data amount is too huge for the physical memory, or the input data comes from a data stream of arbitrary length, HybridSum is recommended.

Chapter 4

Application to LAPACK subroutines

4.1 Vector Dot Product

In this chapter, we apply the summation algorithms ORS, Kahan's Algorithm, Sum3, iFastSum and HybridSum for computing vector dot product, and then compare their timing and accuracy results with the subroutine DDOT of LAPACK.

The calculations in this chapter satisfy the IEEE754 double precision standard and the working precision is about 16 decimal digits. The programs were run on a laptop with Intel 1.6GHz CPU and 1.00GB physical memory. The operation system is Windows 7. All the comparisons of timing and accuracy were done in 32-bit MATLAB2010a.

In [4], T. J. Dekker presented an error-free transformation method.

ALGORITHM: $[x, y] = \text{TwoProduct}(a, b)$

Input: a, b, a pair two floating-point numbers

Output: x, y, the pair of transformed floating-point numbers

1. $x = fl(a \cdot b)$;
2. $[a_1, a_2] = \text{Split}(a)$;
3. $[b_1, b_2] = \text{Split}(b)$;
4. $y = fl(a_2 \cdot b_2 - (((x - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2))$;

This method transforms the product of a pair of floating-point numbers $[a, b]$ into the sum of another pair $[x, y]$ with $a \cdot b = x + y$ and $x = fl(a \cdot b)$ in case no underflow occurs. Denote $eta = 2^{-1074}$ in IEEE754 double precision standard. In case of underflow occurs in any of the five multiplications of TwoProduct, denote the results by x' and y', Rump proved that the error of transforming product to sum is at most $5eta$, that is $|y - y'| \leq 5eta$ [7]. Therefore, in case of a and b are both denormalized numbers, underflow occurs and the error of the transformation is 5eta. Since underflow is rare and 5eta is very small, the error is negligible.

As well known, the dot product of two vectors x and y is calculated by $x^T y = \sum_{i=1}^{n}(x_i \cdot y_i)$. With the Algorithm TwoProduct, Rump extended the idea of error-free transformation for the product of two floating-point numbers to vector dot products of arbitrary length in [7]. Therefore the calculation for the product of vector x and y is transformed into the calculation for the sum of an array r which is a transformation

of vector x and y. That is, $\sum_{i=1}^{2n} r_i = x^T y$. Combining this with the highly accurate summation algorithms we presented in the previous chapter, we expect it would yield some highly accurate dot product algorithms. Dot product algorithms are widely applied in numerical linear algebra, e.g., in computing the product of matrices, in the iterative refinement of the solution of $Ax - b$.

ALGORITHM: $d \leftarrow \text{Ddot}(x, y)$

Input: x, y, two vectors of floating-point numbers

Output: d, the dot product of vector x and y

1. for $i \leftarrow 1$ to n
2. $[r_i, r_{n+i}] = \text{TwoProduct}(x_i, y_i)$
3. $d = $ Sum of r.
4. return d;
5. END

Our goal is to apply the different summation algorithms ORS, Kahan's Algorithm, Sum3, iFastSum and HybridSum in summing up the array r in Step3 of Algorithm Ddot. We also compare their performance with the LAPACK subroutine DDOT. LAPACK is an acronym for Linear Algebra PACKage. It is a library of Fortran 77 subroutines for solving the most commonly occurring problems in numerical linear algebra. DDOT of LAPACK is a subroutine for computing the dot product of two vectors.

In order to test the performance of those Ddot programs combining different approaches, we need vector dot products with different condition numbers which is defined by $\text{cond}(x^Ty)=2 \cdot |x|^T|y|/|x^Ty|$. In [7], Rump presented an algorithm GenDot for generating extremely ill-conditioned dot products. The MATLAB implementation of this algorithm can be found in INTLAB [9].

FUNCTION: $[x, y, d, C] = \text{GenDot}(n, c)$

Input:	n,	dimension of vectors x, y, $n >= 6$
	c,	the anticipated condition number of x^Ty
Output:	x, y,	generated vectors
	d,	dot product x^Ty rounded to nearest
	C,	actual condition number of x^Ty

$n2 = \text{round}(n/2)$; // Initialization

$x = \text{zeros}(n, 1)$;

$y = x$;

$b = log_2(c)$;

$e = \text{round}(\text{rand}(n2, 1)*b/2)$; // e is a vector of exponents between 0 and $b/2$

$e(1) = \text{round}(b/2) + 1$; // ensure exponents $b/2$ and 0 actually occur

$e(end) = 0$;

$x(1 : n2) = (2*\text{rand}(n2, 1) - 1).*(2^e)$; // generate first half of vectors x,y

$y(1 : n2) = (2*\text{rand}(n2, 1) - 1).*(2^e)$;

$e = \text{round}(\text{linspace}(b/2, 0, n - n2))$;

// generate exponents for second half of vectors x, y

```
for i = n2 + 1 : n
    x(i) = (2*rand-1)*2^(e(i-n2));
    y(i) = ((2*rand-1)*2^(e(i-n2))-DocExact(x', y))/x(i);
end
index = randperm(n);            // generate random permutation for x, y
x = x(index);                   // permute x and y
y = y(index);
d = DotExact(x', y);            // the true product rounded to nearest
C = 2*(abs(x')*abs(y))/abs(d);  // the actual condition number
```

The inputs of Algorithm GenDot are the size of the expected vector x and y as well as the anticipated condition number of the vector dot product. The outputs of the algorithm are the generated vectors, the exact dot product of the vectors and the true condition number of the vector dot product. The algorithm requires the existing of a subroutine DotExact which produces a floating-point number close to the exact value of the dot product x^Ty. Rump used some highly accurate floating-point arithmetic and algorithm DotK with suitably chosen K to implement DotExact. The algorithm GenDot was carefully designed to ensure the vectors are general and not following any obvious patterns [7]. The main idea of GenDot is as following. In order to create two vectors with extremely ill-conditioned dot product, the entries of the vectors must cause heavy cancellation. The expected condition number of the dot product x^Ty is proportional to the degree of cancellation. The algorithm generates the first half parts of vector x and y randomly within a large exponent range. The exponent range is chosen according the expected condition number. Then the elements in the

second half of the vector x are generated randomly with decreasing exponent, and the corresponding y_i is generated which will cause some cancellations. Finally, the elements in vectors x and y are permuted randomly and the real condition number of the vector x and y is calculated. More details can be found in [7].

Since we have already implemented the different approaches of summation algorithms ORS, Kahan's algorithm, Sum3, iFastSum and HybridSum in ANSI C in the previous chapter, we take the advantage of MEX file which is an interface between C and MATLAB. Thus those C programs can be called in MATLAB and the corresponding dot product programs can be compared with using the data generated from GenDot. Although the most popularly used implementation of DDOT in LAPACK is in Fortran 77 and the MEX also has the functionality to call functions written in Fortran, we notice that the programs written in Fortran and C may have different performances in the same environment. For a fair comparison with DDOT of LAPACK, we choose the C version implementation of DDOT from [10], and omit its extra functionalities like increment etc.

For the comparisons, we use GenDot to generate 1000 test cases where each test case contains vector x and y of length 100. The condition numbers of the vector dot products are within the range of $(1, 10^{100})$. In our laptop with Intel 1.6GHz CPU and 1.00GB physical memory, it costs about one hour in using GenDot to generate the data set. Either increasing the anticipated condition number or the lengths of the vectors would increase the time costed by GenDot. The condition number in the range of $(1, 10^{100})$ includes the most situations we could meet. Since the dot product

of two length n vectors can be error-freely transformed into the summation of a length $2n$ vector, and based on the numerical results of the summation algorithms we know that the accuracy of the dot product algorithms would be independent of the vector length. However the timing performance would be related to the vector length and the time will increase linearly with the increasing of the vector length. Due to the time limitation, we compare the performance of the dot product algorithms on the vectors of different condition numbers but with the fixed length 100.

GenDot also returns the exact value of the dot product and the condition number of the dot product for every test case. We test every dot product method and compute its relative error by: $|d-x^Ty|/|x^Ty|$, where d is the result of the dot product computed by one method and x^Ty is the real dot product returned by GenDot when generating the vectors. The following four figures are the test results by using the plot function in MATLAB.

We denote the ordinary DDOT subroutine of LAPACK as Ddot_LAPACK, and the combining of dot product transformation with summation algorithm ORS as Ddot_ORS. Similarly, Ddot_Kahan, Ddot_Sum3, Ddot_iFastSum and Ddot_HybridSum follow the same naming rules. We tested them on the data where Vector Length = 100 and Number of Samples = 1000. For creating the more illustrative figures, we set the relative error to 2 if it is greater than 2, since the result is useless if the relative error is greater than 2. The following figures present the performance of dot product algorithms with different approaches. We plot the relative error against the actual condition number of the test data.

From Figure 4.1, Figure 4.2 and Figure 4.3, we observe that with the exponentially increasing of dot product condition number, the relative error $|d-x^Ty|/|x^Ty|$ computed by Ddot_ORS, Ddot_Kahan and Ddot_LAPACK increase exponentially. It seems like that the results returned by Ddot_ORS, Ddot_Kahan and Ddot_LAPACK share the same error estimated. However, Ddot_Kahan performs better than Ddot_ORS and Ddot_LAPACK. Figure 4.8, Figure 4.9 and Figure 4.10 illustrate the testing results of Ddot_ORS, Ddot_Kahan and Ddot_LAPACK on the data with maximum condition number of 10^{20}. From Figure 4.8 and Figure 4.9 we observed that when the condition number is greater than 10^{16}, the relative errors of the results returned by Ddot_ORS and Ddot_LAPACK exceed 2. However, from Figure 4.10 we observed that when the condition number is greater than 10^{17}, the relative errors of the results returned by Ddot_Kahan exceed 2. In another word, Ddot_Kahan is more accurate than Ddot_ORS and Ddot_LAPACK.

From Figure 4.4 we observe that when the condition number is less than 10^{30}, the relative errors of Ddot_Sum3 are zeros. Note that we also checked the numerical results to ensure the relative errors are exactly zeros. We did not list the numerical results due to the space limitation. From Figure 4.5 we observe that when the condition number is greater than 10^{30}, the relative error computed by Ddot_Sum3 increase exponentially with the increasing of dot product condition number. When the condition number is greater than 10^{50}, the relative errors always exceed 1. However, from Figure 4.6 and Figure 4.7 we observe that Ddot_iFastSum and Ddot_HybridSum always return the results with relative errors equal to 0 independent of condtion number. We

also checked the numerical results to ensure all the relative errors are exactly zeros.

Next we present the timing result of each method. Table 4.1 lists the average time for each method to process one test case. The number of samples = 1000, Vector Length = 100 and Maximum Condition Number = 10^{100}.We display the absolute value of the time in the unit of millisecond, and also the ratio in parentheses with the time of Ddot_LAPACK is normed to 1.

Algorithms	Time
Ddot_Lapack	27.5 (1.00)
Ddot_ORS	60.2 (2.19)
Ddot_Kahan	75.1 (2.73)
Ddot_Sum3	110 (4.00)
Ddot_iFastSum	205.6 (7.48)
Ddot_HybridSum	525.8 (19.12)

Table 4.1: Measured computing time for different dot product algorithms with Ddot_LAPACK normed to 1

From the observations, we briefly summarize the algorithms for vector dot product. As expected, the dot product algorithms have the similar properties with the corresponding summation algorithms. When the condition numbers of the test data are varying between 0 and 10^{18}, the relative errors of the computed results returned by Ddot_LAPACK, Ddot_ORS and Ddot_Kahan increase exponentially with the increasing of condition number; When the condition number is greater than 10^{18}, the relative errors exceed 1. Ddot_iFastSum and Ddot_HybridSum always return the computed results with the relative errors equal to zeros. In other word, Ddot_iFastSum and Ddot_HybridSum return the exact results of the vector dot

product independent of condition number. Therefore when the accuracy of the result is an issue, Ddot_iFastSum and Ddot_HybridSum are recommended to return reliable results. From the timing comparisons we observe that although Ddot_iFastSum and Ddot_HybridSum return reliable results, they cost more time than the other approaches. Therefore when the vector dot products are not ill-conditioned, small relative errors are allowable, and time cost is an issue, Ddot_Kahan is recommended. From the comparisons of timing and accuracy, we also found that Ddot_Sum3 is a compromise between unreliable and reliable algorithms. It returns reliable results when the condition number is smaller than 10^{30}. When the condition number is within $(10^{30}, 10^{50})$, the relative error is smaller than 1. When the condition number is greater than 10^{50}, it returns unreliable results.

4.2 Vector-Matrix and Matrix-Matrix multiplications

DGEMV is the LAPACK subroutine for computing vector and matrix multiplications. Its extra functionalities like unequal index increasement are not in our interest, thus for better demonstration, we give the simplified algorithm as follows.

ALGORITHM: $y = \text{DGEMV}(\alpha, A, m, n, x, y, \beta)$

Input: α, the scalar of A

A, a $m \times n$ matrix

m, the first dimension of matrix A

n, the second dimension of matrix A

x, the $n \times 1$ vector

y, the $m \times 1$ vector

β, the scalar of y

Output: y, where $y = \alpha * A * x + \beta * y$

1. $y = \beta * y$;
2. for $i \leftarrow 1$ to n
3. $\quad temp = \alpha * x[i]$;
4. $\quad$ for $j \leftarrow 1$ to m
5. $\quad\quad y_j = y_j + temp * A[j][i]$;
6. END

We can modify this algorithm by using the accurate vector dot product algorithms presented in Section 4.1 as follows.

ALGORITHM: $y =$ DGEMV_ACCURATE$(\alpha, A, m, n, x, y, \beta)$

Input: α, the scalar of A

A, a $m \times n$ matrix

m, the first dimension of matrix A

n, the second dimension of matrix A

x, the $n \times 1$ vector

y, the $m \times 1$ vector

β, the scalar of y

Output: y, where $y = \alpha * A * x + \beta * y$

1. for $i \leftarrow 1$ to m
2. $y_i = \beta * y_i + \alpha * Ddot(x, A[i]);$
3. END

Due to the time limitation, we did not implement the algorithm. However, based on the experience of implementing the DDOT and the comparisons of numerical results, as well as the simple structure of the algorithm DGEMV, we expect using our accurate dot product algorithms would improve the accuracy of computing the vactor-matrix multiplication. DGEMM is the LAPACK subroutine for computing matrix and matrix multiplications, we also expect it would benefit from the accurate vector dot product algorithms.

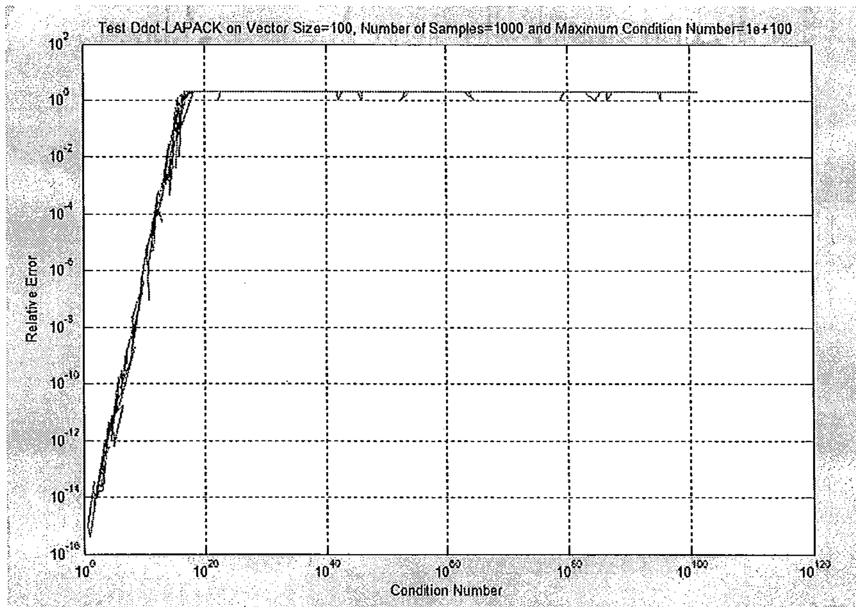

Figure 4.1: Test Results for Ddot_Lapack

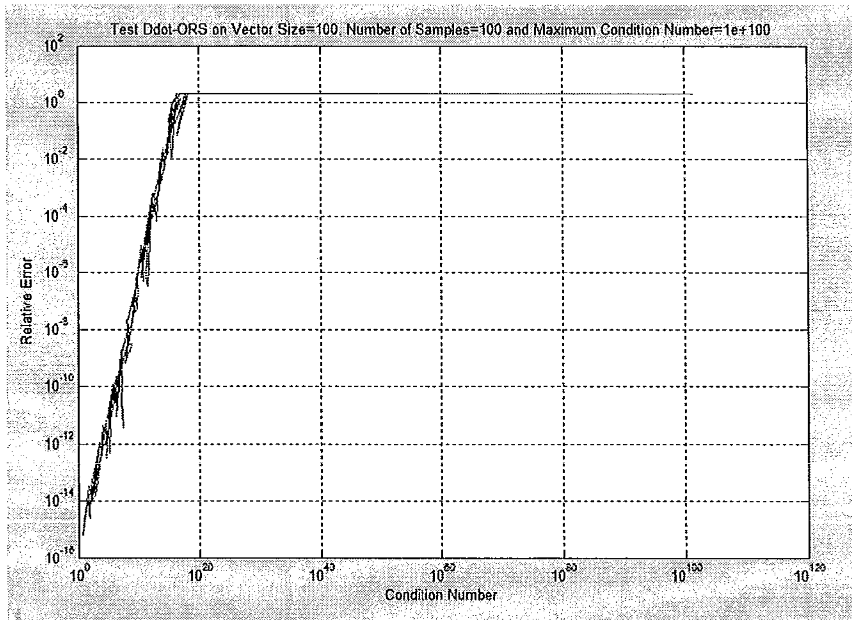

Figure 4.2: Test Results for Ddot_ORS

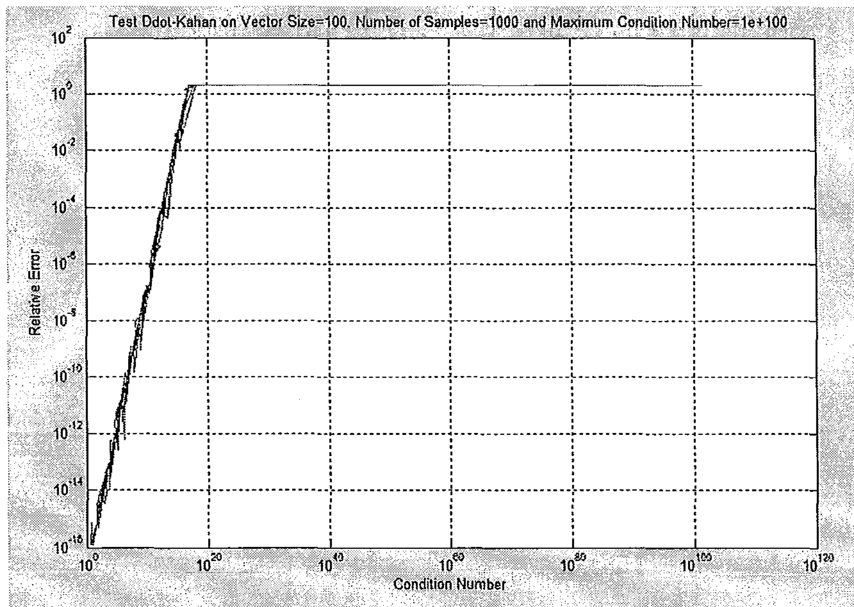

Figure 4.3: Test Results for Ddot_Kahan

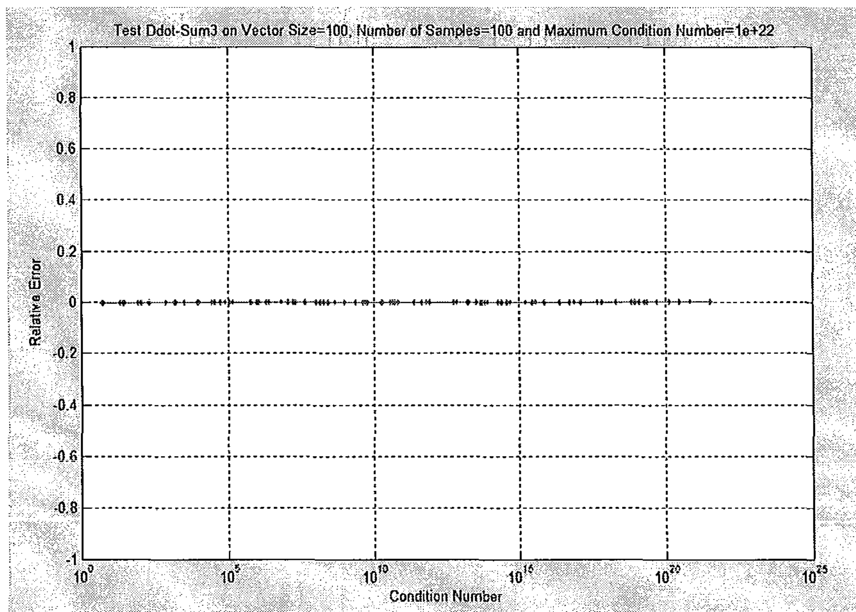

Figure 4.4: Test Results for Ddot_Sum3 (Condition Number $\leq 10^{30}$)

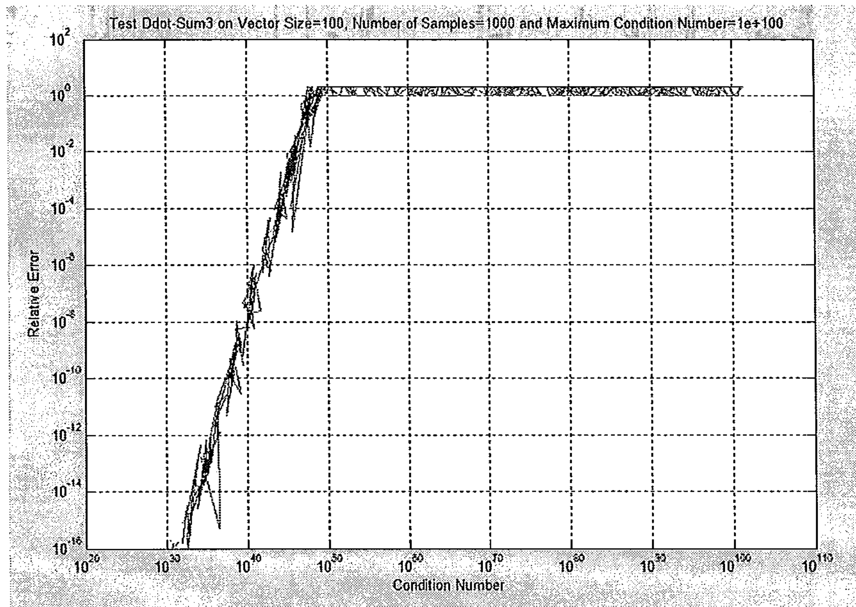

Figure 4.5: Test Results for Ddot_Sum3

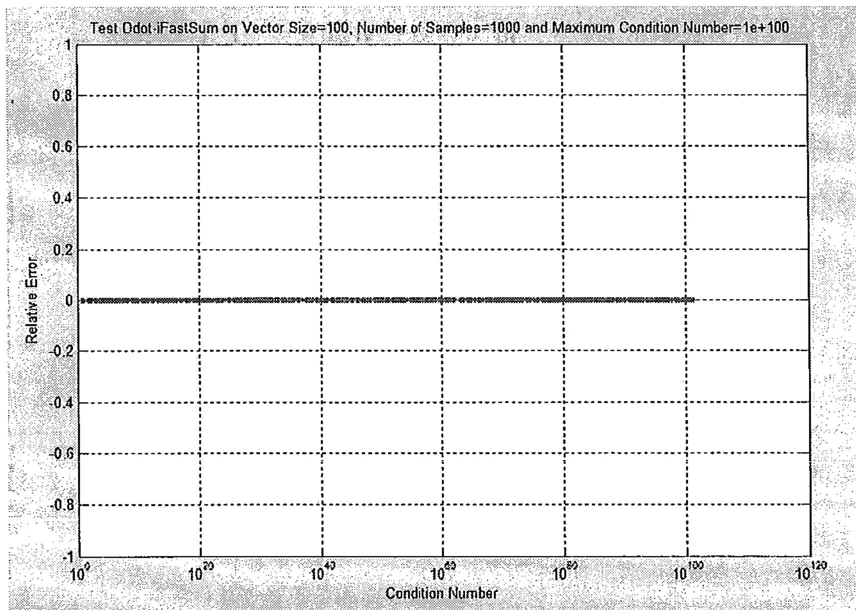

Figure 4.6: Test Results for Ddot_iFastSum

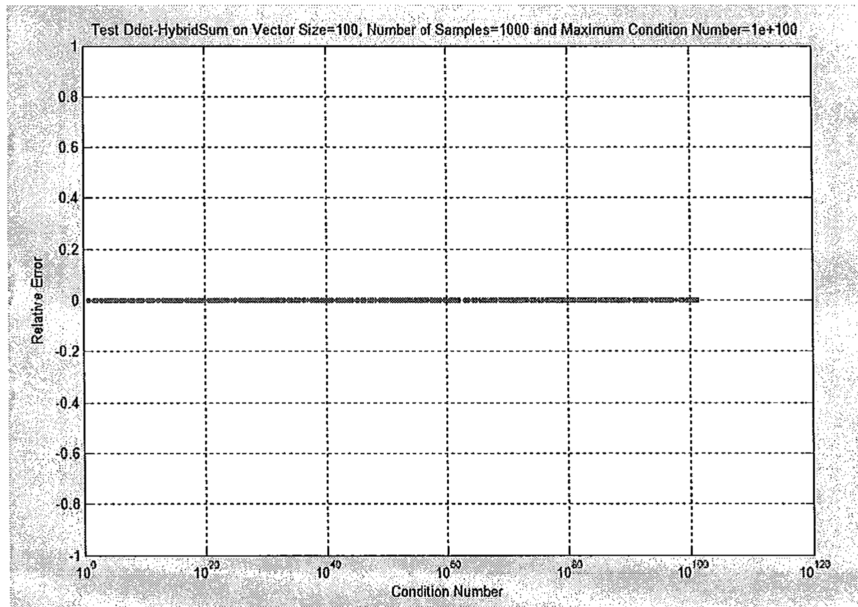

Figure 4.7: Test Results for Ddot_HybridSum

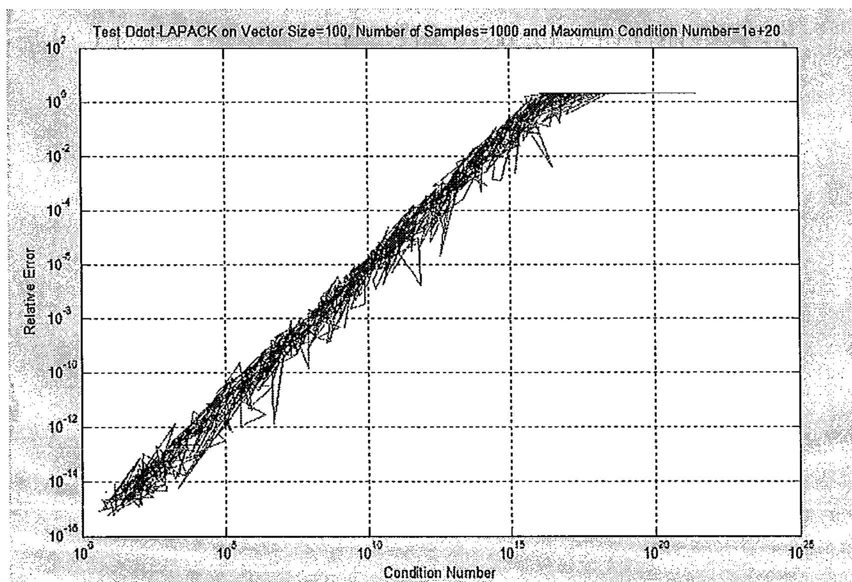

Figure 4.8: Test Results for Ddot_LAPACK (Condition Number $\leq 10^{20}$)

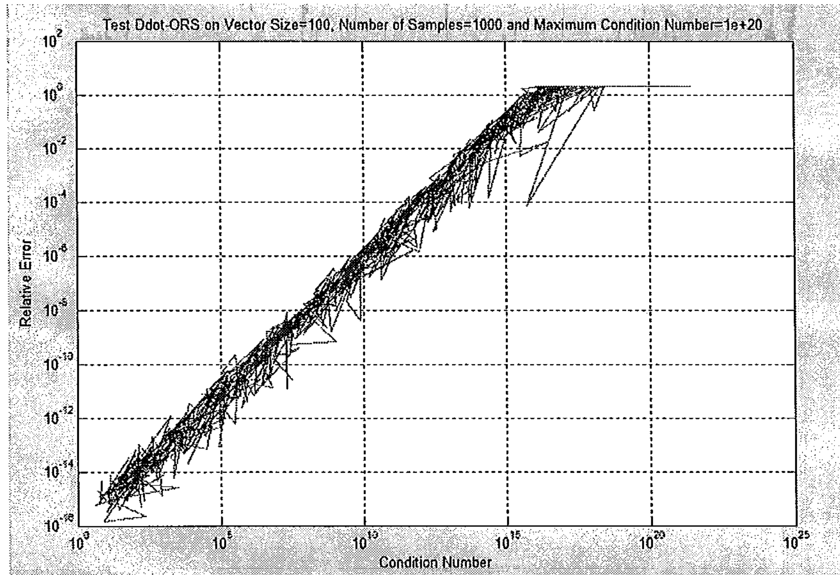

Figure 4.9: Test Results for Ddot_ORS (Condition Number $\leq 10^{20}$)

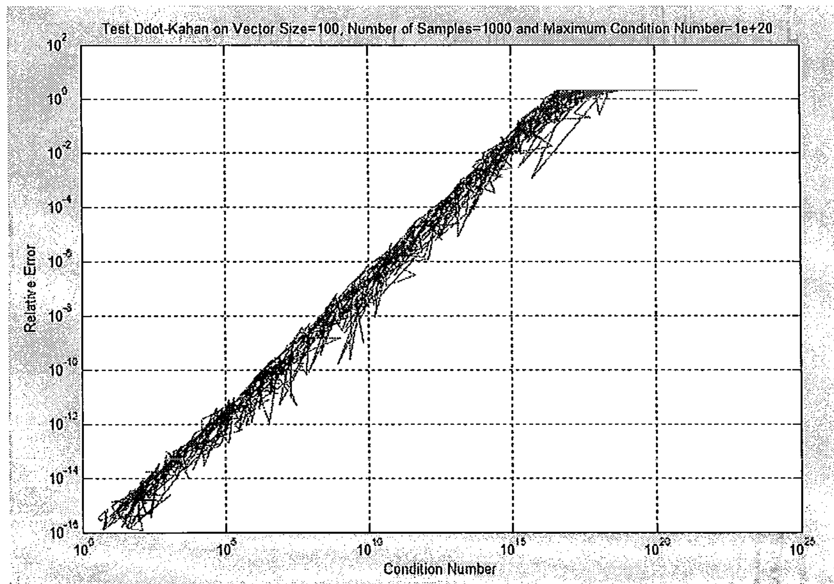

Figure 4.10: Test Results for Ddot_Kahan (Condition Number $\leq 10^{20}$)

Bibliography

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms,* Third edition, Addison-Wesley, Reading, MA, USA, 1998. ISBN 0-201-89684-2.

[2] Siegfried M. Rump, *Inversion Of Extremely Ill-Conditioned Matrices In Floating-Point.* Japan Journal of Industrial and Applied Mathematics, Volume 26, Numbers 2–3, 249–277, DOI: 10.1007/BF03186534, 2008.

[3] N. J. Higham, *Accuracy and Stability of Numerical Algorithm,* Second edition, SIAM, Philadelphia, PA, 2002. ISBN 0-89871-521-0.

[4] T. J. Dekker, *A floating-point technique for extending the available precision,* Numer. Math., 18 (1987), pp. 224–242.

[5] W. Kahan, *A survey of error analysis,* In Proc. IFIP Congress, Ljubjana, Information Processing 71, North-Holland, Amsterdam, The Netherlands, 1972, pages 1214–1239.

[6] Y. K. Zhu and W. B. Hayes, *Correct Rounding and a Hybrid Approach to Exact Floating-Point Summation,* SIAM Journal on Scientific Computing, Volume 31, Issue 4, 2009, Pages: 2981–3001, ISSN:1064-8275.

[7] T. Ogita, S. M. Rump and S. Oishi, *Accurate sum and dot product,* SIAM Journal on Scientific Computing, 26 (2005), pp. 1955–1988.

[8] I. J. Anderson, *A Distillation Algorithm For Floating-Point Summation,* SIAM J. SCI. COMPUT. Vol. 20, No. 5, 1999, pp. 1797–1806.

[9] S. M. Rump, *INTLAB - INTerval LABoratory,* Kluwer Academic Publishers, pp. 77–104, 1999, URL: http://www.ti3.tu-harburg.de/~rump/intlab/

[10] Anderson, E. and Bai, Z. and Bischof, C. and Blackford, S. and Demmel, J. and Dongarra, J. and Du Croz, J. and Greenbaum, A. and Hammarling, S. and McKenney, A. and Sorensen, D., *LAPACK Users' Guide,* Third Edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999, ISBN:0-89871-447-8, URL: http://www.netlib.org/lapack/