THE ACCURACY OF FLOATING POINT SUMMATIONS FOR CG-LIKE METHODS

Etsuko Mizukami

Technical Report 486

Department of Computer Science

Indiana University - Bloomington

July 1997

Contents

Abstract

It is well known that different ordering of summations in floating point arithmetic can give different sums due to rounding error. This dissertation reviews classic analytic error bounds. A new accurate algorithm is explained thoroughly along with its analytic error bound. These summation algorithms were implemented as dotproducts in an iterative solver to determine which summation ordering is more accurate in practice. Another issue is the relationship between dotproduct accuracy and the convergence of iterative solvers. Analysis and experiments indicate there are two primary sources of errors, and show which summation methods are better for reducing these errors. Results also indicate little correlation between dotproduct accuracy and numbers of iterations required by a solver, within a wide range of accuracies.

1

Statement of problem

Floating point summation is a thoroughly studied area in computer science. It is well known that different summation orders can give greatly different sums. In floating point representation, a number is represented by a mantissa and an exponent, where the mantissa part has a limited number of digits. When two floating point numbers are added, round-off occurs. Many numerical analysts have developed analytic error bounds which give upper bounds on the rounding error for different summation algorithms. There has been less success in using these error bounds to predict the *actual* accuracy of different summation algorithms in practice, and typically the algorithms are tested and compared for carefully constructed examples. A famous example of this type sums the Taylor series expansion for e^x at $x = -20$ [17].

The conventional analytic approach has two major flaws. One is that even if error bounds are studied in detail, the actual accuracy achieved in practice by an algorithm cannot be determined. Error bounds only guarantee that the rounding error does not exceed the given bound. The research question here is which algorithm gives the best accuracy most of the time and not which algorithm gives the sharpest error bound. This dissertation reviews these analytic error bounds, but the goal is not a detailed study of error bounds. Instead we ask which algorithm gives the best accuracy. Another flaw with the conventional approach is that in real applications the data is not random, and it rarely fits into the extreme examples constructed to demonstrate analytic error bounds. So the study of well-constructed examples need not help in choosing a method for real applications.

This dissertation analyzes dot products occuring in iterative solvers for linear systems of equations. Dot products account for virtually all of the computational work in iterative solvers, particularly for CG-like methods which are based on matrix-vector products and triangular solvers from incomplete factorizations. The main concern is with the relation of the accuracy of dot products to the overall performance of iterative solvers. As an example of this, Figure 1.1 shows the convergence history for the conjugate gradient (CG) algorithm applied to a linear system $Ax = b$. The matrix A is from discretizing the Laplace operator $\triangle u = 0$ using seven-point centered differences on a uniform $24 \times 24 \times 24$ mesh. Even for

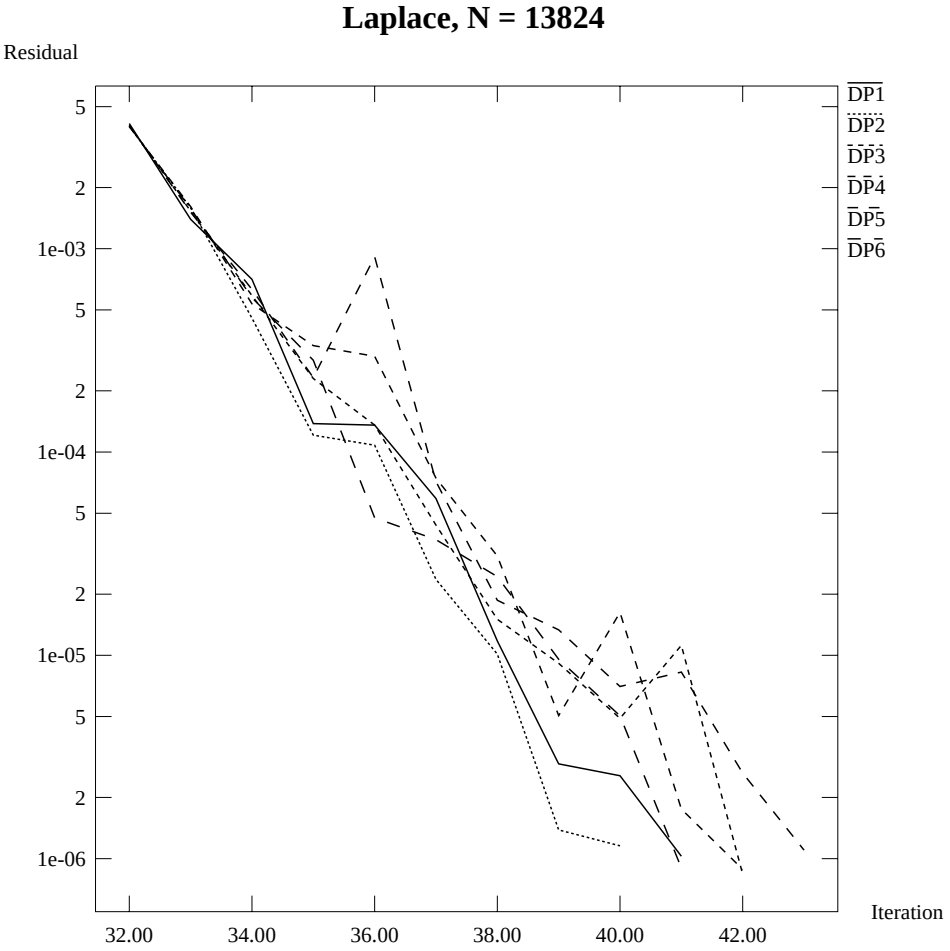


Figure 1.1: Residual Norm for Laplace Operator on 24^3 Grid

such a well-analyzed problem using a well-understood iterative method, there is a 10 percent difference in numbers of iterations required for convergence.

This variation is not surprising; CG typically displays a non-monotonic and even erratic convergence in terms of its residual 2-norm, and sometimes will have "plateaus" where the residual fails to make progress for several iterations. These variations in the numbers of iterations become even more significant for nonsymmetric linear systems. Many of them (like Bi-CGstab) rely on indefinite inner products, which can cause some instability in the computations. Furthermore, even a well-conditioned linear system gives no assurance that the convergence is monotonic, since for nonsymmetric systems the spectral condition number is a poor predictor of behavior.

Different summation orders occur routinely in almost all numerical parallel computing.

A common procedure is to partition the computational domain and assign different subdomains to different processors. Each processor computes a local dot product, all of which are then summed across processors. Because of variations in timing and the way the system assigns subdomains to hardware processors, different summation orders occur on different runs - even when using identical data and the same program.

In iterative methods for linear systems of equations, the vast majority of computation time occurs in matrix-vector products of the form $w = A * d$, and in applying a preconditioner, which typically means solving triangular systems. This implies that if a dot product summation ordering can provide a 20% reduction in the numbers of iterations required, then it can reduce the overall computation time by almost that much - even if the dot product summations themselves are several times more costly to compute.

This dissertation provides some answers to the following questions:

1. Do analytic error bounds indicate which summation orderings are more accurate in practice?
2. What is the relation of dot product accuracy to iteration counts in an iterative solver?

This dissertation also presents a new summation algorithm, intended to give the best accuracy. We will present this algorithm, compare it with other summation orderings, test the effects of different summation orderings in the context of an iterative solver, and finally make recommendations for selecting dot product summation orderings in applications. The results have implications extending beyond scientific computing, and suggest a thorough reexamination of what is commonly considered a solved problem by scientific software library designers and hardware designers.

2

Basic Ideas

Because the summation ordering problem is so pervasive, it has been intensely studied since the beginning of the computational era. This chapter gives only a rough summary of prior work; for more details the reader should consult Higham's excellent survey [4] and recent book [5], from which much of the following is drawn.

2.1 Summary of Previous Work

In 1956, I.A. Stegun and M.Abramowitz [16] pointed out that depending on the nature of the succeeding calculation a rounding error can cause a meaningless result (one with no correct digits). Their work showed that the consequences are especially serious when normalization occurs in the floating point calculations.

Normalization is the representation of nonzero floating point numbers with a leading bit of 1, which then need not be explicitly stored. Normalization is used in the standard IEEE floating point number representation [9] used by almost all computer processors.

In 1960, J. H. Wilkinson [6] derived an error bound using backward error analysis. That work showed that the solution of the set of equations $Ax = b$ is the exact solution of $(A + \delta A)x = b + \delta b$. In deriving a satisfactory bound on δA and δB Wilkinson provided an equation for the inner product (which is an essential suboperation for the overall solution process):

$$fl(x_1y_1 + x_2y_2 + \ldots + x_ry_r) \equiv \sum_i x_iy_i(1 + \varepsilon_i^{(r)})$$

where

$$|\varepsilon_1^{(r)}| \leq r2^{-t},\ |\varepsilon_i^{(r)}| \leq (r + 2 - i)2^{-t},\ (i = 2, \ldots, r)$$

and t is the number of digits in the floating point mantissa.

In 1964 Wolf [7] invented a summation method using cascade accumulators. This method uses several accumulators and ranges corresponding to these accumulators. If a sum is going

to exceed the maximum range of this accumulator, the value in the accumulator is moved to the next accumulator with larger range. Modifications of Wolf's algorithm were presented in 1965 by Ross [2] and Kahan [8]. Kahan also presented the "compensated summation" method, which manipulates only one accumulator.

In 1971, Malcolm [14] presented a further modification of Wolf's algorithm. He gave a detailed error analysis for this method. As with Wolf's algorithm, Malcolm's method is based on using several accumulators. The difference between Wolf's algorithm and Malcolm's is that each accumulator represents a range and each number is decomposed to that range and added to the corresponding accumulator. Finally each accumulator is summed up in decreasing order. He showed that his methods achieve a relative error of order u (unit roundoff).

In 1990, Dixon [12] tested the effect of rounding error on variable metric methods in nonlinear optimization and showed that rounding error can cause catastrophic damage in convergence. He showed that sorting summands before the summation reduces rounding error and improves performance.

In 1993, Higham [4] suggested several different summation methods and compared their error bounds. He analyzed the accuracy of these methods by using rounding error analysis and numerical experiments for carefully constructed examples.

2.2 Rounding Error Equations

Rounding error occurs when a number in a register of the floating-point unit of the computer is copied into memory, and so must be put into single precision or double precision. This may be demonstrated as follows. Assume the number is represented as $x = 2^{b_x} * a_x$, the mantissa a_x has t digits, and the machine has a doubled precision accumulator with $2t$ digits. Consider the case of adding two numbers $x = 2^{b_x} * a_x$, $y = 2^{b_y} * a_y$ and suppose that $b_x > b_y$.

1. If $b_x - b_y > t$ then y is too small to have effect, and so $fl(x + y) \equiv x$
2. If $b_x - b_y \leq t$, then a_y is first divided by $2^{b_x - b_y}$ (the *shift mantissa* phase of the addition.) The sum $a_x + 2^{b_x - b_y} a_y$ is then calculated exactly in 2t digits accumulator. This sum is then multiplied by the appropriate power of 2 to be renormalized. Finally this $2t$ digits mantissa is rounded to t-digits

If the normalized exact sum is $2^b * a$, then the modulus of the error is bounded by $2^b * \frac{1}{2} 2^{-t}$. It is obvious that the rounding off occurs in t-th digits in the mantissa after normalization. If relative errors are considered, the error bound is

$$fl(x \; op \; y) = (x \; op \; y)(1 + \delta), \; |\delta| \leq u, \; op = +, -, *, /, \tag{2.1}$$

where u is the unit round off 2^{-t}.

Equation 2.1 is easily misinterpreted. $(x + y)$ is not the result of the exact operation applied to the exact numbers x and y. Instead, $(x + y)$ is performed by first shifting the mantissa of x (or y) to match exponents, then adding the numbers. During the shift, truncation can occur on x or y, and that error is *not* reflected in equation (2.1). Consider the following example,

$$t = 3, \quad b = 10, \quad x = 110, \quad y = -8.59$$

Using the exact result of the summation on the right hand side of

$$102 = fl(x + y) = (101.41)(1 + \delta)$$

implies

$$\delta = 5.818 * 10^{-3} > \frac{1}{2} b^{1-t} = 5.000 * 10^{-3}$$

However using the shifted and truncated value .085 gives

$$102 = fl(x + y) = (101.5)(1 + \delta)$$

or

$$\delta = 4.926 * 10^{-3} \leq \frac{1}{2} b^{1-t} = 5.000 * 10^{-3},$$

which is correct.

3

Error Bounds for Summation

This chapter examines summing up numbers and the overall errors which occur during that computational process. The sum can be expressed as $S = S_n = \sum_{i=1}^{n} x_i$. Because of rounding errors, the result is different from the exact sum S, so that the resulting sum may be expressed as $\hat{S}$. If the ordering of summation is different, we have a different rounding error at each step of addition, and this in turn yields a different result for $\hat{S}$. Each floating point summation ordering can be associated with a binary summation tree. The leaves of the tree contain the values $x_i, i = 1, 2, \ldots n$, the internal nodes contain the intermediate sums

$$S_i = \sum_{k=1}^{i} x_k, \quad i = 1, 2, \ldots n,$$

and the root of the tree is the final sum S_n.

Next consider the error, which is defined as $E_i = S_i - \hat{S}_i$, $i = 1, 2, \ldots, n$. Instead of using (2.1), we use the equivalent but more convenient form

$$fl(x\ op\ y) = \frac{x\ op\ y}{1+\delta},\ |\delta| \leq u,\ op = +, -, *, /. \tag{3.1}$$

From (3.1), the local error introduced in each intermediate sum is $\delta\hat{S}$, where the intermediate sum is $\hat{S}$. Therefore the overall error is $E_n = \sum_{i=n+1}^{2n-1} \delta_i \hat{S}_i$, and it also follows that the smallest general error bound is

$$|E_n| \leq u \sum_{i=n+1}^{n-1} |\hat{S}_i| \tag{3.2}$$

Minimizing the upper bound in (3.2) requires minimizing the sum of the intermediate sums $\hat{S}_i$. This is the key idea behind the psum and insertion adder method we are going to discuss.

Summation algorithms can be considered as accumulation methods (adding into a single

register) or tree methods (modeled as an expression tree). For each of those two methods, error bounds can be obtained based on the computed partial sums $\hat{S}_k$ or by summing up local errors. This gives four different approaches to analyzing summation accuracy, presented in the next four subsections.

3.1 Accumulated Summation

3.1.1 Definition of Accumulation Summation

Accumulation summation treats the computed partial sum $\hat{S}_n$ as a single accumulator. First this accumulator is initialized to zero. Then each addend, x_i is added to this accumulator. When all the addends have been added in the final value, $\hat{S}_n$, is in the accumulator. The algorithm for accumulation summation is as follows.

Algorithm 1 Accumulation Summation

$s = 0$
for $(i = 1;\ i < n;\ i{+}{+})$ **do**
  $s = s + x_i$
end for

3.1.2 Error Bound Obtained by Calculating $\hat{S}_n$

One way to obtain the error bound for accumulated summation is to first calculate the value of $\hat{S}_n$, and then E_n is obtained by subtracting S_n from $\hat{S}_n$. Following the standard model of floating point arithmetic,

$$\hat{S}_k = fl(\hat{S}_{k-1} + x_k) = (\hat{S}_{k-1} + x_k)(1 + \delta_k),\ |\delta_k| \leq u,\ k = 2, \ldots, n \tag{3.3}$$

Applying (3.3) recursively gives

$$\hat{S}_n = (x_1 + x_2) \prod_{k=2}^{n} (1 + \delta_k) + \sum_{i=3}^{n} x_i \prod_{k=i}^{n} (1 + \delta_k). \tag{3.4}$$

To simplify the product terms, note that $|\delta_i| \leq u$ for $i = 1, \ldots, n$ and so

$$\prod_{i=1}^{k} (1 + \delta_i) = 1 + \theta_k,\ \ |\theta_k| \leq \frac{ku}{1 - ku} \equiv r_k \tag{3.5}$$

Using (3.5), we can rewrite (3.4) as

$$\hat{S}_n = (x_1 + x_2)(1 + \theta_{n-1}) + \sum_{i=3}^{n} (1 + \theta_{n-i+1}) \tag{3.6}$$

Equation (3.6) represents the value of $\hat{S}_n$ in terms of all addends and the local error introduced by each addition. $|E_n|$ is obtained by subtracting S_n which yields

$$|E_n| = \left| (x_1 + x_2)\theta_{n-1} + \sum_{i=3}^{n} x_i \theta_{n-i+1} \right| \tag{3.7}$$

$$\leq (|x_1| + |x_2|)r_{n-1} + \sum_{i=3}^{n} |x_i| r_{n-i+1} \tag{3.8}$$

The upper bound in (3.8) depends on the order of summation. Note that r_i is strictly increasing in i and so the upper bound is minimized when the x_i's are in order of increasing absolute value. However, this ordering minimizes the *error bound* and not necessarily the actual error. The summation ordering obtained by putting the summands in increasing order of absolute value is called **ainc** in this dissertation.

Inequality (3.8) can be weakened to obtain the bound

$$|E_n| \leq r_{n-1} \sum_{i=1}^{n} |x_i| = (n-1)u \sum_{i=1}^{n} |x_i| + o(u^2) \tag{3.9}$$

which is independent of the ordering. All the methods examined in this dissertation do satisfy this bound.

3.1.3 Error Bound Obtained by Summing up Local Errors

Another way of obtaining an error bound for accumulated summation is to sum up the local errors. For each successive addition, local error is introduced. Hence E_n is obtained by summing up all the local errors. We can rewrite the standard model of floating arithmetic as follows:

$$(\hat{S}_{k-1} + x_k)\delta_k = \hat{S}_k \delta_k / (1 + \delta_k) \tag{3.10}$$

The left hand side of equation (3.10) represents the local error introduced when the k-th summand is added. Summing up local errors for all steps gives

$$E_n = \sum_{k=2}^{n} \hat{S}_k \frac{\delta_k}{1 + \delta_k}. \tag{3.11}$$

and so

$$|E_n| = |\hat{S}_n - S_n| \leq \frac{u}{1-u} \sum_{k=2}^{n} |\hat{S}_k|. \tag{3.12}$$

The bound (3.12) suggests the strategy of ordering the summands x_i to minimize $\sum_{k=2}^{n} |\hat{S}_k|$. Determining that order is a combinationally expensive problem; a reasonable compromise is to determine the ordering sequentially by minimizing, in turn, $|x_1|$, $|\hat{S}_2|$,

$|\hat{S}_3|, \ldots, |\hat{S}_{n-1}|$. This ordering strategy, which we call **psum**, can be implemented with $O(n \log n)$ comparisons. Note that this strategy is essentially a dynamic programming method for minimizing the summation of the partial sums. The difference between **psum** and **ainc** is that the **psum** ordering is influenced by the signs of the x_i, while **ainc** is independent of the signs.

3.2 Tree Summation

Tree summation is the second major category of summation methods. Tree summation corresponds to using multiple accumulators, and then summing the accumulators. However, it is best defined and viewed in terms of an expression tree.

3.2.1 Definition of Tree Summation

Summing n addends can be modeled using a binary tree. Each leaf is a summand x_i, and each interior node corresponds to the sum of two numbers from one level deeper in the tree. The final sum is the root of the tree. This method is called **pairwise** summation and was first discussed by [15]. In this method, the x_i are summed in pairs

$$y_i = x_{2i-1} + x_{2i}, \quad i = 1, \ldots, \lfloor n/2 \rfloor \tag{3.13}$$

and this process is repeated recursively on the y_i, $i = 1 \ldots, [(n+1)/2]$. The final sum is obtained after $\log_2 n$ stages.

3.2.2 Error Bound Obtained by Calculating $\hat{S}_n$

As with accumulation summation, an error bound can be found by first calculating $\hat{S}_n$ and then computing E_n by subtracting S_n from $\hat{S}_n$. Assume for simplicity that the numbers of addends is $n = 2^r$ and $r = 2$. Then

$$\begin{aligned}
\hat{S}_{01} &= (1+\delta_{01})(x_1 + x_2) && (3.14)\\
\hat{S}_{02} &= (1+\delta_{02})(x_3 + x_4) && (3.15)\\
\hat{S} &= (1+\delta)(\hat{S}_{01} + \hat{S}_{02}) && (3.16)\\
&= (1+\delta)\left[(1+\delta_{01})x_1 + (1+\delta_{01})x_2 + (1+\delta_{02})x_3 + (1+\delta_{02})x_4\right] && (3.17)
\end{aligned}$$

$\hat{S}$ is obtained by a weighted sum of the x_k's, with weights given by products of δ_j's which are introduced by the intermediate summations. For an arbitrary value of r, $\hat{S}$ is obtained as

$$\hat{S}_n = \sum_{i=1}^{n} x_i \prod_{k=1}^{\log_2 n} (1+\delta_k^{(i)}), \ |\delta_k^{(i)}| \leq u \tag{3.18}$$

which leads to the bound

$$|E_n| \leq r \log_2 n \sum_{i}^{n} |x_i| \tag{3.19}$$

This is a smaller bound than the corresponding one for accumulation summation because the factor n from (3.9) is replaced by $\log_2 n$. However, in special cases the bound in (3.8) for accumulation summation can be smaller than that in (3.19). Furthermore, when hardware supports higher precision accumulation, this method loses some accuracy since the intermediate results must be stored in memory. In this dissertation that compounding factor is circumvented by forcing a store to memory after each addition, both for accumulation and tree summation.

3.2.3 Error Bound Obtained by Summing up Local Errors

For tree summation, the computed intermediate sums S_k satisfy

$$\hat{S}_k = (\hat{S}_{k1} + \hat{S}_{k2})(1 + \delta_k) \tag{3.20}$$

This can be rewritten as

$$(\hat{S}_{k1} + \hat{S}_{k2})\delta_k = \hat{S}_k \delta_k / (1 + \delta_k), \tag{3.21}$$

where the left hand side of this equation is equal to the intermediate error E_k. The total error E_n is then obtained by summing all the intermediate errors:

$$E(n) = \sum_{k=n+1}^{2n-1} \hat{S}_k \delta_k / (1 + \delta_k). \tag{3.22}$$

Therefore the error bound is as follows:

$$|E_n| \leq \frac{u}{1-u} \sum_{k=n+1}^{2n-1} |\hat{S}_k| \tag{3.23}$$

As with inequality (3.12), the error bound can be minimized by ordering the summands so that the sum of absolute values of the intermediate sums $\hat{S}_k$ is minimized.

4

Summation Methods

Seven different summation methods are tested in this dissertation. These methods attempt to minimize the various error bounds presented earlier. These methods are denoted as

- accurate **acc**,
- absolute increase **ainc**,
- absolute decrease **adec**,
- psum **psum**,
- insertion adder **insadd**,
- pairwise **pair**,
- plus/minus increase **pminc**, and
- plus/minus pairwise **pmpair**.

Each is categorized as either accumulation or tree summation, corresponding to the two general summation approaches. The accumulated summation methods are **acc**,**adec**, and **psum**. The rest are categorized as tree summation.

In addition to analytic error bounds, these methods are discussed in terms of shifting error and carrying error. When two summands are added to each other, first the mantissa of the addend with the smaller exponent is shifted (corresponding to lining up the decimal point in the usual pencil-and-paper method of addition of decimal numbers.) This type of rounding error is called *shifting error*. When the mantissas are added, a nonzero carry bit may cause the resultant mantissa to require more bits than is allowed for floating point representation. This type of rounding error is called *carrying error*. Note that carrying error will not happen when the summands are of opposite sign. The larger the difference in

exponents for the two addends, the larger the shifting error can be. For both shifting error and carrying error, if addends are large the rounding error will be large, since the mantissa is multiplied by the exponent.

Now each of the methods will be examined in detail.

4.1 Absolute Increasing Order

This is the algorithm best minimizes the error bound (3.10) in accumulation summation. The key idea of this algorithm is to minimize the shifting and carrying errors for overall summation. In order to do that, addends are sorted by absolute increasing value first. Then each is added to one accumulator in that order. The algorithm is:

4.1.1 Error Bound Obtained by Calculating $\hat{S}_n$

Algorithm 2 ainc: Summation in Absolute Increasing Order

```
Sort x into absolute increasing order
s = 0
for i = 1 to n do
    s = s + x_i
end for
```

When all the addends have the same sign, **ainc** gives best accuracy among the accumulation summation methods. Since it tries to minimize the absolute value in the accumulator at each stage, it reduces the chances of carrying error. Also, as the value in the accumulator grows, the addends remaining are large values and this encourages adding between similar magnitude values, thereby reducing shifting errors. The flaw of this method is it does not consider signs and so when addends have opposite signs, the other methods explained later tend to work better.

4.2 Psum Order

Psum partially minimizes the error bound (3.12), but unlike **ainc** considers the signs of the addends. Recall that it is too costly to exactly minimize $\sum_{k=2}^{n} |\hat{S}_k|$, since this is a combinationally complex problem. Instead, **psum** ordering determines the ordering by sequentially minimizing $|x_1|, |\hat{S}_2|, |\hat{S}_3|, ...|\hat{S}_{n-1}|$. This strategy can be implemented with $\mathcal{O}(n \log n)$ comparisons. In **adec**, heavy cancellation is by chance. But **psum** looks through all the addends and find the nearest value of opposite sign to the current value in the accumulator, and adds it in. The algorithm is:

Algorithm 3 psum: Summation by Psum Method

```
s = 0
for i = 0 to n do
   candidate1 ← find nearest absolute value of opposite sign of s
   candidate2 ← find smallest value of same sign of s
   addend ← either candidate1 or candidate2 which minimize summand
   s = s+ addend
end for
```

For this algorithm, we have to first separate positive and negative addends, and sort each category by absolute value increasing order. For finding *candidate*1 in **psum** algorithm, we can use binary search, as addends are sorted at first. Thus, this algorithm can require $\mathcal{O}(n \log n)$ comparisons.

4.3 Absolute Decreasing Order

This algorithm actually *maximizes* the error bound (3.10). When all the addends are same signs, it works the worst among the accumulation summation methods. However, it works well when the addends have opposite signs. The reason is that it tends to encourage heavy cancellation. Heavy cancellation means that when two addends are large value and opposite signs, the result becomes small value, since it is the difference of the two. Heavy cancellation in the early stages of summation reduces carrying error. The algorithm is:

Algorithm 4 adec: Summation in Absolute Decreasing Order

```
Sort x into absolute decreasing order
s = 0
for i = 1 to n do
   s = s + x_i
end for
```

Even though this method maximizes the error bound (3.10), in some case it minimizes the error bound (3.12). The following [4] is an example of minimizing the error bound of (3.12). Let the vector x of summands be $x = [1, M, 2M, -3M]$, where M is a power of the machine base and is large enough so that $fl(1 + M) = M$. The three summation orderings presented so far produce the following results:

1. **ainc**: $\hat{S}_n = fl(1 + M + 2M - 3M) = 0$
2. **psum**: $\hat{S}_n = fl(1 + M - 3M + 2M) = 0$
3. **adec**: $\hat{S}_n = fl(-3M + 2M + M + 1) = 1$

The reason why the decreasing order is accurate in this case is that it adds the *1* after heavy cancellation has taken place. If we evaluate the term $\mu = \sum_{k=2}^{n} |\hat{S}_k|$ in the error bound (3.12) we find

1. **ainc**: $\mu = 4M$
2. **psum**: $\mu = 3M$
3. **adec**: $\mu = M + 1$

and so (3.12) *predicts* that decreasing order will produce the most accurate answer.

In conclusion decreasing order is likely to yield greater accuracy than the increasing whenever there is heavy cancellation in the sum. That is, whenever $|\sum_{i=1}^{n} x_i| \ll \sum_{i=1}^{n} |x_i|$.

4.4 Pairwise Summation

Pairwise summation is based on the definition of tree summation. As explained earlier, each addend corresponds to a leaf and each addition corresponds to an interior node, forming a binary tree. Therefore, the sum is obtained in $\log_2 n$ stages. **Pairwise** summation is attractive in a parallel setting, because each of the $\log_2 n$ stages can be done in parallel. The algorithm for **pairwise** summation is:

Algorithm 5 pairwise: Summation by Pairwise Method

```
makequeue of x
while queue is not empty do
  addend1 ← move-front-queue of x
  addend2 ← move-front-queue of x
  s ← addend1 + addend2
  Insert-tail-queue(x,s)
end while
```

The error bound for **pairwise** summation is given by equation (3.18). However, (3.18) is derived under the assumption that $n = 2^r$. This is a smaller bound than (3.10) for accumulation summation, since it is proportional to $log_2 n$ rather than n. However, when hardware supports higher precision accumulation, this method loses some accuracy since the intermediate results must be stored in memory.

4.5 Insertion Adder

When all the addends have the same sign, **insadd** minimizes the error bound (3.23) among tree summation methods. The key idea of this algorithm is to encourage the addition

to be between numbers of similar magnitude. Thus it tends to reduce shifting error. Also, it is based on tree summation, where each leaf passes through the same number of additions, contrary to accumulated summation where the first summand will pass through $n-1$ additions. The algorithm for insertion adder is:

Algorithm 6 Insadd: Summation by Insertion Adder

```
BuildHeap(x)
while heap has more than 1 elements do
  addend1 ← move minimum from heap x
  addend2 ← move minimum from heap x
  s = addend1 + addend2
  InsertHeap(x,s)
  ReBuildHeap(x)
end while
```

The flaw of **insadd** is it does not consider signs. So it does not encourage heavy cancellation. It has cancellation by chance when two addends are close in absolute value and of opposite sign. But since the heap is built in absolute increasing order, heavy cancellation will not occur in early stage of additions as opposed to absolute decreasing order.

4.6 +/- Method

The +/- method was first suggested by Lasdon [11]. He derived an algorithm for solving an optimization problem that arises in the design of sonar arrays. This method first divides addends into positive and negative numbers. Then for each group, it uses some method (for example, **ainc** or **pair**) to calculate each sum. Finally these two sums are added to get the final result. The algorithm is:

Algorithm 7 +/-: Summation by +/- Method

```
divide addends into plus-vector and minus-vector
sum-for-plus ← use some method to form sum of plus-vector
sum-for-minus ← use some method to form sum of minus-vector
s = sum-for-plus + sum-for-minus
```

This method is based on the misbelief that cancellation is an undesirable source of numerical errors. However, heavy cancellation works well in terms of reducing carrying errors. The advantage of this method is we can apply the best available method for addends of all the same sign to obtain the separate sums for positive and negative summands. We know which method works best when all the addends have the same sign. But we do not know and it is hard to predict which method works best when addends have opposite signs. Thus we can guarantee minimizing rounding error for both vectors, having only to add two

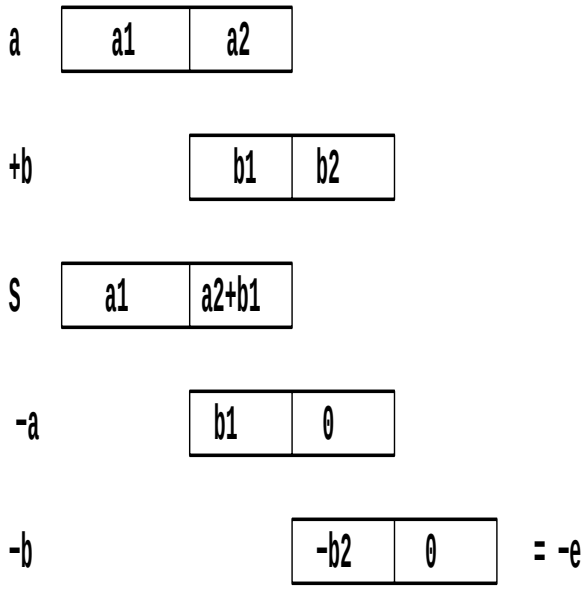


Figure 4.1: Recovering error

numbers of opposite sign at the end. The **psum** or **adec** method gives better accuracy than this method, since it make use of heavy cancellation, but **psum** method is costly and for **adec** cancellation is by chance.

4.7 Compensated Summation

Compensated summation is an accumulation summation method having a correction phase in order to diminish round error. The rounding error can be calculated by following lemma.

Lemma 1 *Let a and b be floating point numbers with $|a| \geq |b|$, and let $\hat{s} = fl(a+b)$. Then* $e = -[((a+b)-a)-b] = (a-\hat{s})+b$

Figure 4.1 illustrates Lemma 1. Kahan's compensated summation [8] method employees the correction e on every step of an accumulated summation method. After each sum is formed, the correction is computed and immediately added to the next term x_i before that term is added to the partial sum. The algorithm may be written as follows.

Knuth [10] shows that the computed sum $\hat{S}_n$ satisfies

$$\hat{S}_n = \sum_{i=1}^{n}(1+\mu_i)x_i,\ |\mu_i| \leq 2u + \mathcal{O}(nu^2) \tag{4.1}$$

Algorithm 8 Compensated Summation

$s = 0$
$e = 0$
for $i = 1$ to n **do**
  $temp = s$
  $y = x_i + e$
  $s = temp + y$
  $e = (temp - s) + y$
end for

which is an almost ideal backward error result. The forward error bound corresponding to inequality (4.1) is

$$|E_n| \leq (2u + O(nu^2)) \sum_{i=1}^{n} |x_i| \tag{4.2}$$

5

Summation Method Accurate

Method **acc** is intended to give better accuracy than the other methods. But in terms of memory required and computational time, this method is quite expensive. Thus it is not intended as a practical method in computation, but instead is developed here to explore the limits of summation accuracy. The method is based on the following concepts:

1. If addends have opposite sign and the same exponent, there is no rounding error.
2. Heavy cancellation is the most important factor for reducing rounding off.

Concept 1 is based on the following lemma.

Lemma 2 *Let $\hat{x}$ and $\hat{y}$ be machine numbers in a binary floating-point machine such that $\hat{x}\hat{y} < 0$ and $2|\hat{y}| \geq |\hat{x}| \geq |\hat{y}|$. Then $fl(\hat{x} + \hat{y}) = \hat{x} + \hat{y}$*

The reason for concept 2 is explained in **adec** method in the previous chapter. When two addends have the same sign and large exponents, large rounding error occur. On the other hand, if two addends have same sign and small exponents, rounding error is small. So a goal is to add differently signed numbers with large exponents, and get a result which has a small exponent for further computations. This leads to adding same sign numbers with small exponents later on, rather than same sign numbers with large exponents.

5.1 Algorithm for Accurate Method

The algorithm works as follows. First add every possible pair of addends which have opposite signs but same exponent. In this step, there is no rounding error. Next add every possible pair of addends which have opposite sign of nearest exponents. This encourages heavy cancellation. After this, all remaining addends have the same sign. Add the rest together using absolute increasing order. The more detailed explanation of each step follows. In this, a "bucket" is used for each exponent; in the IEEE floating point standard for double precision numbers this requires 2048 buckets.

Algorithm 9 acc Summation by Accurate Method

1: Prepare buckets for the number of possible exponents and put all addends in the appropriate bucket (bucket sort).
2: Add entries of opposite sign in same bucket.
3: **for** Largest exponent bucket down to smallest bucket **do**
4: Take elements from larger exponent bucket.
5: Scan to left (towards smaller exponent buckets) and find elements with opposite sign. Add those elements together.
6: Put result in its appropriate bucket. If there is an opposite sign element in that bucket, add them together.
7: **end for**
8: Add entries of same sign in same bucket. After this, each bucket has at most 1 entry.
9: Add remaining entries in order from smallest exponent bucket to largest exponent bucket.

5.2 Error Bound for Accurate Method

We now consider the error for method **acc** in each step of Algorithm 9.

5.2.1 Add Oppositely Signed Numbers with Same Exponent

Based on Lemma 2, this step incurs no rounding errors. So the main concern is by how much the number of addends are reduced for later phases, which may have some rounding error. Notice that since addends are always normalized, the most significant bit of the mantissa is 1. That means there is no chance the sum has the same exponent. The probability that sum has a 1 smaller exponent is 1/2, a 2 smaller exponent is 1/4, a 3 smaller exponent is 1/8 and so forth. In general, the probability that the summand has an s bit smaller exponent is $1/2^s$.

Consider the following example. Suppose there are 8 positive addends and 8 negative addends in the same bucket. According to the probability after all the additions taken place, the result is shown in Figure 5.1.

In this case the number of addends is reduced from 16 to 4. If addends are approximately evenly divided between positive and negative terms and are distributed among the buckets uniformly, a large reduction in the number of summands for later stages occurs.

5.2.2 Add Oppositely Signed Numbers with Nearest Exponents

At this stage, all the addends in one bucket should have same sign. Notice two important things here. One is that when summing two numbers of opposite sign, the sum has the sign of the addend with larger exponent. The other is that the resulting exponent is usually

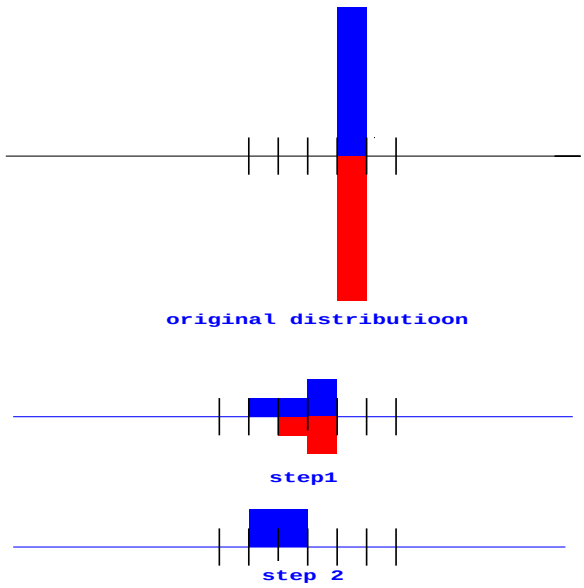


Figure 5.1: Example of adding oppositely signed numbers with the same exponent

the same as the addend of larger exponent. Sometime the result of the exponent has an exponent one smaller than that of one.

In the following, consider all the combinations of addends x and y and calculate the result's exponent.

```
Addend Y has exponent 1 smaller than that of X:
addend X  addend Y
  100      010
  101      011
  110
  111

The probability of sum has the same exponent as X is 3/8,
1 bit smaller 4/8, 2 bit smaller 1/8


Addends Y has exponent 2 smaller than that of X:
addend X  addend Y
  100      001
  101
  110
  111
```

```
The probability of sum has the same exponent as X is 3/4,
1 bit smaller 1/4


Addends Y has exponent 3 smaller than that of X:
addend X  addend Y
  1000     0001
  1001
  1010
  1011
  1100
  1101
  1110
  1111

The probability of sum has the same exponent as X is 7/8,
1 smaller exponent is 1/8
```

Averaging out these cases gives the probability for bit shifting as follows:

1. The probability of sum has the same exponent as X = 1/6 * (3/8 + 7/8 + 15/16 + 31/32 + 63/64) = 0.8151
2. The probability of sum has 1 bit smaller exponent than X = 1/6 * (4/8 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64) = 0.1640
3. The probability of sum has 2 bit smaller exponent than X = 1/6 * 1/8 = 0.02

From these numbers most of the time the sum has the same exponent as X.

Next consider how much rounding error occurs for the separate cases as following. By convention, we assume that the two addends are $x = 2^{a_x} * b_x$, and $y = 2^{a_y} * b_y$, and the mantissa has t digits. Since we search opposite sign addend y from 1 smaller exponent of x toward smallest exponent bucket, the probability we can find y in 1 exponent smaller is 1/2, 2 exponent smaller is 1/4, and 3 exponent smaller is 1/8.

5.2.2.1 Addend exponents differ by one. Suppose x's exponent is 1 larger than that of y. Then before being added together, y is normalized and its mantissa is shifted 1 bit to the left. There are now two cases, corresponding to the least significant bit of y being 0 or 1. If it is 0, there is no rounding error. If it is 1, the rounding error is $1/2 * 2^{-t} * 2^{a_x}$. Notice that probability of getting no rounding error is 1/2.

5.2.2.2 Addend exponents differ by two. Suppose addend x's exponent is 2 larger than that of addend y. Applying the same theory described above, the probability of having no rounding error is 1/8. In the worst case the rounding error is $(1/2 + 1/4) * 2^{-t} * 2^{a_x}$.

5.2.2.3 Addend exponents differ by three. Similar to above, the probability of having no rounding error is 1/16. In the worst case the rounding error is $(1/2+1/4+1/16)*2^{-t}*2^{a_x}$.

5.2.2.4 Error Reduction. As with compensated summation, the amount which is lost when adding two numbers from different buckets is known and can be accounted for (with additional complexity in the algorithm.) If the exponents of two addends differ by one, and if the addend which has smaller exponent has another addend in same bucket, it is better to add those two together. This will cause rounding error for adding same sign of exponent, but the sum shifts to the next larger bucket and will give better cancellation with the addend in that bucket. Consider the error for these two approaches (first adding numbers from different buckets versus first adding the two numbers from the same bucket). Call the first algorithm the original **acc** method, and the second on the revised **acc** method. Suppose there are four buckets with exponents 2^{m-3}, 2^{m-2},2^{m-1}, and 2^m. Each bucket contains only 1 addend and the sign of the addends are -,-,-,+, respectively. The rounding errors for the two algorithms are

$$original = (1/2 * 2^m + (1/2 + 1/4) * 2^m + (1/2 + 1/4 + 1/8) * 2^m) * 2^{-t} = 17/8 * 2^{m-t} \quad (5.1)$$

$$revise = (2^{m-1} + 2^{m-2} + 2^{m-3}) * 2^{-t} = 7/8 * 2^{m-t} \quad (5.2)$$

Next consider by how much the number of addends is reduced at this stage. Suppose there are four buckets and each bucket contains only one addend. All the possible situations and the number of addends remaining at the end are listed below.

addends in bucket	number of addends at last
----	4
---+	1
--+-	3
--++	2
-+--	3
-+-+	2
-++-	1
-+++	3
+---	3
+--+	1
+-+-	2
+-++	3
++--	2
++-+	3

```
+++-              1
++++              4
```

When all the cases are averaged out, the number of addends at the end is 2.3, which is 58% of the original number of addends.

5.2.3 Adding Same Signed Numbers with Same Exponent

Notice that if addends have the same sign and exponent, the sum has exponent which is always larger by one. Again the probability of having no rounding error is 1/2 and having rounding error is 1/2. The rounding error is always $2^{-t} * 2^{b_x}$. Consider an example with eight nonempty buckets and six small exponent buckets, each containing exactly four addends. Assume that each addition causes a carry. Then the number of additions which occurs in each bucket is:

```
The number of additions 2 3 3 3 3 3 1 0
Addends remaining       0 0 1 1 1 1 1 1
```

Note that except for both ends, the number of additions is one smaller than the number of addends in one bucket. Then the error is

$$E \leq 3 * 2^{-t} * \sum_{i=1}^{8} 2^{m-i} \tag{5.3}$$

5.2.4 Add Remaining Numbers From Smallest to Largest Exponent

Two kinds of rounding errors occur at this stage. First, because the addends have different exponents the mantissa of the smaller exponent number has to be shifted and rounding error occurs. Secondly, when the sum has a carry, it has to be normalized and its least significant bit is rounded. At this stage all the buckets contain one or zero addends. Consider the example of addends, $1111*2^0$,$1111*2^1$,$1111*2^2$,$1111*2^3$. The result is:

```
                  shifting error    carrying error
1111 * 2^{1}
 111 * 2^{1} -> 1/2 * 2^{1}

1011 * 2^{2}
1111 * 2^{2}
```

```
1101  * 2^{3}
1111  * 2^{3}

1110  * 2^{4}
1111  * 2^{4}                    -> 1 * 2^{4}
```

Now consider the more general case. Suppose there are three addends x,y,z,and each addend has exponent 2^x,2^y,2^z, respectively. Suppose that the distances are d, that is, $y - x = d$ and $z - y = d$. Also suppose that the mantissa has t digits. First add x and y. The resulting shifting error is $2^{y-t} * \sum_{i=1}^{d}(1/2)^i$. Next there are two cases, one is having carrying error and the other is no carry. In the case of no carry, next add the sum and z. Shifting error is then $2^{z-t} * \sum_{i=1}^{d}(1/2)^i$. If this has a further carrying error, then the error is 2^{z-t}. Next consider the case having a carry error when x and y are added. The carrying error is 2^{y-t} and rest are same with the case for not having carrying error. Then two kinds of error is expressed as follows.

$$2^{y-t} * \sum_{i=1}^{d}(1/2)^i + 2^{z-t} * \sum_{i=0}^{d}(1/2)^i = 2^{z-t} * (3 + 2^{-d} - 2^{-2d}) \tag{5.4}$$

$$2^{y-t} * \sum_{i=0}^{d}(1/2)^i + 2^{z-t} * \sum_{i=0}^{d}(1/2)^i = 2^{z-t} * (3 + 3/2 * 2^{-d} - 2^{-2d}) \tag{5.5}$$

From this the rounding error is better than the two least significant digits of mantissa multiplied by the exponent of the largest addend.

5.2.5 Overall Error Bound Estimation

Now an estimation of the error bound for method **acc** is given in separate cases for summations. Those cases are where all the addends have the same sign, or where they have differing signs. In the latter case we consider equal distribution of signs. To simplify the discussion, the substeps in method **acc** are denoted as:

1. add oppositely signed numbers with same exponent (**opposite-same**)
2. add oppositely signed numbers with different exponents (**opposite-dif**)
3. add numbers with same sign and exponents (**same**)
4. add remaining numbers from smallest to largest bucket (**sumup**)

5.2.5.1 All Addends have the Same Sign. Here cancellation does not occur, and **opposite-same** and **opposite-dif** are not performed. Only **same** and **sumup** need be considered.

For **same**, suppose there are n addends in one bucket whose exponent is 2^m. The error is

$$error = (1/2 * n) * 2^{m-t} + (1/4 * n) * 2^{m-t+1} .. = 1/2 * n * 2^{m-t} * \log n \tag{5.6}$$

Supposing that all the buckets have n addends distributed uniformly between exponents e_{min} and e_{max}, the total error is

$$error = 1/2 * n * \log n * \sum_{i=e_{min}}^{e_{max}} 2^{i-t} \tag{5.7}$$

For the **sumup** phase, when the exponents of the addends are at a distance d apart, the rounding error (including both shifting and carrying error) is

$$error = 2^{m-t} * \sum_{i=1}^{d} (1/2)^i + 2^{m-t} = 2^{m-t} * \sum_{i=0}^{d} (1/2)^i \tag{5.8}$$

Therefore the total error is

$$error = \sum_{j=e_{min}}^{e_{max}} 2^{j-t} * \sum_{i=0}^{d} (1/2)^i \tag{5.9}$$

Adding up the error bounds for both phase, the error bounds become

$$\begin{aligned} error &= \sum_{j=e_{min}}^{e_{max}} 2^{j-t} * (1/2 * n * \log n + \sum_{i=0}^{d} (1/2)^i) && (5.10) \\ &< 2^{e_{max}-t}(4 + n \log n) && (5.11) \end{aligned}$$

From this equation notice that n (the number of addends) and the maximum exponent e_{max} have largest effect on the magnitude of error.

5.2.5.2 Addends of Opposite Sign are Distributed Uniformaly. For **opposite-same**, suppose that each bucket contains the same number of positive and negative addends. Assume that the total number of addends is n here. By the probability calculated in previous section, it is likely that the number of addends is reduced to around $n/4$.

For **opposite-dif**, for simplicity assume that numbers of positive and negative addends are nearly equal. When exponent of addends are distance d, then shifting error is

$$error = 2^{e_{max}-t} * \sum_{i=1}^{d} (1/2)^i \tag{5.12}$$

Supposing there are $n/4$ addends at this stage, the total error is

$$error = n/8 * 2^{e_{max}-t} * \sum_{i=1}^{d} (1/2)^i \tag{5.13}$$

For **same** and **sumup**, from previous computations the number of addends is now about $n/8about$. Then the total error is follows.

$$\begin{aligned} error &= n/8 * 2^{e_{max1}-t} \sum_{i=1}^{d1} (1/2)^i + \sum_{i=e_{min2}}^{e_{max2}} 2^{i-t}(n/16 * \log n/8 + \sum_{i=0}^{d2} (1/2)^i) && (5.14) \\ &< 2^{e_{max1}-t} * n/8 + 2^{e_{max2}-t}(4 + n/8 * \log n) && (5.15) \\ &< 2^{e_{max}-t}(4 + n/8(1 + \log n)) && (5.16) \end{aligned}$$

e_{max1} stands for max exponent at **opposite-dif** phase and $d1$ stands for average distance of exponent between two addends at **opposite-dif** phase. e_{max2} stands for max exponent at **same** and **sumup** and $d2$ stands for average distance of exponent between two addends at **same** and **sumup**.

Again in this formula the value of n and the maximum exponent occuring dominate the entire value. Since in the **opposite-same** phase the number of addends are reduced to $n/4$, this greatly lessens the error. This is an estimate and if there is more cancellation during **opposite-same**, it further reduces the error. Also note that $max2$ is smaller than $max1$, because of the effect of cancellation during **opposite-dif**.

5.2.6 Related work on accurate method

The following material is based on L.Kobbelt[13]. Two other operations do not cause any rounding error:

1. Summation of two numbers with the same exponent, same sign and same genus. The "genus" of a floating-point number is the least significant bit of mantissa.
2. Multiplication of a floating-point number by power of 2.

During **opposite-same** phase, in addition to adding 2 summands which have same exponent and opposite sign, also add summands with the same exponent and same sign and genus. These further reduces the number of addends without introducing any rounding error.

Operation 2 is further used to introduce following equation.

$$a2^{k+j} - b2^k = \sum_{i=0}^{j-1} a2^{k+i} + (a2^k - b2^k) \tag{5.17}$$

$(a2^k - b2^k)$ does not cause any rounding error by means of operation 2. During the **opposite-dif** phase, we can make use of this operation. Take the smaller exponent between the two addends, and calcuate $(a2^k - b2^k)$, where k is the smaller exponent. For $\sum_{i=0}^{j-1} a2^{k+i}$, each summand should be put into the appropriate bucket, for later use.

This works pretty well as long as the difference of the exponents of the two addends is small. Otherwise overhead of calcuation increases and becomes inefficient.

6

Summary of All the Methods and Error Bounds

6.1 Example for All the Methods

Simple examples are shown for each method in order to visualize how each algorithm works. Suppose that there are four numbers to add, $-11 * 2^0$, $-11 * 2^1$, $-10 * 2^2$, and $11 * 2^2$ in binary notation. Assume that the mantissa is two bits and the accumulator also has a two bit mantissa. Figures 6.1 and 6.2 show the summation tree for each method considered. For each method,

$|E|$ and the sum of partial sums $\sum_k^n |S_k|$ are shown in Table 6.1. Methods **adec**, **psum** and **acc** methods give the best accuracy in this example. But in terms of the value $\sum_k^n |S_k|$, **adec** and **acc** methods give smaller values for the error bounds. The value of $\sum_k^n |S_k|$ for **psum** is larger than that of **adec** and **acc** methods. Although in general accuracy and error bounds need not correspond, for this example they do. As a rule of thumb, minimizing the sum of partial sums gives the best final accuracy.

6.2 Comparison Among Error Bounds

The error bounds in Table 6.2 will be compared and analyzed more in detail. By convention, we name the error bounds themselves as the **ainc** error bound, the **psum** error bound, the **pairwise** error bound and the **insadd** error bound, as each error bound is minimized by the corresponding algorithm.

6.2.1 Comparison Between ainc and psum Error Bounds

The coefficient of the **psum** error bound $(u/1 - u)$ is r_1, and notice that r_i is strictly increasing in i. If the sum $\hat{S}$ is obtained by truncating the exact sum S, then $\sum |\hat{S}_k| \leq \sum |\hat{x}_k|$.

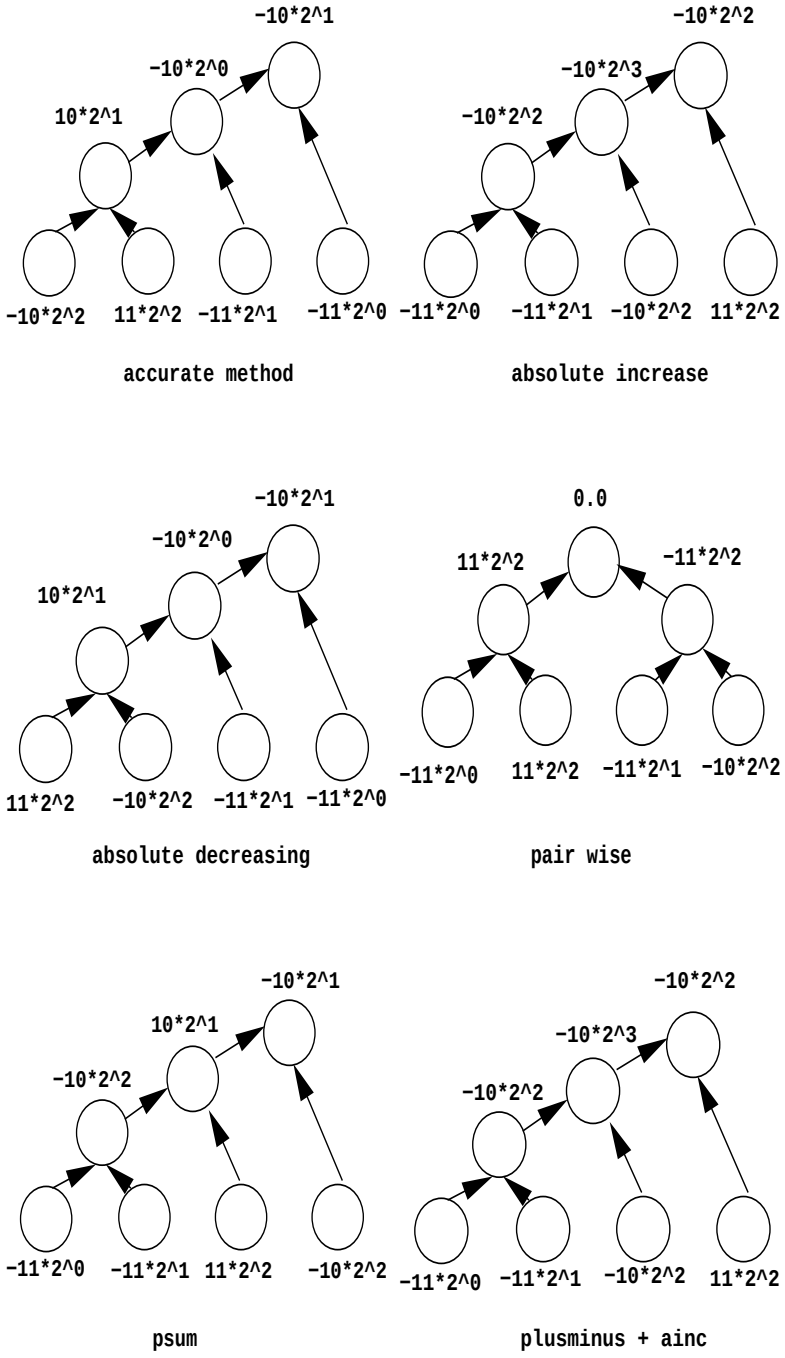


Figure 6.1: Calculated rounding error

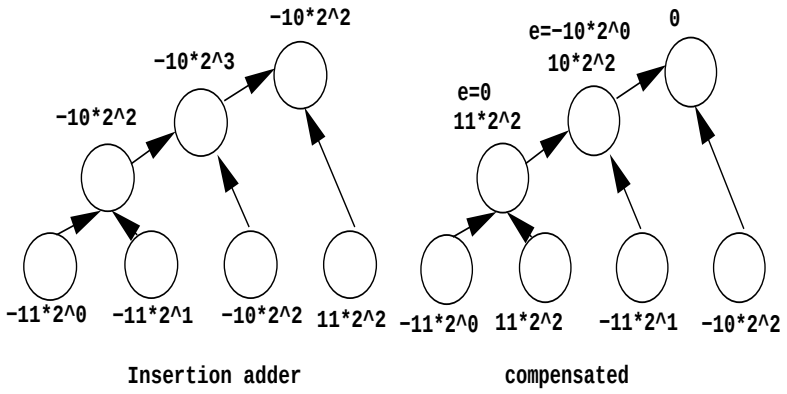


Figure 6.2: Calculated rounding error

methods	**answer**	$\|\mathbf{E}\|$	$\sum_{\mathbf{k}}^{\mathbf{n}} \|\mathbf{T_k}\|$
$ainc$	$-10 * 2^2$	$11 * 2^0$	$10 * 2^4$
$adec$	$-10 * 2^1$	$10 * 2^{-1}$	$10.01 * 2^2$
$psum$	$-10 * 2^1$	$10 * 2^{-1}$	$10 * 2^3$
$insert$	$-10 * 2^2$	$11 * 2^0$	$10 * 2^4$
$pair$	0	$10.1 * 2^1$	$11 * 2^3$
$+/- inc$	$-10 * 2^2$	$11 * 2^0$	$10 * 2^4$
acc	$-10 * 2^1$	$10 * 2^{-1}$	$10.01 * 2^2$
$compensated$	0	$10.1 * 2^1$	$-$

Table 6.1: Calculated error bounds for example

method	**Errorbound**
ainc	$\|E_n\| \leq (\|x_1\| + \|x_2\|)r_{n-1} + \sum_{i=3}^{n} \|x_i\| r_{n-i+1}$
psum	$\|E_n\| \leq \frac{u}{1-u} \sum_{k=2}^{n} \|\hat{S}_k\|$
pairwise	$\|E_n\| \leq r log_2 n \sum_{i}^{n} \|x_i\|$
insadd	$\|E_n\| \leq \frac{u}{1-u} \sum_{k=n+1}^{2n-1} \|\hat{T}_k\|$

Table 6.2: Error bounds and methods which minimize them

Under this assumption, we conclude that the **psum** error bound is sharper than the **ainc** error bound.

6.2.2 Comparison Between pairwise and insadd Error Bounds

Although this error bound is labeled as the **insadd** error bound, it minimizes error bound only when all addends have the same sign. Similarly, the **pairwise** error bound label is only valid when the number of addends is a power of two. The comparison between the two error bounds is similar to that in the previous section. The coefficient of the **insadd** error bound $(u/1-u)$ is equal to r_1. As r_i is strictly increasing in i, $r_1 \leq r_{\log_2 n}$. If we assume truncation is used for as the floating point rounding method, the relation $\sum |\hat{T}_k| \leq \sum |\hat{x}_k|$ will be hold. In this case, the **insadd** error bound is sharper than the **pairwise** error bound.

6.2.3 Comparison Between psum and insadd Error Bounds

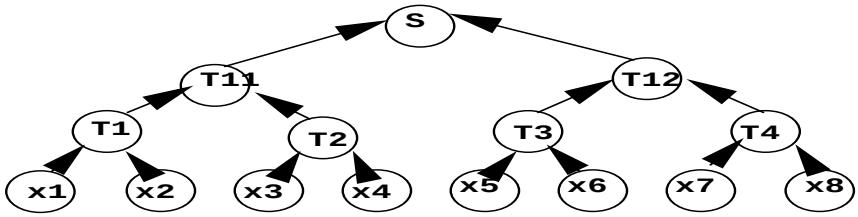
The difference between these two methods is essentially that between accumulation and tree summation. Both error bounds essentially emphasize the same thing: in order to minimize the error, minimize the sum of partial sums. If all the addends have the same sign, then **insadd** gives the best accuracy because it passes each addend through fewer steps of addition than **psum** does. Consider the example illustrated in Figure 6.3. For **insadd**, the sum of partial sums in Figure 6.3 is

$$\sum_{i=1}^{8} |\hat{S}_i| = (r_3 + r_2 + r_1) \sum_{i=1}^{8} x_i, \tag{6.1}$$

while for **psum** the sum of partial sums is

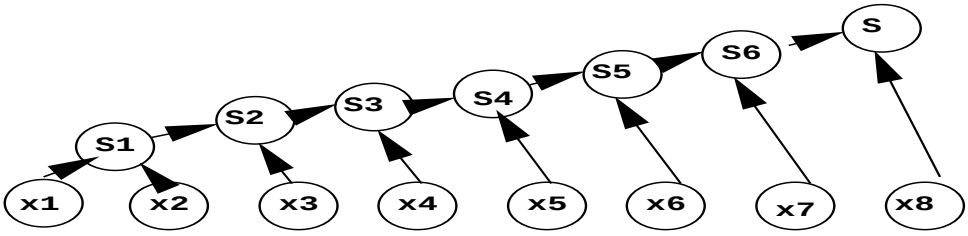
$$\sum_{i=1}^{8} |\hat{s}_i| = \sum_{i=1}^{7} r_i(x_1 + x_2) + \sum_{i=1}^{6} r_i x_3 + \sum_{i=1}^{5} r_i x_4 + \sum_{i=1}^{4} r_i x_5 + \sum_{i=1}^{3} r_i x_6 + \sum_{i=1}^{2} r_i x_7 + r_1 x_8 \tag{6.2}$$

Assume that all the addends have the same sign and compare the equations. If x_7 and x_8 are small enough, then Equation (6.1) is smaller than (6.2). The example here has only eight addends, but in practice there are thousands of addends, in which case the last two addends become a less important factor in the error. Overall we can conclude if all the addends have the same sign, **insadd** performs better at minimizing the error bound than **psum** method in that it makes sum of intermediate sum smaller.



Insertion adder

```
T1 = (x1 + x2)(1 + delta-1)
T2 = (x3 + x4)(1 + delta-2)
T3 = (x5 + x6)(1 + delta-3)
T4 = (x7 + x8)(1 + delta-4)
T11 = (T1 + T2)(1 + delta-11)
T12 = (T3 + T4)(1 + delta-12)
T  = (T11 + T12)(1 + delta21)
```



psum method

```
S1 = (x1 + x2)(1 + delta-1)
s2 = (s1 + x3)(1 + delta-2)
s3 = (s2 + x4)(1 + delta-3)
s4 = (s3 + x5)(1 + delta-4)
s5 = (s4 + x6)(1 + delta-5)
s6 = (s5 + x7)(1 + delta-6)
s  = (s6 + x8)(1 + delta-7)
```

Figure 6.3: Accumulation versus tree summation

7

Testing Results

7.1 Comparison Errors between All Methods

Several error bounds and algorithms which minimize those error bounds have been presented and explained. But the ultimate goal here is not to minimize error bounds, but to find out which algorithms minimize the *actual* error. Experiments are done to determine this. The methods compared here are **ainc**,**adec**, **psum**, **insadd**, **pairwise**, **pminc**, **pmpair** and **acc**.

7.1.1 Testing Methodology and Results

Each summation method are implemented in a dot product function for an iterative solver for large, sparse linear systems of equations. The Bi-CG stabilized algorithm [1] used to solve $Ax = b$ is shown in Algorithm 10. Here A is an $n \times n$ matrix and b a given n-vector.

Inside the while loop, five different kinds of dense dot products are called (sparse dot products may occur in the matrix-vector multiplication of line s 10 and 14, but those will be dealt with separately). The five dense dot products are

1. $p^T r$ (line 7)
2. $p^T v$ (line 11)
3. $t^T s$ (line 15)
4. $t^T t$ (line 15)
5. $r^T r$ (line 6).

Obviously 4 and 5 yield summations where addends all have the same sign. In order to determine accuracy, each method is implemented as a dot product called in place of the

Algorithm 10 Bi-CGstab

1: **Initialize:**
2: $r = b - A * x$
3: $p = r$
4: $v = q = 0$
5: $\omega_h = \beta = \alpha = 1$
6: **while** $r^T r > tol$ and $k < maxits$ **do**
7: $\beta_h = p^T r$; $\omega = (\beta_h \omega_h)/(\beta\alpha)$
8: $\beta = \beta_h$
9: $q = r + \omega(q - \alpha v)$
10: $v = Aq$
11: $\tau = p^T v$
12: $\omega_h = \beta_h / \tau$
13: $s = r - \omega_h v$
14: $t = As$
15: $\sigma = t^T s$; $\tau = t^T t$
16: $\alpha = \sigma / \tau$
17: $x = x + \omega_h q + \alpha s$
18: $r = s - \alpha t$
19: **end while**

usual library routine DDOT. In addition, accurate method with doubled (128-bit) precision is implemented and its result is used as the correct value for the dot product. Using this we calculate the relative error for each method for each iteration.

Figure 7.1 shows the average relative error for the five different kinds of dot products, when Bi-CGstab is run on a matrix from the Laplace operator on a uniform $24 \times 24 \times 24$ mesh . Note that types 1 and 2 have qualitatively similar relative errors and types 3, 4, and 5 are also clustered. This is because types 1 and 2 have oppositely signed addends up, while on the other hand, types 4 and 5 have addends with the same sign. However, the inner product $t^T s$ is unusual; although its terms can have different signs, it behaves more like the case where the terms are of the same sign. However, this is an artifact of the linear system. Note that the matrix A is positive definite, so

$$t^T s = s^T t = s^T A s = s^T R^T R s = y^T y, \tag{7.1}$$

where R is the Cholesky factor of A and $y = Rs$. Interestingly, dot product 3 in general behaves as if the summands were all of the same sign even when A is not positive definite, and when the summands do have differing signs. This is one of the reasons for testing dot product methods – the applications in which they occur can give special summations, which can be taken advantage of.

Notice that when there are oppositely signed addends, cancellation can take place. Thus the methods which make use of heavy cancellation tend to give better accuracy. In general,

	acc	**ainc**	**adec**	**psum**	**insadd**	**pairwise**	**pminc**	**pmpai**
Laplace	63	62	63	63	65	64	60	60
sherman5	1020	1025	1024	801	998	908	974	870
BFS	49	49	49	37	38	37	49	49

Table 7.1: Numbers of iterations for different dotproduct methods

when the dot product corresponds to two vectors which are approaching orthogonality as the algorithm procedes, heavy cancellation will occur. Such cases occur routinely in iterative methods. For example, in constrained optimization methods the standard stopping test is that the gradient vector of the objective function is orthogonal to the tangent plane of the constraint surface.

The two cases, all summands with the same sign and summands which have heavy cancellation, can be treated as extremes of a single measure: the cosine of the angle between the two vectors. When cosine of the two vector is near zero, it means that all the addends are nearly same sign. When cosine of the two vectors is near one, heavy cancellation must occur. Figures 7.2, 7.3, 7.4, 7.5, and 7.6 show the five different dotproducts for all iterations, with the relative error plotted against the cosine of the angle between the two vectors. In Figure 7.2, notice that the relative accuracy falls as angles approach 90 degrees. Methods **psum** and **acc** give the best accuracy in this case. This is reasonable, as both take into account the signs of the addends and encourage cancellation. Notice that **adec** and **pminc** have poor accuracy. Figure 7.3 shows similar results as 7.2, again because there are oppositely signed addends.

Figure 7.5 is the dot product of the vector t with itself. In this case, **insadd** and **acc** give good accuracy. Notice that **pairwise** and **pmpai** also give good accuracy. This is a reasonable result because when all the addends are of the same sign, it is important to reduce the number of addition to minimize the sum of partial sums. Methods **pairwise** and **insadd** tend to make each original addend take part in fewer additions than the other methods. Figure 7.6 shows similar result as 7.5, as it is also the inner product of residual vector with itself and all addends are same sign.

7.2 Effects on Number of Iterations

The accuracy of a dot product surely affect the number of iteration of iterative solver. Table 7.1 shows results for three different linear systems. LAPLACE is the Laplace operator matrix described earlier, SHERMAN5 is from the RUA section of the Harwell-Boeing test collection of sparse matrices [3], and BFS is from a steady-state two-dimensional fluid dynamics model of a backward-facing step. Note the dramatic variation in the numbers of iterations, even for a reasonably well-conditioned problem like LAPLACE. Although this

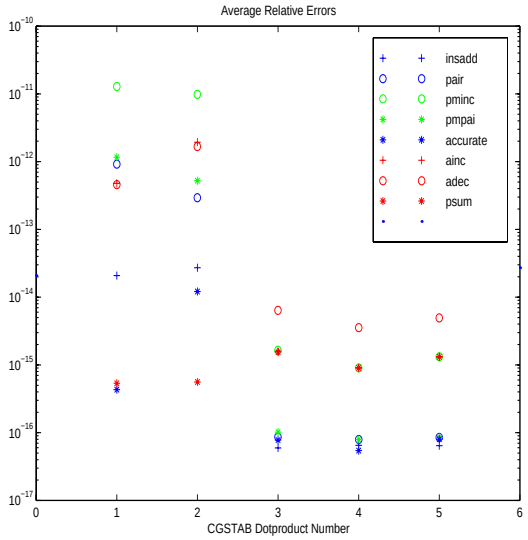


Figure 7.1: Average relative error for 5 different dotproducts (Laplace)

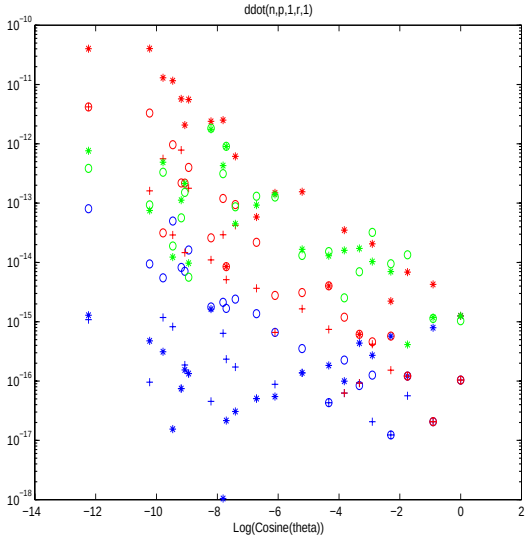


Figure 7.2: Relative error for $p^T r$ (Laplace)

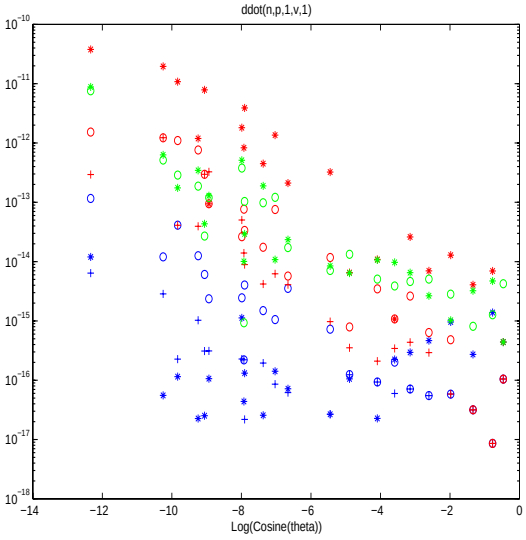


Figure 7.3: Relative error for $p^T v$

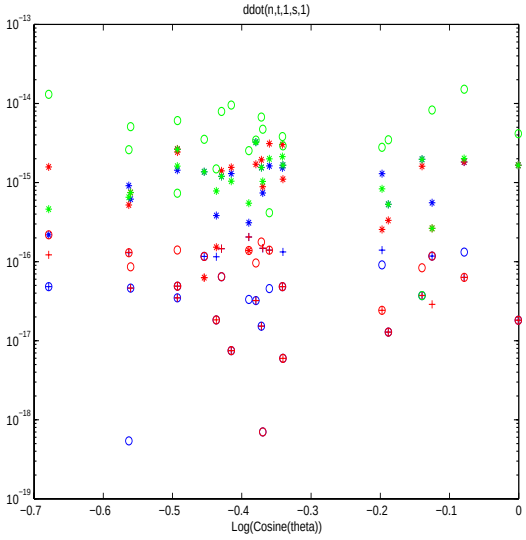


Figure 7.4: Relative error for $t^T s$

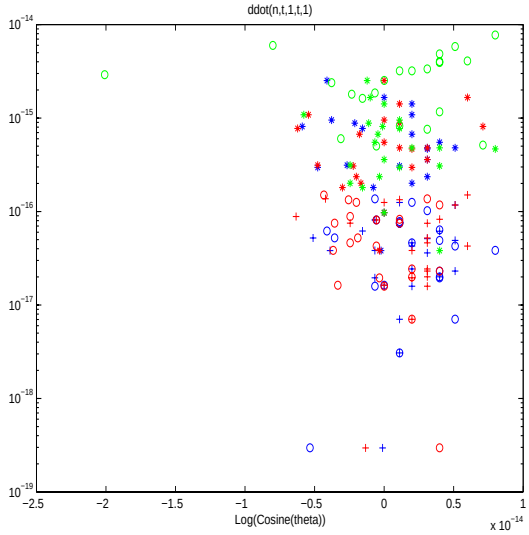


Figure 7.5: Relative error for $t^T t$

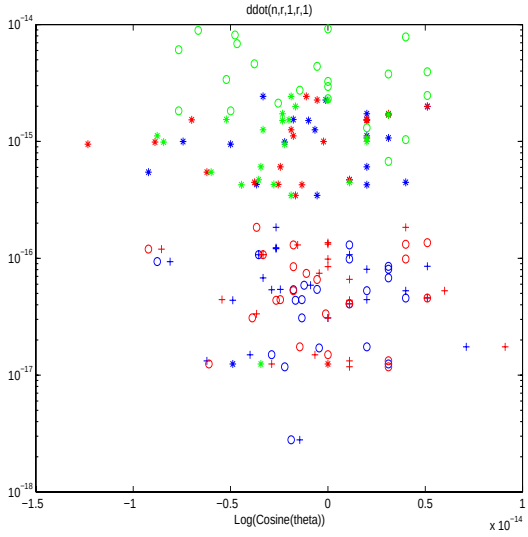


Figure 7.6: Relative error for $r^T r$

seem surprising, Figure 1.1 gives a clue why this is typical. Note that convergence is not monotone, and CG-like solvers, particularly for nonsymmetric systems, can have intermittent large growth in the residual norm as well as long periods with little decrease. Later on we examine the effects of the accuracy of the dotproduct on the numbers of iterations in more detail, but for now note that the fewest number of iterations does not necessarily occur using the most accurate dotproduct method.

7.3 Comparison between Original and Revised Method acc

Both versions of method **acc** are based on a bucket sort and are the same except for one significant difference. The revised version makes use of heavy cancellation, while the original version does not. Results show that the revision is more accurate. We will briefly review these algorithms below. After phase **opposite-dif** of the original version, each bucket contains addends with the same sign. However, some buckets contain positive addends and some buckets contain negative addends. Phase **same** of the original version starts by summing pairs in same bucket, from smaller exponents to larger exponents, until all the buckets contain at most one summand. On the other hand, after phase **opposite-dif** the revised version takes the largest exponent and scans from larger to smaller exponent buckets until it finds an addend of opposite sign, then adds the pair. This continues until all the buckets contain the same sign. This comparison is interesting, because for the original version, when adding pairs of the same sign and exponent, carrying error occurs on each add. On the other hand the revised version has shifting error when adding pairs of opposite sign and different exponent. If pairs of exponents differ greatly, heavy shifting

error occurs. In that case, the original version could be more accurate. But if the buckets contain addends distributed uniformly, it has more cancellation and so the revised version can perform better. Testing methodology for these conjectures will be presented in the following section.

Algorithm 11 Algorithm for version 1 (Original **acc**)

1: Bucket sort into buckets with same binary exponents.
2: Add pairs of opposite sign together in each bucket, putting results into correct bucket. **opposite-same** continue until every bucket has only same sign numbers.
3: Starting with smallest bucket, add pairs in it (putting answer into correct bucket) until it has only one number (or 0 numbers) left. **same** Do this all the way up the bucket list.
4: Add numbers starting from smallest bucket. **sumup**

Algorithm 12 Algorithm for version 2 (Revised **acc**)

1: Bucket sort into buckets with same binary exponents.
2: Add pairs of opposite sign together in each bucket, putting results into correct bucket. continue until every bucket has only same sign numbers.
3: Take elements from larger exponent bucket. Next, scan to left and find elements with opposite sign of the previous addend. Add elements together.Then,put result in appropriate bucket. If there is opposite sign of element in that bucket, add them together.
4: Add entries of same sign in same bucket. (After this, each entry has at most 1 entry.)
5: Add rest of entries from smallest exponent bucket to largest exponent bucket.
6: Add numbers starting from smallest bucket.

7.3.1 Testing Methodology

For comparing the errors, the dotproduct function is modified. to call three different dotproducts: version 1, version 2, and version 2 with long double. Version 2 with long double means that summands are stored in doubled (128 bit) precision. We take this as the "correct" answer. Then error is calculated by subtracting the result of each methods. The ddot algorithm discussed above is in Algorithm 13. For all the iterations of Bi-CGstab,

Algorithm 13 double ddot(x[],y[])

long double accurate ← ddot-version2-long(x,y);
double accurate-version1 ← ddot-version1(x,y);
double accurate-version2 ← ddot-version2(x,y);
double error-version1 ← accurate-version1 − accurate;
double error-version2 ← accurate-version2 − accurate;
return accurate-version2;

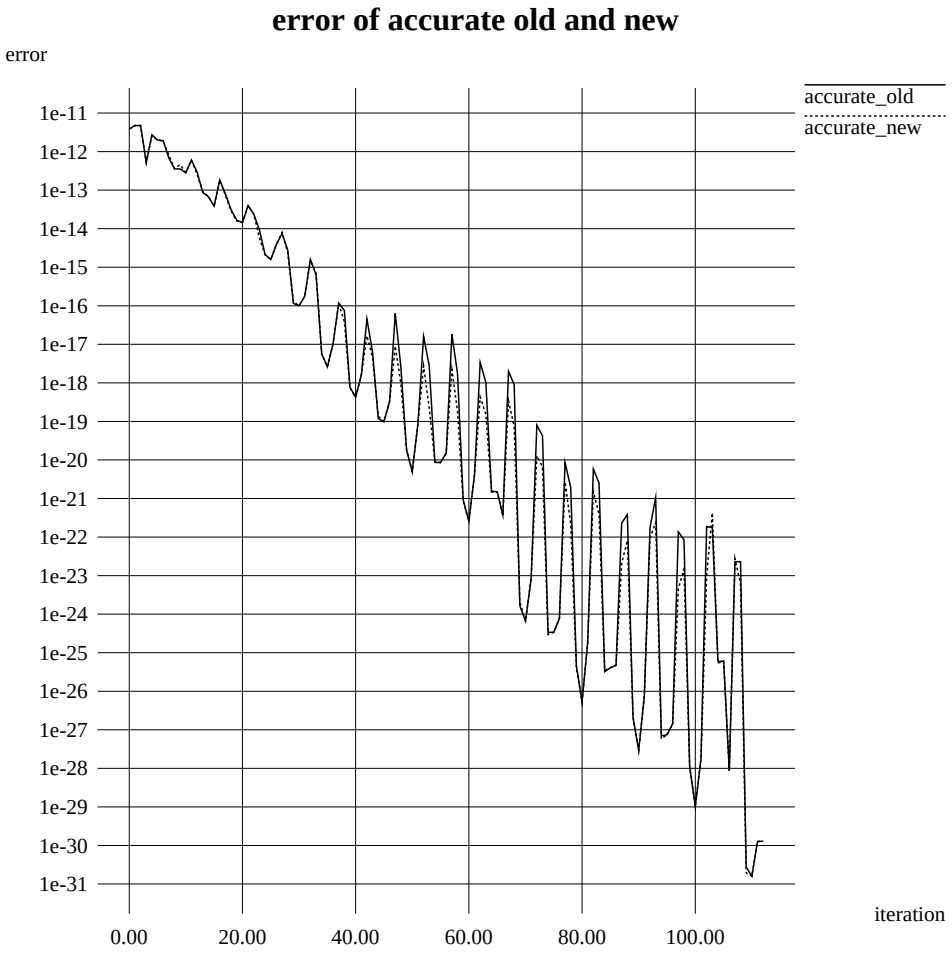


Figure 7.7: Comparison of versions 1 and 2 of method **acc** (Laplace)

error-version1 and error-version2 are saved as data. Figure 7.7 shows the result for the linear system Laplace. Notice that for most iterations, version 2 has less error. Two peak points occur in the graph. One is at iteration 62 where version 2 has less error. The other is at iteration 103, where version 1 has less error. We studied the input data at those two points carefully, in order to know how input data affects accuracy.

Local errors were accumulated in doubled precision, depending on where errors can occur in the two versions. For version 2, errors occur in **opposite-dif**, **same** and **sumup**. For version 1, errors occur in **same** and **sumup**. Table 7.2 shows the total errors accumulated this way during each phase for version 1 and version 2 at iteration 62.

This is a case where encouraging heavy cancellation is effective. As shown in Table 7.2, the error in the **same** phase for version 1 has larger exponent than that in the **same** and **opposite-dif** phases for version 2. Also, the error in **sumup** phase for version 1 has a

	opposite	**same**	**sumup**
$version2$	$4.108 * 10^{-19}$	$1.466 * 10^{-20}$	$4.153 * 10^{-21}$
$version1$	–	$3.331 * 10^{-18}$	$2.517 * 10^{-19}$

Table 7.2: Errors in each phase (iteration 62)

	opposite	**same**	**sumup**
$version2$	373	22	3
$version1$	–	537	7

Table 7.3: Numbers of additions in each phase (iteration 62)

larger exponent than that of version 2. The number of additions is another crucial factor for accuracy. Table 7.3 shows the number of additions in each phase.

Note in Table 7.3 that the number of additions in the **same** phase of version 2 is 22, whereas version 1 has 537. Also notice that the number of additions in the **opposite-dif** phase of version 2 is 373 which is closer to 537, but still has less error in that phase. This means that adding oppositely signed numbers has less error than adding same-sign numbers in general.

Histograms 7.8, 7.9, 7.10, and 7.11 show the distribution of addends in buckets after each phase of version 2. Histograms 7.8, 7.9, and 7.12 show the distribution of addends in buckets after each phase of version 1. The buckets are arranged according to IEEE biased floating point exponent for double precision numbers, and so an exponent of zero corresponds to bucket number 1023. By comparing histograms 7.8 and 7.9, notice that number of addends are not reduced as much after adding oppositely signed numbers with the same exponent. But comparing histograms 7.9 and 7.10 shows that the numbers of addends are greatly reduced after adding oppositely signed numbers with different exponents, and also the largest bucket in 7.9 is 1012 and the largest bucket in 7.10 is 1009. In this case it is likely that cancellation occurs in the largest exponent buckets. On the other hand, in historgram 7.12 the largest exponent is 1019, which is larger than the largest bucket at the beginning. This is caused by carrying from the phase of adding same sign numbers with the same exponent. Also, the numbers are clustered locally and this makes rounding error larger when added in increasing absolute order. From this it is clear that version 2 has great potential to give more accurate answer than version 1, since it encourages heavy cancellation.

Next consider the case where version 1 performs better than version 2, which is iteration at 103. Table 7.4 shows that the **same** and **sumup** phases of version 2 have errors of 0, but version 2 has more error in total than version 1. Next consider Table 7.5. Here the number

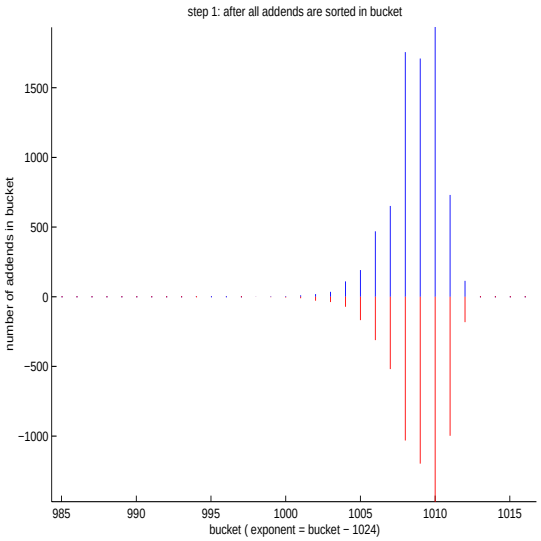


Figure 7.8: addends at the beginning

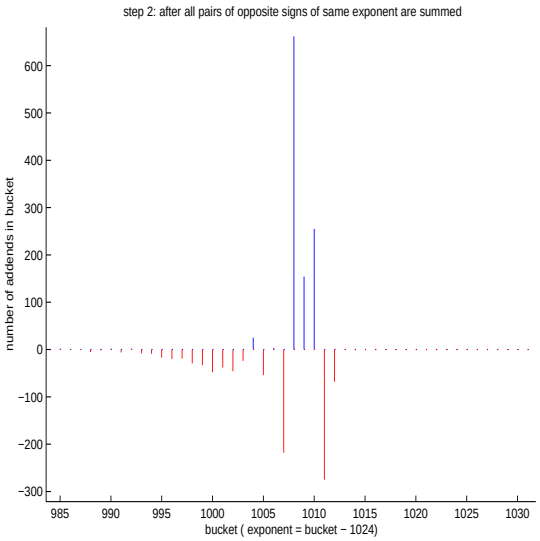


Figure 7.9: After **opposite-same** phase

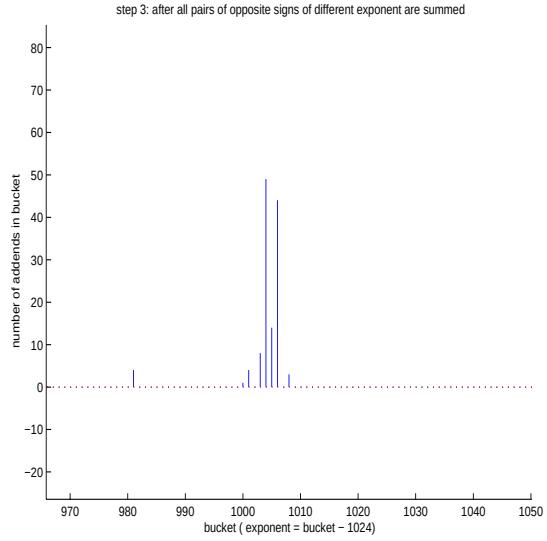


Figure 7.10: After **opposite-dif** phase

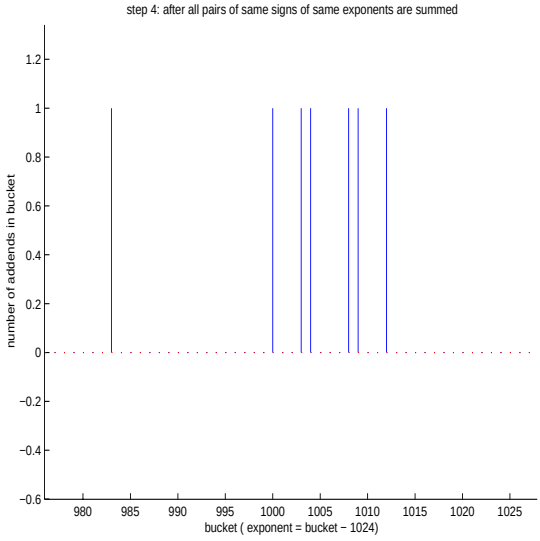


Figure 7.11: After **same** pahse (for version 2)

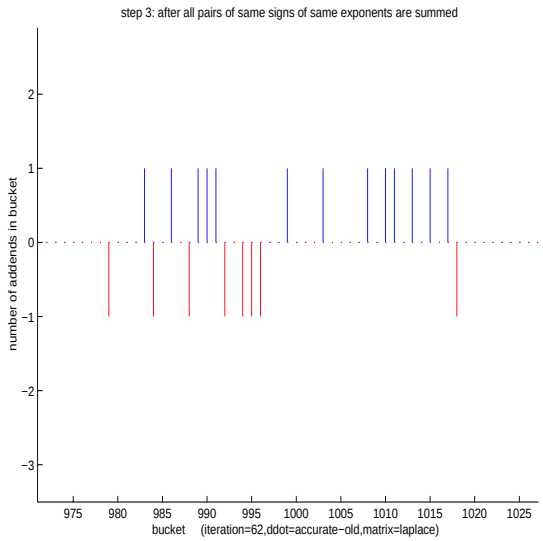


Figure 7.12: After **same** phase (for version 1)

of additions at the **opposite-dif** phase of version 2 is twice as much as that of the **same** phase of version 1. Now consider the corresponding distribution histograms. Histograms 7.13, 7.14, 7.15, and 7.16 show the distribution of addends in buckets after each phase for version 2. Histograms 7.13, 7.14, and 7.17 show the same for version 1. Histogram 7.14 is important; it shows a large number of clustered negative addends and only one bucket of positive addends. For positive addends, it has noticeably large number of addends at bucket 998. For small buckets between 960 and 975, there are a few positive and negative addends distributed. Contrast this histogram with 7.9, which does not have large cluster like in 7.14. This cluster causes unnecessary additions in the **opposite-dif** phase, which increases the number of additions 1166. That is the weakness of version 2. The version works by scanning through oppositely signed addends until it finds one, so that exponents between two addends sometimes differ greatly. This kind of addition occurs often for this case.

There is better solution to reduce error in this phase. After scanning through oppositely signed addend and it finds one, we will not add these two addends immidiately, because exponent of these 2 addends might differ greatly. In stead, Take the addend of smaller exponent, and find oppositely signed and larger exponent to this addend. We will do addition between this addend and newly found one.

We could observe the tendency from the histogram that if positive and negative addends are distributed uniformly around largest exponents, heavy cancellation works well. But if one sign clustered around largest exponent, and the opposite sign is not much, it will end up many of redundant additions and cause large round error. Figures 7.18 and 7.19 show the errors at each addition during **opposite-dif** phase. The point which has sharp negative slope and goes upwards in the next addition is where the algorithm goes past the bucket

	opposite − dif	**same**	**sumup**
$version2$	$4.260 * 10^{-22}$	0.0	0.0
$version1$	–	$2.265 * 10^{-22}$	$2.732 * 10^{-23}$

Table 7.4: Errors in each phase (iteration 103)

	opposite − dif	**same**	**sumup**
$version2$	1166	0	0
$version1$	–	538	6

Table 7.5: Numbers of additions in each phase (iteration 103)

to search the next largest exponent. Recall that the algorithm works by going from the largest to smallest exponents, doing this repeatedly until all the buckets have the same sign. Comparing these graphs, notice that in 7.19 the difference of exponents for both addends is larger than that in 7.18. Also note that the number of additions is much larger than in 7.18, of course, we can not simply compare the values, because these are at different iteration and have different addends.

7.4 Effect of Sparse Dot Product on Number of Iteration

Algorithm 10 has two matrix-vector multiplications per iteration in addition to the five dense products, and those products between a sparse matrix and (typically) dense vector can also be implemented as a set of sparse dot products. These differ greatly from the dense dotproducts; linear systems of modern research interest typically are of order $n = \ calO(10^5)$ or $\ calO(10^6)$, which is then the length of the five dense dotproducts. The sparse dotproducts, however, are typically of length $\ calO(10^2)$ or less - but there are $2n$ such dotproducts on each iteration. Because of this, $\ calO(n \log n)$ methods such as **psum** can be practical. A particular question to be answered is which of the two classes of dotproduct has the largest effect on the numbers of iterations. This section examines the effect of sparse dot products on number of iterations for solving linear system. The testing methodology is the same as in the last section.

Six matrixes are tested, with characteristics shown in Table 7.7. For each matrix, four different kind of combinations of dense dot product and sparse dot product is tested, which are:

- without replacement of the dense product and without replacement of the sparse dot product(**d0s0**)

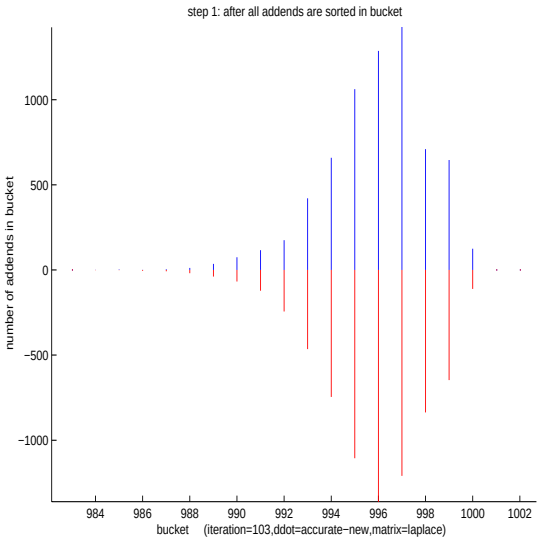


Figure 7.13: addends before process

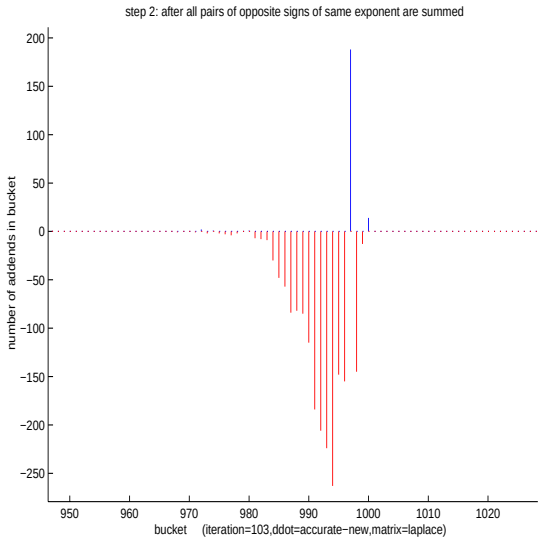


Figure 7.14: After **opposite-same** phase

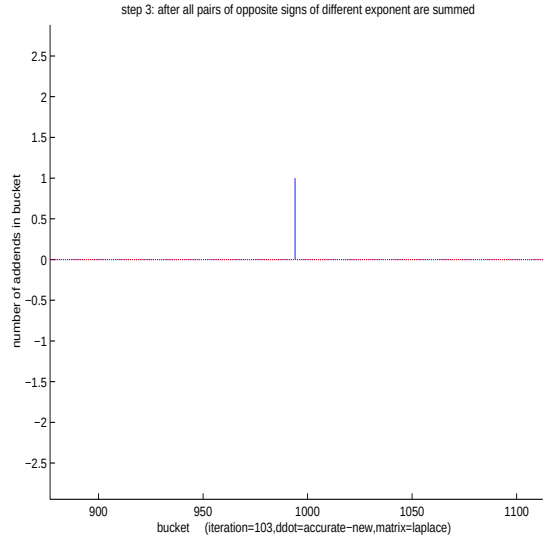


Figure 7.15: After **opposite-dif** phase

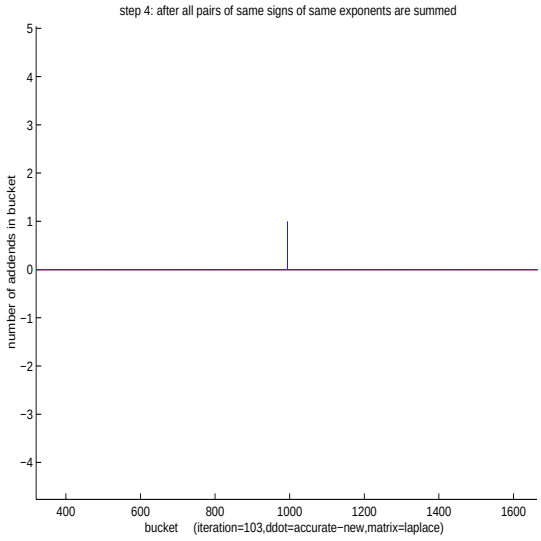


Figure 7.16: After **same** pahse (for version 2)

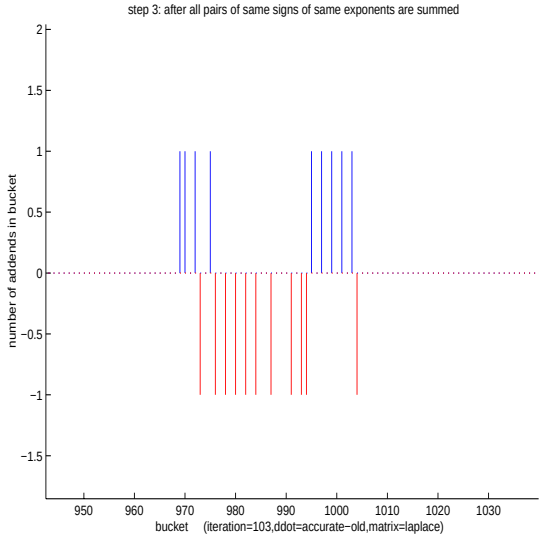


Figure 7.17: After **same** phase (for version 1)

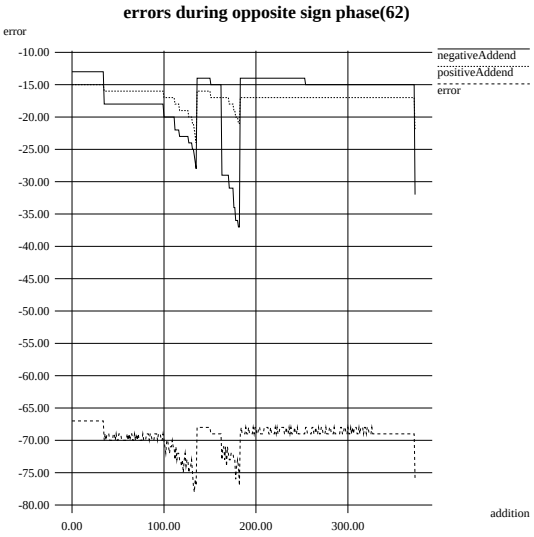


Figure 7.18: Errors at each addition in **opposite-dif** phase (iteraton at 62)

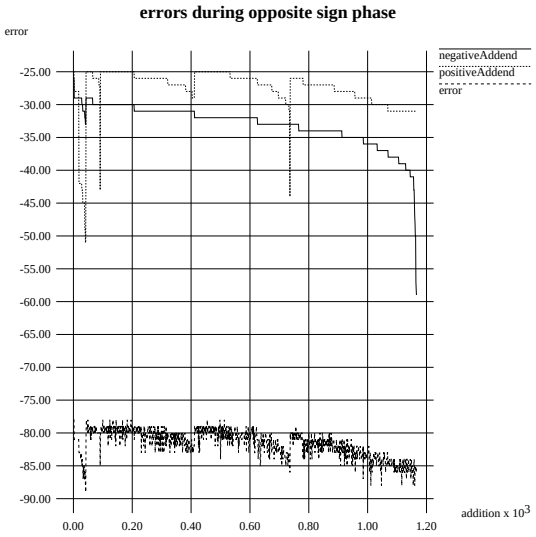


Figure 7.19: Errors at each addition in **opposite-dif** phase (iteration at 103)

- with replacement of the dense product and without replacement of the sparse dot product (**d1s0**)
- without replacement of the sparse dot product and with replacement of the dot product(**d0s1**)
- with replacement of the sparse dot product and with replacement of the dot product(**d0s1**).

When the dotproduct is not replaced with one of the methods described earlier, the usual BLAS routine ddot() is called instead. For each combination, eight different kind of summation methods are tested, the same as in the last section. Results are shown in Table 7.6, where numbers in bold face show the fewest number of iteration achieved in that row. Table 7.8 shows the counts of minimum or maximum numbers of iterations by row.Although methods **acc** and **psum** achieve the best accuracy, these methods do not always achieve the fewest number of iterations. On the contrary, method **acc** often requires the maximum number of iterations. We can conclude that improving accuracy in the dot products does not necessarily lessen the number of iterations required.

Tables 7.9 show counts of minimal or maximal numbers of iterations by column. These tables show that **d1s0** or **d1s1** are slightly better in minimizing the number of iterations required. If dotproduct accuracy were the determinant in reducing the numbers of iterations required, **d1s1** should give the minimum number of iterations. But again this is not the case.

Finally consider which dot product type (sparse or dense) has more impact to affect the number of iterations. Table 7.7 shows the characteristics of each matrices tested. In this table, **avg** stands for number of nonzeros in each row, and **ratio1** stands for the ratio of average vector length of dense dot product over sparse dot product. **Ratio2** stands for the ratio of number of function call of sparse dot product over dense dot product. **Ratio3** stands for product of **ratio1** and **ratio2** of sparse dot over dense dot product. **ratio3** shows that sparse dot products have more impact on the numbers of iterations than dense dot products, since its value is 8 times higher than that of dense dot product.

Figures 7.20, 7.21, 7.22, 7.23, 7.25, and 7.26 show the degree of impact for dot products. Plots for sparse dotproduct are obtained by taking differences between number of iterations of **d1s1** and **d1s0**, divided by number of iterations for **d0s0**. The plots for dense dotproducts are obtained by taking difference between number of iterations of **d1s1** and **d0s1**, divided by number of iteration of **d0s0**. From these figures, it is hard to determine which dot product has more impact on number of iterations. In some case, sparse dot has more impact than dense dot product, and in other case, it is opposite, contrary to the prediction by **ratio3**.

In Table 7.7, the maximum absolute value of sparse plots and dense plots are shown as **dense**,**sparse** respectively. The maximum differences between sparse and dense are shown in Figure **diff**. There is tendency that the larger matrices, the larger is **dense**. Also, the

Matrix	*Dotprods*	**acc**	**ainc**	**adec**	**psum**	**insadd**	**pair**	**pminc**	**pmpai**
bfs.5019	**d0s1**	**1399**	1420	1432	1423	1428	1419	1420	1421
	d1s0	1432	1381	1408	1385	**1380**	1381	1427	1394
	d1s1	1344	1416	1415	1375	1370	**1332**	1426	1404
bfs.20284	**d0s1**	586	506	571	581	554	549	**497**	572
	d1s0	514	513	537	546	549	**467**	502	597
	d1s1	545	540	604	537	**494**	510	500	505
laplace.13824	**d0s1**	46	46	46	46	49	51	49	**43**
	d1s0	43	49	48	**41**	42	46	47	42
	d1s1	46	49	48	46	**42**	46	47	**42**
bfs.5936	**d0s1**	322	329	321	**313**	323	329	324	321
	d1s0	321	**312**	322	321	323	319	322	321
	d1s1	326	322	323	**313**	323	321	321	320
bfs.11997	**d0s1**	511	505	**497**	530	499	507	532	505
	d1s0	511	502	508	503	510	**496**	499	505
	d1s1	**497**	506	501	499	505	515	523	505
garon.3175	**d0s1**	117	120	112	112	**104**	135	111	122
	d1s0	114	**109**	116	121	112	122	**109**	114
	d1s1	129	115	116	**109**	114	111	119	112

Table 7.6: Numbers of iterations with sparse/dense dot product replacement

more nonzero in matrix, the larger is **sparse**. It is natural since more addends in ddot products, it is likely to have more rounding errors and lead to vary the number of iteration.

matrices	nonzero	avg	ratio1	ratio2	ratio3	dense	sparse	diff
$bfs.5019$	105615	21	239	2007	8	0.0393	0.0629	0.1022
$bfs.20284$	452752	22	922	8113	8.9	0.1324	0.1818	0.3124
$laplace.13824$	93312	7	1975	5529	2.7	0.1429	0.1020	0.1429
$bfs.5936$	128692	21	282	2374	8.6	0.0256	0.0319	0.0288
$bfs.11997$	264408	22	545	4799	8.8	0.0621	0.1924	0.1944
$garon.3175$	88927	28	113	1270	11.2	0.1951	0.1220	0.2846

Table 7.7: Dimension and number of nonzeros for test matrices

min/max	acc	ainc	adec	psum	insadd	pair	pminc	pmpai
minimum	2	2	1	4	4	3	2	2
maximum	4	5	2	0	2	4	1	1

Table 7.8: Counts of minimum/maximum numbers of iterations by row

min/max	dotp's	acc	ainc	adec	psum	insadd	pair	pminc	pmpai	total
minimum	**d0s1**	0	2	4	1	2	0	2	0	11
	d1s0	4	4	2	1	1	4	3	0	17
	d1s1	2	0	0	5	3	3	2	3	18
maximum	**d0s1**	3	3	1	4	3	5	3	4	22
	d1s0	2	1	3	2	1	0	2	2	13
	d1s1	3	3	4	1	1	1	1	0	14

Table 7.9: Counts of minimum/maximum numbers of iterations by column

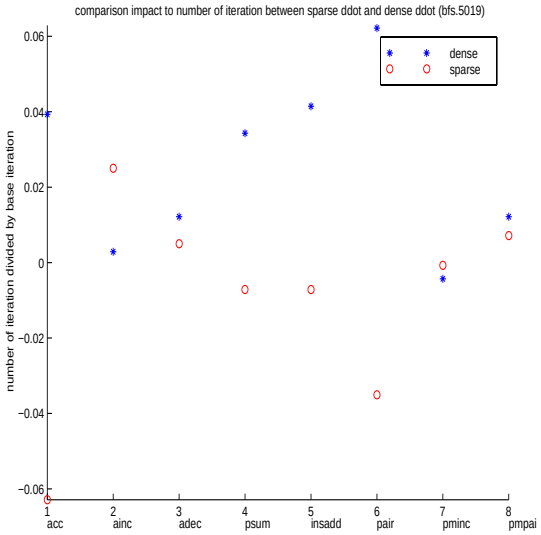


Figure 7.20:

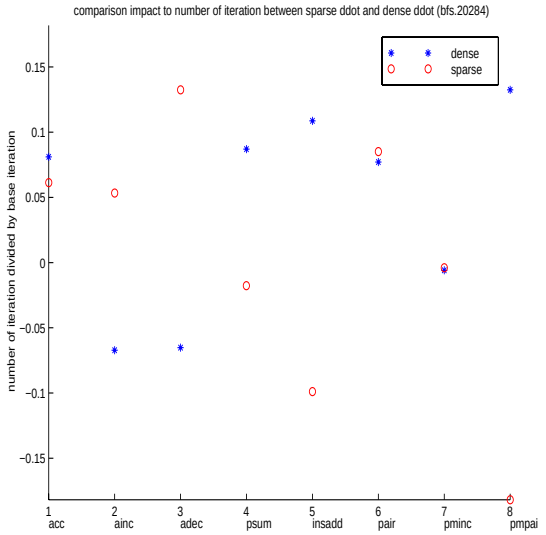


Figure 7.21:

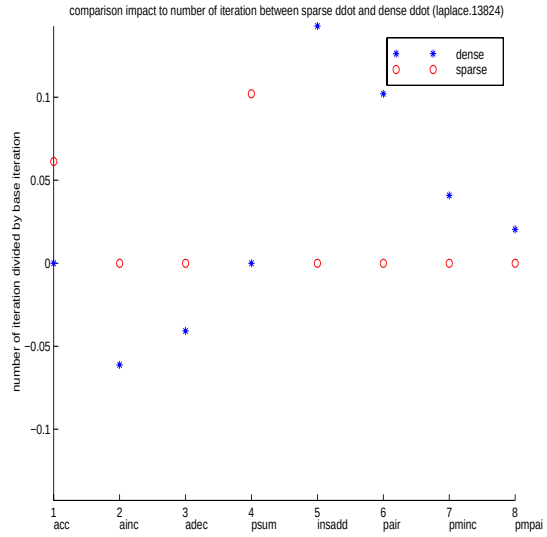


Figure 7.22:

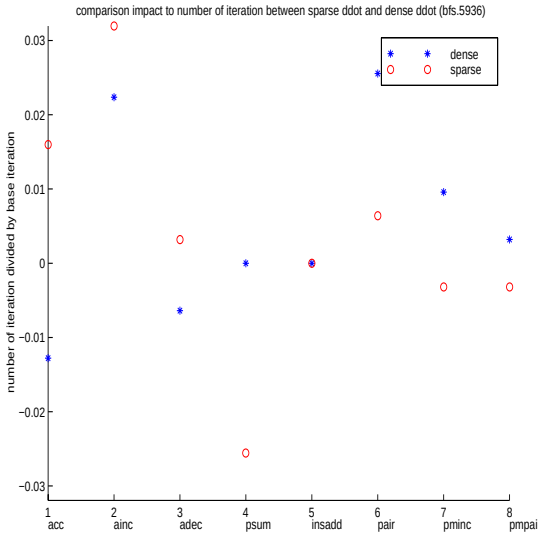


Figure 7.23:

Figure 7.24: Deviations of number of iteration by sparse dot product or dense dot product

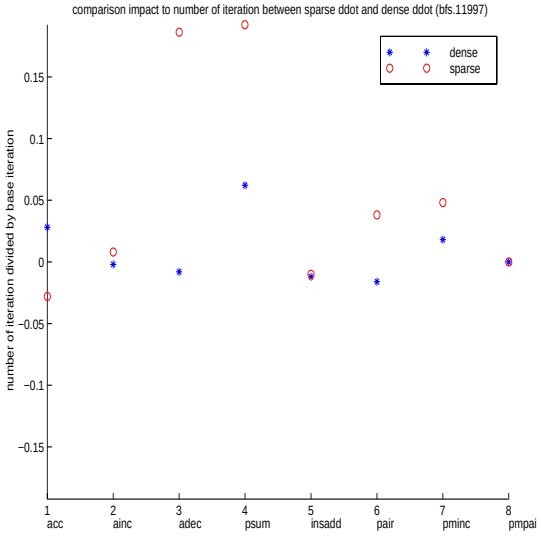


Figure 7.25:

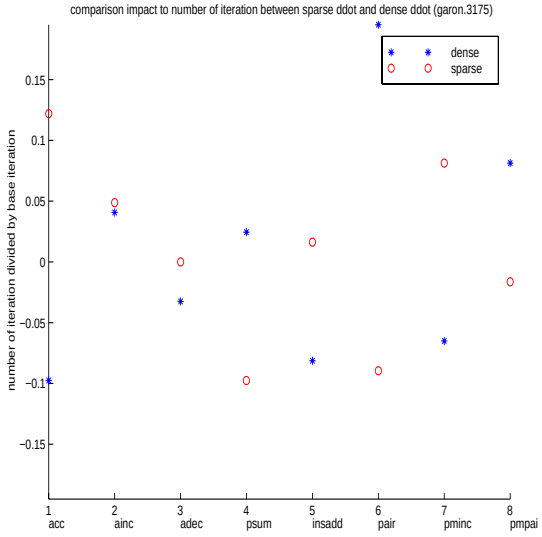


Figure 7.26:

Figure 7.27: Deviations of number of iteration by sparse dot product or dense dot product

8

Conclusion and Further Work

Several summation methods, including a new and more accurate method **acc**, are studied in terms of analytical error bounds and numerical experiments. Results show that methods and **acc** and **psum** give the best accuracy. Both the error bounds and experiments show that in general the method which minimize the sum of partial sums gives the best accuracy. Furthermore, the accuracy of methods depends on the signs of the addends. If all addends are of the same sign, minimizing the number of additions each addend takes part in helps minimizes the sum of partial sums. So in this case insertion addition method **insadd** gives better accuracy than **psum**. In general tree summation gives better accuracy than accumulated summation when all addends are same sign. On the other hand, when addends have differing signs, encouraging heavy cancellation becomes the crucial factor in minimizing rounding error. Thus **psum** or **acc** method performs better than **insadd** in this case. Also, in general **adec** gives better accuracy than **ainc**, since it provides more chances for cancellation. Altough, it has been believed for some scientists and engineers that heavy cancellation should be avoided, our experiments show the opposite results. Making use of heavy cancellation is the single most important for accuracy. Detailed experiments with two versions of method **acc** further showed the importance of heavy cancellation.

The other main question addressed was whether reducing the rounding errors in dot-products also reduces the numbers of iterations in iterative solvers. This was tested for dense and sparse dot products in Bi-CGstab. There was no straightword correspondence between the accuracy of dot products and the numbers of iterations required. In many cases a more accurate dotproduct ended up causing more iterations in the solver.

The accuracy of the dotproducts depends, not surprisingly, on the characteristics of the summands. This was quantified by using the angle between the vectors as a measure that continously goes between the extremes of a dotproduct of a vector with itself (or a scaled copy of itself) to the dotproduct of two orthogonal vectors. Different methods were superior for those two extremes. Since in most scientific computation the characteristics of the vectors are known in advance, this suggests that a user might productively call several different dotproduct functions in a single application, choosing the one to use based on the application.

Even though the time cost of the algorithms was not addressed in this dissertation, time analysis becomes important factor when methods such as **acc** are used for practical applications. For example, in a linear system solver of order $n = 10000$ the vector length in dense dotproducts is usually 10000, which is also the number of addends. For the same problem, the sparse dot products may have average vector length of only 20, but the number of function calls is order of 20000 per iteration. In both cases, the difference between $\mathcal{O}(n)$ and $\mathcal{O}(n \log n)$ leads to large differences for the total computational time. More efficient algorithms are needed for further study of dotproduct summation orderings *in situ*.

Bibliography

[1] H. V. der Vorst, *Bi–CGSTAB: A fast and smoothly converging variant of Bi–CG for the solution of nonsymmetric linear systems*, SIAM Journal of Scientific and Statistical Computing, 13 (1992), pp. 631–644.

[2] D.R.Ross, *Reducing truncation errors using cascading accumulators*, Comm. ACM, (1965), pp. 32–33.

[3] I. S. Duff, R. G. Grimes, and J. G. Lewis, *"Users Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)"*, tech. rep., Cedex and Boeing Computer Services, October 1992.

[4] N. J. Higham, *The accuracy of floating point summation*, SIAM J. SCI.Comput., 14 (1993), pp. 783–799.

[5] ——, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996, pp. 87–101.

[6] J.H.Wilkinson, *Error analysis of floating-point computation*, Numerische Mathematik, (1960), pp. 319–340.

[7] J.M.Wolfe, *Reducing truncation errors by programming*, Comm. ACM, (1964), pp. 335–356.

[8] W. Kahan, *Implementation of algorithms*, Tech. Rep. 20, Dept. of Computer Science University of California, Berkeley, 1980.

[9] ——, *IEEE standard 754 for binary floating-point arithematic*, Lecture Notes on the Status of IEEE 754, (1996).

[10] D. Knuth, *The art of computer programming*, Addison-Wesley, 1981, pp. 572–573.

[11] L. Lasdon, *Optimal design of efficient acoustic antenna arrays*, Math. Prog, (1987), pp. 131–155.

[12] L.C.W.Dixon and D.J.Mills, *Effect of rounding errors on the variable metric method*, Journal of Optimization Theory and Applications, (1994), pp. 175–179.

[13] K. L.Kobbelt, *A fast dot-prodct algorithm with minimal rounding errors*, Computing, (1994), pp. 355–369.

[14] M. A. Malcolm, *On accurate floating-point summation*, Communications of the ACM, 14 (1971), pp. 731–736.

[15] D. McCracken, *Numerical Methods and Fortran Programming*, John Wiley, 1964, pp. 61–63.

[16] I. Stegun and M. Abramowitz, *Pitfalls in computation*, J. Soc. Indust. Appl. Math, (1956), pp. 207–219.

[17] U.W.Kulisch and P. Miranker, *The arithmetic of the digital computer: A new approach*, SIAM Review, 28 (1986), pp. 1–11.