Computer System Support for Scientific and Engineering Computation

Lecture 20 - July 7, 1988 (notes revised June 14, 1990)

1 Multiple Precision Floating Point

1.1 Why Multiple Precision

We mentioned earlier that the QR algorithm and compensated summation as clever algorithms that can avoid the need for multiple precision. Here is another example. Consider a linear system with feedback that is characterized by a linear operator A. Typically, we would like to diagonalize A (that is, $A = C\Lambda C^{-1}$) where the eigenvalues of A are the elements of Λ. The eigenvalues (which may be complex numbers) give a simple way of understanding the system. However, the eigenvalues don't correspond to anything physically, and may change dramatically when the system is slightly perturbed. In fact, the matrix can change from being diagonalizable to not being diagonalizable. Here are two examples. Suppose A is the identity matrix, and we perturb it slightly by adding 10^{-40} above the diagonal.

$$B = \begin{pmatrix} 1 & 10^{-40} & 0 & & & \\ 0 & 1 & 10^{-40} & & & \\ 0 & 0 & 1 & & & \\ & & & \ddots & & \\ & & & & 1 & 10^{-40} \\ & & & & 0 & 1 \end{pmatrix}$$

Its easy to see that B is no longer diagonalizable. If it were, then $C\Lambda C^{-1} = B$ and $\det(B - \alpha I) = \det(C)\det(\Lambda - \alpha I)\det(C)^{-1} = \det(\Lambda - \alpha I) = \prod(\lambda_i - \alpha)$. On the other hand, direct computation gives $\det(B - \alpha I) = (1 - \alpha)^n$. Since these two polynomials must the same, $\lambda_i = 1$ and $\Lambda = I$, which is a contradiction. Thus B is not diagonalizable. A second example of a small perturbation changing a matrix from nondiagonalizable to diagonalizable is to start with

$$A = \begin{pmatrix} 1 & 10^{10} & 0 & & & \\ 0 & 1 & 10^{10} & & & \\ 0 & 0 & 1 & & & \\ & & & \ddots & & \\ & & & & 1 & 10^{10} \\ & & & & 0 & 1 \end{pmatrix}.$$

362

which is not diagonalizable, and perturb it by adding 10^{-50} to the lower left hand corner.

$$B = \begin{pmatrix} 1 & 10^{10} & 0 & & & \\ 0 & 1 & 10^{10} & & & \\ 0 & 0 & 1 & \ddots & & \\ & & & \ddots & & \\ & & & & 1 & 10^{10} \\ 10^{-50} & & & & 0 & 1 \end{pmatrix}$$

Letting $\gamma_i = \sqrt[n]{10^{-50}(10^{10})^{n-1}}$ be the complex nth roots of 10^{10n-60}, the eigenvalues of this matrix are $1 - \gamma_i$. Since these are all distinct, the matrix B is diagonalizable.

Geometrically, let S be the surface of diagonalizable matrices with double eigenvalues in the space of all matrices.[1] The closer a matrix gets to this surface, the harder it is to diagonalize it. In fact, this is an example of the general phenomenon we have mentioned before, that the amount of ill-conditioning is roughly inversely proportional to the distance to the surface of ill conditioned problems. Unfortunately, optimizing linear systems tends to drive their matrices towards this unstable surface. Thus multiple precision is necessary if the systems are to be analyzed by diagonalizing their matrices in a straightforward way. There are clever techniques available for avoiding multiple precision. Some of them are discussed in the book *Sensitivity Analysis in Linear Systems* by Assem Deif (1986).

1.2 Double Precision Addition

In the last lecture we explained how to do multiple precision without "bit-twiddling", that is, as a portable program that will work on any machine with reasonable arithmetic. A multiple precision number was represented by a list of working precision numbers. The sum of two such lists could be collapsed using the distillation algorithm. In this section we will discuss the special case of "double precision", that is, double the working precision, where working precision is the widest precision supported by the hardware. As before, we will use $a+b$ to mean exact addition, and $[a+b]$ to mean machine addition (i.e. rounded to working precision), and similarly for subtraction. We will represent our double precision numbers as pairs (x, ξ), where x and ξ are both working precision numbers, and are disjoint, which means that $x = [x + \xi]$. First recall

Algorithm 1 (Single + Single) *Given x_1 and x_2 in working precision, $x_1 + x_2$ is represented by (z, ζ), where in fact $x_1 + x_2$ is exactly equal to the disjoint sum $z + \zeta$. The decomposition is computed using $z = [x_1 + x_2]$ and $\zeta = [[x_{max} - z] + x_{min}]$. Here, x_{max} is the x_i with the largest absolute value, x_{min} the smallest absolute value.*

We can use this to build up an algorithm for adding a single precision number to a double precision one. The notation $(z, \zeta) \leftarrow x_1 + x_2$ is shorthand for applying this algorithm.

Algorithm 2 (Double + Single) *The sum $(x, \xi) + y$ is represented by (z, ζ), where $|(z + \zeta) - (x + \xi + y)| < \epsilon^2 |x + \xi + y|$, $\epsilon = \beta^{p-1}$ is an ulp of 1.0. If $|y| > |x|$, then swap x and y. If $|y| < |\xi|$ then swap y and ξ, so that $|\xi| \le |y| \le |x|$. Let $t = [[\xi + y] + x]$ and $\tau = [[[x - t] + y] + \xi]$. Then $(z, \zeta) \leftarrow t + \tau$.*

The proof is unpublished, but a number of people have examined it and believe it is correct. Finally, the algorithm for adding two double precision numbers.

[1]Actually, you need to take the closure of this set.

Algorithm 3 (Double + Double, W. Kahan) *The sum $(x, \xi) + (y, \eta)$ is represented by (z, ζ), and $z + \zeta = (x+\xi)(1+\epsilon_1) + (y+\eta)(1+\epsilon_2)$, $|\epsilon_i| \le k\epsilon^2$, k is a small integer and ϵ is 1 ulp as before. Swap (x, ξ) and (y, η) if necessary so that $|x| \ge |y|$. Let $t = [[\xi + y] + x]$, $\tau = [[[[x - t] + y] + \xi] + \eta]$. Then $(z, \zeta) \leftarrow t + \tau$.*

This algorithm has probably been proved to be correct, but may only work for IEEE 754 arithmetic, because it appears to require round to even. An algorithm that works on any hardware is

Algorithm 4 (Double + Double, W. J. Cody) *The sum $(x, \xi) + (y, \eta)$ is represented by (z, ζ), with the accuracy of algorithm 3, where*

$$\begin{aligned}
s &= \max(|x|, |y|) \cdot \beta^2 \\
\Xi &= s \cdot \mathrm{signum}(x) \\
x' &= [\Xi - [\Xi - x]]; \quad \xi' = [[x - x'] + \xi] \\
N &= s \cdot \mathrm{signum}(y) \\
y' &= [N - [N - y]]; \quad \eta' = [[y - y'] + \eta] \\
t &= [x' + y'] \\
\tau &= [[[[x' - t] + y'] + \eta'] + \xi'] \\
z &= [t + \tau]; \quad \zeta = [[t - z] + \tau].
\end{aligned}$$

This algorithm requires about twice as many operations as the previous one, including a multiplication. Both algorithms can produce an answer where z is in error by 1 ulp, in fact where z has every digit wrong!

$$\begin{aligned}
x &= 1010. \\
\xi &= -0.1000 \\
x + \xi &= 1001.1 \\
y &= -1001. \\
\eta &= -0.000001 \\
y + \eta &= -1001.000001 \\
(x + \xi) + (y + \eta) &= 0.011111
\end{aligned}$$

Kahan's algorithm gives $\zeta = t = 0$, $\tau = 0.1$, $z = 0.1000$, $\zeta = 0$. Cody's algorithm gives $x' = x$, $\xi' = \xi$, $y' = -1000$, $\eta' = -1.000$, $t = 10.$, $\tau = -1.1$, $z = 0.1000$, $\zeta = 0$. In each case $z = 0.1000$ instead of .01111. An algorithm that is believed always to have z correct follows.

Algorithm 5 (Accurate Double + Double, W. Kahan) *The sum $(x, \xi) + (y, \eta)$ is represented by (z, ζ), computed as follows. Begin by swapping (x, ξ) and (y, η) if necessary so that $|x| \ge |y|$. Let $t = [[\xi + y] + x]$. If $t = 0$ then {let $t = [[x + y] + \xi]$, and $(z, \zeta) \leftarrow t + \eta$}. Otherwise let $u = [[x - t] + y]$. If $u = 0$ and $\eta \ne 0$ then {replace (x, ξ) with (x', ξ'), where $(x', \xi') \leftarrow t + \xi$. Replace (y, η) with $(0, \eta)$. Go back to the beginning}. Otherwise let $\tau = [[u + \xi] + \eta]$. Then $(z, \zeta) \leftarrow t + \tau$.*

1.3 Multiplication

In order to multiply two multiple precision numbers, all we need is a way of representing a working precision number as a sum of two half precision numbers. That is, suppose the precision is p, then the number $.x_1x_2 \ldots x_p$ can be written as the sum of $.x_1x_2 \ldots x_{p/2}$ and

$.0\cdots 0x_{p/2+1}\ldots x_p$. As long as p is even, the product of two numbers of this form fits exactly into a working precision number. So to multiply x by y exactly, write $x = x_1 + x_2$ and $y = y_1 + y_2$ and the exact product is $xy = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2$. One of the addition algorithms from the previous section can be used to collapse the sum. When p is odd, this simple splitting method won't work. However we can gain an extra bit by using negative numbers. For example, if $p = 5$ and $x = .10111$, we can split it as $x_1 = .11$ and $x_2 = -.00001$. There is more than one way to split a number: we want to pick a splitting that is easy to compute.

Algorithm 6 (Split, Dekker and Veltkamp) *Let p be the precision, $k = \lceil p/2 \rceil$ be half the precision (rounded up), and $m = 2^k + 1$. Then $x = x_1 + x_2$ with x_i representable using $\lfloor p/2 \rfloor$ bits of precision is computed with $x_1 = [[m\cdot x] - [[m\cdot x] - x]$, $x_2 = [x - x_1]$.*

When the radix β is 2, this works for any precision. For $\beta > 2$, p must be even. In order to use this algorithm, we need a way to compute m.

Algorithm 7 *Compute in working precision $b = 1/|(4.0/3.0 - 1.0)3.0 - 1.0|$, $r = \sqrt{4.0\cdot b}$, and $m = ((r + rb) - rb) + 1.0$. Then b is exactly 2^{p-1}, and m is exactly $2^k + 1$.*

To verify algorithm 7, write 4/3 in binary as $1.010101_2\cdots$. If p is odd, then in working precision $4/3 = 1.01\cdots01_2$, $4/3-1$ is $.01\cdots010_2$, $3(4/3-1)$ is $.11\cdots110_2$ so $|3(4/3-1)-1|$ is exactly $.00\cdots010_2 = 2^{-p+1}$. The calculation works the same way when p is even. Next, observe that $r = \sqrt{4b} = \sqrt{2^{p+1}}$. If p is odd, then $p = 2k - 1$ so $r = 2^k$. In this case $r + rb = 2^k + 2^{k+p-1}$ can be represented exactly in working precision, so that $((r + rb) - rb) + 1.0 = 1 + r = 1 + 2^k$ exactly.

Finally, assume p is even, so that $p = 2k$. Then $r = \sqrt{4b} = \sqrt{2^{2k+1}} = \sqrt{2}\cdot 2^k$. In binary, $\sqrt{2} = 1.0110_2\cdots$. We need to compute what $[[r + rb] - rb]$ will be in working precision. For simplicity, scale everything by 2^{-k}, so that rb rounded to working precision is an integer. Then $r + rb = \sqrt{2} + \sqrt{2}2^{p-1}$ looks like this

```
rb  10110...xxx.
r             1.0110...
```

so that $[r + rb]$ is exactly $rb + 1$, and $[[r + rb] - rb]$ is exactly 1. Or since we scaled everything by 2^{-k}, $((r + rb) - rb)$ in working precision is 2^k, so just as when p is odd, again $((r + rb) - rb) + 1.0 = 1 + 2^k$, which shows that algorithm 7 is correct.

Algorithm 6 is based on the following fact.

Algorithm 8 (Round) *Let $0 < k < p$, and set $m = 2^k + 1$. Then $[[m\cdot x] - [[m\cdot x] - x]]$ is exactly equal to x rounded to $p - k$ significant digits.*

It is easy to see why algorithm 8 implies algorithm 6. The computation of x_1 will be x rounded to $p - k = \lfloor p/2 \rfloor$ places. If there is no carry out, then certainly x_1 can be represented with $\lfloor p/2 \rfloor$ significant bits. If there is a carry out, then the low order bit of x_1 must be zero, so again x_1 is representable in $\lfloor p/2 \rfloor$ bits.

What about x_2? We can assume that x is an integer satisfying $2^{p-1} \le x \le 2^p - 1$. Let $x = x_h + x_j$ where x_h is the $p - k$ high order bits of x, and x_j is the k low order bits. If $x_j < 2^{k-1}$, then rounding x to $p - k$ places is the same as chopping and $x_1 = x_h$, and $x_2 = x_j < 2^{k-1}$ is representable with $k - 1 \le \lfloor p/2 \rfloor$ significant bits. If $x_j > 2^{k-1}$ then computing x_1 involves rounding up, so $x_1 = x_h + 2^k$, and $x_2 = x - x_1 = x - x_h - 2^k = x_j - 2^k$, thus $|x_2| < 2^{k-1}$ so can be represented with $k - 1 \le \lfloor p/2 \rfloor$ bits. If $x_j = 2^{k-1}$ and computing

x_1 doesn't round up, then $x_1 = x_h$ and $x_2 = 2^{k-1}$ can be represented with 1 significant bit. Finally, if $x_j = 2^{k-1}$ and x_1 does round up, then $x_2 = 2^{k-1} - 2^k$ which can also be represented with 1 significant bit.

Finally, we will show how to prove the correctness of algorithm 8 (and thus of algorithm 6). The proof breaks up into two cases, depending on whether or not the computation of $mx = 2^k x + x$ has a carry-out or not.

Assume there is no carry out. Then the computation of $mx = x + 2^k x$ looks like this:

```
           aa...aabb...bb
+ aa...aabb...bb
  zz    ...    zzbb...bb
```

where we have partitioned x into two parts. The low order k digits are marked b and the high order $p-k$ digits are marked a. To compute $[m \cdot x]$ from mx involves rounding off the last k digits (the ones marked with b) so

$$[m \cdot x] = mx - x \bmod(2^k) + r2^k. \tag{1}$$

The value of r is 1 if .bb$\cdots$b is greater than $\frac{1}{2}$ and 0 otherwise. That is

$$r = 1 \text{ if .bb}\cdots\text{b} > \frac{1}{2} \text{ or if .bb...b} = \frac{1}{2} \text{ and } a_0 = 1. \tag{2}$$

Next we compute $[m \cdot x] - x = mx - x \bmod(2^k) + r2^k - x = 2^k(x + r) - x \bmod(2^k)$. Below we show the computation of $[m \cdot x] - x$ rounded, that is, $[[m \cdot x] - x]$.

```
aa...aabb...bB00...00
         -  bb...bb
           + r
zz    ...   zzZ00...00
```

If .bb$\cdots$b $< \frac{1}{2}$ then $r = 0$, and when subtracting there is a borrow from the digit marked B, but rounding to p adds 1 back into that place. The rounded result of the subtraction is thus $2^k x$. If .bb$\cdots$b $> \frac{1}{2}$ then $r = 1$, 1 is subtracted from B because of the borrow, and rounding the answer to p places truncates, so again the result is $2^k x$. Finally if .bb$\cdots$b $= \frac{1}{2}$, the digit Z will be forced to 0 independent of the value of r, and since the low order digit of .bb$\cdots$b is also zero, once again the result is $2^k x$. To summarize

$$[[m \cdot x] - x] = 2^k x. \tag{3}$$

Combining equations (1) and (3) gives $[[m \cdot x] - [[m \cdot x] - x]] = x - x \bmod(2^k) + r2^k$. The result of performing this computation is

```
        r
aa...aabb...bb
      - bb...bb
aa...aa00...00
```

The rule for computing r, equation (2), is the same as the rule for rounding a$\cdots$ab$\cdots$b to $p-k$ places. Thus computing $mx - (mx - x))$ in working precision is exactly equal to rounding x to $p-k$ places, in the case when $x + 2^k x$ does not carry out.

When $x + 2^k x$ does carryout, then $mx = 2^k x + x$ looks like this:

```
           aa...aabb...bb
+ aa...aabb...bb
  zzz   ...    zwbb...bb
```

366

Thus $[m \cdot x] = mx - x \bmod(2^k) + w2^k$. The reason for adding $w2^k$ is that if $w = 1$, then you need to subtract off $w2^k$, but you also round up thus adding $w2^{k+1}$ for a net gain of $w2^k$. Next, $[m \cdot x] - x = 2^k x - x \bmod(2^k) + w2^k$. In a picture

```
aa...aabb...bb00...00
          -  bb...bb
          + w
---------------------
```

Rounding gives $[[m \cdot x] - x] = 2^k x + w2^k - r2^k$, where $r = 1$ if $.\mathrm{bb}\cdots\mathrm{b} > \frac{1}{2}$ or if $.\mathrm{bb}\cdots\mathrm{b} = \frac{1}{2}$ and $a_0 = 1$. Finally, $[m \cdot x] - [[m - x] - x] = mx - x \bmod(2^k) + w2^k - (2^k x + w2^k - r2^k) = x - x \bmod(2^k) + r2^k$. And once again, $r = 1$ exactly when rounding $\mathrm{a}\cdots\mathrm{ab}\cdots\mathrm{b}$ to $p - k$ places involves rounding up. Thus algorithm 8 is proved in all cases.

2 Gaussian Elimination

A few comments on the appendix *Gaussian Elimination with Extra-precise Accumulation of Products.* Both HUPA and LUPA use extra-precise accumulation of inner products. LUPA is the traditional algorithm, and does 2 reads and 1 write in its inner loop. HUPA is a more clever algorithm, and only requires two reads in its inner loop. In addition, HUPA has much better paging performance. HUPA improves paging performance over LUPA in much the same way as the column oriented matrix multiply formula discussed in lecture 16 improves paging performance over the obvious multiply algorithm.

Another feature of HUPA and LUPA is that they never fail because the input matrix is singular. If they encounter a pivot of 0, they replace it with machine epsilon. The reason for this is so that the user can determine where the singularity is, namely, which column is a linear combination (or almost a linear combination) of other columns.

There is a method called *preconditioning* that can change the equation $A\vec{x} = \vec{b}$ to $A'\vec{x} = b'$ with the same solution, but where A' is less singular than A. It requires the inexact flag, because when dealing with close to singular matrices, a small roundoff error can make a big change to the solution, so it is important that when preconditioning, no roundoff error is incurred. The inexact flag has a historical precursor in the IBM 7090 and 7094. In those machines, arithmetic operations produced an answer in two registers, the second register containing low order bits. Thus the computation was inexact just when the second register was non-zero.[2]

[2]With one exception, namely that if $a \ll b$, then $a + b$ set the second register to zero. However, in this case, the inexactness didn't matter.

(0)

" $Z := x + y$ in D.P. "

1.0. <u>Given</u> x and y in Working Precision,

<u>Compute</u> $Z = z + \zeta = (z, \zeta) = x + y$

Program:

```
if |x| < |y| then swap(x, y) ;
z := x+y ;                 ... rounded to W.P.
ζ := (x-z)+y .             ... exact.
```

<u>VALID</u> for IEEE 754
DEC VAX
IBM 370

<u>NOT</u> for H-P calculators (rounded decimal)
Burroughs B65xx (rounded octal)
CRAYs, Cybers (no guard digit)
T.I. calculators (" " ")

Kahan
7 July 88

__Given:__ $X \doteq x + \xi \equiv (x, \xi)$ and y,

__Compute__ $Z = z + \zeta = (z, \zeta) \doteq X + y$,

and do so accurately enough that

$|Z - (X+y)| \le \varepsilon^2 |X+y|$

where $\varepsilon = 1.000\cdots001 - 1$ in Working Precision.

__Program__:

```
If $|y| > |x|$ then swap($x$, $y$)
        else if $|y| < |\xi|$ then swap($y$, $\xi$);
... now $|x| \ge |y| \ge |\xi|$.
$t := (\xi + y) + x$ ; ... computed in W.P.
$\tau := ((x-t) + y) + \xi$ ; ... " " "
$Z := t + \tau$ .          ... to D.P.
... i.e. $z := t + \tau$ ; $\zeta := (t - z) + \tau$ .

... Now $Z = z + \zeta = (z, \zeta) \doteq X + y$
```

__Applications__: • $\Sigma_j \, y_j$
• Crude D.P. product

<u>Given</u>: $X = x + \xi \equiv (x, \xi)$ and
$Y = y + \eta \equiv (y, \eta)$;

<u>Compute</u> $Z = z + \zeta \equiv (z, \zeta) \doteqdot X + Y$ roughly
but no worse than if X and Y
were each perturbed by factors
like $1 \pm (\text{small integer}) \cdot \varepsilon^2$, where
$\varepsilon = 1.000\ldots001 - 1$ in Working Precision

W. Kahan's program (for IEEE 754) :

```
If $|x| < |y|$ then swap$(X, Y)$ ;   ... so $|x| \ge |y|$
$t := (\xi + y) + x$ ;   ... computed in W.P.
$\tau := (((x - t) + y) + \xi) + \eta$ ;...   "   "   "
$Z := t + \tau$ .   ... to D.P.
```

W.J. Cody's program (for arbitrary floating-point)

```
$s := \max\{|x|, |y|\} \cdot (\text{radix})^2$ ;
$\Xi := \text{sign}(s, x)$ ;
$x' := \Xi - (\Xi - x)$ ;   $\xi' := (x - x') + \xi$ ;
$N := \text{sign}(s, y)$ ;
$y' := N - (N - y)$ ;   $\eta' := (y - y') + \eta$ ;
$t := x' + y'$
$\tau := ((x' - t) + y') + \eta' + \xi'$
$z := t + \tau$ ;   $\zeta := (t - z) + \tau$ .
```

Example: to 4 sig. bits.

$X = 1001.1$: $x = 1010.$ $\xi = -0.1000$

$Y = -1001.000001$: $y = -1001.$ $\eta = -0.000001$

Correct $X+Y = 0.011111$.

W.K.'s algorithm : $s = t = 0,$ $\tau = 0.1,$ $z = Z = 0.1000$

W.J.C's algorithm : $x' = x$ $\xi' = \xi$

$y' = -1000$ $\eta' = -1.000$

$t = 10.$ $\tau = -1.1$

$z = Z = 0.1000$ $s = 0.$

Given: $X = x+\xi = (x, \xi)$ and

$Y = y+\eta = (y, \eta)$,

Compute $Z = z+\zeta = (z, \zeta) := X+Y$ to D.P.

W.K.'s program:

```
Start:  If $|x| < |y|$ then swap $(X, Y)$ ;
        $t := (\xi + y) + x$ ;          ... computed in W.P.
        if $t = 0$ then
            { $t := (x-y) + \xi$ ;       ... exact in W.P.
              $Z := t + \eta$ ;          ... to D.P.
              return } ;
        $u := (x-t) + y$ ;              ... to W.P
        if $u = 0$ and $\eta \neq 0$ then
            { $X := t + \xi$ ;           ... to D.P.
              $Y := \eta$ ;
              go ~~back~~ to Start } ;
        $\tau := (u + \xi) + \eta$ ;       ... in W.P.
        $Z := t + \tau$ .               ... in D.P.
```

1 572

Heads & Tails :

$h(x)$:= x rounded to ½ · Working Precision
$\theta(x)$:= $x - h(x)$

$$\boxed{\quad x \quad} = \boxed{h(x)} + \boxed{\theta(x)}$$

Computing them : $\boxed{\text{IN BINARY}}$

If $2\varepsilon = 1.000\ldots001 - 1 = 2^{1-2k}$ or 2^{2-2k}
let $m = 2^k + 1 = \boxed{\underbrace{000\ldots00}\underbrace{100\cdots001}}$

Then $h(x) := (x - m \cdot x) + m \cdot x$; ... in W.P.
$\theta(x) := x - h(x)$.

Computing ε and m :

$\varepsilon := |\,(2.0 / 3.0 - 0.5) * 3.0 - 0.5\,|$.

$b := 1/|\,(4.0/3.0 - 1.0) * 3.0 - 1.0\,|$; ... $= 1/(2\varepsilon$
$r := \sqrt{(4.0 * b)}$;
$m := ((r + r \cdot b) - r \cdot b) + 1.0$.