

Various issues associated with finite precision arithmetic

Mike Giles

`mike.giles@maths.ox.ac.uk`

Oxford University Mathematical Institute
Oxford-Man Institute for Quantative Finance
Oxford eResearch Centre

Acknowledgments: support from Oxford-Man Institute, NVIDIA, EPSRC

Overview

- catastrophic cancellation
- binary tree summation
- compensated summation
- low-level maths libraries
 - fixed-point iterations
 - polynomial expansions

Catastrophic cancellation

In my own experience, the biggest problems come from amplification of machine rounding error due to almost perfect cancellation, computing $a - b$ when $a \approx b$.

These can sometimes be avoided by reformulating the mathematics, using an approach which is exactly equivalent mathematically but avoids the cancellation.

Catastrophic cancellation

Example: computing the roots of

$$a x^2 + b x + c = 0$$

when $|a c| \ll b^2$. Roots are obviously

$$x = \frac{-b \pm \sqrt{b^2 - 4 a c}}{2 a}$$

but the smaller root will suffer from very bad rounding error if computed this way. For this one, it is much better to use

$$x = \frac{2 c}{-b \mp \sqrt{b^2 - 4 a c}}$$

to avoid near total cancellation.

Binary tree summation

The second most important set of problems I've experienced come from very large summations, summing a large set of numbers

e.g. in Monte Carlo may want to average the outcome of 10^6 simulations

$$\bar{P} = N^{-1} \sum_{n=1}^N P_n$$

The seemingly trivial question is how to compute $\sum_{n=1}^N P_n$?

Binary tree summation

The naive treatment is to use an accumulator, adding each number one at a time:

$$S_0 = 0$$

$$S_n = S_{n-1} + P_n, \quad n = 1, 2, \dots, N$$

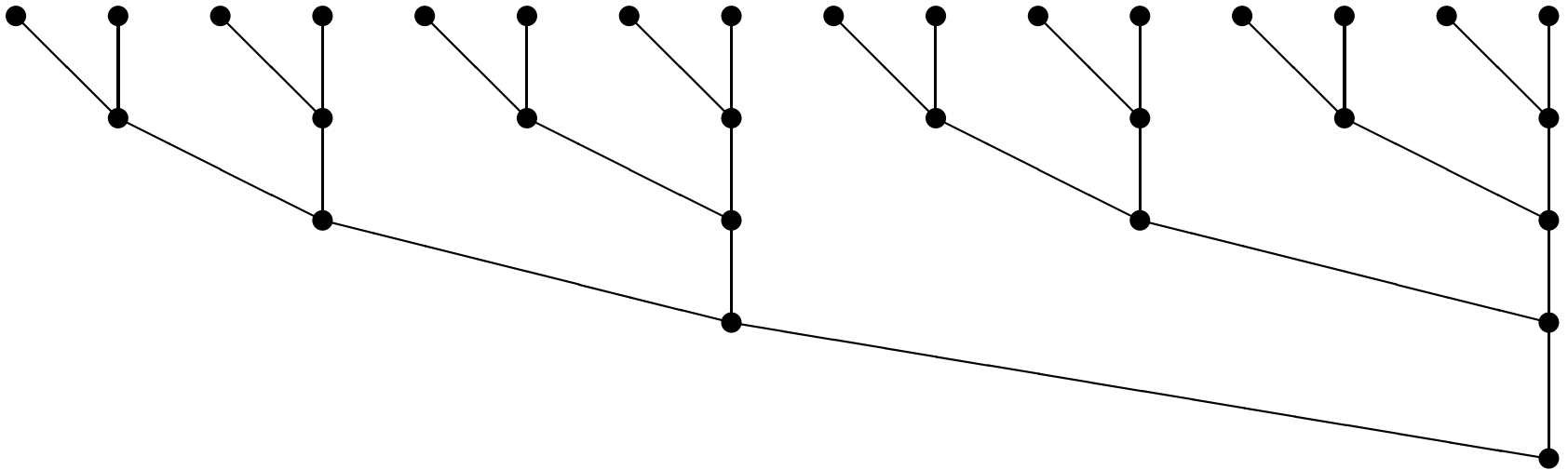
The error introduced at the n^{th} step is roughly $\varepsilon |S_{n-1}| Z_n$ where ε is the f.p. accuracy and Z_n can be modelled as a zero mean random number.

When the P_n are all positive and of similar size, this leads to an r.m.s. error which is $O(\varepsilon \sqrt{N} S_N)$ so we get a \sqrt{N} amplification of the natural error.

Binary tree summation

How do we avoid this?

By summing the numbers in pairs, then summing those pairs in pairs, and so on, recursively, in a binary tree.



Can implement this using $\log_2 N$ storage – see example code on my webpage.

Binary tree summation

This way, the rounding errors are amplified by factor $\log_2 N$ since there are $\log_2 N$ stages, and at each stage we are adding numbers of comparable magnitude.

This approach is also very useful for parallelisation (e.g. for execution on GPUs) and is the basis for the parallel scan algorithm for computing all of the partial sums S_n .

Similar, but less severe, problems occur for large matrix multiplications and the numerical solution of ODEs.

Compensated summation

This is another approach which in words is:

- try to add P_n to the accumulator (but do so inexactly)
- calculate what you have actually added (often exactly)
- calculate what you failed to add, and add it onto the next item to be added, P_{n+1}

In code it becomes:

```
 $S_0 := 0; \quad err := 0;$   
for ( $n = 1, \dots, N$ ) {  
     $P_n := P_n + err;$   
     $S_n := S_{n-1} + P_n;$   
     $err := P_n - (S_n - S_{n-1});$   
}
```

Compensated summation

- this often (usually?) gives full precision, avoiding any accumulation of rounding error
- it's more costly, but the cost is usually negligible
- I still prefer the binary tree summation, probably because of its parallel nature

For more information on both methods, see Nick Higham's book or SIAM paper (available on MRSN webpage).

Low-level maths libraries

Chip circuits are designed for multiplication and addition.

Have you wondered how $1/x$, y/x , \sqrt{x} , $\exp(x)$, $\cos x$ and other similar operations are performed?

In most cases, the methods fall into two categories:

- fixed point iterations
- polynomial expansions

Fixed point iteration

These are of the form

$$z_{n+1} = f(z_n)$$

designed so that computing $f(z_n)$ only involves addition and multiplication, and z_n converges to the required value.

Because errors are naturally corrected, it is well suited for iterative refinement, doing the initial calculations in single precision and then switching to double precision.

The convergence is usually quadratic (Newton iteration) so usually only two double precision iterations are needed.

Fixed point iteration

• Inverse: $z = x^{-1}$

$$z_{n+1} = z_n - z_n(z_n x - 1),$$

which is a Newton iteration applied to

$$\frac{1}{zy} - 1 = 0.$$

If x has the FP representation

$$x = 2^m(1 - \varepsilon), \quad 0 \leq \varepsilon < 1/2.$$

then z_0 can be initialised as $z_0 = 2^{-m}(1 + \varepsilon)$ or through a lookup table.

Fixed point iteration

- Division: $z = y/x$

$$z_{n+1} = z_n - y^{-1}(z_n x - y)$$

is a Newton iteration applied to

$$\frac{y}{z x} - 1 = 0.$$

z_0 can be initialised by

$$z_0 = y * (x^{-1})$$

Fixed point iteration

- Inverse square root: $z = x^{-1/2}$

$$z_{n+1} = z_n - \frac{1}{2}z_n (z_n^2 x - 1)$$

- Square root: $z = x^{1/2}$

$$z_{n+1} = z_n - \frac{1}{2}x^{-1/2}(z_n^2 - x)$$

One point to look out for when using iterative refinement is that the range of SP numbers is smaller than DP numbers.

Fixed point iteration

Even integer division and remainder can be done with a fixed point iteration.

To compute $p = n/m$ and $q = n \% m$ for $m, n > 0$, set $p_0 = 0$, $q_0 = n$ and then use a few iterations of

$$p_{n+1} = p_n + [m^{-1}q_n], \quad q_{n+1} = q_n - m [m^{-1}q_n],$$

where m^{-1} is an approximate FP value and $[\cdot]$ rounds to the nearest integer.

This reduces q_n to the approximate range $[-m/2, m/2]$ and a final correction is made if $q_n < 0$.

Polynomial expansions

This class of methods approximates a function $f(x)$ in one of the following ways:

- high degree polynomial $p(x)$
- high degree rational approximation $p_1(x)/p_2(x)$

The domain of x is usually split into multiple sections with a separate approximation on each.

The accuracy of the approximations is chosen to machine the FP accuracy, so double precision needs more terms than single precision.

Polynomial expansions

- The exponential function can be expressed as

$$\exp(x) = 2^{x/\log 2}.$$

Writing $x/\log 2 = x \log_2 e = n + r$, where n is the nearest integer and r is the remainder, with $|r| \leq \frac{1}{2}$, gives

$$\exp(x) = 2^n \exp(s),$$

where $s = r \log 2 = x - n \log 2$ is in the range $|s| < \frac{1}{2} \log 2 \approx 0.35$, and its exponential can be evaluated as

$$\exp(s) = \sum_{n=0}^N \frac{s^n}{n!},$$

with $N=7$ for SP and $N=13$ for DP.

Polynomial expansions

- If $x = 2^n (1-r)$, where n is an integer and $|r| \leq \frac{1}{3}$, then

$$\log x = n \log 2 + \log(1-r),$$

$\log(1-r)$ can be approximated as

$$\log(1-r) = - \sum_{n>0}^N \frac{r^n}{n}.$$

but this needs $N=15$ for SP and $N=32$ for DP.

Polynomial expansions

- NVIDIA instead uses

$$\log(1-r) = -2 \tanh^{-1} \left(\frac{r}{2-r} \right)$$

with a Taylor series expansion for $\tanh^{-1} y$ which has only odd powers of y and needs just 9 terms for DP.

- $\sin x$, $\cos x$ and $\tan x$ are handled similarly.
- Taylor series expansions are sometimes replaced by best-fit L_∞ approximation – can slightly reduce the number of terms required
- Rational functions also reduce the number of terms needed, but at the cost of one division operation – not used much in practice?

Polynomial expansions

Final comment: to maximise the accuracy of the final result, it is important that

$$y = \sum_{n=0}^N a_n x^n$$

is evaluated by setting $y = a_N$ and then calculating

$$y := x y + a_n, \quad n = N-1, \dots, 0$$

The FMA operations are very efficient, and if x is small the resulting error is usually less than 1 ULP.

Final comments

- learn how to spot and avoid catastrophic cancellation
- use binary tree summation or compensated summation for large sums
- don't need to know about the construction of low-level maths libraries, but it's quite interesting
- fixed point algorithms are ideal because of quadratic convergence and opportunity for iterative refinement
- polynomial expansions can be costly for high accuracy

References

- Accuracy and stability of numerical algorithms (second edition), N.J. Higham, SIAM, 2002
- "The accuracy of floating point summation", N.J. Higham, SIAM Journal on Scientific Computing, 14(4):783-799, 1993
- binary tree summation code:
<http://people.maths.ox.ac.uk/~gilesm/hpc/>
- "Table-lookup algorithms for elementary functions, and their error analysis", P.T.P. Tang, 1991
www.stanford.edu/class/ee486/doc/tang1991.pdf
- "Rigorous and portable standard functions", S.M. Rump, BIT 41(3):540-562, 2001