Notes for Ma221, Lecture 2, Aug 28, 2023

Goals: Floating point arithmetic
Roundoff error analysis for polynomial evaluation
Beyond basic error analysis:
exceptions, high/low/variable precision arithmetic,
reproducibility, interval arithmetic,
exploiting mathematical structure to get accuracy without
high precision

Example: Polynomial Evaluation, and polynomial zero finding

EX: Review how bisection to find a root of f(x)=0 works:
start with an interval [x1,x2] where f changes sign:
f(x1)*f(x2) < 0
evaluate at midpoint: f((x1+x2)/2)
keep bisecting subinterval where f changes sign
Try it on (x-2)(x-3)(x-4) = x^3 - 9*x^2 + 26*x - 24
(Matlab demo)
rts = [2,3,4]
coeff = poly(rts)
help bisect (on web page)
bisect(coeff,2.6,3.1,1e-12)
bisect(coeff,2.2,3.2,1e-12)
... no surprises, get small intervals around 3

Now try it on (x-2)^13
rts = 2*ones(1,13)
coeff = poly(rts) ... no errors yet, take my word for it
bisect(coeff,1.7,2.4,1e-12)
bisect(coeff,1.7,2.2,1e-12)
bisect(coeff,1.9,2.2,1e-12) ... huh? a very different answer
each time?

Horner's rule to evaluate (x-2)^13 - what is the real graph?

```
x = 1.7:1e-4:2.3;
y = polyval(coeff,x);
yacc = (x-2).^13;
plot(x,y,'k.',x,yacc,'r','Linewidth',1.5)
axis([1.7,2.3,-1e-8,1e-8]), grid
```

... can we explain this?

To summarize: Try evaluating (x-2)^13 two ways:
as (x-2)^13 - smooth, monotonic curve, as expected
as x^13 - 26*x^12 + ... - 8192, with Horner's rule:
for x in the range [1.8,2.2], basically get random looking
numbers in the range [-1e-8,1e-8]
Actually, they aren't really "random", as the following zoomed-in
plot shows:

```
s=2^(-45); x=2-256*s:s:2+256*s;
```

```
   y=polyval(coeff,x); yacc=(x-2).^13;
   plot(x,y,'k.',x,yacc,'r'), axis([2-256*s,2+256*s,-2e-9,2e-9])
```

We will not explore the much less random behavior in this plot, but just try to bound the error. To do so, we need to understand the basics of floating point arithmetic.

(There is a large body of work studying roundoff in more detail, leading to more accurate algorithms, see additional notes at the end of this lecture and the class webpage for details.)

Floating Point - How real numbers are represented in a computer

Long ago, computers did floating point in many different ways, making it hard to understand bugs and write portable code. Fortunately Prof. Kahan led an IEEE standards committee that convinced all the computer manufacturers to agree on one way to do it, called the IEEE 754 Floating Point Standard, for which he won the Turing Award. This was in 1985. The standard was updated in 2008, and again in 2019. We'll say more on the significant changes below. See the class webpage for links to more details.

Scientific Notation: +- d.ddd x radix^e

Floating point usually uses radix=2 (or 10 for financial applications) so you need to store the sign bit (+-), exponent (e), and mantissa (d.ddd). Both p = #digits in the mantissa and the exponent range are limited, to fit into 16, 32, 64 or 128 bits. Historically, only 32 and 64 bit precisions have been widely supported in hardware. But lately 16 bits have become popular, for machine learning, and companies like Google, Nvidia, Intel and others are also implementing a 16-bit format that differs from the IEEE Standard, called bfloat16, with even lower precision (p=8 vs p=11). And now there is a new standards committee exploring 8-bit floating point numbers, which have become very popular for machine learning, with many companies inventing their own slightly different versions. So with the same motivation as for the 754 standard, consistency and portability across platforms, the new committee is trying to find the best design for "most" problems. How to use such low precision to reliably solve linear algebra (and other non-machine learning) problems is an area of current research.

For simplicity, we will initially ignore the limit on exponent range, i.e. assume no overflow or underflow.

Normalization: We use 3.1000e0 not 0.0031e3 - i.e. the leading digit is nonzero. Normalization gives uniqueness of representations, which is useful. And in binary, the leading digit must be 1, so it doesn't need to be stored, giving us a free bit of precision (called the "hidden bit").

```
Def: rnd(x) = nearest floating point number to x
     (Note: The default IEEE 754 rule for breaking ties is
      "nearest even", i.e. the number whose least significant digit
      is even (so zero in binary).)
Def: Relative Representation Error (RRE):
     RRE(x) = | x - rnd(x) | / | rnd(x) |
Def: Maximum Relative Representation Error  = max_x  RRE(x)
       (aka machine epsilon, macheps)
       = half distance from 1 to next larger number 1+radix^(1-p)
       = .5 * radix^(1-p) = | (1+.5*radix^(1-p)) - 1 | / 1
       = 2^(-p) in binary
    Note: eps in Matlab = 2^(-52) =  2*macheps

Roundoff error model, assuming no over/underflow:
      fl(a op b) = rnd(a op b) = true result rounded to nearest
                 = (a op b)(1 + delta),   |delta| <= macheps
      where op may be add, subtract, multiply or divide
We will use this throughout the course, it's all you need for most
algorithms. It's also true for complex arithmetic (but using a
bigger macheps, see Q 1.12 for  details).

Existing IEEE formats: single(S)/double(D)/quad(Q)/half(H) (radix = 2)
      S: 32 bits =1 (for sign) +8 (for exponent) +23 (for mantissa),
         So there are p=24 = 1 (hidden bit) + 23 bits to represent
         a number, and so macheps = 2^(-24) ~ 6e-8
         Also -126 <= e <= 127, so
           overflow threshold (OV) ~  2^128 ~ 1e38,
           underflow threshold (UN) = 2^(-126) ~ 1e-38
      D: 64=1+11+52 bits, so p=53, macheps = 2^(-53) ~ 1e-16
           -1022 <= e <= 1023, OV ~  2^1024 ~ 1e308, and
           UN = 2^(-1022) ~ 1e-308
      Q: 128=1+15+112 bits, p = 113, macheps = 2^(-113) ~ 1e-34
           -16382 <= e <= 16383, OV ~  2^16384 ~ 1e4932, and
           UN = 2^(-16382) ~ 1e-4932
      H: 16=1+5+10 bits,  p = 11, macheps = 2^(-11) ~ 5e-4
           -14 <= e <= 15, OV ~  2^15 ~ 1e4, and UN = 2^(-14) ~ 1e-4

The new bloat16 format has the following parameters:
   16 = 1+8+7, so p=8, macheps = 2^(-8) ~ 4e-3
   The exponent e has the same range as IEEE single (by design:
   converting between bfloat16 and S cannot overflow or underflow).

The committee working on 8-bit floating point for machine learning
is working on a document describing our plans so far, which will
Hopefully be released publicly soon.

Referring back to Lecture 1, where we referred to the approach of
using a few steps of Newton's method to be "guaranteed correct except
in rare cases," a common approach is to try to do most of the work in
```

lower (and so faster) precision, and then do just a little work
in higher (and so slower) precision, typically to compute accurate
residuals (like A*x-b), during the Newton steps; the goal is to get
the same accuracy as though the entire computation had been done in
higher precision.

Even higher precision than 128 bits is available via software
simulation (see ARPREC, GMP on the class web page)

We briefly mention E(xtended), which was an 80-bit format on Intel x86
architectures, and was in the old IEEE standard from 1985, but is now
deprecated. See also the IEEE 754 standard for details of decimal
arithmetic (future C standards will include decimal types, as already
in gcc).

That's enough information about floating point arithmetic to
understand the plot of (x-2)^13, but more about floating point later.

Analyze Horner's Rule for evaluating p(x):
  simplest expression:
       p = sum_{i=0 to d} a_i x^i
  algorithm:
       p   = a_d, for i=d-1:-1:0, p   = x*p   + a_i
  label intermediate results (no roundoff yet):
       p_d = a_d, for i=d-1:-1:0, p_i = x*p_{i+1} + a_i
  introduce roundoff terms:
       p_d = a_d,
       for i=d-1:-1:0, p_i = [x*p_{i+1}*(1+d_i) + a_i]*(1+d'_i)
                where |d_i| <= macheps and |d'_i| <= macheps
Thus
    p_0 = sum_{i=0:d-1}
          [(1+d'_i)*prod_{j=0:i-1 (1+d_j)*(1+d'_j)]*a_i*x^i
             + prod_{j=0:d-1} (1+d_j)*(1_d'_j)*a_d*x^d
        = sum_{i=0:d-1} [product of 2i+1 terms like 1+d] a_i*x^i
             + [product of 2d terms like (1+d)] a_d*x^d
        = sum_{i=0:d} a'_i * x^i

In words: Horner is backward stable: you get the exact value of a
polynomial at x but with slightly changed coefficients a'_i from
input p(x).

How to simplify to get error bound:
      prod_{i=1:n} (1 + delta_i)
         <= prod_{i=1:n} (1 + macheps)
          = 1 + n*macheps + O(macheps^2)
            ... usually ignore (macheps^2)
         <= 1 + n*macheps/(1-n*macheps) if n*macheps < 1
            ... (lemma left to students)
  Similarly
      prod_{i=1:n} (1 + delta_i)

```
          >= prod_{i=1:n} (1 - macheps)
           = 1 - n*macheps + O(macheps^2)
             ...  usually ignore (macheps^2)
          >= 1 - n*macheps/(1-n*macheps)  if n*macheps < 1
             ... (lemma left to students)

  So  |prod_{1 to n} (1 + delta_i) - 1| <= n*macheps
         ... ignore macheps^2
  and thus
      |computed p_d - p(x)| <= sum_{i=0:d-1} (2i+1)*macheps*|a_i*x^i|
                               + 2*d*macheps*|a_d*x^d|
                            <= sum_{i=0:d} 2*d*macheps*|a_i*x^i|

 relerr = |computed p_d - p(x)|/|p(x)|
       <=  sum_{i=0:d} |a_i x^i| / |p(x)|   *   2*d*macheps
        =  condition number  *  relative backward error

 How many decimal digits can we trust?
     dd correct digits <=> relative error <= 10^(-dd)
                       <=> -log_10 (relative error) >= dd

 How to modify Horner to compute (an absolute) error bound:
       p = a_d,  ebnd = |a_d|,
       for i=d-1:-1:1, p = x*p + a_i, ebnd = |x|*ebnd + |a_i|
       ebnd = 2*d*macheps*ebnd

Matlab demo:
  coeff = poly(2*ones(13,1));
  x = [1.6:1e-4:2.4];
  y = polyval(coeff,x);
  yacc = (x-2).^13;
  ebnd = 13*eps*polyval(abs(coeff),abs(x));
  %   note eps in Matlab = 2*macheps
  plot(x,y,'k.',x,yacc,'c',x,y-ebnd,'r',x,y+ebnd,'r')
  axis([1.65 2.35 -1e-8 1e-8]), grid
  %   need to make vertical axis wider to see bounds
  axis([1.65 2.35 -1e-6 1e-6]), grid
  %   conclusion: don't trust sign outside roughly [1.72, 2.33]

Consider Question 1.21: how could we use this error bound to
  stop iterating in root finding using bisection?
  ... now try wider range, look at actual and estimated # correct
      digits

  x = -1:.0001:5;
  y = polyval(coeff,x);
  yacc=(x-2).^13;
  ebnd=13*eps*polyval(abs(coeff),abs(x));
  plot(x,-log10(abs((y-yacc)./yacc)),'k.',x,-log10(ebnd./ ...
       abs(yacc)),'r')
```

```
  axis([-1 5 0 16]), grid
  title('Number of correct digits in y')
```

This picture is a foreshadowing of what will happen in linear algebra: The vertical axis is the number of correct digits, both actual (the black dots) and lower-bounded using our error bound (the red curve). The horizontal axis is the problem we are trying to solve, in this simple case the value of x at which we are evaluating a fixed polynomial p(x).

The number of correct digits gets smaller and smaller, until no leading correct digits are computed, the closer the problem gets to the hardest possible problem, in this case the root x=2 of the polynomial. This is the hardest problem because the only way to get a small relative error in the solution p(2)=0, is to compute 0 exactly, i.e. no roundoff is permitted. And changing x very slightly makes the answer p(x) change a lot, relatively speaking. In other words, the condition number, sum_i |a_i x^i| / |p(x)| in this simple case, approaches infinity as p(x) approaches zero.

In linear algebra the horizontal axis still represents the problem being solved, but since the problem is typically defined by an n-by-n matrix, we need n^2 axes to define the problem. There are now many "hardest possible problems", e.g. singular matrices if the problem is matrix inversion. The singular matrices form a set of dimension n^2-1 in the set of all matrices, the surface defined by det(A)=0. And the closer the matrix is to this set, i.e. to being singular, the harder matrix inversion will be, in the sense that the error bound will get worse and worse the closer the matrix is to this set. Later we will show how to measure the distance from a matrix to the nearest singular matrix "exactly" (i.e. except for roundoff) using the SVD, and show that the condition number, and so the error bound, is inversely proportional to this distance.

Here is another way the above figure foreshadows linear algebra. Recall that we could interpret the computed value of the polynomial p(x) = sum_i a_i*x^i, with roundoff errors, as the exactly right value of a slightly wrong polynomial, that is

$$p_{alg}(x) = \sum_{i=0:d} [(1+e_i)*a_i]*x^i,$$

where |e_i| <= 2*d*macheps. We called this property "backward stability", in contrast to "forward stability" which would means that the answer itself is close to correct. So the error bound bounds the difference between the exact solutions of two slightly different problems p(x) and p_alg(x).

For most linear algebra algorithms, we will also show they are backward stable. For example, if we want to solve A*x=b, we will instead get the exact solution of (A+Delta)*xhat = b, where the matrix Delta is "small" compared to A. Then we will get error bounds by

essentially taking the first term of a Taylor expansion of inv(A+Delta):

    xhat - x = inv(A+Delta)*b - inv(A)*b

To do this, we will need to introduce some tools like matrix and vector norms (so we can quantify what "small" means) in the next lecture.

To extend the analysis of Horner's rule to linear algebra algorithms, note the similarity between Horner's rule and computing dot-products:

```
  p   = a_d, for i=d-1:-1:1, p   = x*p     + a_i
  s   =  0 , for i=1:d,      s   = x_i*y_i + s
```

Thus the error analysis of dot products, matrix multiplication, and other algorithms is very similar (homework 1.10 and 1.11).

This is all you need to know to analyze the common behavior of many numerical algorithms. The next part of this lecture goes into more detail on other properties of floating point, such as exception handling, which are necessary for making algorithms reliable.

Next we briefly discuss some properties and uses of floating point that go beyond these most basic properties (even more details are at the end of these notes). Analyzing large, complicated codes by hand to understand all these issues is a challenge, so there is research in automating this analysis; there is a day-long tutorial on available tools that was held at Supercomputing'19.

(1) Exceptions: IEEE arithmetic has rules for cases like:

    Underflow: Tiny / Big = 0
         (or "subnormal", special numbers at bottom of exponent range)
    Overflow and Divide-by-Zero
       1/0 = Inf = "infinity", represented as a special case in the
           usual format, with natural computational rules like:
       Big + Big = Inf, Big*Big = Inf, 3 - Inf = -Inf, etc.
    Invalid Operation::
       0/0 = NaN = "Not a Number", also represented as special case
       Inf - Inf = NaN, sqrt(-1) = NaN, 3 + NaN = NaN, etc
    Flags are available to check if such exceptions occurred.

    Impact on software:
      Reliability:
         Suppose we want to compute s = sqrt(sum_i x(i)^2):
         What could go wrong with the following obvious algorithm?
           s = 0, for i=1:n s = s + x(i)^2, end for, s = sqrt(s)
         To see how standard libraries deal with this, see snrm2.f
         in the BLAS.
         For a worst-case example, see Ariane 5 crash on webpage.
         We are currently investigating how to automatically guarantee

that libraries like LAPACK cannot "misbehave" because of exceptions (go into infinite loops, give wrong answer without warning, etc.)

Error analysis: it is possible to extend error analysis to take underflow into account by using the formula

rnd(a op b) = (1+delta)*(a op b) + eta

where |delta| <= macheps as before, and |eta| is bounded by a tiny number, depending on how underflow is handled.

Speed:

Run ``reckless'' algorithm, that is fast but ignores possible exceptions

Check flags to see if exception occurred

In rare case of exception, rerun with slower, more reliable algorithm

(2) Higher precision arithmetic: possible to simulate using either fixed or floating point, various packages available: MPFR, ARPREC, XBLAS (see web page). There are also special fast tricks just for high precision summation; see Homework 1.18.

(3) Lower precision arithmetic: As mentioned before, many companies are building hardware accelerators for machine learning, which means they provide fast matrix multiplication. As mentioned in Lecture 1, and will be discussed later, it is possible to reformulate many dense linear algebra algorithms to use matrix multiplication as a building block, and so it is natural to want to use these accelerators for other linear algebra problems as well. As illustrated in our error analysis of Horner's rule, many of our error bounds will be proportional to d*macheps, where d is the problem size (eg matrix dimension), and often d^2*macheps or more. These bounds are only useful when d*macheps (or d^2*macheps) are much smaller than 1.
So when macheps is ~4e-3 with bfloat16, this means d must be much smaller than 1/macheps = 256, or much smaller than 1/sqrt(macheps) = 16, for these bounds to be useful. This is obviously very limiting, and raises several questions:
Are our (worst case) error bounds too pessimistic? Can we do most of the work in low precision, and little in high precision, to get an accurate answer? There are some recent positive answers to both these questions.

(4) Variable precision: There have been various proposals over the years to support variable precision arithmetic, sometimes with variable word lengths, and sometimes encoded in a fixed length word. Variable word lengths make accessing arrays difficult (where is A(100,100), if every entry of A can have a different bit-length?). There has been recent interest in this area, with variable precision formats called unums (variable length) and posits (fixed length), proposed by John Gustavson. Posits

allocate more bits for the mantissa when the exponent needs few bits (so the number is not far from 1 in magnitude) and fewer mantissa bits for long exponents. This so-called "tapered precision" complicates error analysis, and it is an open question of how error analyses or algorithms could change to benefit. See the class webpage for links to more details, including a youtube video of a debate between Gustavson and Kahan about the pros and cons of unums.

(5) Reproducibility: Almost all users expect that running the same program more than once gives the bitwise identical answer; this is important for debugging, correctness, even legal reasons sometimes. But this can no longer be expected, even on your multicore laptop, because parallel computers (so nearly all now), may execute sums in different orders, and roundoff makes summation nonassociative:
 (1-1)+1e-20 = 1e-20 neq 0 = (1+1e-20)-1
There is lots of work on coming up with algorithms that fix this, without slowing down too much, such as ReproBLAS (see web page). The 2019 version of the IEEE 754 standard also added a new "recommended instruction", to accelerate both the tricks for high precision arithmetic in (2), and to make summation associative (see web page).

(6) Guaranteed error bounds from Interval Arithmetic: Represent each floating point number by an interval [x_low,x_high], and use special rounding modes in IEEE standard to make sure that lower and upper bounds are maintained throughout the computation:
  [x_low,x_high] + [y_low,y_high] =
       [round_down(x_low+y_low),round_up(x_high+y_high)]
Drawback: naively replacing all variables by intervals like this often makes interval widths grow so fast that they are useless. There have been many years of research on special algorithms that avoid this, especially for linear algebra (see web page).

(7) Exploiting structure to get high accuracy: Some matrices have special mathematical structure that allows formulas to be used where roundoff is provably bounded so that you get high relative accuracy, i.e. most leading digits correct. For example, a Vandermonde matrix has entries V(i,j) = x_i^j, and arises naturally from polynomial interpolation problems. It turns out that this special structure permits many linear algebra problems, even eigenvalues, to be done accurately, no matter how hard (``ill-conditioned'') the problem is using conventional algorithms. See the class webpage for details.

(We end the recorded lecture here. The following notes give more Details about some of the above floating point issues.)

(1) Exceptions: what happens when computed result would have an

exponent that is too big to fit, too small to fit, or isn't mathematically defined, like 0/0? One gets "exceptions", with rules about the results and how to compute with them, plus flags to check to see if an exception occurred.

What if answer > OV? get infinity (overflow)
    fl(1e20*1e20) = fl(1/0) = inf   ... in single
    fl(3+inf) = inf, fl(1/inf) = 0, fl(1/-0) = -inf

What if answer mathematically undefined? get NaN = Not a Number
    fl(sqrt(-1)) = fl(0/0) = fl(inf-inf) = NaN
    3 + NaN = NaN ... so you see NaN on output if one occurred

What if answer < UN? get underflow
  What to do: if you return zero, then what happens with code:
      if (a .ne. b) then x = 1/(a-b)   ... can divide by zero,
  Instead, IEEE standard says you get "subnormal numbers"
     x = +- 0.dddd * 2^exp_min instead of +- 1.dddd * 2^exp
  Without subnormals, we would do error analysis with
      fl(a op b) = (a op b)(1 + delta) + eta,
         where |delta| <= macheps, |eta| <= UN
  With subnormals, can do error analysis with
      fl(a op b) = (a op b)(1 + delta) + eta,
         Where |delta| <= macheps, and
        |eta| <= UN*macheps for *,/ and eta=0 for +-
  Thm: With subnormals, for all floats a,b fl(a-b) = 0 iff a=b
  Purpose: simplify reasoning about floating point algorithms

Why bother with exceptions? Why not always just stop when one occurs?
(1) Reliability: too hard to have test before each floating point operation to avoid exception
     Ex for control system (see Ariane 5 crash on webpage),
     Ex Matlab: don't want to go into infinite loop because of an input NaN (caused several fixes to LAPACK, and also helped motivate an on-going CS research project to build tools to prove that NaNs, infinities, etc cannot cause infinite loops or analogous bugs)
(2) Speed: ditto (too slow to test before each operation)
      (1) run "reckless" code that is fast but assumes no exceptions
      (2) check exception flags
      (3) in case of exception, rerun with slower algorithm
    Ex: s = root-sum-squares = sqrt(sum_{i=1}^n x_i^2)
    What could go wrong? (see Q 1.19)
(3) Sometimes one can prove code correct with exceptions:
    Ex: Current fastest algorithms to find (some) eigenvalues of symmetric matrices depends on these, including 1/(-0) = -Inf

Impact on LAPACK: One of the fastest algorithms in LAPACK for

finding eigenvalues of symmetric matrices assumes that
1/+0 = +infinity, and 1/-0 = -infinity, as specified by the IEEE
standard. We tried to port this code to an ATI GPU,
and discovered that they did not handle exceptions correctly:
1/(-0) was +infinity instead of -infinity.
The code depended on getting -infinity to get the right answer,
and (until they fixed their hardware) we had to take the quantity
in the denominator of 1/d and instead compute 1/(d+0), which made
the -0 turn into a +0, whose reciprocal was correctly computed.
See EECS Tech Report EECS-2007-179 for details.

(2) Exploiting lack of roundoff in special cases to get high precision
Fact: if 1/2 < x/y < 2 then fl(x-y) = x-y ... no roundoff
Proof: Cancellation in x-y means exact result fits in p bits
Fact: Suppose M >= abs(x). Suppose we compute
S = fl(M+x), q = fl(S-M), r = fl(x-q)
In exact arithmetic, we would compute
r = x-q = x-(S-M) = x-((M+x)-M) = 0.
But in floating point, S+r = M+x exactly, with r being
the roundoff error, i.e. (S,r) is the double precision sum
of M+x
(Proof sketch in special case; only roundoff occurs in S=fl(M+x),
other two operations are exact).
This trick, called "Accurate 2-Sum" can be generalized in a
number of ways to get very general algorithms available in the
following software packages:
ARPREC (see class website): provides arbitrary precision
arithmetic
XBLAS (see class website): provides some double-double precision
Basic Linear Algebra Subroutines (BLAS), like
matrix-vector-multiplication. LAPACK uses these routines
to provide high accuracy solvers for Ax=b.
ReproBLAS (see class website): Provides bit-wise reproducible
parallel implementations of some BLAS. The challenge here
is that since roundoff makes floating point addition
nonassociative, computing a sum in a different order
will usually give a (slightly) different answer. On a
parallel machine, where the number of processors (and
other resources) may be scheduled dynamically,
computing a sum in a different order is likely. Since
getting even slightly different results is a challenge for
debugging and (in some cases) correctness,
we and others are working on efficient algorithms that
guarantee reproducibility.
New instruction in IEEE 754 2019: AugmentedAddition: This one
instruction takes any values of M and x, and returns
S = fl(M+x) and r = M+x-S exactly, a more general version
of Accurate 2-Sum (and potentially faster, depending on
how it is implemented). It also rounds M+x slightly
differently, rounding ties toward zero, instead of the

nearest even number, which can be used to make
floating point addition associative. There are analogous
AugmentedSubtraction and AugmentedMultiplication
instructions.

Finally, we point out the paper "Accurate and Efficient Floating
Point Summation," J. Demmel and Y. Hida, SIAM J. Sci. Comp, 2003,
for a efficient way to get full accuracy in a summation despite
roundoff (in brief: to sum n numbers to full accuracy, you need
to use log_2(n) extra bits of precision, and sum them in
decreasing order by magnitude), and the website
   www.ti3.tu-harburg.de/rump/
for a variety of papers on linear algebra with guaranteed
accuracy.

(3) Guaranteed error bounds (sometimes) via interval arithmetic:
So far we have used rnd(x) to mean the nearest floating point
number to x.
(Note: Ties are broken by choosing the floating point number whose
last bit is zero, also called "round to nearest even". This has
the attractive property that half the ties are rounded up and half
rounded down, so there is no bias in long sums.)
But the IEEE Floating Point Standard also requires the operations
  rnd_down(x) = largest floating point number <= x
  rnd_up(x)   = smallest floating point number >= x
Thus [rnd_down(x+y),rnd_up(x+y)] is an interval guaranteed to
contain x+y. And if [x_min,x_max] and [y_min,y_max] are two
intervals guaranteed to contain x and y, resp., then
[rnd_down(x_min+y_min),rnd_up(x_max+y_max)] is guaranteed
to contain x+y. So if each floating point number x in a program is
represented by an interval [x_min,x_max], and we use rnd_down and
rnd_up to get lower and upper bounds on the result of each
floating point operation (note: multiply and division are a
little trickier than addition), then we can get guaranteed error
bounds on the overall computation; this is called interval
arithmetic.
Alas, naively converting all variables and operations in a program
to intervals often results in intervals whose width grows so
rapidly, that they provide little useful information. So there has
been much research over time in designing new algorithms that try
to compute intervals that are narrow at reasonable
cost. Google "interval arithmetic" or see the website
www.ti3.tu-harburg.de/rump/ for more information.

(4) Accurate Linear Algebra despite roundoff:
A natural question is what algebraic formulas have the property
that there is some way to evaluate them that, despite round off,
the final answer is always correct in most of its leading digits:
 Ex: general polynomial: no, not without higher precision
 Ex: x^2+y^2: yes, no cancellation,

Ex: determinant of a Vandermonde matrix: V(i,j) = x_i^(j-1):
General algorithm via Gaussian elimination can lose all
digits, but formula det(V) = prod_{i < j} (x_j - x_i) works

A complete answer is an open question, but there are some necessary and sufficient conditions based on algebraic and geometric properties of the formula, see article by Demmel/Dumitriu/Holtz/Koev on "Accurate and Efficient Algorithms in Linear Algebra" in Acta Numerica v 17, 2008: A class of linear algebra problems are identified that, like det(Vandermonde), permit accurate solution despite roundoff. We will not discuss this further, just say that the mathematics depends on results going back to Hilbert's 17th problem, which asked whether positive rational (or polynomial) functions could always be written as a sum of squares of other rational (or polynomial) functions (answers: rational = yes, polynomial = no). For example, when 0 < x_1 < x_2 < ... in the Vandermonde matrix V , it turns out most any linear algebra operation on V can be done efficiently and to nearly full accuracy despite roundoff, including eig(V). (This work was cited in coauthor Prof. Olga Holtz's award of the 2008 European Math Society Prize).