What has the

Volume of a Tetrahedron

to do with

Computer Programming Languages?

Prof. W. Kahan
Mathematics Dept., and
Elect. Eng. & Computer Science Dept.
University of California at Berkeley

Prepared for
Prof. B.N. Parlett's Seminar on Computational Linear Algebra
Wed. 21 March 2001
and Stanford's Scientific Computing Seminar
Tues. 17 April 2001

<http://www.cs.berkeley.edu/~wkahan/VtetLang.pdf>

Contents:

What has the Volume of a Tetrahedron to do with Programming Languages?

Abstract: The computation of a tetrahedron's volume has been chosen as a didactic example elementary enough to be tolerated by the intended audience (who have forgotten most of the calculus and linear algebra they encountered in college) yet difficult enough to impart an appreciation of the challenges faced by applications programmers lacking in numerical experience though clever about other things. These challenges are exacerbated by programming languages like C++ and Java that perpetuate practices, accepted only as expedients in the 1960s, that now inflate the languages' capture cross-section for error. By treating the tetrahedron's volume as a case study we can formulate better guidelines for programming languages to handle floating-point arithmetic in ways compatible with the few rules of thumb that should be (but are still not being) taught to the vast majority of programmers, who learn no more about floating-point than they hear in a programming class or read in a programming language manual. Complaining about education rarely improves it; our efforts are better spent redesigning computer systems and languages to attenuate occasional harm caused by well-intentioned ignorance among the multitudes.

Many textbooks offer formulas for simple geometrical computations. But they leave …

… too many Questions Unanswered :

If such a formula is copied into a computer program written in a familiar language, say
Java, C++, C, Fortran, Pascal, or …,
how likely will the program yield *satisfactorily accurate* results despite roundoff?

If results are unsatisfactory, how will the programmer know? And then do what?
(Only a tiny percentage of programmers ever take a competent course in *Numerical Analysis*.)

If results are unsatisfactory, how will the program's user know? And then do what?
(The World-Wide Web can promulgate a program instantly to a billion unwitting users.)

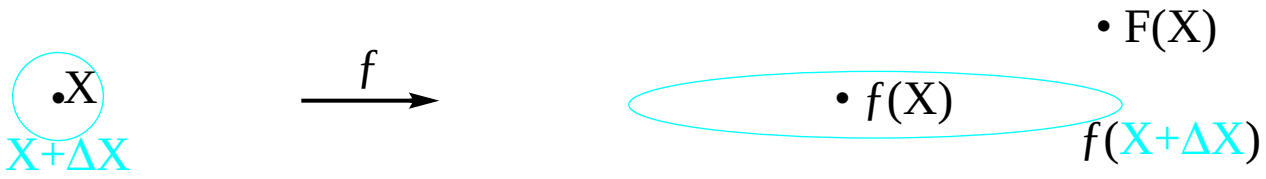
What does "*satisfactorily accurate*" mean?

- Correct in all but the last digit or two stored? Is this feasible? … practical?
- About as accurate as the data *deserve* ? What does "*deserve*" mean?

How accurate is "About as accurate as the data deserve" ?

Backward Error-Analysis :

Let F(X) be what is delivered by a program F intended to compute a function $f(x)$.
Let X+ΔX range over data regarded as practically indistinguishable from X .
Then F(X) is deemed "About as accurate as the data deserve" or "Backward Stable"
just when F(X) – f(X) is not much bigger than f(X+ΔX) – f(X) can get.



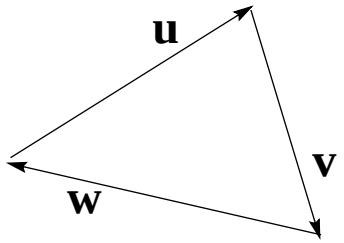
If f(X+ΔX) – f(X) can get too much bigger than ΔX we call f(…) *Ill-Conditioned* at X and blame the inaccuracy of F(X) upon the data X rather than the program F .

Relative error's magnification is gauged by a *Condition Number* like $\left| df(x)/dx \right| / \left| f(x)/x \right|$.

Example: Data X := Array [**B**, **c**] ; $\boldsymbol{f}$(X) := $\mathbf{B}^{-1}\mathbf{c}$ would solve " $\mathbf{B}\cdot\boldsymbol{f} = \mathbf{c}$ " .
Gaussian Elimination with Pivoting computes $\mathbf{F}(X) = (\mathbf{B}+\Delta\mathbf{B})^{-1}(\mathbf{c}+\Delta\mathbf{c})$ for some tiny roundoff-induced perturbations Δ… usually tinier than the data's uncertainty. If error $\mathbf{F}(X) - \boldsymbol{f}(X)$ is huge we blame an "ill-conditioned" matrix **B** rather than the program, nowadays deemed "Backward Stable" though on rare occasions it isn't. (See <www.eecs.berkeley.edu/~wkahan/Math128/FailMode.pdf>)

Warning: Elimination *without* pivoting misbehaves on many well-conditioned matrices **B** with modest condition-numbers $\|\mathbf{B}\|\cdot\|\mathbf{B}^{-1}\|$. Naive numerical programs are too often numerically unstable at otherwise innocuous data.

Area of a Triangle from its Edge-Lengths:



Edges have lengths u, v, w .

$$Area = \sqrt{(-\det\begin{bmatrix} 0 & u^2 & v^2 & 1 \\ u^2 & 0 & w^2 & 1 \\ v^2 & w^2 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix})/4} = \sqrt{(\,s\cdot(s-u)\cdot(s-v)\cdot(s-w))} \quad \text{where} \quad s := (u+v+w)/2 .$$

The formula containing s is attributed to Heron of Alexandria though it was known to Archimedes.

These are classical formulas found in many textbooks. For some data, roundoff can degrade either formula badly, especially when the triangle is too needle-shaped. However *Area* is very Well-Conditioned for all *acute* triangles, no matter how needle-shaped.

For all acute triangles, *Area*'s Condition Number is 2 .

See "Miscalculating Area and Angles of a Needle-Like Triangle" on my web page.

The next formula is fully **Accurate** (not merely Backward Stable):

$$Area = \sqrt{\big(\,(u+v+w)\cdot(v-w+u)\cdot(w-u+v)\cdot(u-v+w)\big)}/4$$

provided every *Facial Difference* like (u–v+w) is computed from an expression like (max{u, w} – v) + min{u, w} respectively.

(If massive cancellation occurs here it is benign because it involves only data prior to roundoff.)

All three formulas for *Area* are algebraically equivalent in the absence of roundoff.

How obvious is the last formula's accuracy *versus* the others' inaccuracy ?

Examples of Computed *Area*s of Triangles

	Edge-length u	Edge-length v	Edge-length w	Determinantal *Area*	Heron's *Area*	Accurate *Area*
*	10	11	12	51.52123350	51.52123349	51.52123349
*	100000	100000	1.00005	50002.50000	50010.0	50002.50003
*	100000	100000	1.5	75000.00000	75000.00000	75000.00000
*	0.1	0.1	0.0000015	7.1_{10}–8	7.500000_{10}–8	7.500000_{10}–8
	99999.99996	0.00003	99999.99994	1.3	**Error**	1.118033988
	0.00003	99999.99994	99999.99996	1.06	**Error**	1.118033988
	10000	5000.000001	15000	605.8	**0**	612.372435**8**
	5000.000001	10000	15000	**778.**	**0**	612.372435**8**
	99999.99999	99999.99999	200000	Error	**0**	Error
	5278.64055	94721.35941	99999.99996	**Error**	**Error**	0
	100002	100002	200004	**Error**	0	0
	31622.77662	0.000023	31622.77661	0.364	0.447	0.327490458
*	31622.77662	0.0155555	31622.77661	245.95405	246.18	245.9540000
	u	v	w	Determinantal	Heron's	Accurate

Digits known to be wrong are displayed **bold**. Triangles marked * have well-conditioned *Area*s. Computations were performed by an HP-15C that stores 10 sig. dec. digits; its DET carries 13 . Note how scaling or permuting the data can degrade the determinantal formula more than the others.

Sensitivity to *Rounding Directions* of two different formulas to calculate the *Area* of a Triangle from the Lengths of its Sides

(calculations performed upon 4-byte `float` data).

Rounding direction	**Heron's Formula** (unstable in `float`)	**Accurate Formula** (stable in `float`)	**Heron's Formula** (all subexpressions `double` like K-R **C**)
Ill-Conditioned	u=12345679 > v=1234567*8* >	w=1.01233995 >	u–v
to nearest	**0.0**	972730.06	972730.06
to +∞	**17459428.0**	972730.**25**	972730.06
to –∞	**0.0**	972729.**88**	972730.0**0**
to 0	**–0.0**	972729.**88**	972730.00
Well-Conditioned	u=12345679 =	v=12345679 >	w=1.01233995 > u–v
to nearest	**12345680.0**	6249012.0	6249012.0
to +∞	**12345680.0**	624901**3.0**	6249012.**5**
to –∞	**0.0**	624901**1.0**	6249012.0
to 0	**0.0**	624901**1.0**	6249012.0

Note that only incorrect results change drastically when the rounding direction changes, and that old-fashioned Kernighan-Ritchie **C** gets fine results from an "unstable" formula by evaluating all subexpressions in `double` even if they contain only `float` variables.
(This matters only to `double s = (u+v+w)/2` whose rounding error is critical.)

How Factorization tends to Attenuate Inaccuracy

Suppose $E = A\cdot C + B\cdot D$ and $F = A\cdot D + B\cdot C$ wherein A, B, C, D, E, F are all positive-valued expressions each computable within a small relative uncertainty $\pm\delta$; *i.e.*, the computed value of A is $A(1\pm\delta)$, of B is $B(1\pm\delta)$, …, of F is $F(1\pm\delta)$.

Which of two algebraically equivalent formulas for G , namely

$G := E–F$ and

$G := (A–B)\cdot(C–D)$,

is likely to be less inaccurate after severe cancellation? Answer: the factored formula.

" $G := (A–B)\cdot(C–D)$ " is uncertain by $\pm((A+B)\cdot|C–D| + |A–B|\cdot(C+D))\delta$ ignoring δ^2 .
" $G := E–F$ " is uncertain by $\pm(E+F)\delta = \pm(A+B)\cdot(C+D)\delta$,
even if computed without first computing A, B, C, D .

The two formulas' uncertainties differ substantially only when both $|A–B|/(A+B)$ and $|C–D|/(C+D)$ are tiny, and then the factored formula is substantially less uncertain than the other. In short, the factored form of G is never much more uncertain, and can be far less uncertain when severe cancellation leaves $|G| = |E–F| << E+F$.

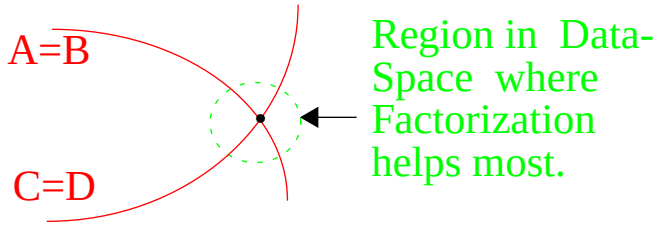
The factored formula's advantage is substantial only when *both* factors are small after cancellation; then each factor's rounding errors get attenuated by multiplication by the other factor.

How Factorization tends to Attenuate Inaccuracy … continued …

In the event that roundoff is followed by severe cancellation,

" G := (A–B)·(C–D) " can be far less uncertain than " G := E–F " .

Locus in Data-Space where G = 0 :



This argues in favor of of factored formulas whenever a function G to be computed …

- vanishes on (self-)intersecting surfaces in the space of all eligible data, provided
- the factored form is not too much more expensive to compute than the other form,
- no other unfactored expression for G far less uncertain than E–F is known.

The idea is illustrated (too) well by the triangle's

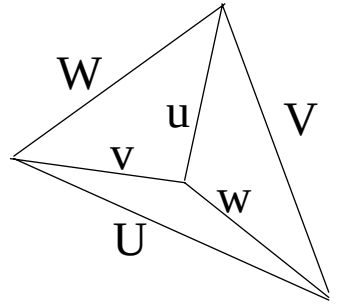
$$Area = \sqrt{((u+v+w)\cdot(v-w+u)\cdot(w-u+v)\cdot(u-v+w))/4} \ .$$

Area = 0 wherever one edge-length equals the other two's sum.

But, to make matters more interesting, factored formulas need not exist; and when they do exist they need not be determinable uniquely, not even for polynomial functions of given data, as we shall see later when we factor the formula for a tetrahedron's volume.

Volume of a Tetrahedron from its Edge-Lengths:

Edge-lengths are u, U, v, V, w, W in any order that pairs opposite edges' lengths thus: (u, U), (v, V), (w, W) .



$$\textit{Volume} = \sqrt{\left(\det\begin{bmatrix} 0 & u^2 & v^2 & w^2 & 1 \\ u^2 & 0 & W^2 & V^2 & 1 \\ v^2 & W^2 & 0 & U^2 & 1 \\ w^2 & V^2 & U^2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} \right)/288}$$

This computation is infrequent. The volume of a tetrahedron is computed far more often from the coordinates of its vertices than from its edge-lengths.

$$= \sqrt{\Big(4u^2v^2w^2 - u^2(v^2+w^2-U^2)^2 - v^2(w^2+u^2-V^2)^2 - w^2(u^2+v^2-W^2)^2 + (v^2+w^2-U^2)(w^2+u^2-V^2)(u^2+v^2-W^2) \Big)/12} .$$

These classical formulas come from textbooks like D.M.Y. Sommerville's *Analytical Geometry of Three Dimensions*, Cambridge Univ. Press, 1951. Neither formula need be Backward Stable when the tetrahedron is flat (like an arrow-head) or needle-shaped.

Volume is a well-conditioned (Cond. No. = 3) function of edge-lengths whenever every vertex projects perpendicularly to the inside of the opposite face, no matter how nearly the tetrahedron may resemble a javelin. This is why *Volume* can be difficult to compute as accurately as the data deserve.

The polynomial inside $\sqrt{(\ldots)}$ was published by Euler in 1752. It has no nontrivial factors that are *polynomials* in the six edge-lengths; but we shall factor it later anyway.

How Hard can Computing Determinants and Polynomials Accurately Be ?

Accuracy of algebraic computations is limited NOT by the precision of a computer's floating-point hardware, but by its Over/Underflow thresholds. Within those limits, arbitrarily high precision floating-point can be simulated by exclusively floating-point operations built into hardware, though the cost rises like the square of the high precision.

See thesis and paper by Doug Priest [1991], and consequent work by Prof. J. Shewchuk at `www.cs.berkeley.edu/~jrs/…` and `www.cs.cmu.edu/~quake/robust.html`.

Thus, the feasibility of computing a tetrahedron's *Volume* accurately is not in question; only the *costs* of doing so (or failing to do so) are at issue:

- Cost to the programmer — time spent on development and testing;
- Cost to the user — execution time and impact of inaccuracy.

Programs that run too slow don't get run.

Usually determinants are computed well enough with ordinary precision provided …

- Precondition by diagonal scaling to avoid severely disadvantageous pivot choices.
- Precondition by *exact* elementary operations inferred from an approximate inverse.
- Iteratively refine the triangular factors from which the determinant is computed.

Three Digressions: These three unfamiliar techniques will be explained to numerical practitioners curious about them, though they are palliatives at best, and often they are no help at all.

Digression: Preconditioning by Diagonal Scaling.

Its purpose is to prevent severely disadvantageous pivot choices. For instance,

$$\mathit{Volume} = \sqrt{\Big(}\ \det\left(\begin{bmatrix} 0 & u^2 & v^2 & w^2 & 1 \\ u^2 & 0 & W^2 & V^2 & 1 \\ v^2 & W^2 & 0 & U^2 & 1 \\ w^2 & V^2 & U^2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}\right)/288\ \Big) \ = \ \sqrt{\Big(}\ \det\left(\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & U^2 & V^2 & w^2 \\ 1 & U^2 & 0 & W^2 & v^2 \\ 1 & V^2 & W^2 & 0 & u^2 \\ 1 & w^2 & v^2 & u^2 & 0 \end{bmatrix}\right)/288\ \Big)\ ,$$

would normally be computed by Gaussian Elimination (triangular factorization) but the selection of pivots could be quite different for these two determinants. However, if all edge-lengths u, U, …, W are tiny enough, the latter determinant will so closely resemble

$$\sqrt{\Big(}\ \det\left(\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}\right)/288\ \Big) \ = 0$$

that the first pivot selection will disregard the data, thereby spoiling it if edge-lengths have widely different magnitudes, even though they do determine the *Volume* relatively well; it would be homogeneous of degree 3 in edge-lengths if roundoff did not interfere. (Unfortunate pivot selection can afflict the first determinant too, but the explanation takes longer.) If some permutations of the tetrahedron's vertices produce *Volume*s too different from the others, suspect poor pivot selection.

Many people regard matrices as well-scaled when every row and every column has at least one element roughly the same in magnitude as the biggest element. ***This criterion is mistaken***.

A better criterion is that every column-sum and row-sum of magnitudes be nearly the same; this criterion can be satisfied only approximately and only by iteration. It's a story for another day.

Digression: Preconditioning by *exact* elementary operations.

These are inferred from an approximate inverse and performed in somewhat extra-precise arithmetic.

Suppose det(B) is tiny because of massive cancellation; then B^{-1} must be huge and $B^{-1}\cdot B = I$ must largely cancel. Therefore rows of B^{-1} computed approximately provide linear combinations of rows of B that all almost vanish by cancellation. Choose one of the bigger rows b' of a computed B^{-1} so that $b'B \approx o'$ after cancellation; it would be a row of I if it weren't approximate.

Now let r' := round(μb') scaled and rounded to integers small enough that r'B is computed *exactly.* "Small enough" depends upon how many bits of precision are available in excess of those needed to represent the given data B exactly. The more extra bits, the bigger are the integers in r' , and the better they work. Now r'B is an exact linear combination of the rows of B that almost cancels out.

Replacing a row of B by such a linear combination computed *exactly* replaces B by a new matrix $\overline{B}$ whose determinant is a known multiple of det(B) . Let ß be an element of biggest magnitude in r' , and let j be its column index. The replacing row j in B by r'B yields a new matrix $\overline{B}$ with $\det(\overline{B}) = ß\cdot\det(B)$, and easier to compute accurately after preconditioning by diagonal scaling.

The process may be repeated so long as r'B can be computed exactly; each repetition moves more cancellation from inside the determinant calculation to ahead of it where the cancellation is harmless.

This process is one of the arcane motives for introducing the *Inexact* flag into IEEE Standard 754 for Floating-Point Arithmetic. The process dates back to the days of desk-top (electro-)mechanical calculators whose operators could see easily whether computations of r'B had been performed exactly. For another application see the *Advanced Functions Handbook* [1982] for the HP-15C calculator.

Digression: Iterative refinement of triangular factors.

In order to compute det(B) given square matrix B , Gaussian Elimination or the Crout-Doolittle method first obtains an approximate triangular factorization $L \cdot U \approx PB$. Here L is unit lower triangular, U is upper triangular, and P is a permutation matrix determined by pivotal exchanges; $\det(L) = 1$ and $\det(P) = \pm 1$, so $\det(B) = \det(P) \cdot \det(U)$ can be computed from the product of U's diagonal elements. But they are contaminated by accrued roundoff that we wish to attenuate. What follows does so with extra-precise arithmetic confined to the computation of the *Residual* .

Compute the residual $P\Delta B := PB - L \cdot U$ accumulating products extra-precisely. To estimate a strictly lower triangular ΔL (with zeros on its diagonal) and an upper triangular ΔU satisfying the equation $(L+\Delta L) \cdot (U+\Delta U) = PB$ more nearly exactly, solve the equation $(L+\Delta L) \cdot \Delta U + \Delta L \cdot U = P\Delta B$ using just working-precision arithmetic. This equation seems nonlinear, but it can be solved for the rows of ΔU and columns of ΔL in interlaced order: row#1 of ΔU , then column#1 of ΔL , then row#2 of ΔU , then column#2 of ΔL , … as illustrated below:

$$P\Delta B = (L+\Delta L) \cdot \Delta U + \Delta L \cdot U \ .$$

$$P\Delta B = \begin{bmatrix} 1 & & & & \\ a & 1 & & & \\ a & c & 1 & & \\ a & c & e & 1 & \\ a & c & e & g & 1 \end{bmatrix} \cdot \begin{bmatrix} & & & & \\ b & b & b & b & \\ & d & d & d & \\ & & f & f & \\ & & & h & \end{bmatrix} + \begin{bmatrix} & & & & \\ a & & & & \\ a & c & & & \\ a & c & e & & \\ a & c & e & g & \end{bmatrix} \cdot \begin{bmatrix} u & u & u & u & u \\ & u & u & u & u \\ & & u & u & u \\ & & & u & u \\ & & & & u \end{bmatrix} \cdot$$

Compute first a's, then b's, then c's, …, then h . .

(There are other workable orderings.) The process then writes L+ΔL over L and U+ΔU over U , rounding them off to working precision, and recomputes det(U) . This process may be repeated until it encounters a *Law of Diminishing Returns*, which sets in soon enough to vitiate repetitions after the first or second. The result's utimate accuracy is limited by the residual's accuracy after cancellation.

An Unobvious Factorization of a Tetrahedron's *Volume*

First compute nine *Facial Differences* belonging to three of the tetrahedron's four faces:

$X_U := w–U+v$, $X_v := U–v+w$, $X_w := v–w+U$, (Recall from *Area* how

$Y_V := u–V+w$, $Y_w := V–w+u$, $Y_u := w–u+V$, to compute all Facial

$Z_W := v–W+u$, $Z_u := W–u+v$, $Z_v := u–v+W$. Differences accurately.)

(These nine are not linearly independent.) Also needed are three facial sums

$$X_0 := U+v+w = X_U+X_v+X_w ,$$
$$Y_0 := V+w+u = Y_V+Y_w+Y_u ,$$
$$Z_0 := W+u+v = Z_W+Z_u+Z_v .$$

Next combine these 12 facial sums and differences in pairs into 6 products:

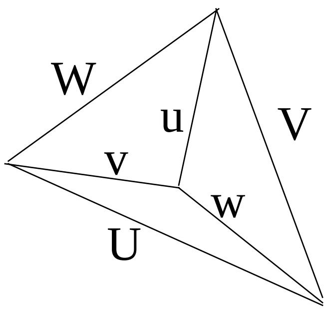
$X := X_U \cdot X_0 = (w–U+v)\cdot(U+v+w)$, $x := X_v \cdot X_w = (U–v+w)\cdot(v–w+U)$,

$Y := Y_V \cdot Y_0 = (u–V+w)\cdot(V+w+u))$, $y := Y_w \cdot Y_u = (V–w+u)\cdot(w–u+V)$,

$Z := Z_W \cdot Z_0 = (v–W+u)\cdot(W+u+v)$, $z := Z_u \cdot Z_v = (W–u+v)\cdot(u–v+W)$.

(Subscripts are no longer needed.) The 6 products are independent positive variables from which all 6 edge-lengths can always be recovered; for instance

$$u = \sqrt{\big(}\ (Y+y)\cdot(Z+z)/(X+x)\ \big)/2 , \quad \text{and}$$
$$U = \sqrt{\big(}\ (\ X\cdot(Y+y–Z–z)^2 + x\cdot(Y+y+Z+z)^2\ )/(\ (Y+y)\cdot(Z+z)\ )\ \big)/2 .$$



So far we have computed accurately the six products

$$X := (w–U+v)\cdot(U+v+w)\ , \qquad x := (U–v+w)\cdot(v–w+U)\ ,$$
$$Y := (u–V+w)\cdot(V+w+u)),\qquad y := (V–w+u)\cdot(w–u+V)\ ,$$
$$Z := (v–W+u)\cdot(W+u+v)\ ,\qquad z := (W–u+v)\cdot(u–v+W)\ .$$

Next compute four square roots of four products of products:

$$\xi := \sqrt{xYZ}\ ,\quad \eta := \sqrt{yZX}\ ,\quad \zeta := \sqrt{zXY}\quad \text{and}\quad \lambda := \sqrt{xyz}\ .$$

Then the tetrahedron's volume turns out to be

$$\mathit{Volume} = \sqrt{(\ (\xi+\eta+\zeta–\lambda)\cdot(\lambda+\xi+\eta–\zeta)\cdot(\eta+\zeta+\lambda–\xi)\cdot(\zeta+\lambda+\xi–\eta)\ )}/(192\ u\cdot v\cdot w)\ .$$

~ ~

All factors turn out to be positive for a non-degenerate tetrahedron. Because this unobvious property is invisible to automated algebra systems like *Derive*, *Maple* and *Mathematica*, they cannot prove our formula for *Volume* valid with just one `Simplify` command.

Our factored *Volume* formula is Backward Stable *provided* the edge-length pairs are so ordered that the three smallest of twelve facial differences lie among the nine used.

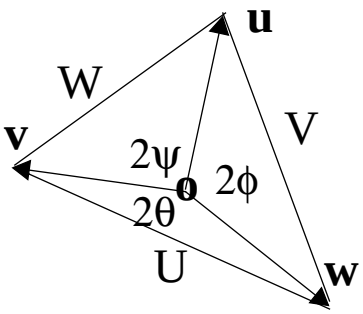
This claim's proof is utterly unobvious yet crucial since *Volume* can be a well-conditioned function (Cond. No. = 3) of edge-lengths for certain javelin-shaped tetrahedra despite numerical cancellation.

Digressions:

When is *Volume*'s Condition Number so small as 3 ?
Whence comes our factored form of Euler's formula for a tetrahedron's *Volume* ?

Whence comes our factored form of Euler's formula for a tetrahedron's *Volume* ?

Let column-vectors **u**, **v**, **w** be edges emanating from a vertex at **o** , with lengths u, v, w respectively; and let the angles between those edges be $2\theta, 2\phi, 2\psi$ as shown opposite edges with squared lengths $U^2 := (\mathbf{v}–\mathbf{w})'(\mathbf{v}–\mathbf{w})$, $V^2 := (\mathbf{w}–\mathbf{u})'(\mathbf{w}–\mathbf{u})$, $W^2 := (\mathbf{u}–\mathbf{v})'(\mathbf{u}–\mathbf{v})$ resp. Angles can be obtained from edge-lengths via the face-triangles' *Half-Angle-Tangent-Law*:



$$\tan^2\theta = \frac{(U–v+w)(v–w+U)}{(w–U+v)(U+v+w)}\,, \quad \tan^2\phi = \frac{(V–w+u)(w–u+V)}{(u–V+w)(V+w+u)}\,, \quad \tan^2\psi = \frac{(W–u+v)(u–v+W)}{(W–u+v)(u–v+W)}.$$

Half-angles θ, ϕ, ψ must satisfy ten inequalities,
first these four: $\theta+\phi+\psi \le \pi$, $\theta \le \phi+\psi$, $\phi \le \psi+\theta$ and $\psi \le \theta+\phi$,
and consequently six more: $0 \le \theta \le \pi/2$, $0 \le \phi \le \pi/2$ and $0 \le \psi \le \pi/2$.

Equality in place of any of these ten inequalities signifies a degenerate tetrahedron. For instance, if $\theta+\phi+\psi = \pi$ then vertex **o** falls into the triangle whose vertices are at the ends of **u**, **v** and **w** ; or if $\theta = \phi+\psi$ then **u** runs through the triangle with vertices at **o**, **v** and **w** . Violation of any inequality is geometrically infeasible.

The tetrahedron's *Volume* is a sixth the volume $\Omega := |\det([\mathbf{u}, \mathbf{v}, \mathbf{w}])|$ of a parallelepiped whose three edges emanating from a vertex at **o** are **u**, **v**, **w** .

Why "a sixth" ? Cut a cube into six congruent tetrahedra each another's reflection in a diagonal cutting plane.

Now Ω^2 can be expressed as a cubic polynomial in the squares of the six edge-lengths:

$$\Omega^2 = \det([\mathbf{u}, \mathbf{v}, \mathbf{w}]'[\mathbf{u}, \mathbf{v}, \mathbf{w}]) = \det\begin{bmatrix} u'u & u'v & u'w \\ v'u & v'v & v'w \\ w'u & w'v & w'w \end{bmatrix};$$

and substitute $\mathbf{u}'\mathbf{v} = (u^2+v^2-W^2)/2$ *etc*. to get

$$4\Omega^2 = \boxed{\begin{array}{c} 4u^2v^2w^2 - u^2(v^2+w^2-U^2)^2 - \\ - v^2(w^2+u^2-V^2)^2 - w^2(u^2+v^2-W^2)^2 + \\ + (v^2+w^2-U^2)(w^2+u^2-V^2)(u^2+v^2-W^2) \end{array}} = \det\begin{bmatrix} 0 & u^2 & v^2 & w^2 & 1 \\ u^2 & 0 & W^2 & V^2 & 1 \\ v^2 & W^2 & 0 & U^2 & 1 \\ w^2 & V^2 & U^2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}/2 .$$

This polynomial is irreducible. However, substituting $\mathbf{u}'\mathbf{v} = uv\cdot\cos(2\psi)$ *etc*. yields

$$\Omega^2 = \det\begin{bmatrix} u^2 & uv\cos(2\psi) & uw\cos(2\varphi) \\ vu\cos(2\psi) & v^2 & vw\cos(2\theta) \\ wu\cos(2\varphi) & wv\cos(2\theta) & w^2 \end{bmatrix} =$$

$$= (uvw)^2\cdot\big(\,1 - \cos^2(2\theta) - \cos^2(2\phi) - \cos^2(2\psi) + 2\cdot\cos(2\theta)\cdot\cos(2\phi)\cdot\cos(2\phi)\,\big) =$$

$$= 4\cdot(uvw)^2\cdot\sin(\theta+\phi+\psi)\cdot\sin(\phi-\psi+\theta)\cdot\sin(\psi-\theta+\phi)\cdot\sin(\theta-\phi+\psi) .$$

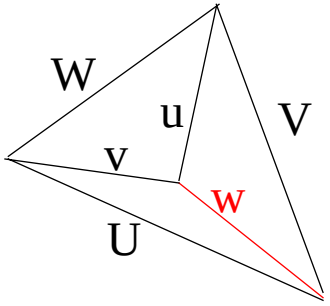
Except for the determinants, everything on this page down to here was known to Euler in the 1750s.

Substituting $\theta = \arctan(\sqrt{\ldots})$ *etc*. produces our factored backward stable formula for *Volume* after laborious simplification assisted by a computerized algebra system.

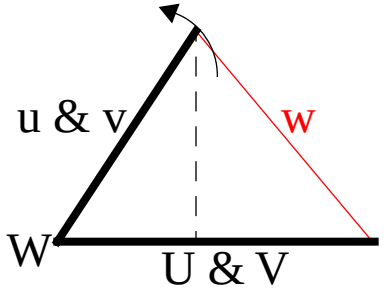
Digression: Why is a tetrahedron's *Volume* a well-conditioned function of edge-lengths provided its every vertex projects perpendicularly to a point inside its opposite face?

Such a tetrahedron's *Volume* must be an increasing function of each edge-length:

Top view



Edge-on view



Volume is a third the product of *Base Area* times *Altitude*, which increases with edge-length w so long as the top vertex projects perpendicularly to the base's interior, as shown here.

In such cases, *Volume* =: ¥(u, U, v, V, w, W) is a function with first partial derivatives

$¥_u := \partial ¥/\partial u > 0$, $¥_U := \partial ¥/\partial U > 0$, $¥_v := \partial ¥/\partial v > 0$, …, $¥_W := \partial ¥/\partial W > 0$.

Changing all edge-lengths by factors no farther from 1 than $(1\pm\delta)$ changes *Volume* by an amount no worse than $\pm(u\cdot¥_u + U\cdot¥_U + v\cdot¥_v + \ldots + W\cdot¥_W)\delta$ if terms of order δ^2 are ignored. Now, ¥ is *Homogeneous of degree* 3 ; this means for every $\lambda > 0$ that $¥(\lambda u, \lambda U, \lambda v, \lambda V, \lambda w, \lambda W) = \lambda^3\cdot¥(u, U, v, V, w, W)$. Therefore Euler's identity says that $u\cdot¥_u + U\cdot¥_U + v\cdot¥_v + \ldots + W\cdot¥_W = 3¥ = 3\cdot$*Volume* , whence it follows that *Volume* changes by a factor no worse than $(1 \pm 3\delta)$ when its data change by $(1\pm\delta)$.

Under these circumstances, *Volume*'s Condition Number is 3 .

The foregoing analysis of *Volume*'s condition number inspires another way to compute *Volume* as the square root of an unfactored polynomial:

Volume = $\sqrt{(\ H(u^2, U^2, v^2, V^2, w^2, W^2)\ )}/12$ where H is the homogeneous polynomial

$$H(x,X,y,Y,z,Z) := 4xyz - x(y+z-X)^2 - y(z+x-Y)^2 - z(x+y-Z)^2 + (y+z-X)(z+x-Y)(x+y-Z)$$

of degree 3. This H satisfies Euler's identity

$$3H = x\,\partial H/\partial x + X\,\partial H/\partial X + y\,\partial H/\partial y + Y\,\partial H/\partial Y + z\,\partial H/\partial z + Z\,\partial H/\partial Z$$

in which each partial derivative $\partial H/\partial\ldots$ is a homogeneous polynomial of degree 2. For instance,

$$\partial H/\partial x = K(x, X, y, Y, z, Z) := (x-z-Y)(x-y-Z) - (x-z+X-Z)(x-y+X-Y)\ .$$

Moreover, because H is unchanged by certain permutations of its arguments, namely those that correspond to the 24 permutations of the tetrahedron's vertices, we find that

$$\partial H/\partial x = K(x,X,y,Y,z,Z)\ , \quad \partial H/\partial y = K(y,Y,z,Z,x,X)\ , \quad \partial H/\partial z = K(z,Z,x,X,y,Y)\ ,$$
$$\partial H/\partial X = K(X,x,y,Y,Z,z)\ , \quad \partial H/\partial Y = K(Y,y,z,Z,X,x)\ , \quad \partial H/\partial Z = K(Z,z,x,X,Y,y)\ .$$

Consequently all six derivatives can be computed from six invocations of one program that computes $K(u^2, U^2, v^2, V^2, w^2, W^2)$ from, say, the carefully crafted expression

$$((u-w)(u+w) - V^2)((u-v)(u+v) - W^2) - ((u-w)(u+w)+(U-W)(U+W))((u-v)(u+v)+(U-V)(U+V))\ .$$

It yields another way to obtain *Volume* = $\sqrt{3H/432}$ with no division and one square root.

Why this method suffers less than the other unfactored expressions do from roundoff is not yet clear.

Numerical Tests: (These will be changed.)

A problem not yet fully solved is the generation of test data consisting of edge-lengths with known *Volume*s that reveal the weaknesses of the various formulas available to test. The six formulas tested and compared here were these:

- Two 5-by-5 determinants $\det\begin{bmatrix} 0 & u^2 & v^2 & w^2 & 1 \\ u^2 & 0 & W^2 & V^2 & 1 \\ v^2 & W^2 & 0 & U^2 & 1 \\ w^2 & V^2 & U^2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$ and $\det\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & U^2 & V^2 & w^2 \\ 1 & U^2 & 0 & W^2 & v^2 \\ 1 & V^2 & W^2 & 0 & u^2 \\ 1 & w^2 & v^2 & u^2 & 0 \end{bmatrix}$.
- Three arrangements of Euler's polynomial: Euler's identity, No divide, One divide.
- The new allegedly backward stable factored expression with four extra square roots.

All data was submitted permuted all 24 ways to find the worst harm due to roundoff.
All data was representable *Exactly* in 4-byte `floats` (24 sig. bits).
Test Data:

Case	u	U	v	V	w	W	*Volume*	Cond.#
1	66	66	66	66	66	66	33881.72852733461	3
2	777	222	444	666	444	555	8205786	6.347
3	1332	666	1443	555	1443	555	65646288	3
4	240	125	117	244	44	267	205920	3
5	4444	3535	3232	7676	3333	7070	3090903	$5.2_{10}7$
6	2121	5858	3232	7676	5656	4747	3090903	$5.0_{10}7$
7	16000001	5656.875	16000000	8000	15999999	5657.25	85339610684978.15	3.000

Test Results: Correct sig. bits for Worst Permutations of Edge-Lengths

Case	Det1	Det2	E's Ident.	No Div.	One Div.	New way	Cond.#
1	21	22	24	24	24	24	3
2	21	21	24	**20**	21	22	6.347
3	21	21	24	**20**	22	22	3
4	21	21	24	**20**	22	24	3
5	0	0	1	0	0	4	$1.5_2 25$
6	1	0	0	0	0	4	$1.5_2 25$
7	**20**	**4**	**6**	**3**	**3**	22	3.000

All results above were obtained from arithmetic rounded to `float` precision (24 s.b.).
Results obtained from arithmetic all rounded to `double` precision (53 s.b.) were

Case	Det1	Det2	E's Ident.	No Div.	One Div.	New way	Cond.#
1	53	53	53	53	53	53	3
2	50	49	53	51	52	52	6.347
3	50	49	53	53	52	53	3
4	51	50	53	53	53	53	3
5	29	29	31	27	30	33	$1.5_2 25$
6	29	29	31	29	32	32	$1.5_2 25$
7	49	32	33	**9**	31	51	3.000

Our backward stable factored formula for *Volume* appears to be new.

Since a backward stable formula has been sought for a long time, our formula must be too unobvious for even expert programmers who seek such a thing to be expected to find it.

I wish it did not exist, because then I could make a stronger case for supporting floating-point variables of arbitrarily high dynamically adjustable precision fully in all programming languages promoted, like Java, for general-purpose use.

(Why not program in Derive, Macsyma, Maple, Mathematica, … ?
Programs that run too slow don't get run.
Besides, these systems lack competent Interval Arithmetic to assist error analysis without which we can't know what precision to use.)

Floating-point computation has been changed drastically by …

- Profound declines in the costs of memory and arithmetic hardware,
- Near-zero time to perform hardware arithmetic, but still slow memory access,
- The Internet broadcasts software from many sources fast, far and wide.

Most by far of the floating-point operations in programs were put there by programmers exposed to at most an hour or two of instruction in the nature of floating-point arithmetic.

How can we defend ourselves against their well-intentioned ignorance?

Most by far of the floating-point operations in programs were put there by programmers exposed to at most an hour or two of instruction in the nature of floating-point arithmetic.

How can we defend ourselves against their well-intentioned ignorance?

Competent error-analysis is expensive of talent and time; the expense cannot be justified for most numerical software, so it it distributed after perfunctory analysis and testing.

How can we defend ourselves against this predictable omission of *Due Diligence* ?

There is a defense. It is based soundly upon Error-Analysis.

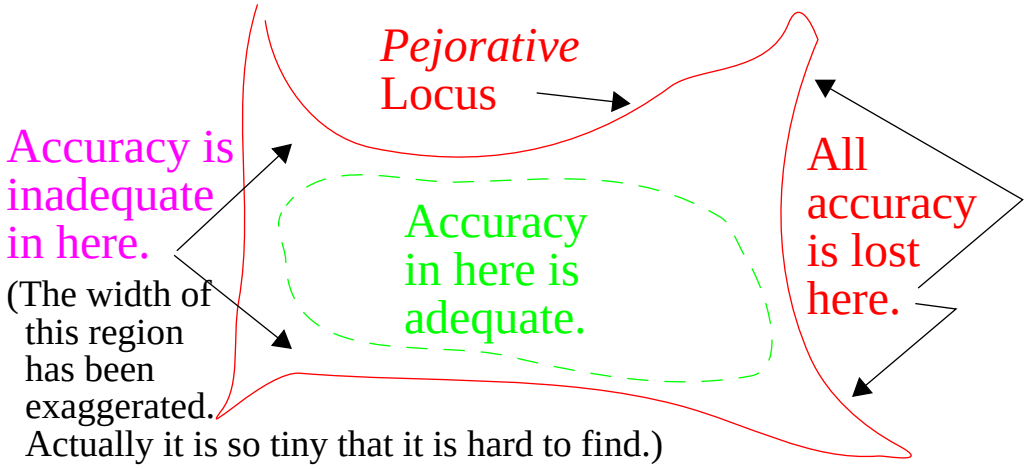
There are two kinds of numerical instability:

- Numerical instability for a substantial fraction of valid data.
  e.g.: unstable formulas for differential equations and matrix computations.
  This need not concern us because even perfunctory testing is likely to expose it and thus inhibit its promulgation. (Otherwise such programs are grist for lawyers' mills.)
- Numerical instability for so tiny a fraction of valid data that its discovery is unlikely.
  This is the threat against which we need a defense, *even if it's not impermeable*.

The defense: Use the widest precision that does not run too slow.

How does increased precision reduce the incidence of embarrassment by roundoff?

Look at the space of all data for a computed function:



Examples of Pejorative Loci:

For matrix inversion:
Singular matrices.

For tetrahedra's volume:
Volume = 0 .

All accuracy is lost at data lying on a locus of singularities either of the function being computed (near which locus the function is deemed ill-conditioned) or else of the algorithm executed by the program (which is therefore deemed numerically unstable near such data). At data far enough away from this *Pejorative* Locus of singularities, the program's accuracy is adequate. Between such data and the Pejorative Locus is a threshold shown as a dashed line above.

Increasing the precision of all intermediate computation (but not the data) reduces the *content* (number, area, volume, …) between the threshold and the Pejorative Locus, thus reducing the incidence of data for which the program's accuracy is inadequate.

Every extra three sig. dec. or ten sig. bits of intermediate precision reduces that incidence by a factor typically of 1/1000 or less, often much less. (1/1000000 for complex arithmetic.)

The moral of this story:

By default, all intermediate computations not explicitly narrowed by the programmer for cause should be carried out in the machine's widest precision that does not run too slow.

Old Kernighan-Ritchie `C` got it right on the PDP-11 :

It evaluated all floating-point *Anonymous Variables* (subexpressions, literal constants not representable exactly, functions' actual arguments) in `double` even if the declared variables' precisions were all `float`. (The PDP-11 ran faster that way.)

The widest precision that's not too slow on today's most nearly ubiquitous "Wintel" computers is not `double` (8 bytes wide, 53 sig. bits) but IEEE 754 `double extended` or `long double` (≥10 bytes wide, 64 sig. bits). This is the format in which all local scalar variables should be declared, in which all anonymous variables should be evaluated by default. C99 would permit this (not require it, alas), but …

- Microsoft's compilers for Windows NT, 2000, … disable that format.
- Java disallows it. Most ANSI C, C++ and Fortran compilers spurn it.

(Apple's *SANE* got it right for 680x0-based Macs, but lost it upon switching to Power-Macs.)

Most language designers and implementors ignorantly perpetuate unsound practices that error-analysts tolerated only reluctantly as expedients in the mid-1960s when computer memories were small and compilers had to work in one pass in a batch environment.

What few Rules of Thumb can we teach students in Programming classes?

Four Rules of Thumb for Best Use of Modern Floating-point Hardware

all condensed to one page, to be supported by programming languages for a mass market

0. All Rules of Thumb but this one are fallible. Good reasons to break rules arise occasionally.

1. Store large volumes of data and results no more precisely than you need and trust.
   Storing superfluous digits wastes memory holding them and time copying them.

2. Evaluate arithmetic expressions and, excepting too huge arrays, declare temporary (local) variables *all* with the widest finite precision that doesn't run too slow.
   Compiler optimizations mitigate a performance penalty incurred by following this rule naively.

3. Objects represented by numbers should ideally have a parsimonious representation, called "fiducial" and rounded as rule 1 says, from which all other representations and attributes are computed using wider precision as rule 2 says.
   For instance, a triangle can be represented fiducially by `float` vertices from which edges are computed in `double`, or by `float` edges from which vertices are computed in `double`. Computing either in `float` from the other may render them inconsistent if the triangle is too obtuse. In general, a satisfactory fiducial representation can be hard to determine. Moreover, an object in motion may require *two* representations, both a moving `double` and, obtained from it by a cast (rounding down), a `float` fiducial snapshot.

For more details see some of the postings on my web page.

Programming Language texts can explain these rules in a short Floating-Point chapter.

Times have changed.

Numerical computation for Science, Engineering, and (differently) Business used to be the *raison d'être* for computers. Now most of them serve merely to entertain; and Computer Science treats Numerical Analysis as if it were a sliver under its fingernail.

Here is my call to the Programming Language community:

Think not of obeisance to Numerical Analysis;
think about Self-Defence.

You too will unwittingly execute programs the World-Wide Web brings you from programmers whose acquaintance with Floating-Point, Numerical Analysis and Error-Analysis is no closer than yours, maybe far more remote. What are your chances?

Do you remember a time when the ability to change tires and spark-plugs, to clean out a carburettor float-bowl, and to adjust ignition timing and choke was required to drive a car? So far as Floating-Point is concerned, Java is still stuck back in that era.

More Stories for Another Day:

How likely is a programmer to find an unobvious numerically stable algorithm?

How likely is a programmer to use an unobviously numerically unstable algorithm?

How can a software user diagnose a rare affliction by numerical instability? (Directed roundings)
And what can he do about it without crippling performance? (Consult an Error-Analyst)

How should *Rules of Thumb for Floating-Point Computation* be taught to programmers while they are students still innocent of any interest in the subject? (With $$$ examples.)

How are the prospects for correct computation enhanced by the provision of …

- Dynamically adjustable precisions for floating-point variables ?
- Proper management of mixed-precision expressions and precision-inheritance ?
- Tools like Interval Arithmetic for the proper choice of higher precisions ?

How should languages like Java, C, Fortran, … handle floating-point variables declared wider than the widest that does not run too slow? (See citations in Farnum [1988] etc.)

Why is "About as accurate as the data deserve" too inaccurate for most situations?

"Use every man after his desert, and who should 'scape whipping?"
(*Hamlet iv.* 561)

Relevant Reading:

Carl B. Boyer [1989] *A History of Mathematics* 2d ed. rev. by Uta C. Mehrbach for Wiley, New York.

Heinrich Dörrie [1965] *100 Great Problems of Elementary Mathematics* translated by David Antin for Dover, New York. See pp. 285-9

K.L. Dove & J.S. Sumner [1992] "Tetrahedra with Integer Edges and Integer Volume" pp. 104-111 of *Mathematics Magazine* **65**. See also `http://mathworld.wolfram.com/HeronianTetrahedron.html` and citations there.

Farnum, Charles [1988] "Compiler Support for Floating-Point Computation" pp. 701-789 of *Software - Practice & Experience* **18**. See also Joseph Darcy's *Borneo* proposal for Java at `http://cs.berkeley.edu/~darcy` .

HP [1982] *HP-15C Advanced Functions Handbook*, part # 00015-90011, Hewlett-Packard Co. A scanned version stored as a `.pdf` file on a CD-ROM is available, CD1 for $15, from the Museum of HP Calculators: `http://www.hpmuseum.org/software/swcd.htm` .

W. Kahan [2000] "Miscalculating Area and Angles of a Needle-like Triangle" posted at `http://http.cs.berkeley.edu/~wkahan/Triangle.pdf` . Also posted there is "How Java's Floating-Point Hurts Everyone Everywhere" `…/JAVAhurt.pdf` , "Marketing vs. Math." `…/MktgMath.pdf` , "Huge Generalized Inverses of Rank-Deficient Matrices" `…/MathH110/GIlite.pdf` , and "Matlab's Loss is Nobody's Gain" `…/MxMulEps.pdf` .

MTHCP [1959] "Mathematical Tables from Handbook of Chemistry & Physics" 11th. Ed. edited by S.M.Shelby *inter alia*, Chemical Rubber Publ. Co., Cleveland. Germane pages are 340-356; but add 2 to page numbers cited herein for subsequent reprintings like the fifth of 1963.

M. Pichat [1972] "Correction d'une Somme en Arithmétique à Virgule Flottant" p. 400-406, *Numerische Mathematik* **19**.

D.M. Priest [1991] "Algorithms for Arbitrary Precision Floating Point Arithmetic" pp. 132-143, *Proc. 10th IEEE Symposium on Computer Arithmetic*, IEEE Computer Soc. Press Order # 2151, Los Alamitos, Calif.

D.M.Y. Sommerville [1951] *Analytical Geometry of Three Dimensions*, Cambridge Univ. Press. Pp. 34-6 & 77.

I. Todhunter [1929] *Spherical Trigonometry for the Use of Colleges and Schools* rev. by J.G. Leatham, MacMillan & Co., London. Pp. 19, 28, 238 problem 11.