On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic

§0: Abstract & Introduction

Current benchmarks give the impression that computation costs no more than the time consumed, and perhaps the memory occupied, only because we can measure these so easily. What about the costs of maintenance (when hardware or operating systems change), of development (algorithm design, tests, proofs of correctness), and of misleading results? Solving a quadratic equation provides a relatively easily understood case study of the way all costs get inflated when arithmetic precision rather higher than the precision of the data and the accuracy demanded of results is unavailable or, worse, unusable because of lapses in the design and/or implementation of widely used programming languages. Then costs are inflated by the trickery required to devise portable programs that compute roots at least about as accurately as the data deserve, and to prove that they are that accurate, and to test them. This trickery will be illustrated by a MATLAB program designed to get results of high quality from diverse versions of MATLAB on the two most popular kinds of hardware, PCs and Macs. Included is a test program and some of its results. Only a small part of the program needs higher precision arithmetic not too slow. There it would cost far less than the consequences of doing without.

Contents:

§1: Formulas that Almost Solve a Quadratic Equation

We learn in High School how, given real coefficients A, B and C of a quadratic equation $Az^2 - 2Bz + C = 0$, to solve it for its roots z: They are $(B \pm \sqrt{D})/A$ wherein the *Discriminant* $D := B^2 - A \cdot C$. Later some of us learn that, if $D > 0$, better accuracy is usually obtained from formulas $z_1 := (B \pm \sqrt{D})/A$ and $z_2 := C/(B \pm \sqrt{D})$ in which the sign of $\pm\sqrt{D}$ is chosen to match the sign of B in order to defend z_2, the root of smaller magnitude, from ruin by roundoff when the roots have extravagantly different magnitudes. There is more to the story than these formulas.

Very few of us learn that roundoff in these formulas can lose more than the roots' last sig. digit or two when the roots come too close to coincidence. Then the number of sig. digits lost roughly equals the number of leading digits in which the roots agree up to a maximum loss of about half the sig. digits carried by the arithmetic. The shock we feel at learning of this loss may abate when we learn from *Backward Error-Analysis* that the roots suffer little worse from roundoff during their computation than if first each of the given coefficients A, B and C had been perturbed by a rounding error or two when it was generated, and then the roots had been computed exactly from the perturbed coefficients.

Thus we might conclude that the computed roots are scarcely more uncertain due to roundoff during their computation than to roundoff before their computation, and therefore the loss of more than a few of their last sig. digits, when this loss occurs, is deserved by "*Ill Conditioned* roots", as they are called when they are hypersensitive to perturbations of the coefficients.

That conclusion is not quite right. It seems somewhat like blaming a crime upon its victim.

Good reasons exist for software to compute the roots each correct in every bit except perhaps its last one or two, as if every coefficient's every bit had been supplied correctly. It may have been, for all the programmer and users of that software know. Software that accurate can be costly to program in a language that lacks support for arithmetic rather more precise than the precision of the coefficients and the desired accuracy of the roots. Why is their accuracy worth that cost?

§2: Why We Care

How do computations resemble "Recreational" drugs? Hardly anybody ever stops with one.

Rarely does a computation end at the roots of a quadratic. They are almost always intermediates towards some other goal. For example, they may characterize responses to impulses of a simple linear system, perhaps a mechanical system consisting of a mass, a spring, and some drag, or perhaps an electrical system consisting of an inductance, a capacitance and some resistance. A question like

"After disturbance by an impulse,
does the system pass through its equilibrium state before returning to it?"

can be answered by determining whether the roots are real or complex, and then they reveal how long the system takes to return so nearly to equilibrium as doesn't matter. In near-borderline cases, near what is called "Critical Damping", the roots almost coincide, and these are the cases most sensitive to roundoff in the coefficients and in the program that computes the roots.

Roots computed from the formulas in §1 belong to coefficients almost indistinguishable from the stored coefficients which, in turn, are almost indistinguishable from intended values and thus belong to a linear system almost indistinguishable from the intended system. The two systems behave in ways so nearly indistinguishable that everybody might as well compute the behavior of the system whose coefficients are stored in the computer. All its properties will be computed from these coefficients. If properties are to be physically consistent, all of them must be computed accurately enough from the same coefficients rather than from slightly different coefficients that vary from one property to another. Otherwise inaccurate roots belonging to coefficients slightly perturbed in one unknown way may turn out inconsistent with some other property belonging to coefficients slightly perturbed in some other unknown way. Let's look at such a property.

A crucial property of a system is the way its response to disturbances changes when the system's parameters change a little. This property is especially important during attempts to optimize the system's design. Then the parameters are changed intentionally to adjust the system's response towards some desired behavior. The optimization process presumes that the system's computed response changes the way it should when the process alters parameters a little but not too little to elicit a palpable change in response. Computed responses that change too differently than do true responses are too likely to confound the optimization process.

Here is a concrete example. Lest the reader drown in digits, the quadratics' coefficients and roots are restricted to 4 sig. dec. and inexact computations are all rounded to 4 sig. dec. The roots of two quadratic equations with slightly altered coefficients are compared to see whether their roots change in the right directions when computed directly from the formulas presented in §1.

Table 1: True *vs.* Crudely Computed Roots of Nearby Quadratic Equations

A	B	C	true D	true roots		computed D	computed roots	
10.27	29.61	85.37	0.0022	2.88772…	2.87859…	0.1000	2.914	2.852
10.28	29.62	85.34	0.0492	2.90290…	2.86075…	0	2.881	2.881
Changes in roots:				+0.01518…	–0.01784…		–0.033	+0.029

Changing the coefficients has changed all roots substantially; but the computed roots, besides losing half the digits carried, change too much in directions opposite to the way they should.

Backward Error Analysis explains the results tabulated above but does not endorse them nor does it answer a question troublesome to both users and providers of numerical software:

> How much accuracy should be expected of the results from high-quality floating-point software when well-known or obvious formulas deliver inadequate accuracy?

"High-quality floating-point software" is the kind upon which we prefer to rely when its results may matter to us more than computerized entertainment does. Such software will influence the trajectory of an anti-ballistic-missile missile, or the control surfaces of a new airliner *flown by wire* as most are now, or predictions of an economy's response to proposed changes in policy.

All such software uses mathematical models to predict future behavior, and all such predictions are extrapolations, and most extrapolations amplify errors in the course of propagating them. The farther the prediction, the greater the amplification. And many a mathematical model relies upon delicate internal relationships far more sensitive to tiny errors than is the modeled system to small perturbations of its parameters. For instance, we characterize the behavior of control systems in terms of the eigensystems of associated matrices even though the eigensystems, both computed and true, are often orders of magnitude more sensitive to tiny perturbations in the matrices than are control systems sensitive to variations in their components. The roots of a quadratic equation can be vastly more sensitive to perturbations of its coefficients than the behavior of a simple linear system (over time intervals not extremely long) can be sensitive to variations in its parameters.

Consequently the foregoing question about the expected accuracy of high-quality floating-point software has no simple quantitative answer. Instead the question gets answered qualitatively:

- "I desire results so accurate that I don't care how accurate they are."
- "I hope to produce results at least about as accurate as the input data deserve and can be extracted from this data in a tolerable amount of time with the computational resources available to my program."

How much will the fulfillment of these hopes and desires cost? This is the question to be explored in what follows. A simple answer almost always adequate in the absence of a competent (and often too expensive) error-analysis is provided by this old *Rule of Thumb* :

> Perform all computation in arithmetic with somewhat more than twice as many sig. digits as are deemed significant in the data and are desired in the final results.

This old rule of thumb works beautifully for Table 1's quadratic equations: Roots computed by arithmetic carrying at least 8 sig. dec. are correct in all digits carried but the last one or two. The rule of thumb malfunctions so seldom that it would almost answer our question about cost but for lapses in today's most popular programming languages. Unlike the original Kernighan-Ritchie ***C*** , which evaluated all floating-point subexpressions, literal constants and functions in `double` precision 8 bytes wide even if an expression's variables were all `floats` 4 bytes wide, ANSI ***C***, *Fortran* and *Java* generally do not promote purely `float` subexpressions to `double`. Instead programmers must remember to insert *casts* like " `D = (double)B*B – (double)A*C ;` " in ***C*** and Java, and use " `D = dprod(B,B) – dprod(A,C)` " in Fortran.

Thus, when `float` is precise enough for the quadratic equation's coefficients and roots, or for data and results more generally, today's programming languages tend to exacerbate end-users' costs due to the occasional unreliability of floating-point software produced by programmers who either forgot the `(double)` casts or did not know to insert them.

In every army large enough there is always somebody who fails to get the message, or gets it wrong, or forgets it.

Of course, a typical end-user is unlikely to know *when* floating-point results are worse than they should be, much less *why* they are worse. Consequently those exacerbated costs are unknown. And if they were known they might well be blamed entirely upon deficiencies in applications programmers' education rather than at least partially upon programming languages' entrapment of numerically naive programmers.

Lacking estimates of the costs to users of misleading floating-point results, we explore instead the costs to conscientious software developers of attempts to avoid producing misleading results. We suppose data and/or results require at least 8 or 9 sig. dec. and are stored in the computer as `doubles`. Then the old rule of thumb would demand more than `double` precision which, though adequate for most computations, is inadequate for nearly coincident roots of quadratic equations.

Here is a simple example with data and all arithmetic in IEEE Standard 754 `double` precision. (Your computer may require the suffix " D0 " be appended to the data's digits.)

Table 2: True *vs.* Crudely Computed Roots of Nearby Quadratic Equations

A	B	C	true D	true roots	computed D	computed roots
94906265.625	94906267.000	94906268.375	1.89…	1.000000028975958… 1.0	0.0	1.000000014487979 1.000000014487979
94906266.375	94906267.375	94906268.375	1.0	1.000000021073424… 1.0	2.0	1.000000025437873 0.999999995635551
	Changes in roots:			-0.000000007902534… 0		+0.000000010949894 –0.000000018852428

Besides half the digits carried, the roots' direction of movement has been lost despite arithmetic carrying rather more sig. digits (53 sig. bits amounts to almost 16 sig. dec.) than the 11 sig. dec. the data might naively be thought to deserve. No such loss afflicts this data when the discriminant $D = B^2 - A \cdot C$ is accumulated in whatever floating-point wider than `double`, whether built into hardware or simulated in software, is provided by any current ***C*** or Fortran compiler.

When coefficients are given in `double` the simplest way to defend accuracy is to accumulate the discriminant D in floating-point arithmetic 16 bytes wide. This may be called `long double` in Hewlett-Packard's or Sun Microsystem's ***C***, or `extended` in IBM's Fortran, or `real*16` or `quadruple precision` elsewhere. Binary floating-point rightly called "quadruple precision" should have 113 sig. bits and an exponent field 15 bits wide. An alternative called "doubled-double" is less well specified, supplying at best 106 sig. bits of precision and an exponent field 11 bits wide; this is the `long double` on Power-PC processors used by IBM and Apple.

Here is a test to tell which kind of 16-byte floating-point you have if you have it:

```
long double  s,  t = 3.0 ;
s = 1.0 - (4.0/t - 1.0)*t .
```

If `s` is zero this test has failed, perhaps because your compiler "optimized" too aggressively.

If `s` is $2^{-112} \approx 1.926\text{e-}34$ your `long double` is probably an honest quadruple precision of the kind found on IBM's z/Architecture ESA/390 and in H-P's and Sun's ***C***, slow but reliable. If |s| is roughly $2^{-105} \approx 2.465\text{e-}32$ your `long double` is probably a grubbier but faster doubled-double. Computing D with either of these arithmetics and then rounding it back to `double` will obviously provide adequately accurate `double` roots for quadratics with `double` coefficients, and will do so at the least possible cost. And if you can do that, you need not read what follows.

If `s` is $2^{-52} \approx 2.22\text{e–}16$ your `long double` is just `double` 8 bytes wide, as happens on recent Microsoft compilers for the PC. If `s` is $-2^{-63} \approx -1.0842\text{e–}19$ your `long double` is IEEE Standard 754's double-extended on an Intel-based PC or clone with a Borland or gnu ***C*** compiler or an early version (6 or 7) from Microsoft. Or you may be running a 680x0-based Apple Macintosh two decades old. (Though 10 bytes wide, each of these double-extended numbers is best stored in 12 or 16 bytes of memory to avoid degradation of performance due to misalignments with 4-byte, 8-byte or 16-byte memory busses.) Neither 8-byte nor 10-byte arithmetic can provide adequately accurate `double` roots for *every* quadratic equation with `double` coefficients unless it is backed up by a tricky scheme to compute the discriminant D .

What follows will explore this tricky scheme and its costs. A mole hill will grow into a mountain before our eyes. It is a sight worth watching because similar inflation afflicts other problems –

Matrix Computations, Equation Solving, Optimization, Financial Forecasting,
Geometrical Analysis, Deflections of Loaded Structures, Trajectories and Orbits, … –

when arithmetic extravagantly more precise than the data and the desired accuracy (deserved or not) of results is inaccessible or too slow to use. Of course, extravagant precision is most likely necessary only for data sets that occur rarely if ever; and were such troublesome data recognized easily in advance we would tolerate extra costs paid on rare occasions. Instead numerical troubles are usually recognized too late if ever, and then misdiagnosed at least at first if not forever. We tend to blame time wasted and other inflated costs upon that data by calling it "Ill-Conditioned" whenever we have persuaded ourselves that our computational methods, having succeeded on all the other data we tested, must be "Numerically Stable".

Name-calling makes the caller feel better without enlightening him.

§3: Arithmetic Available to Compute the Discriminant D

How costly is the computation of 8-byte $D := B^2 - A\cdot C$ from 8-byte coefficients A, B and C when D must be accurate in all but its last few bits, and 16-byte floating-point arithmetic is unavailable? This step is crucial to the solution of the quadratic equation $Az^2 - 2Bz + C = 0$ for 8-byte roots z_1 and z_2 accurate in all but their last bit or two.

Also important are the treatments of exceptional coefficient like ∞ and *NaN* and $A = 0$, and the avoidance of premature over/underflows among intermediate results when the final results should be roots z within the range of 8-byte numbers. We shall skip over these issues because they can be handled easily in ways that would merely distract us from our main concern, which is accuracy despite roundoff.

The scheme by which we shall compute D accurately enough has three steps. The first tests whether the obvious formula for D has lost too many bits to roundoff. If so, the second step recomputes each of B^2 and $A\cdot C$ *exactly* as an unevaluated sum of two 8-byte floating-point numbers; say $P + dP = B^2$ and $Q + dQ = A\cdot C$ wherein $P = (B^2$ rounded to 8 bytes) and $dP = B^2 - P$ exactly, and similarly $Q = (A\cdot C$ rounded to 8 bytes) and $dQ = A\cdot C - Q$ exactly. The third step simply computes $D := (P–Q) + (dP–dQ)$ in ordinary 8-byte arithmetic; the proof of its adequate accuracy must be unobvious because algorithms (including one of mine (1981, Fig. 10 on p.49)) published previously for this step are much more complicated and slower.

Our three-step scheme will accommodate three styles of floating-point arithmetic. The first style rounds every algebraic operation upon 8-byte operands to 8 bytes. This is what Java requires and MATLAB normally does. The second style evaluates some expressions in 10-byte registers before rounding assignments to 8-byte variables in memory. Though this is the way Intel's and Motorola's floating-point arithmetics were designed to be used 25 years ago, the community of programming language designers and implementers has misunderstood its virtues and skimped its support. Still, vestiges of this second style persist in a recent version of MATLAB on PCs, so it will be exploited to enhance substantially the average speed of our quadratic equation solver.

The third style of floating-point arithmetic enjoys a *Fused Multiply-Add* operation that evaluates $X\cdot Y\pm Z$ to obtain an 8-byte result from three 8-byte operands with only one rounding error. This FMA came into existence two decades ago with IBM's Power architecture and is now provided also by Apple's Power Macs and by Cray/Tera's MTA and by Intel's Itanium. An FMA is partially accessible from a version of MATLAB on Power Macs, so it will be exploited to bring the speed of our quadratic equation solver nearly up to the speed of a naively inaccurate solver using the formulas in §1 unaltered.

Which of the three styles of arithmetic may your version of MATLAB use sometimes on your hardware? You can answer this question with a test like the following:

```
x = [1+4.656612873e-10, 1]*[1-4.656612873e-10; -1] ;
y = [1, 1+4.656612873e-10]*[-1; 1-4.656612873e-10] ;
```

If `x` and `y` are both zero, MATLAB follows the first style, rounding every arithmetic operation to the same 8-byte precision as is normally used for variables stored in memory. If `x` and `y` are both $-2^{-62} \approx$ –2.1684e–19, MATLAB follows the second style when computing scalar products of vectors and of rows and columns of matrices; these are accumulated to 64 sig. bits in 10-byte registers before being rounded back to 53 sig. bits when stored into 8-byte variables.

MATLAB 6.5 on PCs follows the first style by default; to enable the second style execute the command

```
system_dependent( 'setprecision',  64 )
```

A similar command

```
system_dependent( 'setprecision',  53 )
```

restores the default first style.

If `x` is zero and `y` is $-2^{-62} \approx -2.1684\text{e}{-19}$, MATLAB follows the third style to evaluate a scalar product [q, r]*[s; t] as (p + r·t rounded to 53 sig. bits) where p = (q·s rounded to 53 sig. bits) . If `x` and `y` take any other values the test for arithmetic style has failed.

The style of arithmetic affects only the trickery used to compute the discriminant $D = B^2 - A \cdot C$ accurately enough. The trickery will be hidden in a MATLAB program `dscrmt(A, B, C)` that computes D . Then the program qdrtc(A, B, C) that computes the roots will look like this:

```
function  [Z1, Z2] = qdrtc(A, B, C)
%  [z1, z2] = qdrtc(a, b, c)  produces the roots  z1  and  z2
%  of the quadratic equation  a*z^2 - 2*b*z + c = 0 .
D = dscrmt(A, B, C) ;  %...  See  dscrmt  below
if  (D <= 0),
    R = B/A ;  S = sqrt(D)/A ;  Z1 = R+S ;  Z2 = R-S ;
    return,  end  %... Complex or coincident real roots
R = sqrt(D)*(sign(B) + (B == 0)) + B ;
Z1 = R/A ;  Z2 = C/R ;  return  %... Real roots

function  D = dscrmt(A, B, C)
%  D = B^2 - A*C  computed accurately enough by a tricky scheme.
```

The tricky scheme's details will be spread out below.

§4: When is a Tricky Scheme Necessary?

Usually the obvious way to compute $D := B^2 - A\cdot C$ is accurate enough. Only when rounded values of B^2 and $A\cdot C$ cancel too many of each other's digits must a tricky scheme be invoked. How many are "too many"? It depends upon the style of arithmetic. This dependence will be encapsulated in a parameter π determined by the style as follows:

$\pi := 3$ for styles #1 and #3	(53 sig. bit accumulation of scalar products)
$\pi := 1024$ for style #2	(64 sig. bit accumulation of scalar products)

Thus, $\pi = 1024$ just when both test values `x` and `y` in §3 above are 2^{-62}. Then we compute D as a scalar product to exploit MATLAB's extra-precise accumulation, if it is enabled, thus:

```
D = [B, A]*[B; -C] ;  E = [B, A]*[B; C] ;
if  pie*abs(D) >= E  then return  %...  D  is accurate enough
%...  Otherwise use trickery to recompute  D  well enough  ...
```

If the quickly computed value of $D \approx B^2 - A\cdot C$ satisfies $\pi\cdot|D| \geq E = B^2 + A\cdot C$ then roundoff's contribution to D's relative error during the scalar product computation of D can be shown to exceed the relative errors in multiplications by a factor no bigger than π. Then roundoff blights D's last two bits when $\pi = 3$, its last bit when $\pi = 1024$. This error is small enough not to require that D be recomputed. The quickly computed D would be good enough most the time when $\pi = 3$, almost all the time when $\pi = 1024$, if coefficients A, B and C were independent random variates. But they aren't. Still, when extra-precise accumulation of scalar products to 11 more sig. bits than the operands' precision is available, it renders tricky and slow recomputation of D so nearly never necessary that the average computation time for accurate roots exceeds only imperceptably the time taken by the formula in §1 that can lose half the digits carried. This is the way 11 extra bits of precision usually pays off.

If the quickly computed value of $D \approx B^2 - A\cdot C$ satisfies $\pi\cdot|D| < E = B^2 + A\cdot C$ then roundoff's contribution to D's relative error during the scalar product computation of D may be intolerably big after cancellation. In fact $1/2 \leq (\pi-1)/(\pi+1) < A\cdot C/B^2 < (\pi+1)/(\pi-1) \leq 2$ can be deduced easily, so cancellation causes the subtraction $B^2 - A\cdot C$ to be performed exactly; all the error in D due to roundoff comes entirely from the two multiplications. This is why they must be carried out exactly during the recomputation of D that will now be needed to get it accurately enough.

§5: Exact Multiplication and Tricky Addition

The third arithmetic style with a fused multiply-add, when test values $x = 0$ and $y = 2^{-62}$ in §3 above, can easily perform exact multiplication in MATLAB by splitting products thus:

```
P = B*B ;  dP = [P, B]*[-1; B] ;  Q = A*C ;  dQ = [Q, A]*[-1; C] ;
```

These assignments actually produce P = (B·B rounded to 53 sig. bits) , dP = B·B – P exactly, Q = (A·C rounded to 53 sig. bits) , and dQ = A·C – Q exactly. Because rounding conforms to IEEE Standard 754, dP and dQ fit into 52 sig. bits; each has at least one trailing zero bit.

The other arithmetic styles accomplish a similar splitting at the cost of more effort by using a tricky algorithm attributed to G.W. Veltkamp by T.J. Dekker (1971). It breaks operands into two half-width fragments barely narrow enough that the product of any two fragments is exact since it fits into 53 sig. bits. Here is a MATLAB program that breaks operands:

```
function  [Xh, Xt] = break2(X)
%  [Xh, Xt] = break2(X)  produces  Xh = X rounded to  26 sig. bits
%  and  Xt = X - Xh  exactly in  26 sig. bits,  so products like
%  Xh*Xh,  Xh*Xt   and  Xt*Xt  can all be computed exactly.
bigX = X*134217729 ;  %... = X*(2^27 + 1)
Y = (X - bigX) ;  Xh = Y + bigX ;  %...  DON'T OPTIMIZE  Y  AWAY!
Xt = X - Xh ;  return
```

A proof that this algorithm works well enough in arithmetic style #1 (every operation rounded correctly to 53 sig. bits) can be found in §5-6 of Dekker (1971). Then the next few MATLAB statements produce the exact products P+dP and Q+dQ split as before, and finally D :

```
[Ah, At] = break2(A) ;  [Bh, Bt] = break2(B) ;  [Ch, Ct] = break2(C) ;
P = B*B ;  dP = ((Bh*Bh - P) + 2*Bh*Bt) + Bt*Bt ;
Q = A*C ;  dQ = ((Ah*Ch - Q) + (Ah*Ct + At*Ch)) + At*Ct ;
D = (P-Q) + (dP-dQ) ;   %...  DON'T OMIT PARENTHESES!
```

Arithmetic style #2 (accumulating scalar products with 11 extra sig. bits) requires a slightly different program because of two technicalities. First, MATLAB rounds arithmetic operations *not* in a scalar product twice, once to 64 sig. bits and again to 53. This double rounding spoils correctness proofs if not the results produced by `break2(X)` and the last four lines of MATLAB code. To protect results from spoilage they are computed as scalar products as much as possible.

Then a second technicality intrudes: Scalar products [a, b, c, d]*[x; y; z; t] = a·x + b·y + c·z + d·t are normally accumulated left-to right, but MATLAB 5.3 on PCs accumulates them right-to-left. This really would spoil our results if MATLAB 5.3 on PCs performed its arithmetic in style #2, but its arithmetic style is #1. Still, prudence obliges us to test for right-to-left accumulation:

```
rl = [eps, 9, -9]*[eps; 9; 9] ;
```

If `rl` is nonzero then accumulation is right-to-left and the foregoing scalar product should be rewritten [d, c, b, a]*[t; z; y; x] when the order of accumulation matters, as it does here in the program for arithmetic style #2 :

```
[Ah, At] = break2(A) ;  [Bh, Bt] = break2(B) ;  [Ch, Ct] = break2(C) ;
P = B*B ;  Q = A*C ;
if  (rl),  %..  Scalar products accumulate right-to-left
      dP = [Bt, 2*Bh, P, Bh]*[Bt; Bt; -1; Bh] ;
      dQ = [At, At, Ah, Q, Ah]*[Ct; Ch; Ct; -1; Ch] ;
      D = [dQ, dP, Q, P]*[-1; 1; -1; 1] ;
    else   %...  Scalar products accumulate left-to-right
      dP = [Bh, P, 2*Bh, Bt]*[Bh; -1; Bt; Bt] ;
      dQ = [Ah, Q, Ah, At, At]*[Ch; -1; Ct; Ch; Ct] ;
      D = [P, Q, dP, dQ]*[1; -1; 1; -1] ;
```

By now the reader should have some idea of how badly inflated are the costs of developing and maintaining high quality floating-point software without arithmetic precision twice as wide as the given data and desired results. What about the cost of proving these tricky programs correct?

§6: Proofs?

The shortest proofs found so far for the correctness of the foregoing algorithms and programs are, as usual for floating-point, far longer and trickier than the algorithms and programs in question. Particularly tricky is the proof that "`D = (P–Q) + (dP–dQ)`" works for arithmetic styles #1 and #3. The (re)discovery of such proofs is left to extremely diligent students. Some day the best available proofs (contributions are welcome) will be appended to these notes with due credits.

§7: Tests

At first sight an obvious way presents itself to generate coefficients to test any quadratic equation solver: Choose coefficient A and roots z_1 and z_2 arbitrarily, then compute $B := (z_1 + z_2)\cdot A$ and $C := z_1\cdot z_2\cdot A$, and then see how accurately the program in question solves the quadratic equation $Az^2 - 2Bz + C = 0$. This test's fatal flaw arises from rounding errors in the computed coefficients for which the correct roots are no longer exactly z_1 and z_2.

Another scheme would choose integer coefficients for which exact roots can be computed easily enough by automated algebra systems like DERIVE, MAPLE and MATHEMATICA. The hard part of this scheme is choosing integer coefficients big enough (with enough digits) to induce typical rounding errors in the program under test, and correlated enough to generate pathologically close roots that will challenge its accuracy. For instance, integer coefficients

$A := 2017810\cdot 8264001469$, $B := 39213\cdot 229699315399$, $C := 45077\cdot 107932908389$

(they are products of integers small enough to be converted from decimal to `double` binary floating-point perfectly after "**.**0D0" is appended to their digits) barely fit exactly into `double` and should produce the tiniest possible negative discriminant $D := B^2 - A\cdot C = -1$. The roots are

$$z = (B \pm \sqrt{D})/A = 0.5401558879570784\ldots \pm \imath\cdot 5.9969350369679512\ldots/10^{17} .$$

Examples with complex roots so nearly coincident as these are tedious to generate in quantity.

Competent tests are often more difficult to design than was the program under test. Ideally, tests should be arbitrarily numerous, randomized, prioritized to expose gross blunders soon, filtered to avoid wasting time testing the same possibilities too often, distributed densely enough along boundaries where bugs are likely to hide, reproducible whenever tests that exposed a bug have to be rerun on a revised version of the program under test, and fast. It's a lot to ask.

Tests to be proposed here have been simplified to focus upon accuracy in the face of roundoff. Omitted are tests that would exercise the full range of magnitudes of coefficients and roots, or would check that exceptional coefficients like ∞ or NaN or $A = 0$ are handled correctly. One of the questions to be addressed asks

"How much more do accuracy tests cost if access to fast extra-precise arithmetic is denied?"

The construction of one test battery begins with the *Fibonacci* numbers $F_n := F_{n-1} + F_{n-2}$ for $n = 2, 3, 4, \ldots, 78$ in turn starting from $F_0 := 0$ and $F_1 := 1$. (F_{79} is too big to fit into the 53 sig. bits of a `double` number.) F_n is the integer nearest $\tau^n/\sqrt{5}$ where $\tau := (1 + \sqrt{5})/2 \approx 1.618\ldots$. The quadratic equation $F_n\cdot z^2 - 2\cdot F_{n-1}\cdot z + F_{n-2} = 0$ has discriminant ${F_{n-1}}^2 - F_n\cdot F_{n-2} = (-1)^n$, so its two roots are $(F_{n-1} \pm I_n)/F_n$, wherein

$$I_n := \sqrt{((-1)^n)} = \{ \text{if } n \text{ is even then } 1 \text{ else } \imath := \sqrt{-1} \} .$$

As $n \to +\infty$ the roots become more nearly coincident, approaching $1/\tau \approx 0.618\ldots$. For smaller integers n the integer coefficients F_n are too small to generate typical rounding errors when acted upon by the quadratic equation solver's arithmetic. These coefficients have to be enlarged.

To enlarge and randomize the test coefficients let R be a random number uniformly distributed between 2^{52} and $2^{53} - 1$. Such a number R can be generated by a MATLAB assignment like "`R = 1/eps + rand(1)*(1/eps - 1) ;`" because MATLAB's eps = 2^{-52}. Then assign

`M = floor(R/`F_n`) ;  A = M*`F_n` ;  B = M*`F_{n-1}` ;  C = M*`F_{n-2}` ;`

to produce integer coefficients A, B and C whose somewhat scrambled bits still fit into 53 sig. bits without changing the quadratic equation's roots $(F_{n-1} \pm I_n)/F_n$. Recomputing R and M for each new n randomizes as well as enlarges the coefficients, though less so as n increases.

Another such battery of test coefficients can be constructed by replacing Fibonacci numbers F_n by members of a shorter sequence $f_n := \alpha \cdot f_{n-1} + f_{n-2}$ for any small integer $\alpha > 1$ starting from $f_0 := 0$ and $f_1 := 1$. Further exploration of such test batteries will not be pursued here since we have other fish to fry.

The accuracy of the program under test will be gauged from the differences between its roots z and the correct roots $(F_{n-1} \pm I_n)/F_n$. Ideally the latter should be computed at least a few bits more accurately than z before subtraction from z. This is straightforward if the correct roots and the differences can be computed extra-precisely. Otherwise, if the same `double` arithmetic as was used to compute z is the only arithmetic available, tricks must be incorporated into the test lest its own roundoff obscure the accuracy of a finely crafted program under test. Again, the lack of convenient access to extra-precise arithmetic inflates costs.

One trick exploits the correct roots' approach to $1/\tau \approx 0.618\ldots$. This is approximated closely enough by 5/8 = 0.625 that the difference between a correct root and the root z being tested can be recast when $n \geq 6$ to $(z - 0.625) + (0.125 \cdot F_{n-6} \pm I_n)/F_n$ with an apt choice for $\pm$. Validation of this unobvious expression for the desired difference is left again to the diligent student. Its advantage comes from exact cancellation in the subtraction $(z - 0.625)$, which depresses a subsequent contribution from the division's roundoff by an order of magnitude. This trick does nothing to enhance the assessment of the accuracy of a root's imaginary part when n is odd; then more trickery along the lines explored in §5 for the computation of D has to be employed. But our tests will cheat by assessing instead the accuracy of the computed value of D, since the imaginary part cannot be much more accurate than that.

An altogether different kind of accuracy assessment can be inferred indirectly but very quickly from the quadratic $Ax^2 - 2Bx + C$ and its derivative evaluated extra-precisely at the approximate roots x being tested without ever computing the correct roots. This kind of assessment loses its appeal if extra-precise arithmetic has to be simulated in software, in which case correct roots of the quadratic might as well be computed extra-accurately. For more complicated equations whose roots cannot be computed explicitly from a simple formula, this kind of indirect assessment is the only kind available, and it is trustworthy in critical cases only if performed with extra-precise arithmetic.

§8: MATLAB Programs

```
function  [Z1, Z2] = qdrtc(A, B, C)
%  [z1, z2] = qdrtc(a, b, c)  solves the quadratic equation
%  a*z^2 - 2*b*z + c == 0  for roots  z1  and  z2  computed
%  accurately out to the last sig. bit or two.  Real arrays
%  of coefficients  a, b, c  yield arrays of roots  z1, z2 .
%  To ease root-tracing,  real roots are ordered:  Z1 <= Z2 .
%  NO PRECAUTIONS AGAINST PREMATURE OVER/UNDERFLOW,  NOR NANS
%  NOR INFINITIES AMONG THE COEFFICIENTS,  NOR  a == 0 ,
%  HAVE BEEN TAKEN IN THIS VERSION OF THE PROGRAM.
%                                          (C) 2004  W. Kahan

sA = size(A) ;  sB = size(B) ;  sC = size(C) ;
sZ = max([sA; sB; sC]) ;  Z1 = ones(sZ(1), sZ(2)) ;
if any( [sZ-sA, sZ-sB, sZ-sC] ) %...  mix scalars and arrays
    if     (sum(sA)==2),  A = A(Z1) ;
    elseif (sum(sB)==2),  B = B(Z1) ;
    elseif (sum(sC)==2),  C = C(Z1) ;
    else  error('Sizes of  qdrtc(A,B,C)''s  arguments mismatch.')
  end,  end
A = A(:) ;  B = B(:) ;  C = C(:) ;  Z1 = Z1(:) ;
if any(imag([A; B; C]))
    error('qdrtc(A, B, C)  accepts only real  A, B, C.'),  end
Z2 = Z1 ;  %...  Allocate initial memory for roots.

D = dscrmt(A, B, C) ;  %...  Discriminant:  see file  dscrmt.m

nD = (D <= 0) ;
if any(nD)  %... Complex conjugate or coincident real roots
    Z1(nD) = B(nD)./A(nD) + sqrt(D(nD))./A(nD) ;
    Z2(nD) = conj(Z1(nD)) ;  end

nD = ~nD ;  if any(nD)  %...  Distinct real roots
    S = B(nD) ;
    S = sqrt(D(nD)).*( sign(S) + (S==0) ) + S ;
    Z1(nD) = S./A(nD) ;  Z2(nD) = C(nD)./S ;  end

nD = (Z1 < Z2) ;  if any(nD)  %...  Sort real roots
    S = Z1(nD) ;  Z1(nD) = Z2(nD) ;  Z2(nD) = S ;  end

Z1 = reshape(Z1, sZ(1), sZ(2)) ;
Z2 = reshape(Z2, sZ(1), sZ(2)) ;

return  %...  End  qdrtc.m
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

function  [Z1, Z2] = qdrtc0(A, B, C)
%  [z1, z2] = qdrtc0(a, b, c)  solves the quadratic equation
%  a*z^2 - 2*b*z + c == 0  for roots  z1 and z2 .  Real arrays
%  of coefficients  a, b, c  yield arrays of roots  z1, z2 .
%  To ease root-tracing,  real roots are ordered:  Z1 <= Z2 .
%  NO PRECAUTIONS AGAINST PREMATURE OVER/UNDERFLOW,  NOR NANS
%  NOR INFINITIES AMONG THE COEFFICIENTS,  NOR  a == 0 ,  NOR
%  INACCURACY HAVE BEEN TAKEN IN THIS VERSION OF THE PROGRAM.
%                                          (C) 2004  W. Kahan
```

Program `qdrtc0` differs from `qdrtc` in only two ways: First "`qdrtc0`" appears everywhere in place of "`qdrtc`". Second, the line

"`D = dscrmt(A, B, C) ;  %...  Discriminant:  see file  dscrmt.m`"

in `qdrtc` has been replaced by the obvious formula in a line

"`D = B.*B - A.*C ;  %...  Discriminant,  perhaps inaccurate.`"

in `qdrtc0` . Thus, tests can compare this obvious way to solve a quadratic with the elaborate and, we hope, more accurate way embodied in `dscrmt` within `qdrtc` .

= =

```
function  D = dscrmt(A, B, C)
%  dscrmt(A, B, C) = B.*B - A.*C  computed extra-precisely
%  if necessary to ensure accuracy out to the last sig. bit
%  or two.  Real columns  A, B, C  must have the same size.
%  This program is intended to work with versions  3.5 - 6.5
%  of  MATLAB  on  PCs,  Power Macs,  and old  680x0 Macs.
%                                      (C) 2004  W. Kahan

%  Determine  Matlab's  Arithmetic Style  AS :
y = 0.5^31 ;  z = -y*y ;
x = [1+y, 1]*[1-y; -1] ;  y = [1, 1+y]*[-1; 1-y] ;
x0 = (x==0) ;  xz = (x==z) ;  y0 = (y==0) ;  yz = (y==z) ;
AS = x0*y0+ 2*xz*yz + 3*x0*yz + 4*xz*y0 ;

%  Determine whether  MATLAB  adds scalar products right-to-left :
rl = ( [eps, 9, -9]*[eps; 9; 9] > 0 ) ;

if  (AS == 0) | ( (AS == 3) & rl ) | ( (AS == 4) & (~rl) )
  ArithmeticStyle = AS ,  RightToLeft = rl ,
  disp(' Something strange has happened!  Please inform  W. Kahan')
  disp(' about your computer and your version of  Matlab  because')
  error(' dscrmt  did not recognize  Matlab''s  arithmetic style.')
  end

%  Is the obvious way to compute  D  adequately accurate?
if  (AS == 2)  %...  Sum scalar products to  64  sig. bits
    pie = 1024 ;  n = length(A) ;  D = zeros(n,1) ;
    for  j = 1:n
        D(j) = [B(j), A(j)]*[B(j); -C(j)] ;
      end  %...  of loop on  j
  else  %...  All arithmetic rounds to  53  sig. bits
    pie = 3 ;  D = B.*B - A.*C ;  end
E = B.*B + A.*C ;
k = ( pie*abs(D) >= E ) ;
if all(k),  return, end  %... If the obvious way was good enough.

%  Recompute those values of  D  too inaccurate the obvious way.
k = ~k ;
a = A(k) ;  b = B(k) ;  c = C(k) ;
p = b.*b ;  q = a.*c ;  n = length(a) ;
dp = p ;  dq = q ;  %... allocate memory.

if  (AS > 2)  %...  Use the hardware's Fused Multiply-Add
    if rl,  for  j = 1:n
             dp(j) = [b(j), p(j)]*[b(j); -1] ;
               dq(j) = [a(j), q(j)]*[c(j); -1] ;  end
```

```
      else  for  j = 1:n
             dp(j) = [p(j), b(j)]*[-1; b(j)] ;
               dq(j) = [q(j), a(j)]*[-1; c(j)] ;  end
      d = (p-q) + (dp-dq) ;  end
  else  %...  Break operands into half-width fragments
    [ah, at] = break2(a) ;  [bh, bt] = break2(b) ;
    [ch, ct] = break2(c) ;
    if  (AS < 2)  %...  All arithmetic rounds to  53  sig. bits
        dp = ((bh.*bh - p) + 2*bh.*bt) + bt.*bt ;
        dq = ((ah.*ch - q) + (ah.*ct + at.*ch)) + at.*ct ;
        d = (p-q) + (dp-dq) ;
      else  %...  Arithmetic may round to  64  and then  53 s.b.
        if rl  %...   Sum scalar products right-to-left
            for j = 1:n
                dp(j) = [bt(j), 2*bh(j), p(j), bh(j)]* ...
                        [bt(j);  bt(j);   -1,  bh(j)] ;
                dq(j) = [at(j), at(j), ah(j), q(j), ah(j)]* ...
                        [ct(j); ch(j); ct(j);  -1;  ch(j)] ;
              end %...  of loop on  j
            d = [dq, dp, q, p]*[-1; 1; -1; 1] ;
          else  %...  Sum scalar products left-to-right
            for j = 1:n
                dp(j) = [bh(j), p(j), 2*bh(j), bt(j)]* ...
                        [bh(j);  -1;   bt(j);  bt(j)] ;
                dq(j) = [ah(j), q(j), ah(j), at(j), at(j)]* ...
                        [ch(j);  -1;  ct(j); ch(j); ct(j)] ;
              end %...  of loop on  j
            d = [p, q, dp, dq]*[1; -1; 1; -1] ;
          end  %...  of extra-precisely summed scalar products
      end  %...  of arithmetic with half-width fragments
  end  %...  Now  d  is fairly accurate.

D(k) = d ;  return  %... End  dscrmt.m
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

function  [Xh, Xt] = break2(X)
%  [Xh, Xt] = break2(X)  produces  Xh = X rounded to 26 sig. bits
%  and  Xt = X - Xh  exactly in 26 sig. bits,  so products like
%  Xh.*Xh,  Xh.*Xt,  Xt.*Xt  can all be computed exactly.  But if
%  arithmetic double-rounds to 64 and then 53 sig. bits,  some of
%  Xt  or  Xh  may require 27 sig. bits for all I know.   W. K.

bigX = X*134217729 ;  %... = X*(2^27 + 1)
Y = X - bigX ;  Xh = Y + bigX ;  %... DON'T OPTIMIZE  Y  AWAY!
Xt = X - Xh ;  return  %... End break2
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

%  qdrtctst.m  is a Matlab script that tests  qdrtc.m
%  and puts its results into a diary file  qdrtctst.res

diary qdrtctst.res
format long e %... or  long g  on later versions of Matlab
format compact

'Test roots with extremely tiny imaginary parts:'
A = 2017810*8264001469,  B = 39213*229699315399,  C = 45077*107932908389
```

```
Z = (B + sqrt(-1))/A
D = -1
Derr = dscrmt(A, B, C) - D ,  Derr0 = (B*B - A*C) - D
[Z1, Z2] = qdrtc(A, B, C),  [z1, z2] = qdrtc0(A,B,C)
qdrtcerr = Z1-Z,  qdrtc0err = z1-Z
'- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -'

'Produce test results for  Table 2:'
B = 94906267 ;  A = B - 1.375 ;  C = B + 1.375 ;
B = [B; B+0.375] ;  A = [A; A+0.75] ;  C = [C; C] ;
ABC = [A, B, C]
D = [121/64; 1]
Derr = dscrmt(A,B,C) - D,  Derr0 = (B.*B - A.*C) - D
[Z1, Z2] = qdrtc(A, B, C),  [z1, z2] = qdrtc0(A,B,C)
dZ = [-1, 1]*[Z1, Z2],  dz = [-1, 1]*[z1, z2]
'- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -'

%  Time  qdrtc  and  qdrtc0  on large random arrays:
A = rand(100) - 0.5 ;
B = rand(100) - 0.5 ;
C = rand(100) - 0.5 ;
' Running  qdrtc  on  10000  random coefficients'
T = clock ;
[Z1, Z2] = qdrtc(A, B, C) ;
T = etime(clock, T) ;

' Running  qdrtc0  on  10000  random coefficients'
t = clock ;
[z1, z2] = qdrtc0(A, B, C) ;
t = etime(clock, t) ;

TIMEqdrtc_vs_qdrtc0 = [T, t]

'- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -'
'Generate test coefficients from  Fibonacci  numbers  F_n :'
F = ones(79,1) ;  I2 = F ;  F(1) = 0 ;  I2(2) = -1 ;
for n = 3:79,  F(n) = F(n-1)+F(n-2) ;
              I2(n) = -I2(n-1) ;  end
%  F(n+1) = F_n ;  I2(n+1) = F(n)^2 - F(n+1)*F(n-1) = (-1)^n

% ...  To check on the test, ...
% 'Display the first dozen  Fibonacci  numbers in a row:'
% F0_11 = [ [0:11]; F(1:12)']
% 'Check the first  20  values  F(n)^2 - F(n+1)*F(n-1) - I2(n+1)'
% Derr = norm( F(2:21).*F(2:21) - F(3:22).*F(1:20) - I2(3:22) )

%  I = sqrt( I2(3:79) ) ;  % THIS FAILS BECAUSE OF A BUG IN MATLAB 6.5

J = sqrt(-1) ;  %...  This is an alternative way to get  I .
I = ones(1,39) ;  I = [I; I*J] ;  I = I(:) ;  I = I(1:77) ;

A = F(3:79) ;  B = F(2:78) ;  C = F(1:77) ;  D = I2(3:79) ;

%  Compute first six  [cZ1, cZ2] = [Roots] - 5/8  crudely,
%     and the rest closely:
c = 0.625 ;  cZ1 = I ;  cZ2 = I ;  %...  to allocate memory
j = [1:6]' ;
cZ1(j) = B(j) - c*A(j) ;
```

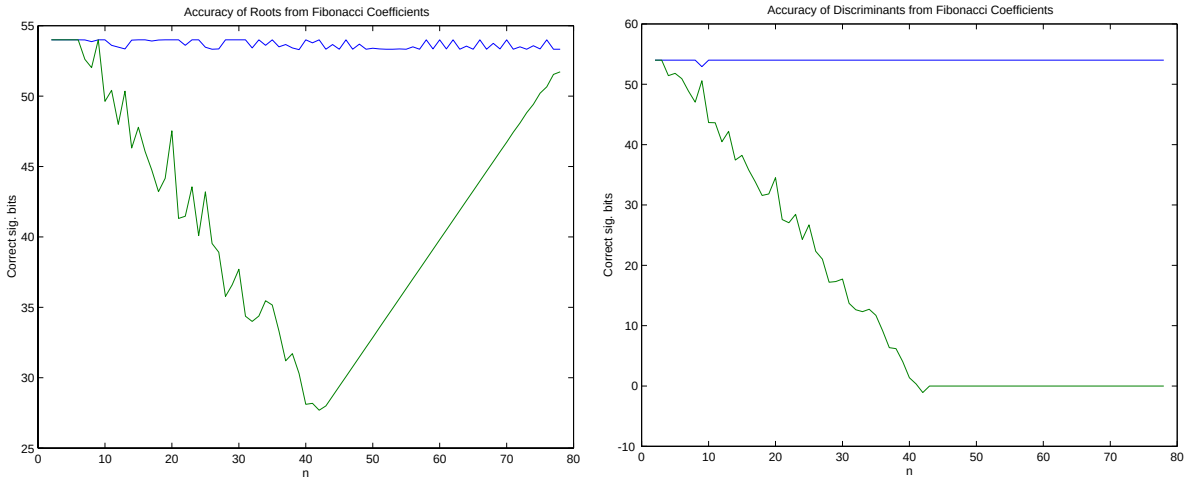
```
j = [7:77]' ;
cZ1(j) = -0.125*A(j-6) ;
cZ2 = (cZ1 - I)./A ;
cZ1 = (cZ1 + I)./A ;

%  Enlarge and randomize the test coefficients:
M = 1/eps ;  M = rand(77, 1)*(M-1) + M ;
M = max(1, floor(M./A)) ;  %...  Let's not let  M == 0 .
A = M.*A ;  B = M.*B ;  C = M.*C ;
trueD = M.*M.*I2(3:79) ;  %... True Discriminant

%  Compute  77  roots with and without  dscrmt :
' Running  qdrtc  on  Fibonacci  coefficients'
T = clock ;
[Z1, Z2] = qdrtc(A, B, C) ;
T = etime(clock, T) ;

' Running  qdrtc0  on  Fibonacci  coefficients'
t = clock ;
[z1, z2] = qdrtc0(A, B, C) ;
t = etime(clock, t) ;

TIMEqdrtc_vs_qdrtc0 = [T, t]

%  ' Display the first dozen zeros to test the test:'
%  j = [1:12]' ;
%  TrueZ = [cZ1(j), cZ2(j)] + c
%  qdrtcZ = [Z1(j), Z2(j)]
%  qdrtc0z = [z1(j), z2(j)]

%  Compute no. of correct sig. bits in roots:
L2e = 1/log(2) ;  eta = eps*eps ; %...  to prevent  .../0
S1 = max( abs(c + cZ1)',  eta ) ;
S2 = max( abs(c + cZ2)',  eta ) ;
Zbits = max([ abs((Z1-c)-cZ1)'./S1; abs((Z2-c)-cZ2)'./S2 ]) ;
Zbits = min( 54, -log(Zbits' + eta)*L2e ) ;
zbits = max([ abs((z1-c)-cZ1)'./S1; abs((z2-c)-cZ2)'./S2 ]) ;
zbits = min( 54, -log(zbits' + eta)*L2e ) ;

%  Plot no. of correct sig. bits in roots:
N = [2:78]' ;
plot(N, Zbits,  N, zbits)
xlabel(' n ')
ylabel(' Correct sig. bits ')
title(' Accuracy of Roots from Fibonacci Coefficients ')
pause

%  Compute no. of correct sig. bits in discriminants:
D = dscrmt(A, B, C) ;
d = B.*B - A.*C ;
Dbits = min(54, -log(abs((D-trueD)./trueD) + eta)*L2e ) ;
dbits = min(54, -log(abs((d-trueD)./trueD) + eta)*L2e ) ;
plot(N, Dbits,  N, dbits)
xlabel(' n ')
ylabel(' Correct sig. bits ')
title(' Accuracy of Discriminants from Fibonacci Coefficients ')
pause
```

§9: Test Results

The test script `qdrtctst` has been run on MATLAB versions 3.5 to 6.5 on PCs, versions 3.5 to 5.2 on both current iMacs and Power Macs, and on an old 68040-based Mac Quadra. The results differ significantly from one computer to another, and from one version to another, only in the relative speeds of the tricky `qdrtc` *vs.* the obvious `qdrtc0`. Some combinations of version and hardware run loops of the form `for j = ... ...; end` slower than other combinations by an order of magnitude, thus penalizing severely the schemes in `dscrmt` that are forced to access 11 extra sig. bits or fused multiply-adds only through awkward scalar products when the given coefficients A, B, C are arrays. This "feature" of MATLAB undermines its ability to compare the intrinsic speeds of the algorithms from which the programs were derived.

A(nother) nasty bug in MATLAB 6.5 was exposed by the test script. Its attempted computation of `I = sqrt([1 -1 1 -1 1 -1 ...])` = `[1 i 1 i 1 i ...]`, where `i` = $\sqrt{-1}$, produced instead `[1 NaN 1 NaN 1 NaN ...]` as if the script's `sqrt` had not been told about the complex numbers that MATLAB's other `sqrts` produce in all other contexts I tried. The bug has been bypassed in `qdrtctst`.

The accuracies of the algorithms compare very much as might be expected. The graphs below plot accuracies, gauged in sig. bits, against the subscript n of the Fibonacci number F_n from which the quadratic's coefficients $A = M{\cdot}F_n$, $B = M{\cdot}F_{n-1}$, $C = M{\cdot}F_{n-2}$ were derived. The first graph exhibits the accuracies of the computed roots; the second … discriminants. The second graph also reveals the accuracies of the imaginary parts of complex roots when n is odd. In both graphs the upper line belongs to `qdrtc` and its tricky `dscrmt`, and the lower line to `qdrtc0`.



These graphs, which came from MATLAB 5.2 on an old Mac Quadra, are typical of graphs from other versions on other hardware produced by the same test script `qdrtctst`. In these tests `qdrtc` produced roots always accurate to more than 52 sig. bits in both real and imaginary parts, while `qdrtc0` produced roots whose real parts were accurate to roughly $27 + 0.7{\cdot}|n - 41|$ sig. bits, imaginary parts (like discriminants when n is odd) to roughly $\max\{0, 1.38{\cdot}(41 - n)\}$.

How often does accuracy lost by a simple program like `qdrtc0` hurt someone later? Not often. Nobody knows for sure. Nobody is keeping score. And users of `qdrtc` wouldn't care.

§10: Conclusions and Recommendations

People who drive cars nowadays no longer have to know how to crank an engine by hand to start it, how to double-clutch when changing gears, how to change a spark-plug or tire, nor how to drain water from a carburettor bowl, though such knowledge used to be prerequisite. And now cars have tempered glass windows, seat-belts and air-bags to protect us somewhat against our own mistakes and others'. A heavier burden borne by automobile manufacturers lightens burdens for the rest of us. This kind of redistribution of burdens advances civilization. Usually. I hope.

The essence of civilization is that we benefit from others' experience without having to relive it.

Those of us who have grown old fighting against the vagaries of floating-point arithmetic and ill-conceived compiler "optimizations" take pride in our victories in that battle. But bequeathing the same battle to the generations that follow us would be contrary to the essence of civilization. Our experience indicts programming languages and development systems as sources of too many of the vagaries against which we have had to fight. Too many are unnecessary, as are certain of the tempting "optimizations" safe for integers but occasionally fatal for floating-point. Ill-advised optimizations inspired by benchmarks oblivious to all but speed constitute a story for another day.

If the designers and implementors of programming languages and development systems wish to diminish their systems' capture cross-section for mistakes made by clever but numerically naive programmers, as are almost all of us, we must build two kinds of capabilities into those systems:

First are augmented aids towards debugging floating-point software distressed by roundoff. Some are described in my tract posted at http://www.cs.berkeley.edu/~wkahan/Mindless.pdf . They would change that debugging task from nearly impossible for almost all programmers, as it is now, to merely difficult. Difficult is still costly. To further reduce costs to both end-users and developers of floating-point software, we must reduce the incidence of bugs due to roundoff.

A second capability, if easy enough to use that almost all programmers use it by default, would reduce by orders of magnitude the incidence of distress caused by roundoff. That capability is extra-precise arithmetic not too slow. By far the majority of laptop and desktop machines today have such a capability in their hardware going unused for lack of apt linguistic support. Now atrophy threatens that capability in tomorrow's machines. Someone has to persuade hardware and language designers and implementors to supply and support extra-precise arithmetic in a way that will be used routinely by most programmers regardless of benchmarks oblivious to all but speed.

> Precision higher than `double`, albeit too slow, is better than no higher precision at all.

A little more precision than `double`, and not too slow, is better than far higher precision too slow.

Who shall persuade hardware and language designers and implementors to do what needs doing?

Not market forces generated by the community of programmers and end-users of floating-point software. In 1998 Dr. James Gosling, an architect of Java, characterized that community well:

> " 95% of the folks out there are completely clueless about floating-point."

A closer estimate may be 99.9% . The clueless folks cannot be expected to know nor demand what needs doing. The demand has to come from project managers concerned with costs, and

from actuaries and engineers concerned with risks, and most of all from thoughtful professionals in the computing industry. They all have to be alerted to costs and risks of which most appear unaware nowadays though these costs and risks weighed upon the minds of an older generation. The older generation has to enlighten the younger as best we can. It is the essence of civilazation.

And, if skeptical about civilization's advances, think about self-defence.

§11: Citations

J. Darcy & W. Kahan (1998) "How Java's Floating-Point Hurts Everyone Everywhere" posted at http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf. Puts James Gosling's quote in context.

W. Kahan (1981) "Why do we need a floating-point arithmetic standard?" Retypeset by D. Bindel, Mar. 2001, posted at http://www.cs.berkeley.edu/~wkahan/ieee754status/Why-IEEE.pdf .

W. Kahan (2004) "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?" posted at http://www.cs.berkeley.edu/~wkahan/Mindless.pdf .

T.J. Dekker (1971) "A Floating-Point Technique for Extending the Available Precision" pp. 234-242 in *Numerische Mathematik* **18**. Attributes breaking & splitting algorithms to G.W. Veltkamp.

§12: Proofs — TO BE APPENDED SOME DAY