A Mathematical Curiosity

David H. Bailey has a sly sense of humor. The sum he supplied to test Delta Debugging is an abbreviation of a mathematical curiosity concerning the graph of $y := g(x)$ that runs from 0 to π as x runs from 0 to π. This curve is continuous but extremely wiggly in a small way; and its every wiggle is also extremely wiggly in a small way; and so on. The curve is plotted on the next page followed by an enlarged section of the curve over the subinterval $0.999995 < x < 1.000005$ whereon $y \approx 1.404045 \pm 0.000003$. The curve comes from a simple infinite series that defines

$$g(x) := x + \sum_{k>0} 2^{-k}\cdot\sin(2^k\cdot x) = \pi - g(\pi - x) .$$

Is the curve's length L finite? If so, can it be computed? L would be defined by an integral,

$$L := \int_0^\pi \sqrt{\left(1 + g'(x)^2\right)}\cdot dx ,$$

if the derivative $g'(x)$ existed, but its series $1 + \sum_{k>0} \cos(2^k\cdot x)$ does not converge; $g'(x) = +\infty$ if any $2^k\cdot x/\pi$ is an integer, else $-\infty$ if $3\cdot 2^k\cdot x/\pi$ is an integer. L poses numerical challenges.

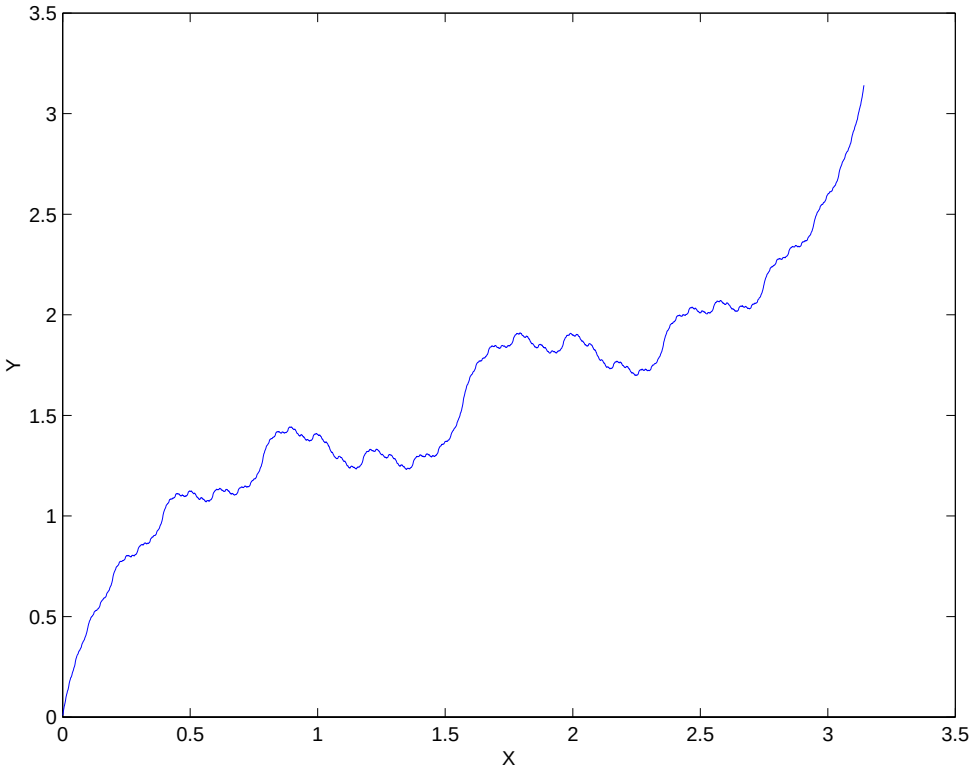
Bailey offered abbreviated sums, replacing $\sum_{k>0}$ by $\sum_{1\le k\le 5}$ and $\int_0^\pi$ by a sum of finitely many chords, to test *Delta Debugging*. He smoothed the graph, reducing its length below 6. His number of chords, fixed at 1000000, is not a variable Delta Debugging has yet been designed to alter; this limitation and some others has limited Delta Debugging's potency, as we shall see.

Our first challenge is the series for $g(x)$. It converges at least as fast as $\sum_{k>0} 2^{-k} = 1$, so the number of terms beyond which they add negligibly to the sum is predictable from the floating-point arithmetic's precision. Since repeated invocations of $\sin(\ldots)$ would consume more time than we wish to spend, our program for $g(x)$ invokes $\sin(x)$ and $\cos(x)$ subprograms only once for each x and uses fast trigonometric identities like $\sin(2\cdot\theta) = 2\cdot\sin(\theta)\cdot\cos(\theta)$ to generate the series' subsequent terms. Here is one of two vcctorized MATLAB programs to compute $g(x)$:

```
function  y = dhb1ng(x)  %...  x  can be an array
%  y = dhb1ng(x) = x + SUM[k>0] sin(x*2^k)/2^k  using  cos(x*2^k)
%  SUM  is  Compensated  for its roundoff.
y = x ;  y1 = -y ;  oldy = y1 ;  c = 0*y ;  %... compensates
cs = cos(x) ;  sn = sin(x) ;  d = 1 ;
k = 0 ;  %...  needed to dampen numerical instability.
while ( any( (y ~= oldy)|(y ~= y1) ) & (k < 53) )
    k = k+1 ;  oldy = y1 ;  y1 = y ;  d = 0.5*d ;  %... = 1/2^k
    t = cs.*sn ;  cs = (cs + sn).*(cs - sn) ;  %... = cos(x*2^k)
    sn = 2*t ;  %... = sin(x*2^k)
    t = sn*d + c ;  %... Compensated Summation
    y = y1 + t ;  c = (y1 - y) + t ;  end
```

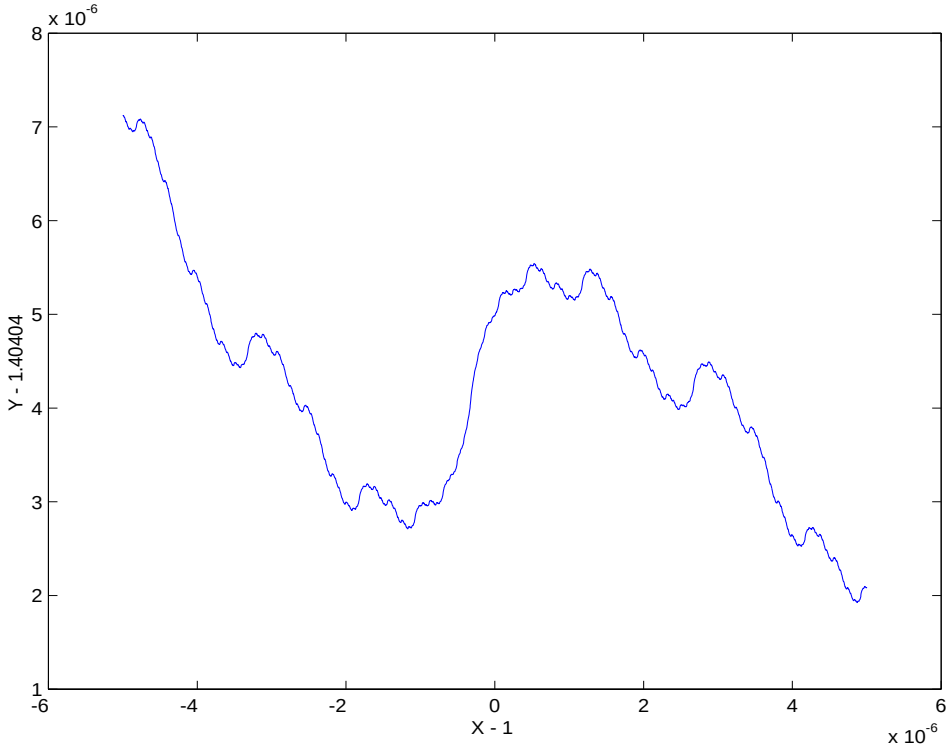
At first sight, variable `k` seems superfluous; but omitting it would allow `cs` and `sn` to explode exponentially because of roundoff when tiny elements of `x` caused those elements of `y` to need several more than 53 terms to stop changing, though these changes would have been negligible but for that spurious explosion. We need variable `k` for the test "`& (k < 53)`" determined by the arithmetic's precision, 53 sig. bits. Delta Debugging is not yet able to detect and alter precision-dependent thresholds like `53` so, to avoid it, a second MATLAB program was written.

A continuous but nowhere differentiable curve $y = g(x)$



Is the curve's length L finite?

A small section of the curve magnified by 1,000,000



The second MATLAB program needs no threshold for stability, but requires a trick and two divisions, and consequently runs noticeably slower than the first did; here is the second:

```
function  y = dhblng(x)  %...  x  can be an array
%  y = dhblng(x) = x + SUM[k>0] sin(x*2^k)/2^k  using  tan(x*2^k)
%  SUM  is  Compensated  for its roundoff.
tn = tan(x) ;  d = 1 ;  y = x ;  y1 = -y ;  oldy = y1 ;
eta = 1.5e-154 ;  %...  4/eta^2  is finite barely
c = 0*y ;  %... compensates for roundoff in summation
%...  Imagine  k = 1, 2, 3, ...  in turn ...
while any( (y ~= oldy)|(y ~= y1) ),  oldy = y1 ;  y1 = y ;
    d = 0.5*d ;  %... = 1/2^k
    sn = 2*tn./(1 + tn.*tn) ;  %... = sin(x*2^k)
    tn = 2*tn./( (1-tn).*(1+tn) + eta ) ;  %... = tan(x*2^k)
    t = sn*d + c ;  %... Compensated Summation
    y = y1 + t ;  c = (y1 - y) + t ;  end
```

Its trick is `eta` which prevents division by zero, keeping `tn = tan(x*2^k)` always finite, while otherwise leaving a nonzero divisor unaltered. Alas, the value assigned to `eta` depends upon the arithmetic's overflow threshold, to which Delta Debugging is oblivious. There would be no need for `eta` if programming languages could be persuaded to provide the *Presubstitution* of zero instead of *NaN* for ∞/∞ just where a programmer asked for it.

Both programs' graphs of $g(x)$ were plotted on the previous page; they match within 13 sig.dec.

Our second challenge is the integral $L = \int \ldots$. It will be approximated by a sum over, say, N panels for any positive integer N. Each panel's width is $h := \pi/N$, and covers the graph of $g(x)$ for x between $x_i := h \cdot i$ and x_{i+1} for $0 \le i < N$. Because of roundoff, each x_i is slightly wrong to an extent significant only when N is huge, as it must be to approximate the integral closely. The integral is approximated by a sum of all N panels' chord-lengths, each one …

$$\sqrt{\left((x_{i+1} - x_i)^2 + (g(x_{i+1}) - g(x_i))^2 \right)} .$$

(Bailey's program used " h " in place of " $x_{i+1} - x_i$ ", which is not bad if N is not too big.)

How well can the computed sums approximate length *L* ?

Compensated Summation of N approximating chord-lengths approximates integral *L* within a discretization error that would, if the graph of $g(x)$ were smooth, dwindle to zero roughly like (Constant)/N^2 as N increased until N got so huge that roundoff obscured the computed sum's last digit or two. But the graph is not smooth. Ordinary uncompensated summation incurs so much extra roundoff as to obscure the computed sum in roughly at least the last third of the arithmetic's sig. bits when N gets big enough. And if N gets too much too big, the rounding errors in $g(x_i)$ may obscure the computed sum almost entirely, compensated or not.

A MATLAB program to compute *L* ran too slowly, so *L* was computed from a few FORTRAN programs written to use arithmetics of different precisions and different summation processes. Hereunder are some of their results.

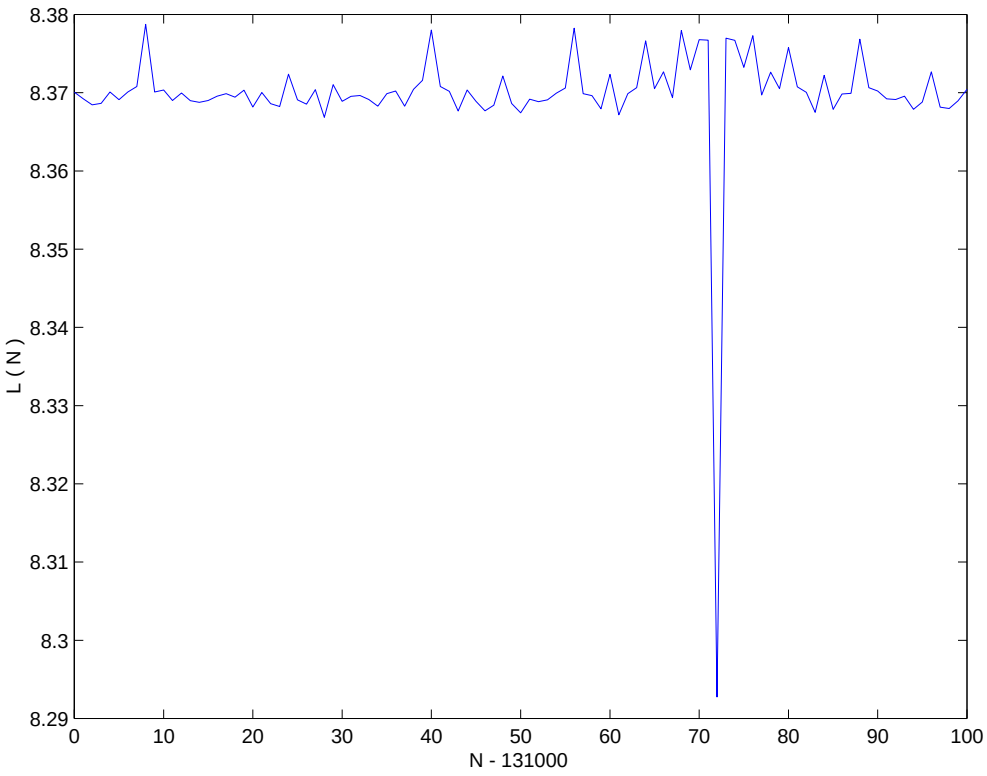
The FORTRAN programs accept any positive integer N not too big, and define $h := \pi/N$ and $x_i := i \cdot h$ for $i = 0, 1, 2, \ldots, N-1$ and N, and then approximate L by computing

$$L(N) := \sum_{0 \le i < N} \sqrt{\big((x_{i+1} - x_i)^2 + (g(x_{i+1}) - g(x_i))^2 \big)} \, .$$

The FORTRAN subprograms for $g(x)$ compute it from the same algorithm as is used by the second MATLAB program `dhblng(x)` above including its Compensated Summation; some subprograms used other processes to attenuate (or not) roundoff's contribution to $\sum \ldots$.

Integers N that are powers of 2 caused the series for $g(x_i)$ to terminate after fewer terms than usual, and to produce values $L(N)$ sooner and slightly smaller than for $L(N \pm 1)$. For example, $L(2^{17}) \approx 8.29275158$ takes less than a third as long to compute as does $L(2^{17}-1) \approx 8.37672557$ or $L(2^{17}+1) \approx 8.37696972$ in arithmetic of sufficiently high precision. In general $L(N)$'s graph is too ragged to be blamed upon roundoff. Here is a sample that straddles $N = 2^{17} = 131072$:

$L(N)$ for $131000 \le N \le 131100$

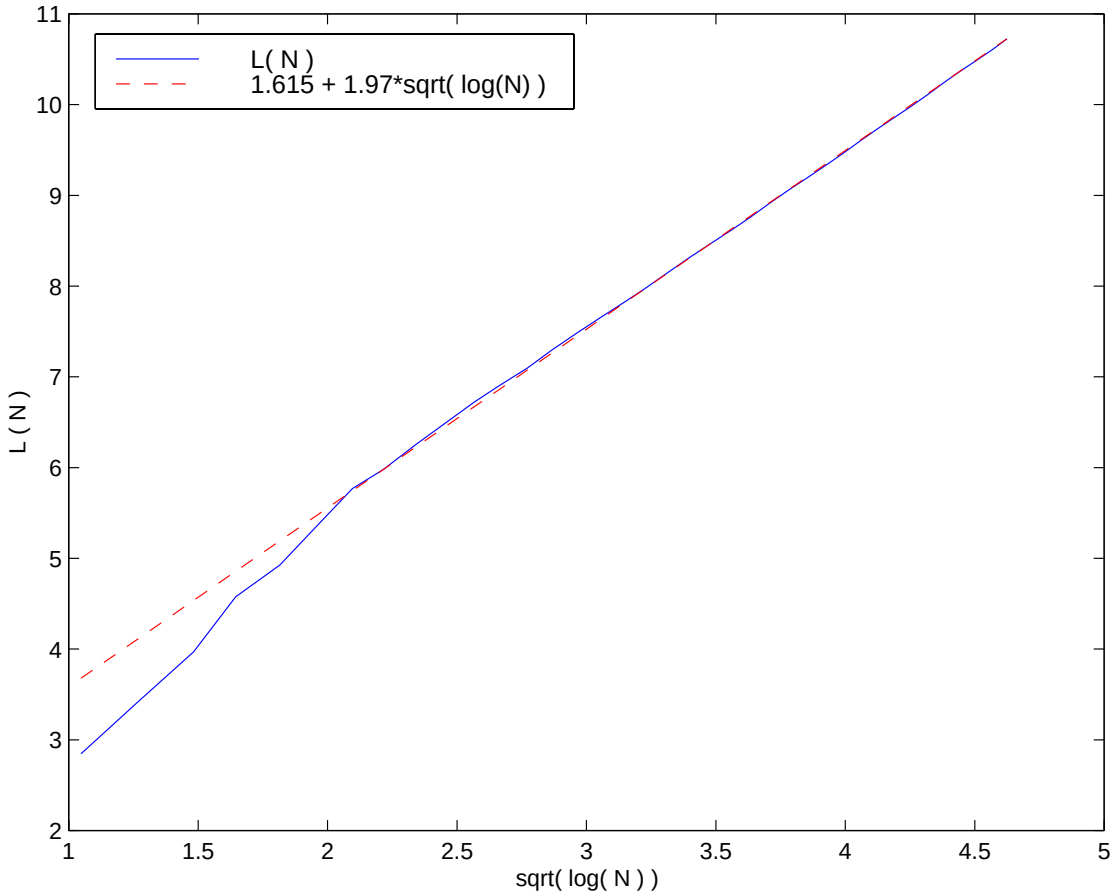


This behavior prejudiced the choice of integers N at which $L(N)$ was computed to plot its graph. Powers of 2 were avoided. Instead the sequence of integers N chosen was

$$3, 5, 9, 15, 27, 45, 81, \ldots, \; 645700815 = 3^{17} \cdot 5, \; 1162261467 = 3^{19}, \; 1937102445 = 3^{18} \cdot 5 \, ,$$

which ends where the next term, 3^{20}, exceeds the biggest 32-bit 2's-complement integer. The 38 values of $L(N)$ took an IBM T21 *ThinkPad* laptop almost a day to compute in long-double precision (64 sig.bits). These 38 values are plotted against $\sqrt{\log(N)}$ on the next page: …

L(N) *vs.* 1.615 + 1.97·√log(N)



This graph of L(N) provides persuasive if not conclusive evidence that L(N) → +∞ slowly as N → +∞ ; if so, the wiggly curve y = g(x) has infinite length *L* .

• •

Why should the foregoing graph-plots be believed?

What follows purports to corroborate the claims put forward above. Without some error-analyses and accuracy tests, the foregoing computations could be no more than vagaries of roundoff for all we know now. For instance, without the "`& (k < 53)`" test, the first MATLAB program above made `y` overflow to ∞ (*Can you see why?*) after k = 64 passes around the `while` loop while attempting to compute `dhb1ng(0.1)`, whereas both `dhb1ng(0.1 ± 5/2^56)` produced nearly the same correct value 0.454885220024027 after k = 56 passes around the loop. Thus can roundoff ruin a floating-point program at patches of otherwise innocuous data. Roundoff can embarrass many a simple program at innocuous data scattered almost everywhere over its domain; see an archetypal example in pp. 36-41 of `www.eecs.berkeley.edu/~wkahan/MktgMath.pdf`.

To trust a floating-point computation without investigating its provenance is imprudent.

Our error-analyses attend to programs for g(x) first, then programs for L(N) := $\sum_{0 \le i < N} \ldots$.

Error-Analysis of g(x) :
The programs for $g(x) := x + \sum_{k>0} 2^{-k}\cdot\sin(2^k\cdot x)$ obtain $\sin(\dots)$ from a trigonometric formula $\sin(2\cdot\theta) = 2\cdot\tan(\theta)/(1 + \tan^2(\theta))$. Another formula $\tan(2\cdot\theta) = 2\cdot\tan(\theta)/\big((1 - \tan(\theta))\cdot(1 + \tan(\theta))\big)$ would produce successive values of $\tan(2^{k-1}\cdot x)$, starting with $\tan(x)$, except for the extremely unlikely possibility of division by zero, which would create $\tan(\dots) = \pm\infty$ followed soon by a non-number *NaN* for $\sin(\dots) = \pm\infty/\infty^2$. Though unlikely, it has happened. For instance, when $\theta \approx \pi/8$ as closely as floating-point allows, $\tan(\theta) \approx 1/(1 + \sqrt{2})$ closely enough that the computed $\tan(2\cdot\theta) = 1$ exactly, followed by ∞ and then *NaN*. This mishap must be prevented.

One way to prevent it is to test the divisor $\big((1 - \tan(\theta))\cdot(1 + \tan(\theta))\big)$ for zero and, if so, break out of the `while` loop. For MATLAB programs, a simpler and faster way replaces this divisor by

$$\big((1 - \tan(\theta))\cdot(1 + \tan(\theta)) + \eta \big)$$

in which η is so tiny that it gets rounded away without altering any nonzero divisor, but not so tiny that $4/\eta^2$ overflows. A little bigger than $2/\sqrt{\text{Overflow threshold}}$ is tiny enough. After this replaces an infinite $\tan(2\cdot\theta)$ by $\pm 2/\eta$, the subsequent $\sin(4\cdot\theta)$ and $\tan(4\cdot\theta)$ will be replaced by $\pm\eta$ instead of zero but with the same effect. (Recall that MATLAB programs are vcctorized.)

Roundoff corrupts at most the last two sig. dec. of the computed $g(x)$. Let's find out why:

Roundoff in computing $\tan(2\cdot\theta)$ replaces it by $(1 \pm \varepsilon)\cdot\tan(2\cdot\theta)$ in which ε is not much bigger than the floating-point arithmetic's roundoff threshold; MATLAB's is $\varepsilon <$ `eps` $= 2^{-52} \approx 2.2\text{e}{-16}$. This replacement can alter subsequent computed values of $\tan(\dots)$ utterly; but these alterations will matter relatively little for two reasons. First, $(1 \pm \varepsilon)\cdot\tan(2\cdot\theta) = \tan(2\cdot\theta\cdot(1 \pm \bar{\varepsilon}))$ in which $\bar{\varepsilon} \approx \varepsilon\cdot\sin(4\cdot\theta)/(4\cdot\theta)$ can't exceed ε. Consequently roundoff replaces each term $2^{-k}\cdot\sin(2^k\cdot x)$ by something no worse than $2^{-k}\cdot(1 \pm \varepsilon)\cdot\sin(2^k\cdot x\cdot(1 \pm \varepsilon)^k)$. Secondly, the loop-count k can't grow arbitrarily big. It scarcely exceeds the arithmetic's number $-\log_2(\varepsilon)$ of sig.bits for all but positive arguments $x < 1$ and, for these, k never much exceeds $-\log_2(x\cdot\varepsilon)$. We shall disregard the interval $0 < x < 2\pi\cdot\varepsilon$ since $N < \min\{2^{31},\ 1/\varepsilon\}$ in our programs for $L(N) := \sum_{0 \le i < N} \dots$. This means that $1 \pm k\cdot\varepsilon$ approximates $(1 \pm \varepsilon)^k$ adequately for our purposes. But this still leads to an excessively pessimistic bound for the errors in the computed values of $2^{-k}\cdot\sin(2^k\cdot x)$.

Less pessimistic error estimates require probabilistic reasoning, among other things. The factor $(1 \pm \varepsilon)^k$ is more accurately a product $\prod_{1\le j\le k}(1 \pm \varepsilon_j) \approx 1 + \sum_{1\le j\le k} \pm\varepsilon_j$ in which each $\pm\varepsilon_j$ can be approximated by an independent random variate bounded between $\pm\varepsilon$ with mean zero. It soon follows that $|\sum_{1\le j\le k} \pm\varepsilon_j|$ is extremely unlikely to exceed the sum's *Standard Deviation*, roughly $\varepsilon\cdot\sqrt{k/6}$. Incidentally, since $\sin(\theta) \approx \tan(\theta) \approx \theta$ with negligible error while $|\theta| < \varepsilon/2$, roundoff's contribution to $\sin(2^k\cdot x)$ is ignorable while tiny $2^k\cdot x < \varepsilon/2$; but nonzero arguments x that tiny did not figure in our programs' computations of $L(N)$.

When all the foregoing estimates are collected, they imply that the error δg in a computed $g(x)$ is very unlikely to exceed roughly $0.27\cdot x\varepsilon\cdot|\log_2(\varepsilon)|^{3/2}$ for $x \ge 1$, or $0.27\cdot x\varepsilon\cdot|\log_2(x\cdot\varepsilon)|^{3/2}$ for $\pi/2^{31} \le x < 1$. (Compensated summation saves at most a few sig.bits in a computed $g(x)$.)

A long error-analysis, much longer than the program analyzed, has a capture-cross-section for error (mistakes) much larger than the program's. Experimental corroboration seems prudent.

The program `dhblng(x)` has been rerun, for arguments `x` scattered between $\pi/2^{31}$ and π, with redirected roundings, and with the arithmetic's precision reduced from 53 to 24 sig.bits, in MATLAB 3.5 on an Intel 302 (i386/387 upgraded to a Cyrix CX486DRx^2 and 83D87) and in MATLAB 6.5 on an IBM T21 *ThinkPad* (Intel Pentium III). Also run and rerun was a version of `dhblng(x)` that omitted compensated summation by deleting `c`. Thus on each computer there were sixteen runs to computate g*(*x*)* for each choice of `x`. Larger disparities among computed results would have revealed hypersensitivity to roundoff beyond the predictions of error-analysis.

Precision reduced to 24 sig.bits retained agreement in all but its last two sig.dec. with 53 sig.bits' results. Compensated summation improved the last few sig.bits. Redirected roundings changed no more than the last two or three sig.dec. of g*(*x*)*; the larger changes occurring with rounding toward $\pm\infty$ were often accompanied by larger numbers `k` of traversals of the `while` loop. This raises another question: When and why does the `while` loop terminate?

Normally we should deprecate a statement like "`while any(y ~= oldy)`" as a way to terminate traversals around a loop computing a sequence of convergent floating-point values `y`, because the computed values of `y` and `oldy` might never agree exactly though `y` has converged about as closely as roundoff allows. Then computed values of `y` could dither in a loop that never stopped.

That cannot happen to `dhblng(x)`. The series for g*(*x*)* converges so quickly that the increment `t` in the statement "`y = y1 + t`" soon becomes tiny enough to round away, leaving `y == y1`. This happens with compensated summation regardless of whether rounding is "*to nearest*" (the default) or redirected "*to* $\pm\infty$". or "*to* 0". It happens also without compensated summation because the recurrence for tan*(…)* and sin*(…)* cannot produce an infinite sequence of dwindling increments `t` all with the same nonzero sign, though some inputs `x` can produce a long such or else alternating sequence. The occasional long sequences caused increases in the numbers `k` of traversals of the uncompensated `while` loop for some reruns with redirected roundings. No rerun had to run long enough for `t` to underflow to zero. Reducing precision reduced `k` too.

. . . . . .

What would Delta Debugging do about statements like "`while any(y ~= oldy)`" ?

If the declared precision of one of `y` and `oldy` were reduced, but not the other, the `while` loop could run forever. Would a naive user of Delta Debugging be able to debug it?

What would Delta Debugging do about the parameter `eta` ?

Reducing a *variable*'s precision also reduces its range. Without an appropriate change to `eta`, it would cause an overflow to ∞ followed by an *Invalid Operation* ∞/∞ which would either turn into a *NaN* or abort computation. Would a naive user of Delta Debugging be able to debug it?

(Reducing *arithmetic*'s precision on the hardware *etc*. employed above did *not* reduce its range. Other hardware and other compilers are likely to behave differently.)

Error-Analysis of $L(N) := \sum_{0 \le i < N} \sqrt{\left((x_{i+1} - x_i)^2 + (g(x_{i+1}) - g(x_i))^2 \right)}$

The rounding errors in $x_i := i \cdot h = i\pi/N$ seem negligible in so far as they merely alter slightly the end-points of the chords without taking them off the curve $y = g(x)$ or breaking it. The two differences $(x_{i+1} - x_i)$ and $(g(x_{i+1}) - g(x_i))$ incur no new rounding errors because they mostly cancel when N is big. The squares, sum and square root suffer rounding errors confined to their last digits; they are negligible. The N rounding errors incurred by the summation process $\sum\ldots$ do not accumulate because they too are rendered negligible by Compensated Summation.

The errors δg inherited in computed values of $g(x_i)$ contribute almost all the uncertainty in the computed $L(N)$; this uncertainty grows much faster than $L(N)$ as N grows, as we shall see.

Abbreviate a chord's length to $\sqrt{\left((\Delta x)^2 + (\Delta g)^2 \right)}$ using $\Delta x := x_{i+1} - x_i$ and $\Delta g := g(x_{i+1}) - g(x_i)$ to get rid of distracting subscripts. The computed length is $\sqrt{\left((\Delta x)^2 + (\Delta g + \delta g)^2 \right)}$ in which δg is the difference between errors δg inherited with the computed values of $g(x_{i+1})$ and $g(x_i)$, and overestimated near the bottom of p. 6. The error in the chord's computed length is therefore

$$\sqrt{\left((\Delta x)^2 + (\Delta g + \delta g)^2 \right)} - \sqrt{\left((\Delta x)^2 + (\Delta g)^2 \right)} \approx \delta g \cdot \Delta g / \sqrt{\left((\Delta x)^2 + (\Delta g)^2 \right)}$$

after terms of the order of $(\delta g)^2$ are ignored. This error's magnitude cannot exceed $|\delta g|$ and is rarely very much tinier since $|\Delta g| >> |\Delta x|$ usually. Therefore the error accumulated in $L(N)$ is most unlikely to exceed $\sum_i |\delta g|$. This is too pessimistic; it ignores probabilities.

A probabilistic over-estimate can replace this overly pessimistic error bound if δg is treated as a bounded random variate with mean zero, unbiased as are individual rounding errors incurred by IEEE Standard 754's default *round-to-nearest (even)*. Then a more realistic overestimate of the error accumulated in $L(N)$ is the standard deviation of $\sum_i \pm\delta g$, which cannot exceed roughly

$$\delta L(N) := (2\pi/9) \cdot \sqrt{N} \cdot \varepsilon \cdot |\log_2(\varepsilon)|^{3/2} .$$

Experiments have not yet corroborated this overestimate $\delta L(N)$ of the likely error in $L(N)$.

Tabulated below are results from two FORTRAN programs to compute $L(N)_P$ carrying $P = 24$ sig.bits and $P = 53$. For $L(N)_{24}$ the appropriate $\varepsilon \approx 1/2^{24} \approx 6e{-}8$.

N	$L(N)_{24}$	$L(N)_{24} - L(N)_{53}$	$\delta L(N)_{24}$
177147	8.4622517	7.63639e-06	2.05921e-03
295245	8.5969534	-9.87013e-06	2.65843e-03
531441	8.7548380	1.43595e-04	3.56666e-03
885735	8.8992472	-2.93253e-04	4.60453e-03
1594323	9.0540075	-1.49450e-03	6.17763e-03
2657205	9.1793098	-1.87670e-03	7.97529e-03
4782969	9.3223581	-3.68765e-03	1.06999e-02
7971615	9.4424019	-1.12409e-02	1.38136e-02
14348907	9.5746126	-2.90817e-02	1.85329e-02

In the table, the difference $|L(N)_{24} - L(N)_{53}|$ grows faster than the last column's error-over-estimate $\delta L(N)$. That difference is not just the error in $L(N)_{24}$, but includes something caused by differences between corresponding arguments x_i in the two programs. They differ, though obtained from the same formula "$x_i := i \cdot h$", because they are rounded to different precisions.

For inputs N much bigger than in the table, some adjacent arguments x_i collide when rounded down to 24 sig.bits. Arguments x_i that almost collide cause more trouble; their differences Δg suffer severe contamination from roundoff's δg. That contamination increases the uncertainty $\delta L(N)$ in computed values $L(N)$ until these become entirely uncertain for all bigger N. Thus does the arithmetic's precision preclude inputs N bigger than $2/\varepsilon$, thereby cramping both the corroboration of error-over-estimates and attempts to exploit faster arithmetic of lower precision to investigate how $L(N)$ behaves as $N \to \infty$.

. . . . . .

What should Delta Debugging do about a program's input ?

If the declared precisions of a program's internal variables are altered, how should changes in precisions and ranges of inputs and accuracies of outputs be correlated with the internal changes?

Apt correlations can be unobvious even to numerical experts. A program can (by design) deliver impeccably accurate results despite their seeming hypersensitivity to input data perturbations so small as to be deemed negligible by someone unaware of physical, geometrical or other structural correlations in the data. Would a naive user of Delta Debugging be able to debug it?

I think Delta Debugging ought to include educational components conveyed through questions its numerically inexperienced users are prompted to answer. If appropriate, some questions, like

How accurate a result is accurate enough?
To what constraints must valid input conform?
What sets of input data probe potential hazards adequately?
What options (loop-counts, precisions, libraries, …) should be explored?

may be difficult, sometimes even for experts. Leaving these questions unexposed helps nobody.