A Brief Tutorial on Gradual Underflow

Prepared for ARITH 17, Tues. 28 June 2005,
and subsequently augmented

Contents: Page

§0: What are Gradual and Abrupt Underflow ?

An attempt to generate a nonzero floating–point number that is too tiny to represent in the usual way precipitates *Underflow*.

Representable floating–point numbers:

Given the specified format's integers . . .

Radix: ß = two (Binary) or ß = ten (Decimal)
Precision: P = Number of "Significant Digits" carried
Exponent Range: [–È, +É]

each finite floating–point number x is represented by its two integers …

Signed "Significand" or "Coefficient" m within $|m| < B^P$
Unbiased Exponent e in the range $–È \le e \le É$

thus: $x = m \cdot ß^{e+1–P}$.

If there are no other constraints upon x , Underflow is *Gradual*.

Representable $x = m\cdot ß^{e+1-P}$ where $|m| < ß^{P}$ and $-È \le e \le É$.

If there are no other constraints upon x , Underflow is *Gradual*.

Gradual Underflow (GU) has the simplest mathematical model.

I.B. Goldberg mentioned GU in *Comm.ACM* (1967). W. Kahan had put GU onto an IBM 7094 and into SHARE by 1965.

If $|m| < ß^{P-1}$ and $-È < e$ then $x = m\cdot ß^{e+1-P} = (ßm)\cdot ß^{e-P}$ too. To represent each representable number x uniquely, its exponent e is minimized. Then if its $|m| \ge ß^{P-1}$ we call its representation "Normalized".

Whether $0 = 0\cdot ß^{È+1-P}$ is to be called "Normalized" too is not decidable mathematically.

A *Subnormal* $x = m\cdot ß^{È+1-P}$ has $0 < |m| < ß^{P-1}$. Such an x has no *normalized* representation.

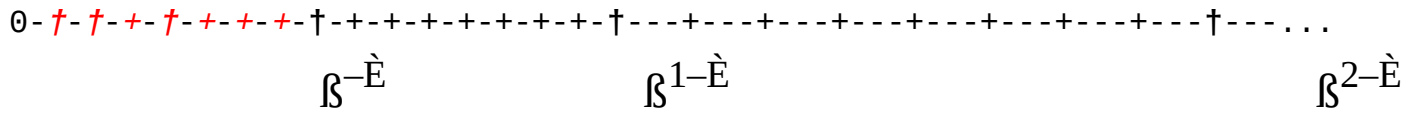
Abrupt Underflow (AU) denies representation to subnormals.

Before 1985 almost all computers underflowed abruptly.

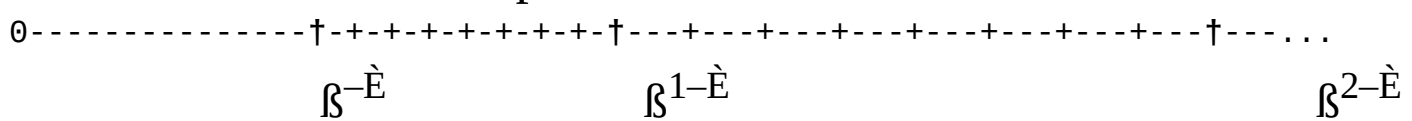
Pictures of Gradual and Abrupt Underflow

For example, take Binary ß := 2 and Precision P := 4 sig. bits. Below, "+" stands for a representable positive floating-point number. Powers of ß are marked "†". The positive subnormal numbers are *red* ; AU has none of them.

When Underflow is Gradual:



When Underflow is Abrupt:



Unlike GU, AU lacks the crucial property that the gaps between a floating-point number x and its nearest neighbors not increase when |x| decreases. Though not always a bad thing, this lack undermines some error-analyses and proofs.

§1: Why IEEE 754's Default is Gradual Underflow

Underflow is destined almost always to be ignored, rightly or wrongly, no matter what tiny value, zero or something else, is presubstituted for the event.

Like some migratory birds, Underflows are relatively rare, though they may come in flocks. Their default presubstitution has to be a compromise best for the vast majority of underflows that can then be ignored justifiably. GU has been found best by enough to elect it to be the default. Still, AU has its uses too.

Other proposals, like an **UN** symbol for underflow, turn out to make matters worse.

If underflows occur frequently in a program, they may be a symptom of poor program design, or a consequence of a well-considered choice. If the former, the program should be revised to preclude underflow. If the latter, a non-default presubstitution like AU→0 may be better than GU. The programmer bears the responsibility of choosing wisely. Underflow's speed will influence his decision.

Let's not complicate his choice unnecessarily by implementing GU too much slower than AU.

See §2.

Advantages of Gradual over Abrupt Underflow to Zero

Gaps between a floating-point number x and its nearest neighbors do not increase when |x| decreases, affecting error-analyses and monotonicity.

GU's $x \pm y$ cannot underflow since a subnormal difference is exact.
This figures in schemes that enhance accuracy by compensating for rounding errors.

Finite " x = y " is the same as " x–y = 0 " with GU, but not AU.

Scalar product $s := \sum_j a_j \cdot x_j$ is not degraded by GU much worse than by roundoff provided at least one product $a_j \cdot x_j$ is normal nonzero. Likewise matrix products. Behavior with AU→0 is more complicated. *Cf.* J.W. Demmel in *SIAM J. Sci. Stat. Comp.* **5** (1984)

Polynomial $p(x) := a_0 + (x - x_1)\cdot(a_1 + (x-x_2)\cdot(a_2 + (x-x_3)\cdot(\ldots a_n)))$ is not degraded by GU much worse than by roundoff provided a_0 is normal nonzero. Behavior with AU→0 is more complicated.
Cf. S. Linnainmaa in *ACM SIGNUM Newsletter* **16** (1981)

But GU seems to spoil some programs. More about them later in §3.

§2: Implementations

GU and $x \pm y$ barely affect each other. AU$\rightarrow$0 can flush $x \pm y$.

Ideally, GU affects $x \cdot y$ and x/y as if . . .

GU of $x \cdot y$ or x/y must be followed by Denormalization, which acts like an add–unnormalized of zero to produce a properly rounded subnormal result.

See C. Lee in *IEEE Trans. on Computers* **28** (1989).

$x \cdot y$ of two subnormals or of very tiny factors underflows to 0 and needs no special treatment. Otherwise $x \cdot y$ entails prenormalization of a subnormal number, which acts like a postnormalized add of zero with an extended exponent.

x/y of a subnormal or two behaves similarly but may require one or two prenormalizations. Division is slow anyway.

Originally GU was intended for *Tagged* floating-point registers with slightly extended exponent fields. A tag tells whether a register …

- is ready to add/subt./store (perhaps Subnormal), &
- is ready for mult., div., sqrt. ((pre)normalized), &
- has to be denormalized, plus a rounded-up bit or two, &
- is zero, infinity or NaN if tagging them handles them faster.

A floating-point register's tag is set whenever the register …

Loads from memory (ready to store, maybe not to mult./div. etc.)
Receives a result (maybe subnormal, maybe prenormalized, …)

Mult./Div./Sqrt of an unready register may need prenormalization first.
Add/Subt./Store of an unready register may need denormalization first.
Handle an unready register like a cache miss; no dependency changes.

Store operations may be tricky when the architecture expects every store to go without delay to a buffer where cache alterations are queued.

Encourage compilers to schedule test-tag-and-de/prenormalize in advance.

Simpler schemes work well if Fused Multiply-Add hardware is available.

IMPORTANT IMPLEMENTATION DETAIL:

If de/prenormalization is handled perhaps slowly by trapping, …

DO NOT TRAP UNNECESSARILY

when the result of the operation will predictably underflow to zero.

Untrapped exponents below –(È+P) in a non-outward rounding mode result in zeros.

In most programs, if an underflow occurs it is most likely to go to zero no matter whether it goes gradually or abruptly. Then, if the Underflow Flag has already been raised, a trap merely wastes time unless the program(mer) has requested a trap to aid debugging or to *Break* out of a long loop.

Still, try not to vitiate the many valuable programs that generate a nonzero subnormal number with almost every gradual underflow.

All Floating-Point Traps can be Lightweight Traps

They do not have to save a complete panel as do operating system traps.

Unless the programming language allows a trap to *Break* out of a loop or block to a preset location, thus leaving unpredictable all variables abandoned in that loop or block, …

Floating-point traps change no previously determined dependency, and resume execution immediately after the exceptional operation. They do not have to allow the pipeline to empty, but may do so.

A floating-point trap-handler's access to a program's variables is restricted to …

- The trap-handler's own internal variables preset when the trap was enabled.
- The location where program execution will resume after the trap.
- A status register or location that holds floating-point flags temporarily.
- The nature (+, –, *, /, √, …) of the exceptional operation and its …
  - Operands' values, not their addresses.
  - Destination register and perhaps an incompletely processed result therein.

IEEE 754 does not mandate traps. They are for implementation of floating-point and for the support of a short menu of exception-handling options from which an applications programmer can choose without ever writing his own handler.

See a Demonstration of Presubstitution … `<http://www.cs.berkeley.edu/~wkahan/Grail.pdf>`.

A Short Menu of Exception-Handling Options

Of the three kinds of options, only the third does not resume execution right after the exceptional operation, having perhaps altered its tentative result.

- **Pause:** Stop to debug an exceptional operation, displaying its circumstances, and allow users to choose to resume execution as if it had not stopped.
  This option may be unavailable or imprecise on systems that exploit concurrency heavily. Operations "before" the exceptional operation may not have completed, operations "after" it may have begun, with obscure correlations to source-text. The act of Pausing may alter timing relations and hide a bug perhaps in the compiler.

- **Presubstitute:** Choose in advance a value to substitute for the exceptional operation's result, perhaps after copying its sign or performing some simple procedure like denormalization. IEEE 754's defaults are presubstitutions.
  Another simple procedure "wraps" an ov/underflowed result's exponent and inc/decrements a counter correspondingly to help rescale lengthy products/quotients of sums/differences. One application is to determinants like det(A(x)) in §3.

- **Abort:** Choose in advance a kind of exception and a location to which to *Break* if that exception occurs within a designated loop or block. The break might not jump out exactly *when* the exception occurs though it must jump before the loop or block finishes, leaving its variables unpredictable.

The menu of exception-handling options challenges programming languages.

A programmer's choice of exception-handling *Mode* from that menu has to be recorded in a *Mode Variable*. What scope, visibility and inheritance rules govern mode and flag variables? Different languages favor different rules.

So far, such variables have been treated as *global*, held in bits in a hardware's control and status words, though IEEE 754 conveys no such prescription.
Apple's old SANE (Standard Apple Numerics Environment) for 680x0-based Macs provided procedures to sense, set, save and restore mode variables; see *Apple Numerics Manual 2d Ed.* (1988) and its successor *Power PC Numerics*, *Inside Macintosh* (1994), both published by Addison-Wesley. ***C***99 and Fortran 2003 provide analogous procedures. Microsoft's ***C*** math library's functions manipulate the bits in the Intel x87 control and staus words. None of these suffice to support a menu of options like the aforementioned. Sun Microsystems' ***C*** offers a complicated trap-handling interface through which such a menu could be programmed, but not easily. All these modes would be global.

A better way to *localize* modes imitates the language ***APL***'s treatment of its *system variables* []*CT* (comparison tolerance), []*IO* (index origin), *etc*. More generally, every subprogram could include a *signature* that tells which modes and flags are inherited and/or propagated implicitly so they need not appear explicitly as arguments in every invocation. Some functions intended to appear *atomic* like +, -, *, /, √, log, exp, … have to inherit, save, set, hide, restore and merge flags and/or modes, all of which is tedious without appropriate hardware and language support. Most people, sanely, would rather not think about these things. Someone has to think about them. They weigh upon the minds of a committee currently considering revisions to IEEE 754 (1985) that will bind better than global variables do to programming languages without burdening applications programmers unnecessarily.

§3: (Gradual) Underflow Avoidance in Applications

Most *but not all* underflows can be presubstituted to any sufficiently tiny numbers without ill effect because these will later be added to big numbers and rounded away. If yours are *provably* this way, choose whichever of GU or AU runs faster on your machine. Unfortunately,

AU runs a lot faster on some machines. On such a machine …

DON'T CHOOSE AU JUST BECAUSE GU RUNS SLOWER.

Making AU the default and ignoring all underflows can be **dangerous**, especially in single-precision (`float`) arithmetic's narrow range. For AU's higher speed to matter, vastly many underflows must occur. *All* of them must be *proved* negligible. Otherwise sums of numbers slightly above underflow threshold $ß^{-\bar{E}}$ plus others slightly below but flushed to 0 will err unobviously and corrupt subsequent mult./div./sqrt/log/… .

Likewise for `double` for reasons of methodological homogeneity: A program tested on `floats` and then promoted to `doubles` is expected to behave the same except for wider range and enhanced accuracy.

No matter which of GU or AU be chosen, clusters of too many underflows are most often a symptom of a programming mistake. …

Examples of Programming Mistakes:

Diffusion Problems: Initialize interior and boundary values to zero, though any tiny nonzero number like 10^{-30} is no worse and would preclude underflow while interior values grow gradually.

Stopping Iterations: Stop when a residual becomes zero perhaps by underflow, though any residual smaller than a roundoff-related threshold would stop sooner, with no loss of accuracy, saving at least one iteration to compensate for the cost of a threshold comparison.

Matrix Reductions: Stop an elimination process, designed to annihilate nonzero elements of a matrix, just when no more nonzero candidates exist, though eliminating nonzero elements smaller than a roundoff-related threshold wastes time. Determining a suitable threshold may require forethought difficult in some cases but worth the effort if it saves time.

Root-Finding: Solving an equation $f(z) = 0$ for a root z whose location is smeared by ignored underflow. This mistake occurs often when $f(x) = \det(A(x))$ for a matrix $A(x)$ of large dimension depending nonlinearly upon x. Tricky scaling may be required to prevent underflows from distracting a root-finder that depends upon f only through ratios like $f'(x)^{-1} \cdot f(x)$ and $f(x_1)/f(x_2)$.

Benchmarks: Including among benchmarks a program that underflows often, perhaps because of a mistake like one of the foregoing, unless the program's purpose is to reveal the slowness of underflow handling. This mistake compels competing compiler vendors to make Abrupt Underflow their default, which is then imposed *willy-nilly* upon programs imported from sources that took full conformity with IEEE 754's Gradual Underflow for granted. Then who should be held responsible if such a program malfunctions for that reason or some other when diagnosis is difficult ?

CAVEAT: THIS DOCUMENT IS A WORK IN PROGRESS CONTINUALLY SUBJECT TO CHANGES IN RESPONSE TO CONSTRUCTIVE SUGGESTIONS FROM READERS.