What is Gradual Underflow?

W. Kahan May 1980

It is a scheme which reduces the effect of underflow to something comparable with the uncertainty due to roundoff in a wide range of numerical computations including most of those with matrices, quadrature, ordinary differential equations, zero-finding, convergence acceleration, ... To understand gradual underflow we have to understand a little about formats used for floating point arithmetic.

Normalized floating point numbers look like

$\pm(D_0.d_1d_2\ldots d_n) \times B^e$ e.g. $\pm(3.14159) \times 10^5$

where the radix B can be 2(binary), 8(octal), 10(decimal) or 16(hexadecimal) on diverse American computers; then the n+1 "significant digits" D_0, d_1, $d_2\ldots d_n$ are each drawn from the set {0,1,2,...,B-1} except that $D_0 \neq 0$; and the exponent e is an integer confined to some interval $\check{e} \leqslant e \leqslant \hat{e}$; e.g. hand-held calculators use B = 10 and $\check{e} = -99 \leqslant e \leqslant \hat{e} = 99$. The foregoing format excludes zero unless a special case is set aside for $0 = \pm(0.00\cdots0) \times (B^{\check{e}}$ or $B^0)$; whether zero is called normalized or not (because its $D_0 = 0$) does not matter. Unnormalized floating point numbers, with $D_0 = 0$ but $e > \check{e}$, are usually outlawed as redundant in so far as their values can be represented equally well by normalized numbers (but the B5500 is one computer which treats unnormalized and normalized numbers indistinguishably). Denormalized numbers are characterized by $D_0 = 0$ and $e = \check{e}$; for instance on a 6 sig. dec. calculator

$(0.03142) \times 10^{-99}$

is the best convenient approximation for $\pi/10^{101}$. Note that denormalized numbers look unnormalized at first until you notice that the exponent $e = \check{e}$ is minimal and realize that no denormalized number can be represented by a normalized number in the same format whereas an unnormalized number can be supplanted exactly by another, normalized or denormalized, with a lesser exponent e.

But most computers have outlawed denormalized numbers too. Those computers "flush" any attempt to compute a floating point number whose normalized form would otherwise underflow (have $e < \check{e}$); for instance, most hand-held calculators will neither allow $(\pi/10^{50})/10^{51}$ to be represented as 3.14159×10^{-101} nor denormalize it to 0.03142×10^{-99} but will instead display zero or 1×10^{-99} or "Error". Only computers that underflow gradually possess denormalized numbers. Among such computers are the Electrologica X-8, Intel 8086-8088 with 8087, Motorola 6809 with 6839 (and IBM 7094, IBM 360-370, Burroughs B5500, DEC 20/PDP 10 with appropriate trap-handling software not currently distributed by their manufacturers).

How does gradual underflow help? Its most obvious effect is to preserve the following relation: if $0 < x < y$ then $x/y < 1$ and $y-x > 0$. The last inequality can be falsified by computers that flush underflows to zero, but not by those that underflow gradually. The effect upon program logic, where a test of one relation is presumed to imply others, is immediately clear, especially when we expect "$y-x = 0$" to imply "$f(y) - f(x) = 0$" unless f is a pathological function (involving, say, division by zero). But the effect goes much deeper. Consider two nearby representable numbers x and y and their difference $z = x - y$, and suppose x and y are so close to each other that $|z| \leqslant |x|$ and $|z| \leqslant |y|$. Then z must be representable exactly despite roundoff and, if underflow is gentle, despite underflow. But just as poorly designed arithmetic units can contaminate small differences unnecessarily with roundoff (as do CDC 6000 class computers and most TI calculators), so will flushing underflows contaminate them. This contamination is a noise that may defeat the feedback loop intended to stabilize a computation. For instance, to solve the equation $f(s) = t$ for s given f and t, we are obliged to calculate the discrepancy $t-f(s)$ and feed it back to alter s, as in Newton's method:

$$\text{new } s = s + (t - f(s))/f'(s).$$

The noise in $t-f(s)$, whether caused by roundoff or underflow, is the principal limitation upon the accuracy to which a solution s can be calculated. Gradual underflow makes much less noise here than does flushing.

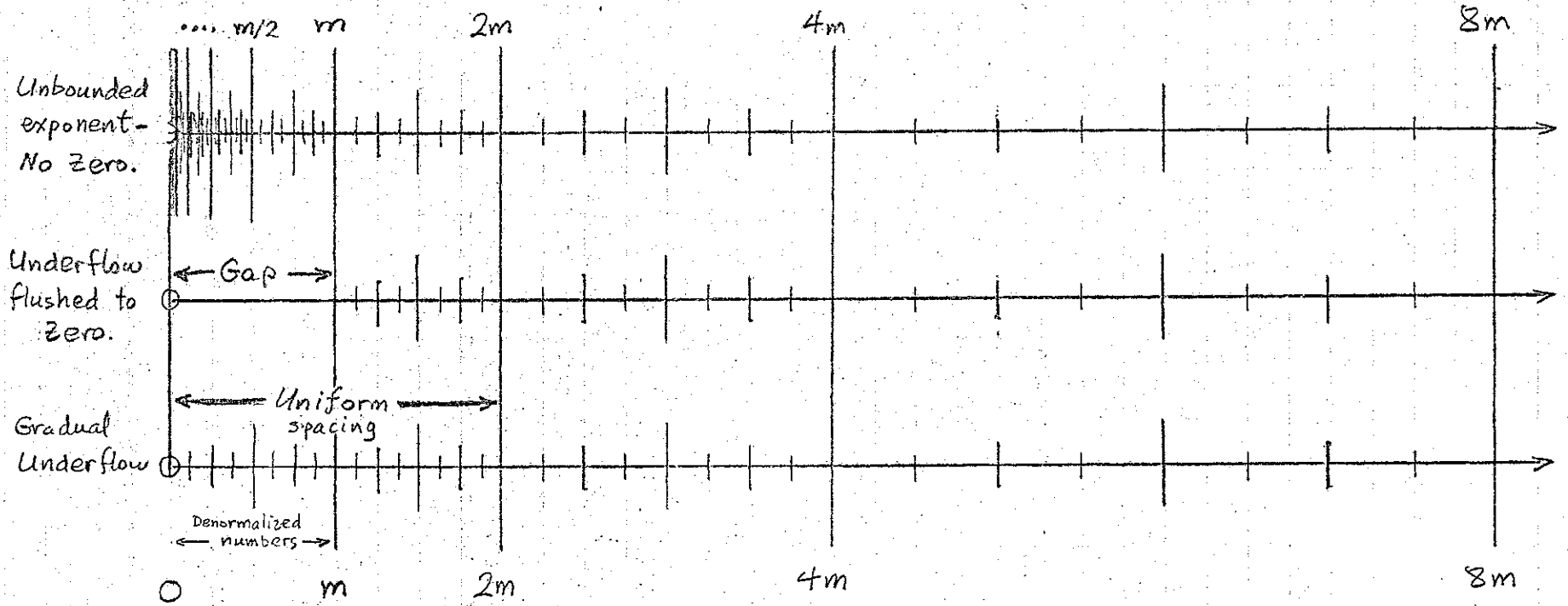
Gradual underflow reduces every instance of underflow to an amount absolutely no bigger than a rounding error in the last significant digit of

the smallest normalized number. This means very often that a program which is proved correct in the face of roundoff can easily be proved correct despite underflow too provided underflow is gradual, whereas when underflow is flushed that program may have to be augmented by tests against format-dependent thresholds to provide a defense against flushing. This is why gradual underflow is valuable for matrix multiplication and inversion, operations so common that their enhancement by gradual underflow is enough to justify providing that feature. Moreover, subroutines programmed conscientiously can often be designed more easily, thanks to gradual underflow, to cope with their own underflows automatically; consequently the naive users of such subroutines can expect to see underflow messages significantly less often on machines that underflow gradually than on machines that flush.

Some kinds of underflow cannot be cured by gradualness, nor by flushing. For intance, long chains of multiplications and divisions unrelieved by alternate additions generally require scaling to defend against over/underflow unless they can be calculated with the aid of a (possibly temporary) exponent field extension (see SHARE SSD |59 item C 4537, Dec. 1966). So gradual underflow is no panacea. All that can be claimed is that it improves a large and recognizable class of programs without making others worse, without complicating our concept of representable numbers, and without much of a penalty in hardware or speed except possibly on pipelined parallel array-oriented machines where a different approach (the KCS "Extended" format described by J. Coonen in "Computer", Jan. 1980) is more appropriate for reasons having little to do with underflow.

Before you rush out to implement gradual underflow on your computer be sure that it has or can be given an Underflow flag. The best way to test whether an expression has underflowed, when underflow is gradual, is to test the flag rather than to test whether the expression is zero or denormalized; see Coonen's article.

Tiny Binary Floating Point Numbers.



Each vertical tic —|— stands for a 4-sig. bit binary floating point number. The underflow threshold **m** is a power of 1/2 depending upon the allowed range of exponents; every floating point number bigger than **m**, but none smaller, is representable as a normalized floating point number. On some machines (IBM 7094, DEC PDP-10, DEC PDP-11, ...) **m** is a normalized number too, on others (HP-3000) not. Flushing underflows to zero introduces a gap between **m** and 0 much wider than between **m** and the next larger number. Gradual Underflow fills that gap with Denormalized numbers as densely packed between **m** and 0 as are normalized numbers between **m** and 2m. Doing so relegates underflow in most computations to a status comparable with roundoff among the normalized numbers.

W. Kahan May, 1980