Computing a Real Cube Root

W. Kahan

Nov. 18, 1991
Retypeset by David Bindel, Apr. 21, 2001

A formula like $\sqrt[3]{y} = \exp(\ln(y)/3)$ may come to mind at first, but second thoughts raise questions and then doubts about it:

- Is that formula economical? Can a cube root be computed at less cost than the transcendental functions $\ln(\ldots)$ and $\exp(\ldots)$?
- Is that formula correct? Instead of $\sqrt[3]{-8} = -2$, it yields a complex $\exp(\ln(-8)/3) = 1 + \sqrt{3} = (-8)^{1/3}$ when $y = -8$.
- Is that formula accurate when it is correct? In two of the three examples below, rounding each operation correctly to 10 sig. dec. yields a result accurate to only 8 sig. dec.:

y	:	$8.0_{10} - 99$	8.0	$8.0_{10}99$
$\ln(\ldots)$	:	-225.8764827	2.079441542	230.0353657
$(\ldots)/3$	:	-75.2921609	0.6931471807	76.67845523
$\exp(\ldots)$	:	$1.999999976_{10} - 33$	2.000000000	$1.999999961_{10}33$

These examples suggest, and error-analysis confirms, that though the formula $\exp(\ln(y)/3)$ is accurate for arguments y near 1, it loses accuracy to roundoff when y is extremely tiny or huge.

To achieve better accuracy and, on many computers, speed, the computation of cube roots can be based upon an iterative approach to the solution of the cubic equation $x^3 = y$ for $x = \sqrt[3]{y}$. A programmer inclined this way must first resolve two questions:

What iteration formula(s) will converge fastest?
What is the best first guess to start the iteration?

Several iterations are known to replace a good approximation x to $\sqrt[3]{y}$ by a much better approximation X. Every such iteration can be derived from Newton's iteration $X := x - f(x)/f'(x)$ by choosing a suitable expression $f(x)$ that vanishes when $x^3 = y$. Five such iterations follow, with their orders of convergence M and asymptotic constants Γ indicated; in each case

$$\frac{X/\sqrt[3]{y} - 1}{(x/\sqrt[3]{y} - 1)^M} \to \Gamma \text{ as } x \to \sqrt[3]{y}.$$

$X := x - (x^3 - y)/(3x^2)$,	$M = 2$, $\Gamma = 1$	(3 * 1 ÷ 2−)
$X := x - (x^3 - y)x/(2x^3 + y)$,	$M = 3$, $\Gamma = 2/3$	(3 * 1 ÷ 4±)
$X := x - x((\frac{14}{81}s - \frac{2}{9})s + \frac{1}{3})s$ at $s := \frac{x^3 - y}{y}$,	$M = 4$, $\Gamma = -35/3$	(6 * 1 ÷ 4±)
$X := x - \frac{(yz)x}{3(y-\lambda z)(y+\lambda z)+2(yz)}$ at $z := x^3 - y, \lambda := \sqrt{2/27}$	$M = 4$, $\Gamma = -1$	(6 * 1 ÷ 8±)
$X := x - \frac{2(x^3-y)(x^3+y)x}{3(x^3+y)^2+(x^3-y)(x^3+y)-(8/9)(x^3-y)^2}$,	$M = 5$, $\Gamma = -7/18$	(8 * 1 ÷ 6±)

Each formula is intended to be *iterated* (used repeatedly) with $x := X$ until it approaches $\sqrt[3]{y}$ as accurately as desired.

Taking arithmetic work into account, the second formula ($M = 3$) converges fastest. However, the last iteration should use one of the last formulas ($M = 4$ or 5) because they can yield the most accurate final result X. The penultimate x must be accurate to almost, and *rounded* to at most, a third as many figures as the arithmetic carries. Doing so ensures that x^3 and then $x^3 - y$ will be computed with no rounding error, whereupon the final X will differ from $\sqrt[3]{y}$ by scarcely more than one rounding error.

None of the five iteration formulas above runs fast on a machine whose floating-point division takes too much longer than its other arithmetic operations. Among such machines are CRAYs, Intel i860's, IBM RS/6000's, ... A faster way for these machines to compute $\sqrt[3]{y}$ is to obtain an estimate z for $1/\sqrt[3]{y}$ first, and then compute $x := z^2 y$ as an estimate for $\sqrt[3]{y}$. The following iteration replaces any good approximation z by a far better one: $Z := z + (1/3)(z - z^4 y)$. This iteration converges quadratically; $(Z^3\sqrt{y} - 1)/(z^3\sqrt{y} - 1)^2 \to -2$ as $z \to 1/\sqrt[3]{y}$. (4 * 2±)

Iteration requires a starting value, but before discussing how to find one let us justify assuming that $0 < y < \infty$. In case $y < 0$ we should compute $\sqrt[3]{y} := -(\sqrt[3]{-y})$; in case $y = \pm 0$ or $\pm\infty$ we should compute $\sqrt[3]{y} := y$. If y is so close to over/underflow thresholds that computed quantities like x^3 or $x^3 \pm y$ or z^4 may over/underflow, then we should compute $\sqrt[3]{y} := \sqrt[3]{\sigma^{-3} y \sigma}$ for a suitable *scale factor* σ, chosen to be a power of the arithmetic's radix ρ so that $\sigma^{-3} y$ can be computed without any rounding error. (The *radix* is $\rho = 10$ for decimal calculators, $\rho = 2$ for most computers' binary floating-point arithmetics, and $\rho = 16$ for hexadecimal floating-point used by IBM's /370 architecture and its many imitations (Amdahl, Fujitsu, ...).)

Ideally, the iterations above should start with an estimate x or z that costs as little time as possible to compute and yet approximates $\sqrt[3]{y}$ or $1/\sqrt[3]{y}$ as accurately as possible. Speed and accuracy are limited in principle only by the amount of memory available for a table of precomputed estimates. Such a table need provide estimates only over a range $1 \le y \le \rho^3$ into which y can be transferred by scaling. With linear interpolation into a large table with $N + 1$ entries, the starting estimate's relative error can be kept below about $((\rho - 1)/N)^2/8$. Apparently radices $\rho > 2$ require rather larger tables than does binary ($\rho = 2$) floating-point arithmetic. Indeed, binary can do surprisingly well with just one table entry, as will now be demonstrated.

What follows works for DEC VAXs and for machines that conform to IEEE Standard 754 for binary floating-point arithmetic; in other words, it works for almost all commercially significant computers except CRAYs and IBM /370s and their clones. Let us consider positive floating-point numbers $y = 2^K(1 + F)$ where the *exponent* K is an integer (it can be negative) and F is a nonnegative binary *fraction*; $0 \le F < 1$. As stored in the computer, the bit string that represents the floating-point number y can also be interpreted as a positive fixed-point number $Y = B + K + F$ whose binary point lies between the integer $B + K$ and the fraction F. B here is an *exponent bias* peculiar to the computer's floating-point format, as indicated in the Table below.

Computer	Floating-pt. Format	Width (bits)	Bias B
DEC VAX	Single F	32 (24 sig.)	129
"	Double D	64 56	129
"	Double G	64 53	1025
"	Extended H	128 113	16385
IEEE 754	Single precision	32 24	127
"	Double precision	64 53	1023
"	Double-Extended	80 64	16383
	(Quadruple)	128 113	16383

From B, derive a fixed-point constant $C := (B - 0.1009678)/3$ with its binary point in the same place as Y's. This figures in a formula to compute a quick approximation q to $\sqrt[3]{y}$; compute $Q := C + Y/3$ in fixed-point and re-interpret it as a floating- point number q. Its relative error will fall below 3.2%. A similar technique supplies a quick approximation r to $1/\sqrt[3]{y}$. The constant is $G := (4B - 0.1984887)/3$. Use it to

compute a fixed-point number $R := G - Y/3$, and then re-interpret that as a floating-point number r with relative error under 3.5%.

These first approximations require a programming environment in which any given bit string can be intepreted as either a fixed-point number or a floating-point number, possibly after some permutation of bytes or words; programs written that way for one machine can hardly ever be used on another. Another approach uses functions like the IEEE standard's LogB and ScalB, or C's *frexp* and *ldexp*, to decompose $y = 2^K(1 + F)$ into its exponent and fraction, and recompose q or r from Q or R; this approach may well work on many computers, but not so fast.

The foregoing information should suffice for a program to compute cube roots on any computer to very near the full precision of its hardware. The programmer must take account of any special cases (like ∞) pertinent to his machine, construct an appropriate first approximation, choose which iteration(s) to use and how many, and perform scaling when necessary. For example, to serve the IEEE Double precision and VAX G formats, a program that follows the 3.2% approximation by an iteration with $M = 3$, and then an iteration with $M = 4$, produces a result in error by less than 0.59 units in its last place at a cost (in the usual unscaled case) of about 2 divides, 10 multiplies, and a similar number of add-subtracts.

Perhaps the last problem is the hardest: choosing the program's name. Ideally it should need no explanation, but a limitation upon its length may preclude that. Although "CBRT" has seen use, I prefer "QBRT" in order that the prefix "C" may be reserved for use with complex-valued functions.