Computing the Cube Root

Ken Turkowski, turk@apple.com

Apple Computer Technical Report #KT-32
10 February 1998

There are occasions in perception and computer graphics where it is necessary to compute the cube root.

\[ r = \sqrt[3]{s} \]

The first thing to note is that

\[ \sqrt[3]{s^{2^{3p}}} = \sqrt[3]{2^p}. \]

From this, we see that the cube root of any number is related in a simple way to that of a number \( 8^p \) times larger. In other words, that computing the cube root of any number can be reduced to computing the cube root of a number between \( \frac{1}{8} \leq s < 1 \).

The following quadratic polynomial yields approximately 6 bits of accuracy between \( \frac{1}{8} \leq s < 1 \):

\[ r = -0.46946116s^2 + 1.072302s + 0.3812513 \]

Given an estimate for the cube root of a number, the accuracy can be improved quadratically by use of the Newton-Raphson-derived iteration

\[ r_{n+1} = \frac{2}{3} r_n + \frac{1}{3} \frac{s}{r_n^2}, \]

One subsequent iteration yields \( \approx 12 \) bits, 2 iterations yield \( \approx 24 \) bits, 3 iterations yield \( \approx 48 \) bits, 4 iterations yield \( \approx 96 \) bits, so that 2 iterations is sufficient for single precision, and 4 is sufficient for double precision IEEE floating point.

An analysis yields 4 M (multiplication/addition/subtraction) operations and 1 D (division) operation per iteration. For single precision floating point, the total operation count is

\[ 4 \text{ M} + 2 \times 4 \text{ M} + 2 \times 1 \text{ D} = 12 \text{ M} + 2 \text{ D}. \]

An alternative is to just use one approximating rational polynomial. An analysis shows that a quartic rational polynomial is sufficient to yield 24 bits of precision between \( \frac{1}{8} \leq s < 1 \). This requires

\[ 16 \text{ M} + 1 \text{ D}. \]

On most modern computers, \( D > 4 \text{ M} \) (i.e. a division takes more than 4 multiplications worth of time), so it is probably preferable to compute a 24 bit result directly.
C Implementation

#include <fp.h> /* or <math.h> */

float CubeRoot(float x)
{
    float fr, r;
    int ex, shx;

    /* Argument reduction */
    fr = frexp(x, &ex); /* separate into mantissa and exponent */
    shx = ex % 3;
    if (shx > 0)
        shx -= 3; /* compute shx such that (ex - shx) is divisible by 3 */
    ex = (ex - shx) / 3; /* exponent of cube root */
    fr = ldexp(fr, shx); /* 0.125 <= fr < 1.0 */

    #ifdef ITERATE
    /* Compute seed with a quadratic qpproximation */
    fr = (-0.46946116F * fr + 1.072302F) * fr + 0.3812513F; /* 0.5<=fr<1 */
    r = ldexp(fr, ex); /* 6 bits of precision */
    /* Newton-Raphson iterations */
    r = (float)(2.0/3.0) * r + (float)(1.0/3.0) * x / (r * r); /* 12 bits */
    r = (float)(2.0/3.0) * r + (float)(1.0/3.0) * x / (r * r); /* 24 bits */
    #else ITERATE
    /* Use quartic rational polynomial with error < 2^(-24) */
    fr = ((((45.2548339756803022511987494 * fr +
        192.2798368355061050458134625) * fr +
        119.1654824285581628956914143) * fr +
        13.43250139086239872172837314) * fr +
        0.1636161226585754240958355063) /
            ((((14.80884093219134573786480845 * fr +
                151.971405104435648658557668) * fr +
                168.5254414101568283957668343) * fr +
                33.9905941350215598754191872) * fr +
                1.0);
    r = ldexp(fr, ex); /* 24 bits of precision */
    #endif

    return(r);
}