Computing the Inverse Square Root

Ken Turkowski
Media Technologies: Computer Graphics
Advanced Technology Group
Apple Computer, Inc.

Abstract: The inverse square root of a number is computed by determining an approximate value by table lookup and refining it through iteration.

3 October 1994

Apple Technical Report No. 95

Computing the Inverse Square Root

Ken Turkowski
3 October 1994

Introduction

In computer graphics calculations, the square root is often followed by a division, as when normalizing vectors:

$$\frac{\mathbf{v}}{\|\mathbf{v}\|} = \mathbf{v}\left(\sum_i v_i^2\right)^{-\frac{1}{2}},$$

This adds a significant amount of computational overhead, as a floating-point division typically costs much more than multiplication.

The cost of division may be mitigated by a *reciprocation*. This gem derives the method and provides an implementation for directly computing the *inverse* square root $f(x) = x^{-\frac{1}{2}}$.

Description of the Algorithm

The algorithm is based upon the method of successive approximations [Ralston78]. This formulation is noteworthy as no divisions are required. The square root may also be computed at the cost of one additional multiplication, as $\sqrt{x} = x \cdot f(x)$.

The algorithm has two parts: computing an initial estimate, and refining the root by using a fixed number of iterations.

Initialization

The initial estimate, or *seed*, is determined by a table lookup. The inverse square root of a floating-point number $m \cdot 2^e$ is given by

$$\left(m \cdot 2^e\right)^{-\frac{1}{2}} = m^{-\frac{1}{2}} \cdot 2^{-\frac{e}{2}}.$$

The exponent e is adjusted by negation and halving (or shifting if radix-2) to form the seed exponent. If the seed exponent $-\frac{e}{2}$ is to be an integer then e must be even. When e is odd, the next smaller even value is considered and the mantissa is doubled (that is, [1…4) becomes its domain of representation). The extended mantissa indexes a look-up table whose entries contain the inverse square root on the restricted domain. The final seed value is formed by merging the seed mantissa and seed exponent.

Single-precision IEEE floating-point numbers employ a 24-bit mantissa (with the most significant one bit “hidden”), an 8-bit excess-127 exponent and a sign bit [IEEE85]. Since the iteration we have chosen has quadratic convergence, the number of significant bits roughly doubles with each iteration. This suggests a seed table indexed by a 12-bit mantissa,

requiring just one iteration. However, the table length (2 2^{12}2-byte entries, hence 16,384 bytes) becomes prohibitive. Additional iterations allow for a much more relaxed table length, described later.

The Iteration

Given an approximate inverse square root y_n , a better one y_{n+1} may be found using the iteration[1]

$$y_{n+1} = \frac{y_n\left(3 - xy_n^{\,2}\right)}{2}$$

An implementation is presented below.

C Implementation

```
/* Compute the Inverse Square Root
 * of an IEEE Single Precision Floating-Point number.
 *
 * Written by Ken Turkowski.
 */

/* Specified parameters */
#define LOOKUP_BITS    6   /* Number of mantissa bits for lookup */
#define EXP_POS       23   /* Position of the exponent */
#define EXP_BIAS     127   /* Bias of exponent */

/* The mantissa is assumed to be just down from the exponent */

/* Type of result */
#ifndef DOUBLE_PRECISION
 typedef float FLOAT;
#else /* DOUBLE_PRECISION */
 typedef double FLOAT;
#endif /* DOUBLE_PRECISION */

/* Derived parameters */
#define LOOKUP_POS   (EXP_POS-LOOKUP_BITS) /* Position of the mantissa lookup */
#define SEED_POS     (EXP_POS-8)           /* Position of the mantissa seed */
#define TABLE_SIZE   (2 << LOOKUP_BITS)    /* Number of entries in the table */
#define LOOKUP_MASK  (TABLE_SIZE - 1)            /* Mask for table input */
#define GET_EXP(a)   (((a) >> EXP_POS) & 0xFF)   /* Extract exponent */
#define SET_EXP(a)   ((a) << EXP_POS)            /* Set exponent */
#define GET_EMANT(a) (((a)>>LOOKUP_POS)&LOOKUP_MASK) /* Get extended mantissa
                                                        MSB's */
#define SET_MANTSEED(a) (((unsigned long)(a)) << SEED_POS)  /* Set mantissa
                                                               8 MSB's */

#include <stdlib.h>
#include <math.h>

static unsigned char *iSqrt = NULL;

union _flint {
```

[1]This algorithm was inspired by the Weitek Technical note “Performing Floating-Point Square Root with the WTL 1032/1033”.

```
    unsigned long    i;
    float            f;
} fi, fo;

static void
MakeInverseSqrtLookupTable(void)
{
    register long f;
    register unsigned char *h;
    union _flint fi, fo;

    iSqrt = malloc(TABLE_SIZE);
    h = iSqrt;

    for (f = 0, h = iSqrt; f < TABLE_SIZE; f++) {
        fi.i = ((EXP_BIAS-1) << EXP_POS) | (f << LOOKUP_POS);
        fo.f = 1.0 / sqrt(fi.f);
        *h++ = ((fo.i + (1 << (SEED_POS-2))) >> SEED_POS) & 0xFF; /* rounding */
    }
    iSqrt[TABLE_SIZE / 2] = 0xFF;    /* Special case for 1.0 */
}


/* The following returns the inverse square root */
FLOAT
InvSqrt(float x)
{
    register unsigned long a = ((union _flint*)(&x))->i;
    register float arg = x;
    union _flint seed;
    register FLOAT r;

    if (iSqrt == NULL)
        MakeInverseSqrtLookupTable();

    seed.i = SET_EXP(((3*EXP_BIAS-1) - GET_EXP(a)) >> 1)
           | SET_MANTSEED(iSqrt[GET_EMANT(a)]);

    /* Seed: accurate to LOOKUP_BITS */
    r = seed.f;

    /* First iteration: accurate to 2*LOOKUP_BITS */
    r = (3.0 - r * r * arg) * r * 0.5;

    /* Second iteration: accurate to 4*LOOKUP_BITS */
    r = (3.0 - r * r * arg) * r * 0.5;

#ifdef DOUBLE_PRECISION
    /* Third iteration: accurate to 8*LOOKUP_BITS */
    r = (3.0 - r * r * arg) * r * 0.5;
#endif /* DOUBLE_PRECISION */

    return(r);
}
```

Numerical Accuracy (Empirical Results)

This procedure has been exhaustively tested for all single-precision IEEE mantissas between 0.5 and 2.0, using IEEE arithmetic[2]. The results are summarized in Table 1.

	Single	**Precision**
iterations	*seed bits*	*final bits*
1	8	16
1	7	14
1	6	12
2	8	23
2	7	23
2	6	23
2	5	21
2	4	17
3	4	23
3	3	23
	Double	**Precision**
2	8	32
2	7	29
2	6	25
2	5	21
3	8	52
3	7	52
3	6	51
3	5	43
3	4	35
3	3	27

Table 1. Effect of Seed Precision on Resultant Precision

Note that the minimum of the maximum errors is one least significant bit[3], i.e. perfect accuracy is never achieved for all possible numbers. This is due to numerical roundoff in intermediate computations. However, in the case of two single-precision iterations from a six, seven and eight bit seed, an “exact” result is computed for nearly all numbers (except for one bit errors in 0.7%, 0.04%, and 0.007% of all numbers, respectively).

From Table 1 it can be seen that the techniques producing the highest accuracy with the minimum memory and computation are a six bit seed with two iterations or a three bit seed with three iterations for single precision, and a seven bit seed with three iterations for double precision. Obviously, a smaller table or less iterations can be used if less precision is adequate for a given task. Note that single precision may be employed to compute the first 23 bits of double precision calculations.

A slight increase in overall accuracy may be achieved by judicious choice of seed values. The method for determining the seed value in this algorithm was found superior to that used in the Weitek Technical Note, but there is still room for further improvement. In particular, the computed exponent for numbers just slightly greater or equal to one is too small, so the mantissa is set to the largest value in the table to compensate for this. Additionally, up to one more effective bit of seed precision could be achieved by set the

[2]IEEE arithmetic implementations include such features as “round-to-nearest or even-if-tie”, a guard bit, a round bit, and a sticky bit, etc.

[3]IEEE mantissas are 24 and 53 bits for single and double precision, respectively.

table value equal to the *average* of the range for the entry, rather than the edge of the range as is done in this implementation.

Implementation Notes

Certain compilers do not pass single-precision values as procedure parameters but instead promote them to double or extended precision. In such cases, pointers may be passed instead. The multiplication by 0.5 amounts to a decrement of the exponent, as supported by the IEEE defined operation `scalb`. Unless hand-coding, the machine multiply is faster then the subroutine overhead lost in invoking `ldexp()`, `scalb()` or related routines to effect the change.

The code is highly portable: non-IEEE (e.g. radix-16) floating point hardware requires merely new macros for proper seed construction. A 128-byte table is small enough to be hard-coded into the sources; this also assures that the correct table entries (to the LSB) are evaluated and further allows for more carefully tuned/tweaked entries whose defining formula might be complex.

Previous gems [Lalonde90] [Hill92] use a similar method for constructing and indexing a mantissa table. However, these solve instead for the conventional square-root and omit the iteration step.

Bibliography

Hill92 Steve Hill, IEEE Fast Square Root, In David Kirk, editor, *Graphics Gems III*, p. 48, Academic Press, 1992.

Hwang79 Kai Hwang, *Computer Arithmetic: Principles, Architecture and Design*, pp. 360-379, Wiley, 1979.

IEEE85 ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, New York, NY, 1985.

Lalonde90 Paul Lalonde and Robert Dawson, A High-Speed, Low-Precision Square Root, In Andrew Glassner, editor, *Graphics Gems*, pp. 424-426, Academic Press, 1990.

Ralston78 Anthony Ralston and Philip Rabinowitz, *A First Course in Numerical Analysis*, pp. 344-347, McGraw-Hill, 1978.