An Approximation for the Inverse Square Root Function

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: May 14, 2015

Contents

1 Introduction

This document is about minimax approximation for computing the inverse square root of a floating-point number. The algorithm uses Newton's method for root estimates and a minimax estimate for relative error. The heart of the problem is selecting a good initial guess for Newton's method leads to a small relative error bound for all finite floating-point numbers, whether float (32-bit) or double (64-bit).

2 Approximation Using Newton's Method

Given a value $x > 0$, the inverse square root is $y = 1/\sqrt{x}$. We wish to approximate y using a small number of arithmetic operations, whether for reducing the cost of evaluation on hardware that has a floating-point unit or because the hardware does not have an implementation of inverse square root. The latter category includes HLSL for Direct3D 11.1 that has no double-precision support for any mathematics library functions, not even for sqrt. We can reformulate the problem in terms of root finding. For the selected x, define $F(y) = 1/y^2 - x$. Solving $F(\bar{y}) = 0$ with the constraint $\bar{y} > 0$, we obtain $\bar{y} = 1/\sqrt{x}$. Let $y_0 > 0$ be an initial guess for $\bar{y}$. The iteration scheme is

$$y_{n+1} = y_n = F(y_n)/F'(y_n), \quad n \geq 0 \tag{1}$$

where $F'(y) = -2/y^3$ is the derivative of $F(y)$. The equation reduces to

$$y_{n+1} = y_n(3 - xy_n^2)/2, \quad n \geq 0 \tag{2}$$

If the sequence of iterations converges as $n \to \infty$, the limit must be $\bar{y}$. If y_0 is a good initial guess, a small number of iterations should give you a decent approximation to $\bar{y}$.

The problem can be reformulated by writing $x = t * 2^e > 0$ with $t \in [1/2, 1)$. The inverse square root is

$$y = \frac{1}{\sqrt{x}} = \begin{cases} \left(\frac{1}{\sqrt{t}}\right) 2^{-e/2}, & e \text{ is even} \\ \left(\frac{\sqrt{2}}{\sqrt{t}}\right) 2^{-(e+1)/2}, & e \text{ is odd} \end{cases} \tag{3}$$

The number $\sqrt{2}$ can be hard-coded in an implementation, so the problem reduces to approximating $1/\sqrt{t}$ for $t \in [1/2, 1)$. The two cases allow an implementation to use the standard mathematics library functions frexp and ldexp, the former for extracting t and e from x and the latter for computing y from the approximation to $1/\sqrt{t}$ and the exponent $-e/2$ when e is even or $-(e+1)/2$ when e is odd. In an implementation, calls to frexp and ldexp are expensive because they have to handle all floating-point inputs. In the implementation provided in this document, bit twiddling is used instead that takes advantage of our knowledge of the inputs and outputs of the function. This leads to an evaluation that is faster than the reciprocal of a square root.

3 Minimax Estimation for Relative Error

We need an initial guess $y_0(t; a, b) = a + bt$ for some choice of parameters (a, b). The function notation indicates that (a, b) are fixed and t varies. The first Newton iterate is then

$$y_1(t; a, b) = (a + bt)(3 - t(a + bt)^2)/2, \quad t \in [1/2, 1) \tag{4}$$

The maximum relative error for t-values is

$$E(a,b) = \max_{t\in[1/2,1)} \left| \frac{y_1(t;a,b) - 1/\sqrt{t}}{1/\sqrt{t}} \right| = \max_{t\in[1/2,1)} \left| \sqrt{t}\, y_1(t;a,b) - 1 \right| \tag{5}$$

We wish to choose (a,b) to minimize the maximum relative error (a minimax problem); that is, we must compute (a,b) for which $E(a,b)$ has a global minimum. The function $h(t) = 1/\sqrt{t}$ is a decreasing function with $h(1/2) = \sqrt{2}$ and $h(1) = 1$. It is natural to constrain $a > 0$ and $b < 0$ based on the geometry of a line segment that fits a positive and decreasing function.

Define the signed relative error

$$S(t;a,b) = \sqrt{t}\, y_1(t;a,b) - 1 \tag{6}$$

The maximum of $E(a,b)$ must occur at a critical point of $|S(t;a,b)|$ in the t-interval $[1/2,1)$. From calculus, the critical points consist of t-values where the derivative is zero or undefined and of the endpoints. The derivative is (potentially) discontinuous where $S(t;a,b) = 0$, but because the S-value is zero, such critical points do not generate the maximum relative error. Thus, we need only analyze t for which the derivative is zero, $S'(t;a,b) = 0$, and at endpoints $t = 1/2$ and $t = 1$. If $C(a,b)$ is the set of critical points for a parameter pair (a,b), then

$$E(a,b) = \max_{t\in C(a,b)} |S(t;a,b)| \tag{7}$$

The derivative of S is

$$\begin{aligned} S'(t;a,b) &= \sqrt{t}\, y_1'(t;a,b) + y_1(t;a,b)/(2\sqrt{t}) \\ &= \left(t y_1'(t;a,b) + y_1(t;a,b)/2\right)/\sqrt{t} \\ &= (-3/4)(a+3bt)(t(a+bt)^2 - 1)/\sqrt{t} \end{aligned} \tag{8}$$

Define the cubic polynomial

$$p(t;a,b) = t(a+bt)^2 - 1 = b^2t^3 + 2abt^2 + a^2t - 1 \tag{9}$$

The solutions to $S'(t;a,b) = 0$ are the real-valued roots of $p(t;a,b)$ and $t = -a/(3b)$.

The analysis of the root classification of $p(t;a,b)$ is of interest. The details of such classification and root construction may be found in Low-Degree Polynomial Roots. The discriminant of the polynomial is $\Delta = -b^3(4a^3 + 27b)$. We are concerned with computing parameter pairs (a,b) for $a > 0$ and $b < 0$, so observe that $\Delta = 0$ when $b = -4a^3/27$. The curve defined by this equation cuts through the fourth quadrant of the ab-plane, so the root classification is not consistent among all (a,b) of interest; that is, we expect one real root when $b < -4a^3/27$, three distinct real roots when $b > -4a^3/27$, or two distinct real roots when $b = -4a^3/27$.

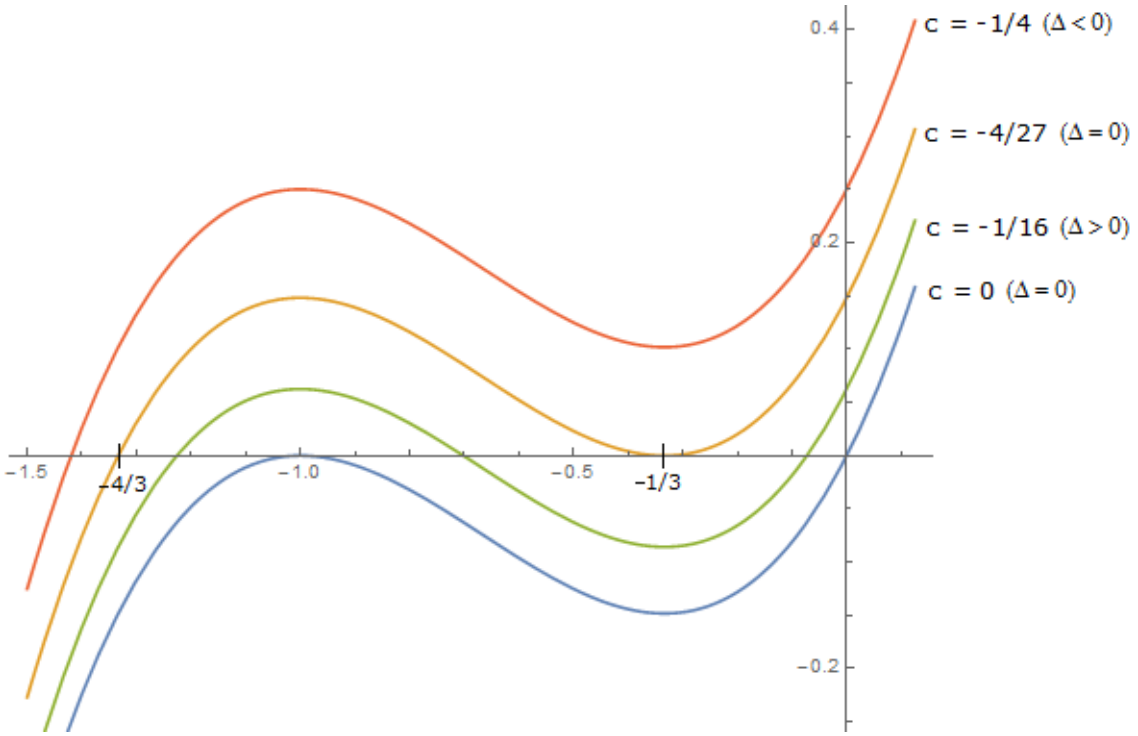
4 Analysis of the Critical Points

A change of variables can be made for the cubic polynomial of Equation (9). Define $z = bt/a$, $c = b/a^3$, and

$$q(z;c) = (b/a^3)p(t) = z^3 + 2z^2 + z - c = z(z+1)^2 - c \tag{10}$$

We are concerned only about (a,b) in the fourth quadrant, so $a > 0$ and $b < 0$ which implies $c < 0$. The discriminant of $q(z)$ is $\delta = -c(4 + 27c)$ and has the same sign as Δ. Figure 1 shows graphs of $q(z;c)$ for several choices of c. The graphs were drawn with Mathematica 10.1 [1] with annotations added manually.

Figure 1. Graphs of $q(z;c)$. The blue graph is for $c = 0$, the green graph is for $c = -1/16$, the yellow graph is for $c = -4/27$, and the red graph is for $c = -1/4$. Each graph is a vertical translation of the blue graph.



Because we have constrained $c < 0$, the blue graph for $c = 0$ is a lower limit on the graphs we care about.

Some observations are in order for $q(z;c)$ when $c < 0$. The real-valued roots are always negative. When $\Delta < 0$, there is a single root $z_0 < -4/3$. When $\Delta > 0$, there are three distinct roots $z_0 < z_1 < z_2$ where $z_0 \in (-4/3, -1)$, $z_1 \in (-1, -1/3)$, and $z_2 \in (-1/3, 0)$. When $\Delta = 0$ and $c < 0$, there are two distinct roots $z_0 = -4/3$ and $z_1 = -1/3$. The z-derivative of q has two real roots regardless of c, namely, $q'(-1;c) = 0$ and $q'(-1/3;c) = 0$.

Let us compute the relative error at t corresponding to a root z of $q(z;c)$. We can do so by making the same change of variables $t = az/b$ and $c = b/a^3$ in the relative error,

$$\varepsilon = |\sqrt{t}\, y_1(t;a,b) - 1| = |\sqrt{az/b}(a(z+1)(3 - (az/b)a^2(z+1)^2))/2 - 1| = |\sqrt{z/c}\,(z+1) - 1| \qquad (11)$$

Observe that $q(z;c) = 0$ implies $z(z+1)^2/c = 1$. For $c < -4/27$, the root $z < -4/3$ leads to $\varepsilon = |\sqrt{z/c}\,(z+1) - 1| = |-\sqrt{z(z+1)^2/c} - 1| = 2$, which is a really large relative error compared to that at the endpoints of $[1/2, 1)$. For $c \geq -4/27$, the root $z < -1$ produces $\varepsilon = 2$ and the roots $z > -1$ lead to $\varepsilon = |\sqrt{z/c}\,(z+1) - 1| = \sqrt{z(z+1)^2/c} - 1| = 0$. In all cases, the roots $z > -1$ produce a global minimum in relative error (of 0).

The remaining critical points are $t = -a/(3b)$ and the interval endpoints $t = 1/2$ and 1. For each (a, b) we can compute the relative errors at the three points and choose the maximum. The final relative error bound is the minimum of all such numbers taken over the fourth quadrant. Equation (5) reduces to

$$\begin{aligned} f(a,b) &= \left|\sqrt{1/2}\,(a+b/2)(3-(a+b/2)^2/2)/2-1\right|, && \text{relative error at } 1/2 \\ g(a,b) &= \left|(a+b)(3-(a+b)^2)/2-1\right|, && \text{relative error at } 1 \\ h(a,b) &= \left|\sqrt{-4a^3/(27b)}\,(3+4a^2/(27b))/2-1\right|, && \text{relative error at } -a/(3b) \\ E(a,b) &= \max\{f(a,b), g(a,b), h(a,b)\} \end{aligned} \tag{12}$$

The minimizer $(\bar{a}, \bar{b})$ is the pair of parameters that minimizes $E(a, b)$ for $a > 0$ and $b < 0$. This pair was computed using using Mathematica 10.1 [1]. In particular, the Mathematica documentation has sample code for minimizing the maximum of functions,

```
FindMinMax[{f_Max, cons_}, vars_, opts___?OptionQ] :=
    With[{res = iFindMinMax[{f, cons}, vars, opts]}, res /; ListQ[res]];
iFindMinMax[{ff_Max, cons_}, vars_, opts___?OptionQ] := Module[{z, res, f = List @@ ff},
    res = FindMinimum[{z, (And @@ cons) && (And @@ Thread[z >= f])},
        Append[Flatten[{vars}, 1], z], opts];
    If[ListQ[res], {z /. res[[2]], Thread[vars -> (vars /. res[[2]])]}]];

err[t_, a_, b_] := Abs[Sqrt[t] * (a + b*t) * (3 - t*(a + b*t)^2)/2 - 1]
FindMinMax[{Max[{err[1/2, a, b], err[1, a, b], err[-a/(3*b), a, b]}],
    {1 <= a && a <= 2 && -1 <= b && b <= -1/2}}, {a, b}, WorkingPrecision -> 20]
```

The results are

$$\begin{aligned} \bar{a} &= 1.7875798999734804109 \\ \bar{b} &= -0.80992000992385987815 \\ E(\bar{a},\bar{b}) &= 0.00074304609193087043843 \end{aligned} \tag{13}$$

5 An Implementation

A C++ implementation is shown in Listing 1, where the input is any positive finite float, whether normal or subnormal.

Listing 1. An implementation for the minimax-based inverse square root function for positive and finite 32-bit floating-point numbers.

```
static int32_t const gsLeadingBitTable[32] =
{
     0,  9,  1, 10, 13, 21,  2, 29,
    11, 14, 16, 18, 22, 25,  3, 30,
     8, 12, 20, 28, 15, 17, 24,  7,
    19, 27, 23,  6, 26,  5,  4, 31
};

int32_t GetLeadingBit(uint32_t value)
{
    value |= value >> 1;
    value |= value >> 2;
    value |= value >> 4;
```

```
    value |= value >> 8;
    value |= value >> 16;
    uint32_t key = (value * 0x07C4ACDDu) >> 27;
    return gsLeadingBitTable[key];
}

float MinimaxInvSqrt(float x)
{
    union { float number; uint32_t encoding; } t, y1;

    // The function call 't = frexp(x, &e)' is too expensive to use.  It
    // has to deal with general floating-point input.  The code here takes
    // advantage of knowledge about the input.
    t.number = x;
    int biased = t.encoding & 0x7F800000;
    int trailing = t.encoding & 0x007FFFFF;
    int e;
    if (biased != 0)
    {
        e = (biased >> 23) - 126;
    }
    else
    {
        int leading = GetLeadingBit(trailing);
        e = leading - 148;
        trailing = (trailing << (23 - leading)) & 0xFF7FFFFF;
    }
    trailing |= 0x3F000000;
    t.encoding = trailing;

    float adjust;
    if (e & 1)
    {
        adjust = 0.707106769f;  // 0.5 * sqrt(2)
        ++e;
    }
    else
    {
        adjust = 0.5f;
    }
    e = -(e / 2);

    float y0 = 1.78757989f - 0.809920013f * t.number;
    y1.number = adjust * y0 * (3.0f - t.number * y0 * y0);

    // The function call 'result = ldexp(y1, e)' is too expensive to use.
    // It has to deal with general floating-point input.  The code here
    // takes advantage of knowledge about the output:  the number y1 is
    // guaranteed to be a finite normal floating-point number.
    y1.encoding += (e << 23);
    return y1.number;
}
```

Evaluating the relative error for all positive float values, whether normal or subnormal, the result was 0.000743150711.

The code was timed on an Intel® Core™ i7-3930K CPU running at 3.20GHz, executing 1/sqrt(x) and MinimaxInvSqrt(x) over all positive and finite float numbers. The function 1/sqrt(x) used 16.86 seconds and the function MinimaxInvSqrt(x) used 11.86 seconds. The global relative error bound for MinimaxInvSqrt(x) from the evaluations at all the inputs is 0.00074321, which is approximately what Mathematica 10.1 [1] generated. A profiler showed that most of the time in MinimaxInvSqrt(x) is spent copying the result from a register to main memory.

A C++ implementation is shown in Listing 2, where the input is any positive finite double, whether normal or subnormal.

Listing 2. An implementation for the minimax-based inverse square root function for positive and finite 64-bit floating-point numbers. The function int32_t GetLeadingBit(int32_t) is the one from Listing 1.

```
int32_t GetLeadingBit(int64_t value)
{
    int32_t v1 = (int32_t)((value >> 32) & 0x00000000FFFFFFFFLL);
    if (v1 != 0)
    {
        return GetLeadingBit(v1) + 32;
    }
    else
    {
        int32_t v0 = (int32_t)(value & 0x00000000FFFFFFFFLL);
        return GetLeadingBit(v0);
    }
}

double InvSqrt(double x)
{
    union { double number; uint64_t encoding; } t, y1;

    // The function call 't = frexp(x, &e)' is too expensive to use.  It
    // has to deal with general floating-point input.  The code here takes
    // advantage of knowledge about the input.
    t.number = x;
    int64_t biased = t.encoding & 0x7FF0000000000000LL;
    int64_t trailing = t.encoding & 0x000FFFFFFFFFFFFFLL;
    int64_t e;
    if (biased != 0)
    {
        e = (biased >> 52) - 1022;
    }
    else
    {
        int leading = GetLeadingBit(trailing);
        e = leading - 1073;
        trailing = (trailing << (52 - leading)) & 0xFFEFFFFFFFFFFFFFLL;
    }
    trailing |= 0x3FE0000000000000LL;
    t.encoding = trailing;

    double adjust;
    if (e & 1)
    {
        adjust = 0.7071067811865475;
        ++e;
    }
    else
    {
        adjust = 0.5;
    }
    e = -(e / 2);

    double y0 = 1.7875798999734804 - 0.80992000992385993 * t.number;
    y1.number = adjust * y0 * (3.0 - t.number * y0 * y0);

    // The function call 'result = ldexp(y1, e)' is too expensive to use.
    // It has to deal with general floating-point input.  The code here
    // takes advantage of knowledge about the output:  the number y1 is
    // guaranteed to be a finite normal floating-point number.
    y1.encoding += (e << 52);
    return y1.number;
}
```

References

[1] Wolfram Research, Inc. *Mathematica 10.1.* Wolfram Research, Inc., Champaign, Illinois, 2015.