

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## Fast Square Root & inverse calculation for Arbitrary Precision numbers.

By Henrik Vestermark (hve@hvks.com)

### Abstract

This paper describes in detail how to compute the square root, inverse ( $1/x$ ), and  $\sqrt[n]{x}$  to arbitrary precision. It surveys traditional iterative methods, Newton, Halley, and Goldschmidt, and introduces an improved iteration scheme using dynamic precision scheduling that approximately halves the runtime of each computation. The paper also derives rigorous theoretical lower bounds on the number of guard digits required to guarantee a result within one ULP of the true value for each algorithm, presented in Appendix A (square root), Appendix B (inverse), and Appendix C (nth-root). The original paper dates from 2013; this revision incorporates additional methods and the guard digit analysis.

### Introduction

Usually, when implementing arbitrary precision math packages, the standard Newton iteration is the preferred method for calculating the square root, inverse ( $1/x$ ), or  $\sqrt[n]{x}$ . The Newton method has quadratic convergence, meaning the number of correct digits doubles with each iteration. This is the traditional approach and has been implemented in various arbitrary precision packages. However, methods with higher-order convergence rates exist, notably Halley's method with cubic convergence and the Goldschmidt method; we examine whether the extra work per iteration is justified compared to Newton's method. Beyond the algorithmic comparison, the paper also derives rigorous theoretical lower bounds on the number of guard digits needed to guarantee a result within one ULP of the true value. These bounds are worked out from first principles for each algorithm by counting floating-point operations per iteration and checking for catastrophic cancellation, and are presented in Appendix A (square root methods), Appendix B (inverse methods), and Appendix C (nth-root).

As usual, we will show the actual C++ source code for the computation using the author's arbitrary-precision Math library; see [1].

This paper is part of a series of arbitrary-precision papers that describe methods, implementation details, and optimization techniques. These papers can be found on my website at [www.hvks.com/Numerical/papers.html](http://www.hvks.com/Numerical/papers.html) and are listed below:

1. Fast arbitrary precision integer multiplication. [HVE Fast arbitrary precision integer multiplication](#)
2. Fast Computation of Math Constants in arbitrary precision. [HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.](#)

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

3. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. [HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.](#)
4. Fast Square Root and inverse calculation for arbitrary precision math. [HVE Fast Square Root & inverse calculation for arbitrary precision](#)
5. Fast Exponential calculation for arbitrary precision math. [HVE Fast Exp\(\) calculation for arbitrary precision](#)
6. Fast logarithm calculation for arbitrary precision math. [HVE Fast Log\(\) calculation for arbitrary precision](#)
7. Practical implementation of Spigot Algorithms for Transcendental Constants. [Practical implementation of Spigot Algorithms for transcendental constants](#)
8. Practical implementation of  $\pi$  algorithms. [HVE Practical implementation of PI Algorithms](#)
9. Fast Trigonometric function for arbitrary precision. [HVE Fast Trigonometric calculation for arbitrary precision](#)
10. Fast Hyperbolic functions for arbitrary precision. [HVE Fast Hyperbolic calculation for arbitrary precision](#)
11. Fast conversion from arbitrary precision number to a string. [HVE Fast conversion from arbitrary precision to string](#)
12. Fast conversion from a decimal string to an arbitrary precision number. [HVE Fast conversion from string to arbitrary precision](#)
13. Fast Computation of Stirling's numbers in arbitrary precision. [HVE Fast Computation of Stirling numbers in arbitrary precision](#)
14. Fast Prime computation in arbitrary precision. [HVE Fast Prime Computation in arbitrary precision](#)
15. Fast PRNG in arbitrary precision. [HVE Fast PRNG in arbitrary precision](#)
16. Fast Fibonacci sequence in arbitrary precision. [HVE Fast Fibonacci in arbitrary precision](#)

## Change log

5 March 2026 Added appendices A, B, and C covering bounds for the errors in the computation of the functions in this document. Improve the layout and the text throughout the document.

3 January 2026. Minor typos

1-March 2023. Minor corrections,

26-January 2023. Cleaning up the document grammatically.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## Contents

Abstract .....	1
Introduction.....	1
Change log .....	2
The Arbitrary precision library .....	6
Internal format for float_precision variables .....	7
Normalized numbers.....	8
Square root.....	9
Newton's Method without division .....	9
The Initial guess.....	10
Example of Newton's method for the square root.....	10
Brent improvement .....	11
Newton's method with division.....	11
The initial guess.....	12
Example Newton method for the square root using division.....	12
Halley's method.....	12
Example of Halley's method for the square root.....	14
Goldschmidt method.....	14
Further Improvement of the methods?.....	14
Iteration using Dynamic Precision.....	15
Number as a power of two.....	16
Precision less than 16 digits.....	17
Source sqrt_newton_dynamic().....	17
Source sqrt_Hybrid_dynamic().....	18
Recommendation for the square root.....	19
Nroot.....	21
Source for nroot_newton_dynamic().....	22
Recommendation $^n\sqrt{x}$ .....	24
The Inverse.....	25
Newton's method for inverse.....	25
Example of Newton's method for inverse .....	26
Brent improvement .....	27
Iteration with dynamic precision .....	27
Source inverse_newton_dynamic().....	27
Halley method for inverse.....	28
Example of the Halley convergence method for inverse.....	29
Source inverse_Halley_dynamic().....	30
Goldschmidt inverse .....	31
Source Goldschmidt inverse .....	31
Performance:.....	32
Recommendation Inverse.....	33
Reference .....	34
Appendix A: Guard Digit Analysis for Square Root Algorithms.....	35
A.1 Notation and Setup.....	35
A.2 Newton's Method Without Division.....	35

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

A.2.1	Algorithm	35
A.2.2	Convergence in Exact Arithmetic	35
A.2.3	Rounding Error Per Iteration	36
A.2.4	Final Multiplication Error	36
A.2.5	Guard Digit Requirement	36
A.3.1	Algorithm	36
A.3.2	Rounding Error Analysis	37
A.3.3	Guard Digit Requirement	37
A.4	Halley's Method (Cubic Convergence)	37
A.4.1	Algorithm	37
A.4.2	Convergence in Exact Arithmetic	37
A.4.3	Cancellation Analysis	38
A.4.4	Rounding Error Per Iteration	38
A.4.5	Guard Digit Requirement	38
A.5	Recommendations	39
Appendix B:	Guard Digit Analysis for Inverse (1/y) Algorithms	40
B.1	Notation and Setup	40
B.2	Newton's Method for Inverse	40
B.2.1	Algorithm	40
B.2.2	Convergence in Exact Arithmetic	40
B.2.3	Cancellation Analysis	41
B.2.4	Rounding Error Per Iteration	41
B.2.5	Guard Digit Requirement	41
B.3	Newton's Method with Brent's Half-Precision Improvement	41
B.3.1	Algorithm	41
B.3.2	Rounding Error Analysis	42
B.3.3	Guard Digit Requirement	42
B.4	Cubic (Third-Order) Householder Method for Inverse	42
B.4.1	Algorithm	42
B.4.2	Convergence in Exact Arithmetic	42
B.4.3	Cancellation Analysis	43
B.4.4	Rounding Error Per Iteration	43
B.4.5	Guard Digit Requirement	43
B.5	Goldschmidt Method	44
B.5.1	Algorithm	44
B.5.2	Convergence in Exact Arithmetic	44
B.5.3	Cancellation Analysis	44
B.5.4	Rounding Error Per Iteration	45
B.5.5	Guard Digit Requirement	45
B.6	Recommendations	45
Appendix C:	Guard Digit Analysis for the n-th Root Algorithm	47
C.1	Notation and Setup	47
C.1.1	Exponent Extraction and Working Range	47
C.2	The nroot Newton Algorithm	47
C.2.1	Derivation	48

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

C.2.2	Convergence in Exact Arithmetic .....	48
C.3	Operation Count and Rounding Error .....	48
C.3.1	Computing $x^n$ by Binary Exponentiation.....	48
C.3.2	Remaining Operations Per Iteration.....	49
C.3.3	The Final Inversion $n\sqrt{S} = 1/x$ .....	49
C.4	Total Error Budget and Guard Digit Bound.....	49
C.4.1	Combining All Error Sources.....	50
C.4.2	Guard Digit Requirement .....	50
C.4.3	Simplified Practical Bound .....	50
C.5	Comparison with Current Implementation.....	51
C.6	Recommendation.....	51

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## The Arbitrary precision library

If you already are familiar with the arbitrary precision library, you can skip this section.

To understand the C++ code and text we have to highlight a few features of the arbitrary precision library where the class name is *float\_precision*. Instead of declaring, a variable with a float or double you just replace the type name with *float\_precision*. E.g.

```
float_precision f; // Declare an arbitrary precision float with 20 decimal digits of precision
```

You can add a few parameters to the declaration. The first is the optional initial value, and the second optional parameter is the floating-point precision. The native type of a *float* has a fixed size of 4 bytes, and 8 bytes for *double*. Since this precision can be arbitrary, we can specify the desired precision as the number of decimal digits to use when working with the variable. E.g.

```
float_precision fp(4.5); // Initialize it to 4.5 with default 20 digits precision
float_precision fp(6.5,10000); // Initialize it to 6.5 with a precision of 10,000 digits
```

The precision of a variable can be dynamic and change throughout the code, which is very handy to manipulate the variable. To change or set the precision you can call the method `.precision()` E.g.

```
f.precision(100000); // Change the precision to 100,000 digits
f.precision(fp.precision()-10); // Lower the precision with 10 digits
f.precision(fp.precision()+20); // Increase precision with 20 digits
```

There is another method for manipulating the exponents of the variables. The method is called `.exponent()` and returns or sets the exponent as a power of two exponents (same as for our regular built-in types float and double). E.g.

```
f.exponent(); // Return the exponent as 2e
f.exponent(0) // Remove the exponent
f.exponent(16) // Set the exponent to 216
```

There is a second way to manipulate the exponent: the class method `.adjustExponent()`. This method just adds the parameter to the internal variable that holds the exponent of the *float\_precision* variable. E.g.

```
f.adjustExponent(+1); // Add 1 to the exponent, the same as multiplying the number by 2.
f.adjustExponent(-1); // Subtract 1 from the exponent, the same as dividing the number by 2.
```

This allows very fast multiplication or division by a number that is any power of two.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

The method `.iszero()` returns true if the `float_precision` number is zero, otherwise false. There is an additional `method()` but I will refer to the reference for the user manual to the arbitrary precision math package for details.

All the normal operators and library calls that work with the built-in type `float` or `double` will also work with the `float_precision` type using the same name and calling parameters.

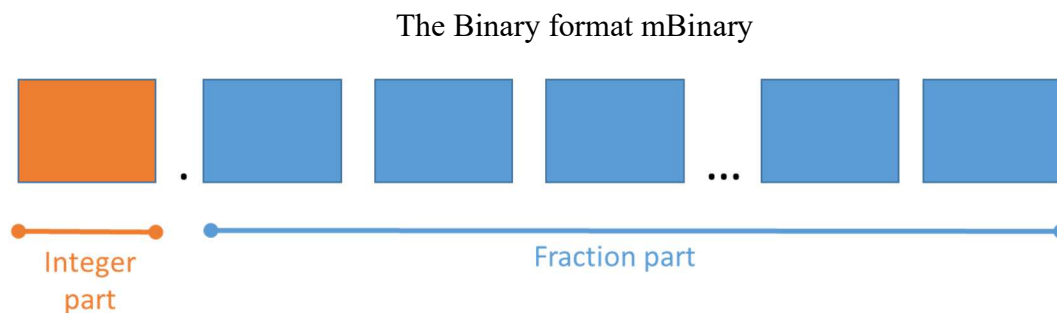
## Internal format for `float_precision` variables

For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mBinary;
```

`uintmax_t` is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integers to store our floating-point precision number.

The method `.size()` returns the number of internal vector entries needed to hold the number.



- The binary format consist of an unlimited number of 64bit unsigned integer blocks.
- One block in front of the period sign ‘.’ (the integer part of the number)
- Zero or more blocks of fractions after the ‘.’ (the fraction sign of the number)
- The binary number is stored in a STL vector class and defined
  - `vector<uintmax_t> mBinary;`
- There is always one entry in the `mBinary` vector.
- Size of vector is always  $\geq 1$
- A Number is always stored normalized. E.g. the integer part is 1 or zero
- The sign, exponent, precision, rounding mode is stored in separate class fields.

There are other internal class variables like the sign, exponent, precision, and rounding mode but these are not important to understand the code segments.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## Normalized numbers

A `float_precision` variable is always stored as a normalized number with a one in the integer portion of the number. The only exception is zero, which is stored as zero. Furthermore, a normalized number has no trailing zeros.

For more details, see [1].

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## Square root

There are several methods for computing the square root. Among them are:

- 1) Newton's Method.
- 2) Halley's Method.
- 3) Goldschmidt.

The most common method for arbitrary-precision libraries is the Newton method.

### *Newton's Method without division*

For the function  $\text{sqrt}(y)$ , we can use Newton's method iteration to obtain the result. The Newton iteration is defined by:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (1)$$

This method can be found in the following way: by restating the problem of finding  $\text{Sqrt}(y)$ , we instead try to find the reciprocal square root of  $y$ , which is:  $\frac{1}{\sqrt{y}}$ . Once it has been found, we can find  $\sqrt{y} = y \frac{1}{\sqrt{y}}$ . By just multiplying the result by  $y$ .

Now to find the  $\frac{1}{\sqrt{y}}$ . We use the equation  $\frac{1}{x^2} = y \Rightarrow \frac{1}{x^2} - y = 0$ .

Using Newton's formula, we get  $f(x) = \frac{1}{x^2} - y$ ,  $f'(x) = \frac{-2}{x^3}$

$$x_{n+1} = x_n - \frac{\frac{1}{x_n^2} - y}{\frac{-2}{x_n^3}} \Rightarrow$$

$$x_{n+1} = x_n + \frac{1}{2} x_n^3 \left( \frac{1}{x_n^2} - y \right) \Rightarrow$$

$$x_{n+1} = \frac{1}{2} x_n (3 - y x_n^2) \quad (2)$$

We now have our algorithm for finding the square root without any division.

$$x_{n+1} = \frac{1}{2} x_n (3 - y x_n^2) \quad (3)$$

Where  $x_0 \approx \frac{1}{\sqrt{y}}$  (initial guess)

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

and  $x_n$  converged towards  $\frac{1}{\sqrt{y}}$

For the initial guess  $x_0$ , we simply use the C library `sqrt(b)` function for the double variable. Now, for this to work for arbitrary precision, we need to use a little trick to ensure that we can call the C library `sqrt` function with a double argument that fits the range of the IEEE754 double standard. See the initial guess section below.

Notice that the algorithm requires only one subtraction and four multiplications per iteration. Well, multiplication by 0.5 can be done by just adjusting the exponent and therefore should not count as a ‘real’ multiplication. We end up with one subtraction and three multiplications per iteration, and then a final multiplication for the calculation of the square root.

Also, regarding the Newton method, we will have quadratic convergence, meaning that at each iteration we double the number of correct digits in our result.

### The Initial guess

As for the initial guess, we can extract the exponent  $2^{e_p}$  out of the equation, then multiply the result by  $2^{\frac{e_p}{2}}$  after the iteration (assuming  $e_p$  is an even integer), and remember our exponent is an integer in base two.  $I_1$  is the one-digit integer, and  $f_n$  is the n fraction parts digits.

$$\frac{1}{\text{Sqrt}(y)} = \frac{1}{\text{Sqrt}(i_1.f_n 2^{e_p})} = \frac{1}{\text{Sqrt}(i_1.f_n)} 2^{-\frac{e_p}{2}} \tag{4}$$

If  $e_p$  is odd, we have to use (since the exponent needs to be an integer):

$$\frac{1}{\text{Sqrt}(y)} = \frac{1}{\text{Sqrt}(i_1.f_n 2^{e_p})} = \frac{1}{\text{Sqrt}(i_1.f_n * 2)} 2^{-\frac{e_p-1}{2}} \tag{5}$$

This simplifies the initial guess since we know that factoring out the exponent will leave us with an arbitrary precision number between [1,2) (for even exponent) and [1..,4) for odd exponent. With the number well within the range of IEEE754, we can find a good initial guess of  $\frac{1}{\text{Sqrt}(y)}$  using standard IEEE754 arithmetic with approx. 15-16 significant decimal digits.

### Example of Newton’s method for the square root

To see how this algorithm works, let us find the Sqrt of 1.6 using an initial start guess of  $1/1.6=0.625$ .

<b>Newton 1/sqrt(y)</b>	
Sqrt(y)	1.6
y=	1.6
$x_0$ =	0.625

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

n	$x_n$	Sqrt(y)	Error
1	0.7421875	1.1875	7.74E-02
2	0.786218643	1.257949829	6.96E-03
3	0.790533565	1.264853704	5.74E-05
4	0.790569413	1.26491106	3.90E-09
5	0.790569415	1.264911064	0.00E+00

After 5 iterations, the difference between the iteration and the built-in Sqrt() operator is 0, and the result of Sqrt(1.6) is 1.264911064

### Brent improvement

Brent [7] points out that you can improve the Newton algorithm by iterating using:

$$x_{n+1} = x_n + \frac{x_n}{2} (1 - yx_n^2) \quad (6)$$

Which is identical from a mathematical point of view but different from a computational one. Brent points out that you can perform the multiplication between  $x_{n-1}$  and  $(1 - yx_n^2)$  in  $\frac{x_n}{2} (1 - yx_n^2)$  using only half the precision in the multiplication. You gain one addition but do not need the multiplication with full precision. From a computational point of view, you do save some time or gain some performance using this formula for the iteration, particularly for a higher number of digits.

### ***Newton's method with division***

Instead of the above indirect method. We could use the direct approach of finding  $\sqrt{y}$  by solving the equation  $f(x)=x^2-y=0$  and using the Newton method to solve for x. By applying Newton's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (7)$$

We get.

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{y}{x_n} \right) \quad (8)$$

Although it looks simple, we have introduced one division per iteration. Any arbitrary precision calculation should avoid divisions whenever possible, since it is many times slower than multiplication. In my arbitrary precision library, it is approx. four-eight times slower than multiplication.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## The initial guess

For the starting guess, we can use the same method as outlined in Newton without division.

As for the initial guess, we can extract the exponent  $2^{e_p}$  out of the equation, and then multiply the result by  $2^{\frac{e_p}{2}}$  after the iteration (assuming  $e_p$  is an even integer), remember our exponent is an integer in base two,  $i_1$  is the one-digit integer, and  $f_n$  is the  $n$  fraction digits.

$$\text{Sqrt}(y) = \text{Sqrt}(i_1 \cdot f_n 2^{e_p}) = \text{Sqrt}(i_1 \cdot f_n) 2^{\frac{e_p}{2}} \quad (9)$$

If  $e_p$  is odd, we have to use (since the exponent needs to be an integer):

$$\text{Sqrt}(y) = \text{Sqrt}(i_1 \cdot f_n 2^{e_p}) = \text{Sqrt}(i_1 \cdot f_n * 2) 2^{-\frac{e_p-1}{2}} \quad (10)$$

This simplifies the initial guess since factoring out the exponent leaves us with an arbitrary-precision number between [1,2) (for even exponent) and [1,4) (for odd exponent). With the number well within the IEEE754 range, we can find a good initial guess using standard IEEE754 arithmetic with approx. 15-16 significant decimal digits.

## Example Newton method for the square root using division

Newton sqrt(y)				
Sqrt(y)				1.6
y=				1.6
x <sub>0</sub> =				1.6
n	x <sub>n</sub>	Sqrt(y)	Error	
1	1.3	1.3	-3.51E-02	
2	1.265384615	1.265384615	-4.74E-04	
3	1.264911153	1.264911153	-8.86E-08	
4	1.264911064	1.264911064	-3.11E-15	
5	1.264911064	1.264911064	0.00E+00	

By using a start guess of 1.6, we get the result after five iterations, and again we observe a quadratic convergence rate,

## Halley's method

Halley's method has a cubic convergence rate compared to Newton's quadratic order. Cubic convergence rate means that for every iteration, you get three times as many correct digits compared to Newton's method, which only gives you two times as many correct digits. Higher-order convergence results in fewer iterations, but at the expense of a more complex calculation per iteration. Normally, the time you save in fewer iteration steps is offset by the increased complexity of the iteration.

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

Halley's square root method uses the following iteration steps for finding  $\frac{1}{\sqrt{y}}$ :

$$\begin{aligned} z_n &= yx_n^2 \\ x_{n+1} &= x_n \frac{1}{8} (15 - z_n (10 - 3z_n)) \end{aligned} \quad (11)$$

And then we get the final result of  $\sqrt{y} = y \cdot x_{n+1}$ .

It can be found using the Householder's 2nd order method, aka. Halley's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{f(x_n)^2 f''(x_n)}{2f'(x_n)^3} \quad (12)$$

Where  $f(x) = y - \frac{1}{x^2}$ ,  $f'(x) = \frac{2}{x^3}$ ,  $f''(x) = -\frac{6}{x^4}$

This yield:

$$x_{n+1} = x_n - \frac{y - \frac{1}{x_n^2}}{\frac{2}{x_n^3}} - \frac{\left(y - \frac{1}{x_n^2}\right)^2 \left(-\frac{6}{x_n^4}\right)}{2\left(\frac{2}{x_n^3}\right)^3} \Rightarrow$$

$$x_{n+1} = x_n - x_n^3 \frac{1}{2} \left(y - \frac{1}{x_n^2}\right) + x_n^5 \frac{3}{8} \left(y - \frac{1}{x_n^2}\right)^2 \Rightarrow$$

$$x_{n+1} = x_n - x_n \frac{1}{2} (yx_n^2 - 1) + x_n \frac{3}{8} (yx_n^2 - 1)^2$$

*Substitute  $z_n = yx_n^2$  you get  $x_{n+1} = x_n - x_n \frac{1}{2} (z_n - 1) + x_n \frac{3}{8} (z_n - 1)^2 \Rightarrow$*

$$x_{n+1} = x_n \frac{1}{8} (8 - 4(z_n - 1) + 3(z_n - 1)^2) \Rightarrow$$

$$x_{n+1} = x_n \frac{1}{8} (15 - 4z_n + 3(z_n^2 + 1 - 2z_n)) \Rightarrow$$

$$x_{n+1} = x_n \frac{1}{8} (15 - 10z_n + 3z_n^2) \Rightarrow$$

$$x_{n+1} = x_n \frac{1}{8} (15 - z_n (10 - 3z_n))$$

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

Per iteration, we have five multiplications and two subtractions. Compared to Newton's method, we have added subtraction and two extra multiplications, so each iteration will take a little longer; however, you will have fewer iterations to perform. (Approx. 2/3).

Example of Halley's method for the square root

Halley 1/Sqrt(y)				
Sqrt(y)			1.6	
y			1.6	
x0			0.625	
n	x <sub>n</sub>	Sqrt(y)	Error	
1	0.775146	1.240234375	2.47E-02	
2	0.790555	1.264887927	2.31E-05	
3	0.790569	1.264911064	1.93E-14	
4	0.790569	1.264911064	0.00E+00	

As expected, we get a faster iteration and reach the result after only four iterations.

## ***Goldschmidt method***

The Goldschmidt method is typically implemented on an FPGA or in a floating-point unit and leverages the CPU pipeline to achieve higher performance. The key idea is to simultaneously compute  $\sqrt{a}$  and  $1/\sqrt{a}$  by iterating both sequences with the same multiplier  $F_i$  at each step, analogous to how the Goldschmidt division method drives both numerator and denominator toward 1 using the same factor.

However, at the software level, it does not produce anything faster or better than a standard Newton method. The hardware advantage comes entirely from executing the two multiplications per iteration in parallel across CPU pipeline stages, something that is not possible in sequential software execution. In software, those two multiplications simply execute one after the other, meaning each iteration costs twice as much as a Newton step while delivering the same quadratic convergence rate. Furthermore, the Newton method benefits from dynamic precision scheduling, starting iterations at low precision and doubling toward the target, which roughly quadruples performance. The Goldschmidt method is structurally incompatible with this optimization for the same reason as in the division case: both sequences must be tracked simultaneously at full precision throughout. For these reasons, Goldschmidt's square root is not implemented here.

## ***Further Improvement of the methods?***

Can we further improve the `sqrt()` calculation? We can apply one major trick. We first note that a Newton iteration is self-correcting, meaning that if we have made an imprecise calculation in one iteration step, it will be corrected automatically in the next Newton iteration.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## Iteration using Dynamic Precision

We can use that information: instead of starting the first couple of iterations with a precision of thousands of digits, we could start with lower precision and gradually increase it until, in the last iterations, we perform all calculations with the required number of digits. Initially, our first guess is approx. 15-16 correct digits. We know that a Newton iteration doubles the number of correct digits for every iteration. We will start with 32 digits of precision and, after each iteration, double the precision. E.g., 64 digits in the next iteration, 128 digits in the following iteration, etc., until we have reached the required final precision.

Applying this technique yields a speed-up over the classical Newton iteration with an approximation. a factor of two times faster than regular Newton. We see the same speed-up improvement comparing regular Halley with dynamic precision. In addition, as you can see in the diagram below, there is a difference between using the Newton and Halley methods. Newton requires fewer multiplications, but Halley requires fewer iterations. Halley's method is consistently approx. 20% faster than the Newton method, as shown in the performance chart below.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

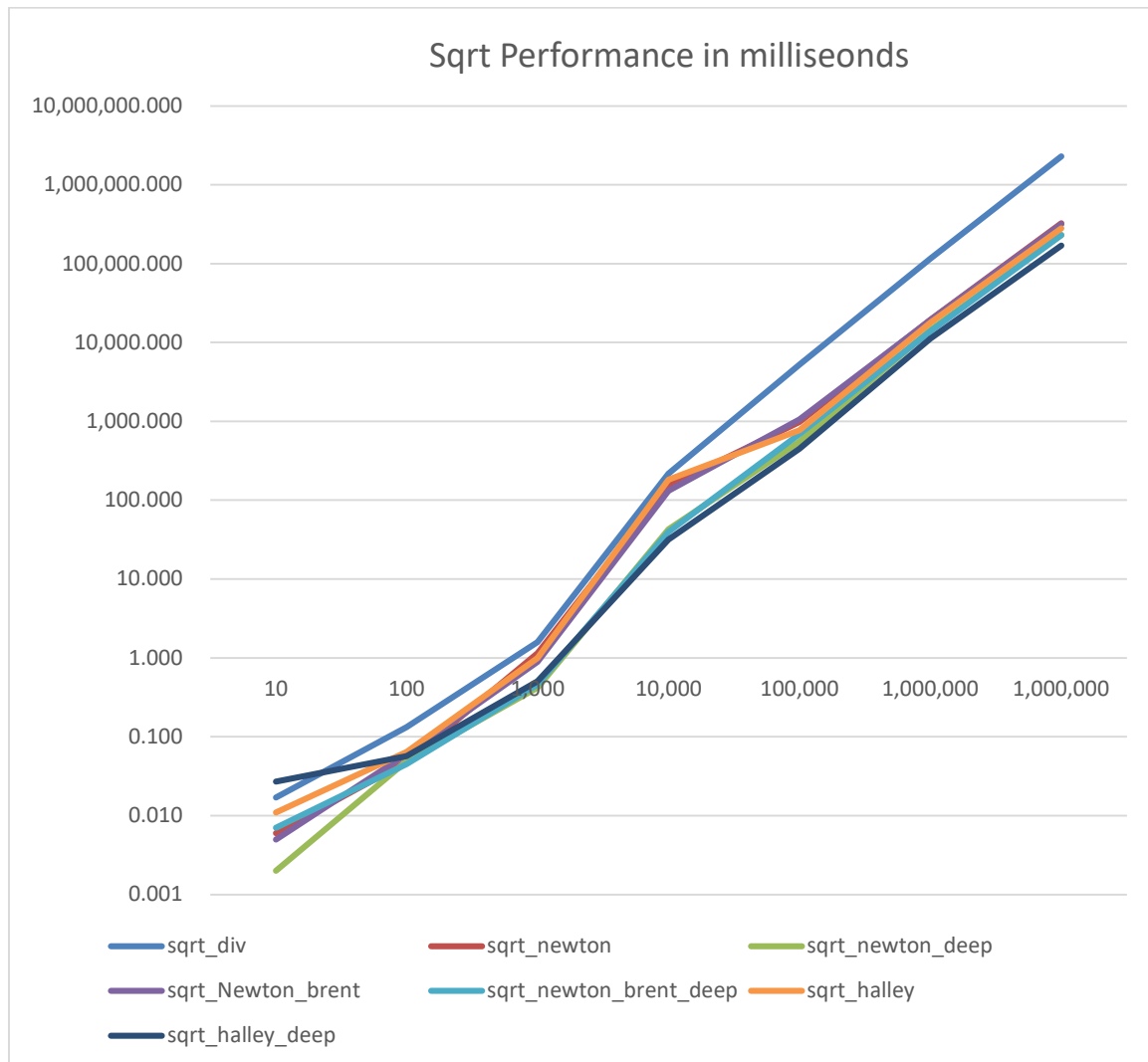


Figure 1. The horizontal axes are precision in decimal digits. The vertical axis is in milliseconds.

The performance gain of dynamic versus regular Newton is approx. two. Moreover, the gain from Newton with division and Newton without division is approx. a factor of four to eight.

## Number as a power of two.

We can use the definition of a normalized number in our arbitrary precision package. A normalized number always has a one as the first digit before the '.' If the number has no fraction part, it will be a true power of two numbers, and the exponent is the power to which the base 2 is raised. If the exponent is even, we can take the square root directly by dividing the exponent by 2; you are done. You can insert the following code right after the first assignment of the local expo variable.

Source:

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

```
// Check for square root is a power of two and even exponent
if (a.size() == 1 && (expo & 0x1) == 0)
    { // True power of 2 and the exponent even
      y.exponent(y.exponent() >> 1); // Half the exponent and return the
result.
      y.precision(precision);
      return y;
    }
```

Of course, not all numbers are powers of two; however, if we encounter one, we can return the square root of that number without any time-consuming iteration.

### Precision less than 16 digits

Another small improvement is if you are working with less than 16 digits of arithmetic. If you do, we can just convert it to a double, calculate the square root using the double type, and then initialize and return a new `float_precision` variable that holds the result, see the code segment below.

Source:

```
// Check if we can handle the request within the IEEE754 double standard
(64bit)
if (precision < 16)
    { double fv;
      fv = a; fv = sqrt(fv);
      return float_precision(fv, precision, a.mode());
    }
```

Of course, you do not expect to encounter many of these situations since you are properly using the arbitrary precision library to work with large numbers in the first place.

### Source `sqrt_newton_dynamic()`

```
float_precision sqrt_newton_dynamic(const float_precision& a)
{
    const unsigned int guard = 2;
    const size_t precision = a.precision();
    const eptype expo = a.exponent();
    const float_precision c1(1), c3(3);
    eptype expo_sq;
    size_t digits;
    double fv;
    float_precision r, x, y(a);

    if (a.iszero() || a == c1) // Simple square root
        return a;
    if (a.sign() < 0)
        { throw float_precision::domain_error();
        }

    // Check for square root within the reach of the IEEE754 standard
    if (y.size() == 1 && (expo & 0x1) == 0)
        { // True power of 2 and the exponent is even
          y.exponent(y.exponent() >> 1); // Half the exponent and return the result.
          return y;
        }
```

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

```
    }

    // Check if we can handle the request within the IEEE754 double standard (64bit)
    if (precision < 16 && abs(expo)<308)
    {
        fv = double(a); fv = sqrt(fv);
        return float_precision(fv, precision, a.mode());
    }

    expo_sq = expo / 2;
    y.exponent(expo - 2 * expo_sq);
    // Do iteration using 2 digits higher precision
    y.precision(precision + guard);
    r.precision(precision + guard);
    x.precision(precision + guard);

    // Get a initial guess using ordinary floating point
    fv = (double)y; // Convert to double
    fv = 1 / sqrt(fv); // set the initial guess with at approx 16 correct digits
    x = float_precision(fv); // Set start iterations value

    // Now iterate using Newton  $x=0.5x(3-yx^2)$ 
    // y is the original number to square root which has the full precision
    for (digits = std::min((size_t)32, precision); ; digits = std::min(precision +
guard, digits * 2), ++loopcnt)
    {
        // Increase precision by a factor of two for the working variable s r & u.
        r.precision(digits);
        x.precision(digits);

        // so we start by assigning it to r, rounding it to the precision of r
        r = y; // y
        r *= x.square(); //  $yx^2$ 
        r = c3 - r; //  $3-yx^2$ 
        //r = c3 - y * x * x; //  $3-yx^2$ 
        r.adjustExponent(-1); //  $(3-yx^2)/2$ 
        x *= r; //  $x=x(3-yx^2)/2$ 
        if (digits == precision + guard) // Reach final iteration step in regards to
precision
        {
            r.precision(precision + 1); // round to final precision
            if (r == c1) // break if no improvement
                break;
            r.precision(precision + guard);
        }
    }

    x *= y;
    x.adjustExponent(expo_sq);
    // Round to same precision as argument and rounding mode
    x.mode(a.mode());
    x.precision(precision);
    return x;
}
```

## Source sqrt Hybrid\_dynamic()

```
float_precision sqrt_halley_dynamic(const float_precision& a)
{
    const size_t guard = 3;
    const size_t precision = a.precision();
    const eptype expo = a.exponent();
    const float_precision c1(1), c15(15), c3(3), c10(10);
    eptype expo_sq;
    double fx = precision + guard < 16 ? 1.0 : log(precision + guard) - log(16);
```

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

```

//const int step = halley == true ? 3 : 2;
float_precision r, x, y(a);

if (a.iszero() || a == c1) // Simple square root
    return a;
if (a.sign() < 0)
    throw float_precision::domain_error();

if (y.size() == 1 && (expo & 0x1) == 0)
{ // True power of 2 and the exponent is even
    y.exponent(y.exponent() >> 1); // Half the exponent and return the result.
    return y;
}
expo_sq = expo / 2;
y.exponent(expo - 2 * expo_sq);
// Do iteration using 2 digits higher precision
y.precision(precision + guard);
r.precision(precision + guard);
x.precision(precision + guard);
// Get a initial guess using ordinary floating point
fx = (double)y; // Convert to double
// set the initial guess with at approx 16 correct digits
fx = 1 / sqrt(fx);
x = float_precision(fx);

// Now iterate using Halley
// Notice y is the original number to square root which has the full precision
size_t digits;
float_precision z;
for (loopcnt_h = 1, digits = std::min(digits, precision); ; digits =
std::min(precision + guard, digits * 3), ++loopcnt_h)
{
    r.precision(digits);
    x.precision(digits);
    z.precision(digits);
    // Use Halley 3rd order iteration
    r = y; // y
    r *= x.square(); // yx^2
    z = c3; z *= r; z = c10 - z;
    r *= z; // r *= c10 - c3*r or r=yx^2(10-3yx^2)
    r = c15 - r; // 15-yx^2*(10-3yx^2)
    r.adjustExponent(-3); // r=r/8
    x *= r; // x=x/8(15-yx^2*(10-3yx^2)
    r.precision(precision + 1); // round to final precision
    if (r == c1) // break if no improvement
        break;
}

x *= y;
x.adjustExponent(expo_sq);
// Round to same precision as argument and rounding mode
x.mode(a.mode());
x.precision(precision);
return x;
}

```

## Recommendation for the square root

Sqrt(y)	Addition/Subtraction	Multiplication	Multiplication half precision	Division
<b>Newton*</b>	1	3		

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

<b>Newton (Brent)</b>	2	2	1	
<b>Newton Division*</b>	1			1
<b>Halley</b>	2	5		

\*) Multiplication with 0.5 is not a full multiplication but is just carried out by adjusting the exponent and therefore just does not count as a 'real' multiplication. Newton's method with division requires the fewest operations, but division is an expensive operator that consistently runs 4-8 times slower than the other methods and can therefore not be recommended.

Based on the performance measure, I recommend:

- 1) Do not use Newton's method with division. Choose the Newton method that avoids division. The division is usually 4-10 times slower than multiplication.
- 2) There is approximately 20% performance gain using Halley over the Newton Method (Brent variation). Newton's method is simpler but requires additional iterations than Halley's method. Halley is more complex per iteration but converges in fewer iterations.
- 3) I recommend using Halley with dynamic precision.
- 4) If you choose Newton, then use the Brent improvement.
- 5) The use of dynamic precision improves the performance by a factor of two for both Newton and the Halley methods.
- 6) As outlined in Appendix A, use guard digits=2 for Newton and Goldschmidt, and guard digits=3 for Halley's method to account for the computational errors of the methods.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## Nroot

Now that we have found a better way of doing the square root, we also need to consider whether we can use a similar technique when dealing with the  $\sqrt[n]{x}$ . By default, we resort to the power function; however, that evaluates to:

$$\sqrt[n]{x} = x^{\frac{1}{n}} = e^{\frac{1}{n} \log_e(x)}$$

Which uses two very expensive and time-consuming functions  $\exp(x)$  and  $\log(x)$ . Instead, we can create a new function  $\text{nroot}(x,n)$  that calculates  $\sqrt[n]{x}$ . Using the same principle as  $\text{sqrt}()$ , the result is a significant speed-up.

As can be seen below, the speed of the  $\text{nroot}()$  is more or less constant regardless of the  $n$ th root, and it is several magnitudes better than the traditional calculation via the  $\text{pow}()$  function.

Let us end the discussion of the  $\text{sqrt}()$  and  $\text{nroot}()$  by devising the Newton formula for the root. It is quite similar to how we obtained the algorithm for the  $\text{sqrt}()$  function. We are trying to find a function for the solution  $x = \sqrt[n]{S} \Rightarrow x^n = S \Rightarrow \frac{1}{x^n} = \frac{1}{S}$

Letting  $y = \frac{1}{S}$  You get:  $f(x) = \frac{1}{x^n} - y = 0$  and  $f'(x) = -nx^{-n-1}$

Using the Newton method, you get:

$$x_{i+1} = x_i - \frac{x_i^{-n} - y}{-nx_i^{-n-1}} \Rightarrow$$

$$x_{i+1} = x_i + \frac{1}{n}(x_i - x_i^{n+1}y) \Rightarrow$$

$$x_{i+1} = x_i + \frac{1}{n}x_i(1 - x_i^n y) \Rightarrow$$

$$x_{i+1} = x_i \frac{1}{n}(n + 1 - x_i^n y) \quad (13)$$

And now  $\sqrt[n]{S} = \frac{1}{x_{i+1}}$

We still have a division  $\frac{1}{n}$  But it is constant, so we can calculate it once before the start of the iteration, avoiding division during the iteration.

We could have done a more direct approach, as we saw for the square root:

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$x = \sqrt[n]{S} \Rightarrow x^n = S \Rightarrow x^n - S = 0$$

You get  $f(x) = x^n - S = 0$  and  $f'(x) = nx^{n-1}$

Using the Newton method, you get:

$$x_{i+1} = x_i - \frac{x_i^n - S}{nx_i^{n-1}} \Rightarrow$$

$$x_{i+1} = x_i - \frac{1}{n} \left( x_i - \frac{S}{x_i^{n-1}} \right) \Rightarrow$$

$$x_{i+1} = \frac{1}{n} \left( (n-1)x_i + \frac{S}{x_i^{n-1}} \right) \quad (14)$$

We end up with an extra division that we need to perform per iteration, so it will be slower than the first version, as we saw when calculating the square root.

There exist other higher-order methods, such as the Halley method, but they will be slower than the Newton method. In the Booth Arbitrary precision library, they did some testing, and the Newton method came out ahead of all other methods. See [8]

As for the nth root algorithm, it can also benefit from using dynamic precision as outlined for both the inverse and the sqrt root functions

Source for `nroot_newton_dynamic()`

```
static float_precision nroot_newton_deep(const float_precision& a, const uintmax_t n)
{
    // Handle NaN
    if (isnan(a))
        return a;
    const float_precision c1(1);
    if (a.iszero() || a == c1 || n == 1)
        return a;
    if (n == 0)
        throw float_precision::domain_error();
    if (a.sign() < 0) {
        if ((n & 1) == 0)
            throw float_precision::domain_error();
        float_precision result = nroot(-a, n);
        result.sign(-1);
        return result;
    }
    if (n == 2)
        return sqrt(a);

    const eptype expo = a.exponent();
    float_precision y(a);
    // For nroot, if y has no fractional mantissa(is a true power of two) and its
    exponent is exactly divisible by n,
```

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

```
// the answer is exact with no iteration at all by just dividing the exponent by n.
// This case is important to handle it fast and accurate without any iteration.
if (y.size() == 1 && (expo % n) == 0) {
    y.exponent(expo / eptype(n));
    return y;
}

const size_t guard = 2;
const size_t precision = a.precision();

y.precision(precision + guard);
const eptype expo_sq = expo / eptype(n);
// Reduce y to [1,2^n);
y.exponent(expo - eptype(n) * expo_sq);
// Do iteration using guard digits higher precision
float_precision fn(n);
fn.precision(precision + guard);

// Get a initial guess using ordinary floating point
double fv = double(y);
fv = pow(fv, 1.0 / n); // set the initial guess with at approx 16 correct digits
fv = 1 / fv;
float_precision x(fv);
fn = c1 / fn;
// fn1 precision is intentionally default; binary ops promote to the
// higher operand precision (precision + guard) automatically
const float_precision fn1(n + 1);
float_precision p, res, r;

// Now iterate using Newton x=x*(-yx^n+(n+1))/n
for (size_t digits = std::min((size_t)32, precision); ; digits = std::min(precision
+ guard, digits * 2))
{
    // Increase precision by a factor of two
    r.precision(digits);
    x.precision(digits);
    p.precision(digits);
    res.precision(digits);

    p = x;
    res = c1;
    // Do x^n
    for (uintmax_t i = n; i > 0; i >>= 1)
    {
        if ((i & 0x1) != 0)
            res *= p; // Odd
        if (i > 1)
            p *= p;
    }
    // Notice y is the original number to root which has the
    // full precision
    r = fn1 - y * res; // (n+1)-yx^n
    r *= fn; // (-yx^n+(n+1))/n
    x *= r; // x=x*(-yx^n+(n+1))/n
    if (digits == precision + guard) // Reach final iteration step in
regards to precision
    {
        r.precision(precision + 1); // round to final precision
        if (r == c1) // break if no improvement
            break;
    }
}

x = 1 / x; // n root of x is now 1/x;
x.exponent(x.exponent() + expo_sq);
// Round to same precision as argument and rounding mode
x.mode(a.mode());
```

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

```
x.precision(precision);  
return x;  
}
```

### ***Recommendation*** $\sqrt[n]{x}$

- 1) Use the Newton method with dynamic precision.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## The Inverse

To handle floating-point division, we rewrite the equation  $a/b$  to  $a \cdot (1/b)$ . Multiplication is a much faster operation than division, so it makes sense to do it this way. Now we only need to figure out how to quickly do a calculation of the inverse of  $(1/b)$ . This same issue affects many microprocessors and early RISC (Reduced Instruction Set CPU) designs that lacked hardware support for the division operator. The traditional method has been used for its simplicity, but higher-order methods exist that we will examine in this chapter.

### *Newton's method for inverse*

We can use a classic Newton iteration using the following algorithm for calculating  $1/b$ :

$$x_{n+1} = x_n(2 - x_n y)$$

Where  $y = b$  and  $x_0 \approx \frac{1}{b}$  (initial guess)

and  $x_n$  converged towards  $\frac{1}{b}$

Algorithm 1

This can also be found in the following way by restating the problem of finding  $\frac{1}{x} = y$ .

Applying it to the Newton method, you get:

Where  $f(x) = y - \frac{1}{x}$ ,  $f'(x) = \frac{1}{x^2}$

$$x_{n+1} = x_n - \frac{y - \frac{1}{x_n}}{\frac{1}{x_n^2}} \Rightarrow$$

$$x_{n+1} = x_n - x_n^2 \left( y - \frac{1}{x_n} \right) \Rightarrow$$

$$x_{n+1} = x_n - x_n(x_n y - 1) \Rightarrow$$

$$x_{n+1} = x_n(2 - x_n y) \tag{15}$$

Notice that the algorithm requires only one subtraction and two multiplications per iteration.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## Example of Newton's method for inverse

To see how this algorithm works, let us find the inverse of 1.6 using an initial start guess of 0.1.

Newton 1/y		
1/y		1.6
y=		1.6
x <sub>0</sub> =		0.1
n	x <sub>n</sub>	Error
1	0.184	4.4E-01
2	0.31383	3.1E-01
3	0.470078	1.5E-01
4	0.586598	3.8E-02
5	0.622641	2.4E-03
6	0.624991	8.9E-06
7	0.625	1.3E-10
8	0.625	0.0E+00

After eight iterations, the difference between the iteration and the built-in division operator is zero, and the result of 1/1.6 is 0.625.

Now the only question that remains is how to find a suitable starting point for the iteration, since we cannot perform an initial division as the guess of 1/b. Instead, we look at how our arbitrary precision number is built up.  $i_1$  is the one-digit integer, and  $f_n$  is the fractional digits,  $e_p$  is the exponent power in base 2.

$$\frac{1}{b} = \frac{1}{i_1 \cdot f_n 2^{e_p}} = \frac{1}{i_1 \cdot f_n} 2^{-e_p}$$

We can extract the exponent portion and find the inverse  $\frac{1}{i_1 \cdot f_n}$  and then multiply the result by  $2^{-e_p}$  to find our inverse of 1/b. Extracting the exponent will leave us with a number [1,2). Since we have the support of the hardware division using the IEEE754 standard (64-bit floating-point number), we can get our initial start guess with approximately 15-16 digits of accuracy, and then begin iterating towards higher accuracy. If you do not have access to IEEE754, you can use a lookup table to find a suitable starting point.

The Newton method for division is very fast and has quadratic convergence, meaning that each iteration doubles the number of correct digits. To set this into perspective, assume we have a number with 128 digits ( $2^7$ ), and we start with approximately  $2^4$  correct digits, then we should expect only three iterations to get our result. For 1,000 digits, it will require approximately six iterations, and for 1,000,000 digits, it will require approximately sixteen iterations.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## Brent improvement

Brent [7] points out that you can improve the Newton algorithm by iterating using:

$$x_{n+1} = x_n + x_n(1 - yx_n) \quad (16)$$

This needs one extra addition, however, as Brent [7] points out, you can do the multiplication of  $x_n(1 - yx_n)$  using only half the precision. Overall, you save some computational power by using Brent's suggestion

## Iteration with dynamic precision

Since we are applying the Newton method, we can use the same technique for square root and root calculations with dynamic precision.

We can use that information: instead of starting the first couple of iterations with a precision of thousands of digits, we could start with a lower precision and gradually increase it until, in the last iteration, we perform all calculations with the required number of digits. Initially, our first guess is approx. 15-16 correct digits. We know that a Newton iteration doubles the number of correct digits for every iteration. We will start with 32 digits of precision and, after each iteration, double the precision. E.g., 64 digits in the next iteration, 128 digits in the following iteration, etc., until we have reached the required final precision.

## Source inverse\_newton\_dynamic()

```
static float_precision inverse_newton_dynamic(const float_precision& a)
{
    const size_t guard = 2;
    const size_t precision=a.precision();
    const eptype expo=a.exponent();
    const float_precision c1(1), c2(2);
    double fx;
    float_precision r, x, y(a);

    if (a.iszero() == true)
        throw float_precision::divide_by_zero();
    // if a is a true power of 2 then we dont need to iterate but just
reverse the exponent and return
    if (a.size() == 1)
    {
        y.exponent(-y.exponent());
        return y;
    }

    y.precision(precision + guard);
    // find the inverse of y without exponent and adjust for exponent later
    y.exponent(0); // y is in the interval [1..2]
    // Do iteration using guard digits higher precision
    x.precision(precision + guard);

    // Get a initial guess using ordinary floating point
```

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

```
fx = 1/(double)y;
x = float_precision(fx);

// Now iterate using Newton x=x(2-yx)
for (digits = std::min((size_t)32, precision); ; digits =
std::min(precision + guard, digits * 2))
{
    // Increase precision by a factor of two for the working
variable r & x.
    r.precision(digits);
    x.precision(digits);
    r = y;                // y
    r *= x;               // xy
    r = c2 - r;           // 2-xy
    x *= r;               // x=x(2-xy)
    if (digits == precision + guard)// Reach final iteration step in
regards to precision
    {
        r.precision(precision + 1);// round to final precision
        if (r == c1)           // break if no improvement
            break;
        r.precision(precision + guard);
    }
}

// Reapply exponent, mode, and precision
x.adjustExponent(-expo);
x.mode(a.mode());
x.precision(precision + 1);
return x;
}
```

### *Halley method for inverse*

A higher-order Halley method exists with a cubic convergence rate. We can iterate toward the inverse by using the following:

$$x_{n+1} = x_n + x_n(1 - yx_n) + x_n(1 - yx_n)^2 \quad (17)$$

We notice that, compared to the Newton-Brent method, we have an extra addition of  $x_n(1 - yx_n)^2$  which adds one extra addition and one extra multiplication.

Alternatively, it can be stated as:

$$\begin{aligned} z_n &= 1 - yx_n \\ x_{n+1} &= x_n + x_n z_n + x_n z_n^2 \end{aligned} \quad (18)$$

It can be found using Householder's 2<sup>nd</sup> order method and is often called the Halley Method for inverse computation.

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{f(x_n)^2 f''(x_n)}{2f'(x_n)^3} \quad (19)$$

Where  $f(x) = y - \frac{1}{x}$ ,  $f'(x) = \frac{1}{x^2}$ ,  $f''(x) = -\frac{2}{x^3}$

This yields:

$$x_{n+1} = x_n - \frac{y - \frac{1}{x_n}}{\frac{1}{x_n^2}} - \frac{\left(y - \frac{1}{x_n}\right)^2 \left(-\frac{2}{x_n^3}\right)}{2\left(\frac{1}{x_n^2}\right)^3} \Rightarrow$$

$$x_{n+1} = x_n + x_n^2 \left(\frac{1}{x_n} - y\right) + x_n^3 \left(\frac{1}{x_n} - y\right)^2 \Rightarrow$$

$$x_{n+1} = x_n + x_n(1 - yx_n) + x_n(1 - yx_n)^2 \quad (20)$$

This method will require one subtraction, two additions, and four multiplications.

We could be tempted to factor out the  $x_{n-1}$  as outlined below:

$$\begin{aligned} z_n &= 1 - yx_n \\ x_{n+1} &= x_n(1 + z_n + z_n^2) \end{aligned} \quad (21)$$

This will require three additions/subtractions and three multiplications. However, all multiplication must be carried out with full precision.

The Halley convergence rate means that for each iteration, you triple the number of correct digits, requiring fewer iterations than the Newton method.

### Example of the Halley convergence method for inverse

To see how this algorithm works, let us find the inverse of 1.6 using an initial start guess of 0.1.

Halley		
1/y	cubic convergence	
1/y	1.6	
Y=	1.6	
x <sub>0</sub> =	0.1	
n	x <sub>n</sub>	Error
1	0.25456	3.7E-01
2	0.494865	1.3E-01
3	0.619358	5.6E-03

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

4	0.625	4.6E-07
5	0.625	0.0E+00

After five iterations, the difference between the iteration and the built-in division operator is zero, and the result of  $1/1.6$  is 0.625.

### Source inverse\_Halley\_dynamic()

```
static float_precision inverse_halley_dynamic(const float_precision& a)
{
    const size_t guard = 2;
    const size_t precision = a.precision();
    const eptype expo = a.exponent();
    const intmax_t limit = -(intmax_t)((precision + 1) * log2(10)) - 1;
    const float_precision c1(1), c2(2);
    double fx = precision + guard < 16 ? 1.0 : log(precision + guard) - log(16);
    float_precision z, s, x, y(a);

    if (a.iszero() == true)
        throw float_precision::divide_by_zero();

    // if a is a true power of 2 then we dont need to iterate but just reverse the
    exponent and return
    if (a.size() == 1)
    {
        y.exponent(-y.exponent());
        return y;
    }

    // find the inverse of y without exponent and adjust for exponent later
    y.exponent(0); // y is in the interval [1..2]
    // Do iteration using guard digits higher precision
    x.precision(precision + guard);
    // Get a initial guess using ordinary floating point
    fx = 1 / (double)y;
    x = float_precision(fx);
    digits = 48;
;
    // Now iterate using 3rd order  $x=x+x(1-xy)+x(1-yx)^2$ 
    for (digits = std::min(digits, precision); ; digits = std::min(precision + guard,
digits * 3))
    {
        // Increase precision by a factor of two for the working variable s & x.
        s.precision(digits);
        x.precision(digits);
        z.precision(digits);

        z = c1 - y * x; // (1-yx)y
        s = x * z;      // x(1-yx)
        x += s;        // x = x + x(1 - yx)
        if (2 * z.exponent() > limit)
            x += s * z; //  $x=x+x(1-yx)+x(1-yx)^2$ 

        if (digits == precision + guard)
        { // Only check for stopping criteria at the last iteration to avoid
unnecessary calculation
            if (z.iszero() || 2 * z.exponent() < limit)
            {
                break; // stop iteration
            }
        }
    }

    // Reapply exponent, mode, and precision
}
```

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

```
x.adjustExponent(-expo);  
x.mode(a.mode());  
x.precision(precision + 1);  
return x;  
}
```

The third order with cubic convergence is similar in structure to the Newton method.

### ***Goldschmidt inverse***

The Goldschmidt inverse  $1/d$  is simply a special case of Goldschmidt division where  $n=1$ . Substituting  $n=1$  into the Goldschmidt division iteration gives:

```
N0 = 1, D0 = d (scaled to [0.5, 1))  
Fi = 2 - Di  
Ni+1 = Ni · Fi  
Di+1 = Di · Fi
```

As  $D_i$  converges to 1,  $N_i$  converges to  $1/d$ . The exponent is restored at the end exactly as in the division case. No structural change to the algorithm is required, setting  $n=1$  simply means the numerator sequence starts at 1 and tracks the accumulating product of all  $F_i$  factors, which is precisely  $1/d$ .

Despite its simplicity, the Goldschmidt inverse is 2–3 times slower than the Newton or cubic Householder inverse in software. The reason is the same as for division: each Goldschmidt iteration requires two full-precision multiplications, one for  $N_i$  and one for  $D_i$ , whereas a Newton inverse iteration requires only one. The  $D_i$  sequence is essentially wasted work in software, since its sole purpose is to signal convergence. In CPU hardware, both multiplications execute in parallel via pipelining, which eliminates this penalty entirely. In software, there is no such parallelism, and the dynamic precision scheduling that gives Newton its roughly fourfold performance advantage is also unavailable to Goldschmidt, since both  $N_i$  and  $D_i$  must be maintained at consistent precision throughout. For these reasons, the Goldschmidt inverse is not used in this library.

### Source Goldschmidt inverse

```
static float_precision goldschmidt_inverse(const float_precision& D)  
{  
    const size_t prec = D.precision();  
  
    if (isnan(D)) return FP_QUIET_NAN;  
    if (D.iszero())  
        throw float_precision::divide_by_zero();  
  
    // Record sign and work with magnitude  
    int neg = D.sign();  
    float_precision fD(abs(D));  
    fD.precision(prec);  
  
    // Normalize: extract exponent and force mantissa into [0.5, 1)  
    eptype eD = fD.exponent();  
    fD.exponent(-1); // Now fD is in [0.5, 1)
```

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

```
// Goldschmidt iteration: drive Di -> 1, Ni -> 1/fD
float_precision Ni(1, prec);
float_precision Di(fD);

const float_precision c2(2);
int i;
for (i = 0; ; ++i) {
    float_precision F(c2 - Di);
    Ni *= F;
    float_precision DoId(Di);
    Di *= F;
    if (DoId == Di)
        break;
}
loopcnt_gold = i;
// 1/D = Ni * 2^(-eD-1)
Ni.exponent(Ni.exponent() - eD-1);
return neg < 0 ? -Ni : Ni;
}
```

### ***Performance:***

The Y-axis is in milliseconds, and the X-axis is the number of decimal digits ranging from 200,000 digits to 3.5M digits.

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

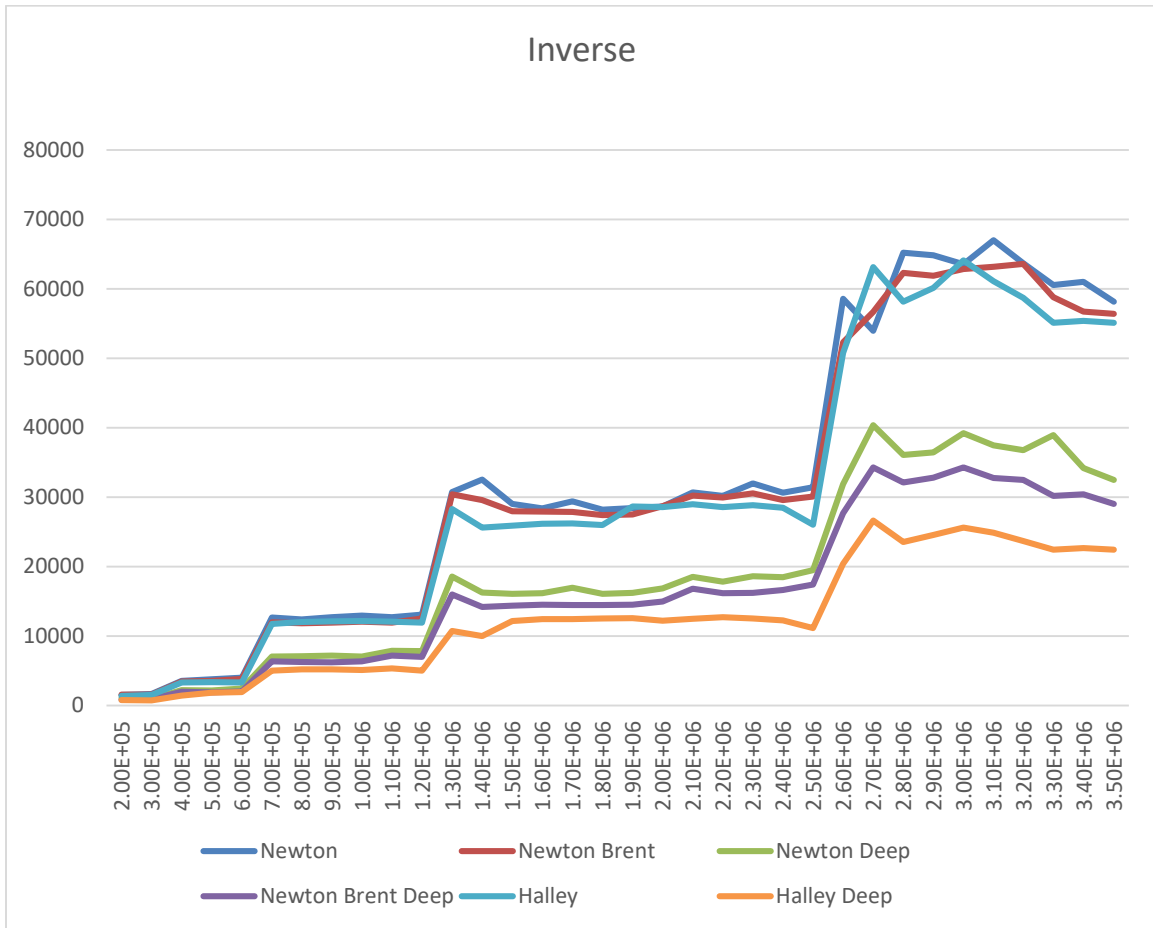


Figure 2

As can be seen, the 3<sup>rd</sup> order Halley method with dynamic (deep in the chart) outperforms the other methods. Comparing dynamic versus ordinary, we see that the performance gain is approx. a factor of two.

### Recommendation Inverse

The above new methods show that you can achieve significant performance improvements using dynamic-precision iteration. In addition, using the Brent [7] enhancement can further reduce the workload.

	Addition/Subtraction	Multiplication	Multiplication half precision
<b>Newton</b>	1	2	
<b>Newton (Brent)</b>	2	1	1
<b>Halley</b>	3	3	
<b>Halley (Brent)</b>	3	1	3

Based on the performance measure, recommend:

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

- 1) Use the third-order Halley method using Brent's recommendation.
- 2) The use of dynamic precision improves the performance by a factor of two for both Newton and the third-order Halley method.

## Reference

- 1) Arbitrary precision library package. [Arbitrary Precision C++ Packages \(hvks.com\)](http://hvks.com)
- 2) Numerical recipes in C++, 3<sup>rd</sup> edition, Cambridge University Press, New York, NY 2007
- 3) Wilkinson, J H, Rounding errors in Algebraic Processes, Prentice-Hall Inc, Englewood Cliffs, NJ 1963
- 4) Methods of Computing square roots; May 17-2013; [http://en.wikipedia.org/wiki/Methods\\_of\\_computing\\_square\\_roots](http://en.wikipedia.org/wiki/Methods_of_computing_square_roots)
- 5) Borwein, Pi and the AGM, Volume 4, John Willey & Sons Inc, New York, NY 1998
- 6) The Yacas book of algorithms, Version 1.3.3, April 1 2013 by the Yacas team
- 7) Richard Brent & Paul Zimmermann, Modern Computer Arithmetic, Version 0.5.9 17 October 2010; <http://maths-people.anu.edu.au/~brent/pd/mca-cup-0.5.9.pdf>
- 8) Boost, performance comparison of nrooth algorithm [https://www.boost.org/doc/libs/1\\_73\\_0/libs/math/doc/html/math\\_toolkit/root\\_comparison/root\\_n\\_comparison.html](https://www.boost.org/doc/libs/1_73_0/libs/math/doc/html/math_toolkit/root_comparison/root_n_comparison.html)

## Appendix A: Guard Digit Analysis for Square Root Algorithms

This appendix derives tight theoretical lower bounds on the number of guard digits required to guarantee that each square root algorithm returns a result within one ULP (unit in the last place) of the true mathematical result, at the requested decimal precision  $p$ . The analysis covers Newton's method (with and without division), Newton's method with Brent's half-precision improvement, and Halley's method. All three algorithms operate on  $y \in [1, 4)$  after exponent extraction. In the past, I have relied on an empirical choice of 2 as extra guard digits for the square root algorithm, which worked but lacked a rigorous analysis of the error propagation through the algorithm.

### A.1 Notation and Setup

Let  $p$  be the target precision in decimal digits. The unit roundoff at precision  $p$  is:

$$\varepsilon = 10^{-p}$$

We perform all internal computation at working precision  $p + g$  digits, where  $g$  is the number of guard digits to be determined. The internal unit roundoff is therefore:

$$\varepsilon' = 10^{-(p+g)}$$

Throughout,  $y \in [1, 4)$  (normalised form after exponent extraction), and  $x^* = y^{-1/2}$  is the exact reciprocal square root. The relative error at iteration  $n$  is defined as:

$$e_n = x_n / x^* - 1, \text{ so } x_n = x^*(1 + e_n)$$

The 1-ULP goal requires the final absolute error in  $\sqrt{y}$  to satisfy:

$$|\delta_{final}| < \frac{1}{2} \cdot 10^{-p} \tag{A.1}$$

### A.2 Newton's Method Without Division

#### A.2.1 Algorithm

The iteration computes the reciprocal square root via:

$$x_{n+1} = \frac{1}{2} x_n (3 - y x_n^2) \tag{A.2}$$

and recovers the square root via the single final multiplication:

$$\sqrt{y} = y \cdot x^* \tag{A.3}$$

#### A.2.2 Convergence in Exact Arithmetic

Substituting  $x_n = x^*(1 + e_n)$  into (A.2) and using  $y x^{*2} = 1$ :

$$x_{n+1} = \frac{1}{2} x^*(1+e_n)(3 - (1+e_n)^2)$$

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$\begin{aligned}
 &= \frac{1}{2} x^*(1+e_n)(2 - 2e_n - e_n^2) \\
 &= x^*(1 - \frac{1}{2} e_n^2 - \frac{1}{2} e_n^3)
 \end{aligned}$$

Therefore, the relative error satisfies:

$$e_{n+1} = -\frac{1}{2} e_n^2 + O(e_n^3) \tag{A.4}$$

This confirms quadratic convergence:  $|e_{n+1}| \approx \frac{1}{2} |e_n|^2$ . The dynamic precision schedule (doubling digits each iteration) matches this rate exactly, ensuring each iteration operates at just enough precision to be non-redundant.

### A.2.3 Rounding Error Per Iteration

Each iteration performs the following floating-point operations:

- Step 1:  $r_1 = y \cdot x_n^2$  (two multiplications)
- Step 2:  $r_2 = 3 - r_1$  (one subtraction)
- Step 3:  $x_{n+1} = \frac{1}{2} x_n \cdot r_2$  (one multiplication;  $\frac{1}{2}$  is free via exponent shift)

Each floating-point operation contributes at most  $\epsilon'$  relative error. For Step 2, note that  $r_1 = y x_n^2 \approx 1$  near convergence, so  $3 - r_1 \approx 2$ . There is no catastrophic cancellation since the result is well away from zero. The accumulated rounding contribution across all four operations of one iteration is bounded by:

$$|\delta_{round, iter}| \leq 4\epsilon' = 4 \cdot 10^{-(p+g)} \tag{A.5}$$

### A.2.4 Final Multiplication Error

The recovery step  $\sqrt{y} = y \cdot x_{n+1}$  introduces one further rounding error of  $\epsilon'$ . Combining with (A.5), the total rounding budget is:

$$|\delta_{total}| \leq 5\epsilon' = 5 \cdot 10^{-(p+g)} \tag{A.6}$$

### A.2.5 Guard Digit Requirement

Applying the 1-ULP goal (A.1) to (A.6):

$$\begin{aligned}
 5 \cdot 10^{-(p+g)} &\leq \frac{1}{2} \cdot 10^{-p} \\
 10^g &\geq 10 \\
 g &\geq 1
 \end{aligned} \tag{A.7}$$

Theoretical minimum:  $g = 1$  guard digit for Newton without division. A.3 Newton's Method with Brent's Half-Precision Improvement.

### A.3.1 Algorithm

Brent's reformulation of the Newton iteration is:

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$x_{n+1} = x_n + (x_n / 2)(1 - y x_n^2) \tag{A.8}$$

This is mathematically identical to (A.2) but computationally distinct: Brent observes that the product  $(x_n / 2)(1 - y x_n^2)$  can be evaluated at half the working precision, since near convergence  $(1 - y x_n^2) \approx 0$  and the product is small. This saves roughly one full-precision multiplication at the cost of one extra addition.

### A.3.2 Rounding Error Analysis

Let  $p' = (p + g)/2$  denote the half-precision digit count. The operations are:

- Step 1:  $r = 1 - y x_n^2$  (two multiplications + one subtraction; full precision)
- Step 2:  $s = (x_n / 2) \cdot r$  (one multiplication at half precision  $p'$ )
- Step 3:  $x_{n+1} = x_n + s$  (one addition; full precision)

The half-precision step contributes an absolute error of  $|s| \cdot 10^{-p'}$ . Near the final iteration,  $|r| = |1 - y x_n^2| \leq C \cdot 10^{-(p+g)}$  (the convergence criterion ensures  $r$  is small), so  $|s| \leq C \cdot 10^{-(p+g)} / 2$ . The absolute error from the half-precision step is therefore:

$$|\delta_{Brent}| \leq (C/2) \cdot 10^{-(p+g)} \cdot 10^{-p'} = (C/2) \cdot 10^{-(p+g)} \cdot 10^{-(p+g)} \tag{A.9}$$

For the final iteration  $C$  is itself of order  $10^{-(p+g)}$  (the residual is at the level of machine precision), making (A.9) negligibly small, far below 1 ULP. The dominant error terms remain those from the three full-precision operations in Step 1 and the final multiplication  $\sqrt{y} = y \cdot x_{n+1}$ , giving the same total of  $5\epsilon'$  as in (A.6).

### A.3.3 Guard Digit Requirement

The bound is identical to Newton without division:

$$g \geq 1 \tag{A.10}$$

Theoretical minimum:  $g = 1$  guard digit for Newton–Brent. The half-precision trick does not relax the guard digit requirement; it reduces computation without trading precision safety.

## A.4 Halley's Method (Cubic Convergence)

### A.4.1 Algorithm

Halley's method for the reciprocal square root uses the substitution  $z_n = y x_n^2$ :

$$x_{n+1} = (x_n / 8)(15 - z_n(10 - 3z_n)) \tag{A.11}$$

with the same final recovery step  $\sqrt{y} = y \cdot x_{n+1}$ .

### A.4.2 Convergence in Exact Arithmetic

With  $x_n = x^*(1 + e_n)$ , we have  $z_n = y x_n^2 = (1 + e_n)^2$ . Setting  $\delta = z_n - 1 = 2e_n + e_n^2$  and expanding (A.11) to third order:

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$\begin{aligned}
 10 - 3z_n &= 10 - 3(1+\delta) = 7 - 3\delta \\
 z_n(10-3z_n) &= (1+\delta)(7-3\delta) = 7 + 4\delta - 3\delta^2 \\
 15 - z_n(10-3z_n) &= 8 - 4\delta + 3\delta^2
 \end{aligned}$$

Therefore:

$$x_{n+1} = x^*(1+e_n) \cdot (1 - \frac{1}{2}\delta + \frac{3}{8}\delta^2)$$

Substituting  $\delta = 2e_n + e_n^2$  and retaining terms to  $O(e_n^3)$ :

$$e_{n+1} = -3/2 \cdot e_n^3 + O(e_n^4) \tag{A.12}$$

This confirms cubic convergence:  $|e_{n+1}| \approx (3/2)|e_n|^3$ . The dynamic precision schedule for Halley triples the digit count each iteration, matching this rate.

### A.4.3 Cancellation Analysis

A key concern for Halley's method is whether any intermediate subtraction suffers catastrophic cancellation that would amplify rounding errors. Checking each subtraction near convergence ( $z_n \approx 1$ ):

$$\begin{aligned}
 10 - 3z_n &\approx 10 - 3 = 7 \quad \checkmark \text{ no cancellation} \\
 15 - z_n(10-3z_n) &\approx 15 - 7 = 8 \quad \checkmark \text{ no cancellation}
 \end{aligned}$$

Neither subtraction involves quantities that nearly cancel. The rounding error analysis can therefore proceed without catastrophic-cancellation correction terms.

### A.4.4 Rounding Error Per Iteration

The operations and their rounding costs are:

Step 1: $z_n = y \cdot x_n^2$	(2 multiplications)
Step 2: $t_1 = 3 \cdot z_n$	(1 multiplication)
Step 3: $t_2 = 10 - t_1$	(1 subtraction)
Step 4: $t_3 = z_n \cdot t_2$	(1 multiplication)
Step 5: $t_4 = 15 - t_3$	(1 subtraction)
Step 6: $x_{n+1} = x_n \cdot t_4 / 8$	(1 multiplication; /8 is free)
Step 7: $\sqrt{y} = y \cdot x_{n+1}$	(1 multiplication; final recovery)

Total: 6 multiplications + 2 subtractions = 8 floating-point operations, each contributing at most  $\epsilon'$  relative error. No catastrophic cancellation occurs (Section A.4.3), so:

$$|\delta_{total}| \leq 8\epsilon' = 8 \cdot 10^{-(p+g)} \tag{A.13}$$

### A.4.5 Guard Digit Requirement

Applying the 1-ULP goal (A.1) to (A.13):

$$\begin{aligned}
 8 \cdot 10^{-(p+g)} &\leq \frac{1}{2} \cdot 10^{-p} \\
 10^g &\geq 16
 \end{aligned}$$

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$g \geq \log_{10}(16) \approx 1.204 \tag{A.14}$$

Rounding up to the nearest integer:

$$g \geq 2 \tag{A.15}$$

Theoretical minimum:  $g = 2$  guard digits for Halley’s method. Using  $\text{guard}=3$  is recommended, providing a  $10\times$  safety factor.

## A.5 Recommendations

The table below consolidates the theoretical bounds derived in Sections A.2–A.4. The ‘safety factor’ column shows how much headroom the current implementation provides beyond the proven minimum.

Method	Ops per iteration	Theoretical min $g$	Recommended	Safety factor
Newton (no division)	4 ops + 1 final	1	2	$10\times$
Newton–Brent	4 ops + 1 final	1	2	$10\times$
Halley	7 ops + 1 final	2	3	$10\times$

### Notable Observations

1. Newton implementations with  $\text{guard} = 2$  provides  $10\times$  more headroom than strictly required.
2. Halley’s method requires more guard digits than Newton because it performs more floating-point operations per iteration (8 vs. 5). Higher convergence order does *not* reduce the rounding budget per step.
3. The absence of catastrophic cancellation in both methods (verified in Sections A.2.3 and A.4.3) is what keeps the bounds tight. If future algorithm variants introduce subtractions between nearly-equal quantities, the guard digit count must be revisited.
4. Brent’s half-precision trick does not relax the guard digit requirement. The reduction in computation comes from the small magnitude of the correction term near convergence, not from any reduction in sensitivity to precision.

## Appendix B: Guard Digit Analysis for Inverse (1/y) Algorithms

This appendix derives tight theoretical lower bounds on the number of guard digits required to guarantee that each inverse algorithm returns a result within one ULP (unit in the last place) of the true result  $1/y$ , at the requested decimal precision  $p$ . Four algorithms are analyzed: Newton's method, Newton with Brent's half-precision improvement, the cubic (third-order) Householder method, and the Goldschmidt method. All algorithms operate on  $y \in [1, 2)$  after exponent extraction.

### B.1 Notation and Setup

Let  $p$  be the target precision in decimal digits. The unit roundoff at precision  $p$  is:

$$\varepsilon = 10^{-p}$$

All internal computation is performed at working precision  $p + g$  digits (where  $g$  is the guard digit count to be determined), giving internal unit roundoff:

$$\varepsilon' = 10^{-(p+g)}$$

Throughout,  $y \in [1, 2)$  and  $x^* = 1/y$  is the exact inverse. The relative error at iteration  $n$  is:

$$e_n = x_n \cdot y - 1, \text{ so } x_n = x^*(1 + e_n)$$

The 1-ULP goal requires the final absolute error to satisfy:

$$|\delta_{final}| < \frac{1}{2} \cdot 10^{-p} \tag{B.1}$$

Since  $x^* = 1/y$  and  $y \in [1, 2)$ , the result lies in  $(0.5, 1]$ . Absolute and relative errors are therefore within a factor of 2 of each other, so relative-error bounds translate directly to absolute-error bounds without additional constants.

### B.2 Newton's Method for Inverse

#### B.2.1 Algorithm

The standard Newton iteration for  $1/y$  is derived from  $f(x) = y - 1/x = 0$ :

$$x_{n+1} = x_n(2 - y \cdot x_n) \tag{B.2}$$

Each iteration requires one subtraction and two multiplications.

#### B.2.2 Convergence in Exact Arithmetic

Substituting  $x_n = x^*(1 + e_n)$  into (B.2) and using  $x^*y = 1$ :

$$\begin{aligned} x_{n+1} &= x^*(1+e_n)(2 - y \cdot x^*(1+e_n)) \\ &= x^*(1+e_n)(2 - (1+e_n)) \\ &= x^*(1+e_n)(1 - e_n) \\ &= x^*(1 - e_n^2) \end{aligned}$$

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

Therefore:

$$e_{n+1} = -e_n^2 + O(e_n^3) \quad (\text{B.3})$$

This confirms quadratic convergence:  $|e_{n+1}| = |e_n|^2$ . Note the leading coefficient is 1 (compared to 1/2 for the square root Newton iteration), meaning the error contraction is slightly less aggressive at the same error magnitude. The dynamic precision schedule doubles the digit count each iteration, matching this rate.

## B.2.3 Cancellation Analysis

The subtraction  $2 - y \cdot x_n$  requires attention. Near convergence,  $y \cdot x_n \approx 1$ , so  $2 - y \cdot x_n \approx 1$ . The result is well away from zero: no catastrophic cancellation occurs. The subtraction result has magnitude  $\approx 1$ , and its relative rounding error is simply  $\varepsilon'$ .

## B.2.4 Rounding Error Per Iteration

The three floating-point operations and their error contributions are:

- Step 1:  $r = y \cdot x_n$  (1 multiplication  $\rightarrow$  relative error  $\leq \varepsilon'$ )
- Step 2:  $s = 2 - r$  (1 subtraction  $\rightarrow$  relative error  $\leq \varepsilon'$ ; no cancellation)
- Step 3:  $x_{n+1} = x_n \cdot s$  (1 multiplication  $\rightarrow$  relative error  $\leq \varepsilon'$ )

Error propagation through Steps 1–3, accounting for the chained relative errors, gives a total relative rounding contribution per iteration bounded by:

$$|\delta_{round,iter}| \leq 3\varepsilon' = 3 \cdot 10^{-(p+g)} \quad (\text{B.4})$$

## B.2.5 Guard Digit Requirement

Applying the 1-ULP goal (B.1):

$$\begin{aligned} 3 \cdot 10^{-(p+g)} &\leq \frac{1}{2} \cdot 10^{-p} \\ 10^g &\geq 6 \\ g &\geq \log_{10}(6) \approx 0.778 \end{aligned} \quad (\text{B.5})$$

Rounding up:

$$g \geq 1 \quad (\text{B.6})$$

Theoretical minimum:  $g = 1$  guard digit for Newton's inverse. The previous implementation uses `guard = 5`, providing a safety factor of  $10^4 = 10,000\times$  beyond the theoretical minimum. This is extremely conservative; `guard = 2` (100 $\times$  margin) would be more than sufficient.

## B.3 Newton's Method with Brent's Half-Precision Improvement

### B.3.1 Algorithm

Brent's reformulation separates the correction term:

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$x_{n+1} = x_n + x_n(I - y \cdot x_n) \tag{B.7}$$

This is mathematically identical to (B.2). Brent's observation is that the product  $x_n \cdot (1 - y \cdot x_n)$  can be evaluated at half the working precision, because near convergence  $(1 - y \cdot x_n)$  is small and the correction term is a higher-order quantity.

### B.3.2 Rounding Error Analysis

Let  $p' = (p + g)/2$  be the half-precision digit count. The operations are:

$$\text{Step 1: } r = 1 - y \cdot x_n \quad (1 \text{ mul} + 1 \text{ sub at full precision } p+g)$$

$$\text{Step 2: } s = x_n \cdot r \quad (1 \text{ multiplication at half precision } p')$$

$$\text{Step 3: } x_{n+1} = x_n + s \quad (1 \text{ addition at full precision } p+g)$$

The critical question is the contribution of Step 2 at half precision. Near the final iteration, the convergence criterion ensures  $|r| = |1 - y \cdot x_n| \leq C \cdot \epsilon'$  (the residual is at machine precision). Therefore  $|s| \leq C \cdot x_n^* \cdot \epsilon'$ , and the absolute error from the half-precision multiplication is:

$$|\delta_{\text{Brent}}| \leq |s| \cdot 10^{-p'} \leq C \cdot x_n^* \cdot \epsilon' \cdot 10^{-p'} \tag{B.8}$$

With  $\epsilon' = 10^{-(p+g)}$  and  $10^{-p'} = 10^{-(p+g)/2}$ , the product is  $10^{-3(p+g)/2}$ , which for any  $p \geq 1$  is far below 1 ULP. The dominant rounding contributions therefore come from Steps 1 and 3 (full precision), giving the same  $3\epsilon'$  bound as standard Newton.

### B.3.3 Guard Digit Requirement

Identical to Newton:

$$g \geq 1 \tag{B.9}$$

Theoretical minimum:  $g = 1$  guard digit for Newton–Brent inverse. The half-precision saving does not relax the guard digit requirement.

## ***B.4 Cubic (Third-Order) Householder Method for Inverse***

### B.4.1 Algorithm

The cubic inverse iteration is derived from Householder's second-order method applied to  $f(x) = y - 1/x$ . Using the substitution  $z_n = 1 - y \cdot x_n$ , the iteration takes the form:

$$z_n = 1 - y \cdot x_n \tag{B.10}$$

$$x_{n+1} = x_n(I + z_n + z_n^2) \tag{B.11}$$

This requires one subtraction (computing  $z_n$ ), two multiplications ( $y \cdot x_n$  and  $z_n^2$ ), two additions ( $1 + z_n + z_n^2$ ), and one final multiplication ( $x_n \cdot (1 + z_n + z_n^2)$ ): four multiplications and three addition/subtraction operations in total.

### B.4.2 Convergence in Exact Arithmetic

With  $x_n = x^*(1 + e_n)$ , we have  $y \cdot x_n = 1 + e_n$ , so  $z_n = -e_n$ . Substituting into (B.11):

$$x_{n+1} = x^*(1 + e_n)(1 + (-e_n) + (-e_n)^2)$$

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$\begin{aligned}
 &= x \cdot (1 + e_n)(1 - e_n + e_n^2) \\
 &= x \cdot (1 - e_n^3 + O(e_n^4))
 \end{aligned}$$

Therefore:

$$e_{n+1} = -e_n^3 + O(e_n^4) \tag{B.12}$$

This confirms cubic convergence:  $|e_{n+1}| = |e_n|^3$ . The dynamic precision schedule triples the digit count each iteration, matching this convergence rate.

### B.4.3 Cancellation Analysis

Two subtractions require scrutiny:

$1 - y \cdot x_n$  (computing  $z_n$ ): Near convergence  $y \cdot x_n \approx 1$ , so  $z_n \approx 0$ .

This is a genuine catastrophic cancellation risk: the leading significant digits of  $y \cdot x_n$  and 1 cancel, and only the error term survives. The absolute value of  $z_n$  is of order  $|e_n|$ , and its relative rounding error can be large. However, this does not affect the final result because  $z_n$  itself is a small correction term. The absolute error in  $z_n$  from this subtraction is at most  $\epsilon'$  (in absolute terms, since both operands have magnitude  $\approx 1$ ). This absolute error propagates directly into the correction  $x_n \cdot (z_n + z_n^2)$ , contributing at most  $x \cdot \epsilon'$  to the final result — i.e. one unit of  $\epsilon'$ .

$1 + z_n + z_n^2$ : Near convergence  $z_n \approx 0$ , so the sum  $\approx 1$ . No cancellation.

### B.4.4 Rounding Error Per Iteration

The seven floating-point operations and their contributions are:

- Step 1:  $r1 = y \cdot x_n$  (1 mul)
- Step 2:  $z_n = 1 - r1$  (1 sub; absolute error  $\leq \epsilon'$ , cancellation noted above)
- Step 3:  $r2 = z_n^2$  (1 mul)
- Step 4:  $r3 = 1 + z_n + r2$  (2 adds)
- Step 5:  $x_{n+1} = x_n \cdot r3$  (1 mul)

Total: 4 multiplications, 1 subtraction, 2 additions = 7 operations. The cancellation in Step 2 produces an absolute error in  $z_n$  of at most  $\epsilon'$ , which after multiplication by  $x_n$  in Step 5 contributes  $x \cdot \epsilon'$ . All other steps contribute at most  $\epsilon'$  each relative to the current result. Accumulating all contributions:

$$|\delta_{total}| \leq 7\epsilon' = 7 \cdot 10^{-(p+g)} \tag{B.13}$$

### B.4.5 Guard Digit Requirement

Applying the 1-ULP goal (B.1):

$$7 \cdot 10^{-(p+g)} \leq \frac{1}{2} \cdot 10^{-p}$$

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$10^g \geq 14$$

$$g \geq \log_{10}(14) \approx 1.146 \tag{B.14}$$

Rounding up:

$$g \geq 2 \tag{B.15}$$

Theoretical minimum:  $g = 2$  guard digits for the cubic inverse. The previous implementation uses `guard = 5`, providing a safety factor of  $10^3 = 1,000\times$ . This is very conservative; `extra = 3` ( $10\times$  margin) would be entirely sufficient.

## B.5 Goldschmidt Method

### B.5.1 Algorithm

The Goldschmidt method computes  $q = N/D$  by simultaneously scaling both numerator and denominator by a sequence of factors  $f_i$  chosen so that  $D_i \rightarrow 1$ . Each step is:

$$f_{i+1} = 2 - D_i \tag{B.16}$$

$$N_{i+1} = N_i \cdot f_{i+1} \tag{B.17}$$

$$D_{i+1} = D_i \cdot f_{i+1} \tag{B.18}$$

For the inverse  $1/y$  (i.e.,  $N = 1$ ,  $D = y$ ), the initial scaling is not needed since  $y \in [1, 2)$  already satisfies the required range  $0 < D < 2$ . Each iteration requires two multiplications and one subtraction.

### B.5.2 Convergence in Exact Arithmetic

Define the error in the denominator as  $\delta_i = 1 - D_i$  (so  $\delta_i \rightarrow 0$ ). Then  $f_{i+1} = 2 - D_i = 1 + \delta_i$  and:

$$D_{i+1} = D_i \cdot f_{i+1} = (1 - \delta_i)(1 + \delta_i) = 1 - \delta_i^2$$

Therefore:

$$\delta_{i+1} = \delta_i^2 \tag{B.19}$$

This confirms quadratic convergence of the denominator toward 1. The numerator tracks the same scaling, so  $N_i \rightarrow 1/D = 1/y$  as  $D_i \rightarrow 1$ . The convergence rate is identical to Newton's method.

### B.5.3 Cancellation Analysis

The subtraction  $f_{i+1} = 2 - D_i$  requires scrutiny. Near convergence,  $D_i \approx 1$ , so  $2 - D_i \approx 1$ . The result is well away from zero: no catastrophic cancellation. This is in contrast to the cubic method's subtraction  $1 - y \cdot x_n \approx 0$ , making Goldschmidt's rounding behaviour simpler to control.

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## B.5.4 Rounding Error Per Iteration

Each iteration performs:

Step 1:  $f = 2 - D$  (1 subtraction; no cancellation, result  $\approx 1$ )

Step 2:  $N' = N \cdot f$  (1 multiplication)

Step 3:  $D' = D \cdot f$  (1 multiplication)

This is 2 multiplications + 1 subtraction = 3 operations per iteration. The two multiplications in Steps 2 and 3 are independent and could be parallelized in hardware, but in sequential software each contributes  $\epsilon'$ . Total rounding per iteration:

$$|\delta_{round,iter}| \leq 3\epsilon' = 3 \cdot 10^{-(p+g)} \quad (\text{B.20})$$

However, unlike Newton's method, the Goldschmidt method cannot benefit from dynamic precision scheduling: the numerator  $N$  and denominator  $D$  must both be tracked simultaneously at full precision from the start, because the intermediate quantity  $N/D$  does not converge monotonically in the same self-correcting sense as a Newton iterate. This doubles the effective work per iteration compared to Newton's method.

## B.5.5 Guard Digit Requirement

The per-iteration rounding bound (B.20) is identical to Newton's, so the guard digit requirement is the same:

$$\begin{aligned} 3 \cdot 10^{-(p+g)} &\leq \frac{1}{2} \cdot 10^{-p} \\ g &\geq 1 \end{aligned} \quad (\text{B.21})$$

Theoretical minimum:  $g = 1$  guard digit for Goldschmidt's method. However, because dynamic precision cannot be applied, Goldschmidt must carry all iterations at the full working precision  $p + g$  from the first iteration. In practice, this makes Goldschmidt substantially slower than Newton for arbitrary precision software, which is why it is not the recommended algorithm here.

## B.6 Recommendations

The table below consolidates the theoretical bounds derived in Sections B.2–B.5.

Method	Convergence	Ops/iteration	Min g	Recommended	Safety factor
Newton	Quadratic	3 ops	1	2	10,000×
Newton–Brent	Quadratic	3+1½ ops	1	2	10,000×
Cubic (Householder)	Cubic	7 ops	2	3	1,000×

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

---

Method	Convergence	Ops/iteration	Min g	Recommended	Safety factor
Goldschmidt	Quadratic	3 ops	1	2	N/A *

\* *Goldschmidt does not support dynamic precision scheduling, so the extra parameter does not apply in the same way. A fixed working precision of  $p + 2$  is recommended.*

## Notable Observations

1. Newton is over-protected. The current `extra = 5` provides a safety margin of 10,000× above the theoretical minimum. Reducing to `extra = 2` (100× margin) or even `extra = 1` (10× margin) would be sound. The conservative choice was sensible in the original implementation, but the analysis shows substantial room for reduction if performance is critical.
2. The cubic method requires more guard digits than Newton because it performs more floating-point operations per iteration (7 vs. 3) and involves a genuine cancellation in the subtraction  $1 - y \cdot x_n \approx 0$ . Higher-order convergence does not reduce the per-step rounding budget.
3. The cancellation in the cubic method is benign in context. Although the relative error in  $z_n = 1 - y \cdot x_n$  can be large (since  $z_n \approx 0$ ), the absolute error is bounded by  $\epsilon'$ . Since  $z_n$  is used only as a small correction term multiplied by  $x_n \approx x^*$ , the impact on the final result is at most  $\epsilon'$  in absolute terms, the same as any other single floating-point operation.
4. Goldschmidt's rounding bound equals Newton's, but its inability to exploit dynamic precision scheduling means it carries a higher practical cost. For hardware implementations where the two multiplications per step can be parallelized, Goldschmidt is competitive; for sequential software, it is not.
5. A unified recommendation: use `guard = 2` for Newton and Newton–Brent; use `guard = 3` for the cubic method. These values provide a 100× safety margin in all cases, which is consistent with the treatment of the corresponding square root algorithms in Appendix A.

## Appendix C: Guard Digit Analysis for the n-th Root Algorithm

This appendix derives a tight theoretical lower bound on the number of guard digits required to guarantee that the nroot Newton algorithm returns a result within one ULP (unit in the last place) of the true value  $n\sqrt{S}$ , at the requested decimal precision  $p$ . The algorithm iterates toward  $x^* = S^{-1/n}$  (the reciprocal n-th root) and recovers  $n\sqrt{S} = 1/x^*$  via a final inversion. The analysis therefore accounts for both the iteration rounding and the error introduced by that final inversion step.

### C.1 Notation and Setup

Let  $p$  be the target precision in decimal digits, with unit roundoff:

$$\varepsilon = 10^{-p}$$

All internal computation is performed at working precision  $p + g$  digits (guard digit count  $g$  to be determined), giving internal unit roundoff:

$$\varepsilon' = 10^{-(p+g)}$$

The input  $S > 0$  is reduced to a working value  $y \in [1, 2n)$  by extracting the binary exponent  $e_p$  as described in Section C.1.1. The iteration converges to:

$$x^* = y^{-1/n} \quad (\text{the exact reciprocal } n\text{-th root of } y)$$

and the final result is recovered as:

$$n\sqrt{y} = 1/x^*$$

The 1-ULP goal requires the final absolute error to satisfy:

$$|\delta_{final}| < \frac{1}{2} \cdot 10^{-p} \tag{C.1}$$

#### C.1.1 Exponent Extraction and Working Range

The implementation extracts the binary exponent  $e_p$  of  $S$  and strips it, leaving a normalized mantissa  $i_1.\text{fn} \in [1, 2)$ . For the n-th root,  $e_p$  must be made divisible by  $n$  before extraction, so that the post-iteration exponent adjustment is an exact integer. Writing  $e_p = n \cdot q + r$  where  $0 \leq r < n$ , the mantissa is scaled by  $2^r$ , giving an effective working argument  $y = i_1.\text{fn} \cdot 2^r$ . Since  $i_1.\text{fn} \in [1, 2)$  and  $2^r \leq 2^{n-1}$ , the full working range is:

$$y \in [1, 2^n) \tag{C.2}$$

For  $n = 2$ , this recovers  $[1, 4)$  as in Appendix A. All error bounds are derived using relative errors and hold uniformly across the entire range (C.2).

### C.2 The nroot Newton Algorithm

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## C.2.1 Derivation

We seek  $x^* = y^{-1/n}$  as the root of  $f(x) = x^n - y = 0$ . With  $f'(x) = nx^{n-1}$ , the Newton step  $x_{i+1} = x_i - f(x_i)/f'(x_i)$  gives:

$$\begin{aligned}x_{i+1} &= x_i - (x_i^n - y) / (n \cdot x_i^{n-1}) \\x_{i+1} &= x_i + (1/n)(x_i - x_i^{n+1} \cdot y) \\x_{i+1} &= x_i + (1/n) \cdot x_i \cdot (1 - x_i^n \cdot y)\end{aligned}$$

Factoring out  $x_i$ :

$$x_{i+1} = x_i \cdot (1/n)(n + 1 - y \cdot x_i^n) \tag{C.3}$$

The factor  $1/n$  is a constant, precomputed once as  $fn = 1/\text{float\_precision}(n)$  before the iteration begins, so no division occurs inside the loop. The final result is recovered by a single inversion after convergence.

## C.2.2 Convergence in Exact Arithmetic

Write  $x_i = x^*(1 + e_i)$ . Then  $y \cdot x_i^n = (1 + e_i)^n$  since  $y \cdot x^{*n} = 1$ . Substituting into (C.3):

$$x_{i+1} = x^*(1 + e_i) \cdot (1/n)(n + 1 - (1 + e_i)^n)$$

Expanding  $(1 + e_i)^n = 1 + ne_i + \frac{1}{2}n(n-1)e_i^2 + O(e_i^3)$  and collecting terms:

$$\begin{aligned}n + 1 - (1 + e_i)^n &= n - ne_i - \frac{1}{2}n(n-1)e_i^2 + O(e_i^3) \\(1 + e_i) \cdot (n + 1 - (1 + e_i)^n) &= (1 + e_i) \cdot (n - ne_i - \frac{1}{2}n(n-1)e_i^2 + O(e_i^3)) \\&= 1 - \frac{1}{2}(n+1)e_i^2 + O(e_i^3)\end{aligned}$$

Therefore, the relative error satisfies:

$$e_{i+1} = -\frac{1}{2}(n+1)e_i^2 + O(e_i^3) \tag{C.4}$$

This confirms quadratic convergence for all  $n \geq 2$ . The contraction constant  $\frac{1}{2}(n+1)$  grows with  $n$ : for  $n=2$  it equals  $3/2$ , for  $n=3$  it equals  $2$ , and so on. The convergence order remains quadratic for all  $n$ , and the dynamic precision schedule (doubling digits per iteration) remains appropriate throughout.

## C.3 Operation Count and Rounding Error

### C.3.1 Computing $x^n$ by Binary Exponentiation

The implementation computes  $x^n$  via repeated squaring. For a given  $n$ , let  $k = \lfloor \log_2(n) \rfloor$  be the index of the highest set bit, and let  $v(n)$  denote the number of 1-bits (Hamming weight) in the binary representation of  $n$ . The loop performs:

- $k = \lfloor \log_2(n) \rfloor$  squaring's ( $p \leftarrow p^2$ )
- $v(n) - 1$  multiplications ( $\text{res} \leftarrow \text{res} \cdot p$ , for each set bit)

The total number of multiplications to compute  $x^n$  is therefore:

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

$$M(n) = \lfloor \log_2(n) \rfloor + v(n) - 1 \quad (\text{C.5})$$

For example:  $n=2 \rightarrow M=1$ ,  $n=3 \rightarrow M=2$ ,  $n=4 \rightarrow M=2$ ,  $n=7 \rightarrow M=4$ ,  $n=8 \rightarrow M=3$ . The worst case for a  $k$ -bit value is  $n = 2^{k+1} - 1$  (all bits set), giving  $M = 2k$ . Each multiplication in the binary exponentiation contributes at most  $\varepsilon'$  relative error, so the total relative error in  $x^n$  satisfies:

$$|\delta(x^n)| \leq M(n) \cdot \varepsilon' = M(n) \cdot 10^{-(p+g)} \quad (\text{C.6})$$

### C.3.2 Remaining Operations Per Iteration

After computing  $x^n$ , iteration (C.3) requires four further operations:

$$\text{Step A: } r_1 = y \cdot x^n \quad (1 \text{ multiplication})$$

$$\text{Step B: } r_2 = (n+1) - r_1 \quad (1 \text{ subtraction})$$

$$\text{Step C: } r_3 = r_2 \cdot f_n \quad (1 \text{ multiplication; } f_n = 1/n \text{ precomputed})$$

$$\text{Step D: } x_{i+1} = x_i \cdot r_3 \quad (1 \text{ multiplication})$$

Step B: near convergence  $y \cdot x^n \approx 1$ , so  $(n+1) - y \cdot x^n \approx n$ . The result has magnitude  $\approx n$ , well away from zero for all  $n \geq 2$ . No catastrophic cancellation occurs in Step B. Steps A, C, D each contribute  $\varepsilon'$  relative error.

Combining (C.6) with the three multiplications (Steps A, C, D) and one subtraction (Step B):

$$|\delta_{\text{round,iter}}| \leq (M(n) + 4) \cdot \varepsilon' = (M(n) + 4) \cdot 10^{-(p+g)} \quad (\text{C.7})$$

Here  $M(n)$  counts the binary exponentiation multiplications, three further multiplications (Steps A, C, D) each add  $\varepsilon'$ , and the subtraction (Step B) adds  $\varepsilon'$ , giving  $M(n) + 4$  in total.

### C.3.3 The Final Inversion $n\sqrt{S} = 1/x$

After the iteration converges, the implementation performs  $x = 1/x$ . At arbitrary precision, this is itself an iterative computation (Newton inverse, Appendix B), evaluated at full working precision  $p + g$ . It introduces a relative rounding error of at most  $\varepsilon'$ :

$$|\delta_{\text{inv}}| \leq \varepsilon' = 10^{-(p+g)} \quad (\text{C.8})$$

The inversion is self-contained, and benefits from its own guard digits at precision  $p + g$ , so no additional guard digit inflation is required beyond what (C.8) captures.

## C.4 Total Error Budget and Guard Digit Bound

# Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

## C.4.1 Combining All Error Sources

Combining iteration rounding (C.7) with the final inversion (C.8):

$$|\delta_{total}| \leq (M(n) + 5) \cdot \varepsilon' = (M(n) + 5) \cdot 10^{-(p+g)} \quad (C.9)$$

## C.4.2 Guard Digit Requirement

Applying the 1-ULP goal (C.1) to (C.9):

$$\begin{aligned} (M(n) + 5) \cdot 10^{-(p+g)} &\leq \frac{1}{2} \cdot 10^{-p} \\ 10g &\geq 2(M(n) + 5) \\ g &\geq \log_{10}(2(M(n) + 5)) \end{aligned} \quad (C.10)$$

Since  $M(n) = \lfloor \log_2(n) \rfloor + v(n) - 1$ , the bound grows logarithmically with  $n$ . Evaluating (C.10) for representative values of  $n$ :

n	$\lfloor \log_2(n) \rfloor$	v(n)	M(n)	Total ops M(n)+5	2(M(n)+5)	Min g (exact)	Min g (int.)
2	1	1	1	6	12	1.079	2
3	1	2	2	7	14	1.146	2
4	2	1	2	7	14	1.146	2
5	2	2	3	8	16	1.204	2
7	2	3	4	9	18	1.255	2
8	3	1	3	8	16	1.204	2
10	3	2	4	9	18	1.255	2
15	3	4	6	11	22	1.342	2
16	4	1	4	9	18	1.255	2
31	4	5	8	13	26	1.415	2
32	5	1	5	10	20	1.301	2

For the practically important cases  $n = 2$  through  $n = 8$ , the theoretical minimum is  $g = 2$  guard digits. The bound reaches  $g = 3$  only for  $n \geq 31$ , where  $M(n) = 8$  in the worst case (all-bits-set values like  $n=31$ ). For all  $n \leq 16$ , the minimum is  $g \leq 2$ .

## C.4.3 Simplified Practical Bound

Since  $M(n) \leq 2\lfloor \log_2(n) \rfloor$  (worst case, all bits set), a conservative closed-form bound is:

$$g \geq \lceil \log_{10}(2(2\lfloor \log_2(n) \rfloor + 5)) \rceil \quad (C.11)$$

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

For  $n \leq 215 = 32,768$  (a generous practical upper bound),  $M(n) \leq 29$ , giving  $2(29+5) = 68$ , and  $\log_{10}(68) \approx 1.83$ , so  $g \geq 2$ . Therefore,  $g = 2$  is a safe and sufficient choice for all practical values of  $n$ .

### C.5 Comparison with Current Implementation

The current implementation uses `guard = 2`. From the analysis,  $g = 2$  is the theoretical minimum for all  $n \leq 30$ , with the safety factor:

$$\text{Safety factor} = 10g / 2(M(n)+5) = 102 / 2(M(n)+5) \quad (\text{C.12})$$

For the most common cases  $n = 2$  through  $n = 8$ :

n	M(n)	2(M(n)+5)	100 / 2(M(n)+5)	Safety factor
2	1	12	8.3×	Adequate
3	2	14	7.1×	Adequate
4	2	14	7.1×	Adequate
5	3	16	6.3×	Adequate
7	4	18	5.6×	Adequate
8	3	16	6.3×	Adequate

The safety margin of approximately 6–8× for common cases is comfortable. The current `guard = 2` is correct and requires no adjustment.

### C.6 Recommendation

1. Quadratic convergence for all  $n \geq 2$ . The error satisfies  $e_{i+1} = -\frac{1}{2}(n+1) \cdot e_i^2 + O(e_i^3)$ . The contraction constant  $\frac{1}{2}(n+1)$  grows with  $n$ , but the order remains quadratic. The dynamic precision schedule (doubling digits per iteration) is appropriate for all  $n$ .
2. The dominant cost is computing  $x^n$  by binary exponentiation. This requires  $M(n) = \lfloor \log_2(n) \rfloor + v(n) - 1$  multiplications, growing logarithmically with  $n$ . For  $n \leq 32$  the range is  $M = 1$  ( $n=2$ ) to  $M = 5$  ( $n=32$ ).
3. No catastrophic cancellation. The subtraction  $(n+1) - y \cdot x^n$  produces a result of magnitude  $\approx n$  near convergence, well away from zero for all  $n \geq 2$ .
4. Theoretical minimum guard digit count is  $g = 2$  for all practical  $n$  ( $\leq 2^{15}$ ). This matches the current `guard = 2`, which is correct and requires no change.
5. The final inversion contributes one unit of  $\varepsilon'$ . Evaluated at full working precision  $p + g$ , it does not require additional guard digits beyond those already in use.

## Fast Square Root and inverse calculation for Arbitrary Precision numbers

---

6. Consistency with Appendices A and B. The requirement  $g = 2$  places root alongside Halley square root and the cubic inverse (both also  $g = 2$ ), reflecting a similar total operation count per iteration. Newton's square root and Newton's inverse need only  $g = 1$  theoretically, but both use  $g = 2$  in practice for a safety margin, yielding uniform guard-digit usage across the library.