Fast Square Root & inverse calculation for Arbitrary Precision numbers.

By Henrik Vestermark (hve@hvks.com)

Abstract:

This is a follow-up to a previous paper that describes the math behind arbitrary precision numbers. First of all the original paper was written back in 2013 and quite a few things had happens since then, secondly, I have come across some other interesting methods to do the calculation. The paper describes in more detail how to do square root, inverse (1/x) & $\sqrt[n]{x}$ calculations with arbitrary precision and outlines some traditional methods but also introduces an improved version that doubles the speed of each calculation. Particular a new concept for iterations using dynamic precision to make a fast calculation of the square root, inverse, or $\sqrt[n]{x}$ calculation.

Introduction:

Usually, when implementing arbitrary precision math packages you would use the standard Newton calculation as the preferred iterative method for calculating the square root, the inverse (1/x), or, $\sqrt[n]{x}$ for arbitrary precisions. The Newton method has a convergence rate of two meaning that for each iteration the number of correct digits doubles. This is traditional and has been implemented in various arbitrary precision packages. However, there exist other methods with higher order convergence rates (e.g. Halley's method with cubic convergence rate). We will examine if using higher order methods (e.g. Halley) is worth the extra work and compare it with second order methods (e.g. Newton).

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

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

1. Fast Computation of Math Constants in arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
2. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
3. Fast Square Root & Inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision
4. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision
5. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision

6. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants
7. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms
8. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision
9. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision
10. Fast conversion from arbitrary precision number to a string. HVE Fast conversion from arbitrary precision to string
11. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision

Change log

1-March 2023. Minor corrections,
26-January 2023. Cleaning up the document grammatically.

Contents

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 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*, however since this precision can be arbitrary we can declare the wanted precision as the number of ***decimal digits*** we want to use when dealing 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 to manipulate the exponent 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 build-in types *float* and *double*) E.g.

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

There is a second way to manipulate the exponent and that is 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 with 2.
f.adjustExponent(-1);  // Subtract 1 from the exponent, the same as dividing the number with 2.
```

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

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

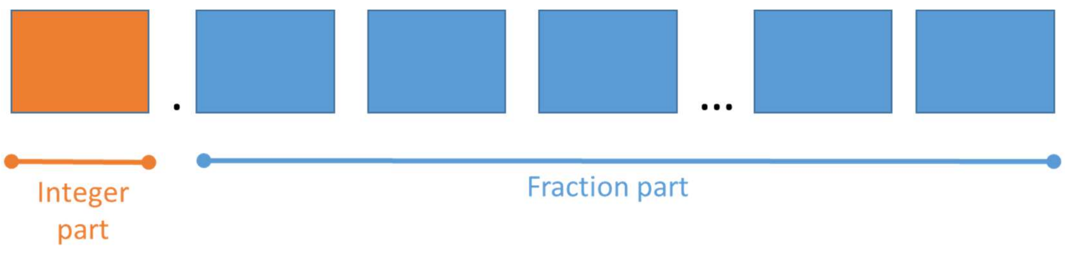
For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
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 mBinary



- 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 >=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.

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].

Square root

There exist several methods to compute the square root. Among them are:

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

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

Newton's Method without division

For the function sqrt(y) we can use a Newton iteration algorithm to get our result. The Newton iteration is defined by:

$$x_{n+1} = \mathrm{x}_n - \frac{f(x_n)}{f'(x_n)} \qquad (1)$$

This method can be found the following way by restating the problem of finding 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 with 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 using f(x)= $\frac{1}{x^2} - y$, f'(x)= $\frac{-2}{x^3}$

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

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

$$x_{n+1} = \frac{1}{2}\mathrm{x}_n(3 - yx_n^2) \qquad (2)$$

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

$$x_{n+1} = \frac{1}{2}\mathrm{x}_n(3 - yx_n^2) \qquad (3)$$

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

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

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 the algorithm only requires us to do 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 multiplication per iteration and then a final multiplication for the calculation of the square root.
Also as for the Newton method, we will have quadratic convergence meaning that for each iteration we will 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 with $2^{\frac{e_p}{2}}$ 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.

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

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

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

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 $\frac{1}{Sqrt(y)}$ using standard IEEE754 arithmetic with approx. 15-16 significant decimal digits.

Example of Newton's method for 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.

Newton 1/sqrt(y)

Sqrt(y)		1.6		
y=		1.6		
x_0=		0.625		
	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 build 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:

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

Which is identical from a mathematical point of view but different from a computational point of view. Brent points out that you can perform the multiplication between x_{n-1} and $(1 - yx_n^2)$ in $x_n(1 - yx_n^2)$ 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 use the Newton method to solve for x. By applying Newton's method:

$$x_{n+1} = \mathrm{x}_n - \frac{f(x_n)}{f'(x_n)} \tag{7}$$

We get.

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

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.

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 with $2^{\frac{e_p}{2}}$ 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.

$$Sqrt(y) = Sqrt(i_1.f_n 2^{e_p}) = Sqrt(i_1.f_n)2^{\frac{e_p}{2}} \qquad (9)$$

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

$$Sqrt(y) = Sqrt(i_1.f_n 2^{e_p}) = Sqrt(i_1.f_n * 2)2^{-\frac{e_{p-1}}{2}} \qquad (10)$$

This simplifies the initial guess since we know that factoring out the exponent will leave us 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 using standard IEEE754 arithmetic with approx. 15-16 significant decimal digits.

Example Newon method for square root using division

Newton sqrt(y)				
Sqrt(y)		1.6		
y=		1.6		
x_0=		1.6		
	n	**x_n**	**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 compare to Newton's method which only gives you two times as many correct digits. Higher order convergence results in fewer iterations step at the expense of a more complex calculation per iteration. Normally it tends to even out that the time you save in fewer iterations steps is lost by a more complex iteration.

Halley square root method is using the following iterations step for finding $\frac{1}{\sqrt{y}}$:

$$z_n = yx_n^2$$
$$x_{n+1} = x_n\frac{1}{8}(15 - z_n(10 - 3z_n)) \qquad (11)$$

And then we get the final result of $\sqrt{y}$=y·x$_{n+1}$.

It can be found using Householders' 2nd order method aka. Halley's method:

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

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

This yield:

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

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

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

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

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

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

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

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

Per iteration, we have five multiplication and two subtraction. Compare to Newton we have added a subtraction and two extra multiplication so each iteration will take a little bit longer however you will have fewer iterations to perform. (Approx. 2/3).

Example of Halley's method for square root

Halley 1/Sqrt(y)			
Sqrt(y)	1.6		
y	1.6		
x0	0.625		

n	**x_n**	**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

Goldschmidt method is typically implemented in FPGA or Floating point unit and takes advantage of the CPU pipeline structure to archive higher performance. However, at the software level, it does not produce anything faster or better than a standard Newton method.

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 in case we have made an imprecise calculation in one iteration step it will be corrected automatically in the following Newton iteration.

Iteration using Dynamic Precision

We can use that information to instead of starting the first couple of iterations with a precision of thousands of digits we could instead start with a smaller precision and then gradually increase the precision in our calculation until we in the last iterations do all calculations using the required number of digits. Initially, in our first guess, we have

approx. 15-16 correct digits. We know that a Newton iteration doubles the number of correct digits for every iteration, We will start the first iteration using 32 digits precision and then after each iteration step, we 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.
I have not seen this technic applied before so in lack of a precedent I will call this a Newton with iterative deepening or Newton using dynamic precision.

Applying this technic, we get a speed up over the classic Newton iteration with an approx. 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 notice on the diagram below that is a difference between using Newton and Halley methods. Newton has the benefit of needing fewer multiplications but Halley can do the calculation using fewer iterations. Halley method is consistently approx. 20% faster than the Newton method in the below performance chart.

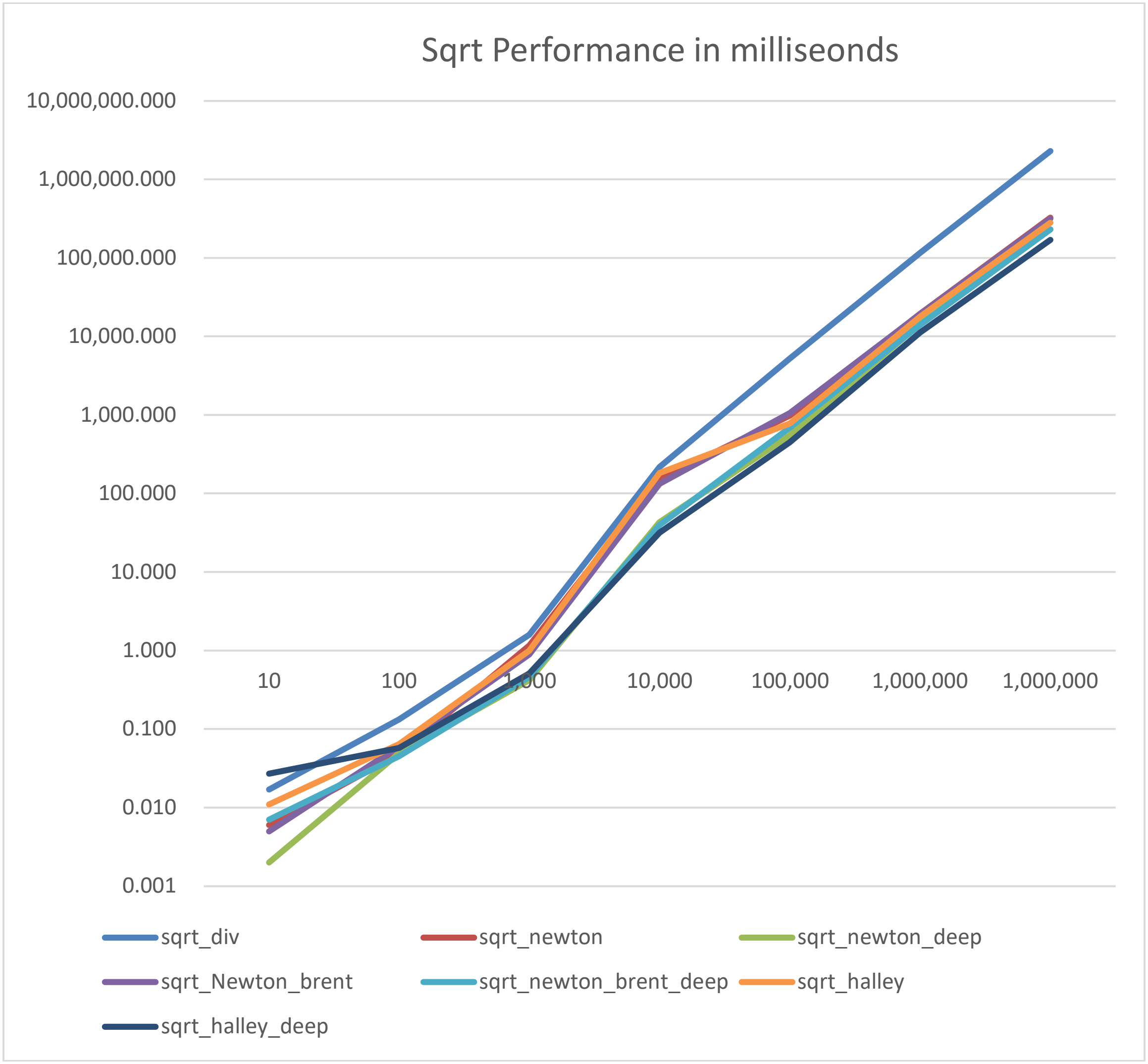


Figure 1. The horizontal axes are precision in decimal digits. The vertical axis is in milliseconds.

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.

Source sqrt newton_dynamic()

```
float_precision sqrt_newton_dynamic(const float_precision& a)
	{
	const unsigned int extra = 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 squareroot
		return a;
	y.precision(precision + extra);
	expo_sq = expo / 2;
	y.exponent(expo - 2 * expo_sq);
	// Do iteration using 2 digits higher precision
	r.precision(precision + extra);
	x.precision(precision + extra);

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

	// Now iterate using Netwon x=0.5x(3-yx^2)
	// y is the original number of the square root which has the full precision
	for (digits = std::min((size_t)32, precision); ; digits =
std::min(precision + extra, digits * 2))
		{
		// Increase precision by two for the working variable r,x
		r.precision(digits);
		x.precision(digits);
		r = c3 - y * x * x;         // 3-yx^2
		r.adjustExponent(-1);       // (3-yx^2)/2
		x *= r;                     // x=x(3-yx^2)/2
		// Reach final iteration step in regards to precision
		if (digits == precision + extra)
			{
			r.precision(precision + 1); // round to final precision
			if (r == c1)         // break if no improvement
				break;
			r.precision(precision + extra);
			}
		}
	x *= y;
	x.adjustExponent(expo_sq);
	// Round to the same precision as argument and rounding mode
	x.mode(a.mode());
	x.precision(precision);
	return x;
	}
```

Source sqrt Halley_dynamic()

```
float_precision sqrt_halley_dynamic(const float_precision& a)
	{
	const unsigned int extra = 2;
	const size_t precision=a.precision();
	const eptype expo=a.exponent();
	eptype expo_sq;
	size_t digits;
	double fv;
	float_precision r, x, y(a);
	const float_precision c1(1), c15(15), c3(3), c10(10);

	if (a.iszero() || a == c1)  // Simple square root
		return a;

	expo_sq = expo / 2;
	y.exponent(expo - 2 * expo_sq);
	// Do iteration using 2 digits higher precision
	y.precision(precision + extra);
	r.precision(precision + extra);
	x.precision(precision + extra);
	// Get an initial guess using an ordinary floating point
	fv = (double)y;      // Convert to double
	// set the initial guess with at approx 16 correct digits
	fv = 1 / sqrt(fv);
	x = float_precision(fv);

	// Now iterate using Halley
	// y is the original number of the square root which has the full precision
	for (digits = std::min((size_t)48, precision); ; digits =
std::min(precision + extra, digits * 3) )
		{
		r.precision(digits);
		x.precision(digits);
		r = y * x * x;              // yx^2
		r = (c15 - r*(c10 - c3*r)); // 15-yx^2*(10-3*yx^2)
		r.adjustExponent(-3);       // r=r/8
		x *= r;                     // x=x/8(15-yx^2*(10-3*yx^2)
		// Reach final iteration step in regards to precision
		if (digits == precision + extra)
			{
			r.precision(precision + 1); // round to final precision
			if (r == c1)  // break if no improvement
				break;
			}
```

```
              }

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

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 raise to the power is the exponent of the number (base 2). If the exponent is, even then, we have directly our square root by just dividing the exponent by two and you are done. You can insert the following code right after the first assignment of the local expo variable.

Source:

```
       // 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 a power of two numbers 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 initialized 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.

Recommendation for the square root

Sqrt(y)	Addition/Subtraction	Multiplication	Multiplication half precision	Division
Newton*	1	3		
Newton (Brent)	2	2	1	
Newton Division*	1			1
Halley	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 with division requires the least among of operations but the division is an expensive operator that consistently performs 4-8 times slower than the other methods and can therefore not be recommended.

Based on the performance measure 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 approx. 20% performance gain using Halley over the Newton Method (Brent variation). Newton is simpler but requires additional iterations than Halley. Halley is more complex per iteration but requires fewer of them to complete.
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.

Nrooth

Now that we have found a better way of doing the square root we also need to consider if we can use a similar technic 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 use two very expensive and time-consuming functions exp(x) and log(x). Instead, we can create a new function nroot(x,n) that calculate$\sqrt[n]{x}$. Using the same principle as the sqrt() the result is a huge speed-up improvement.

As can be seen below the speed of the nroot() is more or less constant regardless of the nth root and it is several magnitudes better than the traditional calculation via the pow() function.

Let us end the discussion of the sqrt() and nroot() by devising the Newton formula for the nroot. It is quite similar to the way we got the algorithm for the sqrt() function. We are trying to find a function to 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:

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

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

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

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

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

We still have a division $\frac{1}{n}$ but it is with the constant *n* so we can calculate it once before the start of the iteration avoiding any division while iterating.

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

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

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

Using the Newton method, you get:

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

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

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

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

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

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

Source for nrooth_newton_dynamic()

```
float_precision nroot_newton_dynamic(const float_precision& a, const uintmax_t n)
	{
	const size_t extra = 2;
	const size_t precision=a.precision();
	const eptype expo = a.exponent();
	const float_precision c1(1);
	eptype expo_sq;
	size_t digits;
	double fv;
	float_precision r, x, y(a), fn(n);

	if (a.iszero() || a == c1 || n == 1)
		return a;
	y.precision(precision + extra);
	expo_sq = expo / 2;
	y.exponent(expo - 2 * expo_sq);
	// Do iteration using guard digits with higher precision
	x.precision(precision + extra);
	fn.precision(precision + extra);
	// Get an initial guess using an ordinary floating point
	fv = y;
	fv = pow(fv, 1.0 / n);
	fv = 1 / fv;
	// set the initial guess with at ~ 16 correct digits
	x = float_precision(fv);
	fn = 1 / fn;
	// Now iterate using Netwon  x=x*(-yx^n+(n+1))/n
	// y is the original number to nroot which has the full precision
	for (digits = std::min((size_t)32, precision); ; digits =
std::min(precision + extra, digits * 2))
		{
		// Increase precision by a factor of two
		r.precision(digits);
		x.precision(digits);
		float_precision p(x);
		float_precision res(1, digits);
		// Do x^n
		for (uintmax_t i = n; i > 0; i >>= 1)
			{
			if ((i & 0x1) != 0)
				res *= p;  // Odd
			if (i>1)
				p *= p;
			}

		r = float_precision(n + 1) - y*res; // (n+1)-yx^n
		r *= fn;                           // (-yx^n+(n+1))/n
		x *= r;                            // x=x*(-yx^n+(n+1))/n
		// Reach final iteration step in regards to precision
		if (digits == precision + extra)
			{
			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());
	x.precision(precision);
	return x;
	}
```

Recommendation $\sqrt[n]{x}$

1) Use the Newton method with dynamic precision.

The Inverse

To handle floating-point division we rewrite the equation a/b to a· (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 faces many microprocessors or early RISC (Reduced Instruction Set CPU) that did not have hardware support for the division operator. The traditional method has been used due to its simplicity but there exist other higher-order methods 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 the following way by restating the problem of finding $\frac{1}{x} = y$.

Applying it to the Newton method, you get:

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

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

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

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

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

Notice the algorithm only requires us to do one subtraction and two multiplications per iteration.

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_0=		0.1
n	**x_n**	**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 build-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 n fraction parts 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 $\frac{1}{i_1 . f_n}$ and then multiply the result with 2^{-e_p} to find our inverse of 1/b. Extracting the exponent will leave us with a number [1..2[. Since we do have the support of hardware division using the IEE754 standard (64bit floating-point number) we can get our initial start guess, with approximately 15-16 digits accuracy, and then begin to iterate towards a higher number of accuracy. In case you do not have access to IEEE754, you can do a lookup table to find a suitable starting point.

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

Brent improvement

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

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

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 apply the same technique for the square root and nroot calculation using dynamic precision.

We can use that information to instead of starting the first couple of iterations with a precision of thousands of digits we could instead start with a smaller precision and then gradually increase the precision in our calculation until we in the last iteration do all calculations using the required number of digits. Initially, in our first guess, we have approx. 15-16 correct digits. We know that a Newton iteration doubles the number of correct digits for every iteration, We will start the first iteration using 32 digits precision and then after each iteration step, we 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.
I have not seen this technic applied before so in lack of a precedent I will call this a Newton with iterative deepening or Newton with dynamic precision.

Source inverse_newton_dynamic()

```
float_precision _float_precision_inverse_newton_dynamic(const float_precision& a)
	{
	const size_t extra = 5;
	const size_t precision=a.precision();
	const eptype expo=a.exponent();
	const float_precision c1(1), c2(2);
	size_t digits;
	double fx;
	float_precision r, x, y(a);

	// if a is a true power of 2 then we do not need to iterate but
	// just reverse the exponent and return
	if (a.size() == 1)
		{
		y.exponent(-y.exponent());
		return y;
```

```
              }

       y.precision(precision + extra);
       // 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 with higher precision
       x.precision(precision + extra);

       // Get an initial guess using an ordinary floating point
       fx = 1/(double)y;
       x = float_precision(fx);

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

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

Higher-order Newton-like methods for inverse

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

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

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:

$$z_n = 1 - yx_n$$
$$x_{n+1} = x_n + x_n(z_n) + x_n(z_n)^2 \tag{18}$$

It can be found using Householders 2nd order method:

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

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

This yield:

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

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

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

This method will require one subtraction, two addition, and four multiplication.

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

$$z_n = 1 - yx_n$$
$$x_{n+1} = x_n(1 + z_n + (z_n)^2 \tag{21}$$

This will require three addition/subtraction and three multiplication. However, all the multiplication needs to carry out using full precision.

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

Example of Cubic 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_0= 0.1

n	x_n	Error
1	0.25456	3.7E-01
2	0.494865	1.3E-01
3	0.619358	5.6E-03
4	0.625	4.6E-07
5	0.625	0.0E+00

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

Source inverse_3rd order_dynamic()

```
float_precision _float_precision_inverse_cubic_deep(const float_precision& a)
	{
	const size_t extra = 5;
	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);
	size_t digits;
	double fx;
	float_precision r, s, x, y(a);

	// if a is a true power of 2 then we do not 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 extra digits with higher precision
	x.precision(precision + extra);
	// Get an initial guess using an ordinary floating point
	fx = 1 / (double)y;
	x = float_precision(fx);

	// Now iterate using 3rd order x=x+x(1-xy)+x(1-yx)^2
	for (digits = std::min((size_t)48, precision); ; digits =
std::min(precision + extra, digits * 3) )
		{
		// Increase precision by two for the working variable s & x.
		s.precision(digits);
		x.precision(digits);
		// Only half the precision for r as suggested by Richard P. Brent
		r.precision(digits / 2 + 1);
		r = c1 - y * x;             // (1-yx)
		s = x * r;                  // x(1-yx)
		x += s;                     // x = x + x(1 - yx)
		if (2 * r.exponent() > limit)
			x += s * r;         // x=x+x(1-yx)+x(1-yx)^2
```

```
            if (digits == precision + extra  &&
                   ( r.iszero() || 2 * r.exponent() < limit) )
                   break;
            }

        // Reapply exponent, mode, and precision
        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.

Performance:

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

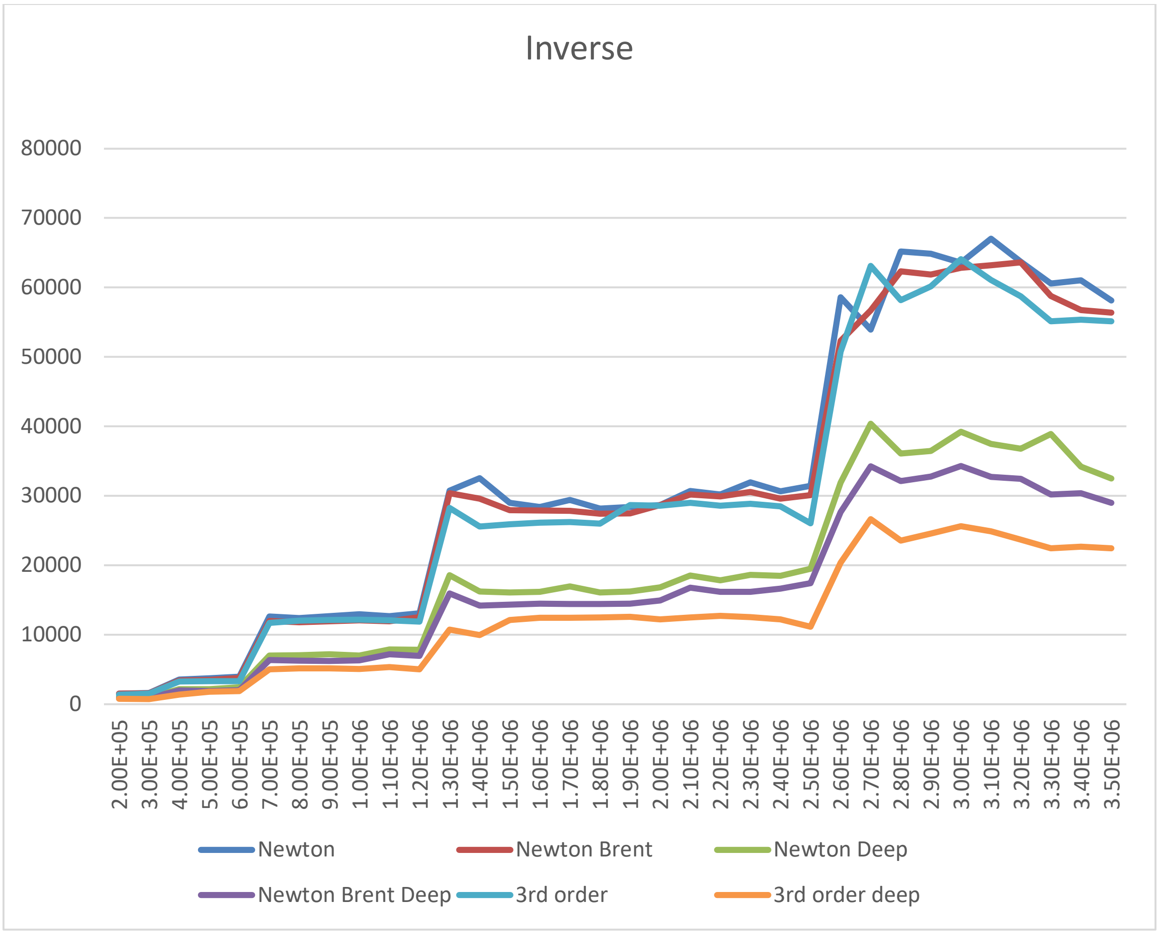


Figure 2

As can be seen, the 3rd order with dynamic (deep in the above 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 gain a significant performance improvement using dynamic precision iteration. In addition, with the use of Brent [7] enhancement, you can further save on the workload.

	Addition/Subtraction	Multiplication	Multiplication half precision
Newton	1	2	
Newton (Brent)	2	1	1
3rd order	3	3	
3rd order (Brent)	3	1	3

Based on the performance measure recommend:

1) Use the third-order 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 method.

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages (hvks.com)
2) Numerical recipes in C++, 3rd 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
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