WIKIPEDIA

Methods of computing square roots

Methods of computing square roots are numerical analysis algorithms for finding the principal, or non-negative, square root (usually denoted $\sqrt{S}$, $\sqrt[2]{S}$, or $S^{1/2}$) of a real number. Arithmetically, it means given S, a procedure for finding a number which when multiplied by itself, yields S; algebraically, it means a procedure for finding the non-negative root of the equation $x^2 - S = 0$; geometrically, it means given the area of a square, a procedure for constructing a side of the square.

Every real number has two square roots.[Note 1] The principal square root of most numbers is an irrational number with an infinite decimal expansion. As a result, the decimal expansion of any such square root can only be computed to some finite-precision approximation. However, even if we are taking the square root of a perfect square integer, so that the result does have an exact finite representation, the procedure used to compute it may only return a series of increasingly accurate approximations.

The continued fraction representation of a real number can be used instead of its decimal or binary expansion and this representation has the property that the square root of any rational number (which is not already a perfect square) has a periodic, repeating expansion, similar to how rational numbers have repeating expansions in the decimal notation system.

The most common analytical methods are iterative and consist of two steps: finding a suitable starting value, followed by iterative refinement until some termination criteria is met. The starting value can be any number, but fewer iterations will be required the closer it is to the final result. The most familiar such method, most suited for programmatic calculation, is Newton's method, which is based on a property of the derivative in the calculus. A few methods like paper-and-pencil synthetic division and series expansion, do not require a starting value. In some applications, an integer square root is required, which is the square root rounded or truncated to the nearest integer (a modified procedure may be employed in this case).

The method employed depends on what the result is to be used for (i.e. how accurate it has to be), how much effort one is willing to put into the procedure, and what tools are at hand. The methods may be roughly classified as those suitable for mental calculation, those usually requiring at least paper and pencil, and those which are implemented as programs to be executed on a digital electronic computer or other computing device. Algorithms may take into account convergence (how many iterations are required to achieve a specified precision), computational complexity of individual operations (i.e. division) or iterations, and error propagation (the accuracy of the final result).

Procedures for finding square roots (particularly the square root of 2) have been known since at least the period of ancient Babylon in the 17th century BCE. Heron's method from first century Egypt was the first ascertainable algorithm for computing square root. Modern analytic methods began to be developed after introduction of the Arabic numeral system to western Europe in the early Renaissance. Today, nearly all computing devices have a fast and accurate square root function, either as a programming language construct, a compiler intrinsic or library function, or as a hardware operator, based on one of the described procedures.

Contents

Initial estimate

Many iterative square root algorithms require an initial seed value. The seed must be a non-zero positive number; it should be between 1 and S, the number whose square root is desired, because the square root must be in that range. If the seed is far away from the root, the algorithm will require more iterations. If one initializes with $x_0 = 1$ (or S), then approximately $\frac{1}{2}|\log_2 S|$ iterations will be wasted just getting the order of magnitude of the root. It is therefore useful to have a rough estimate, which may have limited accuracy but is easy to calculate. In general, the better the initial estimate, the faster the convergence. For Newton's method (also called Babylonian or Heron's method), a seed somewhat larger than the root will converge slightly faster than a seed somewhat smaller than the root.

In general, an estimate is pursuant to an arbitrary interval known to contain the root (such as $[x_0, 1/x_0]$). The estimate is a specific value of a functional approximation to $f(x) = \sqrt{x}$ over the interval. Obtaining a better estimate involves either obtaining tighter bounds on the interval, or finding a better functional approximation to f(x). The latter usually means using a higher order polynomial in the approximation, though not all approximations are polynomial. Common methods of estimating include scalar, linear, hyperbolic and

logarithmic. A decimal base is usually used for mental or paper-and-pencil estimating. A binary base is more suitable for computer estimates. In estimating, the exponent and mantissa are usually treated separately, as the number would be expressed in scientific notation.

Decimal estimates

Typically the number S is expressed in scientific notation as $a \times 10^{2n}$ where $1 \le a < 100$ and n is an integer, and the range of possible square roots is $\sqrt{a} \times 10^n$ where $1 \le \sqrt{a} < 10$.

Scalar estimates

Scalar methods divide the range into intervals, and the estimate in each interval is represented by a single scalar number. If the range is considered as a single interval, the arithmetic mean (5) or geometric mean ($\sqrt{10} \approx 3.16$) times 10^n are plausible estimates. The absolute and relative error for these will differ. In general, a single scalar will be very inaccurate. Better estimates divide the range into two or more intervals, but scalar estimates have inherently low accuracy.

For two intervals, divided geometrically, the square root $\sqrt{S} = \sqrt{a} \times 10^n$ can be estimated as[Note 2]

$$\sqrt{S} \approx \begin{cases} 2 \cdot 10^n & \text{if } a < 10, \\ 6 \cdot 10^n & \text{if } a \ge 10. \end{cases}$$

This estimate has maximum absolute error of $4 \cdot 10^n$ at a = 100, and maximum relative error of 100% at a = 1.

For example, for $S = 125348$ factored as 12.5348×10^4, the estimate is $\sqrt{S} \approx 6 \cdot 10^2 = 600$. $\sqrt{125348} = 354.0$, an absolute error of 246 and relative error of almost 70%.

Linear estimates

A better estimate, and the standard method used, is a linear approximation to the function $y = x^2$ over a small arc. If, as above, powers of the base are factored out of the number S and the interval reduced to [1,100], a secant line spanning the arc, or a tangent line somewhere along the arc may be used as the approximation, but a least-squares regression line intersecting the arc will be more accurate.

A least-squares regression line minimizes the average difference between the estimate and the value of the function. Its equation is $y = 8.7x - 10$. Reordering, $x = 0.115y + 1.15$. Rounding the coefficients for ease of computation,

$$\sqrt{S} \approx (a/10 + 1.2) \cdot 10^n$$

That is the best estimate *on average* that can be achieved with a single piece linear approximation of the function $y=x^2$ in the interval [1,100]. It has a maximum absolute error of 1.2 at a=100, and maximum relative error of 30% at S=1 and 10.[Note 3]

To divide by 10, subtract one from the exponent of a, or figuratively move the decimal point one digit to the left. For this formulation, any additive constant 1 plus a small increment will make a satisfactory estimate so remembering the exact number isn't a burden. The approximation (rounded or not) using a single line spanning the range [1,100] is less than one significant digit of precision; the relative error is greater than $1/2^2$, so less than 2 bits of information are provided. The accuracy is severely limited because the range is two orders of magnitude, quite large for this kind of estimation.

A much better estimate can be obtained by a piece-wise linear approximation: multiple line segments, each approximating some subarc of the original. The more line segments used, the better the approximation. The most common way is to use tangent lines; the critical choices are how to divide the arc and where to place the tangent points. An efficacious way to divide the arc from y=1 to y=100 is geometrically: for two intervals, the bounds of the intervals are the square root of the bounds of the original interval, 1*100, i.e. $[1,\sqrt[2]{100}]$ and $[\sqrt[2]{100},100]$. For three intervals, the bounds are the cube roots of 100: $[1,\sqrt[3]{100}]$, $[\sqrt[3]{100},(\sqrt[3]{100})^2]$, and $[(\sqrt[3]{100})^2,100]$, etc. For two intervals, $\sqrt[2]{100} = 10$, a very convenient number. Tangent lines are easy to derive, and are located at $x = \sqrt{1}*\sqrt{10}$ and $x = \sqrt{10}*\sqrt{10}$. Their equations are: y = 3.56x - 3.16 and y = 11.2x - 31.6. Inverting, the square roots are: x = 0.28y + 0.89 and x = .089y + 2.8. Thus for $S = a * 10^{2n}$:

$$\sqrt{S} \approx \begin{cases} (0.28a + 0.89) \cdot 10^n & \text{if } a < 10, \\ (.089a + 2.8) \cdot 10^n & \text{if } a \geq 10. \end{cases}$$

The maximum absolute errors occur at the high points of the intervals, at a=10 and 100, and are 0.54 and 1.7 respectively. The maximum relative errors are at the endpoints of the intervals, at a=1, 10 and 100, and are 17% in both cases. 17% or 0.17 is larger than 1/10, so the method yields less than a decimal digit of accuracy.

Hyperbolic estimates

In some cases, hyperbolic estimates may be efficacious, because a hyperbola is also a convex curve and may lie along an arc of $Y = x^2$ better than a line. Hyperbolic estimates are more computationally complex, because they necessarily require a floating division. A near-optimal hyperbolic approximation to x^2 on the interval [1,100] is y=190/(10-x)-20. Transposing, the square root is x = -190/(y+20)+10. Thus for $S = a \cdot 10^{2n}$:

$$\sqrt{S} \approx \left(\frac{-190}{a + 20} + 10\right) \cdot 10^n$$

The floating division need be accurate to only one decimal digit, because the estimate overall is only that accurate, and can be done mentally. A hyperbolic estimate is better on average than scalar or linear estimates. It has maximum absolute error of 1.58 at 100 and maximum relative error of 16.0% at 10. For the worst case at a=10, the estimate is 3.67. If one starts with 10 and applies Newton-Raphson iterations straight away, two iterations will be required, yielding 3.66, before the accuracy of the hyperbolic estimate is exceeded. For a more typical case like 75, the hyperbolic estimate is 8.00, and 5 Newton-Raphson iterations starting at 75 would be required to obtain a more accurate result.

Arithmetic estimates

A method analogous to piece-wise linear approximation but using only arithmetic instead of algebraic equations, uses the multiplication tables in reverse: the square root of a number between 1 and 100 is between 1 and 10, so if we know 25 is a perfect square (5 × 5), and 36 is a perfect square (6 × 6), then the square root of a number greater than or equal to 25 but less than 36, begins with a 5. Similarly for numbers between other squares. This method will yield a correct first digit, but it is not accurate to one digit: the first digit of the square root of 35 for example, is 5, but the square root of 35 is almost 6.

A better way is to the divide the range into intervals half way between the squares. So any number between 25 and half way to 36, which is 30.5, estimate 5; any number greater than 30.5 up to 36, estimate 6.[Note 4] The procedure only requires a little arithmetic to find a boundary number in the middle of two products from the multiplication table. Here is a reference table of those boundaries:

a	nearest square	$k = \sqrt{a}$ est.
1 to 2.5	1 (= 1^2)	1
2.5 to 6.5	4 (= 2^2)	2
6.5 to 12.5	9 (= 3^2)	3
12.5 to 20.5	16 (= 4^2)	4
20.5 to 30.5	25 (= 5^2)	5
30.5 to 42.5	36 (= 6^2)	6
42.5 to 56.5	49 (= 7^2)	7
56.5 to 72.5	64 (= 8^2)	8
72.5 to 90.5	81 (= 9^2)	9
90.5 to 100	100 (= 10^2)	10

The final operation is to multiply the estimate k by the power of ten divided by 2, so for $S = a \cdot 10^{2n}$,

$$\sqrt{S} \approx k \cdot 10^n$$

The method implicitly yields one significant digit of accuracy, since it rounds to the best first digit.

The method can be extended 3 significant digits in most cases, by interpolating between the nearest squares bounding the operand. If $k^2 \leq a < (k+1)^2$, then $\sqrt{a}$ is approximately k plus a fraction, the difference between a and k^2 divided by the difference between the two squares:

$$\sqrt{a} \approx k + R \text{ where } R = \frac{(a - k^2)}{(k+1)^2 - k^2}$$

The final operation, as above, is to multiply the result by the power of ten divided by 2;

$$\sqrt{S} = \sqrt{a} \cdot 10^n \approx (k + R) \cdot 10^n$$

k is a decimal digit and R is a fraction that must be converted to decimal. It usually has only a single digit in the numerator, and one or two digits in the denominator, so the conversion to decimal can be done mentally.

Example: find the square root of 75. $75 = 75 \times 10^{2 \cdot 0}$, so a is 75 and n is 0. From the multiplication tables, the square root of the mantissa must be 8 point *something* because 8 × 8 is 64, but 9 × 9 is 81, too big, so k is 8; *something* is the decimal representation of R. The fraction R is 75 - k^2 = 11, the numerator, and 81 - k^2 = 17, the denominator. 11/17 is a little less than 12/18, which is 2/3s or .67, so guess .66 (it's ok to guess here, the error is very small). So the estimate is 8 + .66 = 8.66. $\sqrt{75}$ to three significant digits is 8.66, so the estmate is good to 3 significant digits. Not all such estimates using this method will be so accurate, but they will be close.

Binary estimates

When working in the binary numeral system (as computers do internally), by expressing S as $a \times 2^{2n}$ where $0.1_2 \leq a < 10_2$, the square root $\sqrt{S} = \sqrt{a} \times 2^n$ can be estimated as

$$\sqrt{S} \approx (0.485 + 0.485 \cdot a) \cdot 2^n$$

which is the least-squares regression line to 3 significant digit coefficients. $\sqrt{a}$ has maximum absolute error of 0.0408 at a=2, and maximum relative error of 3.0% at a=1. A computationally convenient rounded estimate (because the coefficients are powers of 2) is:

$$\sqrt{S} \approx (0.5 + 0.5 \cdot a) \cdot 2^n$$[Note 5]

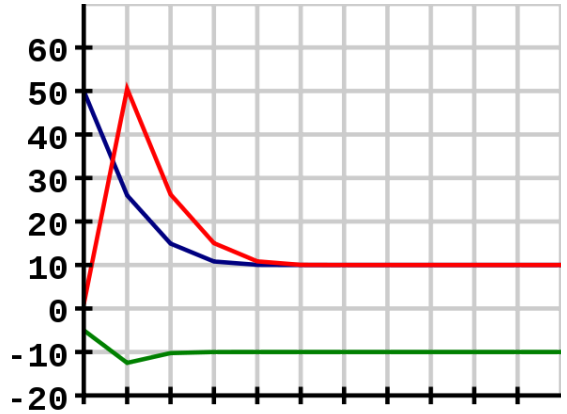
which has maximum absolute error of 0.086 at 2 and maximum relative error of 6.1% at a=0.5 and a=2.0.

For $S = 125348 = 1\ 1110\ 1001\ 1010\ 0100_2 = 1.1110\ 1001\ 1010\ 0100_2 \times 2^{16}$ the binary approximation gives $\sqrt{S} \approx (0.5 + 0.5 \cdot a) \cdot 2^8 = 1.0111\ 0100\ 1101\ 0010_2 \cdot 1\ 0000\ 0000_2 = 1.456 \cdot 256 = 372.8$. $\sqrt{125348} = 354.0$, so the estimate has an absolute error of 19 and relative error of 5.3%. The relative error is a little less than $1/2^4$, so the estimate is good to 4+ bits.

An estimate for a good to 8 bits can be obtained by table lookup on the high 8 bits of a, remembering that the high bit is implicit in most floating point representations, and the bottom bit of the 8 should be rounded. The table is 256 bytes of precomputed 8-bit square root values. For example, for the index 11101101_2 representing 1.8515625_{10}, the entry is 10101110_2 representing 1.359375_{10}, the square root of 1.8515625_{10} to 8 bit precision (2+ decimal digits).

Babylonian method

Perhaps the first algorithm used for approximating $\sqrt{S}$ is known as the **Babylonian method**, despite there being no direct evidence, beyond informed conjecture, that the eponymous Babylonian mathematicians employed exactly this method.[1] The method is also known as **Heron's method**, after the first-century Greek mathematician Hero of Alexandria who gave the first explicit description of the method in his AD 60 work *Metrica*.[2] The basic idea is that if x is an overestimate to the square root of a non-negative real number S then $\frac{S}{x}$ will be an underestimate, or vice versa, and so the average of these two numbers may reasonably be expected to provide a better approximation (though the formal proof of that assertion depends on the inequality of arithmetic and geometric means that shows this average is always an overestimate of the square root, as noted in the article on square roots, thus assuring convergence). This is equivalent to using Newton's method to solve $x^2 - S = 0$.



Graph charting the use of the Babylonian method for approximating a square root of 100 (±10) using starting values x_0 = 50, x_0 = 1, and x_0 = –5. Note that a positive starting value yields the positive root, and a negative starting value the negative root.

More precisely, if x is our initial guess of $\sqrt{S}$ and e is the error in our estimate such that $S = (x+e)^2$, then we can expand the binomial and solve for

$$e = \frac{S - x^2}{2x + e} \approx \frac{S - x^2}{2x}, \text{ since } e \ll x.$$

Therefore, we can compensate for the error and update our old estimate as

$$x + e \approx x + \frac{S - x^2}{2x} = \frac{S + x^2}{2x} = \frac{\frac{S}{x} + x}{2} \equiv x_{\text{revised}}$$

Since the computed error was not exact, this becomes our next best guess. The process of updating is iterated until desired accuracy is obtained. This is a quadratically convergent algorithm, which means that the number of correct digits of the approximation roughly doubles with each iteration. It proceeds as follows:

1. Begin with an arbitrary positive starting value x_0 (the closer to the actual square root of S, the better).
2. Let x_{n+1} be the average of x_n and $\frac{S}{x_n}$ (using the arithmetic mean to approximate the geometric mean).
3. Repeat step 2 until the desired accuracy is achieved.

It can also be represented as:

$$\begin{aligned} &x_0 \approx \sqrt{S}, \\ &x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right), \\ &\sqrt{S} = \lim_{n\to\infty} x_n. \end{aligned}$$

This algorithm works equally well in the p-adic numbers, but cannot be used to identify real square roots with p-adic square roots; one can, for example, construct a sequence of rational numbers by this method that converges to +3 in the reals, but to −3 in the 2-adics.

Example

To calculate $\sqrt{S}$, where S = 125348, to six significant figures, use the rough estimation method above to get

$$\begin{aligned} x_0 &= 6 \cdot 10^2 &&&= 600.000 \\ x_1 &= \tfrac{1}{2}\left(x_0 + \tfrac{S}{x_0}\right) &= \tfrac{1}{2}\left(600.000 + \tfrac{125348}{600.000}\right) &= 404.457 \\ x_2 &= \tfrac{1}{2}\left(x_1 + \tfrac{S}{x_1}\right) &= \tfrac{1}{2}\left(404.457 + \tfrac{125348}{404.457}\right) &= 357.187 \\ x_3 &= \tfrac{1}{2}\left(x_2 + \tfrac{S}{x_2}\right) &= \tfrac{1}{2}\left(357.187 + \tfrac{125348}{357.187}\right) &= 354.059 \\ x_4 &= \tfrac{1}{2}\left(x_3 + \tfrac{S}{x_3}\right) &= \tfrac{1}{2}\left(354.059 + \tfrac{125348}{354.059}\right) &= 354.045 \\ x_5 &= \tfrac{1}{2}\left(x_4 + \tfrac{S}{x_4}\right) &= \tfrac{1}{2}\left(354.045 + \tfrac{125348}{354.045}\right) &= 354.045 \end{aligned}$$

Therefore, $\sqrt{125348} \approx 354.045$.

Convergence

Suppose that $x_0 > 0$ and $S > 0$. Then for any natural number n, $x_n > 0$. Let the relative error in x_n be defined by

$$\varepsilon_n = \frac{x_n}{\sqrt{S}} - 1 > -1$$

and thus

$$x_n = \sqrt{S} \cdot (1 + \varepsilon_n).$$

Then it can be shown that

$$\varepsilon_{n+1} = \frac{\varepsilon_n^2}{2(1+\varepsilon_n)} \geq 0.$$

And thus that

$$\varepsilon_{n+2} \leq \min\left\{\frac{\varepsilon_{n+1}^2}{2}, \frac{\varepsilon_{n+1}}{2}\right\}$$

and consequently that convergence is assured, and quadratic.

Worst case for convergence

If using the rough estimate above with the Babylonian method, then the least accurate cases in ascending order are as follows:

$$\begin{array}{llll} S = 1; & x_0 = 2; & x_1 = 1.250; & \varepsilon_1 = 0.250. \\ S = 10; & x_0 = 2; & x_1 = 3.500; & \varepsilon_1 < 0.107. \\ S = 10; & x_0 = 6; & x_1 = 3.833; & \varepsilon_1 < 0.213. \\ S = 100; & x_0 = 6; & x_1 = 11.333; & \varepsilon_1 < 0.134. \end{array}$$

Thus in any case,

$$\begin{array}{l} \varepsilon_1 \leq 2^{-2}. \\ \varepsilon_2 < 2^{-5} < 10^{-1}. \\ \varepsilon_3 < 2^{-11} < 10^{-3}. \\ \varepsilon_4 < 2^{-23} < 10^{-6}. \\ \varepsilon_5 < 2^{-47} < 10^{-14}. \\ \varepsilon_6 < 2^{-95} < 10^{-28}. \\ \varepsilon_7 < 2^{-191} < 10^{-57}. \\ \varepsilon_8 < 2^{-383} < 10^{-115}. \end{array}$$

Rounding errors will slow the convergence. It is recommended to keep at least one extra digit beyond the desired accuracy of the x_n being calculated to minimize round off error.

Bakhshali method

This method for finding an approximation to a square root was described in an ancient Indian mathematical manuscript called the Bakhshali manuscript. It is equivalent to two iterations of the Babylonian method beginning with x_0. Thus, the algorithm is quartically convergent, which means that the number of correct digits of the approximation roughly quadruples with each iteration.[3] The original presentation, using modern notation, is as follows: To calculate $\sqrt{S}$, let $x_0{}^2$ be the initial approximation to S. Then, successively iterate as:

$$a_n = \frac{S - x_n^2}{2x_n},$$

$$b_n = x_n + a_n,$$

$$x_{n+1} = b_n - \frac{a_n^2}{2b_n}.$$

Written explicitly, it becomes

$$x_{n+1} = (x_n + a_n) - \frac{a_n^2}{2(x_n + a_n)}.$$

Let $x_0 = N$ be an integer which is the nearest perfect square to S. Also, let the difference $d = S - N^2$, then the first iteration can be written as:

$$\sqrt{S} \approx N + \frac{d}{2N} - \frac{d^2}{8N^3 + 4Nd} = \frac{8N^4 + 8N^2 d + d^2}{8N^3 + 4Nd} = \frac{N^4 + 6N^2 S + S^2}{4N^3 + 4NS} = \frac{N^2(N^2 + 6S) + S^2}{4N(N^2 + S)}.$$

This gives a rational approximation to the square root.

Example

Using the same example as given in Babylonian method, let $S = 125348.$ Then, the first iterations gives

$$x_0 = 600$$

$$a_1 = \frac{125348 - 600^2}{2 \times 600} = -195.543$$

$$b_1 = 600 + (-195.543) = 404.456$$

$$x_1 = 404.456 - \frac{(-195.543)^2}{2 \times 404.456} = 357.186$$

Likewise the second iteration gives

$$a_2 = \frac{125348 - 357.186^2}{2 \times 357.186} = -3.126$$

$$b_2 = 357.186 + (-3.126) = 354.06$$

$$x_2 = 354.06 - \frac{(-3.1269)^2}{2 \times 354.06} = 354.046$$

Digit-by-digit calculation

This is a method to find each digit of the square root in a sequence. It is slower than the Babylonian method, but it has several advantages:

- It can be easier for manual calculations.
- Every digit of the root found is known to be correct, i.e., it does not have to be changed later.
- If the square root has an expansion that terminates, the algorithm terminates after the last digit is found. Thus, it can be used to check whether a given integer is a square number.
- The algorithm works for any base, and naturally, the way it proceeds depends on the base chosen.

Napier's bones include an aid for the execution of this algorithm. The shifting *n*th root algorithm is a generalization of this method.

Basic principle

First, consider the case of finding the square root of a number *Z*, that is the square of a two-digit number *XY*, where *X* is the tens digit and *Y* is the units digit. Specifically:

$Z = (10X + Y)^2 = 100X^2 + 20XY + Y^2$

Now using the Digit-by-Digit algorithm, we first determine the value of *X*. *X* is the largest digit such that X^2 is less or equal to *Z* from which we removed the two rightmost digits.

In the next iteration, we pair the digits, multiply *X* by 2, and place it in the tenth's place while we try to figure out what the value of *Y* is.

Since this is a simple case where the answer is a perfect square root *XY*, the algorithm stops here.

The same idea can be extended to any arbitrary square root computation next. Suppose we are able to find the square root of *N* by expressing it as a sum of *n* positive numbers such that

$$N = (a_1 + a_2 + a_3 + \cdots + a_n)^2.$$

By repeatedly applying the basic identity

$$(x + y)^2 = x^2 + 2xy + y^2,$$

the right-hand-side term can be expanded as

$$\begin{aligned}
&(a_1 + a_2 + a_3 + \cdots + a_n)^2 \\
&= a_1^2 + 2a_1a_2 + a_2^2 + 2(a_1 + a_2)a_3 + a_3^2 + \cdots + a_{n-1}^2 + 2\left(\sum_{i=1}^{n-1} a_i\right) a_n + a_n^2 \\
&= a_1^2 + [2a_1 + a_2]a_2 + [2(a_1 + a_2) + a_3]a_3 + \cdots + \left[2\left(\sum_{i=1}^{n-1} a_i\right) + a_n\right] a_n.
\end{aligned}$$

This expression allows us to find the square root by sequentially guessing the values of a_is. Suppose that the numbers $a_1, \ldots, a_{m-1}$ have already been guessed, then the m-th term of the right-hand-side of above summation is given by $Y_m = [2P_{m-1} + a_m]a_m$, where $P_{m-1} = \sum_{i=1}^{m-1} a_i$ is the approximate square root found so far. Now each new guess a_m should satisfy the recursion

$$X_m = X_{m-1} - Y_m,$$

such that $X_m \geq 0$ for all $1 \leq m \leq n$, with initialization $X_0 = N$. When $X_n = 0$, the exact square root has been found; if not, then the sum of a_is gives a suitable approximation of the square root, with X_n being the approximation error.

For example, in the decimal number system we have

$$N = (a_1 \cdot 10^{n-1} + a_2 \cdot 10^{n-2} + \cdots + a_{n-1} \cdot 10 + a_n)^2,$$

where 10^{n-i} are place holders and the coefficients $a_i \in \{0, 1, 2, \ldots, 9\}$. At any m-th stage of the square root calculation, the approximate root found so far, P_{m-1} and the summation term Y_m are given by

$$P_{m-1} = \sum_{i=1}^{m-1} a_i \cdot 10^{n-i} = 10^{n-m+1} \sum_{i=1}^{m-1} a_i \cdot 10^{m-i-1},$$

$$Y_m = [2P_{m-1} + a_m \cdot 10^{n-m}]a_m \cdot 10^{n-m} = \left[20 \sum_{i=1}^{m-1} a_i \cdot 10^{m-i-1} + a_m\right] a_m \cdot 10^{2(n-m)}.$$

Here since the place value of Y_m is an even power of 10, we only need to work with the pair of most significant digits of the remaining term X_{m-1} at any m-th stage. The section below codifies this procedure.

It is obvious that a similar method can be used to compute the square root in number systems other than the decimal number system. For instance, finding the digit-by-digit square root in the binary number system is quite efficient since the value of a_i is searched from a smaller set of binary digits {0,1}. This makes the computation faster since at each stage the value of Y_m is either $Y_m = 0$ for $a_m = 0$ or $Y_m = 2P_{m-1} + 1$ for $a_m = 1$. The fact that we have only two possible options for a_m also makes the process of deciding the value of a_m at m-th stage of calculation easier. This is because we only need to check if $Y_m \leq X_{m-1}$ for $a_m = 1$. If this condition is satisfied, then we take $a_m = 1$; if not then $a_m = 0$. Also, the fact that multiplication by 2 is done by left bit-shifts helps in the computation.

Decimal (base 10)

Write the original number in decimal form. The numbers are written similar to the long division algorithm, and, as in long division, the root will be written on the line above. Now separate the digits into pairs, starting from the decimal point and going both left and right. The decimal point of the root will be above the decimal point of the square. One digit of the root will appear above each pair of digits of the square.

Beginning with the left-most pair of digits, do the following procedure for each pair:

1. Starting on the left, bring down the most significant (leftmost) pair of digits not yet used (if all the digits have been used, write "00") and write them to the right of the remainder from the previous step (on the first step, there will be no remainder). In other words, multiply the remainder by 100 and add the two digits. This will be the **current value *c***.
2. Find *p*, *y* and *x*, as follows:
   - Let ***p*** be the **part of the root found so far**, ignoring any decimal point. (For the first step, $p = 0$.)
   - Determine the greatest digit ***x*** such that $x(20p + x) \leq c$. We will use a new variable ***y*** = $x(20p + x)$.
     - Note: $20p + x$ is simply twice *p*, with the digit *x* appended to the right.
     - Note: *x* can be found by guessing what $c/(20 \cdot p)$ is and doing a trial calculation of *y*, then adjusting *x* upward or downward as necessary.
   - Place the digit x as the next digit of the root, i.e., above the two digits of the square you just brought down. Thus the next *p* will be the old *p* times 10 plus *x*.
3. Subtract *y* from *c* to form a new remainder.
4. If the remainder is zero and there are no more digits to bring down, then the algorithm has terminated. Otherwise go back to step 1 for another iteration.

Examples

Find the square root of 152.2756.

```
      1  2. 3  4
    ____________
   /
 \/  01 52.27 56
```

```
 01                  1*1 <= 1 < 2*2                x = 1
 01                    y = x*x = 1*1 = 1
 00 52               22*2 <= 52 < 23*3             x = 2
 00 44                 y = (20+x)*x = 22*2 = 44
    08 27            243*3 <= 827 < 244*4          x = 3
    07 29               y = (240+x)*x = 243*3 = 729
    98 56            2464*4 <= 9856 < 2465*5       x = 4
    98 56               y = (2460+x)*x = 2464*4 = 9856
    00 00            Algorithm terminates: Answer is 12.34
```

Binary numeral system (base 2)

Inherent to digit-by-digit algorithms is a search and test step: find a digit, e, when added to the right of a current solution r, such that $(r+e)\cdot(r+e) \le x$, where x is the value for which a root is desired. Expanding: $r\cdot r + 2re + e\cdot e \le x$. The current value of $r\cdot r$—or, usually, the remainder—can be incrementally updated efficiently when working in binary, as the value of e will have a single bit set (a power of 2), and the operations needed to compute $2\cdot r\cdot e$ and $e\cdot e$ can be replaced with faster bit shift operations.

Example

Here we obtain the square root of 81, which when converted into binary gives 1010001. The numbers in the left column gives the option between that number or zero to be used for subtraction at that stage of computation. The final answer is 1001, which in decimal is 9.

```
             1 0 0 1
            ---------
          √ 1010001

      1      1
             1
            ---------
      101     01
              0
            --------
      1001     100
                 0
            --------
      10001    10001
               10001
              -------
                   0
```

This gives rise to simple computer implementations:[4]

```
int32_t isqrt(int32_t num) {
    assert(("sqrt input should be non-negative", num > 0));
    int32_t res = 0;
    int32_t bit = 1 << 30; // The second-to-top bit is set.
                           // Same as ((unsigned) INT32_MAX + 1) / 2.

    // "bit" starts at the highest power of four <= the argument.
    while (bit > num)
        bit >>= 2;

    while (bit != 0) {
        if (num >= res + bit) {
            num -= res + bit;
            res = (res >> 1) + bit;
        } else
            res >>= 1;
        bit >>= 2;
    }
    return res;
}
```

Using the notation above, the variable "bit" corresponds to e_m^2 which is $(2^m)^2 = 4^m$, the variable "res" is equal to $2re_m$, and the variable "num" is equal to the current X_m which is the difference of the number we want the square root of and the square of our current approximation with all bits set up to 2^{m+1}. Thus in the first loop, we want to find the highest power of 4 in "bit" to find the highest power of 2 in e. In the second loop, if num is greater than res + bit, then X_m is greater than $2re_m + e_m^2$ and we can subtract it. The next line, we want to add e_m to r which means we want to add $2e_m^2$ to $2re_m$ so we want `res = res + bit<<1`. Then update e_m to e_{m-1} inside res which involves dividing by 2 or another shift to the right. Combining these 2 into one line leads to `res = res>>1 + bit`. If X_m isn't greater than $2re_m + e_m^2$ then we just update e_m to e_{m-1} inside res and divide it by 2. Then we update e_m to e_{m-1} in bit by dividing it by 4. The final iteration of the 2nd loop has bit equal to 1 and will cause update of e to run one extra time removing the factor of 2 from res making it our integer approximation of the root.

Faster algorithms, in binary and decimal or any other base, can be realized by using lookup tables—in effect trading more storage space for reduced run time.[5]

Exponential identity

Pocket calculators typically implement good routines to compute the exponential function and the natural logarithm, and then compute the square root of *S* using the identity found using the properties of logarithms ($\ln x^n = n \ln x$) and exponentials ($e^{\ln x} = x$):

$$\sqrt{S} = e^{\frac{1}{2} \ln S}.$$

The denominator in the fraction corresponds to the *n*th root. In the case above the denominator is 2, hence the equation specifies that the square root is to be found. The same identity is used when computing square roots with logarithm tables or slide rules.

A two-variable iterative method

This method is applicable for finding the square root of $0 < S < 3$ and converges best for $S \approx 1$. This, however, is no real limitation for a computer based calculation, as in base 2 floating point and fixed point representations, it is trivial to multiply S by an integer power of 4, and therefore $\sqrt{S}$ by the corresponding power of 2, by changing the exponent or by shifting, respectively. Therefore, S can be moved to the range $\frac{1}{2} \leq S < 2$. Moreover, the following method does not employ general divisions, but only additions, subtractions, multiplications, and divisions by powers of two, which are again trivial to implement. A disadvantage of the method is that numerical errors accumulate, in contrast to single variable iterative methods such as the Babylonian one.

The initialization step of this method is

$$a_0 = S$$
$$c_0 = S - 1$$

while the iterative steps read

$$a_{n+1} = a_n - a_n c_n / 2$$
$$c_{n+1} = c_n^2 (c_n - 3)/4$$

Then, $a_n \to \sqrt{S}$ (while $c_n \to 0$).

Note that the convergence of c_n, and therefore also of a_n, is quadratic.

The proof of the method is rather easy. First, rewrite the iterative definition of c_n as

$$1 + c_{n+1} = (1 + c_n)(1 - c_n/2)^2.$$

Then it is straightforward to prove by induction that

$$S(1 + c_n) = a_n^2$$

and therefore the convergence of a_n to the desired result $\sqrt{S}$ is ensured by the convergence of c_n to 0, which in turn follows from $-1 < c_0 < 2$.

This method was developed around 1950 by M. V. Wilkes, D. J. Wheeler and S. Gill[6] for use on EDSAC, one of the first electronic computers.[7] The method was later generalized, allowing the computation of non-square roots.[8]

Iterative methods for reciprocal square roots

The following are iterative methods for finding the reciprocal square root of *S* which is $1/\sqrt{S}$. Once it has been found, find $\sqrt{S}$ by simple multiplication: $\sqrt{S} = S \cdot (1/\sqrt{S})$. These iterations involve only multiplication, and not division. They are therefore faster than the Babylonian method. However, they are not stable. If the initial value is not close to the reciprocal square root, the iterations will diverge away from it rather than converge to it. It can therefore be advantageous to perform an iteration of the Babylonian method on a rough estimate before starting to apply these methods.

- Applying Newton's method to the equation $(1/x^2) - S = 0$ produces a method that converges quadratically using three multiplications per step:

$$x_{n+1} = \frac{x_n}{2} \cdot (3 - S \cdot x_n^2).$$

- Another iteration is obtained by Halley's method, which is the Householder's method of order two. This converges cubically, but involves four multiplications per iteration:

$y_n = S \cdot x_n^2$, and
$x_{n+1} = \frac{x_n}{8} \cdot (15 - y_n \cdot (10 - 3 \cdot y_n))$.

Goldschmidt's algorithm

Some computers use Goldschmidt's algorithm to simultaneously calculate $\sqrt{S}$ and $1/\sqrt{S}$. Goldschmidt's algorithm finds $\sqrt{S}$ faster than Newton-Raphson iteration on a computer with a fused multiply–add instruction and either a pipelined floating point unit or two independent floating-point units.[9]

The first way of writing Goldschmidt's algorithm begins

$b_0 = S$
$Y_0 \approx 1/\sqrt{S}$ (typically using a table lookup)
$y_0 = Y_0$
$x_0 = Sy_0$

and iterates

$$b_{n+1} = b_n Y_n^2$$
$$Y_{n+1} = (3 - b_{n+1})/2$$
$$x_{n+1} = x_n Y_{n+1}$$
$$y_{n+1} = y_n Y_{n+1}$$

until b_i is sufficiently close to 1, or a fixed number of iterations. The iterations converge to

$\lim_{n\to\infty} x_n = \sqrt{S}$, and
$\lim_{n\to\infty} y_n = 1/\sqrt{S}$.

Note that it is possible to omit either x_n and y_n from the computation, and if both are desired then $x_n = Sy_n$ may be used at the end rather than computing it through in each iteration.

A second form, using fused multiply-add operations, begins

$y_0 \approx 1/\sqrt{S}$ (typically using a table lookup)
$x_0 = Sy_0$
$h_0 = y_0/2$

and iterates

$$r_n = 0.5 - x_n h_n$$
$$x_{n+1} = x_n + x_n r_n$$
$$h_{n+1} = h_n + h_n r_n$$

until r_i is sufficiently close to 0, or a fixed number of iterations. This converges to

$\lim_{n\to\infty} x_n = \sqrt{S}$, and
$\lim_{n\to\infty} 2h_n = 1/\sqrt{S}$.

Taylor series

If N is an approximation to $\sqrt{S}$, a better approximation can be found by using the Taylor series of the square root function:

$$\sqrt{N^2 + d} = N \sum_{n=0}^{\infty} \frac{(-1)^n (2n)!}{(1-2n)n!^2 4^n} \frac{d^n}{N^{2n}} = N\left(1 + \frac{d}{2N^2} - \frac{d^2}{8N^4} + \frac{d^3}{16N^6} - \frac{5d^4}{128N^8} + \cdots\right)$$

As an iterative method, the order of convergence is equal to the number of terms used. With two terms, it is identical to the Babylonian method. With three terms, each iteration takes almost as many operations as the Bakhshali approximation, but converges more slowly. Therefore, this is not a particularly efficient way of calculation. To maximize the rate of convergence, choose N so that $\frac{|d|}{N^2}$ is as small as possible.

Continued fraction expansion

Quadratic irrationals (numbers of the form $\frac{a+\sqrt{b}}{c}$, where a, b and c are integers), and in particular, square roots of integers, have periodic continued fractions. Sometimes what is desired is finding not the numerical value of a square root, but rather its continued fraction expansion, and hence its rational approximation. Let S be the positive number for which we are required to find the square root. Then assuming a to be a number that serves as an initial guess and r to be the remainder term, we can write $S = a^2 + r$. Since we have $S - a^2 = (\sqrt{S} + a)(\sqrt{S} - a) = r$, we can express the square root of S as

$$\sqrt{S} = a + \frac{r}{a + \sqrt{S}}.$$

By applying this expression for $\sqrt{S}$ to the denominator term of the fraction, we have

$$\sqrt{S} = a + \frac{r}{a + (a + \frac{r}{a+\sqrt{S}})} = a + \frac{r}{2a + \frac{r}{a+\sqrt{S}}}.$$

Proceeding this way, we get a generalized continued fraction for the square root as

$$\sqrt{S} = a + \cfrac{r}{2a + \cfrac{r}{2a + \cfrac{r}{2a + \ddots}}}$$

The first step to evaluating such a fraction to obtain a root is to do numerical substitutions for the root of the number desired, and number of denominators selected. For example, in canonical form, r is 1 and for $\sqrt{2}$, a is 1, so the numerical continued fraction for 3 denominators is:

$$\sqrt{2} \approx 1 + \cfrac{1}{2 + \cfrac{1}{2 + \frac{1}{2}}}$$

Step 2 is to reduce the continued fraction from the bottom up, one denominator at a time, to yield a rational fraction whose numerator and denominator are integers. The reduction proceeds thus (taking the first three denominators):

$$1 + \cfrac{1}{2 + \cfrac{1}{2 + \frac{1}{2}}} = 1 + \cfrac{1}{2 + \frac{1}{\frac{5}{2}}}$$

Compact notation

The numerator/denominator expansion for continued fractions (see left) is cumbersome to write as well as to embed in text formatting systems. Therefore, special notation has been developed to compactly represent the integer and repeating parts of continued fractions. One such convention is use of a lexical "dog leg" to represent the vinculum between numerator and denominator, which allows the fraction to be expanded horizontally instead of vertically:

$$\sqrt{S} = a + \frac{r|}{|2a} + \frac{r|}{|2a} + \frac{r|}{|2a} + \cdots$$

Here, each vinculum is represented by three line segments, two vertical and one horizontal, separating r from $2a$.

An even more compact notation which omits lexical devices takes a special form:

$$[a; 2a, 2a, 2a, \ldots]$$

For repeating continued fractions (which all square roots do), the repetend is represented only once, with an overline to signify a non-terminating repetition of the overlined part:

$$[a; \overline{2a}]$$

For $\sqrt{2}$, the value of a is 1, so its representation is:

$$[1; \overline{2}]$$

$$= 1 + \frac{1}{2 + \frac{2}{5}} = 1 + \frac{1}{\frac{12}{5}}$$

$$= 1 + \frac{5}{12} = \frac{17}{12}$$

Finally (step 3), divide the numerator by the denominator of the rational fraction to obtain the approximate value of the root:

$17 \div 12 = 1.42$ rounded to three digits of precision.

The actual value of $\sqrt{2}$ is 1.41 to three significant digits. The relative error is 0.17%, so the rational fraction is good to almost three digits of precision. Taking more denominators gives successively better approximations: four denominators yields the fraction $\frac{41}{29} = 1.4137$, good to almost 4 digits of precision, etc.

Usually, the continued fraction for a given square root is looked up rather than expanded in place because it's tedious to expand it. Continued fractions are available for at least square roots of small integers and common constants. For an arbitrary decimal number, precomputed sources are likely to be useless. The following is a table of small rational fractions called *convergents* reduced from canonical continued fractions for the square roots of a few common constants:

$\sqrt{S}$	cont. fraction	~decimal	serial convergents
$\sqrt{2}$	$[1;\overline{2}]$	1.41421	$\frac{3}{2}, \frac{7}{5}, \frac{17}{12}, \frac{41}{29}, \frac{99}{70}, \frac{140}{99}$
$\sqrt{3}$	$[1;\overline{1,2}]$	1.73205	$\frac{2}{1}, \frac{5}{3}, \frac{7}{4}, \frac{19}{11}, \frac{26}{15}, \frac{71}{41}, \frac{97}{56}$
$\sqrt{5}$	$[2;\overline{4}]$	2.23607	$\frac{9}{4}, \frac{38}{17}, \frac{161}{72}$
$\sqrt{6}$	$[2;\overline{2,4}]$	2.44949	$\frac{5}{2}, \frac{22}{9}, \frac{49}{20}, \frac{218}{89}$
$\sqrt{10}$	$[3;\overline{6}]$	3.16228	$\frac{19}{6}, \frac{117}{37}$
$\sqrt{\pi}$	$[1;1,3,2,1,1,6...]$	1.77245	$\frac{2}{1}, \frac{7}{4}, \frac{16}{9}, \frac{23}{13}, \frac{39}{22}$
$\sqrt{e}$	$[1;1,1,1,5,1,1...]$	1.64872	$\frac{2}{1}, \frac{3}{2}, \frac{8}{5}, \frac{28}{17}, \frac{33}{20}, \frac{61}{37}$
$\sqrt{\phi}$	$[1;3,1,2,11,3,7...]$	1.27202	$\frac{4}{3}, \frac{5}{4}, \frac{14}{11}$

Note: all convergents up to and including denominator 99 listed.

In general, the larger the denominator of a rational fraction, the better the approximation. It can also be shown that truncating a continued fraction yields a rational fraction that is the best approximation to the root of any fraction with denominator less than or equal to the denominator of that fraction - e.g., no fraction with a denominator less than or equal to 99 is as good an approximation to $\sqrt{2}$ as 140/99.

Lucas sequence method

the Lucas sequence of the first kind $U_n(P,Q)$ is defined by the recurrence relations:

$$U_n(P,Q) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ P \cdot U_{n-1}(P,Q) - Q \cdot U_{n-2}(P,Q) & \text{Otherwise} \end{cases}$$

and the characteristic equation of it is:

$$x^2 - P \cdot x + Q = 0$$

it has the discriminant $D = P^2 - 4Q$ and the roots:

$$x_1 = \frac{P+\sqrt{D}}{2}, \quad x_2 = \frac{P-\sqrt{D}}{2}$$

all that yield the following positive value:

$$\lim_{n\to\infty} \frac{U_{n+1}}{U_n} = x_1$$

so when we want $\sqrt{a}$, we can choose $P = 2$ and $Q = 1 - a$, and then calculate $x_1 = 1 + \sqrt{a}$ using U_{n+1} and $U_n$for large value of n. The most effective way to calculate U_{n+1} and $U_n$is:

$$\begin{bmatrix} U_n \\ U_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -Q & P \end{bmatrix} \cdot \begin{bmatrix} U_{n-1} \\ U_n \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -Q & P \end{bmatrix}^n \cdot \begin{bmatrix} U_0 \\ U_1 \end{bmatrix}$$

Summary:

$$\begin{bmatrix} 0 & 1 \\ a-1 & 2 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} U_n \\ U_{n+1} \end{bmatrix}$$

then when $n \to \infty$:

$$\sqrt{a} = \frac{U_{n+1}}{U_n} - 1$$

Approximations that depend on the floating point representation

A number is represented in a floating point format as $m \times b^p$ which is also called scientific notation. Its square root is $\sqrt{m} \times b^{p/2}$ and similar formulae would apply for cube roots and logarithms. On the face of it, this is no improvement in simplicity, but suppose that only an approximation is required: then just $b^{p/2}$ is good to an order of magnitude. Next, recognise that some powers, *p*, will be odd, thus for $3141.59 = 3.14159 \times 10^3$ rather than deal with fractional powers of the base, multiply the mantissa by the base and subtract one from the power to make it even. The adjusted representation will become the equivalent of 31.4159×10^2 so that the square root will be $\sqrt{31.4159} \times 10$.

If the integer part of the adjusted mantissa is taken, there can only be the values 1 to 99, and that could be used as an index into a table of 99 pre-computed square roots to complete the estimate. A computer using base sixteen would require a larger table, but one using base two would require only three entries: the possible bits of the integer part of the adjusted mantissa are 01 (the power being even so there was no shift, remembering that a normalised floating point number always has a non-zero high-order digit) or if the power was odd, 10 or 11, these being the first *two* bits of the original mantissa. Thus, 6.25 = 110.01 in binary, normalised to 1.1001×2^2 an even power so the paired bits of the mantissa are 01, while .625 = 0.101 in binary normalises to 1.01×2^{-1} an odd power so the adjustment is to 10.1×2^{-2} and the paired bits are 10. Notice that the low order bit of the power is echoed in the high order bit of the pairwise mantissa. An even power has its low-order bit zero and the adjusted mantissa will start with 0, whereas for an odd power that bit is one and the adjusted mantissa will start with 1. Thus, when the power is halved, it is as if its low order bit is shifted out to become the first bit of the pairwise mantissa.

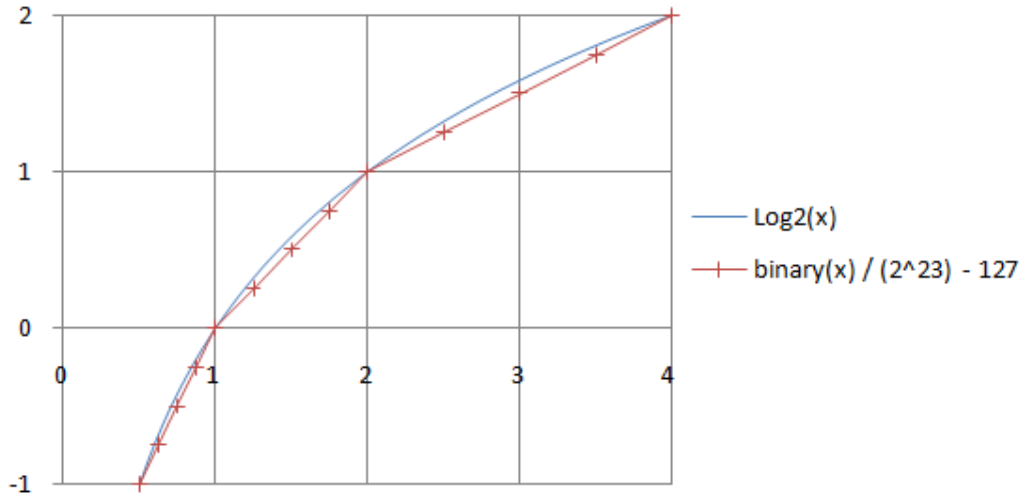
A table with only three entries could be enlarged by incorporating additional bits of the mantissa. However, with computers, rather than calculate an interpolation into a table, it is often better to find some simpler calculation giving equivalent results. Everything now depends on the exact details of the format of the representation, plus what operations are available to access and manipulate the parts of the number. For example, Fortran offers an `EXPONENT(x)` function to obtain the power. Effort expended in devising a good initial approximation is to be recouped by thereby avoiding the additional iterations of the refinement process that would have been needed for a poor approximation. Since these are few (one iteration requires a divide, an add, and a halving) the constraint is severe.

Many computers follow the IEEE (or sufficiently similar) representation, and a very rapid approximation to the square root can be obtained for starting Newton's method. The technique that follows is based on the fact that the floating point format (in base two) approximates the base-2 logarithm. That is $\log_2(m \times 2^p) = p + \log_2(m)$

So for a 32-bit single precision floating point number in IEEE format (where notably, the power has a bias of 127 added for the represented form) you can get the approximate logarithm by interpreting its binary representation as a 32-bit integer, scaling it by 2^{-23}, and removing a bias of 127, i.e.

$$x_{\text{int}} \cdot 2^{-23} - 127 \approx \log_2(x).$$

For example, 1.0 is represented by a hexadecimal number 0x3F800000, which would represent $1065353216 = 127 \cdot 2^{23}$ if taken as an integer. Using the formula above you get $1065353216 \cdot 2^{-23} - 127 = 0$, as expected from $\log_2(1.0)$. In a similar fashion you get 0.5 from 1.5 (0x3FC00000).



To get the square root, divide the logarithm by 2 and convert the value back. The following program demonstrates the idea. Note that the exponent's lowest bit is intentionally allowed to propagate into the mantissa. One way to justify the steps in this program is to assume b is the exponent bias and n is the number of explicitly stored bits in the mantissa and then show that

$$(((x_{\text{int}}/2^n - b)/2) + b) \cdot 2^n = (x_{\text{int}} - 2^n)/2 + ((b+1)/2) \cdot 2^n.$$

```c
/* Assumes that float is in the IEEE 754 single precision floating point format
 * and that int is 32 bits. */
float sqrt_approx(float z) {
    int val_int = *(int*)&z; /* Same bits, but as an int */
    /*
     * To justify the following code, prove that
     *
     * ((((val_int / 2^m) - b) / 2) + b) * 2^m = ((val_int - 2^m) / 2) + ((b + 1) / 2) * 2^m)
     *
     * where
     *
     * b = exponent bias
     * m = number of mantissa bits
     *
     * .
     */

    val_int -= 1 << 23; /* Subtract 2^m. */
    val_int >>= 1; /* Divide by 2. */
    val_int += 1 << 29; /* Add ((b + 1) / 2) * 2^m. */

    return *(float*)&val_int; /* Interpret again as float */
}
```

The three mathematical operations forming the core of the above function can be expressed in a single line. An additional adjustment can be added to reduce the maximum relative error. So, the three operations, not including the cast, can be rewritten as

```
val_int = (1 << 29) + (val_int >> 1) - (1 << 22) + a;
```

where a is a bias for adjusting the approximation errors. For example, with a = 0 the results are accurate for even powers of 2 (e.g., 1.0), but for other numbers the results will be slightly too big (e.g.,1.5 for 2.0 instead of 1.414... with 6% error). With a = -0x4B0D2, the maximum relative error is minimized to ±3.5%.

If the approximation is to be used for an initial guess for Newton's method to the equation $(1/x^2) - S = 0$, then the reciprocal form shown in the following section is preferred.

Reciprocal of the square root

A variant of the above routine is included below, which can be used to compute the reciprocal of the square root, i.e., $x^{-\frac{1}{2}}$ instead, was written by Greg Walsh. The integer-shift approximation produced a relative error of less than 4%, and the error dropped further to 0.15% with one iteration of Newton's method on the following line.[10] In computer graphics it is a very efficient way to normalize a vector.

```
float invSqrt(float x) {
    float xhalf = 0.5f*x;
    union {
        float x;
        int i;
    } u;
    u.x = x;
    u.i = 0x5f375a86 - (u.i >> 1);
    /* The next line can be repeated any number of times to increase accuracy */
    u.x = u.x * (1.5f - xhalf * u.x * u.x);
    return u.x;
}
```

Some VLSI hardware implements inverse square root using a second degree polynomial estimation followed by a Goldschmidt iteration.[11]

Negative or complex square

If $S < 0$, then its principal square root is

$$\sqrt{S} = \sqrt{|S|}\, i\,.$$

If $S = a+bi$ where a and b are real and $b \neq 0$, then its principal square root is

$$\sqrt{S} = \sqrt{\frac{|S|+a}{2}} + \operatorname{sgn}(b)\sqrt{\frac{|S|-a}{2}}\, i\,.$$

This can be verified by squaring the root.[12][13] Here

$$|S| = \sqrt{a^2+b^2}$$

is the modulus of S. The principal square root of a complex number is defined to be the root with the non-negative real part.

See also

- Alpha max plus beta min algorithm
- *n*th root algorithm
- Square root of 2

Notes

1. In addition to the principal square root, there is a negative square root equal in magnitude but opposite in sign to the principal square root, except for zero, which has double square roots of zero.

2. The factors two and six are used because they approximate the geometric means of the lowest and highest possible values with the given number of digits: $\sqrt{\sqrt{1}\cdot\sqrt{10}} = \sqrt[4]{10} \approx 1.78$ and $\sqrt{\sqrt{10}\cdot\sqrt{100}} = \sqrt[4]{1000} \approx 5.62$.
3. The unrounded estimate has maximum absolute error of 2.65 at 100 and maximum relative error of 26.5% at y=1, 10 and 100
4. If the number is exactly half way between two squares, like 30.5, guess the higher number which is 6 in this case
5. This is incidentally the equation of the tangent line to $y=x^2$ at y=1.

References

1. Fowler, David; Robson, Eleanor (1998). "Square Root Approximations in Old Babylonian Mathematics: YBC 7289 in Context". *Historia Mathematica*. **25** (4): 376. doi:10.1006/hmat.1998.2209 (https://doi.org/10.1006%2Fhmat.1998.2209).
2. Heath, Thomas (1921). *A History of Greek Mathematics, Vol. 2* (https://archive.org/details/ahistorygreekma00heatgoog). Oxford: Clarendon Press. pp. 323 (https://archive.org/details/ahistorygreekma00heatgoog/page/n340)–324.
3. Bailey, David; Borwein, Jonathan (2012). "Ancient Indian Square Roots: An Exercise in Forensic Paleo-Mathematics" (http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/india-sqrt.pdf) (PDF). *American Mathematical Monthly*. **119** (8). pp. 646–657. Retrieved 2017-09-14.
4. Fast integer square root by Mr. Woo's abacus algorithm (archived) (https://web.archive.org/web/20120306040058/http://medialab.freaknet.org/martin/src/sqrt/sqrt.c)
5. Integer Square Root function (http://atoms.alife.co.uk/sqrt/SquareRoot.java)
6. M. V. Wilkes, D. J. Wheeler and S. Gill, "The Preparation of Programs for an Electronic Digital Computer", Addison-Wesley, 1951.
7. M. Campbell-Kelly, "Origin of Computing", Scientific American, September 2009.
8. J. C. Gower, "A Note on an Iterative Method for Root Extraction", The Computer Journal 1(3):142–143, 1958.
9. Markstein, Peter (November 2004). *Software Division and Square Root Using Goldschmidt's Algorithms* (http://www.informatik.uni-trier.de/Reports/TR-08-2004/rnc6_12_markstein.pdf) (PDF). 6th Conference on Real Numbers and Computers (http://cca-net.de/rnc6/). Dagstuhl, Germany. CiteSeerX 10.1.1.85.9648 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.9648).
10. Fast Inverse Square Root (http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf) by Chris Lomont
11. "High-Speed Double-Precision Computation of Reciprocal, Division, Square Root and Inverse Square Root" (http://portal.acm.org/citation.cfm?id=627261) by José-Alejandro Piñeiro and Javier Díaz Bruguera 2002 (abstract)
12. Abramowitz, Miltonn; Stegun, Irene A. (1964). *Handbook of mathematical functions with formulas, graphs, and mathematical tables* (https://books.google.com/books?id=MtU8uP7XMvoC). Courier Dover Publications. p. 17. ISBN 978-0-486-61272-0., Section 3.7.26, p. 17 (http://www.math.sfu.ca/~cbm/aands/page_17.htm)
13. Cooke, Roger (2008). *Classical algebra: its nature, origins, and uses* (https://books.google.com/books?id=lUcTsYopfhkC). John Wiley and Sons. p. 59. ISBN 978-0-470-25952-8., Extract: page 59 (https://books.google.com/books?id=lUcTsYopfhkC&pg=PA59)

External links

- Weisstein, Eric W. "Square root algorithms" (https://mathworld.wolfram.com/SquareRootAlgorithms.html). *MathWorld*.
- Square roots by subtraction (http://www.afjarvis.staff.shef.ac.uk/maths/jarvisspec02.pdf)
- Integer Square Root Algorithm by Andrija Radović (http://www.andrijar.com/algorithms/algorithms.htm#qusr)

- Personal Calculator Algorithms I : Square Roots (William E. Egbert), Hewlett-Packard Journal (may 1977) : page 22 (http://www.hparchive.com/Journals/HPJ-1977-05.pdf)
- Calculator to learn the square root (http://www.calculatorsquareroot.com)
