Personal Calculator Algorithms I: Square Roots

A detailed description of the algorithm used in Hewlett-Packard hand-held calculators to compute square roots.

by William E. Egbert

BEGINNING WITH THE HP-35, all HP personal calculators have used essentially the same algorithms for computing complex mathematical functions in their BCD (binary-coded decimal) microprocessors. While improvements have been made in newer calculators, the changes have affected primarily special cases and not the fundamental algorithms.

This article is the first of a series that examines these algorithms and their implementation. Each article will present in detail the methods used to implement a common mathematical function. For simplicity, rigorous proofs will not be given, and special cases other than those of particular interest will be omitted.

Although tailored for efficiency within the environment of a special-purpose BCD microprocessor, the basic mathematical equations and the techniques used to transform and implement them are applicable to a wide range of computing problems and devices.

The Square Root Algorithm

This article will discuss the algorithm and methods used to implement the square root function.

The core of the square root algorithm is a simple approximation technique tailored to be efficient using the instruction set of a BCD processor. The technique is as follows:

\[ \sqrt{x} \text{ is desired} \]
1. Guess an answer \( a \)
2. Generate \( a^2 \)
3. Find \( R = x - a^2 \)
4. If the magnitude of \( R \) is sufficiently small, \( a = \sqrt{x} \).
5. If \( R \) is a positive number, \( a \) is too small.
   If \( R \) is a negative number, \( a \) is too big.
6. Depending on the result of step 5, modify \( a \) and return to step 2.

The magnitude of \( R \) will progressively decrease until the desired accuracy is reached.

This procedure is only a rough outline of the actual square root routine used. The first refinement is to avoid having to find \( a^2 \) and \( x - a^2 \) each time \( a \) is changed. This is done by finding \( a \) one decade at a time. In other words, find the hundreds digit of \( a \), then the tens digit, the units digit, and so on. Once the hundreds digit is found, it is squared and subtracted from \( x \), and the tens digit is found. This process, however, is not exactly straightforward, so some algebra is in order.

The following definitions will be used:
- \( x \) = the number whose square root is desired
- \( a \) = most significant digit(s) of \( \sqrt{x} \) previously computed
- \( b \) = the next digit of \( \sqrt{x} \) to be found
- \( j \) = the power of 10 associated with \( b \)
- \( R_a = x - a^2 \), the current remainder
- \( a_l \) = the new \( a \) when digit \( b \) is added in its proper place. \( a_l = a + (b \times 10^j) \) \hspace{1cm} (1)
- \( R_b \) = the portion of remainder \( R_a \) that would be removed by adding \( b \) to \( a \). \( R_b = a_l^2 - a^2 \) \hspace{1cm} (2)

For example, let \( x = 54756 \). Then \( \sqrt{x} = 234 \).

Let \( a = 200 \).

\( b \) = the digit we are seeking (3, in this case)
\( j \) = 1 (the 10's digit is being computed)

\( R_a = 54756 - (200)^2 = 14756 \).

Note that \( a_l \) and \( R_b \) will vary with the choice of \( b \).

The process of finding \( \sqrt{x} \) one decade at a time approaches the value of \( \sqrt{x} \) from below. That is, at any point in the computation, \( a \leq \sqrt{x} \). Consequently, \( R_a > 0 \).

With this in mind it is easy to see that for any decade \( j \), the value of \( b \) is the largest possible digit so that

\[ R_a - R_b \geq 0 \]

or

\[ R_b \leq R_a. \] \hspace{1cm} (3)

Using equations 1 and 2 we have

\[ R_b = [a + (b \times 10^j)]^2 - a^2. \]

Expanding and simplifying,

\[ R_b = 2ab 	imes 10^j + (b \times 10^j)^2. \] \hspace{1cm} (4)

Inserting (4) into (3) yields the following rule for finding digit \( b \).

Digit \( b \) is the largest possible digit so that

\[ 2ab \times 10^j + (b \times 10^j)^2 \leq R_a \] \hspace{1cm} (5)

When the digit that satisfies equation 5 is found, a new \( a \) is formed by adding \( b \times 10^j \) to the old \( a \), the decade counter \( j \) is decremented by 1, and a new \( R_a \) is created; the new \( R_a \) is the old \( R_a \) minus \( R_b \).
Continuing the previous example,
\[ x = 54756 \]
\[ j = 1 \]
\[ a = 200 \]
\[ x - a^2 = R_a = 14756 \]
Applying equation 5 to find \( b \):
\[
R_b = \begin{array}{ccc}
0 & 0 & 14756 \\
1 & 4100 & 10656 \\
2 & 8400 & 6356 \\
3 & 12900 & 1856 \\
4 & 17600 & -2844 \\
\end{array}
\]
Thus \( b = 3 \), since \( b = 4 \) causes overdraft, i.e., \( R_a - R_b < 0 \). The new \( a = 200 + 3 \times 10 \times 1 = 230 \). The new \( R_a = 1856 \), the new \( j = 0 \). With these new parameters, the units digit can be found.

This process may seem vaguely familiar, which is not surprising since upon close inspection it turns out to be the (usually forgotten) scheme taught in grade school to find square roots longhand. Of course, trailing zeros and digits are not written in the longhand scheme.

To make this process efficient for a calculator, still another refinement is needed.

\((b \times 10^j)^2\) can be expressed as a series, using the fact that the square of an integer \( b \) is equal to the sum of the first \( b \) odd integers. Thus,
\[
(b \times 10^j)^2 = b^2 \times 10^{2j} = \sum_{i=1}^{b} (2i-1) \times 10^{2j}
\]
For example,
\[
(3 \times 10^1)^2 = 1 \times 10^{2j} + 3 \times 10^{2j} + 5 \times 10^{2j} = 9 \times 10^{2j}
\]
Thus \(2ab \times 10^j + (b \times 10^j)^2\) can be expressed as:
\[
2ab \times 10^j + (b \times 10^j)^2 = \sum_{i=1}^{b} 2a \times 10^j + (2i-1) \times 10^{2j}
\]
or
\[
R_b = \sum_{i=1}^{b} 2a \times 10^j + (2i-1) \times 10^{2j}
\]

Now comes a key transformation in the square root routine. It was shown earlier how inequality 3 will give the value \( b \) for the next digit of \( a \). Since multiplying both sides of an inequality by a positive constant does not change the inequality, equations 3 and 6 can be multiplied by the number 5.

\[
5R_b \leq 5R_a
\]
\[
5R_b = \sum_{i=1}^{b} 10a \times 10^j + (10i-5) \times 10^{2j}
\]
\( b \) becomes the largest digit so that \( 5R_b \leq 5R_a \). The new \( 5R_a \) is equal to the old \( 5R_a \) minus \( 5R_b \).

These transformations may seem useless until we examine a few examples of the last term of the right side of (7) for various values of \( b \).

\[
\begin{array}{c|c|c}
\hline
b & 10a \times 10^j + 15 \times 10^{2j}, b = 1 & 10a \times 10^j + 25 \times 10^{2j}, b = 2 \\
\hline
1 & 20500 & 31780 \\
2 & 21500 & 31780 \\
3 & 22500 & 9280 new \ 5R_a \\
4 & 23500 & -14220 overdraft \ \\
\hline
\end{array}
\]

Notice that the two-digit coefficient of \( 10^j \) consists of \( (b-1) \) and a 5. These two digits will be expressed as \( (b-1) \times 5 \) in succeeding equations. \( 10a \) is formed by a simple right shift and does not change between terms. If the sum defined in equation 7, as \( b \) is incremented by 1, is subtracted from \( 5R_a \) until overdraft occurs, the digit in the next-to-last digit position is \( b \). Best of all, it is in the exact position to form the next digit of \( a \) without further manipulation. Redoing the previous example may help clarify matters.

\[
\begin{array}{c|c|c}
\hline
R_a & 14756 & 9280 new \ 5R_a \\
j & 1 & 0 \\
a & 200 & 230 \\
5R_a & 73780 & -14220 overdraft \ \\
\hline
\end{array}
\]

Notice that when overdraft occurs the new value of \( a \) is already created and the new value of \( 5R_a \) can be found by restoring the previous remainder.

Decrementing the value of \( j \) would cause, in effect, \( (10a \times 10^j) \) to shift right one place, and \( (b-1) \times 5 \times 10^{2j} \) to shift right two places. The result is that the final 5 shifts one place to the right to make room for a new digit. Continuing with the same example,

\[
\begin{array}{c|c|c}
\hline
5R_a & 9280 & \\
a & 230 & \\
j & 0 & \\
\hline
\end{array}
\]
For ease of understanding, the preceding example treated a large positive number. A number in the calculator actually consists of a mantissa between 1 and 10 and an exponent. The problem is to find the square root of both parts of this argument. Happily, if the input exponent is an even number, the portion of the answer resulting from it turns out to be the exponent of the final answer and is simply the input exponent divided by 2. Thus to find $\sqrt{x}$, the exponent of $x$ is first made even and the mantissa shifted to keep the number the same. The exponent of $\sqrt{x}$ is found by dividing the corrected input exponent by 2. The method described above is then used to find the square root of the shifted input mantissa, which (after possibly being shifted) can be between 1 and 100. The result will then be between 1 and 10, which is the range required for the mantissa of $\sqrt{x}$.

During the process of finding $\sqrt{x}$ the remainder $R_b$ progressively decreases. To avoid losing accuracy, this remainder is multiplied by $10^N$ after finding each new digit $b$. This avoids shifting $a$ at all, once the square root extraction process begins. A 12-digit mantissa is generated, which insures accuracy to $\pm 1$ in the tenth digit of the mantissa of $\sqrt{x}$.

In summary, the computation of $\sqrt{x}$ proceeds as follows:

1. Generate exponent of answer.
2. Multiply mantissa by 5 to create original $5R_a$.
3. With an original $a$ of 0, use the method described above to find 12 $b$ digits to form the mantissa of the answer.
4. Round the mantissa and attach the exponent found previously.
5. Display the answer.

The calculator is now ready for another operation.

References
Personal Calculator Algorithms II: Trigonometric Functions

A detailed explanation of the algorithms used by HP hand-held calculators to compute sine, cosine, and tangent.

by William E. Egbert

BEGINNING WITH THE HP-35, all HP personal calculators have used essentially the same algorithms for computing complex mathematical functions in their BCD (binary-coded decimal) microprocessors. While improvements have been made in newer calculators, the changes have affected primarily special cases and not the fundamental algorithms.

This article is the second of a series that examines these algorithms and their implementation. Each article will present in detail the methods used to implement a common mathematical function. For simplicity, rigorous proofs will not be given, and special cases other than those of particular interest will be omitted.

Although tailored for efficiency within the environment of a special-purpose BCD microprocessor, the basic mathematical equations and the techniques used to transform and implement them are applicable to a wide range of computing problems and devices.

The Trigonometric Function Algorithm

This article will discuss the method of generating sine, cosine, and tangent. To minimize program length, a single function, \( \tan \theta \), is generated first. Once \( \tan \theta \) is calculated, \( \sin \theta \) is found by the formula

\[
\sin \theta = \frac{\pm \tan \theta}{\sqrt{1 + \tan^2 \theta}}.
\]

It turns out (as will be explained later) that \( \cot \theta \) can easily be generated while generating \( \tan \theta \). Then \( \cos \theta \) is calculated using the formula

\[
\cos \theta = \frac{\pm \cot \theta}{\sqrt{1 + \cot^2 \theta}}.
\]

It can be seen that these formulas are identical, except for the cotangent replacing the tangent. Thus the same routine can solve for either sine or cosine depending on whether the argument is tangent or cotangent.

Scaling

Since \( \theta \) and \( \theta + n(360^\circ) \) yield identical trigonometric functions, every angular argument is resolved to a positive angle between 0° and 360°. For reasons to be explained later, all calculations assume angles expressed in radians. An angle in degrees is first converted to radians by:

\[
\theta_{\text{rad}} = \theta_{\text{deg}} \times \pi/180.
\]

Angles expressed in grads are also converted using the appropriate scale factor.

Once \( \theta \) is in radians, \( 2\pi \) is subtracted repeatedly from \( |\theta| \) until the absolute remainder is between 0 and \( 2\pi \). For large angles this would take a long time. In such cases \( 2\pi \times 10^n \) can be subtracted in a process similar to division. Suppose an angle \( \theta \) is expressed in scientific notation (e.g., \( 8.5 \times 10^3 \)). \( 2\pi \times 10^n \), or \( 6.28... \times 10^n \) is then repeatedly subtracted from \( \theta \) until the result becomes negative (underflow). Thus \( 6.28... \times 10^n \) is subtracted from \( 8.5 \times 10^3 \) twice and underflow occurs. \( 6.28... \times 10^n \) is then added to the negative remainder to give a number between 0 and \( 2\pi \), in this case \( 2.2 \times 10^4 \). The remainder is expressed now as \( 22 \times 10^4 \) and the process is repeated, this time subtracting \( 2\pi \times 10^4 \). With this method, large angles are quickly resolved.

The problem with this scaling process is that in current computers numbers can be expressed only to a limited number of digits, so \( 2\pi \) and therefore \( 2\pi \times 10^n \) cannot be expressed exactly. Error creeps in with each shift of the remainder. Thus, the larger the angle, the fewer significant digits remain in the scaled result. A rule of thumb for rough estimates is that for each count in the exponent, one digit of accuracy will be lost. For example, \( 5 \times 10^5 \) when scaled will lose five digits of accuracy.

A negative argument is treated the same as a positive number until the end, when the scaling routine returns a number between 0 and \(-2\pi\). Then \( 2\pi \) is added to the negative result, giving again a number between 0 and \( 2\pi \). This addition of \( 2\pi \) causes a digit
to be lost, which results in asymmetry such as cos(86°) ≠ cos(-86°). Newer calculators obviate this problem by scaling to a number between 0 and π/4.

Vector Rotation

An angle can be expressed as a vector having X and Y components and a resultant R (see Fig. 1). If R is the unit vector, then X = cos θ and Y = sin θ. However, regardless of the length of R, Y/X = tan θ and X/Y = cot θ. This holds true for all values of θ from 0 to 2π. Thus, if some way could be found to generate X and Y for a given θ, all the trigonometric functions could be found.

In vector geometry a useful formula results when one rotates a vector through a given angle. Let us suppose we have a vector whose angle is θ₁, and we know its components X₁ and Y₁ (see Fig. 2). The X₂ and Y₂ that result when the vector is rotated an additional angle θ₂ are given by:

\[
X₂ = X₁ \cos θ₂ - Y₁ \sin θ₂ \\
Y₂ = Y₁ \cos θ₂ + X₁ \sin θ₂
\]

Dividing both sides of these equations by cos θ₂ gives:

\[
\frac{X₂}{\cos θ₂} = X₁ - Y₁ tan θ₂ = X₂' \\
\frac{Y₂}{\cos θ₂} = Y₁ + X₁ tan θ₂ = Y₂'
\] (1)

Note that X₂' and Y₂', while not the true values of X₂ and Y₂, both differ by the same factor, cos θ₂. Thus Y₂'/X₂' = Y₂/X₂. From Fig. 2 it is plain that the quotient Y₂'/X₂' is equal to tan (θ₁ + θ₂). Thus the tangent of a large angle can be found by manipulating smaller angles whose sum equals the large one. Returning to equation 1 above, it can be seen that to generate X₂' and Y₂', X₁ and Y₁ need to be multiplied by tan θ₂ and added or subtracted as needed. If θ₂ is chosen so that tan θ₂ is a simple power of 10 (i.e., 1, 0.1, 0.01, ...,) then the multiplications simply amount to shifting X₁ and Y₁. Thus to generate X₂' and Y₂', only a shift and an add or subtract are needed.

Pseudo-Division

The tangent of θ is found as follows. First θ is divided into a sum of smaller angles whose tangents are powers of 10. The angles are tan⁻¹ (1) = 45°, tan⁻¹ (0.1) = 5.7°, tan⁻¹ (0.01) = 0.57°, tan⁻¹ (0.001) = 0.057°, tan⁻¹ (0.0001) = 0.0057°, and so on.

This process is called pseudo-division. First, 45° is subtracted from θ until overdraft, keeping track of the number of subtractions. The remainder is restored by adding 45°. Then 5.7° is repeatedly subtracted, again keeping track of the number of subtractions. This process is repeated with smaller and smaller angles. Thus:

\[
θ = q₀ \tan^{-1} (1) + q₁ \tan^{-1} (0.1) + q₂ \tan^{-1} (0.01) + \ldots + q_r
\]

The coefficients qᵢ refer to the number of subtractions possible in each decade. Each qᵢ is equal to or less than 10, so it can be stored in a single four-bit digit.

This process of pseudo-division is one reason that all the trigonometric functions are done in radians. For accuracy, tan⁻¹(10⁻ⁿ) needs to be expressed to ten digits. In degrees, these constants are random digits and require considerable ROM (read-only memory) space to store. However, in radians, they become, for the most part, nines followed by sixes. Because of this, they can be generated arithmetically, thus using fewer ROM states. Also, in radians, tan⁻¹ (1) = π/4, which is needed anyway to generate π. The problem with using radians is that since π is an irrational number, scaling errors occur as discussed earlier. This means cardinal points do not give exact answers. For example, sin (720°) ≠ 0 when calculated this way but rather 4×10⁻⁹. See reference 3 for a discussion of this point.
So far, a pseudo-quotient has been generated that represents the division of the given angle \( \theta \) into smaller angles whose tangents are powers of 10. In many HP calculators the pseudo-quotient is five hexadecimal digits long. Each digit represents a series of subtractions and is a number from 0 to 10. For example, if \( \theta \) were \( 359.9999^\circ \), the pseudo-quotient would be \( 77877 \), representing \( \theta = 7\tan^{-1}(1) + 7\tan^{-1}(0.1) + 8\tan^{-1}(0.01) + 7\tan^{-1}(0.001) + 7\tan^{-1}(0.0001) \). There may also be a remainder \( r \), which is the angle remaining after the previous partial quotient subtractions have taken place.

Tan \( \theta \) can now be found using the vector rotation process discussed earlier.

**Pseudo-Multiplication**

To use equation 1 we need an initial \( X_1 \) and \( Y_1 \). These correspond to the \( X \) and \( Y \) of the residual angle \( r \) discussed previously. This angle is small (less than 0.001°), and for small angles in radians, \( \sin \theta = \theta \) (another reason to use radians instead of degrees). Thus, to good accuracy, the initial \( Y_1 \) can be set to the residual angle, and the initial \( X_1 \) set to 1. Equation 1 can now be repeatedly applied, where \( \theta_2 \) is the angle whose tangent is \( 10^{-j} \). Each time equation 1 is applied, a new \( X_1 \) and \( Y_1 \) are generated, i.e., \( X_2 \) and \( Y_2 \). The number of times equation 1 is applied is determined by the count in the pseudo-quotient digit for that \( \theta \). Thus if the original angle had a 3 in the pseudo-quotient digit corresponding to \( \tan^{-0.1} \), or 5.7°, equation 1 would be applied three times with \( X_1 \) and \( Y_1 \) being shifted one place right for \( \tan^{-0.1} \) before the addition or subtraction. In this manner, new \( X_1 \) and \( Y_1 \) are formed as the vector is rotated the amount corresponding to the count in the pseudo-quotient digits which, of course, sum to the original angle \( \theta \).

Equation 1 shows that to generate \( X_2 \) requires a shift of \( Y_1 \) and a subtraction from \( X_1 \). Likewise \( Y_2 \) requires a shift of \( X_1 \) and an addition to \( Y_1 \). To implement this would require either two extra registers to hold the shifted values of \( X_1 \) and \( Y_1 \), or else shifting one register twice and the other once. It would be desirable to shift only one register once. Happily, this is possible. Consider the following: Let \( Y = 123 \) and \( X = 456 \). Suppose we want \( Y + (X \times 0.01) \). This can be obtained by keeping the decimal points in the same places and shifting \( X \) two places right.

\[
\begin{array}{c}
123 \\
+ \quad 4.56 \\
\hline
127.56
\end{array}
\]

Now suppose instead of shifting \( X \) two places right, we multiply \( Y \) by 100, shifting it two places left. What happens?

The digits in both answers are exactly the same. The only difference between the two is that the second answer is 100 times the correct value, which is the same value by which \( Y \) was multiplied before the addition. Thus to avoid shifting \( X \), \( Y \) must be multiplied by \( 10^j \).

Expanding this method to the problem at hand also helps us solve another problem, that of accuracy. During pseudo-division, the angle \( \theta \) is resolved until a small angle \( r \) is left as the original \( Y \) value. Since this is done in fixed point arithmetic, zero digits are generated following the decimal point (e.g., \( .00123 \)). Since zero digits do not convey information except to indicate the decimal point, the remainder is shifted left one place (multiplied by 10) during each decade of pseudo-division. This preserves an extra digit of accuracy with each decade. The final remainder is equal to \( r \times 10^4 \) if the pseudo-quotient is five digits long.

To demonstrate mathematically the implementation that requires only a single register shift, return to equation 1 and replace \( \tan \theta_2 \) by \( 10^{-j} \). This substitution is legal because \( \theta_2 = \tan^{-1}(10^{-j}) \), where \( j \) is the decade digit.

\[
\begin{align*}
X_2' &= X_1 - Y_1 \times 10^{-j} \\
Y_2' &= Y_1 + X_1 \times 10^{-j}
\end{align*}
\]

Now let \( Z = Y_1 \times 10^j \), or \( Y_1 = Z \times 10^{-j} \). Substituting in equation 2,

\[
\begin{align*}
X_1' &= X_1 - Z \times 10^{-2j} \\
Y_1' &= Z \times 10^{-j} + X_1 \times 10^{-j}
\end{align*}
\]

Multiplying the second equation by \( 10^j \) gives:

\[
Y_1' \times 10^j = Z + X_1
\]

The left-hand side \((Y_1' \times 10^j)\) is in the correct form to be the new \( Z \) for the next iteration. Thus for each iteration within a decade:

\[
\begin{align*}
X_2' &= X_1 - Z \times 10^{-2j} \\
Y_2' &= Z \times 10^{-j} + X_1 \times 10^{-j}
\end{align*}
\]

Now suppose that for the first iteration, the original \( j \) is 4.

Since the shifted remainder \((r \times 10^4)\) is desired as \( Z \) for the first iteration, the original \( j \) is 4.

To implement equation 3, \( X_1 \) and \( Z \) are stored in two registers, \( Z \times 10^{-2j} \) is formed and stored in a third register. \( X_1 \) is added to \( Z \) to form the new \( Z \). This leaves \( X_1 \) undisturbed so that \( Z \times 10^{-2j} \) can be subtracted from it to form the new \( X_2' \).

This implementation saves extra shifts and increases accuracy by removing leading zeros in \( Z \). The only register shifted is \( Z \).
After equation 3 has been applied the number of times indicated by one pseudo-quotient digit, Z is shifted right one place, and a new pseudo-quotient digit is fetched. This in effect creates $Y_j \times 10^j$, where $j$ is one less than before. Again equation 3 is applied, and the process is repeated until all five pseudo-quotient digits have been exhausted. The result is an $X$ and a $Y$ that are proportional to the cosine and sine of the angle $\theta$. Because the final $j$ is zero, the final $Y (=Z)$ is correctly normalized with respect to $X$.

So far, then, an $X$ and a $Y$ have been generated by a pseudo-multiply operation consisting of shifts and additions. If $\tan \theta$ is required, $Y/X$ is generated, which is the correct answer. For $\sin \theta$, $Y/X$ is calculated, and for $\cos \theta$, $X/Y$ is calculated. Then either $X/Y$ or $Y/X$ is operated on by the routine described at the beginning of this article. The only difference between the computation for $\sin \theta$ and that for $\cos \theta$ is whether $X$ and $Y$ are exchanged.

In summary, the computation of trigonometric functions proceeds as follows:
1. Scale the input angle to a number in radians between 0 and $2\pi$.
2. Using the pseudo-division process divide the scaled number into groups of selected smaller angles.
3. With the pseudo-multiply process of equation 3 applied once for each angle resulting from the division of the input argument, generate an $X$ and a $Y$ that are proportional to the sine and cosine of the input angle.
4. With $X$ and $Y$, compute the required function using elementary operations.
5. Round and display the answer.

The calculator is now ready for another operation.

References

William E. Egbert
Bill Egbert is a project leader at HP's Corvallis, Oregon Division. He produced this series of algorithm articles as part of his work on the HP-67 and HP-97 Programmable Calculators. He was project leader for the HP-67 and did microprogramming for both calculators. Bill received a BSEE degree from Brigham Young University in 1973 and his MSEE from Stanford University in 1976. He’s been with HP since 1973. Born in Fallon, Nevada, he’s married, has two small children, and lives in Corvallis.
Personal Calculator Algorithms III: Inverse Trigonometric Functions

A detailed description of the algorithms used in Hewlett-Packard hand-held calculators to compute arc sine, arc cosine, and arc tangent.

by William E. Egbert

BEGINNING WITH THE HP-35, all HP personal calculators have used essentially the same algorithms for computing complex mathematical functions in their BCD (binary-coded decimal) microprocessors. While improvements have been made in newer calculators, the changes have affected primarily special cases and not the fundamental algorithms.

This article is the third of a series that examines these algorithms and their implementation. Each article presents in detail the methods used to implement a common mathematical function. For simplicity, rigorous proofs are not given, and special cases other than those of particular interest are omitted.

Although tailored for efficiency within the environment of a special-purpose BCD microprocessor, the basic mathematical equations and the techniques used to transform and implement them are applicable to a wide range of computing problems and devices.

Inverse Trigonometric Functions

This article will discuss the method of generating sin⁻¹, cos⁻¹, and tan⁻¹. An understanding of the trigonometric function algorithm is assumed. This was covered in the second article of this series and the detailed discussion will not be repeated here.

To minimize program length, the function tan⁻¹ A is always computed, regardless of the inverse trigonometric function required. If sin⁻¹ A is desired, A/√(1 - A²) is computed first, since

\[ \sin^{-1} A = \tan^{-1} \frac{A}{\sqrt{1 - A^2}}. \]

For cos⁻¹ A, sin⁻¹ A is computed as above and then cos⁻¹ A is calculated using

\[ \cos^{-1} A = \pi/2 - \sin^{-1} A. \]

Cos⁻¹ is found in the range 0 ≤ θ ≤ π and sin⁻¹ and tan⁻¹ are computed for the range -π/2 ≤ θ ≤ π/2. The tan⁻¹ routine solves only for angles between 0 and π/2, since -tan A = tan (-A). Thus A may be assumed to be positive and the sign of the input argument becomes the sign of the answer. All angles are calculated in radians and converted to degrees or grads if necessary.

General Algorithm

A vector rotation process similar to that used in the trigonometric routine is used in the inverse process as well. A vector expressed in its X and Y components can easily be rotated through certain specific angles using nothing more than shifts and adds of simple integers. In the algorithm for tan⁻¹ |A|, the input argument is |A|, or |tan θ|, where θ is the unknown. Letting tan θ = Y₁/X₁, |A| can be expressed as |A|/1, where Y₁ = |A| and X₁ = 1. A vector rotation process (see Fig. 1) is then used to rotate the vector clockwise through a series of successively smaller angles θᵢ, counting the number of rotations for each angle, until the Y₂ component approaches zero. If qᵢ denotes the number of rotations for θᵢ then

\[ |\theta| = q₀ + q₁θ₁ + ... + qᵢθᵢ + ... \]

This process is described in detail below.

Vector Rotation

To initialize the algorithm, A and 1 are stored in fixed-point format in registers corresponding to Y₁

![Fig. 1. Vector rotation.](image)
and $X_1$. This is done in such a way as to preserve as many digits of $A$ as possible when the exponent of $A$ differs from zero.

At this point the sign of $A$ is saved and $Y_1 = |A|$. Now comes the vector rotation (see Fig. 1). If the vector $R$ is rotated in a clockwise direction, $Y_2$ becomes smaller and smaller until it passes zero and becomes negative. As soon as $Y_2$ becomes negative, we know that we have rotated $R$ just past the desired angle $\theta$. Thus to find $\theta$, $R$ is simply rotated clockwise until $Y_2$ becomes negative. The amount of rotation is remembered and is equal to the desired angle $\theta = \tan^{-1} |A|$. To rotate $R$, the following formula is used.

$$X_2 = X_1 + Y_1 \tan \theta_2 = X_2' \quad (1)$$
$$Y_2 = Y_1 - X_1 \tan \theta_2 = Y_2'$$

This equation is the same as equation 1 of the article on trigonometric functions,\(^4\) except that the plus and minus signs are exchanged because $R$ is rotated in the opposite direction. As before, $\tan \theta_2$ is chosen such that the implementation requires a simple shift and add ($\tan \theta_2 = 10^{-1}$). To find $\theta$, $R$ is initially rotated with $\tan \theta_2 = 1$ ($\theta_2 = 45^\circ$). $Y_2'$ soon becomes negative and the number of successful rotations is stored as the first digit of what is known as the pseudo-quotient. $Y_2'$ is then restored to the last value it had before becoming negative and $R$ is rotated again, this time through a smaller angle, i.e., $\tan \theta_2 = 0.1$ ($\theta_2 = 5.7^\circ$). This process is repeated with the angle of rotation becoming smaller and smaller until five pseudo-quotient digits have been generated.

At the end of each series of rotations, $Y_2$ is multiplied by 10 to preserve accuracy.

**Pseudo-Multiplication**

It is now time to shift gears and add up all the small angles represented by the pseudo-quotient digits. There remains a residual angle $r$, represented by the final $X_2'$ and $Y_2'$. Since the residual angle is small, we would like to say $Y_2' = \sin r = r$. However, this is true only if $X_2' = 1$. Unfortunately, $X_2'$ in this case is the product of all the $1/\cos \theta$ terms resulting from several applications of equation 1. However, $Y_2'$ is this same product times $Y_2$. Thus $Y_2'/X_2' = Y_2/1$. Therefore, the final $Y_2'$ is divided by the final $X_2'$ and the result is $\sin r$, which for small angles in radians is approximately equal to $r$, the residual angle.

With the residual angle as the first partial sum, $\theta$ is generated by adding the angles represented by the digits of the pseudo-quotient. This is exactly the reverse of the pseudo-division operation in the trigonometric routine. Thus:

$$\theta = q_0 \tan^{-1} (1) + q_1 \tan^{-1} (0.1) + q_2 \tan^{-1} (0.01) + \ldots + r \quad (2)$$

Each coefficient $q_i$ refers to the count in a particular pseudo-quotient digit.

The result of this summation process, also called pseudo-multiplication, is an angle $\theta$ that is equal to $\tan^{-1} |A|$, where $|A|$ is the input argument to the $\tan^{-1}$ routine. At this point the original sign of $A$ is appended to $\theta$. For $\tan^{-1}$ this angle is normalized, converted to degrees or grads if necessary, and displayed. Recall that for $\sin^{-1}$, $A/\sqrt{1-A^2}$ was first generated. Thus for $\sin^{-1}$, the result of the $\tan^{-1}$ routine is again simply normalized, converted to degrees or grads if necessary, and displayed. For $\cos^{-1}$, the $\tan^{-1}$ routine returns $\sin^{-1}$. $\cos^{-1}$ is then simply found as $\pi/2 - \sin^{-1} A$.

**Summary**

In summary, the computation of inverse trigonometric functions proceeds as follows:

1. Calculate $A/\sqrt{1-A^2}$ if the desired function is $\sin^{-1} A$ or $\cos^{-1} A$.
2. Place $|A|$ and 1 in fixed-point format into appropriate registers, while preserving the sign of $A$.
3. Repeatedly rotate the vector with $A=Y$ and 1=$X$ clockwise using equation 1 until $Y$ approaches zero. The number of rotations and the amount of each rotation is stored as a pseudo-quotient along the way.
4. Using the pseudo-multiplication process of equation 2, sum all of the angles used in the rotation to form $|\theta|$.
5. Append the proper sign to the answer and calculate $\cos^{-1} A = \pi/2 - \sin^{-1} A$ if required.
6. Convert to the selected angle mode, and round and display the answer.

The calculator is now ready for another operation.

**References**


Bill Egbert is a project manager at HP's Corvallis (Oregon) Division.
BEGINNING WITH THE HP-35,1,2 all HP personal calculators have used essentially the same algorithms for computing complex mathematical functions in their BCD (binary-coded decimal) microprocessors. While improvements have been made in newer calculators,3 the changes have affected primarily special cases and not the fundamental algorithms.

This article is the fourth in a series that examines these algorithms and their implementation.4'5'6 Each article presents in detail the methods used to implement a common mathematical function. For simplicity, rigorous proofs are not given and special cases other than those of particular interest are omitted.

Although tailored for efficiency within the environment of a special-purpose BCD microprocessor, the basic mathematical equations and the techniques used to transform and implement them are applicable to a wide range of computing problems and devices.

The Logarithmic Function Algorithm

This article will discuss the method of generating the ln(x) and log_{10}(x) functions. To minimize program length, a single function, ln(x), is always computed first. Once ln(x) is calculated, log_{10}(x) is found by the formula

\[ \log_{10}(x) = \frac{\ln(x)}{\ln(10)} \]

\[ \ln(x) \text{ is generated using an approximation process much the same as the one used to compute trigonometric functions.}^{5} \]

The fundamental equation used in this case is the logarithmic property that

\[ \ln(a_1 \cdot a_2 \cdot a_3 \cdot \ldots \cdot a_n) = \ln(a_1) + \ln(a_2) + \ln(a_3) + \ldots + \ln(a_n) \] (1)

This algorithm simply transforms the input number x into a product of several terms whose logarithms are known. The sum of the logarithms of these various partial-product terms forms ln(x).

Exponent

Numbers in HP calculators are stored in scientific notation in the form \( x = M \cdot 10^K \). M is a number whose magnitude is between 1.00 and 9.999999999 and K is an integer between -99 and +99. Using equation 1, it is easy to see that

\[ \ln(M \cdot 10^K) = \ln(M) + \ln(10^K) \]

At this point, another logarithmic property becomes useful, which is

\[ \ln(A^b) = b \cdot \ln(A) \]

Using this relationship

\[ \ln(M \cdot 10^K) = \ln(M) + K \cdot \ln(10) \]

Thus to find the logarithm of a number in scientific notation, one calculates the logarithm of the mantissa of the number and adds that to the exponent times ln(10).

Mantissa

The problem of finding ln(x) is now reduced to finding the logarithm of its mantissa M.

Let \( P = \frac{1}{M} \). Then

\[ \ln(PM) = \ln(P) + \ln(M) \]

\[ \ln(1) = \ln(P) + \ln(M) \]

\[ 0 = \ln(P) + \ln(M) \]

\[ -\ln(P) = \ln(M) \] (2)

This may appear to be a useless exercise since at first glance \(-\ln(P)\) seems to be as hard to compute as ln(M).

Suppose, however, that a new number \( P_n \) is formed by multiplying \( P \) by \( r \) which is a small number close to 1.

\[ P_n = P \cdot r \]

In addition, let \( P_n \) be defined as a product of powers
of numbers $a_i$ whose natural logarithms are known.

$$P_n = a_0^{k_0} \cdot a_1^{k_1} \cdot ... \cdot a_j^{k_j} \cdot ... \cdot a_n^{k_n}$$

Thus

$$P = P_n/r$$

$$\ln(P) = \ln(P_n) - \ln(r)$$

Using equation 2

$$\ln(M) = \ln(r) - \ln(P_n)$$

Finally

$$\ln(M) = \ln(r) - (K_0\ln(a_0) + K_1\ln(a_1) + ... + K_j\ln(a_j) + ... + K_n\ln(a_n))$$

Thus to find $\ln(M)$ one simply multiplies $M$ by the carefully selected numbers $a_i$ so that the product $MP_n$ is forced to approach 1. If all the logarithms of $a_i$ are added up along the way to form $\ln(P_n)$ then $\ln(M)$ is the logarithm of the remainder $r$ minus this sum. Notice that the remainder $r$ is nothing more than the final product $MP_n$.

**Implementation**

How is this algorithm implemented in a special-purpose microprocessor? First of all, the terms of $P_n$ were chosen to reduce computation time and minimize the amount of ROM (read-only memory) needed to store $a_i$ and its logarithm. The numbers chosen for the $a_i$ terms are of the form $a_i = (1 + 10^{-j})$, where $j = 0-4$ (see Table 1).

<table>
<thead>
<tr>
<th>$j$</th>
<th>$a_i$</th>
<th>$\ln a_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0.6931</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>0.09531</td>
</tr>
<tr>
<td>2</td>
<td>1.01</td>
<td>0.009950</td>
</tr>
<tr>
<td>3</td>
<td>1.001</td>
<td>0.00009995</td>
</tr>
<tr>
<td>4</td>
<td>1.0001</td>
<td>0.000009995</td>
</tr>
</tbody>
</table>

Table 1  Values of $a_i$ Terms

To achieve high accuracy using relatively few $a_i$ terms, an approximation is used when $r = MP_n$ approaches 1. For numbers close to 1, $\ln(r) = r - 1$. This yields

$$\ln M \approx (r - 1) - \sum_{j=0}^{n} K_j\ln(a_j)$$  (3)

Since all of the $a_i$ terms are larger than 1, $M$ must be

between 0 and 1 if the product $P_nM$ is to approach 1. As $M$ is defined to be between 1 and 10, a new quantity $A$ is formed by dividing $M$ by 10. $A$ is now in the proper range (0.1 ≤ $A$ < 1) so that using the $a_i$ terms as defined will cause the product $AP_n$ to approach 1 without exceeding 1.

The product $P_n$ can now be formally defined as a series, where $j$ goes from 0 to $n$. Each partial product $AP_j$ has the form

$$A \cdot P_j = A \cdot P_{j-1}(1 + 10^{-j})^{K_j}, j = 0, 1, 2, ..., n$$

$P_{-1} = 1$, and $K_j$ is the largest integer such that $P_j < 1$.

In practice, each $A \cdot P_j$ is formed by multiplying $A \cdot P_{j-1}$ by $(1 + 10^{-j})$, $K_j$ times. There is one intermediate product, $T_j$, for each count of $K_j$, as shown below.

$$T_0 = A(1 + 10^{-0})$$
$$T_1 = A(1 + 10^{-1})$$
$$T_{K_0} = A(1 + 10^{-0})^{K_0}$$
$$T_{K_0+1} = A(1 + 10^{-0})^{K_0}(1 + 10^{-1})$$
$$T_m = A(1 + 10^{-0})^{K_0}(1 + 10^{-1})^{K_1}$$
$$... (1 + 10^{-n})^{K_n} = AP_n$$

$m = K_0 + K_1 + ... + K_n$

$$T_1 = T_{i-1}(1 + 10^{-i})$$ for some $j$  (4)

Notice that each multiplication of the intermediate product $T_{i-1}$ by $a_i$ simply amounts to shifting $T_{i-1}$ right the number of digits denoted by the current value of $j$ and adding the shifted value to the original $T_{i-1}$. This very efficient multiplication method is similar to the pseudo-multiplication of the trigonometric algorithm.5

**An Example**

A numeric example to illustrate this process is now in order. Let $A = 0.155$. To compute $\ln(A)$, $A$ must be multiplied by factors of $a_i$ until $AP_n$ approaches 1.

To begin the process $A = 0.155$ is multiplied by $a_0 = 2$ to form the intermediate product $T_0 = 0.31$. Another multiplication by $a_0$ gives $T_1 = 0.62$. A third multiplication by 2 results in 1.24, which is larger than 1. Thus $K_0 = 2$ and $AP_0 = 0.62$. The process is continued in Table 2.
Table 2 Generation of ln(0.155)

<table>
<thead>
<tr>
<th>j</th>
<th>a_j</th>
<th>AP_j</th>
<th>K_j</th>
<th>T_j</th>
<th>ln(a_j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0.155</td>
<td>0.155</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0.62</td>
<td>2</td>
<td>0.62</td>
<td>0.6931</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>1</td>
<td>0.682</td>
<td>0.0953</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>2</td>
<td>0.7502</td>
<td>0.0953</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>3</td>
<td>0.82522</td>
<td>0.0953</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>4</td>
<td>0.9077</td>
<td>0.0953</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>0.9985</td>
<td>5</td>
<td>0.9985</td>
<td>0.0953</td>
</tr>
<tr>
<td>2</td>
<td>1.01</td>
<td>0.9985</td>
<td>0</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.001</td>
<td>0.9995</td>
<td>1</td>
<td>0.9995</td>
<td>0.000099</td>
</tr>
<tr>
<td>4</td>
<td>1.0001</td>
<td>0.9996</td>
<td>1</td>
<td>0.9996</td>
<td>0.00009</td>
</tr>
</tbody>
</table>

0.9996 = A·P_4 = r  
1.8638 = ∑ ln(a_j)

*Another ×2 would result in AP_3 > 1. Thus a_j is changed to 1.1.
**The 1.01 constant is skipped entirely.

Applying the values found in Table 2 to equation 3 results in

\[
\ln(0.155) = (0.9996 - 1) - 1.8638 = -1.8642
\]

This answer approximates very closely the correct 10-digit answer of -1.864330162.

This example demonstrates the simplicity of this method of logarithm generation. All that is required is a multiplication (shift and add) and a test for 1. To implement this process using only three working registers, a pseudo-quotient similar to the one generated in the trigonometric algorithm is formed. Each digit represents the number of successful multiplications by a particular a_j. For the preceding example, the pseudo-quotient would be

2 5 0 1 1
↑ ↑ ↑ ↑ ↑

j = 0  j = 1  j = 2  j = 3  j = 4

With -ln(r) = (r - 1) as the first term, the appropriate logarithms of (a_j) are then summed according to the count in the pseudo-quotient digit corresponding to the proper a_j. The final sum is -ln(A).

At this point one more transformation is needed to optimize this algorithm perfectly to the microprocessor’s capabilities. Recall that the factors a_j were chosen to force the product P_nA towards 1. Suppose B_j = T_j - 1. Forcing B_m towards 0 causes P_nA to be forced to 1. Substituting B_j into (4) and simplifying yields

\[
(B_j + 1) = (B_{j-1} + 1)(1 + 10^{-j}) \text{ for some } j
\]

\[
B_j + 1 = B_{j-1}(1 + 10^{-j}) + 1 + 10^{-j}
\]

\[
B_j = B_{j-1}(1 + 10^{-j}) + 10^{-j}
\]

Multiplying through by -1 results in the following equation, which is equivalent to equation 4.

\[
-B_j = -B_{j-1}(1 + 10^{-j}) - 10^{-j} \text{ for some } j
\]

This expression is now in a very useful form, since the a_j term is the same as before, but the zero test is performed automatically when the 10^{-j} subtraction is done. A test for a borrow is all that is required. An additional benefit of this transformation is that accuracy can be increased by shifting -B_j left one digit for each a_j term after it has been applied the maximum number of times possible. This increases accuracy by replacing zeros generated as B_j approaches zero with significant digits that otherwise would have been lost out of the right end of the register. This shifting, which is equivalent to a multiplication by 10^j, gives yet another benefit. Multiplying equation 5 by 10^j and simplifying,

\[
-B_j \times 10^j = (-B_{j-1}(1 + 10^{-j}) - 10^{-j}) \times 10^j
\]

\[
-B_j \times 10^j = -B_{j-1} \times 10^j (1 + 10^{-j}) - 1 \text{ for some } j
\]

Notice that the 10^{-j} subtraction reduces to a simple -1 regardless of the value of j. The formation of the initial -B_0 = (A - 1) = 1 - A. This is formed by taking the 10's complement of M (the original mantissa), creating 10 - M. A right shift divides this by 10 to give 1 - M/10 = 1 - A = -B_0. A final, almost incredible, benefit of the B_j transformation is that the final remainder -B_m × 10^j is in the exact form required to be the first term of the summation process of equation 4 without further modification. The correct ln(a_j) constants are added directly to -B_m × 10^j, shifting the sum right one digit after each pseudo-quotient digit to preserve accuracy and restore the proper normalized form disrupted by equation 6. The result is -ln(A).

Finally, the required ln(M) is easily found by subtracting the computed result -ln(A) from ln(10).
\[
\ln(10) - (-\ln(A)) = \ln(10) + \ln(M/10) \\
= \ln(10 \cdot M/10) \\
= \ln(M)
\]

Once \(\ln(M)\) is computed, \(K \cdot \ln(10)\) is added as previously discussed to form \(\ln(x)\). At this point \(\log(x)\) can be generated by dividing \(\ln(x)\) by \(\ln(10)\).

**Summary**

In summary, the computation of logarithmic functions proceeds as follows:
1. Find the logarithm of \(10^K\) using \(K \cdot \ln(10)\).
2. Transform the input mantissa to the proper form required by \(-B_0\).
3. Apply equation 6 repeatedly and form a pseudo-quotient representing the number of successful multiplications by each \(a_i\).
4. Form \(-\ln(A)\) by summing the \(\ln(P_i)\) constants corresponding to the pseudo-quotient digits with the remainder \(-B_m \times 10^{l}\) as the first term in the series.
5. Find \(\ln(x)\) or \(\log(x)\) using simple arithmetic operations.
6. Round and display the answer.

The calculator is now ready for another operation. 

**References**