Three Early Algorithms

Keith S. Reid-Green

Three well-known algorithms, implemented separately within a span of 20 years, demonstrate the use of mathematics developed hundreds of years ago.

The following three algorithms don't have much in common except for two things: They were implemented between 1949 and 1965—during the so-called first generation of computing hardware—and they are good examples of rigorous thinking. First, I explain Bresenham's algorithm and apply it to a fundamental process of computer graphics—drawing a line on a raster screen. Second, although there are several ways to calculate a square root, the square root algorithm is interesting because it takes advantage of hardware design. And, finally, while the calculation of π goes back more than 2,200 years, Machin's algorithm from 1706 is of importance to the history of computing.

Bresenham's algorithm

Cathode-ray tube (CRT) displays for computers were developed principally along two lines, vector and raster, until the 1970s, when memory became inexpensive. Initially, the advantage of a vector display was that it could draw a line by passing its end points to analog circuitry, which moved an electron beam across the CRT. To draw a line on a raster display, on the other hand, required calculating the locations of every point that the line passed through and illuminating each one.

Raster technology won out when it became economically feasible to draw a display in a dedicated memory and refresh the CRT from it. Even though raster computation is more complicated, vector display time varied with the complexity of the diagram, making realistic animation, for example, impossible.

In the 1950s, the IBM 704 and 709 computers included a CRT display with an addressable raster of $1{,}024 \times 1{,}024$ dots. Once illuminated, a dot would persist on the screen for about two seconds. The CPU—excessively slow by today's standards—controlled the display, which had no refresh buffer. The 704, for example, had a 12-microsecond cycle, meaning that the fastest instruction executed in 24 microseconds. Floating-point calculations were much slower; a floating divide instruction took 18 cycles, or 216 microseconds.

Each dot was activated by the program's building a 36-bit word containing the X- and Y-address of the dot, with the X-address in bits 8–17 and the Y-address in bits 26–35 and "copying" the word to the CRT (see Figure 1). Obviously, the fastest way to refresh the screen was to create a word for each illuminated dot and to set up a "copy" loop. However, the maximum memory size for the 704 was 32,000 words, including the program, so prestoring all the dots wasn't always feasible.

In actual use, the visible CRT was a "slave" to another unit in a black box with a shutterless 35-mm camera. The general purpose of this arrangement was to enable the camera to take a picture of the master screen by illuminating dots only once. Nevertheless, the process of calculating the positions of dots was so slow that only a few dots could be displayed before the first one began to fade. There was a demonstration program—a tic-tac-toe game—for the display, and even this simple picture flickered badly.

It was in this context, around 1960, that computer scientist Jack Bresenham[1] considered how to minimize the calculation time for the dots that make up a line. For a line with a slope between 0 and 1, the Y-difference between consecutive points on a straight line is less than or equal to 1. For the line between $X1,Y1$ and $X2,Y2$, such that $0 \le \Delta Y \le \Delta X$ and $X1 < X2$, the X-addresses of the dots are $X1$, $X1 + 1$, $X1 + 2$, … , $X2$. If $\Delta X = X2 - X1$ and $\Delta Y = Y2 - Y1$, then for each step in X, the values of Y are $Y1$, $Y1 + \Delta Y/\Delta X$, $Y1 + 2\Delta Y/\Delta X$, … ,$Y2$; but raster point addresses are integers, so we

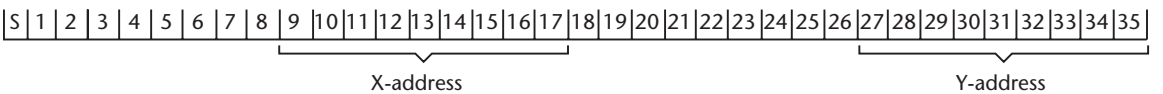


Figure 1. This 36-bit word activates a dot of a raster display.

1058-6180/02/$17.00 © 2002 IEEE

use $Y1$, int($Y1 + \Delta Y/\Delta X$), int($Y1 + 2\Delta Y/\Delta X$), and so forth, without rounding.

In the following implementation of Bresenham's algorithm, Ratio and Delta are real numbers; all other variables are integers. Note that only addition and subtraction are used in the For … Next loop.

```
DeltaX = X2 - X1
DeltaY = Y2 - Y1
Ratio = DeltaY / DeltaX
Delta = Ratio - 0.5

j = Y1
For i = X1 to X2
   Call Dot (i, j)
   If Delta > 0 Then
      Delta = Delta - 1.0
      j = j + 1
   End If
   Delta = Delta + Ratio
Next i
```

The For … Next loop is executed once per dot, and so this part of the code should be minimized. This loop would have taken about 60 cycles, or 720 microseconds. Because the raster size of the master CRT was 3.366×3.366 inches, there were about 300 dots per inch. Thus it took about 0.25 seconds to plot a line one inch long. Note that the algorithm can be improved by avoiding floating-point operations entirely:

```
DeltaX = X2 - X1´   All integer
   variables
DeltaY = Y2 - Y1
Dy2 = 2 x DeltaY
Ratio = Dy2 - 2 x DeltaX
Delta = Dy2 - DeltaX

j = Y1
For i = X1 to X2
   Call Dot (i, j)
   If Delta > 0 Then
      Delta = Delta + Ratio
      j = j + 1
   Else
      Delta = Delta + Dy2
   End If
Next i
```

This algorithm improves the loop's speed by about 15 percent.

Generalizing the algorithm to plot any line requires determining which X-value is the smaller, determining if the slope is ≤ 1 or > 1, and requires using X or Y, respectively, as the indexing value. A special case is if $X1 = X2$, but this is easy to deal with.

A square root algorithm

From the very beginning of computing, mathematical subroutines such as sine, cosine, logarithm, A^X, square root, and so on were available to programmers. The importance of optimizing their speed is obvious because most mathematical programs used them, and a single program could call them thousands of times.

In the early days of minicomputers, hardware costs were kept as low as possible so most minicomputers did not have floating-point hardware. Instead, floating-point operations were accomplished in subroutines. At the same time, functions like SQRT were done in software, so a square root function with floating-point software could be slow. The usual way to find a square root was to use a Newton's iteration:

$$G_{n+1} = G_n - \frac{G_n^2 - V}{2G_n}$$

where V is the value for which the square root is to be found, and $G1$ is the first approximation (or first guess) of the square root. The calculation is repeated until the difference between $G_n + 1$ and G_n is small enough, that is, until G_{n+1}^2 is approximately equal to V.

Given that V can range over the set of positive numbers, a good first approximation of G_1 is not easy to find, and of course the better the first guess, the faster the iteration converges. However, for values of V less than 1, convergence is fast.

Here's where knowledge of the structure of a floating-point number comes in handy. A floating-point number mimics scientific notation—for example, we write 12.35 as 0.1235×10^2. The power of 10, called the *characteristic*, is stored as a signed integer along with the fraction and its sign. This structure suggests that we can calculate the square root of the number by dividing the characteristic by 2 and finding the square root of the fraction.

Floating-point numbers are *normalized*, that is, the first digit to the right of the point is never zero (unless the value of the number is zero). In a binary floating-point number, this means that the first digit to the right of the binary point is always a 1, thus the value of the fraction is at least 0.5. So the square root of a binary floating-point number $0.11000 \ldots \times 2^3$ (in decimal, 6.0) can be found by halving the characteristic and finding the square root of 0.11000 …, except that the characteristic is an

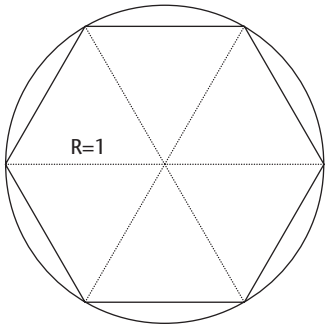


Figure 2. An inscribed hexagon decomposed into six equilateral triangles.

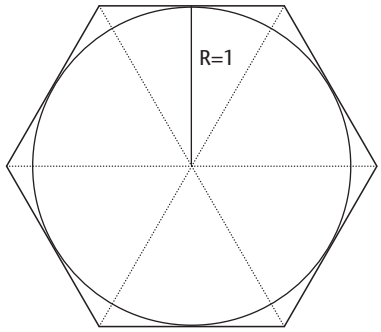


Figure 3. A circumscribed hexagon.

odd number. This is easily remedied by adding 1 to the characteristic if it is odd and shifting the fraction one place to the right, thereby dividing it by two, yielding $0.011000\ldots \times 2^4$, or 0.375×16—still 6.0. The square root of the characteristic is 2^2, and we can find the square root of the fraction in four Newton's iterations to an error of not more than 0.000002, using $G_1 = 0.7$. (It's necessary to set the characteristic to 2^0 so that the fraction V is between 0.25 and 1.) Furthermore, with a little effort, we can determine a single calculation that develops four iterations without a loop:

$$G_2 = G_1 - \frac{G_1^2 - V}{2G_1}$$

and

$$G_3 = G_2 - \frac{G_2^2 - V}{2G_2}$$

thus

$$G_3 = G_1 - \frac{G_1^2 - V}{2G_1} - \frac{\left(G_1 \frac{G_1^2 - V}{2G_1}\right)^2 - V}{2\left(G_1 - \frac{G_1^2 - V}{2G_1}\right)^2}$$

and so on.

Finally, the resulting square root G_4 must be multiplied by 10 to the power of the original characteristic divided by 2, to yield the square root of the original number V.

Machin's algorithm; computation of π

The pursuit of more and more digits of π may start with a reference in the Old Testament,[2] to zero decimal places; that is, π = 3. The Babylonians got by nicely with π = 3.125 (well, strictly speaking, 3 1/8, because decimal fractions were unknown to the Babylonians). The ancient Egyptians used $\pi = 4(8/9)^2$, which is about 3.1605, not really any better than the Babylonian estimate.

It was Archimedes (287–212 BCE) who started the contest of finding more and more digits of π by inscribing and circumscribing a regular polygon about a circle. This was a big step, because it provided both an upper and a lower bound for π, and the more sides to the polygon, the tighter the bounds became. The area of a circle is πr^2, so the area of a circle of unit radius is π.

For a hexagon, we see in Figure 2 that the inscribed polygon consists of six equilateral triangles with sides of length 1. Each has an area of the square root of 3 divided by 4, so the area of the inscribed hexagon is a bit more than 2.598.

For a circumscribed hexagon (see Figure 3) the height of each triangle is 1, so the area is slightly greater than 0.57735, giving the area of the circumscribed hexagon as about 3.4641.

Archimedes used these numbers to bound π. He wrote the equivalent of

$$2.598 < \pi < 3.464$$

which suggests that π is about 3.03. This isn't very good, except that Archimedes went on to use the same approach with 96-sided polygons and refined his bounds to

$$3.14084 < \pi < 3.142858$$

although he had to use proper fractions:

$$3\ 10/71 < \pi < 3\ 1/7$$

This is more like it—the average of the bounds is 3.14185; accurate to three decimals.

For almost 2,000 years, refinements of π as calculated in the Western world were done by increasing the number of sides of the polygons.

Finally, with the advent of the calculus in the 17th century came new ways to find π. Notable was John Machin, a London professor of astronomy, who in 1706 used the difference between two arctangents. It was this method that scientists used on ENIAC[3] in 1949 at the University of Pennsylvania, setting a new record of 2,037 digits:

$$\pi / 4 = 4 \tan^{-1}(1/5) - \tan^{-1}(1/239)$$

We know that π / 4 radians is 45 degrees, and that the angle whose tangent is 1 is 45 degrees. Unfortunately, we can't calculate $\tan^{-1}(1)$ using a series because the arctangent series only converges for values less than 1—hence Machin's equation.

It is easy to test the equation:

```
Debug.Print 4# * (4# * Atn(1# / 5#) -
Atn(1# / 239#))
```

gives the result 3.14159265358979, exactly right as far as it goes, but not a proof.

To show that

$$\tan^{-1}(1) = 4 \tan^{-1}(1/5) - \tan^{-1}(1/239)$$

we need two trigonometric identities:

$$\tan(2a) = (2 \tan(a)) / (1 - \tan^2(a)) \quad (1)$$

and

$$\tan(a + b) = (\tan(a) + \tan(b)) / (1 - \tan(a) \tan(b)) \quad (2)$$

If we let $x = \text{atn}(1/5)$, then using Equation 1 we get

$$\tan(2x) = (2 \tan x) / (1 - \tan^2(x)) = 5/12$$

and

$$\tan(4x) = (2 \tan(2x)) / (1 - \tan^2(2x)) = 120/119$$

but we want the result to be 1. We can find the difference from Equation 2:

$$\tan(4x - \pi / 4) = (\tan(4x) - 1) / (1 + \tan(4x)) = 1/239$$

hence tan(4 atn(1/5) – atn(1/239)) resolves to 1.

Machin found π / 4 to 100 places using two arctangent series:

$$\pi / 4 = 4 (1/5 - 1 / (3 * 5^3) + 1 / (5 * 5^5) - \ldots) - (1/239 - 1 / (3 * 239^3) + 1 / (5 * 239^5) - \ldots)$$

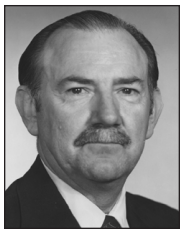
and this is how it was done on ENIAC to 2,037 places. Finding a very high precision result is a nontrivial programming exercise, but that's another matter.

Acknowledgments

I would like to acknowledge the assistance of Michael Greenwald, Dick Painter, and Dennis Quardt in the preparation of this article.

References and notes

1. W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York, 1979. Jack Bresenham is currently a professor of computer science at Winthrop University. He retired from IBM as a senior technical staff member in 1987 after 27 years. He is best known for fast line and circle algorithms.
2. 1 Kings 23, Authorized (King James) Version. The verse reads: "And he made a molten sea, ten cubits from the one brim to the other: it was round all about, and his height was five cubits: and a line of thirty cubits did compass it round about." The "molten sea" is a brass vat. Its diameter is ten cubits, thus its radius is five cubits. The given circumference—the line of 30 cubits that "did compass it round about"—implies that π = 3, because the circumference is computed as 2πr, from 2π * 5 = 30.
3. P. Beckmann, *A History of Pi*, Dorset Press, New York, 1989.



Keith S. Reid-Green retired in 1999 after 42 years as a computer scientist. He began as an IBM 704 programmer in 1957 and is known for pioneering work in data reduction, computer-aided manufacturing, and image processing. He is widely published in the history of computing and was an adjunct professor at the University of Utah and at Rutgers, where he taught computer graphics to graduate students. He holds bachelor's degrees in mathematics and physics and a master's degree in computer science.

Readers may contact Keith Reid-Green at 23 Point Court, Lawrenceville, NJ 08648, email KReid-Green@msn.com.