Features

Floating-point Arithmetic in Applesoft BASIC

James W. Thomas

Floating-point numbers are represented in memory using five bytes (40 bits). The first byte is an 8-bit exponent which includes a bias of 128. The high order bit of the second byte is the sign bit. The remaining 31 bits are used with an implicit high-order 1 bit to make a 32-bit significand. Representation is signed magnitude. Thus:

\[ \text{exp} \quad \text{s} \quad \text{significand bits} \]

represents \((-1)^s \times 2^{\text{exp} - 128} \times 1.\text{significand bits}\). Zero is represented whenever exp is 0, regardless of the significand bits. Valid representations range from \(2^{128}\) (actually \(2^{128} - 1\) is invalid in some cases) to \(2^{127} \times (1 - 2^{-23})\), or roughly \(10^{-38}\) to \(10^{38}\). The 32 bit significands can distinguish between nine decimal digit significands.

While arithmetic operations are carried out, operands may have an additional guard byte, giving a significand of 40 bits. In addition and subtraction, both operands and the result use guard bytes. In multiplication, the multiplier and the product use guard bytes. In division, only the quotient has extra significance, but just 1 bit. When addition, subtraction, or multiplication is initiated, one operand has a 32 bit significand and the other a 40 bit significand. Although both operands in addition and subtraction use guard bytes, the operand with the 32 bit significand does so only if it has the smaller exponent, shifting its significant right. In formula evaluations involving several operations, the sizes of the significands depend on the order of the operations as well as on the operations themselves. Intermediate operands with 40 bit significands may be rounded to 32 bits and pushed onto a stack, or not, depending on the details of the formula.

Rounding consists of inspecting the high order bit of the guard byte and, if a 1 is found, incrementing the higher order 32 bit significand. Thus rounding is to the nearest value with ties going away from zero. Numbers are normalized before rounding, except when an addition or subtraction cancels the high order 32 bits but leaves a non-zero guard byte. Then the result is set to zero. In addition to stack pushes by formula evaluation, memory stores also are preceded by rounding. Divisors are rounded before divisions occur. The guard byte has no sticky bit, so that any 1’s shifted out the right of the guard byte are lost.

Overflow results in an OVERFLOW ERROR message and termination of the program. Underflows are set to zero.

Irregularities

By irregularities we mean behavior of the arithmetic which the user should not be expected to anticipate. Such behavior may occur intentionally, because of design decisions, or unintentionally, because of bugs. Actually the distinction here is not clear—cut because design decisions often are made to live with presumably insignificant or tolerable bugs.
Features

Formula-evaluation design flaw. Intermediate operations may present different results to subsequent operations, depending on the positions of the operations in the formula.

Example (non-commutative addition):

```
JLIST
10 A = 2 ! ( - 2 ) + 2 ! ( - 3 ) + 2 ! ( - 33 )
20 A = - A
30 B = 2 ! ( - 1 ) + 2 ! ( - 32 )
40 C = 2 ! ( - 1 ) + 2 ! ( - 2 )
50 PRINT A + B * C * B * C + A
```

The remedy here would involve nontrivial design decisions. Two obvious approaches are i) including the guard byte on the stack and ii) rounding operands before operations occur. Alternative i) requires modification of the addition, subtraction, and multiplication routines which now use operands of different size significands.

Evaluator-comparator conspiracy. In comparing two numbers for =, <, or >, the propagation of carries from a guard byte beyond the low order byte is not considered. Thus A op B = A op B may be false, since the formula evaluator rounds and pushes one A op B result onto the stack but leaves the other A op B result unrounded in the floating-point accumulator.

Example (non-reflexive equality):

```
JLIST
10 A = 2 ! ( - 1 )
20 B = 2 ! ( - 24 ) - 2 ! ( - 33 )
30 IF A + B = A + B THEN PRINT "A + B = A + B"
40 IF A + B > A + B THEN PRINT "A + B > A + B"
```

Either remedy suggested above in the "Formula-evaluation design flaw" section would correct this flaw. A fix not involving the formula evaluator must compare a rounded number with a non-rounded number as if the latter were rounded. This appears to require some code.

Sign-of-small-quotient bug. If the exponent of a quotient is -128, then a positive quotient will result regardless of the signs of the divisor and dividend.

Example (A/2A = -5):

```
JLIST
10 A = -(2 ! ( - 127 ))
20 PRINT A * A / A
```

In division, the exponent of the divisor is negated and added to the exponent of the dividend. If the result including its bias is zero, then control branches to an underflow routine in order to produce a zero quotient. The zero exponent already indicates zero, so the underflow routine simply sets the sign to "+". But then the return is back into the division routine which increments the exponent to account for a shift and carries out the division, so the result is no longer zero. It appears the underflow should not have been signaled, since the result is good except for the sign.

Multiplier bug. When two consecutive bytes of the multiplier are 0, the accumulated product bits to that point will be shifted one place too far to the right.

Example (1 * A + A):

```
JCALL -151
1A = 2 ! ( - 1 ) + 2 ! ( - 24 ) - 2 ! ( - 31 )
JB = 1A
```

Note that the low order byte FE of A becomes 7F in B = 1A, a shift of one bit to the right. The error can be as large as 127 units in the last place of the binary significand. With A as in the example A yields A, so the multiplication bug produces non-commutative multiplications. With a slight modification, the example violates A > 1 -> AIB > BIB > 0. This bug could be easily exterminated by resetting a carry which is cleared by a special routine which handles the zero byte multiplication.

Attempting a clearer demonstration of the effect of the multiplication bug, we continue with the preceding example.

```
JPRINT A : B = REM B = 1A
.500000003 .500000015
```

But hand conversion of the binary representations shows that A and B differ by approximately 30 in the ninth decimal place. The effect of the multiplication bug is confounded by the following.

Binary → decimal conversion error

Example:

```
JCALL -151
1A = .500000009
```

```
0803- 41 00 80 00 00
0808- 00 FE
```

```
JPRINT A
.500000029
```

Call - A.P.P.L.E. July 1985
Features

The internal binary representation of A is correct to all 32 binary places, so the decimal-to-binary conversion has done its job well. Since 2^{-32} < 2.4 \times 10^{-10} the binary number in the example is nearer to 5000000009 than to 5000000008 or 5000000060. Hence we might expect the conversion decimal-to-binary-to-decimal to be the identity on 5000000009. However we obtain an error of 30 in the ninth decimal place! The conversion routines use the floating-point arithmetic and so are subject to the errors therein. Perhaps the multiplication bug, via multiplications in the binary-to-decimal conversion, causes the surprisingly large error above.

Trigonometric function error. The sine function exhibits extremely poor accuracy near zero.

Example (\sin(A)A is not near 1 when A is small):

\[
\begin{align*}
\text{JLIST} & \\
10 & \text{FOR } I = 1 \text{ TO } 12 \\
20 & A = 10 \times ( -1 ) \\
30 & \text{PRINT A, } \sin(A) / A \\
40 & \text{NEXT}
\end{align*}
\]

\[
\begin{align*}
\text{RUN} & \\
.1 & \text{.998334166} \\
.01 & \text{.99999999} \\
1E-03 & \text{.999999999} \\
1E-04 & \text{.99999999} \\
1E-05 & \text{.99999997} \\
1E-06 & \text{.99998451} \\
1E-07 & \text{.99975593} \\
1E-08 & \text{.997184394} \\
9.99999998E-11 & \text{0} \\
9.99999997E-12 & \text{0} \\
9.99999996E-13 & \text{0}
\end{align*}
\]

The sine function is identically zero for arguments greater than 5 \times 10^{10}.

Example (\sin(A) = 0 for large A):

\[
\begin{align*}
\text{JLIST} & \\
10 & \text{FOR } I = 5 \text{ TO } 12 \\
20 & A = 10 \times I \\
30 & \text{PRINT A, } \sin(A) \\
40 & \text{NEXT}
\end{align*}
\]

\[
\begin{align*}
\text{JRUN} & \\
100000 & \text{.356574928} \\
1000000 & \text{-349137508} \\
10000000 & \text{.41642956} \\
100000000 & \text{.91420756} \\
1E+09 & \text{.707106781} \\
1E+10 & \text{0} \\
1E+11 & \text{0} \\
1E+12 & \text{0}
\end{align*}
\]

The flaw in the sine routine lies in its argument reduction. The argument is divided by 2^{\pi} and F, the fractional part of the quotient, is obtained. As the first step of determining the quadrant, F is subtracted from 1/4 and the difference is the basis of all further calculations. If F is small, 1/4 - F has little or no significance. If the original argument is large, the fractional part of its quotient by 2^{\pi} is zero.

The cosine and tangent functions are obtained through the identities

\[
\begin{align*}
\cos(x) = \sin(x + pi/2) \\
\tan(x) = \sin(x)/\cos(x)
\end{align*}
\]

and hence inherit the errors in the sine functions.

Example (\tan(A)A is 1):

\[
\begin{align*}
\text{JLIST} & \\
10 & \text{FOR } I = 1 \text{ TO } 12 \\
20 & A = 10 \times (-1) \\
30 & \text{PRINT A, TAN(A) / A} \\
40 & \text{NEXT}
\end{align*}
\]

\[
\begin{align*}
\text{RUN} & \\
.1 & \text{.1000334672} \\
.01 & \text{.100000334} \\
1E-03 & \text{.100000033} \\
1E-04 & \text{1} \\
1E-05 & \text{.999999941} \\
1E-06 & \text{.999999798} \\
1E-07 & \text{.999984543} \\
1E-08 & \text{.999755933} \\
1E-09 & \text{.997184394} \\
9.99999998E-11 & \text{0} \\
9.99999997E-12 & \text{0} \\
9.99999996E-13 & \text{0}
\end{align*}
\]

Trigonometric identities are not reliable.

---

**BACKUP PROTECTED SOFTWARE with COPY II PLUS™ ver. 5**

From the team who first brought you COPY II PLUS in 1981 comes a completely updated disk backup utility for your Apple // computer. New features include:

- **Fully automatic bit copy**. All parameters are stored on disk. Simply type in the name of the program you wish to backup, and COPY II PLUS does the rest!
- **New utilities** including Alphabetize Catalog, Fast 2-pass Disk Copy on a //c or //e, and an all-new Sector Editor.
- **Supplied on a standard DOS diskette**. Runs on the Apple //, Apple //+, Apple //e, Apple //c. Requires 64K and one or two disk drives.

Increase the power of your Apple //…

Use COPY II PLUS™ 5.0

Call M-F 8-5:30 (W. Coast time) with your 1-3-2-1-2-3-1-2-3-4-5-6-7-8-9-10-1. Or send a check (add $3 s/h, $8 overseas) to

**$39.95**

CENTRAL POINT SOFTWARE, INC.
9700 SW Capitol Hwy. *100
Portland, OR 97219

*We update Copy II Plus regularly to handle new protections; you as a registered owner may update at any time for $1/2 price! (To update, just send original disk and $20.)*

This product is provided for the purpose of enabling you to make archival copies only.

---

Call —A.P.P.L.E. July 1985
Even if the sine’s argument reduction were repaired, the cosine would perform poorly for large arguments, \( x \), because \( \pi/2 \) would have little or no effect in \( x + \pi/2 \). The simple repair for the trigonometric functions is to replace the module with an implementation of good existing algorithms.

**Implications**

The examples in the preceding section point to several areas in which the arithmetic’s behavior is other than a programmer would expect. This behavior is caused by subtle workings of the arithmetic routines (intentional or unintentional), the understanding of which would prove a heavy burden for the non-expert programmer.

- The programmer may have less accuracy than the 32 bit significance with rounding to nearest plus guard bytes for intermediate results would lead him to believe. The binary-to-decimal conversions and the multiplication bug may produce single operation errors of 30 in the ninth decimal place. The trigonometric functions may produce results with no significance. The guard byte provides extra accuracy only for rather special formulas which avoid the rounding stack pushes. The guard byte does not facilitate the significance one would hope for when addition or subtraction cancels the 32 high order bits, because such results are set to zero.

- The most innocent algebraic manipulations of formulas may alter a program’s results. Virtually every axiom of real numbers can fail (maybe not \( 0 + x = x \)). The compilation of a usable set of rules on which a programmer could depend would be very difficult.

- Branch tests may be difficult to control. The sign-of–small–quotients bug may reverse a sign. The formula evaluator may produce a zero for an expression which elsewhere appears to be non-zero. The comparator-evaluator conspiracy may provide incorrect comparisons. The loss of accuracy mentioned above may affect comparisons. Thus the unwary programmer’s tests to separate cases or guard against invalid operands may be sabotaged.

Although the irregularities discussed in this paper appear simple to correct (with the possible exception of the formula evaluation flaw), Apple Computer Inc., will be reluctant to implement changes. The Applesoft BASIC was purchased from Microsoft and no one with Apple is truly familiar with the details of the code. The code has little modularity, so it is difficult to assess the ramifications of changes. Multitudinous software exists for the current version and it would be difficult to check that these programs still run.

On the other hand, with the increased interest in arithmetic among the experts and with the increased knowledge of computing among the general population, the number of users who are aware of anomalies in the arithmetic can be expected to increase. More users will pursue the possibility that errors in their results are caused by computer arithmetic.

**References**