

Features

Floating-point Arithmetic in Applesoft BASIC

The Apple Numerics Group has been working for several years to implement state-of-the-art numerics on all Apple computers. The result of these efforts is called SANE, for Standard Apple Numerics Environment. It is available for Pascal and assembly programmers on Apple II and III computers, and is the native arithmetic on Macintosh. AppleWorks, MacPascal, MacBASIC, the Lisa Workshop, and several other Macintosh languages and application programs use SANE.

This article, which identifies several of the problems in Applesoft arithmetic, was written 3 years ago by Jim Thomas while a student at the University of California, Berkeley. Jim is now the head of the Apple Numerics Group, which includes Kenton Hanson and Clayton Lewis. We are indebted to Clayton for bringing the article to our attention.

James W. Thomas

Many people are no longer surprised to find their computers executing:

```
S = 0
FOR I = 1 TO 1000
S = S + 0.1
NEXT I
PRINT S
```

and not printing 100. They realize that computer arithmetic is in some ways different from the arithmetic they learned in mathematics classes. The typical computer user will attempt to ignore these discrepancies, proceed as if dealing with ordinary (real) numbers, and then acknowledge that his results may contain some "insignificant" roundoff error. The user may know that for some computations, a computer's arithmetic fails to produce usable results. He may believe this happens only for contrived problems, or only for people with unusually bad luck. If his program fails to obtain results, or if he recognizes his results contain significant errors, he still may not suspect computer arithmetic as the culprit.

The expert, the numerical analyst or the scientific programmer, is less naive. In his circles there is active concern about the manner in which computers do arithmetic, evidenced by the 1981 IEEE Proposed Standard for Binary Floating-Point arithmetic (which became IEEE Standard 754 in April 1985 and is the basis for SANE-Ed.) Partly because of the aforementioned attitudes among naive users, computer designers have exercised great liberty with the nature and quality of their machines' arithmetics. The resulting peculiarities are many and varied (7). The more irregular the arithmetic, the more difficult becomes the programmer's job in programming around trouble areas and in verifying program correctness. If portability of program is desired, the difficulty increases accordingly. (In light of growing software costs, these concerns are more than aesthetic.)

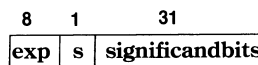
Of course arithmetic irregularities affect the non-expert's programs too. Since he

may not undertake an extensive proof of correctness, he may be spared the task of programming around trouble areas spotted in the analysis. Then, however, he may have results whose accuracy is eroded by the arithmetic to an extent completely unknown. On the other hand, if the arithmetic produces errors which do attract his attention, either in output or in other unintended program behavior, and if he suspects the computer arithmetic as the culprit, then the non-expert may still have enormous difficulty in debugging without acquiring a detailed knowledge of his machine's particular arithmetic.

In this article, we discuss some of the peculiarities of Apple II Applesoft BASIC, a floating-point arithmetic which is widely used by non-experts. In the section entitled Architecture, we summarize the architecture of the arithmetic; under Irregularities we present and discuss several behavioral irregularities; and under Implications we assess the implications the architecture and irregularities hold for the users.

Architecture

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



represents $(-1)^s \times 2^{\text{exp}-128} \times .1\text{significandbits}$. Zero is represented whenever exp is 0, regardless of the significand bits. Valid representations range from 2^{-128} (actually 2^{-128} is invalid in some cases—then 2^{-127}) to $2^{127} \times (1 - 2^{-32})$, 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 significand 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
```

JRUN

0 1.16415322E-10

This phenomenon occurs as a direct consequence of the design of the formula evaluation routine. For $A + B * C$ the result of $B * C$ stays in a floating-point accumulator, unrounded and with guard byte. For $B * C + A$ the result of $B * C$ is rounded and stacked. Hence different numbers are added to A in the two cases. The following is similarly explained.

Example (non-commutative multiplication):

JLIST

```
10 A = 1 - 2↑(-31)
20 B = 2↑(-33)
30 C = 2↑(-1)
```

```
40 PRINT C + (A + B) * C - A
50 PRINT C + C * (A + B) - A
```

JRUN

4.65661287E-10
2.32830644E-10

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 with 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 $A \text{ op } B = A \text{ op } B$ 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"
```

JRUN

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 $((A/2)/A = -.5)$:

JLIST

```
10 A = -(2↑(-127))
20 PRINT A, (A / 2) / A
```

JRUN

-5.87747176E-39 -.5

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 $(1 * A \neq A)$:

JA = 2↑(-1) + 2↑(-24) - 2↑(-31)

JB = 1 * A

JCALL -151

*803.809

0803- 41 00 80 00 00
0808- 00 FE

*80A.810

080A- 42 00 80 00 00 00
0810- 7F

Note that the low order byte FE of A becomes 7F in $B = 1 * A$, 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 $A * 1$ yields A, so the multiplication bug produces non-commutative multiplications. With a slight modification, the example violates $A > 1 \rightarrow A * B > B (B > 0)$. 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 = 1 * A
.50000003 .50000015
```

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:

JA = .500000059

JCALL -151

*803.809

0803- 41 00 80 00 00
0808- 00 FD

```
JPRINT A
.500000029
```

ONE
OF THESE
FEATURES COULD
COST YOU MORE
THAN MY
ENTIRE
SYSTEM

10 day money
back guarantee

BLANKENSHIP BASIC
For the Apple II+, IIe, and IIc

1. Real interpreter, not a pre-processor
2. WHILE-ENDWHILE and REPEAT-UNTIL loops
3. True IF-THEN-ELSE-ENDIF (Using WHEN)
4. PRINT, USING, FILE, MERGE, RANDOMIZE
5. PRINT and TAB commands work in HIRIS
6. 80 columns supported on IIe and IIc
7. Full Editor with AUTO-NUM and RENUM
8. Listings are indented automatically
9. Fast SORT, SEARCH and INSTR\$ commands
10. BOX, BOXFILL, DRAW.USING and SOUND
11. DISK command replaces DOS's CHR\$(4)
12. DEFINE and PERFORM NAMED procedures
13. 99% Upward compatible with Applesoft
14. All commands entered normally, no &'s
15. 100's of satisfied users world wide
16. FREE newsletter available to owners

Apple is a registered trademark of Apple Computer Inc.

mail check to:	DOS 3.3 version	\$25 ⁰⁰
John Blankenship	ProDOS version	\$25 ⁰⁰
P.O. Box #7934	Both versions	\$39 ⁹⁵
Atlanta GA 30382	Add \$1.50 postage & handling	

Circle 57 on Reader Service Card

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 .500000059 than to .500000058 or .500000060. Hence we might expect the conversion decimal-to-binary-to-decimal to be the identity on .500000059. 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):

```
JLIST
10 FOR I = 1 TO 12
20 A = 10 ↑ (- I)
30 PRINT A, SIN (A) / A
40 NEXT

RUN
.1 .998334166
.01 .999983334
1E-03 .99999833
1E-04 .99999995
1E-05 .99999994
1E-06 .99999879
1E-07 .999984511
1E-08 .99975593
1E-09 .997184394
9.9999998E-11 0
9.9999999E-12 0
9.9999999E-13 0
```

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

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

```
JLIST
10 FOR I = 5 TO 12
20 A = 10 ↑ I
30 PRINT A, SIN (A)
40 NEXT

JRUN
100000 .0356574928
1000000 -.349137508
10000000 .41642956
100000000 .914209756
1E + 09 .707106781
1E + 10 0
1E + 11 0
1E + 12 0
```

The flaw in the sine routine lies in its argument reduction. The argument is divided by $2 \times \pi$ 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 $2 \times \pi$ is zero.

The cosine and tangent functions are obtained through the identities

$$\cos(x) = \sin(x + \pi/2)$$

$$\tan(x) = \sin(x)/\cos(x)$$

and hence inherit the errors in the sine functions.

Example ($\tan(A)/A < 1$):

```
JLIST
10 FOR I = 1 TO 12
20 A = 10 ↑ (- I)
30 PRINT A, TAN (A) / A
40 NEXT
```

```
JRUN
.1 1.00334672
.01 1.00003334
1E-03 1.00000033
1E-04 1
1E-05 .99999994
1E-06 .999998798
1E-07 .999984543
1E-08 .999755933
1E-09 .997184394
9.9999998E-11 0
9.9999999E-12 0
9.9999999E-13 0
```

Trigonometric identities are not reliable.

Example ($\sin(A)^2 + \cos(A)^2 \neq 1$):

```
JLIST
10 FOR I = 5 TO 12
20 A = 10 ↑ I
30 PRINT A, SIN (A) ↑ 2 + COS (A)
  ↑ 2
40 NEXT

RUN
100000 1
1000000 1
10000000 1
100000000 .982226087
1E + 09 1.35355339
1E + 10 0
1E + 11 0
1E + 12 0
```

Example ($\sin(2 \times A) \neq 2 \times \sin(A) \times \cos(A)$):

```
JLIST
10 FOR I = 5 TO 12
20 A = 10 ↑ I
30 PRINT SIN ( 2 * A ), 2 * SIN ( A ) *
  COS ( A )
40 NEXT



JRUN
-.0712696343 -.0712696343
-.654333618 -.654333618
-.757208846 -.757208846
-.740951125 -.699705854
1 1.30656297
0 0
0 0
0 0
```

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   : 503/244-5782.
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.

Circle 58 on Reader Service Card

Features

COMPUTEREYES™

VIDEO IMAGES ON YOUR APPLE!

Finally — an inexpensive way to capture real-world images on your Apple's HiRes display! COMPUTEREYES™ is an innovative slow-scan device that connects between any standard video source (video tape recorder, video camera, videodisk, etc.) and the Apple's game I/O socket. Under simple software control, a b/w image is acquired in less than five seconds. A unique multi-scan mode also provides realistic grey-scale images. Hundreds of applications!

Package includes interface module, cable, complete on-line software support on disk, owner's manual, and one year warranty. For 48K Apple II series and compatibles, with Applesoft and DOS 3.3. COMPUTEREYES™ is available from your dealer or direct from DIGITAL VISION for just \$129.95 plus \$4.00 S&H (U.S.A.).

Also available as a complete package including:

- COMPUTEREYES™
- Quality b/w video camera
- Connecting cable

for only \$349.95 plus \$9.00 S&H.

Demo disk available for only \$10.00 postpaid.

Mass. residents add 5% sales tax. Mastercard, Visa accepted. Orders or for more information, write or call.

Screen dumps of actual COMPUTEREYES™ images.

Now also for Apple II!

DIGITAL VISION

DIGITAL VISION, INC.
14 Oak Street — Suite 2
Needham, MA 02192
(617) 444-9040

Circle 62 on Reader Service Card

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.

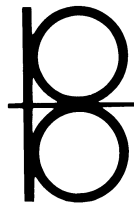
References

1. *Apple II Reference Manual*. Apple Computer Inc., 1979.
2. *Applesoft BASIC Programming Reference Manual*, Apple Computer Inc., 1978.
3. IEEE Subcommittee p754, "A Proposed Standard for Binary Floating-Point Arithmetic," Draft 8.0, *Computer*, 14, 3, March 1981, pp 53-62.
4. Camp, Smay, Triska, *Microprocessor Systems Engineering*, Matrix Publishers Inc., 1979.
5. Coonen, J.T., "Accurate, Economical Binary - Decimal Conversions," May 1981.
6. Kahan, W., "Error in Numerical Computation," Course Notes, 1963.
7. Kahan, W., "Why Do We Need A Floating-Point Arithmetic Standard?," March 1981.
8. Luebbart, W.F., "What's Where in the Apple," *MICRO - the 6502 Journal*, August 1979.

Life in the Fast Lane, Slow Lane or Any Lane...

...is expensive when it comes to your car! A typical driver can spend \$1000 or more on fuel per year driving 15-20,000 miles at 20 miles per gallon. Driving with a dirty air filter, low tire pressure and other problems can add \$100-200 to fuel costs, plus higher repair bills and decreased reliability!! Minimizing those costs means keeping your car in top condition and spotting problems early. Using CAR-TRAK, your APPLE II can eliminate the guesswork from preventative maintenance and help avoid unnecessary repairs and fuel costs. CAR-TRAK is two programs in one! A SERVICE SCHEDULER reminds you of inspection and service requirements by date and mileage for up to 50 items per car. A PERFORMANCE ANALYZER graphically highlights suspicious changes in fuel consumption which may signal early trouble, and provides a number of SUMMARY REPORTS to help with business expenses, income taxes, and billing verification. It's simple and easy to learn and use! Put your APPLE II to work for you. Order CAR-TRAK by CHECK or M.O. for \$39.95 plus \$2.00 shipping and handling. (CA residents add \$2.40 sales tax.)

CAR-TRAK requires II+ //c or //e w/48K.
 APPLE is a trademark of APPLE Computer, Inc.



10/10 SOFTWARE
 22996 El Toro Rd.
 Suite 138
 El Toro, CA 92630
 (714) 380-1063

Circle 76 on Reader Service Card