Everything you never wanted to know about IBM and IEEE floating point numbers

Kevin W. Hall

ABSTRACT

The CREWES Matlab® toolbox SEG-Y I/O functions have long been able to read IBM floating-point, but have not been able to differentiate between IBM and IEEE trace data. This was left to the user. In addition, it was not possible to write IBM floating point SEG-Y files using the CREWES tools. New functions have been written, tested, and are now available in the toolbox for writing IBM floats. We include new recommendations for handling both large and small numbers in IEEE format

INTRODUCTION

Other than integer formats, the SEG-Y revision 0 standard only allows seismic data to be stored in big-endian four-byte IBM floating-point format (IBM) with the data sample format code (format code) in the SEG-Y binary file header set to 1. Big-endian is defined to mean that the most-significant byte is closest to the beginning of the tape or disk file (Barry et al, 1975). In this report binary and hexadecimal numbers are shown as big-endian, with the most significant byte to the left of the page or table. Note that all modern computers that use Intel or AMD CPU chips are little-endian.

With the advent of computer hardware that used IEEE floating-point format (IEEE), vendors began to write non-standard SEG-Y revision 0 files with four-byte IEEE seismic trace data. The binary file header format code was still set to 1, meaning the trace data was stored as floating point, but there was no way to tell from the binary file header whether the trace data are IBM or IEEE. This ambiguity leads to the possibility of reading SEG-Y trace data incorrectly, if software implicitly trusts the format code. Figures 1 and Figure 2 show the results of reading correlated Vibroseis data using the correct and incorrect floating-point formats. While it is clear from the amplitude spectra which output trace is correct in this case, the differences can be more visually subtle for uncorrelated data (not shown) and dynamite data (Figures 3 and 4).

SEG-Y revision 1 explicitly allows the use of IEEE with the format code set to 5, but continues to require that the data be big-endian (Norris and Faichney, 2002). SEG-Y revision 2 allows little-endian byte order as well as eight-byte IEEE floating-point with the format code set to 6 (Hagelund and Stewart, 2017). Using IEEE and setting the format code to 5 or 6 removes the ambiguity present in SEG-Y revision 0 files.

Some non-standard revision 0 and 1 files are little-endian (sometimes called PC byte-order), which can also lead to reading the seismic data incorrectly. In this case, the binary file header and trace header values are also read incorrectly, so this issue is easier to detect than the floating-point format issue for the trace data.

Legacy SEG-Y data is most often in IBM format, so software that reads these files must be able to convert from IBM to IEEE before the data can be processed or interpreted on most modern computers. In addition, many people still prefer to write SEG-Y files using IBM, possibly for historical reasons, and possibly because older software may not support IEEE.

The CREWES Matlab® toolbox SEG-Y reading functions have been able to read IBM for many years, but were never been able to write IBM correctly. Three new functions have been introduced to the toolbox this year that enable reading and writing IBM floating-point numbers, *num2ibm()*, *ibm2num()* and *log16()*. These functions are described in greater detail below.

FLOATING POINT FORMATS

The equation that is used to decompose decimal numbers to store them in a floating-point format is

$$(-1)^{sign} * fraction * \text{base}^{exponent+bias}, \qquad (1)$$

where the *sign, fraction* and *exponent+bias* are stored as *N*-bit unsigned integers, typically in either four-bytes (single-precision) or eight-bytes (double-precision). Table 1 summarizes the differences between IBM 360 (IBM, 1967) and IEEE 754 (IEEE, 2008), floating point formats. Appendix A shows examples of bits, nibbles, bytes, bitshifts and binary, hexadecimal and decimal numbers.

The *fraction* (Equation 1) turns out to be binary identical for IBM and IEEE formats, but is normalized differently. The base 2 IEEE *fraction* is normalized by bitshifting one bit to the left until bit24 is a 1, which is then not stored (implicit one). Each bit-shift requires an update to the *exponent*. The IEEE *fraction* is always stored with 24-bit precision, but only uses 23-bits to store the *fraction* because of the implicit one.

The base 16 (hexadecimal) IBM *fraction* is bitshifted one nibble (four bits) to the left until the most significant nibble (bits 20-24) is non-zero. Since this means the IBM *fraction* can have up to three leading zero bits, the *fraction* winds up being stored with anything from 21 to 24-bit precision, even though it is stored in 24-bits. This precision 'wobble' can cause computation issues (eg. Harding, 1966, Tomayko, 1995), and can also affect the accuracy with which seismic data values are stored on disk or tape (see below).

In practice, the above means that we could bitshift the *fraction* and update the *exponent+bias* to convert between the 4-byte formats. However, this approach leads to errors due to truncating rather than rounding the *fraction*, as well as overflows and underflows that require additional code to handle. The author's preference is to mathematically convert a decimal number to a *fraction* and *exponent* and then encode those results.

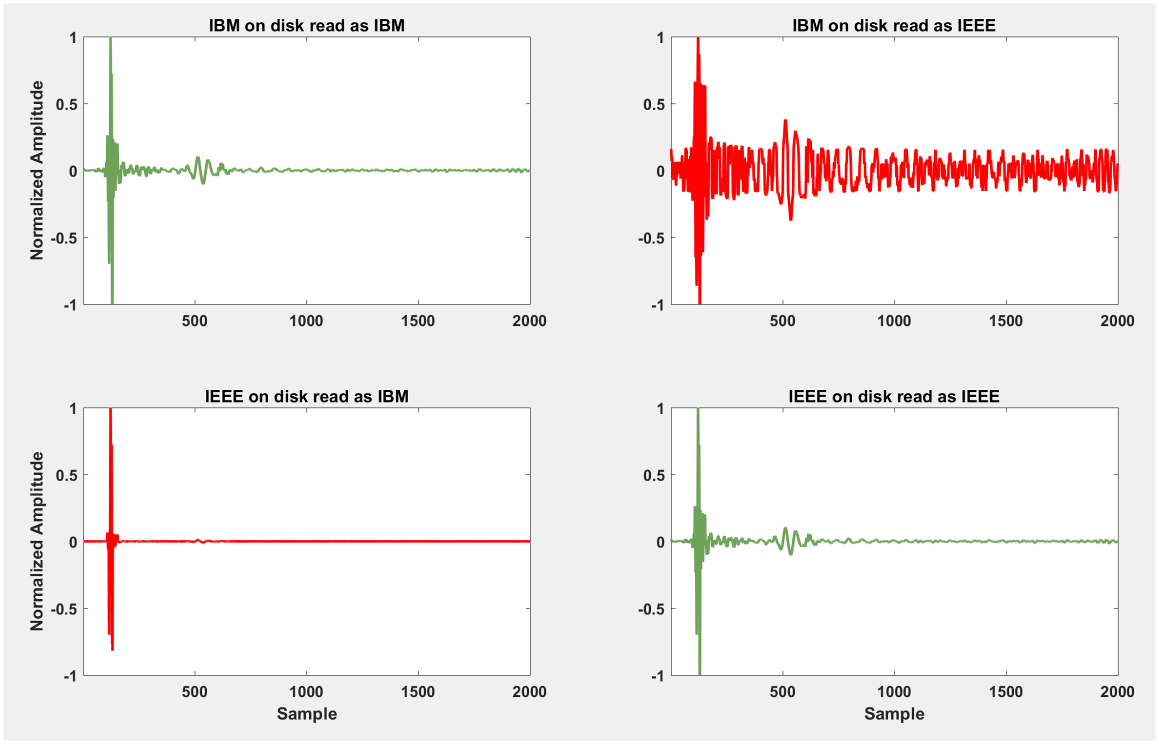

FIG. 1. Correlated Vibroseis data acquired with a 10-250 Hz sweep stored as IBM (top) and IEEE (bottom) floating point, that has been read into memory as IBM (left) and IEEE (right). Trace amplitudes have been normalized for comparison, but no other processing has been applied. Correct answers are shown in green.

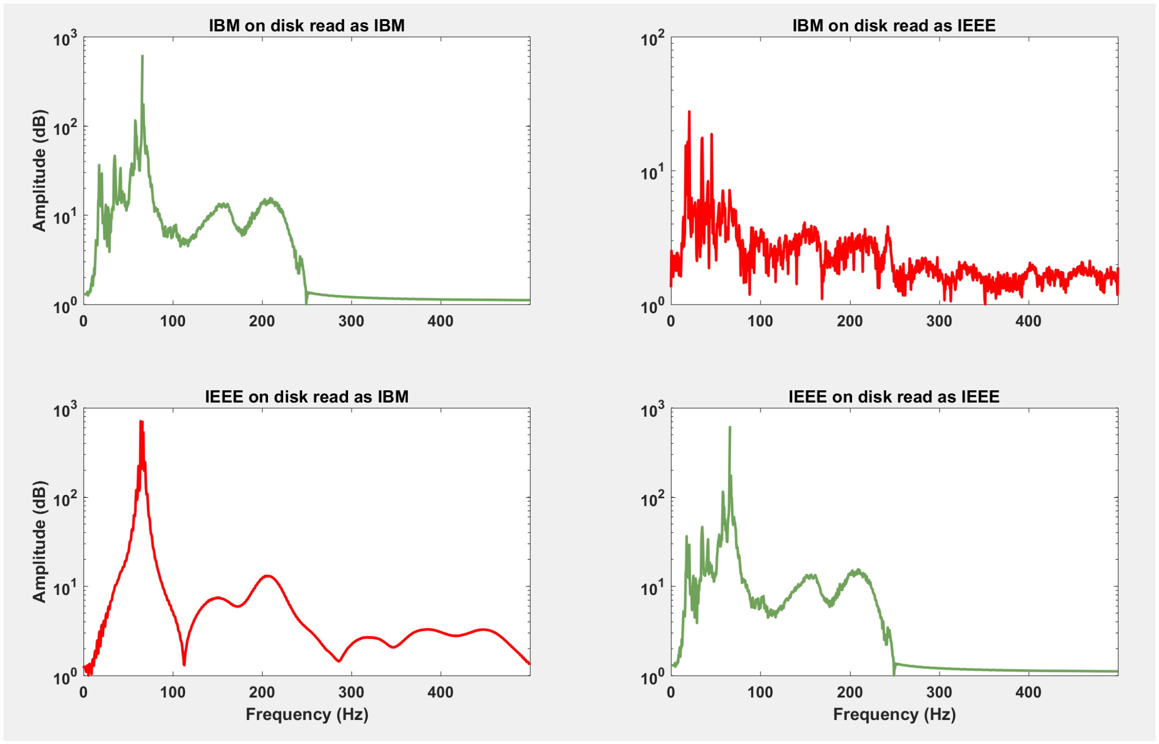

FIG. 2. Amplitude spectra corresponding to the traces shown in FIG. 1.

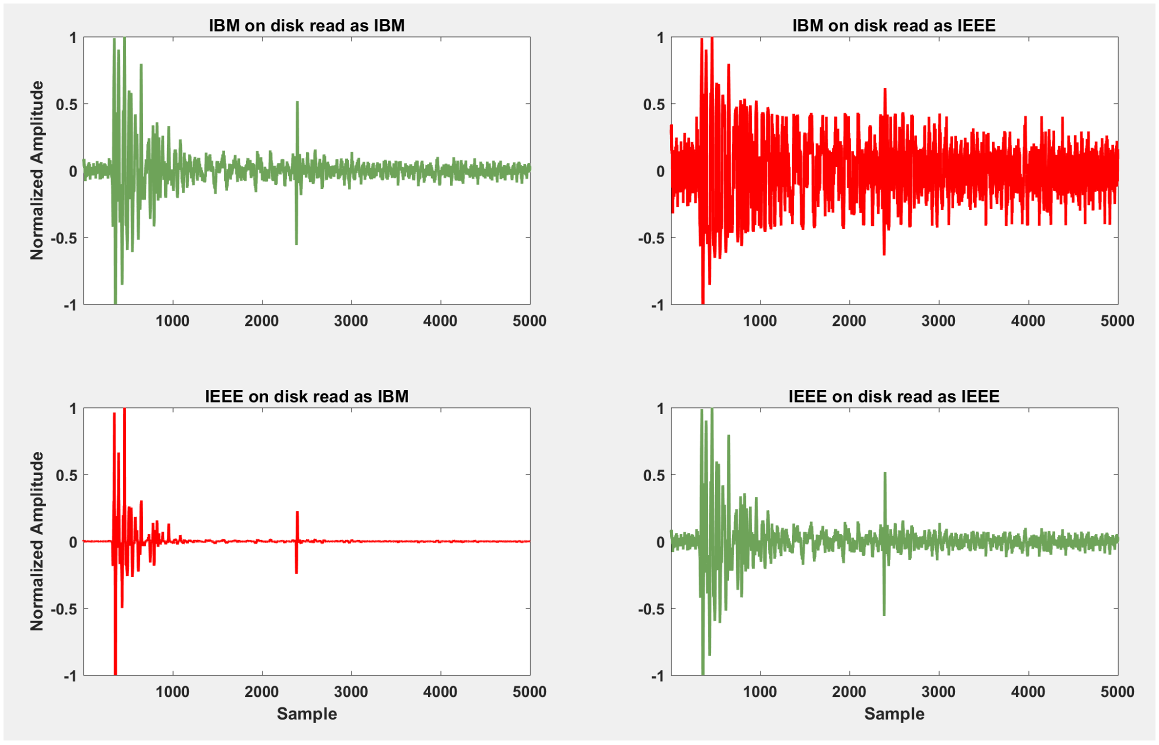

FIG. 3. Dynamite data stored as IBM (top) and IEEE (bottom) floating point, that has been read into memory as IBM (left) and IEEE (right). Trace amplitudes have been normalized for comparison, but no other processing has been applied. Correct answers are shown in green.

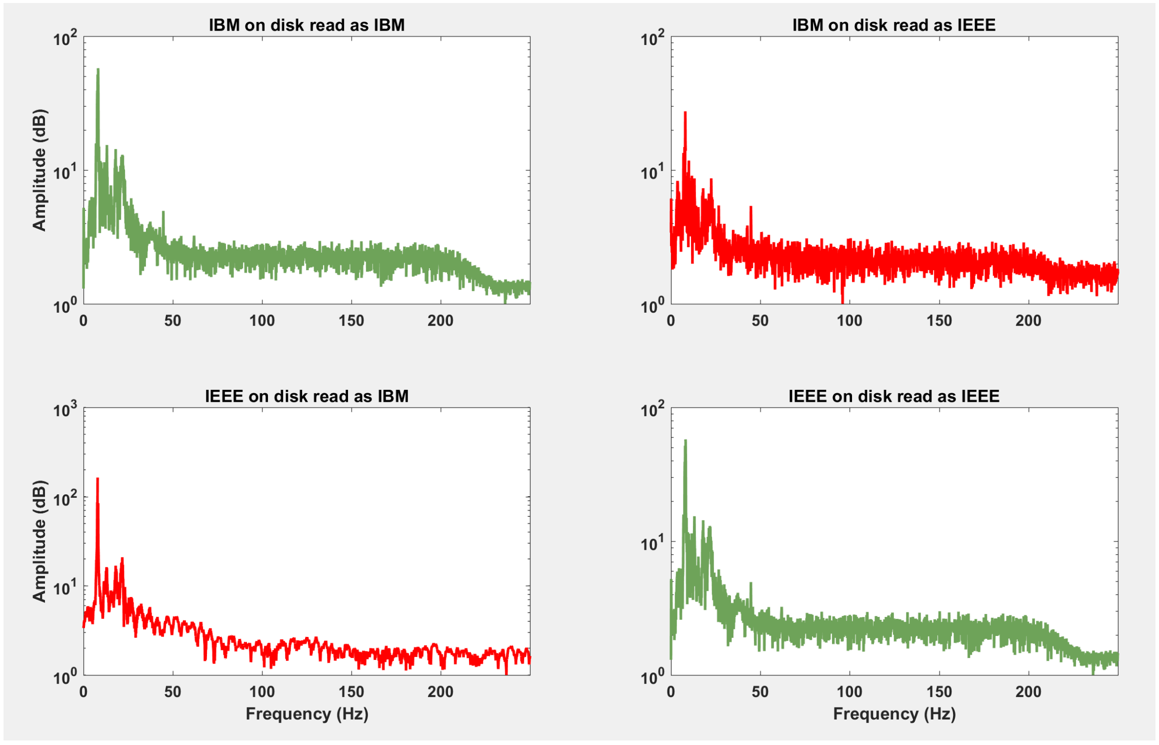

FIG. 4. Amplitude spectra corresponding to the traces shown in Figure 3.

Table 1. Comparison of IBM and IEEE floating point formats.

	IBM 32-bit	IEEE 32-bit	IEEE 64-bit
Base	16	2	2
Exponent Bias	+64	+127	+1023
Min Exponent	-64	-126	-1022
Max Exponent	63	127	1023
Sign-bits	bit 32 (1-bit)	bit 32 (1-bit)	bit 64 (1-bit)
Exponent-bits	bits 25-31 (7-bits)	bits 24-31 (8-bits)	bits 53-63 (11-bits)
Fraction bits	bits 1-24 (24-bits)	bits 1-23 (23-bits)	bits 1-52 (52-bits)
Fraction	fraction = 0+Fraction	fraction = 1+Fraction	fraction = 1+Fraction
Minimum Value	5.3976e-79	1.1755e-38	2.2251e-308
Maximum Value	7.2370e+75	3.4028e+38	1.7977e+308
Hard Zero	Fraction = 0, Exponent = 0	Fraction = 0, Exponent = 0	Fraction = 0, Exponent = 0
IEEE exceptions			
Zero and Denormal numbers	N/A	Exponent = -127 fraction = 0+Fraction	Exponent = 0 or -1023 fraction = 0+Fraction
+Inf, -Inf	N/A	Exponent =128 Fraction = 0.0	Exponent =1024 Fraction > 0.0
NaN	N/A	Exponent =128 Fraction > 0.0	Exponent =1024 Fraction > 0.0

Table 2. Examples.

Matlab Input	Input Type	Output Type	Binary	Hexadecimal	Sign (Dec)	Base (Dec)	Exp+Bias (Dec)	Bias (Dec)	Exp (Dec)	Fraction (Dec)	Result (Excel)	Result (MATLAB)	Difference
0.15625	single	ibm32	01000000 00101000 00000000 00000000	40280000	0	16	64	64	0	0.156250	1.56250E-01	1.56250E-01	0.00000E+00
0.15625	single	ieee32	00111110 00100000 00000000 00000000	3E200000	0	2	124	127	-3	0.250000	1.56250E-01	1.56250E-01	0.00000E+00
0.0	single	ibm32	00000000 00000000 00000000 00000000	00000000	0	16	0	64	-64	0.000000	0.00000E+00	0.00000E+00	0.00000E+00
0.0	single	ieee32	00000000 00000000 00000000 00000000	00000000	0	2	0	127	-127	0.000000	0.00000E+00	0.00000E+00	0.00000E+00
-0.0	single	ibm32	00000000 00000000 00000000 00000000	00000000	0	16	0	64	-64	0.000000	0.00000E+00	-0.00000E+00	0.00000E+00
-0.0	single	ieee32	10000000 00000000 00000000 00000000	80000000	1	2	0	127	-127	0.000000	0.00000E+00	-0.00000E+00	0.00000E+00
IBM_MAX	double	ibm32	01111111 11111111 11111111 11111111	7FFFFFFF	0	16	127	64	63	1.000000	7.23701E+75	7.23701E+75	0.00000E+00
-IBM_MAX	double	ibm32	11111111 11111111 11111111 11111111	FFFFFFFF	1	16	127	64	63	1.000000	-7.23701E+75	-7.23701E+75	0.00000E+00
IBM_MIN	double	ibm32	00000000 00010000 00000000 00000000	00100000	0	16	0	64	-64	0.062500	5.39761E-79	5.39761E-79	0.00000E+00
-IBM_MIN	double	ibm32	10000000 00010000 00000000 00000000	80100000	1	16	0	64	-64	0.062500	-5.39761E-79	-5.39761E-79	0.00000E+00
Inf	single	ibm32	01111111 11111111 11111111 11111111	7FFFFFFF	0	16	127	64	63	1.000000	7.23701E+75	7.23701E+75	0.00000E+00
Inf	single	ieee32	01111111 10000000 00000000 00000000	7F800000	0	2	255	127	128	0.000000		Inf	
-Inf	single	ibm32	11111111 11111111 11111111 11111111	FFFFFFFF	1	16	127	64	63	1.000000	-7.23701E+75	-7.23701E+75	0.00000E+00
-Inf	single	ieee32	11111111 10000000 00000000 00000000	FF800000	1	2	255	127	128	0.000000		-Inf	
NaN	single	ibm32	01111111 11111111 11111111 11111111	7FFFFFFF	0	16	127	64	63	1.000000	7.23701E+75	7.23701E+75	0.00000E+00
NaN	single	ieee32	11111111 11000000 00000000 00000000	FFC00000	1	2	255	127	128	0.500000		NaN	
pi	single	ibm32	01000001 00110010 01000011 11110111	413243F7	0	16	65	64	1	0.196350	3.14159E+00	3.14159E+00	0.00000E+00
pi	single	ieee32	01000000 01001001 00001111 11011011	40490FDB	0	2	128	127	1	0.570796	3.14159E+00	3.14159E+00	0.00000E+00
realmax	single	ibm32	01100000 11111111 11111111 11111111	60FFFFFF	0	16	96	64	32	1.000000	3.40282E+38	3.40282E+38	0.00000E+00
realmax	single	ieee32	01111111 01111111 11111111 11111111	7F7FFFFF	0	2	254	127	127	1.000000	3.40282E+38	3.40282E+38	0.00000E+00
-realmax	single	ibm32	11100000 11111111 11111111 11111111	E0FFFFFF	1	16	96	64	32	1.000000	-3.40282E+38	-3.40282E+38	0.00000E+00
-realmax	single	ieee32	11111111 01111111 11111111 11111111	FF7FFFFF	1	2	254	127	127	1.000000	-3.40282E+38	-3.40282E+38	0.00000E+00
realmin	single	ibm32	00100001 01000000 00000000 00000000	21400000	0	16	33	64	-31	0.250000	1.17549E-38	1.17549E-38	-0.00000E+00
realmin	single	ieee32	00000000 10000000 00000000 00000000	00800000	0	2	1	127	-126	0.000000	1.17549E-38	1.17549E-38	-0.00000E+00
-realmin	single	ibm32	10100001 01000000 00000000 00000000	A1400000	1	16	33	64	-31	0.250000	-1.17549E-38	-1.17549E-38	-0.00000E+00
-realmin	single	ieee32	10000000 10000000 00000000 00000000	80800000	1	2	1	127	-126	0.000000	-1.17549E-38	-1.17549E-38	-0.00000E+00

METHOD

A new function called *log16()* has been written that emulates the behavior of the Matlab built-in *log2()* function:

$$l = log16(d) \text{ and} \tag{a}$$

$$[f,e] = log16(d), \tag{b}$$

where *log16()* returns the base 16 logarithm of *d* if it is called with a single output argument, or the fraction (*f*) and integer exponent *(e)* required to encode *d* as IBM floating-point. This function is called by

$$u = num2ibm(d,lims), \tag{c}$$

where *u* is an IBM float stored in an unsigned 4-byte integer and *d* represents the decimal value(s) to encode. *d* can be any real Matlab data type. *lims* can be set to either ‘ibm’ (default) or ‘ieee’, and refers to the minimum and maximum values that can be stored in IBM and IEEE four-byte formats. Table 1 and Figure 5 show the behaviour of *num2ibm()* for the various limits. Values between the positive and negative minimum are set to 0.0. Values greater than the positive and negative maximum are set to the maximum.

Table 2 shows examples for a selection of representative numbers. Column one shows the Matlab command used to generate the number (eg. realmax(‘single’)), column 2 shows whether the number is double or single precision in Matlab, column 3 shows whether the number was converted to IBM floating point ‘ibm32’ using *num2ibm()* or not ‘ieee32’, and columns 4 and 5 show the output from *num2ibm()* in binary (*sign* is magenta, *exponent* is blue and *fraction* is green; see Equation 1) and hexadecimal. The remaining columns show the values of the *sign*, *exponent* and *fraction* extracted from the binary string in column 4 using Microsoft Excel, the decimal value from Equation 1, and a final comparison to the Matlab number. Note that +/-Inf and NaN have become +/-IBM maximum. *Exponents* in red correspond to IEEE exceptions (see Table 1).

Any IBM floating-point number stored as an unsigned four-byte integer *(u)* can be converted to a decimal value (*d)* by calling

$$d = ibm2num(u). \tag{d}$$

Example 1

```
>> fid = fopen(‘test.sgy’,’r’)    % open file for reading
>> fseek(fid,3840,’bof’)           % skip SEG-Y text file header, binary file header and first trace header
>> u=fread(fid, 500, 'uint32=>uint32')  % read 500 4-byte unsigned integers and return as uint32
>> fclose(fid)                     % close file
>> d = ibm2num(u)                  % convert IBM stored in uint32 to a Matlab single (4-byte IEEE)
```

Example 2

```
>> fid = fopen(‘test.sgy’,’a’)    % open file with existing headers in order to append trace data
>> fwrite(fid, num2ibm(d), 'uint32')  % write 500 4-byte unsigned integers to disk
>> fclose(fid)                     % close file
```

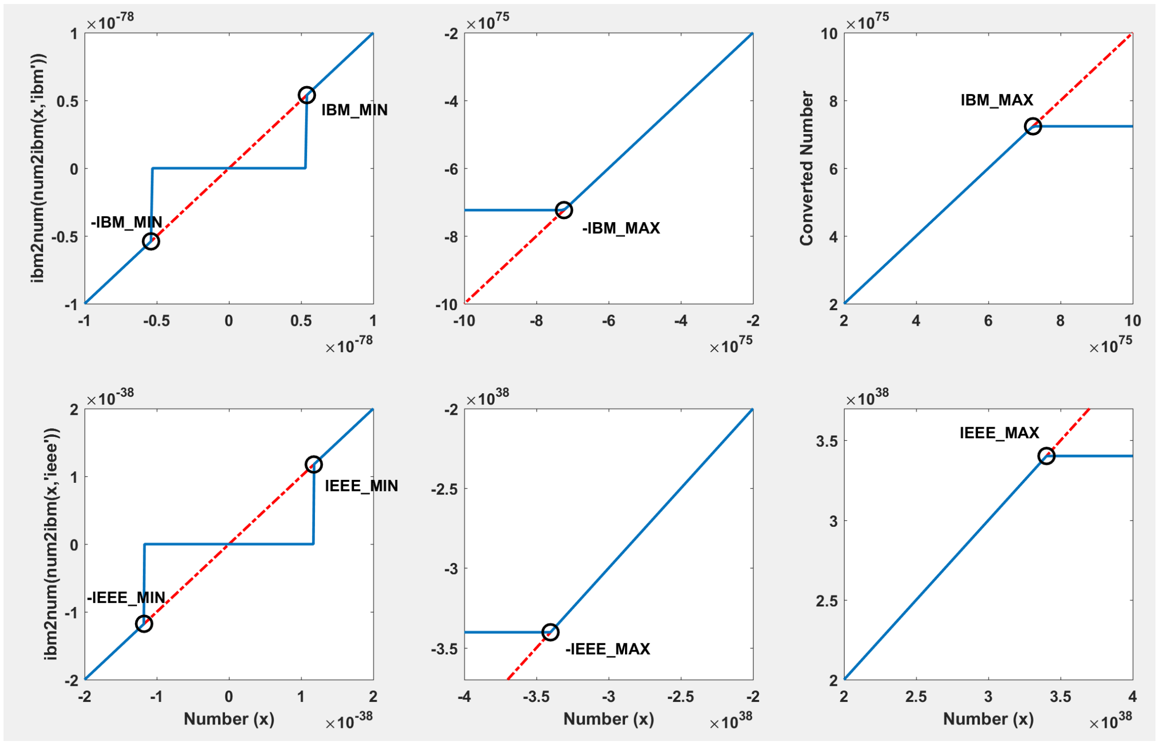

FIG. 5. Behaviour of *num2ibm()* at the maximum and minimum values that can be stored in IBM and IEEE floating point formats for *lims*='ibm' (top) and *lims*='ieee' (bottom). The red dashed line shows the data to be stored (Matlab double) and the solid blue line shows what is actually stored after converting to IBM floating-point. Values between the positive and negative minimum are set to 0.0. Values greater than the positive and negative maximum are set to the maximum.

SYNTHETIC EXAMPLES

A sinusoid with amplitudes varying between minus one and plus one was created, replicated, and multiplied by powers of 10 so the resulting amplitudes cover both the IBM and the IEEE maximum and minimum numbers that can be stored. This synthetic dataset was written to disk using *writesegy()* as IBM floating point using both *num2ibm(d,'ibm')* and *num2ibm(d,'ieee')*. The results when read back using *readsegy()* are shown in Figure 6 and Figure 7. See Hall and Margrave (2017) for descriptions of *readsegy()* and *writesegy()*. These figures show the expected results.

Figures 8, 9, and 10 show the results of reading the SEG-Y file shown in Figure 6 into three different seismic processing software packages. Clearly, while it is possible to store larger and smaller numbers in a four-byte IBM float than in a four-byte IEEE float, it will not be possible to read them back for processing. If you need to store very small or very large numbers, it would be better to use eight-byte IEEE floats and SEG-Y revision 2.

ARE MY TRACE DATA IBM OR IEEE?

Figure 11 shows a 3C source gather stored as IEEE on disk that has been read incorrectly as IBM and converted to IEEE by *readsegy()*. If the data are converted to IBM, back to IEEE, and subtracted from the data shown in Figure 11 we get the results shown in Figure 12. The inset graph shows cumulative amplitude differences per trace. The largest amplitude differences are associated with the source location (bigger

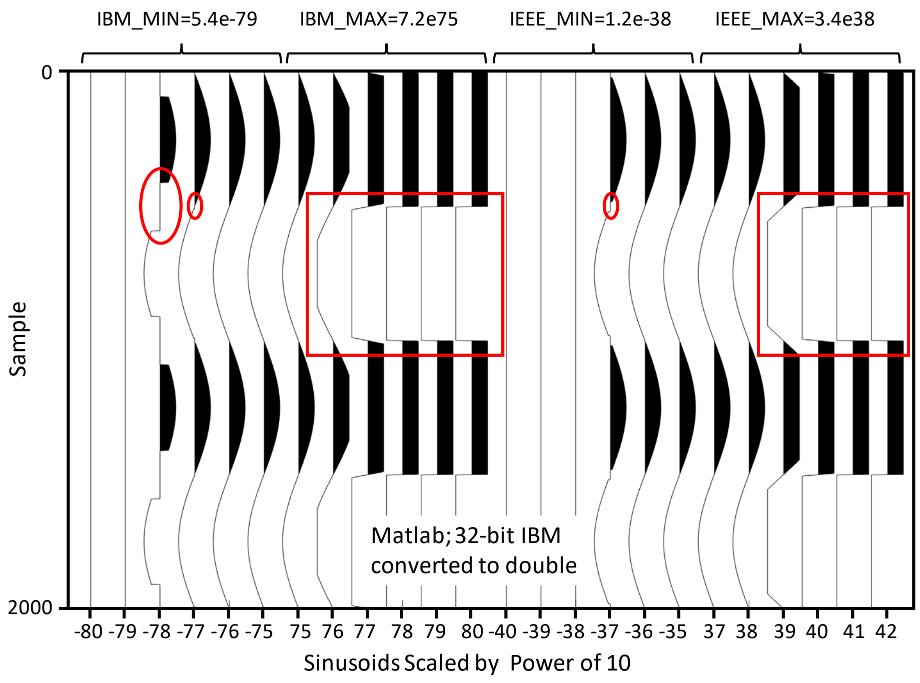

FIG. 6. Sinusoids written to SEG-Y file as IBM floats with IBM limits with *writesegy(),* then read back into Matlab with *readsegy()* and displayed using *plotseis()*. Red ovals highlight artifacts at values less than IBM minimum, and red rectangles highlight artifacts at values greater than IBM maximum.

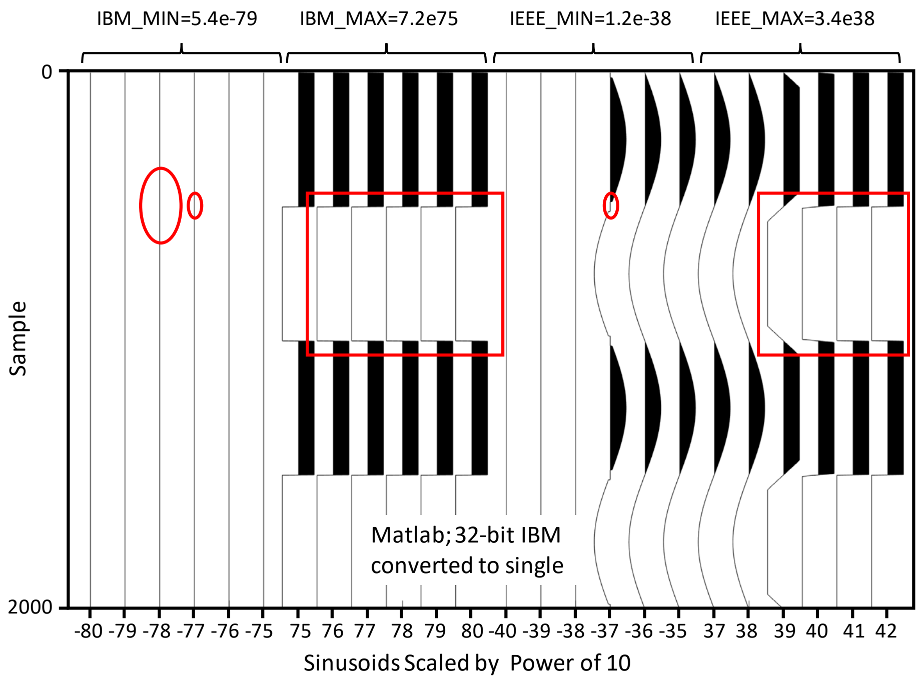

FIG. 7. Sinusoids written to SEG-Y file as IBM floats with IEEE limits with *writesegy(),* then read back into Matlab with *readsegy()* and displayed using *plotseis()*. Red ovals highlight artifacts at values less than IBM minimum, and red rectangles highlight artifacts at values greater than IBM maximum.

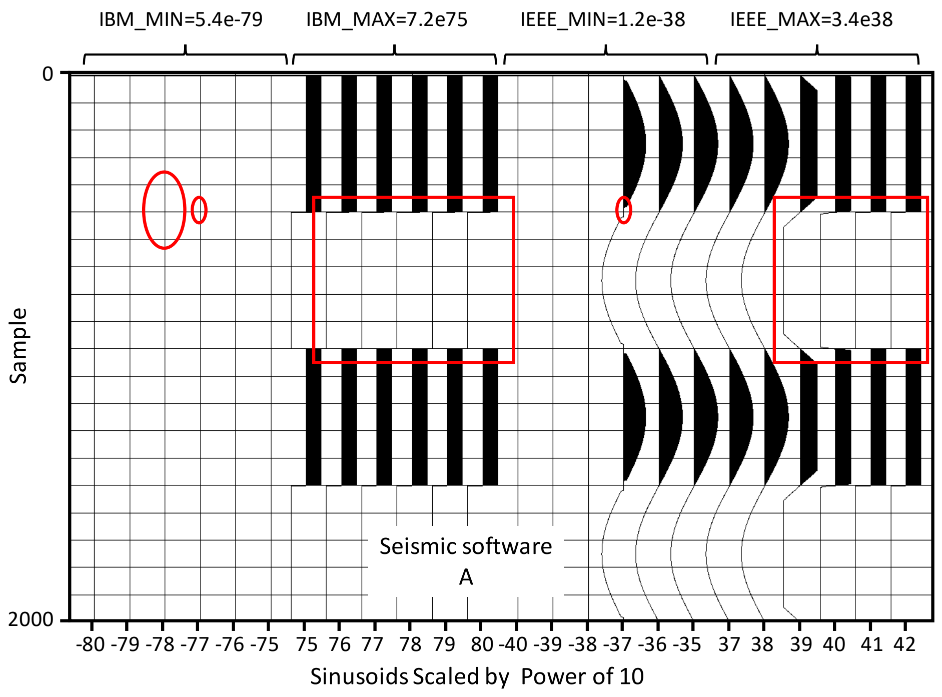

FIG. 8. Sinusoids written to SEG-Y file as IBM floats with IBM limits with *writesegy(),* then read back into Seismic Software A and displayed. Red ovals highlight artifacts at values less than IBM minimum, and red rectangles highlight artifacts at values greater than IBM maximum.

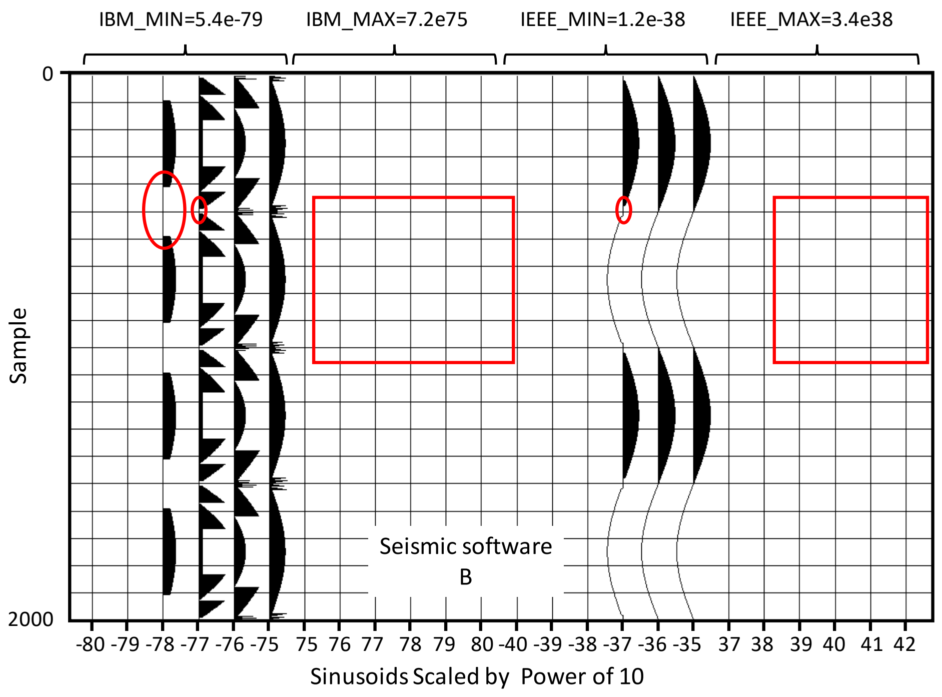

FIG. 9. Sinusoids written to SEG-Y file as IBM floats with IBM limits with *writesegy(),* then read back into Seismic Software B and displayed. Red ovals highlight artifacts at values less than IBM minimum, and red rectangles highlight artifacts at values greater than IBM maximum.

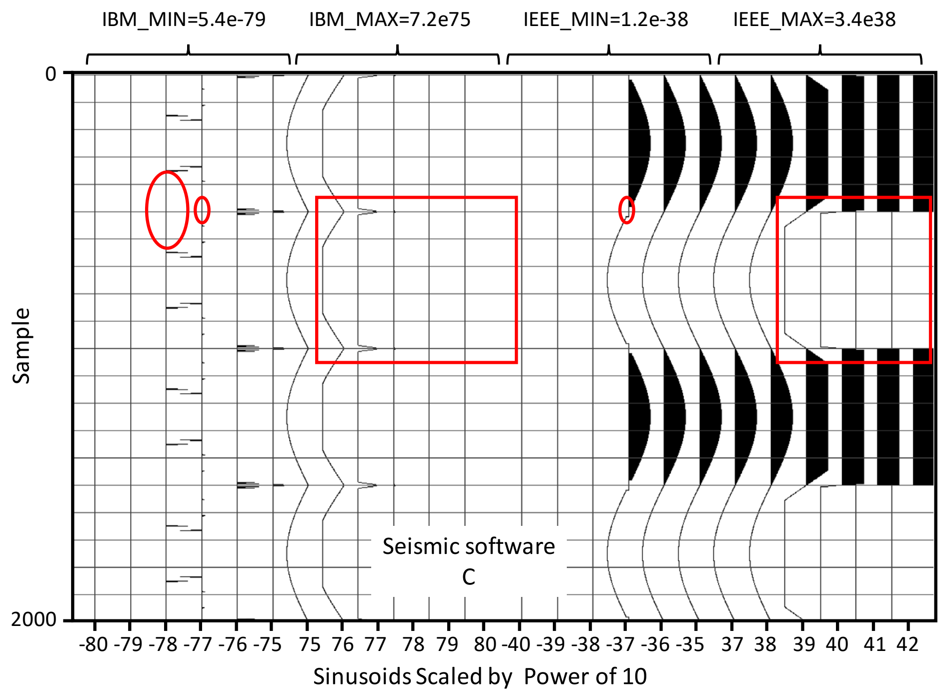

FIG. 10. Sinusoids written to SEG-Y file as IBM floats with IBM limits with *writesegy(),* then read back into Seismic Software C and displayed. Red ovals highlight artifacts at values less than IBM minimum, and red rectangles highlight artifacts at values greater than IBM maximum.

amplitudes?) and the auxiliary traces (right-hand side). While most of the cumulative differences per trace are near zero, none of them are exactly zero. If we start with IBM data on disk, read it correctly as IBM and go through the same process, the cumulative differences are all exactly zero (not shown; blank graphs are boring).

Figure 13 shows the IBM precision wobble that results if the actual IEEE trace amplitudes are converted to IBM. Dark blue represents IBM *fractions* that are stored with 24-bit precision (no leading zero bits) and yellow represents *fractions* that are stored with 21-bit precision (three leading zero bits). Interestingly, IBM *fractions* with 21, 22, 23 and 24-bit precision are evenly distributed throughout the source gather (Figure 13, and histogram inset Figure 14). As you would expect, the largest amplitude differences correlate to the worst precision (ie. 21-bit precision, or 3 leading zero bits; Figure 14).

The CREWES Matlab toolbox SEG-Y reading functions now take advantage of this to guess whether the trace data in a disk file are IBM or IEEE if the format code is 1 using the following algorithm:

1) Read single traces from input file as IBM in a loop until we find a trace that is not all-zeros (takes care of trace padding) and allow *readsegy()* to convert to IEEE.
2) Convert to IBM and back using and *ibm2num(num2ibm()).*
3) Subtract (2) from (1) and sum the differences.
4) **IF** (3) is non-zero guess IEEE format, display a warning, and carry on after updating the format code to 5 **ELSE** guess IBM and carry on.

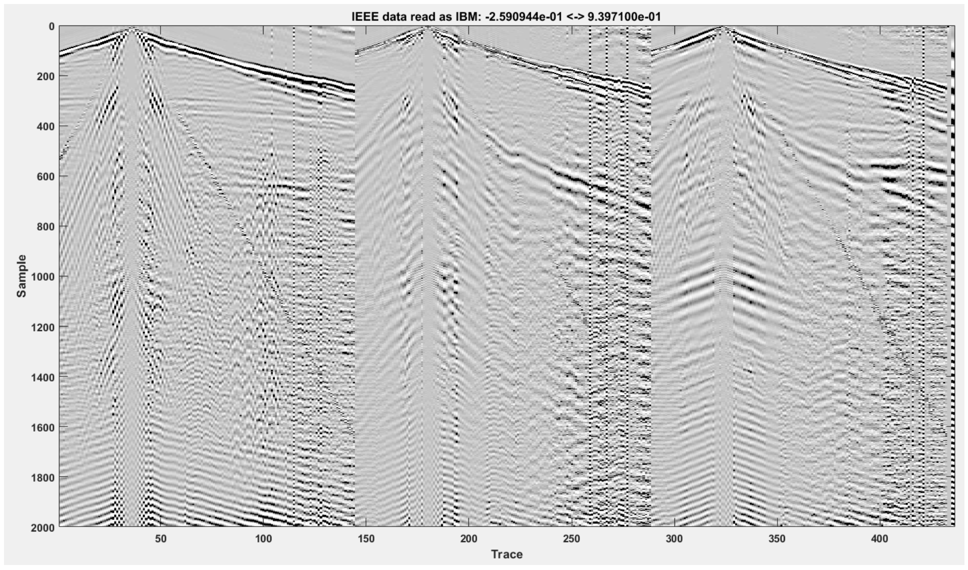

FIG. 11. Real data stored as little-endian IEEE floating point and read incorrectly into Matlab as IBM floating point with *readsegy()* and displayed using *plotimage()*.

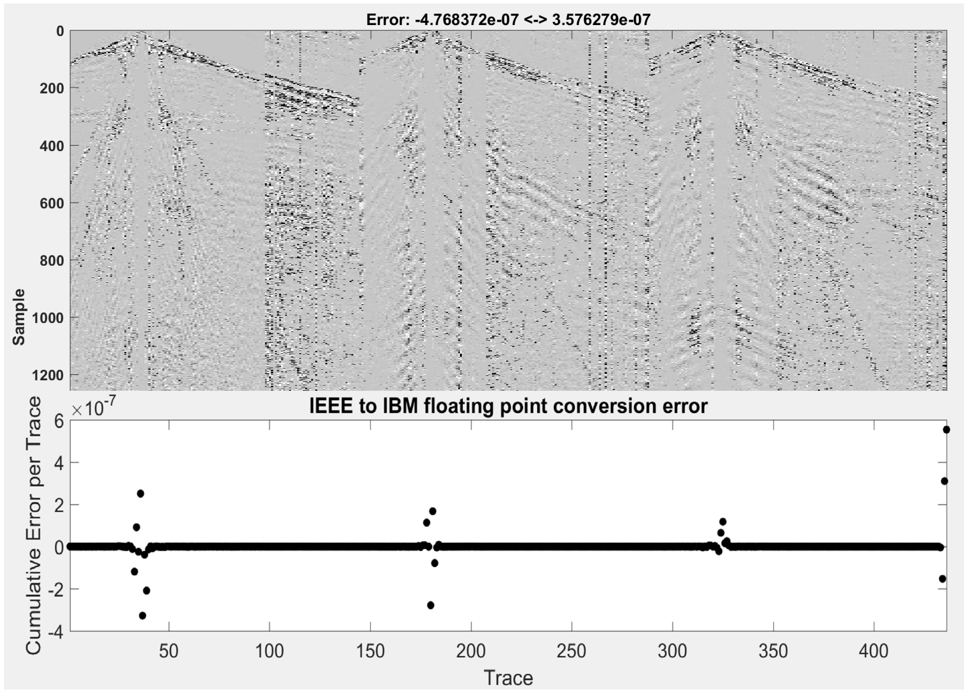

FIG. 12. Data shown in Figure 11 after conversion to IBM and back, subtracting from the data shown in Figure 11 and displayed using *plotimage()*. The cumulative error (sum of amplitude differences) per trace is displayed across the bottom.

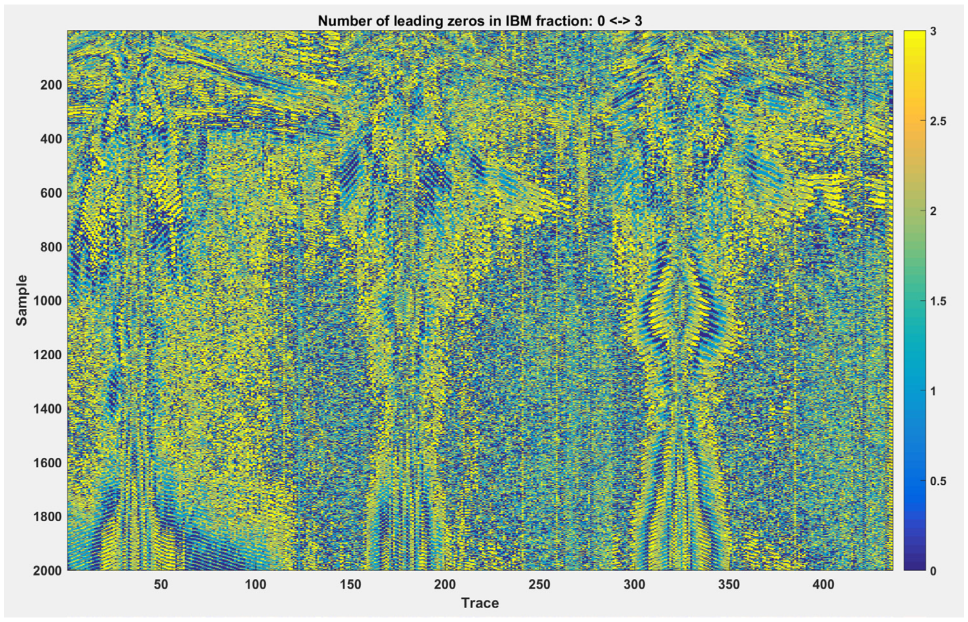

FIG. 13. Number of leading zeros in the most significant 4-bits (nibble) of the IBM floating-point fraction. Zero leading zeros means the fraction is stored with 24-bit precision. Three leading zeros means the fraction is stored with 21-bit precision.

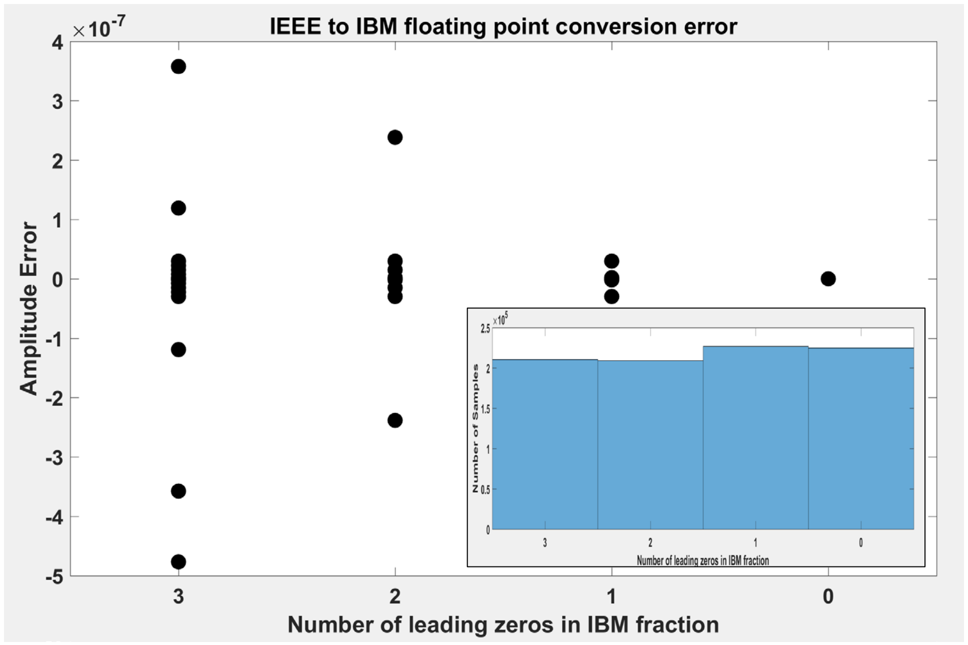

FIG. 14. Sum of amplitude differences plotted against the number of leading zeros in the most significant 4-bits (nibble) of the IBM floating-point fraction.

CONCLUSIONS

The CREWES Matlab® toolbox has long been able to read IBM floating-point, but has not been able to write IBM to disk. New functions have been written, tested, and released in the toolbox that allow IBM floating-point format to be written to disk. However, doing this results in a loss of precision due to the IBM fraction being stored with between 21 and 24-bit precision, meaning that trace amplitudes written to disk as IBM will not be identical to the original IEEE amplitudes.

While IBM can store both larger and smaller numbers in four-bytes than IEEE, these values could not be read successfully by three seismic software packages that were tested. One of the software packages returned numbers that were correct, but constrained by IEEE limits on size. Two of the software packages were not able to do this. If you need to store very large or very small numbers you should use IEEE eight-byte (double precision) floating-point and SEG-Y revision 2, or scale your numbers so they can be stored in a four-byte IEEE float. Software that can read SEG-Y revision 2 should become more prevalent as time goes on.

SEG-Y files with the data sample format code set to one (IBM floating point) can be ambiguous, since sometimes the trace data are actually stored as IEEE floating-point. If IEEE data are read incorrectly as IBM, and then converted to IEEE and back to IBM and subtracted from the original data, the sum of the differences is non-zero. If IBM data are read as IBM, and then converted to IEEE and back to IBM and subtracted from the original data, the sum of the differences is zero. This can be used to distinguish between IBM and IEEE floating-point data on disk.

It is faster to write SEG-Y to disk using the byte-order and floating-point format native to the computer upon which the data resides, rather than having to swap bytes and convert floating point formats. For example, the author would recommend little-endian byte order and IEEE floating-point on a modern PC, which is allowed by the SEG-Y revision 2 standard, rather than converting to big-endian as required by revisions 0 and 1, and IBM floating point as required by revision 0.

REFERENCES

Barry, K. M., Cavers, D. A. and Kneale, C. W., 1975, Report on recommended standards for digital tape formats: Geophysics, 40, no. 02, 344-352.

Hagelund, R., and Stewart, A.L., Eds., SEG Technical Standards Committee, 2017, SEG-Y _r2.0: SEG-Y revision 2.0 Data Exchange Format: SEG, Tulsa, OK, www.seg.org.

Hall, Douglas V., 1980, Microprocessors and Digital Systems, McGraw-Hill.

Hall, K.W. and Margrave, G.F., Updates to SEG-Y I/O in the CREWES Matlab Toolbox, this volume.

Harding, L. J. (1966), "Idiosyncrasies of System/360 Floating-Point", Proceedings of SHARE 27, Aug. 8–12 1966, Presented at SHARE XXVII, Toronto, Canada

IBM, 1967, IBM System/360 Principles of Operation, IBM Publication A22-6821-6, Seventh Edition.

IEEE Computer Society, 2008, *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008.*

Norris, M.W. and Faichney, A.K., Eds., SEG Technical Standards Committee, 2002, SEG-Y rev 1 Data Exchange Format: SEG, Tulsa, OK, www.seg.org.

Tomayko, J.,1995, System 360 Floating-Point Problems, IEEE Annals of the History of Computing, **17**, 62–63.

ACKNOWLEDGEMENTS

We thank the sponsors of CREWES for continued support. This work was funded by CREWES industrial sponsors and NSERC (Natural Science and Engineering Research Council of Canada) through the grant CRDPJ 461179-13.

APPENDIX A

The smallest piece of information that can be stored by a computer is called a bit. A bit can have a value of zero or one (Table A.1). The smallest number of bits that can be easily accessed is called a byte, where one byte is eight bits in size. Half of one byte (four bits) is called a nibble, and conveniently, one nibble is big enough to store a hexadecimal number. Two hexadecimal numbers can be used to describe the contents of all the bits in one byte. For example, 'FF' means that all the bits in one byte are set to one (Table A.2).

While IEEE is base 2, IBM is base 16 (Table A.1). On a little-endian computer (bit 32 is visualized to be on the left), multiplying the significand by a power of 2^1 moves the decimal point to the left one bit and by 2^{-1} moves the decimal point to the right one bit. Similarly, multiplying by 16^1 moves the decimal point 4-bits (one nibble) to the left and 16^{-1} moves the decimal point 4-bits to the right.

Table A.3 shows how bits are converted to integer values by interpreting the location of a bit within a byte as representing a power of 2, then summing.

Table A.1. Bits, nibbles, bytes and bitshifts.

Example	(Bin)	(Hex)	(Dec)
Bit	0 or 1	0 or 1	0 or 1
Nibble (4-bits)	0001	1	1
Byte (8-bits)	0000 0001	01	1
Bitshift N bits towards most significant bit == x*2^N	0000 0001 << 4 = 0001 0000	10	1*2^4 = 16
Bitshift N bits towards least significant bit == x*2^-N	0001 0000 >> 2 = 0000 0100	04	16*2^-2 = 4

Table A.2. Nibbles

Nibble (Bin)	(Hex)	(Dec)
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Table A.3. Unsigned integer value of a byte

8-bit unsigned integer								
Bit #	8	7	6	5	4	3	2	1
Bit Value= 2^(Bit#-1)	128	64	32	16	8	4	2	1
Example								
Bit	1	0	0	0	0	1	1	0
Bit Value	128	0	0	0	0	4	2	0
Sum (Dec)	134							