The new FST compression algorithms :
Compression of floating point data (32-bit IEEE format)

Yves Chartier – October 2006

Introduction

In the summer of 2004, the author implemented a fast and efficient compression algorithm to compress the quantized integer stream of values used in RPN standard file (datyp=1 or 6). The following document describes another method used to compress floating point data stored in the IEEE 32-bit format (datyp=5). The method can be used for lossy or lossless compression.

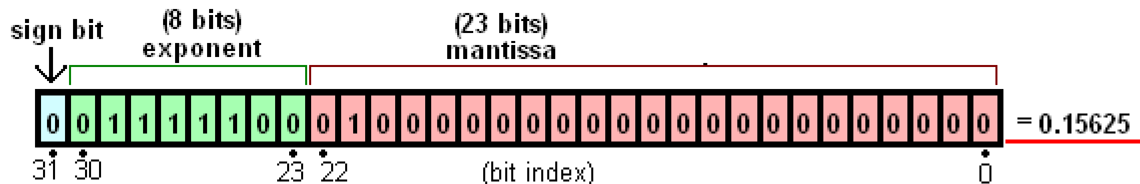
Generally, floating point numbers inside RPN standard files are stored internally as a stream of unsigned integers through a process called **quantization**. The size of each quantized values is NBITS. NBITS can vary from 1 to 24 bits, 12 and 16 bits being the values used the most often. This yields a compression ratio of 2.0 for the R16 data type (an R16 field is composed of 16-bit unsigned integers) and 2.67 for the R12 (12-bit unsigned integers) data type.

However, 32-bits and 64-bits floating point numbers can also be stored in RPN standard files under the IEEE format. This can be done by setting datyp=5 and nbits=-32. What is stored in the file is a verbatim representation of what there is in memory.

At the moment of writing there is a good web reference of the IEEE 32-bit floating point representation at the Wikipedia web site URL : http://en.wikipedia.org/wiki/IEEE_Floating_Point_Standard.

In a single precision 32-bit number, there is 1 bit for the sign, e=8 for the exponent and f=23 for the mantissa. [1]

The following picture shows the representation of the number 0.15625



Fron the Wikipedia article :

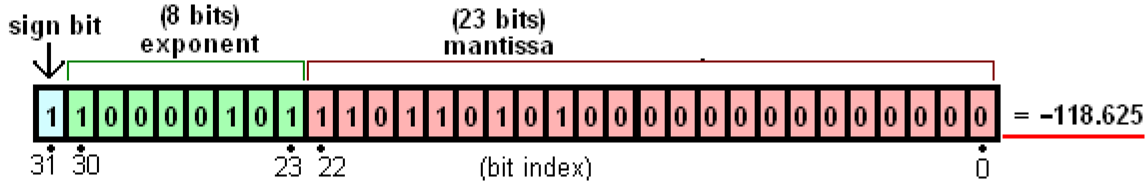
> *The exponent is biased by $2^{8-1} - 1 = 127$ in this case, so that exponents in the range −126 to +127 are representable. An exponent of −127 would be biased to the value 0 but this is reserved to encode that the value is a denormalized number or zero. An exponent of 128 would be biased to the value 255 but this is reserved to encode an infinity or not a number.*
>
> *For normalised numbers, the most common, Exp is the biased exponent and Fraction is the fractional part of the significand. The number has value v:*
>
> $v = s \times 2^e \times m$
>
> *Where*
>
> *s = +1 (positive numbers) when the sign bit is 0*
>
> *s = −1 (negative numbers) when the sign bit is 1*
>
> *e = Exp − 127 (in other words the exponent is stored with 127 added to it, also called "biased with 127")*
>
> *m = 1.Fraction in binary (that is, the significand is the binary number 1 followed by the radix point followed by the binary bits of Fraction). Therefore, $1 \le m < 2$.*

1 Image credits : http://en.wikipedia.org/wiki/IEEE_Floating_Point_Standard.

An example

Let us encode the decimal number −118.625 using the IEEE 754 system.

1. *First we need to get the sign, the exponent and the fraction. Because it is a negative number, the sign is "1".*
2. *Now, we write the number (without the sign) using [binary notation](). The result is 1110110.101.*
3. *Next, let's move the radix point left, leaving only a 1 at its left: 1110110.101 = 1.110110101 × 2^6. This is a normalized floating point number. The [mantissa]() is the part at the right of the radix point, filled with 0 on the right until we get all 23 bits. That is 11011010100000000000000.*
4. *The exponent is 6, but we need to convert it to binary and bias it (so the most negative exponent is 0, and all exponents are non-negative binary numbers). For the 32-bit IEEE 754 format, the bias is 127 and so 6 + 127 = 133. In binary, this is written as 10000101.*



In base 10,

$-118.625 = -1 * (1 * 10^2 + 1 * 10^1 + 8 * 10^0 + 6 * 10^{-1} + 2 * 10^{-2} + 5 * 10^{-3})$

Sign	10^2	10^1	10^0	.	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}
1	1	1	8	.	6	2	5	0	0	0	0

In base 2,

$$
\begin{aligned}
-118.625 &= -1 * (64 + 32 + 16 + 4 + 2 + 1/2 + 1/8) \\
&= -1 * (1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^2 + 1 * 2^1 + 1 * 2^{-1} + 1 * 2^{-3}) \\
&= -1 * 2^{6} * (1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-4} + 1 * 2^{-5} + 1 * 2^{-7} + 1 * 2^{-9})
\end{aligned}
$$

Since the first term of the mantissa is always 1, it is omitted from the representation but its presence is always implied (the famous hidden one). Thus the IEEE representation of this number is

Sg Exponent Power of 2

									-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
1	1	0	0	0	0	1	0	1	1	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Lossy compression through mantissa reduction

A fast way to compress the space taken by the floating point values is to reduce the size of the mantissa. Using the fact that log10/log2 is 3.32, it takes 3.32 bits to approximate an order of magnitude (or a significant digit).

Here is a short table of equivalence between the size of the mantissa and the number of significant digits

Significant digits	***Number of bits***
7	23
6	19
5	16
4	13
3	9
2	6
1	3

Thus, writing a standard file record while keeping 4 significant digits would imply to call FSTECR with datyp=5 and NBITS=-22 (13 bits for the mantissa + 8 bits for the exponent+ 1 sign bit). The compression ratio obtained here would be 32 /22 = 1.45, and the savings would be (32-22)/32 = 10/32 = 31.2 %. But each number would have a relative error of +/- 1/1000.

Lossless compression through the Lorenzo predictor

Since the Lorenzo predictor gave very good results at compression of quantized unsigned integers, it was decided to re-use this technique for the IEEE format. However, using it on the raw 32-bit tokens would have given less than optimal results. So it was decided to split the tokens into 3 different streams (sign, exponent and mantissa) and to compress each of them independently.

Compression of the sign

This one may be very gratifying, since for data fields that are either all positive or all negative, only 1 bit is necessary to define the sign of the whole field (0=+ve, 1=-ve). However, when a mix of +ve and -ve values are present, we need to apply a special algorithm to the stream.

The Lorenzo predictor is not very efficient on 1-bit value stream, because there is already 1 bit needed for the sign of the prediction errors. So unless the stream contains a high percentage of areas having the same sign, a Lorenzo compressed stream will be often larger than the original.

The following algorithm was introduced, using 8-bit tokens.

- Start with the 1st token
- Look forward in the data stream until
  - a complement value (ie 1 if start value is 0 and 0 if start value is 1) is found
  - the end of the data stream is reached
- If the sequence is <= 7, encode the sequence itself
- If the sequence is > 7, encode the bit + the length of the sequence, in chunks of 62
- There is a special case that if the sequence length is 63, then the repeat count is 255.

If the bit 7 of token is 0, bits 6-0 are taken as is. If bit 7 is 1, then bit 6 is the value to be repeated, and bits 5-0 contain the byte count.

So the sequence

0000000010001110101111111111111111111111111111111111111111111111

Would be encoded as

The savings here are 50 %.

Compression of the exponent

This the part where the compression is the most efficient. Meteorological fields rarely have an exponent span of 255 values (it would take a field that goes from 10^{-38} to 10^{+38} ! In fact it rarely exceeds 32 (2^5). In that case, 5 bits are sufficient to represent the exponents of the data field. Also, since the exponents tend to cover vast areas where they have the same values, the Lorenzo algorithm gives a very good compression ratio on these areas since the prediction are almost always exact.

The algorithm is as follows :

- Assemble the exponents into 8-bits tokens
- Determine the range of the exponents (min and max)
- Subtract the min exponent from all
- Encode the remaining stream with the Lorenzo algorithm

It happens sometimes that the exponent is the same for the whole field. In that case, the compression ration is extreme : a single value is kept for the whole field.

Compression of the mantissa

The processing of the mantissa stream is very close to the one of the exponent, except that the tokens are treated as unsigned 32-bit integers and there is no unbiasing of the data.

Stream assembly

The sign, exponent and mantissa streams are assembled together to form the compressed record.

Lossy compression through the Lorenzo predictor

The compression algorithm works the same way as in the lossless case, except that the mantissa is cut by (23 – nbits) before being compressed. This does not change anything to the processing of the sign and the exponent.

Performance

Compression performance

As one can expect, the compression performance depends a lot on the type of data field being compressed, as well as its physical resolution.

- Fields that have a highly variable dynamic range do not compress very well, and that is expected. The separation of the mantissa and the exponent can make the Lorenzo predictor quite inefficient, as the shift in the exponent can have the same effect as to multiply or divide the mantissa by several powers of 2.
- The ratio of the increased savings by the reduction the mantissa are proportional to the one observed for un-reduced data... Ie if a 400 Mb file reduces to 200 Mb in lossless mode, reducing the mantissa to 16 bit from 23 will give the same 16/23 ratio, ie a file that will be about 16 * 200 / 23 = 139 Mbytes.
- Regarding the resolution... the higher the resolution, the better the compression.

On a pure 801x600 meso-global 32-bit 528 Mbytes file containing about 300 records, the overall lossless compression ratio obtained is 2.0. As said, the compression ratios vary a lot among the fields. Here is a quick table of what was observed.

Nomvar	***Compression ratio***	***Equivalent nbits***
GZ	7.2	4,5
ES	2.6	12.2
TT	2,0	16
PN	2,7	11,6
PR	1,13	28,5
UU	1,42	22,5
VV	1,31	24,4

Compression performance compared to gzip and bzip2

These popular compression utilities were run on the same file, and here are the results.

Program	***Compressed size (Mb)***	***Comp. ratio***	***Real time***	***User time***	***System time***
fstcompress_300	259,3	2,03	38,52	12,97	3,72
gzip -9	322,9	1,64	119,22	104,78	2,06
bzip2 -9	314,7	1,68	309,72	304,86	2,20

Computation time

On a 3.0 GHz Pentium 4 under Debian Linux, fstcompress can process about 11 million floating point numbers / second, regardless of the source (ie E32, R12 and R16)., although R16 is slightly slower (10.5 millions fp/s). It takes about 38 seconds (real time) to compress an E32 521 Mbytes data file, and 16 seconds a 203 Mbytes R12 data file. Writing the compressed file to another hard drive (instead of working on the same drive) saves 15-20 % of wall clock time.