Denormal numbers in floating point signal processing applications

Laurent de Soras <laurent@ohmforce.com>
2002.01.11

ABSTRACT

Nowadays many DSP applications are running on personal computers using general purpose CPUs. However these processors do not have only advantages in the DSP field. This article will discuss the annoying calculation slowness, well known as "denormal bug" that is caused naturally in some circumstances during the processing of very small numbers. Solutions and useful workarounds are also proposed.

1. Floating point number coding overview

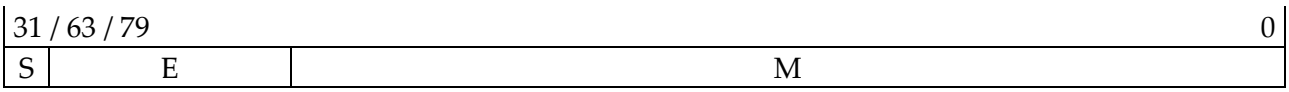
In order to follow the explanations in this paper, it is important to understand clearly how floating point IEEE numbers are coded [1]. There are three kinds of precision for these numbers:

- 32 bits, single precision
- 64 bits, double precision
- 80 bits, extended double precision

The first two ones are the most used for signal processing. A preference is given for 32-bit numbers because they use less memory bandwidth and are often accurate enough. In the following examples we will also use this number format. Algorithm and code snippets can be easily translated to the other floating point formats.

1.1 Normal mode

The normal mode is used to cover most of the floating point range. The representation of a number is divided into three parts: sign, mantissa and biased exponent. Mantissa and biased exponent are treated as positive integer numbers. Sign is only one bit, set for negative numbers.



The number value is given by the formula below:

$$x = \pm\left(1 + \frac{M}{2^P}\right) \times 2^{E-B} \quad (1)$$

Where:

- B is the exponent bias, a positive number depending on the precision. For example, it is equal to 127 for 32-bit numbers. B is used to produce negative exponent because E is coded as a positive integer.
- P is the mantissa resolution, expressed in bits.

Of course, there are some special cases where the coding is different. For example, 0 is coded by all bits to 0, except for the sign bit which is not taken into account. The other example is the denormal coding. NB: 80-bit numbers have the 1 of the formula explicitly coded in the mantissa.

1.2 Denormal mode

It is possible to code numbers which are so small that they cannot be coded with the normal format by a compromise on accuracy. The denormal coding is specified by a null biased exponent. Thus the value can be calculated by this second formula:

$$x = \pm M \times 2^{1-B-P} \quad (2)$$

The implicit "1" is removed in order to reach the smallest numbers by zeroing the most significant bits of the mantissa. Of course the resolution decreases because there are less significant bits. 1 is added to the exponent in order to achieve the continuity with the normal mode.

The major problem with denormal represented numbers is their processing time. It is much slower than for normal represented numbers. For example, on an AMD Thunderbird CPU, a multiplication with a denormal operand takes about 170 cycles, which is more than 30 times slower than with normal operands only ! When almost all the processed numbers are denormal, CPU load increases a lot and can affect the stability of real-time applications, even on the fastest systems.

2. Denormalization in DSP algorithms

The most perverse effect of denormalization comes from the structure of DSP algorithms. Signal flow generally follows a chain of basic building blocks. When an element generates denormal numbers, they will pass through the subsequent elements, slowing down the calculations again.

Denormalization is often caused by the use of feedback structures, and IIR filters are the most problematic ones. These filters are often preferred to FIR because of their computational cheapness, despite of their poor phase response properties.

2.1 One-pole filter

To simplify the analysis, let's start with the simplest recursive element, the one-pole filer. Its equation in the z-plane is:

$$H(z) = \frac{1}{1 - az^{-1}} \tag{3}$$

We can deduce the recursive equation:

$$y[k] = x[k] + ay[k-1] \tag{4}$$

For stability reasons, we always assume $|a| < 1$. When input is a not-null signal, there is usually no problem. On the other hand, as soon as input becomes and stays null, $|y[k]|$ exponentially converges to 0. The convergence speed depends on the constant a. Invariably, output is smaller and smaller, and finally turns into denormal before reaching 0.

Theoretically, after having been denormalized, output reaches 0, and remains null. However, the "to nearest" rounding rule has bad effects. When $y[k]$ reaches the smallest numbers, multiplication result can stay constant. I.e. 4 multiplied by 0.9 and rounded to the nearest integer gives 4 again. Therefore $|a| > \frac{1}{2}$ will keep the output constant below a given threshold. Output stays denormalized until input stops being 0.

Well, we can now point out two aspects of the problem:

- Output turning into denormal state, temporarily or not.
- Stability of this state, far worse for general performances.

2.2 Extension to feedback elements

Straight IIR implementation can be split into two parts connected in a serial way:

- Direct part (zeros): $u[k] = \sum_{i=0}^{N} b_i \, x[k-i]$ (5)
- Recursive part (poles): $y[k] = u[k] + \sum_{i=1}^{N} a_i \, y[k-i]$ (6)

The direct part can be seen as a FIR filter. Such filters rarely give denormal numbers. Numbers in the multiplications are generally many orders of magnitude above the denormal threshold. Moreover, if a product is far smaller than the other ones, it gets absorbed by the sum because of the limited resolution of the mantissa. Indeed, with $|x| >> |y|$, $x + y = x$. In spite of that, filters likely to completely cancel a signal should be handled carefully.

It is the all-pole part which is mainly responsible for the denormalization. When $u[k]$ becomes and stays null, one can see the same case as for the one-pole studied previously. This does not always mean that filter input $x[k]$ is null, because it can be cancelled by the FIR part. It is often the case in high-pass filters with a constant signal as input. Obviously, one can invert direct and recursive part order or by changing the implementation, if it is feasible. But the important thing to remember is the fact that a null input gives the pure recursive scheme.

We know how denormal state may appear. Its stability is far more difficult to understand. It depends on various parameters:

- Multiplying coefficient values.
- FPU (Floating Point Unit) rounding rules. It is generally "round to nearest", which increases state stability.
- Intermediary calculation order. In some cases, it is possible that two numbers may cancel each other during a sum.
- Intermediary calculation resolution. Final values could change depending on the precision of computation and storage.

Unfortunately, there is no miraculous formula to know if an algorithm is denormal-stable or not. One has to analyse carefully each algorithm implementation, given typical and exceptional input. However these clues should help to perform this analysis.

Another important thing is to note that these behaviours are not filter-specific. Any feedback system could be prone to these kinds of issue. For example, it is the case of feedback delay lines.

3. Solutions

We're going to examine two kinds of solutions. The first one is the denormal elimination, done after they appear. The second kind will try to prevent signal from becoming denormalized. In all the cases, it's more important to treat the feedback path than the output itself. The latter would be more a symptom dissimulation than a real cure.

Below are presented some methods to prevent the bad effects of denormal numbers. Most of them assume that meaningful data are many magnitude orders higher than the denormal threshold, which is generally the case. For example a number whose absolute value is below 10^{-20} can be considered as 0.

3.1 Explicit elimination of denormalized values

3.1.1 Detecting denormal numbers

The first idea is to replace every denormal number by 0. It implies to detect these numbers. As we said before, one recognizes a denormal number by its null biased exponent and not null

mantissa. This detection can be done by testing directly the bits of the data. The C++ function below tests a 32-bit floating point number and sets it to 0 if needed. The first line is used to treat floating point data as an integer to access its bit. This technique is very convenient and will also be used further down in this document.

```
void  test_and_kill_denormal (float &val)
{
   const int   x = *reinterpret_cast <const int *> (&val);  // needs 32-bit int
   const int   abs_mantissa = x & 0x007FFFFF;
   const int   biased_exponent = x & 0x7F800000;
   if (biased_exponent == 0 && abs_mantissa != 0)
   {
      val = 0;
   }
}
```

Although one can optimise this function, it was written here mainly for educational purposes. The point was to make it clear and readable to easily catch the concepts. One optimisation could be the removal of mantissa testing. However it still requires a test, which could be a pain for modern CPUs with long instruction pipelines.

+	Replace denormal numbers by true 0's Does not alter normal numbers
–	Test may break the instruction pipeline, slowing down the code. Requires a memory store to access the bits. Must be repeated at each stage of the process.

3.1.2 Elimination by quantification

The following function can remove denormal numbers without any test. The drawback is a slight loss of accuracy for the smallest number; anyway it is usually without consequence.

```
void  kill_denormal_by_quantization (float &val)
{
   static const float   anti_denormal = 1e-18;
   val += anti_denormal;
   val -= anti_denormal;
}
```

It relies on the limited resolution of mantissa. When a denormal number is added to the `anti_denormal` constant, it is absorbed because it is too small and cannot be coded in the final mantissa. Subtracting the constant again results in a true 0. With big (normal) numbers, it is the constant which is absorbed, and the operations have no effect. On the other hand, the first operation may slow down the code if done with a denormal. However this kind of operation has another useful role, as we will see later.

Depending on the case, one may have to correct the constant value. For example if the whole calculation is done in the FPU registers, a 80-bit arithmetic may be used, with 64-bit mantissas. The `anti_denormal` value should therefore be 2^{64} times higher than FLT_MIN. On the other hand, if

everything is done with a 32-bit accuracy, one may reduce the `anti_denormal` value to get a better accuracy for the small values.

+	Replaces denormal numbers by true 0's Quantification has a preventive effect on the outputted numbers Quite fast, only two additions.
–	Processing already denormal numbers slightly slows down the computation. Quantifies the smallest numbers, adding relative error. Must be repeated at each stage of the process.

3.1.3 Complementary detection

Another way to detect denormal numbers is FPU flags. Indeed, they can raise an exception (fortunately generally masked) at each operation with a denormal operand. A flag helps to keep the trace of this exception, and remains set until it is explicitly cleared. Thus, one just needs to check the flag regularly to ensure that no denormal numbers were used in operations. After having detected a denormal, one may hunt it with the functions shown above. However it requires assembly code or system specific functions. Here is an example, coded for x86 processor class, and compilable with MS Visual C++:

```
bool  is_denormalized ()
{
   short int   status;
   __asm
   {
      fstsw word ptr [status] ; Retrieve FPU status word
      fclex                   ; Clear FPU exceptions (denormal flag)
   }
   return ((status & 0x0002) != 0);
}
```

3.2 Preventing denormalization

Primarily, the task consists in adding noise to the signal, before or during the processing. This noise should be low enough not to alter the data, but should be able to pull the denormals over the fatal threshold when added to the signal.

The big benefit of noise addition is its propagation to the next stages in the processing flow. Except for special cases, this allows to cure the problem at the beginning and not to care about it afterwards.

There are various ways to generate noise, we will discuss some of them. Of course, the perfect and universal method does not exist: every solution should be carefully chosen and adapted to the context. Indeed, the noise spectral content is important, subsequent filters should not cancel it completely, making it useless.

3.2.1 White noise addition

A simple method is white noise generation. White noise has a uniform spectral content which makes it very well fit for numerous situations. It allows to "forget" the denormal issue on subsequent processing. As we do not need pure white noise, quality not being our main aim, we will focus on the generation speed.

```
unsigned int  rand_state = 1; // Needs 32-bit int

add_white_noise (float &val)
{
   rand_state = rand_state * 1234567UL + 890123UL;
   int   mantissa = rand_state & 0x807F0000; // Keep only most significant bits
   int   flt_rnd = mantissa | 0x1E000000;    // Set exponent
   val += *reinterpret_cast <const float *> (&flt_rnd);
}
```

It outputs random numbers whose magnitude is about 10^{-20}. It can be adjusted by modifying the line setting the exponent. Beware: if the spectrum is uniform, the DC component is very low. This may be a problem in certain cases. A workaround is to generate only positive numbers by modifying the second line with `rand_state & 0x007F0000`.

+ Fills the spectrum uniformly.
  Propagates to next stages.

– Not that fast.
  Requires a memory store/load to access the number bit-field.
  Alters the smallest numbers by quantifying them.

3.2.2 Noise addition – buffered version

In the `add_white_noise()` function, a significant percentage of CPU time is taken by the noise calculation. A common technique consists in pre-calculating the noise once, and storing it in a look-up table for later use. To process signal, we will just have to add the buffer content to it.

```
add_white_noise_buffer (float val_arr [],const float noise_arr [],long nbr_spl)
{
   for (long pos = 0; pos < nbr_spl; ++pos)
   {
      val_arr [pos] += noise_arr [pos];
   }
}
```

The proposed method is block-based, but it is possible to process data sample per sample. The small size of the buffer and thus the fact it will be repeated often is not a concern. Again, the noise quality is not the most important point. The real problem may come from the cache pollution generated by the buffer reading. Keeping it small should get rid of that.

+ Fast.
  Fills the spectrum almost uniformly.

Propagates to next stages.

– Pollutes the cache memory.
Alters the smallest numbers by quantifying them.

3.2.3 Constant value addition

A uniform noise is not always required to fix the algorithm. Thus, if the signal passes through elements preserving the low frequencies, it is possible to add only a constant (DC). The operation is pretty similar to `kill_denormal_by_-quantization()`:

```
void  add_dc (float &val)
{
   static const float   anti_denormal = 1e-20;
   val += anti_denormal;
}
```

In a feedback loop, this process has an integration effect. But you may use it, even if the feedback gain is high (but less or equal to 1). Indeed, let us say that the tolerated noise should be below –200 dB (that is 10^{-10}). Then it would require about 10 billion operations to reach this value with a feedback gain of exactly 1. Of course if the processed data are not null, the `anti_denormal` constant will be absorbed.

\+ Fast.
Propagates to next stages.

– Cancelled by DC-blockers or high-pass filters.
Alters the smallest numbers by quantifying them.

3.2.4 Constant value + Nyquist frequency addition

This method is a variation of the function described above. The point is to use `add_dc()` only every other time. Thus signal is added to the sequence +A, 0, +A, 0... It contains Nyquist frequency (half the sample rate) and a constant offset. However calculation reduction is balanced by the increased complexity of the algorithm, whose loops require to be unrolled to fix efficiently one sample on two without any branch.

\+ Fast.
Propagates to next stages.

– Cancelled by band-pass filters.
It does not kill every denormal number, but prevents them to appear in feedback loops.
Requires to unroll loops to avoid branch.
Alters the smallest numbers by quantifying them.

3.2.5 Alternatives and efficient solutions

The methods below are derived from the previous ones and are close to perfection. They fit in numerous common cases and are very fast. They are based on the addition of two alternative values to the signal. As soon as the added value changes, the denormal apparition is prevented in or for future feedback loops, for a limited number of samples. Value alternations just have to be frequent enough to prevent most of the denormal apparitions.

Spectrum of the generated noise is quite rich. Moreover, by choosing 0 for one of the values, one can create a DC offset while reducing calculations

To minimize calculations, one can also call `add_dc()` sporadically, for example one time every 20 samples, or even more. This produces random pulses, the spectral content of which is rich and close to the white noise. However generating this random time component is not always obvious and easy. One may just call `add_dc()` regularly to generate an impulse train. This method also produces a lot of frequencies everywhere in the spectrum. It is especially adapted to block-based processing. Of course block size should be carefully chosen depending on DSP algorithm architecture.

\+ Very fast.
Fills the spectrum almost uniformly.
Propagates to next stages.

– It does not eliminate denormal numbers, but prevents their apparition.
To be maximally efficient, requires block-processing algorithms whose size should be carefully chosen.
Alters the smallest numbers by quantifying them.

4. Conclusion

The various clues presented in this paper should help developers to fight efficiently number denormalization in DSP algorithms. There is no universal miraculous method, but numerous variations on the same theme, each one adapted to a given situation. Explicit and systematic denormalized number elimination is interesting to fix signals which may be “impure”. The preventive method class is more intended to vaccinate the signal to avoid denormal number generation.

5. Acknowledgments

Writing this article has been possible with the contributor help of the MusicDSP mailing list and #musicdsp IRC channel on EFNet servers. Thanks also to Frédérique and Julien Bœuf for their advice, suggestions and rereading.

6. References

[1] David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, Computing Surveys, 1991

[2] Intel Corporation, IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture, 2000

[3] Various contributors from the MusicDSP e-mailing list, Denormal Numbers, http://www.smartelectronix.com/musicdsp/text/other001.txt, 2000