Les nombres dénormalisés dans les applications de traitement du signal en virgule flottante

Laurent de Soras

2005.04.19

web: http://ldesoras.free.fr

ABSTRACT

Les applications de traitement du signal qui tournent sur des ordinateurs personnels ou architectures assimilées sont chaque jour plus nombreuses. Cependant ces processeurs n'ont pas que des avantages dans le domaine du traitement du signal. Cet article technique traite du ralentissement parasite connu sous le nom de "bug des nombres dénormalisés", causé naturellement par le traitement de nombres de magnitude très faible. Des solutions d'éradication et de vaccination du signal sont également proposés.

MOTS-CLÉS

Nombre dénormalisé, traitement du signal, CPU

Copyright 2002-2005 – Laurent de Soras

0. Structure du document

0.1 Table des matières

0.2 Historique des révisions

Version	Date	Modifications
1.0	2002.01.11	Première version publiée (en anglais)
1.1	2005.04.19	Traduction en français, quelques mots sur le SSE

0.3 Glossaire

CPU Central Processing Unit

DSP Digital Signal Processing

FIR Finite Impulse Response (filtre direct)

FPU Floating Point Unit

IIR Infinite Impulse Response (filtre récursif, aussi appelé ARMA)

Copyright 2002-2005 – Laurent de Soras

1. Le codage des nombres en virgule flottante

De façon à pouvoir suivre les explication de cet article, il est important de comprendre clairement comment les nombres en virgule flottante sont codés dans la norme IEEE [1]. Il y a trois formats, correspondant à trois niveaux de précision :

- 32 bits, simple précision
- 64 bits, double précision
- 80 bits, double précision étendue

Les deux premiers sont les plus utilisés en traitement numérique du signal. On favorise généralement les nombres en 32 bits parce qu'ils consomment moins de bande passante que les 64 bits et sont généralement suffisant en terme de précision. Dans les exemples qui vont suivre, nous utiliserons aussi ce format. Cependant les algorithmes décrits peuvent facilement être adaptés aux autres formats de nombre en virgule flottante.

1.1 Mode normal

Le mode normal est utilisé pour couvrir la plus grande partie du domaine de représentation des nombres en virgule flottante. Cette représentation divise la donnée en trois champs de bits : le signe, la mantisse et l'exposant biaisé. Les deux derniers champs sont traités en interne comme des entiers positifs. Le signe est constitué d'un seul bit, qui est mis à 1 pour signaler un nombre négatif — c'est un fonctionnement différent du complément à 2 utilisé pour les nombres entiers.

Bit	31/63/79		0
Signification	S	E	M

La valeur du nombre est donnée par la formule suivante :

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

Où :

- B est le biais de l'exposant, un nombre positif dépendant de la précision. Il est égal à 127 dans le cas des nombres en simple précision. B est utilisé pour produire des exposants négatifs car E est codé comme un nombre positif.
- P est la résolution de la mantisse, exprimée en bits.

Bien sûr, il y a des cas particuliers dans lesquels le codage est différent. Par exemple, 0 est codé par tous les bits à 0 sauf peut-être le signe. Un autre exemple est le codage des nombres dénormalisés. N.B. : les nombres en précision double étendue ont le 1 de la formule explicitement codé dans la mantisse.

1.2 Mode dénormalisé

Certains nombres sont si petits qu'ils ne peuvent être codés dans le mode normal. Le mode denormalisé permet d'accroître la plage de représentation en faisant un compromis sur la

Copyright 2002-2005 – Laurent de Soras

résolution. L'exposant biaisé est alors fixé à 0. Ainsi la valeur peut être calculée selon cette seconde formule :

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

Le « 1 » implicite est maintenant codé explicitement de façon à pouvoir atteindre les plus petits nombres en mettant à 0 les bits de poids fort de la mantisse. Bien sûr la résolution décroît avec le nombre de bits significatifs utilisés. L'exposant est augmenté de 1 de façon à assurer la continuité des représentation avec le mode normal.

Le temps de traitement est le problème majeur avec les nombres dénormalisés. Il est beaucoup plus important qu'avec les nombres normaux. Par exemple sur un processeur AMD Thunderbird, une multiplication avec une opérande dénormalisée prend autour de 170 cycles d'horloge, ce qui est plus de 30 fois plus lent qu'avec des opérandes normales ! Quand presque tous les nombres traités sont dénormalisés, la charge CPU augmente dramatiquement et peut compromettre la stabilité des applications temps-réel, même sur les systèmes les plus rapides.

2. La dénormalisation dans les algorithmes DSP

L'effet le plus pervers de la dénormalisation vient de la structure des algorithmes DSP. Le signal suit généralement une chaîne de blocs élémentaires de traitement. Les nombres dénormalisés générés par un des éléments passent à travers les blocs situés en aval, ralentissant les calculs à chaque fois.

La dénormalisation est souvent casée par les structures utilisant la réaction (feedback). Les filtres IIR sont les plus problématiques. Dans de nombreuses applications ils sont préférés aux FIR pour leur coût avantageux en ressources, malgré une réponse en phase de moindre qualité.

2.1 Le filtre 1-pôle

Pour simplifier l'analyse, commençons avec le plus simple élément récursif, le filtre 1-pôle. Dans le domaine z, son équation est la suivante :

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

Nous pouvons en déduire son équation récursive :

$$y[k] = x[k] + ay[k-1] \quad (4)$$

Pour des raisons de stabilité, nous assumons toujours que $|a| < 1$. Quand l'entrée est un signal non nul, il n'y a habituellement pas de problème. Par contre, dès que l'entrée devient et reste nulle, $|y[k]|$ converge exponentiellement vers 0. La vitesse de convergence dépend de la constante a. Invariablement, la sortie devient de plus en plus petite et finit en nombre dénormalisé avant d'atteindre 0.

Théoriquement, après avoir été dénormalisée, la sortie atteint 0 et reste nulle. Cependant, la règle d'arrondi « au plus près » utilisée habituellement par la FPU a ici un effet pervers. Quand $y[k]$ atteint les plus petits nombres représentables, le résultat de la multiplication avec a peut rester constant. Par exemple 4 multiplié par 0.9 et arrondi au plus près donne encore 4. Ainsi les valeurs de $|a| > \frac{1}{2}$ donneront une sortie constante passé un certain seuil. La sortie reste alors dénormalisée jusqu'à ce que l'entrée redevienne non nulle.

Nous pouvons d'ores et déjà pointer deux aspects du problème :

- La sortie devenant dénormalisée à cause d'un changement du signal d'entrée.

- La stabilité de cet état, bien pire pour les performance générale.

2.2 Extension aux éléments récursifs généraux

L'implémentation des filtres IIR peut être divisée en deux parties cascadées en série :

- Partie directe (zéros): $u[k]=\sum_{i=0}^{N} b_i x[k-i]$ (5)
- Partie récursive (pôles): $y[k]=u[k]-\sum_{i=0}^{N} a_i y[k-i]$ (6)

La partie directe peut être vue comme un filtre FIR. De tels filtres donnent rarement des nombres dénormalisés, la magnitude des coefficients restant relativement élevée sur une échelle logarithmique, plusieurs ordre de grandeur au-dessus du seuil de dénormalisation. De plus, si pour une raison quelconque un produit est anormalement petit, il se fait absorber par les autres dans la somme à cause de la résolution limitée de la mantisse. En effet, avec $|x| \gg |y|$, $x+y=x$. En dépit de cela, des filtres capable de complètement annuler un signal devraient être manipulés avec précaution.

C'est la partie récursive qui est principalement responsable de la dénormalisation. Quand $u[k]$ devient et reste nul, on peut voir apparaître un cas similaire à celui du filtre 1-pôle étudié précédemment. Cela ne veut pas toujours dire que l'entrée du filtre $x[k]$ est nulle, parce que cette entrée peut être auparavant complètement annulée par la partie directe. C'est souvent le cas avec les filtres passe-haut recevant un signal constant en entrée. Évidemment, on peut inverser les parties directes et récursives ou changer la structure d'implémentation du filtre si c'est possible. Mais l'important est de se rappeler que si on aboutit quelque part à une entrée nulle dans un circuit récursif, on retombe dans l'exemple du filtre 1-pôle.

Nous avons vu comment l'état dénormalisé pouvait apparaître. Sa stabilité est plus difficile à comprendre. Elle dépend de plusieurs paramètres :

- Les valeurs des coefficients multiplicateurs.
- Le mode d'arrondi de la FPU. Il est généralement fixé « au plus près », ce qui augmente la stabilité de l'état dénormalisé.
- L'ordre des calculs intermédiaires. Dans certains cas, deux termes peuvent s'annuler l'un l'autre au cours d'une somme.
- La résolution des calculs intermédiaires. Les valeurs finales peuvent changer en fonction de la précision de calcul et de stockage.

Malheureusement, il n'y a à notre connaissance pas de formule miracle pour savoir si un algorithme est stable ou pas du point de vue des nombres dénormalisés. Il est nécessaire d'analyser consciencieusement chaque implémentation d'algorithme en fonction des entrées tout aussi bien typiques qu'exceptionnelles. Cependant les critères donnés précédemment devraient aider à effectuer une telle analyse.

Une autre chose importante à se rappeler est que ces comportements ne sont pas spécifiques aux filtres. N'importe quel système récursif peut être sujet à ce genre de problème. Par exemple, c'est le cas des lignes à retard à réinjection.

3. Solutions

Nous allons examiner deux types de solution. Le premier, l'éradication, se focalise sur l'élimination des nombres dénormalisés aussitôt qu'ils sont détectés. Le second, la vaccination, consiste à modifier le signal de façons à éviter qu'il se dénormalise. Dans tous les cas, il est

plus important d'intervenir sur le chemin récursif que le chemin direct. Le second correspondrait plus à une dissimulation du symptôme qu'à un réel traitement.

La suite de cet article présente les méthode d'évitement des nuisances des nombres dénormalisés. La plupart d'entre elles assume que les données utiles sont plusieurs ordre de grandeur au-dessus du seuil de dénormalsation, ce qui est généralement le cas. Par exemple un nombre dont la valeur absolue est en-dessous de 10^{-20} peut être considéré comme nul.

3.1 Élimination explicite des valeurs dénormalisées

3.1.1 Détection des nombres dénormalisés

La première idée est de remplacer tous les nombres dénormalisés par de vrais 0. Elle implique de détecter ces nombres. Comme mentionné précédemment, on reconnaît un nombre dénormalisé par son exposant biaisé nul et sa mantisse non nulle. Cette détection peut être effectuée en testant directement les bits des données. La fonction C++ ci-dessous teste un nombre 32 bit (simple précision) et le met à 0 si nécessaire. La première ligne est utilisée pour traiter la valeurs en virgule flottante comme un entier pour pouvoir accéder aux bits de codage. Cette technique est très pratique et sera réutilisées plus tard dans ce document.

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

On peut bien sûr optimiser cette fonction, mais elle a été ici écrite dans un but principalement pédagogique. L'important était de la rendre claire et facile à lire afin d'en comprendre le concept. Une des optimisation possible est de retirer le test de la mantisse, une mantisse nulle donnant déjà un zéro. L'unique test restant peut être utilisé par un « conditonal move », ce type d'instruction ne cassant pas le pipeline d'instruction en cas de mauvaise prédiction du résultat par le processeur.

+	Remplace les nombres dénormalisés par de véritables 0.
	N'altère pas les nombre normaux (le signal utile)
—	Le pipeline d'instruction peut être rompu, ralentissant le code.
	Nécessite le stockage des nombres en mémoire pour pouvoir accéder à leurs bits.
	Doit être répété à tous les étages du traitement.

Certains processeurs possèdent des jeux d'instruction spécifiques vectorisés (SIMD), intéressants pour le traitement du signal, comme le SSE par exemple. Dans ce jeu particulier, en fixant le flag *Denormal Are Zeros* (DAZ) du registre MXCSR, on autorise le processeur à traiter automatiquement tout nombre dénormalisé comme étant nul. Dans ce cas, cette solution est probablement la meilleure de toutes. Cependant elle ne marque que sur le jeu d'instruction SSE.

Copyright 2002-2005 – Laurent de Soras

3.1.2 Élimination par quantification

La fonction suivante élimine les nombres dénormalisés sans aucun test. L'inconvénient est une légère perte de résolution pour les nombres les plus petits ; cependant c'est usuellement sans conséquence.

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

Le principe repose sur la résolution limitée de la mantisse. Quand un nombre dénormal est ajouté à la constante `anti_denormal`, il est absorbé parce qu'il est trop petit et ne peut pas être codé dans la mantisse finale. La soustraction suivante de la constante aboutit à un zéro réel. Avec un nombre de magnitude beaucoup plus importante (signal utile), c'est la constante qui est absorbée, les deux opération n'ont alors aucun effet. Il faut tout de même signaler que la première opération peut ralentir le code si elle est effectuée avec un nombre dénormalisé. Cependant ce type d'opération a un autre rôle utile, comme nous le verrons plus tard.

En fonction des cas, la constante devra être changée. Par exemple si toutes les opérations sont faites au sein des registres de la FPU sans stockage intermédiaire en mémoire, il est possible que l'arithmétique se fasse sur 80 bits donc avec une mantisse de 64 bits. La valeur `anti_denormal` devra donc être suffisamment élevée. Par contre, si tout est calculé avec une précision de 32 bits, on peut abaisser la constante `anti_denormal` afin d'obtenir la meilleure résolution possible pour le calcul.

+	Remplace les nombres dénormalisés par de véritables 0.
	La quantification a un effet préventif sur les nombres en sortie.
	Relativement rapide, seulement deux additions.
—	Traiter des nombres déjà dénormalisés ralentit le calcul.
	Les plus petits nombres sont quantifiés, ajoutant du bruit dans le signal.
	Doit être répété à chaque étage du traitement.

3.1.3 Détection complémentaire

Une autre façon de détecter les nombres dénormalisés est d'utiliser les drapeaux d'état de la FPU. En effet, le processeur peut lever une exception (heureusement masquée par défaut) à chaque opération dont une des opérande est dénormalisée. Sur les 80x86, un drapeau aide à garder la trace de cette exception et reste levé tant qu'il n'a pas été explicitement effacé. Ainsi, une vérification régulière de ce drapeau permet de s'assurer qu'il n'y a pas de nombre dénormalisé dans le circuit. Après avoir détecté un nombre dénormalisé, on peut le chasser à l'aide de la fonction listée ci-dessous. Cependant cela demande un peu d'assembleur. Voici un exemple compilable sur 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);
}
```

Mais sous des airs de simplicité, cette fonction reste relativement lente car la lecture du statut de la FPU peut prendre un certain temps.

3.2 Prévenir la dénormalisation

La tâche consiste principalement à ajouter du bruit au signal, avant ou durant le traitement. The bruit doit être suffisamment faible pour ne pas altérer les données, mais doit être suffisamment élevé pour tirer les nombres dénormalisés au-dessus du seuil fatal quand il est ajouté au signal.

Le grand bénéfice du bruit additif est sa propagation aux étages suivant de la chaîne de traitement. Mis à part certains cas spéciaux, le procédé permet de remédier au problème au début de la chaîne et de ne plus s'en soucier ensuite.

Il y a différentes manières de générer du bruit, nous allons décrire certaines d'entre elles. Comme d'habitude, une méthode parfaite et universelle n'existe pas : chaque solution doit être choisie avec discernement et adaptée au contexte. En effet, le spectre du bruit revêt une certaine importance, les filtres rencontrés par le signal ne devant pas l'annuler, ce qui le rendrait inutile.

3.2.1 Addition de bruit blanc

La première méthode que nous verrons est l'addition de bruit blanc. Ce bruit a un contenu spectral uniforme, ce qui est le rend efficace dans de nombreuses situations. Il permet en quelque sorte d'oublier le problème des nombres dénormalisés dans les étapes suivantes du traitement. Comme nous n'avons pas besoin d'un bruit blanc parfait, la qualité de ce bruit n'étant pas notre principal but, nous nous concentrerons sur la rapidité de la synthèse.

```
unsigned int  rand_state = 1; // Nous avons besoins d'int 32-bits

void add_white_noise (float &val)
{
   rand_state = rand_state * 1234567UL + 890123UL;
   int   mantissa = rand_state & 0x807F0000; // Bits significatifs uniquement
   int   flt_rnd = mantissa | 0x1E000000;    // Fixe l'exposant
   val += *reinterpret_cast <const float *> (&flt_rnd);
}
```

Ce code génère des nombres aléatoire dont la magnitude est environ 10^{-20}. Le niveau peut être ajusté en changeant la constante dans la ligne fixant l'exposant. Attention : comme le spectre est uniforme, la composante DC est très faible. Cela peut représenter un problème dans certains cas. Nous pouvons le contourner en ne générant que des nombres positifs. Pour cela, modifier la seconde ligne avec `rand_state & 0x007F0000`.

+	Remplit le spectre uniformément. Se propage aux étages suivants.
—	Pas si rapide. Nécessite le stockage des nombres en mémoire pour pouvoir accéder à leurs bits. Les plus petits nombres sont quantifiés.

3.2.2 Addition de bruit — version stockée

Dans la fonction `add_white_noise()`, une partie significative du pourcentage de CPU est pris par le calcul du bruit. Une technique commune consiste à pré-calculer du bruit sur une courte période une fois pour toutes et le stocker dans un buffer pour usage ultérieur. Pour traiter le signal, nous n'avons qu'à ajouter le buffer au signal.

```
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];
    }
}
```

La méthode proposée est basée sur un traitement en bloc (plusieurs échantillons temporels à la fois). Il est également possible d'appliquer la méthode échantillon par échantillon. La courte taille du buffer et ainsi le fait qu'il soit répété souvent n'a pas à nous inquiéter. Encore une fois, la qualité du bruit généré n'est pas important. Le vrai problème que nous pourrions avoir est la pollution du cache générée par la lecture du buffer. Une petite taille de buffer devrait permettre de d'affranchir de ce soucis.

+	Rapide. Remplit le spectre quasiment uniformément. Se propage aux étages suivants.
—	Pollue la mémoire cache Les plus petits nombres sont quantifiés.

3.2.3 Addition de valeur constante

L'uniformité spectrale du bruit pas toujours requise pour protéger l'algorithme DSP. Ainsi, si le signal passe à travers des éléments préservant les basses fréquences, il est possible de n'ajouter qu'une constante au signal (DC). L'opération est assez similaire à la fonction `kill_denormal_by_quantization()` :

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

Dans une boucle à réaction positive (algorithme récursif), ce processus a un effet d'intégration de constante, ce qui amène théoriquement le circuit à la divergence. Il est cependant possible de l'utiliser même si le gain de réaction est élevé (mais inférieur à 1). En effet, disons que nous tolérons un bruit d'amplitude inférieure à –200 dB (10^{-10} en linéaire). Cela demanderait environ 10 milliards d'opération pour atteindre cette valeur avec un gain de réaction unitaire. Et bien sûr si les données traitées sont non-nulles, la constante `anti_denormal` sera absorbée.

+	Rapide. Se propage aux étages suivants.
—	Annulé par les anti-DC et autres filtres passe-hauts. Les plus petits nombres sont quantifiés.

3.2.4 Addition de valeur constant et fréquence de Nyquist

Cette méthode est une variation de la fonction décrite ci-dessus. L'astuce est de n'utiliser `add_dc()` qu'un échantillon sur deux. Ainsi le signal ajouté correspond à la séquence +A, 0, +A, 0... Elle contient la fréquence de Nyquist (la moitié de la fréquence d'échantillonnage) et une valeur constante. Cependant la réduction des calculs est compensée par la complexité accrue de l'algorithme (test). Un déroulement de boucle peut corriger l'affaire dans le cas d'un traitement par bloc.

+	Rapide. Se propage aux étages suivants.

— Annulé par les filtres passe-bande

N'élimine pas tous les nombres dénormalisés mais les empêche d'apparaître dans les boucles à réaction.

Plus complexe que le simple `add_dc()`.

Les plus petits nombres sont quantifiés.

3.2.5 Solutions alternatives efficaces

Les méthodes présentées ci-dessous sont dérivées des précédentes et sont proches de la perfection (hé oui), particulièrement dans le cas de traitement par bloc. Elles sont adaptées à la plupart des cas et leur exécution est très rapide. Elles sont basées sur l'addition au signal d'une valeur qui change régulièrement. Dès que cette valeur change, l'apparition de nombre dénormalisés est repoussée dans les boucles de réaction pour un certain nombre d'échantillons. Les changements de la valeur doivent simplement être suffisamment fréquents pour éviter la plupart des apparitions de nombres dénormalisés.

Le spectre du bruit généré est assez riche. De plus, en choisissant 0 pour l'une des deux valeurs, on peut créer une composante continue tout en réduisant les calculs.

Pour minimiser les calculs, on peut aussi appeler `add_dc()` sporadiquement, par exemple une fois tous les 20 échantillons en moyenne, voire plus souvent. Cela produit des impulsions arrivant plus ou moins aléatoirement, dont le contenu spectral est proche du bruit blanc. Cependant générer cette composante temporelle aléatoire n'est pas toujours évident et facile. On peut se contenter d'appeler `add_dc()` régulièrement pour générer un train d'impulsions. Cette méthode, idéale pour le traitement par bloc, produit un bruit qui n'est plus blanc mais qui conserve un grand nombre de composantes réparties régulièrement sur tout le spectre. Bien sûr la taille du bloc doit être choisie en fonction de l'architecture de l'algorithme vacciné.

\+ Très rapide

Remplit le spectre à peu près uniformément.

Se propage aux étages suivants.

— N'élimine pas tous les nombres dénormalisés mais les empêche d'apparaître dans les boucles à réaction.

L'efficacité maximale n'est atteinte que dans le cas d'un traitement par bloc.

Les plus petits nombres sont quantifiés.

4. Conclusion

Les pistes diverses présentées dans cet article devraient aider les développeureuses à combattre efficacement le pullulement des nombres dénormalisés dans les algorithmes DSP. Il n'y a pas de recette miracle, mais d'innombrables variations sur un même thème, chacune adaptée à une situation particulière. L'éradication explicite et systématique des nombres dénormalisés est intéressante pour guérir des signaux qui pourraient se présenter malades à l'entrée de la chaîne de traitement. Au sein même de la chaîne, on préférera la vaccination pour éviter l'apparition des nombres dénormalisés.

Copyright 2002-2005 – Laurent de Soras

5. Remerciements

L'écriture de cet article a été rendue possible par les participant-e-s de la mailing list MusicDSP et du canal IRC #musicdsp sur les serveurs EFNet. Nous tenons également à remercier Frédérique and Julien Bœuf pour leurs conseils, suggestions et relectures.

6. Références

[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] Diverses contributions publiées sur la mailing list MusicDSP, http://www.smartelectronix.com/musicdsp/text/other001.txt , 2000

Copyright 2002-2005 – Laurent de Soras