Arrondis Rapides de Nombres en Virgule Flottante en C/C++ sur la plate-forme Wintel

Laurent de Soras

Mis à jour le 2008.03.08

web: http://ldesoras.free.fr

ABSTRACT

Ce document est un article technique traitant des aspects pratiques de la programmation dans les langages informatiques C/C++. L'arrondi numérique est une opération mathématique assez courante qui semble relativement triviale au premier coup d'oeil. Cependant les spécifications du traitement des nombres en virgules flottante par l'architecture x86 et par la norme C++ rendent l'opération plus lente qu'on pourrait s'y attendre pour un tel degré de complexité. Le ralentissement peut être très pénalisant, voire critique, pour des algorithmes utilisant intensivement ces opérations. Dans cet article, nous allons décrire le processus d'arrondi et les causes du ralentissement. Puis nous proposerons et analyserons divers compromis entre précision et vitesse d'exécution.

MOTS-CLÉS

Arrondi, trunc, round, floor, ceil, C/C++

0. Structure du document

0.1 Table des matières

0.2 Historique des révisions

Version	Date	Modifications
–	2003.10.19	Création
1.0	2004.07.04	Publication
1.1	2005.11.27	Erreur corrigée dans le code source de `truncate_int()`
1.2	2006.06.17	Erreurs mineures dans le code source
1.3	2008.03.08	Solution pour utiliser la totalité des 32 bits des nombres arrondis

0.3 Glossaire

CPU	Central Processing Unit
FPU	Floating Point Unit
IEEE	Institute of Electrical and Electronics Engineers
WINTEL	Système d'exploitation Microsoft Windows tournant sur une architecture Intel x86.
x86	Famille de processeurs de la firme Intel

1. Pourquoi les instructions d'arrondi prennent une éternité en C/C++ ?

En langage C, quand on convertit un nombre en virgule flottante en entier, l'arrondi suit une règle précise. Ce procédé est appelé *arrondi vers 0*, et consiste à retirer tout ce qui est derrière la virgule décimale (troncature). Le compilateur y parvient en fixant le mode d'arrondi correct sur la FPU, en tronquant le nombre par utilisation des instructions assembleur `fld` et `fistp`, et en restaurant le mode d'arrondi initial de la FPU. Les instructions `floor()` et `ceil()` marchent d'une façon très similaire, mis à part que le nombre obtenu est toujours dans une représentation en virgule flottante, et doit être converti à nouveau si un vrai nombre entier est désiré.

Voici le code de la fonction `__ftol()`, incluse automatiquement par le compilateur Microsoft Visual C++ 6.0 compiler pour effectuer la conversion par troncature :

```
push     ebp                     ; Gestion de la pile
mov      ebp, esp                ;
add      esp, -12                ;
wait
fnstcw   word ptr [ebp-2]        ; Récupère le FPU Ctrl Word
wait
mov      ax, word ptr [ebp-2]
or       ah, 12                  ; Modifie le CW pour fixer le mode
mov      word ptr [ebp-4], ax
fldcw    word ptr [ebp-4]        ; Effectue le chgt. de mode d'arrondi
fistp    qword ptr [ebp-12]      ; Conversion en entier (arrondi)
fldcw    word ptr [ebp-2]        ; Restaure l'ancien mode d'arrondi
mov      eax, dword ptr [ebp-12] ; Récupère les 32 bits de poids faible
mov      edx, dword ptr [ebp-8]  ; Récupère les 32 bits de poids fort
leave                            ; Gestion de la pile
ret
```

Il y a plusieurs défauts ici :

- La fonction n'est pas `inline` et effectue une gestion de pile qui peut se révéler inutile selon les utilisations.
- C'est une fonction générique qui renvoie un entier de 64 bits. Si nous n'avons besoin que de 32 bits, c'est une perte de temps.
- L'instruction `fistp` s'exécute très rapidement, mais c'est tout le contraire pour le changement de mode d'arrondi. En effet, lire ou modifier le FPU Control Word vide le pipeline d'instruction. C'est une cause de ralentissement majeur sur un CPU moderne si l'instruction est répétée de nombreuses fois dans une boucle très courte.

2. La méthode la plus rapide

Nous allons décrire dans cette partie deux solutions très rapide donnant un résultat relativement exact. La méthode la plus rapide des deux dépend complètement de l'application finale, du contexte et de la précision désirée.

2.1 FILD/FISTP

Le morceau de code vu précédemment ne faisait absolument aucune supposition sur le mode d'arrondi *courant* de la FPU. C'est pourquoi il est nécessaire de changer explicitement le

mode, ou au moins de vérifier qu'il correspond à nos besoins. Mais si nous supposons que la FPU est toujours dans le bon mode d'arrondi – celui que nous voulons utiliser, nous n'avons pas besoin de sauver, changer et restaurer ce mode à chaque fois.

Cependant pour pouvoir faire cette supposition, nous devons avoir un contrôle complet sur le code source de notre programme, de façon à être sûr que le mode d'arrondi est cohérent à chaque endroit dans lequel nous utilisons les fonctions d'arrondi. Si ce mode est changé quelque part, nous devons en tenir compte pour garder les résultats corrects. Heureusement, le FPU Control Word est contextuel à chaque thread d'exécution, ce qui signifie qu'un thread n'interfère pas sur l'autre s'il change le mode de la FPU. Cette considération est importante avec une configuration de type *host/plug-in*, dans laquelle plusieurs programmes indépendants (de sources multiples) coopèrent au sein même processus ou thread.

Donc nous pouvons fixer une fois le mode d'arrondi comme bon nous semble, faire les arrondis plusieurs fois, puis restaurer le mode à la fin des opérations. Quand un programme démarre sous Windows, le mode par défaut est fixé à *arrondi à l'entier le plus proche*. C'est une bonne chose à savoir, car ce mode d'arrondi est très courant. Et en jouant avec le signe du nombre et en ajoutant des constantes, on peut réaliser n'importe quel autre mode d'arrondi à partir de celui-ci – nous verrons plus tard comment accomplir cela.

La fonction suivante arrondit un nombre suivant le mode d'arrondi courant, donc *à l'entier le plus proche*, étant données les conditions citées précédemment.

```
inline int conv_float_to_int (float x)
{
   int      a;
   __asm
   {
      fld     x
      fistp    a
   }

   return (a);
}
```

Nous pouvons aussi surcharger la fonction pour le type `double`, avec exactement le même code.

2.2 Utiliser la mémoire

La seconde solution consiste à utiliser le format de stockage en mémoire des nombres en virgule flottante. Nous pouvons effectuer l'arrondi en manipulant directement les bits de la représentation flottante.

2.2.1 Résumé de la structure des nombres en virgule flottante

Un nombre en virgule flottante IEEE est constitué de trois champs de bits : le signe, l'exposant et la mantisse. Le signe est toujours stocké sur un bit, mais son fonctionnement est différent de la représentation des entiers en *complément à 2*. A 1, il veut juste dire « le reste des bits représente la valeur absolue du nombre, celui-ci étant en fait négatif ».

L'exposant est une puissance de deux qui multiplie la mantisse. Cette puissance est biaisée : une valeur constante lui est ajoutée pour en faire un entier non-signé. Précision (nombre de bits) et biais dépendent de la précision globale du nombre. Pour un `float` en 32 bits, l'exposant est stocké sur 8 bits est son biais B = 127. Pour les nombres 64 bits en double précision, l'exposant fait 11 bits et B = 1023.

La mantisse prend les P bits restants. C'est un entier non-signé. Par soucis d'économie, le bit le plus significatif n'est pas stocké. En effet, il vaut toujours 1. Ainsi nous pouvons établir la formule suivante pour donner la valeur du nombre en fonction des trois parties de son codage :

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

Où M est la valeur de la mantisse, P la précision de la mantisse, E la valeur de l'exposant et B le biais. Cette formule est valide pour les nombres dits *normalisés*. En effet, il y a des cas spécifiques, comme les nombres *dénormalisés*, ou les NaN (Not a Number) dans lesquels la signification des champs de bits est modifiée.

2.2.2 La méthode

Elle consiste à faire apparaître le nombre arrondi dans le champ de la mantisse, sans avoir besoin de masquer les bits ou de leur faire faire des déplacements additionnels. Ainsi, nous pouvons stocker directement en mémoire le nombre en virgule flottante et le lire comme s'il s'agissait d'un nombre entier. Comment pouvons-nous réaliser ce tour de magie ?

La partie la plus difficile est d'ajuster correctement la mantisse. En effet, dans sa représentation naturelle, les bits de la mantisse sont alignés à gauche. Le bit le plus significatif est toujours en position P, quelque soit la magnitude du nombre final. Or, nous voulons aligner la mantisse sur la droite, de façon à se conformer à une représentation en virgule fixe. La solution est d'ajouter un nombre constant C, très grand, plus grand que le nombre à arrondir. Ainsi, le bit le plus significatif est toujours celui de C, il ne dépend pas du nombre en entrée.

Notre nombre peut maintenant être aligné à droite, ses bits étant fixés et leurs emplacements connus dans la mantisse. Mais que contiennent ces bits ? Si C est trop important, les bits de poids faible du nombre en entrée vont être perdus. S'il est trop faible, nous allons récupérer ses bits dans le résultat, ce que nous ne voulons pas. Il faut préciser que durant l'addition, le mode d'arrondi de la FPU détermine la valeur du bit de poids faible de la mantisse. Du coup, nous devons choisir C de façon à ce que le bit 0 du nombre arrondi désiré corresponde avec le bit 0 de la mantisse. Ainsi le nombre sera arrondi en fonction du mode d'arrondi.

La façon la plus simple d'y parvenir est de trouver le nombre à ajouter à 1.0 de façon à obtenir une mantisse pleine de 0, sauf le bit de poids faible, ce qui donne 1 en représentation entière. La valeur absolue de la mantisse tient sur P bits. Nous savons que le bit le plus significatif est implicite, non-stocké. Nous devons donc obtenir le nombre binaire 1000...001 dans P + 1 bits. Ainsi nous choisissions :

$$C = 2^P$$

Maintenant, quel serait le domaine du nombre en entrée ? La borne supérieure dépend évidemment de la résolution de la mantisse. Elle doit donc être inférieure à 2^P . La borne inférieure est 0. En effet, cette méthode ne marche pas avec les nombres négatifs. Pourquoi ? Simplement parce qu'en ajoutant un nombre négatif à C, nous obtenons un nombre sous la forme binaire 0111...xxx. Le bit significatif est à présent à la position P – 1 au lieu de P, décalant notre résultat. Cependant, il y a une solution au problème. Il suffit de choisir C plus grand, contenant toujours des 0 dans ses bits de poids faible, et avec le même bit de poids fort. La solution optimale est la forme binaire 11000...000, parce que nous obtenons le plus grand intervalle d'entrée pour les deux signes. Avec un nombre négatif, le deuxième 1 disparaît mais le premier est toujours là, ce qui donne une mantisse sous la forme 10111...xxx. Donc :

$$C_S = 3 \times 2^{P-1}$$

L'intervalle de validité devient alors $[-2^{P-1} ; 2^{P-1}-1]$.

Dans le code ci-dessous, nous devons diviser C_S en deux parties parce qu'il n'est pas représentable comme nombre entier ; les normes C/C++ ne garantissent que 32 bits pour le type `long`. De plus, les bits que nous devons lire en mémoire sont stockés au début du nombre, à cause du modèle mémoire « Little Endian » de l'architecture x86. Les bits significatifs sont au début et les moins significatifs à la fin. La vérification du domaine du

nombre en entrée est faite en deux temps. Le premier test est grossier parce que nous ne faisons aucune supposition sur le mode d'arrondi courant et utilisé. Le second vérifie que le nombre en entrée et le nombre arrondi sont bien de même signe, ce qui devrait détecter les dépassements mineurs ayant échappé au premier test.

```
inline int conv_float_to_int_mem (double x)
{
   const int      p = 52;
   const double   c_p1 = static_cast <double> (1L << (p / 2));
   const double   c_p2 = static_cast <double> (1L << (p - p / 2));
   const double   c_mul = c_p1 * c_p2;
   assert (x > -0.5 * c_mul - 1);
   assert (x < 0.5 * c_mul);
   assert (x > static_cast <double> (INT_MIN) - 1.0);
   assert (x < static_cast <double> (INT_MAX) + 1.0);

   const double   cs = 1.5 * c_mul;
   x += cs;

   const int      a = *(reinterpret_cast <const int *> (&x));
   assert (a * x >= 0);

   return (a);
}
```

2.3 Raisons de l'échec

Les deux méthodes décrites précédemment sont plutôt rapides. Cependant leurs résultats dépendent du mode d'arrondi, qui est *au plus proche* par défaut. Ainsi nous aurions écrit une fonction d'arrondi *au plus proche* ? Pas vraiment. En fait, le mode par défaut de la FPU est appelé plus exactement *au plus proche entier pair*. Pratiquement, c'est quasiment la même chose. Sauf pour les nombres qui sont exactement à mi-chemin entre deux entiers : $\frac{3}{2}$, $\frac{25}{2}$, $-\frac{7}{2}$, etc. Dans ce cas, la FPU choisit l'entier pair le plus proche, alors que l'arrondi mathématique *au plus proche* qu'on attend choisit le nombre entier immédiatement suppérieur.

Pour mieux visualiser la différence entre les deux modes, voici un exemple :

Nombre	Au plus proche	Au plus proche entier
−2.25	−2	−2
−1.75	−2	−2
−1.50	**−1**	**−2**
−1.25	−1	−1
−0.75	−1	−1
−0.50	0	0
−0.25	0	0
0.25	0	0
0.50	**1**	**0**
0.75	1	1
1.25	1	1

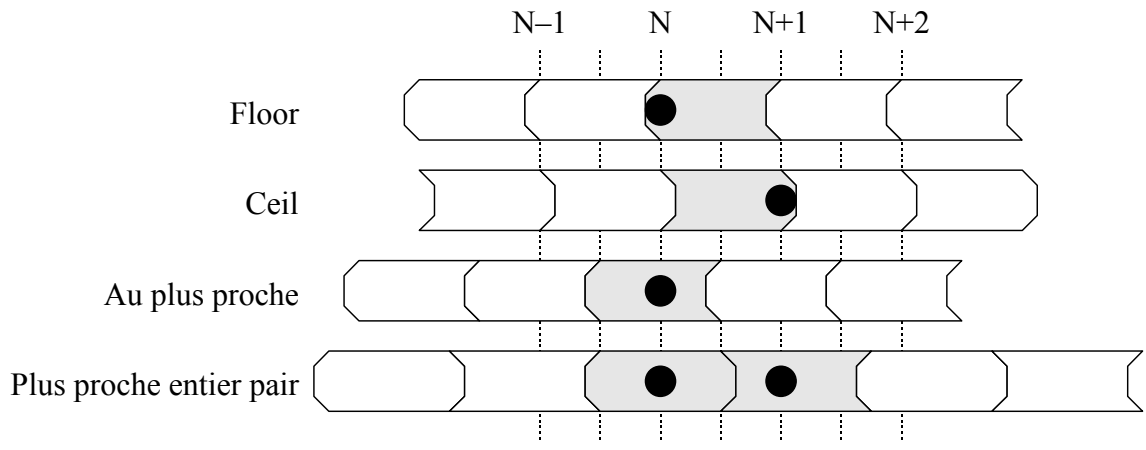
Nombre	Au plus proche	Au plus proche entier
1.50	2	2
1.75	2	2
2.25	2	2

En fonction de l'application, cette particularité peut être critique ou pas. Nous allons exposer une solution pour obtenir un véritable arrondi *au plus proche*.

3. Méthode précise

Le premier concept à comprendre est le modèle d'arrondi pour chaque mode. Ce modèle s'applique à tous les modes, sauf la troncature qui est un peu particulière. Il est caractérisé principalement par une *période* et un *schéma*. Le schéma est comme un motif pour l'arrondi qui peut être répété suivant une certaine période au sein du domaine de fonctionnement. Résumons cela dans un tableau pour tirer les choses au clair.

Mode d'arrondi	Période	Schéma
Vers moins l'infini (`floor`)	1	$[N\ ;\ N+1[\rightarrow N$
Vers plus l'infini (`ceil`)	1	$]N-1\ ;\ N] \rightarrow N$
Au plus proche entier	1	$[N-\frac{1}{2}\ ;\ N+\frac{1}{2}[\rightarrow N$
Au plus proche entier pair	2	$\begin{cases} [2N-\frac{1}{2}\ ;\ 2N+\frac{1}{2}] \rightarrow 2N \\ ]2N+\frac{1}{2}\ ;\ 2N+\frac{3}{2}[\rightarrow 2N+1 \end{cases}$



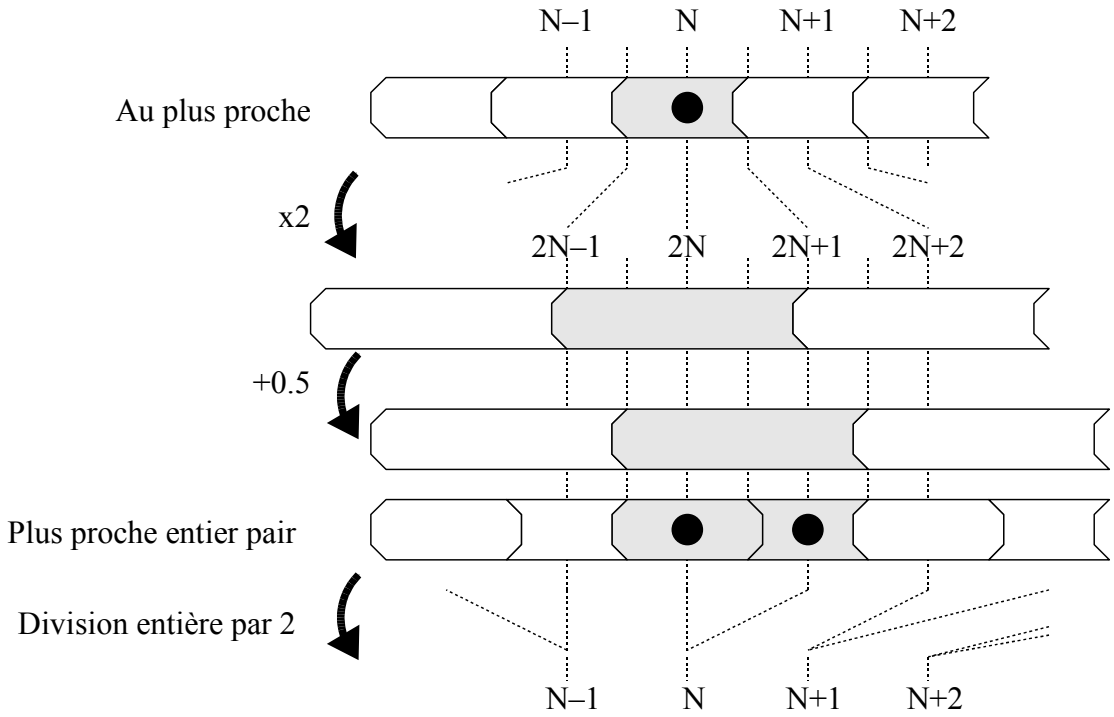
3.1 Concept

Avec le mode *au plus proche entier pair*, nous pouvons effectuer facilement et précisément les autres modes d'arrondi. En effet, nous observons une certaine similarité entre les motifs d'arrondi. Ils ont globalement la même forme. En déplaçant les bornes (offset), en appliquant

des facteurs multiplicateurs et en jouant avec le signe, nous pouvons toujours retomber sur le schéma *au plus proche entier pair* et revenir ensuite au mode original pour extraire le résultat.

3.2 Arrondi à l'entier le plus proche

Nous n'allons détailler que ce mode, les autres peuvent être facilement reconstruits une fois le processus compris et assimilé. Le dessin ci-dessous montre la transformation que nous appliquons au mode original pour atteindre le mode *au plus proche* et enfin récupérer le résultat par transformation inverse.



La dernière étape (transformation inverse) est une division entière, mais nous devons prendre nos précautions. De façon à le faire marcher avec les nombres négatifs, cette division doit être toujours arrondie *vers moins l'inifini*. La division C/C++ utilise la troncature, elle doit donc être évitée dans ce cas. A la place, nous allons utiliser le décalage binaire arithmétique, qui a cette propriété. Seconde écueil à éviter, l'opérateur C/C++ >>, parce que la norme ne garantit pas que le signe des nombres négatif soit préservé. L'utilisation du langage assembleur est alors inévitable si nous voulons garder l'exécution aussi rapide que possible.

```
int round_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_to_nearest = 0.5f;
   int            i;
   __asm
   {
      fld   x
      fadd  st, st (0)
      fadd  round_to_nearest
      fistp i
      sar   i, 1
   }

   return (i);
}
```

La division entière restreint la plage des nombres arrondis de -2^{30} à $2^{30}-1$. Pour palier à ce problème et utiliser la totalité de la plage 32 bits, nous pouvons passer temporairement par l'arithmétique 64 bits :

```
int round_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_to_nearest = 0.5f;
   int            i;
   __int64        tmp;
   __asm
   {
      fld   x
      fadd  st, st (0)
      fadd  round_to_nearest
      fistp qword ptr tmp
      mov   eax, dword ptr tmp
      sar   dword ptr tmp + 4, 1
      rcr   eax, 1
      mov   i, eax
   }

   return (i);
}
```

Cette astuce pourra être réutilisée pour les autres fonctions d'arrondi.

3.3 Arrondi vers moins l'infini (floor)

La seule différence entre `floor` et l'arrondi *au plus proche* est l'offset.

```
int floor_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_towards_m_i = -0.5f;
   int            i;
   __asm
   {
      fld   x
      fadd  st, st (0)
      fadd  round_towards_m_i
      fistp i
      sar   i, 1
   }

   return (i);
```

3.4 Arrondi vers plus l'infini (ceil)

Ce mode est une sorte de miroir du mode `floor`.

```
int ceil_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_towards_p_i = -0.5f;
   int            i;
```

```
   __asm
   {
      fld   x
      fadd  st, st (0)
      fsubr round_towards_p_i
      fistp i
      sar   i, 1
   }

   return (-i);
}
```

3.5 Troncature

La troncature est le mode par défaut des conversions en C/C++. Il consiste à supprimer les nombres après la virgule. On peut considérer qu'il s'agit d'un arrondi *vers moins l'infini* pour les nombres positifs, et *vers plus l'infini* pour les nombres négatifs.

```
int truncate_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_towards_m_i = -0.5f;
   int            i;
   __asm
   {
      fld   x
      fadd  st, st (0)
      fabs
      fadd  round_towards_m_i
      fistp i
      sar   i, 1
   }

   if (x < 0)
   {
      i = -i;
   }

   return (i);
}
```

CONCLUSION

Nous avons passé en revue plusieurs manières d'effectuer rapidement les arrondis, ainsi que les différents défauts et compromis accompagnant ces méthodes. Globalement, elles assument un certain mode d'arrondi du processeur. Il est aussi possible d'étendre cette idée à n'importe quelle combinaison [arrondi désiré / arrondi de la FPU].

