Nombres flottants

Ivan Noyer

Lycée Thiers

1. Présentation
2. La norme IEEE 754
   - Arrondir un nombre
   - Du flottant à sa valeur réelle
3. Pièges et conséquences des approximations numériques
4. Erreurs d'arrondi
5. Arithmétique psychédélique

1 Présentation

2 La norme IEEE 754
- Arrondir un nombre
- Du flottant à sa valeur réelle

3 Pièges et conséquences des approximations numériques

4 Erreurs d'arrondi

5 Arithmétique psychédélique

Pourquoi des nombres à virgule flottante ?

- Avec un nombre fixé de bits, on ne peut représenter qu'un nombre fini de valeurs : représenter tous les réels exactement est impossible.

Pourquoi des nombres à virgule flottante ?

- Avec un nombre fixé de bits, on ne peut représenter qu'un nombre fini de valeurs : représenter tous les réels exactement est impossible.
- Les calculs scientifiques nécessitent de manipuler des valeurs d'ordres de grandeur très différents, par exemple 10^{-20} et 10^{20}.

Pourquoi des nombres à virgule flottante ?

- Avec un nombre fixé de bits, on ne peut représenter qu'un nombre fini de valeurs : représenter tous les réels exactement est impossible.
- Les calculs scientifiques nécessitent de manipuler des valeurs d'ordres de grandeur très différents, par exemple 10^{-20} et 10^{20}.
- Comme en notation scientifique, on représente un nombre par un signe, une partie significative et un exposant :

$$-1234.5 = -1.2345 \times 10^3.$$

L'exposant permet de faire varier la position de la virgule.

Pourquoi des nombres à virgule flottante ?

- Avec un nombre fixé de bits, on ne peut représenter qu'un nombre fini de valeurs : représenter tous les réels exactement est impossible.
- Les calculs scientifiques nécessitent de manipuler des valeurs d'ordres de grandeur très différents, par exemple 10^{-20} et 10^{20}.
- Comme en notation scientifique, on représente un nombre par un signe, une partie significative et un exposant :

$$-1234.5 = -1.2345 \times 10^3.$$

  L'exposant permet de faire varier la position de la virgule.
- À espace de stockage fixé, la virgule flottante permet de couvrir une grande plage de valeurs en conservant une précision relative approximativement constante pour les nombres normalisés.

Le principe en base deux

- Tout réel non nul possède une écriture normalisée

$$x = (-1)^s \times (1.c_1 c_2 c_3 \ldots)_2 \times 2^e,$$

avec $s \in \{0, 1\}$, $e \in \mathbb{Z}$ et $c_i \in \{0, 1\}$. Pour les nombres ayant une écriture binaire finie, on choisit celle prolongée par des zéros.

Le principe en base deux

- Tout réel non nul possède une écriture normalisée

$$x = (-1)^s \times (1.c_1 c_2 c_3 \ldots)_2 \times 2^e,$$

avec $s \in \{0, 1\}$, $e \in \mathbb{Z}$ et $c_i \in \{0, 1\}$. Pour les nombres ayant une écriture binaire finie, on choisit celle prolongée par des zéros.

- Par exemple :

$$13.25 = (1101.01)_2 = (1.10101)_2 \times 2^3.$$

Le principe en base deux

- Tout réel non nul possède une écriture normalisée

$$x = (-1)^s \times (1.c_1 c_2 c_3 \ldots)_2 \times 2^e,$$

avec $s \in \{0, 1\}$, $e \in \mathbb{Z}$ et $c_i \in \{0, 1\}$. Pour les nombres ayant une écriture binaire finie, on choisit celle prolongée par des zéros.

- Par exemple :

$$13.25 = (1101.01)_2 = (1.10101)_2 \times 2^3.$$

- Dans un format flottant, l'exposant est borné et le nombre de bits significatifs est fixé. Pour un nombre normalisé, le premier bit, toujours égal à 1, peut rester implicite.

Le principe en base deux

- Tout réel non nul possède une écriture normalisée

$$x = (-1)^s \times (1.c_1 c_2 c_3 \ldots)_2 \times 2^e,$$

avec $s \in \{0, 1\}$, $e \in \mathbb{Z}$ et $c_i \in \{0, 1\}$. Pour les nombres ayant une écriture binaire finie, on choisit celle prolongée par des zéros.

- Par exemple :

$$13.25 = (1101.01)_2 = (1.10101)_2 \times 2^3.$$

- Dans un format flottant, l'exposant est borné et le nombre de bits significatifs est fixé. Pour un nombre normalisé, le premier bit, toujours égal à 1, peut rester implicite.
- Un nombre dont l'écriture nécessite trop de bits doit être *arrondi* à une valeur représentable. C'est notamment le cas de 0.1, dont l'écriture en base deux est infinie et périodique.

La norme IEEE 754

- IEEE 754 est la norme la plus répandue pour représenter les nombres à virgule flottante et effectuer des calculs sur ces nombres.

La norme IEEE 754

- IEEE 754 est la norme la plus répandue pour représenter les nombres à virgule flottante et effectuer des calculs sur ces nombres.
- Elle définit notamment les formats, les valeurs spéciales (zéros signés, infinis, NaN) et les règles d'arrondi des opérations.

La norme IEEE 754

- IEEE 754 est la norme la plus répandue pour représenter les nombres à virgule flottante et effectuer des calculs sur ces nombres.
- Elle définit notamment les formats, les valeurs spéciales (zéros signés, infinis, NaN) et les règles d'arrondi des opérations.
- Nous étudierons les formats binaires usuels :

Format	Signe	Exposant	Fraction
binary32	1	8	23
binary64	1	11	52

Pour les nombres normalisés, le bit 1 implicite donne respectivement 24 et 53 bits de précision.

La norme IEEE 754

- IEEE 754 est la norme la plus répandue pour représenter les nombres à virgule flottante et effectuer des calculs sur ces nombres.
- Elle définit notamment les formats, les valeurs spéciales (zéros signés, infinis, NaN) et les règles d'arrondi des opérations.
- Nous étudierons les formats binaires usuels :

Format	Signe	Exposant	Fraction
binary32	1	8	23
binary64	1	11	52

  Pour les nombres normalisés, le bit 1 implicite donne respectivement 24 et 53 bits de précision.
- L'arrondi par défaut est l'arrondi au plus proche, avec départage des égalités vers le résultat dont le dernier bit significatif est 0.

La norme IEEE 754

- IEEE 754 est la norme la plus répandue pour représenter les nombres à virgule flottante et effectuer des calculs sur ces nombres.
- Elle définit notamment les formats, les valeurs spéciales (zéros signés, infinis, NaN) et les règles d'arrondi des opérations.
- Nous étudierons les formats binaires usuels :

Format	Signe	Exposant	Fraction
binary32	1	8	23
binary64	1	11	52

  Pour les nombres normalisés, le bit 1 implicite donne respectivement 24 et 53 bits de précision.
- L'arrondi par défaut est l'arrondi au plus proche, avec départage des égalités vers le résultat dont le dernier bit significatif est 0.
- Pour décider de cet arrondi, on peut utiliser trois informations supplémentaires : les bits *guard*, *round* et *sticky* (GRS).

1 Présentation

2 La norme IEEE 754
- Arrondir un nombre
- Du flottant à sa valeur réelle

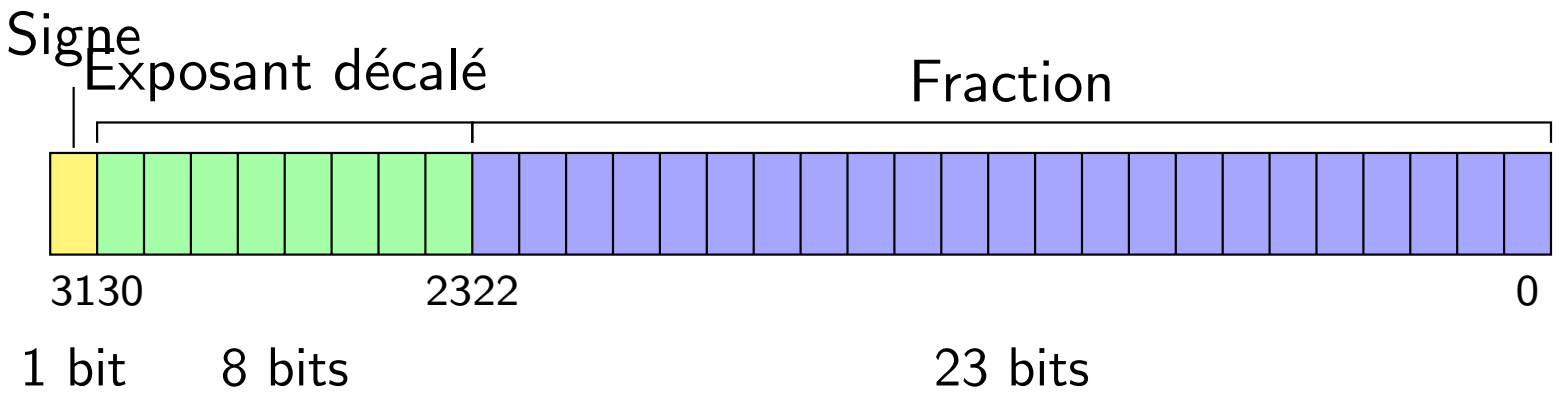
3 Pièges et conséquences des approximations numériques

4 Erreurs d'arrondi

5 Arithmétique psychédélique

Représentation binaire des nombres flottants

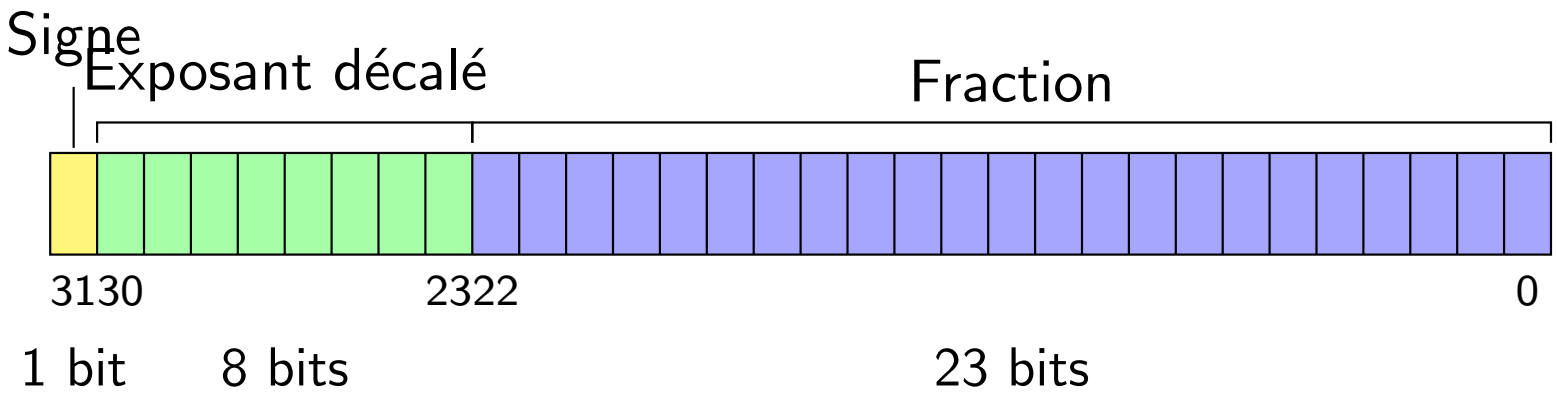
Simple précision : format binary32



- Le format *binary32*, ou simple précision, occupe 32 bits, soit 4 octets.

Représentation binaire des nombres flottants

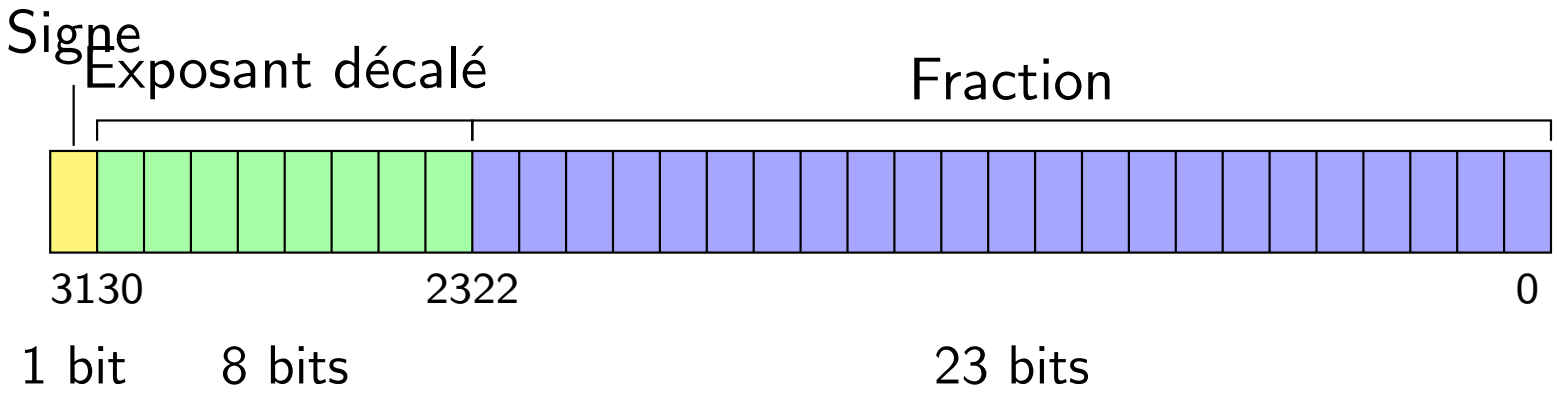
Simple précision : format binary32



- Le format *binary32*, ou simple précision, occupe 32 bits, soit 4 octets.
- Plus généralement, on note $(1, E, M)$ un format comportant un bit de signe, E bits d'exposant décalé et M bits de fraction.

Représentation binaire des nombres flottants

Simple précision : format binary32



- Le format *binary32*, ou simple précision, occupe 32 bits, soit 4 octets.
- Plus généralement, on note $(1, E, M)$ un format comportant un bit de signe, E bits d'exposant décalé et M bits de fraction.
- La fraction est aussi appelée *mantisse stockée*. Il faut la distinguer du significande complet.

Signe et exposant décalé

On considère un format $(1, E, M)$.

- Le bit de signe s vaut 0 pour un nombre positif, 1 pour un nombre négatif.

Signe et exposant décalé

On considère un format $(1, E, M)$.

- Le bit de signe s vaut 0 pour un nombre positif, 1 pour un nombre négatif.
- Le champ exposant est interprété comme un entier *non signé*, noté e_d :

$$0 \leq e_d \leq 2^E - 1.$$

Signe et exposant décalé

On considère un format $(1, E, M)$.

- Le bit de signe s vaut 0 pour un nombre positif, 1 pour un nombre négatif.
- Le champ exposant est interprété comme un entier *non signé*, noté e_d :

$$0 \leq e_d \leq 2^E - 1.$$

- Pour un nombre *normalisé*, l'exposant réel e est obtenu en soustrayant un biais :

$$d = 2^{E-1} - 1, \qquad e = e_d - d.$$

Signe et exposant décalé

On considère un format $(1, E, M)$.

- Le bit de signe s vaut 0 pour un nombre positif, 1 pour un nombre négatif.
- Le champ exposant est interprété comme un entier *non signé*, noté e_d :

$$0 \leq e_d \leq 2^E - 1.$$

- Pour un nombre *normalisé*, l'exposant réel e est obtenu en soustrayant un biais :

$$d = 2^{E-1} - 1, \qquad e = e_d - d.$$

- Les valeurs $e_d = 0$ et $e_d = 2^E - 1$ sont réservées à des cas particuliers.

Signe et exposant décalé

On considère un format $(1, E, M)$.

- Le bit de signe s vaut 0 pour un nombre positif, 1 pour un nombre négatif.
- Le champ exposant est interprété comme un entier *non signé*, noté e_d :

$$0 \leq e_d \leq 2^E - 1.$$

- Pour un nombre *normalisé*, l'exposant réel e est obtenu en soustrayant un biais :

$$d = 2^{E-1} - 1, \qquad e = e_d - d.$$

- Les valeurs $e_d = 0$ et $e_d = 2^E - 1$ sont réservées à des cas particuliers.
- En simple précision :

$$E = 8, \qquad d = 127, \qquad 1 \leq e_d \leq 254, \qquad -126 \leq e \leq 127.$$

Fraction et bit implicite

- Les M bits du champ fraction représentent le nombre

$$m = (0.c_1 c_2 \dots c_M)_2 = \sum_{i=1}^{M} c_i 2^{-i}.$$

Fraction et bit implicite

- Les M bits du champ fraction représentent le nombre

$$m = (0.c_1 c_2 \dots c_M)_2 = \sum_{i=1}^{M} c_i 2^{-i}.$$

- Pour un nombre normalisé, le significande est

$$1 + m = (1.c_1 c_2 \dots c_M)_2.$$

Son premier bit vaut toujours 1 : on ne le stocke pas, il est *implicite*.

Fraction et bit implicite

- Les M bits du champ fraction représentent le nombre

$$m = (0.c_1 c_2 \dots c_M)_2 = \sum_{i=1}^{M} c_i 2^{-i}.$$

- Pour un nombre normalisé, le significande est

$$1 + m = (1.c_1 c_2 \dots c_M)_2.$$

  Son premier bit vaut toujours 1 : on ne le stocke pas, il est *implicite*.
- La valeur représentée est donc

$$\boxed{x = (-1)^s (1 + m) 2^{e_d - d}}.$$

Fraction et bit implicite

- Les M bits du champ fraction représentent le nombre

$$m = (0.c_1 c_2 \dots c_M)_2 = \sum_{i=1}^{M} c_i 2^{-i}.$$

- Pour un nombre normalisé, le significande est

$$1 + m = (1.c_1 c_2 \dots c_M)_2.$$

  Son premier bit vaut toujours 1 : on ne le stocke pas, il est *implicite*.
- La valeur représentée est donc

$$\boxed{x = (-1)^s (1 + m) 2^{e_d - d}}.$$

- En simple précision, les 23 bits de fraction et le bit implicite donnent 24 bits de précision. En double précision, on dispose de 53 bits : 52 stockés et un implicite.

Valeur représentée et affichage décimal

- Une écriture décimale finie n'a pas nécessairement une écriture binaire finie. Par exemple :

$$0.1 = (0.0001100110011\ldots)_2.$$

Ce nombre n'est donc représentable exactement ni en binary32 ni en binary64.

Valeur représentée et affichage décimal

- Une écriture décimale finie n'a pas nécessairement une écriture binaire finie. Par exemple :

$$0.1 = (0.0001100110011\ldots)_2.$$

  Ce nombre n'est donc représentable exactement ni en binary32 ni en binary64.
- En binary32, son arrondi au plus proche vaut exactement

$$\frac{13421773}{134217728} = 0.100000001490116119384765625.$$

Valeur représentée et affichage décimal

- Une écriture décimale finie n'a pas nécessairement une écriture binaire finie. Par exemple :

$$0.1 = (0.0001100110011\ldots)_2.$$

  Ce nombre n'est donc représentable exactement ni en binary32 ni en binary64.
- En binary32, son arrondi au plus proche vaut exactement

$$\frac{13421773}{134217728} = 0.100000001490116119384765625.$$

- L'affichage est une conversion vers une écriture décimale. Le nombre de chiffres affichés dépend du langage et du format demandé.

Valeur représentée et affichage décimal

- Une écriture décimale finie n'a pas nécessairement une écriture binaire finie. Par exemple :

$$0.1 = (0.0001100110011\ldots)_2.$$

  Ce nombre n'est donc représentable exactement ni en binary32 ni en binary64.
- En binary32, son arrondi au plus proche vaut exactement

$$\frac{13421773}{134217728} = 0.100000001490116119384765625.$$

- L'affichage est une conversion vers une écriture décimale. Le nombre de chiffres affichés dépend du langage et du format demandé.
- Un affichage court comme `0.1` ne signifie donc pas que la valeur stockée est exactement 0.1.

Valeur représentée et affichage décimal

- Une écriture décimale finie n'a pas nécessairement une écriture binaire finie. Par exemple :

$$0.1 = (0.0001100110011\ldots)_2.$$

  Ce nombre n'est donc représentable exactement ni en binary32 ni en binary64.
- En binary32, son arrondi au plus proche vaut exactement

$$\frac{13421773}{134217728} = 0.100000001490116119384765625.$$

- L'affichage est une conversion vers une écriture décimale. Le nombre de chiffres affichés dépend du langage et du format demandé.
- Un affichage court comme `0.1` ne signifie donc pas que la valeur stockée est exactement 0.1.
- En revanche, 1.0, 3.75 et 13.25 sont représentables exactement dans ces deux formats.

Interprétation des champs

Pour un format $(1, E, M)$, on pose

$$d = 2^{E-1} - 1, \qquad m = (0.c_1 \dots c_M)_2.$$

Type	Exposant e_d	Fraction m	Valeur
Zéros signés	0	0	$+0$ ou -0
Subnormaux	0	$\neq 0$	$(-1)^s m\, 2^{1-d}$
Normalisés	1 à $2^E - 2$	quelconque	$(-1)^s(1+m)2^{e_d-d}$
Infinis	$2^E - 1$	0	$+\infty$ ou $-\infty$
NaN	$2^E - 1$	$\neq 0$	*Not a Number*

- Les nombres *subnormaux*, aussi appelés *dénormalisés*, permettent de représenter de très petites valeurs non nulles.

Interprétation des champs

Pour un format $(1, E, M)$, on pose

$$d = 2^{E-1} - 1, \qquad m = (0.c_1 \dots c_M)_2.$$

Type	Exposant e_d	Fraction m	Valeur
Zéros signés	0	0	$+0$ ou -0
Subnormaux	0	$\neq 0$	$(-1)^s m\, 2^{1-d}$
Normalisés	1 à $2^E - 2$	quelconque	$(-1)^s(1+m)2^{e_d-d}$
Infinis	$2^E - 1$	0	$+\infty$ ou $-\infty$
NaN	$2^E - 1$	$\neq 0$	*Not a Number*

- Les nombres *subnormaux*, aussi appelés *dénormalisés*, permettent de représenter de très petites valeurs non nulles.
- Les NaN servent notamment à représenter le résultat d'opérations invalides, comme $0/0$.

Pourquoi des nombres subnormaux ?

- Le plus petit nombre normalisé strictement positif est

$$x_{\text{min}} = 2^{1-d}.$$

Pourquoi des nombres subnormaux ?

- Le plus petit nombre normalisé strictement positif est
$$x_{\text{min}} = 2^{1-d}.$$
- Pour représenter des valeurs plus proches de zéro, on conserve cet exposant minimal, mais on remplace le bit implicite 1 par 0 :
$$x = (0.c_1 \dots c_M)_2 \, 2^{1-d}.$$

Pourquoi des nombres subnormaux ?

- Le plus petit nombre normalisé strictement positif est
$$x_{\min} = 2^{1-d}.$$
- Pour représenter des valeurs plus proches de zéro, on conserve cet exposant minimal, mais on remplace le bit implicite 1 par 0 :
$$x = (0.c_1 \dots c_M)_2 \, 2^{1-d}.$$
- Le plus petit subnormal strictement positif vaut
$$2^{-M} \, 2^{1-d} = 2^{1-d-M}.$$

Pourquoi des nombres subnormaux ?

- Le plus petit nombre normalisé strictement positif est
$$x_{\min} = 2^{1-d}.$$
- Pour représenter des valeurs plus proches de zéro, on conserve cet exposant minimal, mais on remplace le bit implicite 1 par 0 :
$$x = (0.c_1 \dots c_M)_2 \, 2^{1-d}.$$
- Le plus petit subnormal strictement positif vaut
$$2^{-M} \, 2^{1-d} = 2^{1-d-M}.$$
- Le plus grand subnormal positif et le plus petit normalisé positif sont deux valeurs consécutives :
$$\underbrace{(1 - 2^{-M})2^{1-d}}_{\text{plus grand subnormal}} \longrightarrow \underbrace{2^{1-d}}_{\text{plus petit normalisé}}.$$

Pourquoi des nombres subnormaux ?

- Le plus petit nombre normalisé strictement positif est
$$x_{\min} = 2^{1-d}.$$
- Pour représenter des valeurs plus proches de zéro, on conserve cet exposant minimal, mais on remplace le bit implicite 1 par 0 :
$$x = (0.c_1 \dots c_M)_2 \, 2^{1-d}.$$
- Le plus petit subnormal strictement positif vaut
$$2^{-M} \, 2^{1-d} = 2^{1-d-M}.$$
- Le plus grand subnormal positif et le plus petit normalisé positif sont deux valeurs consécutives :
$$\underbrace{(1 - 2^{-M})2^{1-d}}_{\text{plus grand subnormal}} \longrightarrow \underbrace{2^{1-d}}_{\text{plus petit normalisé}}.$$
- Les subnormaux assurent ainsi une transition progressive vers zéro, au prix d'une diminution de la précision relative.

Du réel aux champs binaires

Détermination de (s, e, m)

On considère un réel $X \neq 0$ dont la valeur absolue est dans la plage des nombres normalisés.

- Déterminer le signe :

$$s = \begin{cases} 0 & \text{si } X > 0, \\ 1 & \text{si } X < 0. \end{cases}$$

Du réel aux champs binaires

Détermination de (s, e, m)

On considère un réel $X \neq 0$ dont la valeur absolue est dans la plage des nombres normalisés.

- Déterminer le signe :

$$s = \begin{cases} 0 & \text{si } X > 0, \\ 1 & \text{si } X < 0. \end{cases}$$

- Déterminer l'unique entier e tel que

$$1 \leq \frac{|X|}{2^e} < 2, \qquad \text{soit} \qquad 2^e \leq |X| < 2^{e+1}.$$

Du réel aux champs binaires

Détermination de (s, e, m)

On considère un réel $X \neq 0$ dont la valeur absolue est dans la plage des nombres normalisés.

- Déterminer le signe :

$$s = \begin{cases} 0 & \text{si } X > 0, \\ 1 & \text{si } X < 0. \end{cases}$$

- Déterminer l'unique entier e tel que

$$1 \leq \frac{|X|}{2^e} < 2, \qquad \text{soit} \qquad 2^e \leq |X| < 2^{e+1}.$$

- En déduire la fraction m :

$$m = \frac{|X|}{2^e} - 1, \qquad 0 \leq m < 1.$$

On obtient ainsi

$$X = (-1)^s (1 + m) 2^e.$$

Déterminer l'exposant par normalisation

- Partir de $a = |X|$ et de $e = 0$.

Déterminer l'exposant par normalisation

- Partir de $a = |X|$ et de $e = 0$.
- Tant que $a \geq 2$, diviser a par 2 et augmenter e de 1.

Déterminer l'exposant par normalisation

- Partir de $a = |X|$ et de $e = 0$.
- Tant que $a \geq 2$, diviser a par 2 et augmenter e de 1.
- Tant que $a < 1$, multiplier a par 2 et diminuer e de 1.

Déterminer l'exposant par normalisation

- Partir de $a = |X|$ et de $e = 0$.
- Tant que $a \geq 2$, diviser a par 2 et augmenter e de 1.
- Tant que $a < 1$, multiplier a par 2 et diminuer e de 1.
- À la fin :

$$|X| = a\,2^e, \qquad 1 \leq a < 2, \qquad m = a - 1.$$

Déterminer l'exposant par normalisation

- Partir de $a = |X|$ et de $e = 0$.
- Tant que $a \geq 2$, diviser a par 2 et augmenter e de 1.
- Tant que $a < 1$, multiplier a par 2 et diminuer e de 1.
- À la fin :

$$|X| = a\,2^e, \qquad 1 \leq a < 2, \qquad m = a - 1.$$

- Exemple avec $X = -9.6$:

$$9.6 \longrightarrow 4.8 \longrightarrow 2.4 \longrightarrow 1.2.$$

Trois divisions donnent

$$s = 1, \qquad e = 3, \qquad m = 0.2, \qquad -9.6 = -1.2 \times 2^3.$$

Du réel aux champs binaires

Construction du triplet (s_2, e_2, m_2)

- Le champ signe contient directement le bit s :

$$s_2 = s.$$

Du réel aux champs binaires

Construction du triplet (s_2, e_2, m_2)

- Le champ signe contient directement le bit s :

$$s_2 = s.$$

- Le champ exposant contient l'écriture binaire de $e_d = e + d$, sur E bits.
  Pour $X = -9.6$ en binary32 :

$$e_d = 3 + 127 = 130 = (10000010)_2.$$

Du réel aux champs binaires

Construction du triplet (s_2, e_2, m_2)

- Le champ signe contient directement le bit s :

$$s_2 = s.$$

- Le champ exposant contient l'écriture binaire de $e_d = e + d$, sur E bits.
  Pour $X = -9.6$ en binary32 :

$$e_d = 3 + 127 = 130 = (10000010)_2.$$

- Pour obtenir les bits de la fraction, partir de $r = m$, puis répéter :

$$t = 2r, \qquad c_i = \lfloor t \rfloor, \qquad r = t - c_i.$$

  Chaque étape produit un bit $c_i \in \{0, 1\}$.

Du réel aux champs binaires

Construction du triplet (s_2, e_2, m_2)

- Le champ signe contient directement le bit s :

$$s_2 = s.$$

- Le champ exposant contient l'écriture binaire de $e_d = e + d$, sur E bits.
  Pour $X = -9.6$ en binary32 :

$$e_d = 3 + 127 = 130 = (10000010)_2.$$

- Pour obtenir les bits de la fraction, partir de $r = m$, puis répéter :

$$t = 2r, \qquad c_i = \lfloor t \rfloor, \qquad r = t - c_i.$$

  Chaque étape produit un bit $c_i \in \{0, 1\}$.
- Si le reste devient nul, compléter par des zéros. Sinon, l'écriture binaire se poursuit.

Du réel aux champs binaires

Construction du triplet (s_2, e_2, m_2)

- Le champ signe contient directement le bit s :

$$s_2 = s.$$

- Le champ exposant contient l'écriture binaire de $e_d = e + d$, sur E bits.
  Pour $X = -9.6$ en binary32 :

$$e_d = 3 + 127 = 130 = (10000010)_2.$$

- Pour obtenir les bits de la fraction, partir de $r = m$, puis répéter :

$$t = 2r, \qquad c_i = \lfloor t \rfloor, \qquad r = t - c_i.$$

  Chaque étape produit un bit $c_i \in \{0, 1\}$.
- Si le reste devient nul, compléter par des zéros. Sinon, l'écriture binaire se poursuit.
- Le champ m_2 ne comporte que M bits : lorsque l'écriture se poursuit au-delà, il faut déterminer comment l'arrondir.

1 Présentation

2 La norme IEEE 754
- Arrondir un nombre
- Du flottant à sa valeur réelle

3 Pièges et conséquences des approximations numériques

4 Erreurs d'arrondi

5 Arithmétique psychédélique

Quel flottant choisir ?

- Lorsque l'écriture binaire d'un nombre nécessite davantage de bits que le format n'en conserve, il faut choisir une valeur représentable.

Quel flottant choisir ?

- Lorsque l'écriture binaire d'un nombre nécessite davantage de bits que le format n'en conserve, il faut choisir une valeur représentable.
- Nous utilisons l'*arrondi au plus proche* : choisir le flottant le plus proche du nombre exact.

Quel flottant choisir ?

- Lorsque l'écriture binaire d'un nombre nécessite davantage de bits que le format n'en conserve, il faut choisir une valeur représentable.
- Nous utilisons l'*arrondi au plus proche* : choisir le flottant le plus proche du nombre exact.
- Si le nombre exact est à égale distance de deux flottants, choisir celui dont le dernier bit significatif vaut 0. C'est l'arrondi *au plus proche, au pair*.

Quel flottant choisir ?

- Lorsque l'écriture binaire d'un nombre nécessite davantage de bits que le format n'en conserve, il faut choisir une valeur représentable.
- Nous utilisons l'*arrondi au plus proche* : choisir le flottant le plus proche du nombre exact.
- Si le nombre exact est à égale distance de deux flottants, choisir celui dont le dernier bit significatif vaut 0. C'est l'arrondi *au plus proche, au pair*.
- Exemple avec trois bits après le point :

$$(1.0101)_2 = \frac{(1.010)_2 + (1.011)_2}{10_2} \texttt{ avec } 10_2 \texttt{ pour } 2_{10}.$$

Les deux valeurs sont aussi proches : on choisit $(1.010)_2$.

Quel flottant choisir ?

- Lorsque l'écriture binaire d'un nombre nécessite davantage de bits que le format n'en conserve, il faut choisir une valeur représentable.
- Nous utilisons l'*arrondi au plus proche* : choisir le flottant le plus proche du nombre exact.
- Si le nombre exact est à égale distance de deux flottants, choisir celui dont le dernier bit significatif vaut 0. C'est l'arrondi *au plus proche, au pair*.
- Exemple avec trois bits après le point :

$$(1.0101)_2 = \frac{(1.010)_2 + (1.011)_2}{10_2} \texttt{ avec } 10_2 \texttt{ pour } 2_{10}.$$

  Les deux valeurs sont aussi proches : on choisit $(1.010)_2$.
- Pour prendre cette décision, il suffit de connaître le dernier bit conservé et trois informations sur les bits supprimés : *guard*, *round*, *sticky*.

Les bits guard, round et sticky

Pour un nombre normalisé, on écrit le significande exact :

$$1.\underbrace{c_1 c_2 \dots c_M}_{\text{fraction conservée}}\ \underbrace{c_{M+1}}_{G}\ \underbrace{c_{M+2}}_{R}\ \underbrace{c_{M+3} c_{M+4} \dots}_{\text{résumés par } S}.$$

- Le bit de garde, *guard*, est le premier bit supprimé :

$$G = c_{M+1}.$$

Les bits guard, round et sticky

Pour un nombre normalisé, on écrit le significande exact :

$$1.\ \underbrace{c_1 c_2 \dots c_M}_{\text{fraction conservée}}\ \underbrace{c_{M+1}}_{G}\ \underbrace{c_{M+2}}_{R}\ \underbrace{c_{M+3} c_{M+4} \dots}_{\text{résumés par } S}.$$

- Le bit de garde, *guard*, est le premier bit supprimé :

$$G = c_{M+1}.$$

- Le bit d'arrondi, *round*, est le suivant :

$$R = c_{M+2}.$$

Les bits guard, round et sticky

Pour un nombre normalisé, on écrit le significande exact :

$$1.\ \underbrace{c_1 c_2 \dots c_M}_{\text{fraction conservée}}\ \underbrace{c_{M+1}}_{G}\ \underbrace{c_{M+2}}_{R}\ \underbrace{c_{M+3} c_{M+4} \cdots}_{\text{résumés par } S}.$$

- Le bit de garde, *guard*, est le premier bit supprimé :

$$G = c_{M+1}.$$

- Le bit d'arrondi, *round*, est le suivant :

$$R = c_{M+2}.$$

- Le bit *sticky* vaut 1 si au moins un des bits suivants vaut 1 :

$$S = c_{M+3} \vee c_{M+4} \vee \cdots .$$

  Il vaut 0 si tous ces bits sont nuls.

Les bits guard, round et sticky

Pour un nombre normalisé, on écrit le significande exact :

$$1.\ \underbrace{c_1 c_2 \dots c_M}_{\text{fraction conservée}}\ \underbrace{c_{M+1}}_{G}\ \underbrace{c_{M+2}}_{R}\ \underbrace{c_{M+3} c_{M+4} \cdots}_{\text{résumés par } S}.$$

- Le bit de garde, *guard*, est le premier bit supprimé :

$$G = c_{M+1}.$$

- Le bit d'arrondi, *round*, est le suivant :

$$R = c_{M+2}.$$

- Le bit *sticky* vaut 1 si au moins un des bits suivants vaut 1 :

$$S = c_{M+3} \vee c_{M+4} \vee \cdots .$$

  Il vaut 0 si tous ces bits sont nuls.
- Les bits GRS servent à décider de l'arrondi : ils ne font pas partie du flottant finalement stocké.

Bit Sticky

Il n'est pas nécessaire de calculer une infinité de bits pour déterminer S : après le tour $M+2$, le reste r représente la partie de l'écriture binaire qui reste à produire. Ainsi,

$$S = 1 \iff r \neq 0.$$

Si $r = 0$, tous les bits suivants sont nuls ; sinon, au moins l'un d'entre eux vaut 1.

Décider de l'arrondi avec GRS

On note $L = c_M$ le dernier bit conservé et

$$T = (1.c_1 \dots c_M)_2$$

le significande tronqué.

GRS	Partie supprimée	Décision
$0\,xx$	Moins d'une demi-unité	Conserver T
$1\,00$	Exactement une demi-unité	Incrémenter si $L = 1$
$1\,01$	Plus d'une demi-unité	Incrémenter
$1\,1x$	Plus d'une demi-unité	Incrémenter

- Une unité du dernier bit conservé vaut ici 2^{-M}. Incrémenter signifie remplacer T par $T + 2^{-M}$.

Décider de l'arrondi avec GRS

On note $L = c_M$ le dernier bit conservé et

$$T = (1.c_1 \dots c_M)_2$$

le significande tronqué.

GRS	Partie supprimée	Décision
$0\,xx$	Moins d'une demi-unité	Conserver T
$1\,00$	Exactement une demi-unité	Incrémenter si $L = 1$
$1\,01$	Plus d'une demi-unité	Incrémenter
$1\,1x$	Plus d'une demi-unité	Incrémenter

- Une unité du dernier bit conservé vaut ici 2^{-M}. Incrémenter signifie remplacer T par $T + 2^{-M}$.
- La condition d'incrémentation se résume à

$$\boxed{G \wedge (R \vee S \vee L)}.$$

Décider de l'arrondi avec GRS

On note $L = c_M$ le dernier bit conservé et

$$T = (1.c_1 \dots c_M)_2$$

le significande tronqué.

GRS	Partie supprimée	Décision
$0\,xx$	Moins d'une demi-unité	Conserver T
$1\,00$	Exactement une demi-unité	Incrémenter si $L = 1$
$1\,01$	Plus d'une demi-unité	Incrémenter
$1\,1x$	Plus d'une demi-unité	Incrémenter

- Une unité du dernier bit conservé vaut ici 2^{-M}. Incrémenter signifie remplacer T par $T + 2^{-M}$.
- La condition d'incrémentation se résume à

$$\boxed{G \wedge (R \vee S \vee L)}.$$

Du réel au flottant : exemple de -0.6

Normalisation et écriture binaire

- On travaille en simple précision :

$$E = 8, \qquad M = 23, \qquad d = 127.$$

Du réel au flottant : exemple de -0.6

Normalisation et écriture binaire

- On travaille en simple précision :

$$E = 8, \qquad M = 23, \qquad d = 127.$$

- Comme

$$-0.6 = (-1)^1 \times 1.2 \times 2^{-1},$$

on obtient

$$s = 1, \qquad e = -1, \qquad m = 0.2.$$

Du réel au flottant : exemple de -0.6

Normalisation et écriture binaire

- On travaille en simple précision :

$$E = 8, \qquad M = 23, \qquad d = 127.$$

- Comme

$$-0.6 = (-1)^1 \times 1.2 \times 2^{-1},$$

on obtient

$$s = 1, \qquad e = -1, \qquad m = 0.2.$$

- L'exposant décalé vaut

$$e_d = -1 + 127 = 126 = (01111110)_2.$$

Du réel au flottant : exemple de -0.6

Normalisation et écriture binaire

- On travaille en simple précision :

$$E = 8, \qquad M = 23, \qquad d = 127.$$

- Comme

$$-0.6 = (-1)^1 \times 1.2 \times 2^{-1},$$

on obtient

$$s = 1, \qquad e = -1, \qquad m = 0.2.$$

- L'exposant décalé vaut

$$e_d = -1 + 127 = 126 = (01111110)_2.$$

- Pour convertir la fraction $m = 0.2$:

$$\begin{aligned} 0.2 \times 2 &= \mathbf{0} + 0.4 \\ 0.4 \times 2 &= \mathbf{0} + 0.8 \\ 0.8 \times 2 &= \mathbf{1} + 0.6 \\ 0.6 \times 2 &= \mathbf{1} + 0.2 \end{aligned}$$

Du réel au flottant : exemple de -0.6

Arrondi de la fraction sur 23 bits

Le significande exact s'écrit

$$1.\underbrace{0011\,0011\,0011\,0011\,0011\,001}_{\text{23 bits conservés}}\underbrace{1}_{G}\underbrace{0}_{R}\underbrace{0110011\ldots}_{S=1}.$$

- On a $G = 1$, $R = 0$ et $S = 1$: la partie supprimée dépasse une demi-unité du dernier bit conservé.

Du réel au flottant : exemple de -0.6

Arrondi de la fraction sur 23 bits

Le significande exact s'écrit

$$1.\underbrace{0011\,0011\,0011\,0011\,0011\,001}_{\text{23 bits conservés}}\underbrace{1}_{G}\underbrace{0}_{R}\underbrace{0110011\ldots}_{S=1}.$$

- On a $G = 1$, $R = 0$ et $S = 1$: la partie supprimée dépasse une demi-unité du dernier bit conservé.
- On incrémente donc la fraction tronquée :

$$\begin{array}{r} 0011\,0011\,0011\,0011\,0011\,001 \\ +\,1 \\ \hline 0011\,0011\,0011\,0011\,0011\,010 \end{array}$$

Du réel au flottant : exemple de -0.6

Arrondi de la fraction sur 23 bits

Le significande exact s'écrit

$$1.\underbrace{0011\,0011\,0011\,0011\,0011\,001}_{\text{23 bits conservés}}\underbrace{1}_{G}\underbrace{0}_{R}\underbrace{0110011\ldots}_{S=1}.$$

- On a $G = 1$, $R = 0$ et $S = 1$: la partie supprimée dépasse une demi-unité du dernier bit conservé.
- On incrémente donc la fraction tronquée :

$$\begin{array}{r} 0011\,0011\,0011\,0011\,0011\,001 \\ +\,1 \\ \hline 0011\,0011\,0011\,0011\,0011\,010 \end{array}$$

- Le codage final est

$$\boxed{\underbrace{1}_{s_2}\ \underbrace{01111110}_{e_2}\ \underbrace{00110011001100110011010}_{m_2}}.$$

La valeur stockée est exactement $-0.600000023841857910156 25$.

1 Présentation

2 La norme IEEE 754
- Arrondir un nombre
- Du flottant à sa valeur réelle

3 Pièges et conséquences des approximations numériques

4 Erreurs d'arrondi

5 Arithmétique psychédélique

Des champs binaires à la valeur représentée

On considère un flottant au format $(1, E, M)$. On pose $d = 2^{E-1} - 1$.

- Lire le bit de signe s et interpréter le champ exposant comme un entier non signé e_d.

Des champs binaires à la valeur représentée

On considère un flottant au format $(1, E, M)$. On pose $d = 2^{E-1} - 1$.

- Lire le bit de signe s et interpréter le champ exposant comme un entier non signé e_d.
- Les bits $c_1, \ldots, c_M$ du champ fraction donnent

$$m = (0.c_1 \ldots c_M)_2 = \sum_{i=1}^{M} c_i 2^{-i}.$$

Des champs binaires à la valeur représentée

On considère un flottant au format $(1, E, M)$. On pose $d = 2^{E-1} - 1$.

- Lire le bit de signe s et interpréter le champ exposant comme un entier non signé e_d.
- Les bits $c_1, \ldots, c_M$ du champ fraction donnent

$$m = (0.c_1 \ldots c_M)_2 = \sum_{i=1}^{M} c_i 2^{-i}.$$

- Déterminer la valeur selon le champ exposant :

$$x = \begin{cases} (-1)^s m \, 2^{1-d} & \text{si } e_d = 0, \\ (-1)^s (1+m) 2^{e_d - d} & \text{si } 1 \leq e_d \leq 2^E - 2. \end{cases}$$

Le premier cas comprend les subnormaux et les zéros signés.

Des champs binaires à la valeur représentée

On considère un flottant au format $(1, E, M)$. On pose $d = 2^{E-1} - 1$.

- Lire le bit de signe s et interpréter le champ exposant comme un entier non signé e_d.
- Les bits $c_1, \ldots, c_M$ du champ fraction donnent

$$m = (0.c_1 \ldots c_M)_2 = \sum_{i=1}^{M} c_i 2^{-i}.$$

- Déterminer la valeur selon le champ exposant :

$$x = \begin{cases} (-1)^s m\, 2^{1-d} & \text{si } e_d = 0, \\ (-1)^s (1+m) 2^{e_d - d} & \text{si } 1 \leq e_d \leq 2^E - 2. \end{cases}$$

  Le premier cas comprend les subnormaux et les zéros signés.
- Si $e_d = 2^E - 1$, le codage représente un infini lorsque $m = 0$, un NaN sinon.

Des champs binaires à la valeur représentée

On considère un flottant au format $(1, E, M)$. On pose $d = 2^{E-1} - 1$.

- Lire le bit de signe s et interpréter le champ exposant comme un entier non signé e_d.
- Les bits $c_1, \ldots, c_M$ du champ fraction donnent

$$m = (0.c_1 \ldots c_M)_2 = \sum_{i=1}^{M} c_i 2^{-i}.$$

- Déterminer la valeur selon le champ exposant :

$$x = \begin{cases} (-1)^s m\, 2^{1-d} & \text{si } e_d = 0, \\ (-1)^s (1+m) 2^{e_d - d} & \text{si } 1 \leq e_d \leq 2^E - 2. \end{cases}$$

  Le premier cas comprend les subnormaux et les zéros signés.
- Si $e_d = 2^E - 1$, le codage représente un infini lorsque $m = 0$, un NaN sinon.
- Pour une valeur finie, ce décodage est *exact* : aucun arrondi n'est nécessaire.

Décodage : retrouver la valeur stockée

En binary32, considérons le codage

$$\underbrace{1}_{\text{signe}}\ \underbrace{01111110}_{\text{exposant}}\ \underbrace{00110011001100110011010}_{\text{fraction}}.$$

- Le bit de signe donne $s = 1$.

Décodage : retrouver la valeur stockée

En binary32, considérons le codage

$$\underbrace{1}_{\text{signe}}\ \underbrace{01111110}_{\text{exposant}}\ \underbrace{00110011001100110011010}_{\text{fraction}}.$$

- Le bit de signe donne $s = 1$.
- Le champ exposant donne

$$e_d = (01111110)_2 = 126.$$

Le nombre est normalisé, avec $e = 126 - 127 = -1$.

Décodage : retrouver la valeur stockée

En binary32, considérons le codage

$$\underbrace{1}_{\text{signe}}\ \underbrace{01111110}_{\text{exposant}}\ \underbrace{00110011001100110011010}_{\text{fraction}}.$$

- Le bit de signe donne $s = 1$.
- Le champ exposant donne

$$e_d = (01111110)_2 = 126.$$

  Le nombre est normalisé, avec $e = 126 - 127 = -1$.
- Interprétée comme un entier non signé, la fraction stockée vaut

$$F = (00110011001100110011010)_2 = 1677722.$$

  Ainsi, $m = F\,2^{-23}$.

Décodage : retrouver la valeur stockée (suite)

On considère le codage

$$\underbrace{1}_{\text{signe}}\ \underbrace{01111110}_{\text{exposant}}\ \underbrace{00110011001100110011010}_{\text{fraction}}.$$

- La valeur exacte du flottant est donc

$$\begin{aligned} x &= -\left(1 + \frac{1677722}{2^{23}}\right) 2^{-1} \\ &= -\frac{5033165}{8388608} \\ &= -0.60000002384185791015625. \end{aligned}$$

Décodage : retrouver la valeur stockée (suite)

On considère le codage

$$\underbrace{1}_{\text{signe}}\ \underbrace{01111110}_{\text{exposant}}\ \underbrace{00110011001100110011010}_{\text{fraction}}.$$

- La valeur exacte du flottant est donc

$$\begin{aligned} x &= -\left(1+\frac{1677722}{2^{23}}\right)2^{-1} \\ &= -\frac{5033165}{8388608} \\ &= -0.60000002384185791015625. \end{aligned}$$

- On retrouve la valeur stockée après arrondi, et non le nombre initial -0.6.

1 Présentation

2 La norme IEEE 754
- Arrondir un nombre
- Du flottant à sa valeur réelle

3 Pièges et conséquences des approximations numériques

4 Erreurs d'arrondi

5 Arithmétique psychédélique

Le problème du double arrondi

- Un résultat peut être calculé dans un format intermédiaire, puis converti vers un format moins précis.

Le problème du double arrondi

- Un résultat peut être calculé dans un format intermédiaire, puis converti vers un format moins précis.
- Il subit alors deux arrondis successifs :

$$x \longrightarrow \mathrm{arr}_{M'}(x) \longrightarrow \mathrm{arr}_{M}\big(\mathrm{arr}_{M'}(x)\big), \qquad M' > M.$$

Ici, arr_k désigne l'arrondi au plus proche, au pair, avec k bits de fraction.

Le problème du double arrondi

- Un résultat peut être calculé dans un format intermédiaire, puis converti vers un format moins précis.
- Il subit alors deux arrondis successifs :

$$x \longrightarrow \mathrm{arr}_{M'}(x) \longrightarrow \mathrm{arr}_{M}\big(\mathrm{arr}_{M'}(x)\big), \qquad M' > M.$$

  Ici, arr_k désigne l'arrondi au plus proche, au pair, avec k bits de fraction.
- On pourrait penser que cela revient à arrondir directement dans le format final. Pourtant, en général,

$$\boxed{\mathrm{arr}_{M}\big(\mathrm{arr}_{M'}(x)\big) \neq \mathrm{arr}_{M}(x)}$$

  est possible.

Le problème du double arrondi

- Un résultat peut être calculé dans un format intermédiaire, puis converti vers un format moins précis.
- Il subit alors deux arrondis successifs :

$$x \longrightarrow \mathrm{arr}_{M'}(x) \longrightarrow \mathrm{arr}_{M}\big(\mathrm{arr}_{M'}(x)\big), \qquad M' > M.$$

  Ici, arr_k désigne l'arrondi au plus proche, au pair, avec k bits de fraction.
- On pourrait penser que cela revient à arrondir directement dans le format final. Pourtant, en général,

$$\boxed{\mathrm{arr}_{M}\big(\mathrm{arr}_{M'}(x)\big) \neq \mathrm{arr}_{M}(x)}$$

  est possible.
- Le premier arrondi peut placer le résultat *exactement au milieu* de deux valeurs du format final, alors que le nombre initial se trouvait d'un côté du milieu.

Double arrondi : un exemple binaire

On veut arrondir $x = (1.010101)_2$ à trois bits de fraction, au plus proche, au pair.

- Les deux valeurs qui encadrent x sont

$$a = (1.010)_2, \qquad b = (1.011)_2.$$

Leur milieu est $h = (1.010100)_2$. Comme $x > h$, l'arrondi direct donne b.

Double arrondi : un exemple binaire

On veut arrondir $x = (1.010101)_2$ à trois bits de fraction, au plus proche, au pair.

- Les deux valeurs qui encadrent x sont

$$a = (1.010)_2, \qquad b = (1.011)_2.$$

  Leur milieu est $h = (1.010100)_2$. Comme $x > h$, l'arrondi direct donne b.

- Arrondissons d'abord à cinq bits de fraction :

$$(1.010101)_2 \longrightarrow (1.01010)_2 = h.$$

  On était à équidistance ; le dernier bit conservé est choisi égal à 0.

Double arrondi : un exemple binaire

On veut arrondir $x = (1.010101)_2$ à trois bits de fraction, au plus proche, au pair.

- Les deux valeurs qui encadrent x sont

$$a = (1.010)_2, \qquad b = (1.011)_2.$$

  Leur milieu est $h = (1.010100)_2$. Comme $x > h$, l'arrondi direct donne b.
- Arrondissons d'abord à cinq bits de fraction :

$$(1.010101)_2 \longrightarrow (1.01010)_2 = h.$$

  On était à équidistance ; le dernier bit conservé est choisi égal à 0.
- Arrondissons ensuite à trois bits de fraction :

$$h = (1.0101)_2 \longrightarrow (1.010)_2 = a.$$

  Cette fois encore, l'égalité est départagée au pair.

Double arrondi : un exemple binaire (fin)

On obtient donc deux résultats différents :

$$\underbrace{x \longrightarrow b}_{\text{arrondi direct}}, \qquad \underbrace{x \longrightarrow h \longrightarrow a}_{\text{double arrondi}}.$$

Anti-missile Patriot

- En 1991, une batterie Patriot ne parvient pas à suivre un missile irakien Scud. Bilan : 100 blessés, 28 morts.

Anti-missile Patriot

- En 1991, une batterie Patriot ne parvient pas à suivre un missile irakien Scud. Bilan : 100 blessés, 28 morts.
- Un micro-processeur interne calcule l'heure en multiples de dixièmes de secondes.

Anti-missile Patriot

- En 1991, une batterie Patriot ne parvient pas à suivre un missile irakien Scud. Bilan : 100 blessés, 28 morts.
- Un micro-processeur interne calcule l'heure en multiples de dixièmes de secondes.
- Le nombre de dixièmes de secondes depuis le démarrage est stocké dans un registre entier puis multiplié par une approx. de $\frac{1}{10}$ sur 24 bits pour obtenir le temps en seconde.

Anti-missile Patriot

- En 1991, une batterie Patriot ne parvient pas à suivre un missile irakien Scud. Bilan : 100 blessés, 28 morts.
- Un micro-processeur interne calcule l'heure en multiples de dixièmes de secondes.
- Le nombre de dixièmes de secondes depuis le démarrage est stocké dans un registre entier puis multiplié par une approx. de $\frac{1}{10}$ sur 24 bits pour obtenir le temps en seconde.
- Approximation utilisée : $0.1 \simeq 209715 \cdot 2^{-21} = 0.09999990463256836$, erreur d'environ 10^{-7}.

Anti-missile Patriot (suite)

- Approximation utilisée (rappel) :
  $0.1 \simeq 209715 \cdot 2^{-21} = 0.0999999046325683 6$, erreur d'environ 10^{-7}.

Anti-missile Patriot (suite)

- Approximation utilisée (rappel) :
  $0.1 \simeq 209715 \cdot 2^{-21} = 0.0999999046325683 6$, erreur d'environ 10^{-7}.
- Après environ 100 heures de fonctionnement, la conversion qui utilise le dixième tronqué sous-estime le temps écoulé depuis le démarrage d'environ 0,34 seconde.

Anti-missile Patriot (suite)

- Approximation utilisée (rappel) :
  $0.1 \simeq 209715 \cdot 2^{-21} = 0.0999999046325683 6$, erreur d'environ 10^{-7}.
- Après environ 100 heures de fonctionnement, la conversion qui utilise le dixième tronqué sous-estime le temps écoulé depuis le démarrage d'environ 0,34 seconde.
- Pour suivre le Scud, le logiciel calcule la durée entre deux observations radar. Les deux dates sont converties avec des précisions différentes. La durée calculée est ainsi **surestimée d'environ** 0,34 **seconde**.

Anti-missile Patriot (suite)

- Approximation utilisée (rappel) :
  $0.1 \simeq 209715 \cdot 2^{-21} = 0.0999999046325683 6$, erreur d'environ 10^{-7}.
- Après environ 100 heures de fonctionnement, la conversion qui utilise le dixième tronqué sous-estime le temps écoulé depuis le démarrage d'environ 0,34 seconde.
- Pour suivre le Scud, le logiciel calcule la durée entre deux observations radar. Les deux dates sont converties avec des précisions différentes. La durée calculée est ainsi **surestimée d'environ** 0,34 **seconde**.
- La zone où le radar s'attend à retrouver le Scud est décalée de plus de 500 mètres. Le radar perd le suivi et la batterie ne lance aucun intercepteur.

1 Présentation

2 La norme IEEE 754
- Arrondir un nombre
- Du flottant à sa valeur réelle

3 Pièges et conséquences des approximations numériques

4 Erreurs d'arrondi

5 Arithmétique psychédélique

L'epsilon machine

On considère un format binaire $(1, E, M)$, avec M bits de fraction.

- Le nombre 1 est représentable exactement :

$$1 = (1.\underbrace{00\ldots0}_{M \text{ bits}})_2 \times 2^0.$$

L'epsilon machine

On considère un format binaire $(1, E, M)$, avec M bits de fraction.

- Le nombre 1 est représentable exactement :

$$1 = (1.\underbrace{00\ldots0}_{M \text{ bits}})_2 \times 2^0.$$

- Son successeur dans ce format est

$$(1.\underbrace{00\ldots01}_{M \text{ bits}})_2 \times 2^0 = 1 + 2^{-M}.$$

L'epsilon machine

On considère un format binaire $(1, E, M)$, avec M bits de fraction.

- Le nombre 1 est représentable exactement :

$$1 = (1.\underbrace{00\ldots0}_{M \text{ bits}})_2 \times 2^0.$$

- Son successeur dans ce format est

$$(1.\underbrace{00\ldots01}_{M \text{ bits}})_2 \times 2^0 = 1 + 2^{-M}.$$

- On appelle *epsilon machine* l'écart entre 1 et son successeur :

$$\boxed{\varepsilon = 2^{-M}}.$$

L'epsilon machine

On considère un format binaire $(1, E, M)$, avec M bits de fraction.

- Le nombre 1 est représentable exactement :

$$1 = (1.\underbrace{00\ldots0}_{M \text{ bits}})_2 \times 2^0.$$

- Son successeur dans ce format est

$$(1.\underbrace{00\ldots01}_{M \text{ bits}})_2 \times 2^0 = 1 + 2^{-M}.$$

- On appelle *epsilon machine* l'écart entre 1 et son successeur :

$$\boxed{\varepsilon = 2^{-M}}.$$

- Ainsi :

Format	M	ε
binary32	23	$2^{-23} \approx 1.19 \times 10^{-7}$
binary64	52	$2^{-52} \approx 2.22 \times 10^{-16}$

Erreur absolue de représentation

On note $f(x)$ la valeur réelle du flottant obtenu en arrondissant le réel x au plus proche, au pair.
On suppose

$$x_{\min} \leq |x| \leq x_{\max},$$

où $x_{\min}$ est le plus petit normalisé positif et $x_{\max}$ le plus grand flottant fini.

- L'*erreur absolue* de représentation est

$$\boxed{\mathcal{E}_{\text{abs}}(x) = |f(x) - x|}.$$

Erreur absolue de représentation

On note $f(x)$ la valeur réelle du flottant obtenu en arrondissant le réel x au plus proche, au pair.
On suppose

$$x_{\min} \leq |x| \leq x_{\max},$$

où $x_{\min}$ est le plus petit normalisé positif et $x_{\max}$ le plus grand flottant fini.

- L'*erreur absolue* de représentation est

$$\boxed{\mathcal{E}_{\text{abs}}(x) = |f(x) - x|}.$$

- Soit e l'unique entier tel que

$$2^e \leq |x| < 2^{e+1}.$$

Dans cette plage, le pas entre flottants est 2^{e-M}.

Erreur absolue de représentation

On note $f(x)$ la valeur réelle du flottant obtenu en arrondissant le réel x au plus proche, au pair.
On suppose

$$x_{\min} \leq |x| \leq x_{\max},$$

où $x_{\min}$ est le plus petit normalisé positif et $x_{\max}$ le plus grand flottant fini.

- L'*erreur absolue* de représentation est

$$\boxed{\mathcal{E}_{\text{abs}}(x) = |f(x) - x|}.$$

- Soit e l'unique entier tel que

$$2^e \leq |x| < 2^{e+1}.$$

  Dans cette plage, le pas entre flottants est 2^{e-M}.
- L'arrondi au plus proche donne une erreur au plus égale à la moitié de ce pas :

$$\boxed{|f(x) - x| \leq 2^{e-M-1} = \frac{\varepsilon}{2} 2^e}.$$

Erreur absolue de représentation

On note $f(x)$ la valeur réelle du flottant obtenu en arrondissant le réel x au plus proche, au pair.
On suppose

$$x_{\min} \leq |x| \leq x_{\max},$$

où $x_{\min}$ est le plus petit normalisé positif et $x_{\max}$ le plus grand flottant fini.

- L'*erreur absolue* de représentation est

$$\boxed{\mathcal{E}_{\text{abs}}(x) = |f(x) - x|}.$$

- Soit e l'unique entier tel que

$$2^e \leq |x| < 2^{e+1}.$$

  Dans cette plage, le pas entre flottants est 2^{e-M}.
- L'arrondi au plus proche donne une erreur au plus égale à la moitié de ce pas :

$$\boxed{|f(x) - x| \leq 2^{e-M-1} = \frac{\varepsilon}{2} 2^e}.$$

Erreur relative de représentation

Sous les mêmes hypothèses, $x \neq 0$.

- L'*erreur relative* de représentation est

$$\boxed{\mathcal{E}_{\text{rel}}(x) = \frac{|f(x) - x|}{|x|}}.$$

Erreur relative de représentation

Sous les mêmes hypothèses, $x \neq 0$.

- L'*erreur relative* de représentation est

$$\boxed{\mathcal{E}_{\text{rel}}(x) = \frac{|f(x) - x|}{|x|}}.$$

- Comme $|x| \geq 2^e$, on obtient

$$\frac{|f(x) - x|}{|x|} \leq \frac{2^{e-M-1}}{2^e} = 2^{-M-1}.$$

Ainsi,

$$\boxed{\mathcal{E}_{\text{rel}}(x) \leq \frac{\varepsilon}{2}}.$$

Erreur relative de représentation

Sous les mêmes hypothèses, $x \neq 0$.

- L'*erreur relative* de représentation est

$$\boxed{\mathcal{E}_{\text{rel}}(x) = \frac{|f(x) - x|}{|x|}}.$$

- Comme $|x| \geq 2^e$, on obtient

$$\frac{|f(x) - x|}{|x|} \leq \frac{2^{e-M-1}}{2^e} = 2^{-M-1}.$$

Ainsi,

$$\boxed{\mathcal{E}_{\text{rel}}(x) \leq \frac{\varepsilon}{2}}.$$

- On peut aussi écrire

$$\boxed{f(x) = x(1+\delta), \qquad |\delta| \leq \frac{\varepsilon}{2}}.$$

Erreur relative de représentation

Sous les mêmes hypothèses, $x \neq 0$.

$$\boxed{\mathcal{E}_{\text{rel}}(x) \leq \frac{\varepsilon}{2}}.$$

Cette borne uniforme ne s'applique plus, en général, aux réels de valeur absolue inférieure à x_{min} : près de zéro, les subnormaux conservent un pas constant, mais la précision relative se dégrade.

1 Présentation

2 La norme IEEE 754
- Arrondir un nombre
- Du flottant à sa valeur réelle

3 Pièges et conséquences des approximations numériques

4 Erreurs d'arrondi

5 Arithmétique psychédélique

Associativité

```
1  void associativite(void) {
2    float eps = FLT_EPSILON;//epsilon machine
3      float demi = eps / 2.0f;
4
5      float gauche = (1.0f + demi) + demi;
6      float droite = 1.0f + (demi + demi);
7
8      printf("(1 + eps/2) + eps/2 = %.9g\n", gauche);
9      printf("1 + (eps/2 + eps/2) = %.9g\n\n", droite);
10 }
11
```

Distributivité

```
1  void distributivite(void) {
2      float demi = FLT_EPSILON / 2.0f;// Les deux opérandes sont des float
       .
3      float somme = 1.0f + demi;
4      float produit = 3.0f * demi;
5
6      float gauche = 3.0f * somme;
7      float droite = 3.0f + produit;
8
9      printf("3 * (1 + eps/2)    = %.9g\n", gauche);
10     printf("3 * 1 + 3 * eps/2 = %.9g\n\n", droite);
11 }
12
```

Représentation de un dixième

```c
void representation_d_un_dixieme(void) {
    float xf = 0.1f;
    double xd = 0.1;//0.1 est un double par défaut

    printf("1/10   = 0.1\n");
    printf("0.1f   = %.27f\n", xf);
    printf("0.1    = %.55f\n", xd);
    printf("0.1f > 0.1 : %d\n\n",(xf>xd));
}
```

Additions successives

```c
void additions_successives(void) {
    float somme_f = 0.0f;
    double somme_d = 0.0;
    int i = 0;

    while (i < 10) {
        somme_f = somme_f + 0.1f;
        somme_d = somme_d + 0.1;
        i++;
    }

    printf("Somme en float  = %.17g\n", somme_f);
    printf("Somme en double = %.17g\n", somme_d);
}

```

Main

```c
#include <float.h>
#include <stdio.h>

void associativite(void);
void distributivite(void);
void representation_d_un_dixieme(void);
void additions_successives(void);

int main(void) {
    associativite();
    distributivite();
    representation_d_un_dixieme();
    additions_successives();
    return 0;
}

```

Rendu

```
$ gcc -Wall flottants_psychedelique.c
$ ./a.out
(1 + eps/2) + eps/2 = 1
1 + (eps/2 + eps/2) = 1.00000012

3 * (1 + eps/2)    = 3
3 * 1 + 3 * eps/2 = 3.00000024

1/10    = 0.1
0.1f    = 0.100000001490116119384765625
0.1     = 0.1000000000000000055511151231257827021181583->
                                                3404541015625
0.1f > 0.1 : 1

Somme en float  = 1.0000001192092896
Somme en double = 0.99999999999999989
```