Nombres flottants

Ivan Noyer

Lycée Thiers

Crédits

- `Informatique pour tous en classes préparatoires aux grande écoles (Eyrolles)`
- Nombres flottants : Wikipedia
- cpprefrence (en C)

Introduction

Objectif : représenter une partie des nombre rationnels : des nombres avec une quantité bornée de chiffres après la virgule.

- Idée : on se donne tous les chiffres composants le nombre plus la position de la virgule. Dans cet esprit $(1234, 2)$ représente 12,34, c'est à dire $1.234 \cdot 10^1$.

Introduction

Objectif : représenter une partie des nombre rationnels : des nombres avec une quantité bornée de chiffres après la virgule.

- Idée : on se donne tous les chiffres composants le nombre plus la position de la virgule. Dans cet esprit $(1234, 2)$ représente 12,34, c'est à dire $1.234 \cdot 10^1$.
- Moyen : tous les nombres réels non nuls peuvent de manière unique s'écrire sous la forme $(-1)^s(1+m)2^e$ dans laquelle

Introduction

Objectif : représenter une partie des nombre rationnels : des nombres avec une quantité bornée de chiffres après la virgule.

- Idée : on se donne tous les chiffres composants le nombre plus la position de la virgule. Dans cet esprit $(1234, 2)$ représente 12,34, c'est à dire $1.234 \cdot 10^1$.
- Moyen : tous les nombres réels non nuls peuvent de manière unique s'écrire sous la forme $(-1)^s(1 + m)2^e$ dans laquelle
  - $s \in \{0, 1\}$ est le *signe* du nombre

Introduction

Objectif : représenter une partie des nombre rationnels : des nombres avec une quantité bornée de chiffres après la virgule.

- Idée : on se donne tous les chiffres composants le nombre plus la position de la virgule. Dans cet esprit $(1234, 2)$ représente 12,34, c'est à dire $1.234 \cdot 10^1$.
- Moyen : tous les nombres réels non nuls peuvent de manière unique s'écrire sous la forme $(-1)^s(1+m)2^e$ dans laquelle
  - $s \in \{0, 1\}$ est le *signe* du nombre
  - $m \in [0, 1[$ est sa *mantisse*

Introduction

Objectif : représenter une partie des nombre rationnels : des nombres avec une quantité bornée de chiffres après la virgule.

- Idée : on se donne tous les chiffres composants le nombre plus la position de la virgule. Dans cet esprit $(1234, 2)$ représente 12,34, c'est à dire $1.234 \cdot 10^1$.
- Moyen : tous les nombres réels non nuls peuvent de manière unique s'écrire sous la forme $(-1)^s(1+m)2^e$ dans laquelle
  - $s \in \{0, 1\}$ est le *signe* du nombre
  - $m \in [0, 1[$ est sa *mantisse*
  - $e \in \mathbb{Z}$ est son *exposant*

Introduction

- Moyen : tous les nombres réels non nuls peuvent de manière unique s'écrire sous la forme $(-1)^s(1+m)2^e$ dans laquelle $s \in \{0,1\}$ est le *signe* du nombre, $m \in [0,1[$ est sa *mantisse*, $e \in \mathbb{Z}$ est son *exposant*.

Introduction

- Moyen : tous les nombres réels non nuls peuvent de manière unique s'écrire sous la forme $(-1)^s(1+m)2^e$ dans laquelle $s \in \{0,1\}$ est le *signe* du nombre, $m \in [0,1[$ est sa *mantisse*, $e \in \mathbb{Z}$ est son *exposant*.
- L'exposant du nombre représente la position de la virgule dans son expression en base 2.

Introduction

- Moyen : tous les nombres réels non nuls peuvent de manière unique s'écrire sous la forme $(-1)^s(1+m)2^e$ dans laquelle $s \in \{0,1\}$ est le *signe* du nombre, $m \in [0,1[$ est sa *mantisse*, $e \in \mathbb{Z}$ est son *exposant*.
- L'exposant du nombre représente la position de la virgule dans son expression en base 2.
- $1+m$ est un nombre entre 1 et 2 (exclu). Il s'écrit sous la forme $1.c_1 \ldots c_i \ldots$ avec les $c_i \in \{0,1\}$. Comme les ordinateurs ne gère que des quantités finies, le nombre de chiffres binaires c_i est borné (en fait il est fixe, le plus souvent égal à 23 ou 52).

Pourquoi les nombres à virgule flottante ?

- L'avantage de la représentation en virgule flottante par rapport à la virgule fixe est que la virgule flottante est capable, à nombre de chiffres égal, de gérer un intervalle de nombres réels plus important.

Pourquoi les nombres à virgule flottante ?

- L'avantage de la représentation en virgule flottante par rapport à la virgule fixe est que la virgule flottante est capable, à nombre de chiffres égal, de gérer un intervalle de nombres réels plus important.
- Considérons une représentation en virgule fixe qui a 5 chiffres dont un après la virgule. Elle peut exprimer 10^5 nombres décimaux dans $[0; 9999.9]$. Avec 5 fois le chiffre 1, on ne représente que 1111.1.

Pourquoi les nombres à virgule flottante ?

- L'avantage de la représentation en virgule flottante par rapport à la virgule fixe est que la virgule flottante est capable, à nombre de chiffres égal, de gérer un intervalle de nombres réels plus important.
- Considérons une représentation en virgule fixe qui a 5 chiffres dont un après la virgule. Elle peut exprimer 10^5 nombres décimaux dans $[0; 9999.9]$. Avec 5 fois le chiffre 1, on ne représente que 1111.1.
- Avec une représentation en virgule flottante et 5 chiffres 1 : on peut représenter 11111, 1111.1, 111.11, 11.111, 1.1111, .11111, donc 6 fois plus d'expressions en virgule flottante qu'en virgule fixe. De plus l'intervalle de représentation est plus grand : $[0; 99999]$.

Pourquoi les nombres à virgule flottante ?

- L'avantage de la représentation en virgule flottante par rapport à la virgule fixe est que la virgule flottante est capable, à nombre de chiffres égal, de gérer un intervalle de nombres réels plus important.
- Considérons une représentation en virgule fixe qui a 5 chiffres dont un après la virgule. Elle peut exprimer 10^5 nombres décimaux dans $[0; 9999.9]$. Avec 5 fois le chiffre 1, on ne représente que 1111.1.
- Avec une représentation en virgule flottante et 5 chiffres 1 : 6 fois plus d'expressions en virgule flottante qu'en virgule fixe. De plus l'intervalle de représentation est plus grand : $[0; 99999]$.
- Cependant, il faut alors coder la position de la virgule. Cela demande donc un peu plus de place.

IEEE754, présentation

IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) (standard IEEE pour l'arithmétique binaire en virgule flottante) : IEEE 754.

- Standard le plus employé actuellement pour le calcul des nombres à virgule flottante dans le domaine informatique, avec les CPU et les FPU.

IEEE754, présentation

IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) (standard IEEE pour l'arithmétique binaire en virgule flottante) : IEEE 754.

- Standard le plus employé actuellement pour le calcul des nombres à virgule flottante dans le domaine informatique, avec les CPU et les FPU.
- Le standard définit les formats de représentation des nombres à virgule flottante (signe, mantisse, exposant, nombres dénormalisés) et valeurs spéciales (infinis et NaN), en même temps qu'un ensemble d'opérations sur les nombres flottants.

IEEE754, présentation

IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) (standard IEEE pour l'arithmétique binaire en virgule flottante) : IEEE 754.

- Standard le plus employé actuellement pour le calcul des nombres à virgule flottante dans le domaine informatique, avec les CPU et les FPU.
- Le standard définit les formats de représentation des nombres à virgule flottante (signe, mantisse, exposant, nombres dénormalisés) et valeurs spéciales (infinis et NaN), en même temps qu'un ensemble d'opérations sur les nombres flottants.
- Il décrit aussi quatre modes d'arrondi et cinq exceptions (comprenant les conditions dans lesquelles une exception se produit, et ce qui se passe dans ce cas).

IEEE754, présentation (1)

- Les quatre modes d'arrondi :

IEEE754, présentation (1)

- Les quatre modes d'arrondi :
  1. Vers moins l'infini (ex : l'arrondi de la partie entière, $\lfloor -3.4 \rfloor = -4$).

IEEE754, présentation (1)

- Les quatre modes d'arrondi :
  1. Vers moins l'infini (ex : l'arrondi de la partie entière, $\lfloor -3.4 \rfloor = -4$).
  2. Vers plus l'infini (ex : l'arrondi à l'unité de -3.4 est -3)

IEEE754, présentation (1)

- Les quatre modes d'arrondi :
  1. Vers moins l'infini (ex : l'arrondi de la partie entière, $\lfloor -3.4 \rfloor = -4$).
  2. Vers plus l'infini (ex : l'arrondi à l'unité de -3.4 est -3)
  3. Vers zéro (-3.4 arrondi en -3, 3.4 arrondi en 3).

IEEE754, présentation (1)

- Les quatre modes d'arrondi :
  1. Vers moins l'infini (ex : l'arrondi de la partie entière, $\lfloor -3.4 \rfloor = -4$).
  2. Vers plus l'infini (ex : l'arrondi à l'unité de -3.4 est -3)
  3. Vers zéro (-3.4 arrondi en -3, 3.4 arrondi en 3).
  4. Au plus proche (avec le cas particulier de l'équidistance : le nb 1.5 doit-il être arrondi à l'unité à 1 ou 2 ?).

IEEE754, présentation (1)

- Les quatre modes d'arrondi :
  1. Vers moins l'infini (ex : l'arrondi de la partie entière, $\lfloor -3.4 \rfloor = -4$).
  2. Vers plus l'infini (ex : l'arrondi à l'unité de -3.4 est -3)
  3. Vers zéro (-3.4 arrondi en -3, 3.4 arrondi en 3).
  4. Au plus proche (avec le cas particulier de l'équidistance : le nb 1.5 doit-il être arrondi à l'unité à 1 ou 2 ?).
- La version 1985 de la norme IEEE 754 définit 4 formats pour représenter des nombres à virgule flottante :

IEEE754, présentation (1)

- Les quatre modes d'arrondi :
  1. Vers moins l'infini (ex : l'arrondi de la partie entière, $\lfloor -3.4 \rfloor = -4$).
  2. Vers plus l'infini (ex : l'arrondi à l'unité de -3.4 est -3)
  3. Vers zéro (-3.4 arrondi en -3, 3.4 arrondi en 3).
  4. Au plus proche (avec le cas particulier de l'équidistance : le nb 1.5 doit-il être arrondi à l'unité à 1 ou 2 ?).
- La version 1985 de la norme IEEE 754 définit 4 formats pour représenter des nombres à virgule flottante :
  1. simple précision (32 bits : 1 bit de signe, 8 bits d'exposant (-126 à 127), 23 bits de mantisse, avec bit 1 implicite),

IEEE754, présentation (1)

- Les quatre modes d'arrondi :
  1. Vers moins l'infini (ex : l'arrondi de la partie entière, $\lfloor -3.4 \rfloor = -4$).
  2. Vers plus l'infini (ex : l'arrondi à l'unité de -3.4 est -3)
  3. Vers zéro (-3.4 arrondi en -3, 3.4 arrondi en 3).
  4. Au plus proche (avec le cas particulier de l'équidistance : le nb 1.5 doit-il être arrondi à l'unité à 1 ou 2 ?).
- La version 1985 de la norme IEEE 754 définit 4 formats pour représenter des nombres à virgule flottante :
  1. simple précision (32 bits : 1 bit de signe, 8 bits d'exposant (-126 à 127), 23 bits de mantisse, avec bit 1 implicite),
  2. simple précision étendue ($\geq$ 43 bits, obsolète, implémenté en pratique par la double précision),

IEEE754, présentation (1)

- Les quatre modes d'arrondi :
  1. Vers moins l'infini (ex : l'arrondi de la partie entière, $\lfloor -3.4 \rfloor = -4$).
  2. Vers plus l'infini (ex : l'arrondi à l'unité de -3.4 est -3)
  3. Vers zéro (-3.4 arrondi en -3, 3.4 arrondi en 3).
  4. Au plus proche (avec le cas particulier de l'équidistance : le nb 1.5 doit-il être arrondi à l'unité à 1 ou 2 ?).
- La version 1985 de la norme IEEE 754 définit 4 formats pour représenter des nombres à virgule flottante :
  1. simple précision (32 bits : 1 bit de signe, 8 bits d'exposant (-126 à 127), 23 bits de mantisse, avec bit 1 implicite),
  2. simple précision étendue ($\geq$ 43 bits, obsolète, implémenté en pratique par la double précision),
  3. double précision (64 bits : 1 bit de signe, 11 bits d'exposant (-1022 à 1023), 52 bits de mantisse, avec bit 1 implicite),

IEEE754, présentation (1)

- Les quatre modes d'arrondi :
  1. Vers moins l'infini (ex : l'arrondi de la partie entière, $\lfloor -3.4 \rfloor = -4$).
  2. Vers plus l'infini (ex : l'arrondi à l'unité de -3.4 est -3)
  3. Vers zéro (-3.4 arrondi en -3, 3.4 arrondi en 3).
  4. Au plus proche (avec le cas particulier de l'équidistance : le nb 1.5 doit-il être arrondi à l'unité à 1 ou 2 ?).
- La version 1985 de la norme IEEE 754 définit 4 formats pour représenter des nombres à virgule flottante :
  1. simple précision (32 bits : 1 bit de signe, 8 bits d'exposant (-126 à 127), 23 bits de mantisse, avec bit 1 implicite),
  2. simple précision étendue ($\geq$ 43 bits, obsolète, implémenté en pratique par la double précision),
  3. double précision (64 bits : 1 bit de signe, 11 bits d'exposant (-1022 à 1023), 52 bits de mantisse, avec bit 1 implicite),
  4. double précision étendue ($\geq$ 79 bits)

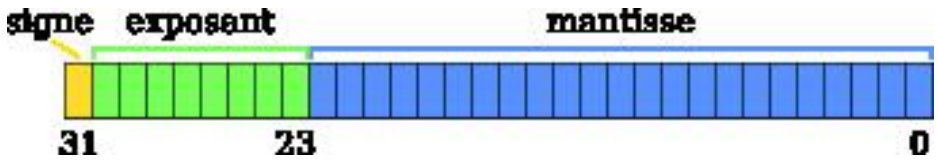
Bit 1, bit *implicite*

Explication

la mantisse représente un nombre décimal entre 1 et 2 (exclu), par exemple 1.1101100011. Rendre le bit 1 implicite consiste à écrire 1101100011, la partie « 1. » étant sous-entendue puisque toujours la même.

Représentation binaire des nombres flottants à précision simple

FIGURE – IEEE-754 simple précision



Plus généralement, les nombres sont écrits au format $(1, E, M)$ donc sur $1 + E + M$ bits :

Signe	Exposant décalé	Mantisse
(1 bit)	(E bits)	(M bits)

Représentation binaire des nombres flottants à précision simple

Ce format (1,8,23) obsolète est privilégié ici parce qu'on peut faire tenir 32 bits sur un tansparent.

- bit de poids fort à 1 : négatif, à 0 : positif.

Représentation binaire des nombres flottants à précision simple

Ce format (1,8,23) obsolète est privilégié ici parce qu'on peut faire tenir 32 bits sur un tansparent.

- bit de poids fort à 1 : négatif, à 0 : positif.
- Exposant : pas de représentation en complément à 2 (car comparer des nombres serait difficile). L'exposant est *décalé*, afin de le stocker sous forme d'un nombre non signé.
  En notant E le nombre de chiffres (toujours le même nombre) de l'exposant, on ajoute un *décalage* (ou *biais*) de $d = 2^{E-1} - 1$.

Représentation binaire des nombres flottants à précision simple

Ce format (1,8,23) obsolète est privilégié ici parce qu'on peut faire tenir 32 bits sur un tansparent.

- bit de poids fort à 1 : négatif, à 0 : positif.
- Exposant : pas de représentation en complément à 2 (car comparer des nombres serait difficile). L'exposant est *décalé*, afin de le stocker sous forme d'un nombre non signé.
  En notant E le nombre de chiffres (toujours le même nombre) de l'exposant, on ajoute un *décalage* (ou *biais*) de $d = 2^{E-1} - 1$.
- Avec $E = 8$, $d = 127$. L'exposant est dans l'intervalle $[-127; 128]$, donc l'exposant décalé est dans $[0, 255]$; 0 et 255 ayant une signification spéciale.

Représentation binaire des nombres flottants à précision simple

- Ils sont longs de 4 octets (32 bits) : (1,8,23)

Représentation binaire des nombres flottants à précision simple

- Ils sont longs de 4 octets (32 bits) : (1,8,23)
- La mantisse complète, le significande, doit être considérée comme une valeur sur 24 bits. Si la mantisse avec bit 1 implicite est 101000 . . . alors le significande en base 2 est 1.101000 . . . .

Représentation binaire des nombres flottants à précision simple

- Ils sont longs de 4 octets (32 bits) : (1,8,23)
- La mantisse complète, le significande, doit être considérée comme une valeur sur 24 bits. Si la mantisse avec bit 1 implicite est 101000 . . . alors le significande en base 2 est 1.101000 . . . .
- La quantité de nombres représentables au format $(1, E, M)$ est grande mais finie. L'ordinateur travaille donc avec des valeurs en général approchées :
  Par exemple avec **float x = 0.1 ;**, l'ordinateur travaille en interne avec $0\ 01111011\ 10011001100110011001101_2$ c'est à dire 0.100000001490116_{10} qui est égal à $\frac{13421773}{134217728}$.

Affichage

- Quand on entre un nombre au clavier, l'ordinateur en calcule une représentation en virgule flottante. Du fait des arrondis, une infinité de nombres peuvent avoir la même représentation comme flottant.

Affichage

- Quand on entre un nombre au clavier, l'ordinateur en calcule une représentation en virgule flottante. Du fait des arrondis, une infinité de nombres peuvent avoir la même représentation comme flottant.
- La relation $\ll$ a la même représentation que $\gg$ est une relation d'équivalence.

Affichage

- Quand on entre un nombre au clavier, l'ordinateur en calcule une représentation en virgule flottante. Du fait des arrondis, une infinité de nombres peuvent avoir la même représentation comme flottant.
- La relation $\ll$ a la même représentation que $\gg$ est une relation d'équivalence.
- Pour le confort de lecture, l'ordinateur affiche l'unique représentant de cette classe d'équivalence qui nécessite le moins de chiffres décimaux.

Affichage

- Quand on entre un nombre au clavier, l'ordinateur en calcule une représentation en virgule flottante. Du fait des arrondis, une infinité de nombres peuvent avoir la même représentation comme flottant.
- La relation « a la même représentation que » est une relation d'équivalence.
- Pour le confort de lecture, l'ordinateur affiche l'unique représentant de cette classe d'équivalence qui nécessite le moins de chiffres décimaux.
- C'est la raison pour laquelle l'ordinateur affiche **0.1** et non le nombre rationnel avec laquel il travaille effectivement.

Affichage

- Quand on entre un nombre au clavier, l'ordinateur en calcule une représentation en virgule flottante. Du fait des arrondis, une infinité de nombres peuvent avoir la même représentation comme flottant.
- La relation « a la même représentation que » est une relation d'équivalence.
- Pour le confort de lecture, l'ordinateur affiche l'unique représentant de cette classe d'équivalence qui nécessite le moins de chiffres décimaux.
- C'est la raison pour laquelle l'ordinateur affiche **0.1** et non le nombre rationnel avec laquel il travaille effectivement.
- Un nombre qui est égal à sa représentation en flottant est dit *représentable exactement* en machine. **0.1** n'est pas représentable exactement en machine ; **1.0** et **3.75** oui.

À propos de l'exposant

Format $(1, E, M)$

- En fonction de la valeur e_d du champ exposant décalé (0, 255, ou autre), certains nombres peuvent avoir une signification spéciale. Ils peuvent être : Des nombres dénormalisés $(e_d = 0)$; Zéro $(e_d = 0)$; Infini $(e_d = 2^{N-1} - 1)$; NaN (*Not a Number* $(e_d = 2^{N-1} - 1)$ $\ll$ pas un nombre $\gg$, comme $0/0$ ou $\sqrt{-1}$).

À propos de l'exposant

Format $(1, E, M)$

- En fonction de la valeur e_d du champ exposant décalé (0, 255, ou autre), certains nombres peuvent avoir une signification spéciale. Ils peuvent être : Des nombres dénormalisés $(e_d = 0)$; Zéro $(e_d = 0)$; Infini $(e_d = 2^{N-1} - 1)$; NaN (*Not a Number* $(e_d = 2^{N-1} - 1)$ « pas un nombre », comme $0/0$ ou $\sqrt{-1}$).
- L'exposant décalé est dans $[\![0, 2^E - 1]\!]$, le décalage est $2^{E-1} - 1$. Les nombres « normalisés » ont un exposant décalé dans $[\![1, 2^E - 2]\!]$.

À propos de l'exposant

Format $(1, E, M)$

- En fonction de la valeur e_d du champ exposant décalé (0, 255, ou autre), certains nombres peuvent avoir une signification spéciale. Ils peuvent être : Des nombres dénormalisés $(e_d = 0)$; Zéro $(e_d = 0)$; Infini $(e_d = 2^{N-1} - 1)$; NaN (*Not a Number* $(e_d = 2^{N-1} - 1)$ « pas un nombre », comme $0/0$ ou $\sqrt{-1}$).
- L'exposant décalé est dans $[\![0, 2^E - 1]\!]$, le décalage est $2^{E-1} - 1$. Les nombres « normalisés » ont un exposant décalé dans $[\![1, 2^E - 2]\!]$.
- Le bit implicite de la mantisse est déterminé par la valeur de l'exposant décalé. Il vaut 0 si l'exposant décalé est égal à 0 et 1 sinon.

À propos de l'exposant

Nombres normalisés et dé-normalisés

Soit le format $(1, E, M)$

- Si l'exposant décalé est différent de 0 et de $2^E - 1$, le bit implicite est 1, et le nombre est dit *normalisé* : $(-1)^s(1+m)2^e$ avec $m \in [0; 1[$.

À propos de l'exposant

Nombres normalisés et dé-normalisés

Soit le format $(1, E, M)$

- Si l'exposant décalé est différent de 0 et de $2^E - 1$, le bit implicite est 1, et le nombre est dit *normalisé* : $(-1)^s(1+m)2^e$ avec $m \in [0;1[$.
- Si l'exposant décalé est nul et la mantisse non nulle, par convention, le bit implicite vaut 0. Le nombre est dit *dé-normalisé*. La représentation au format dé-normalisé est destinée aux très petites quantités en valeur absolue : $(-1)^s(0+m)2^{-d+1}$.

À propos de l'exposant

Nombres normalisés et dé-normalisés

Soit le format $(1, E, M)$

- Si l'exposant décalé est différent de 0 et de $2^E - 1$, le bit implicite est 1, et le nombre est dit *normalisé*.
- Si l'exposant décalé est nul et la mantisse non nulle, par convention, le bit implicite vaut 0. Le nombre est dit *dé-normalisé*. La représentation au format dé-normalisé est destinée aux très petites quantités en valeur absolue.
- La quantité de nombres à virgules flottante sur une machine et grande mais finie. Chaque nombre positif (sauf le plus grand) a un *successeur* et un *prédecesseur* (s'il n'est pas nul) positif.

À propos de l'exposant

Nombres normalisés et dé-normalisés

Soit le format $(1, E, M)$

- Si l'exposant décalé est différent de 0 et de $2^E - 1$, le bit implicite est 1, et le nombre est dit *normalisé*.
- Si l'exposant décalé est nul et la mantisse non nulle, par convention, le bit implicite vaut 0. Le nombre est dit *dé-normalisé*. La représentation au format dé-normalisé est destinée aux très petites quantités en valeur absolue.
- La quantité de nombres à virgules flottante sur une machine et grande mais finie. Chaque nombre positif (sauf le plus grand) a un *successeur* et un *prédecesseur* (s'il n'est pas nul) positif.
- Le successeur du nombre dé-normalisé positif le plus grand est le plus petit nombre normalisé positif.

À propos de l'exposant

Nombres normalisés et dé-normalisés

Soit le format $(1, E, M)$

- Si l'exposant décalé est différent de 0 et de $2^E - 1$, le bit implicite est 1, et le nombre est dit *normalisé*.
- Si l'exposant décalé est nul et la mantisse non nulle, par convention, le bit implicite vaut 0. Le nombre est dit *dé-normalisé*. La représentation au format dé-normalisé est destinée aux très petites quantités en valeur absolue.
- La quantité de nombres à virgules flottante sur une machine et grande mais finie. Chaque nombre positif (sauf le plus grand) a un *successeur* et un *prédecesseur* (s'il n'est pas nul) positif.
- Le successeur du nombre dé-normalisé positif le plus grand est le plus petit nombre normalisé positif.
- Zéro : ni normalisé ni dé-normalisé. Deux écritures : $+0$ et -0.

Nombres dé-normalisés

Pour un format $1, E, M$:

- Mantisse non nulle, champ exposant décalé : E bits à zéro.

Nombres dé-normalisés

Pour un format $1, E, M$:

- Mantisse non nulle, champ exposant décalé : E bits à zéro.
- Tous les nombres dé-normalisés ont le même exposant

Nombres dé-normalisés

Pour un format $1, E, M$:

- Mantisse non nulle, champ exposant décalé : E bits à zéro.
- Tous les nombres dé-normalisés ont le même exposant
- Si la règle était la même que pour les nombres normalisés, l'exposant serait donc de $0 - \mathrm{decalage}$ soit $-2^{E-1} + 1$, donc -1023 pour les flottants double précision.

Nombres dé-normalisés

Pour un format $1, E, M$:

- Mantisse non nulle, champ exposant décalé : E bits à zéro.
- Tous les nombres dé-normalisés ont le même exposant
- Si la règle était la même que pour les nombres normalisés, l'exposant serait donc de $0 - \mathrm{decalage}$ soit $-2^{E-1} + 1$, donc -1023 pour les flottants double précision.
- Mais, par convention, l'exposant pour les nombres dé-normalisés est en fait égal au plus petit exposant de nombre normalisé soit $-2^{E-1} + 1 + 1$. Ce qui change, c'est le bit implicite 0 (dé-normalisé) ou 1 (normalisé) :

Nombres dé-normalisés

Pour un format $1, E, M$:

- Mantisse non nulle, champ exposant décalé : E bits à zéro.
- Tous les nombres dé-normalisés ont le même exposant
- Si la règle était la même que pour les nombres normalisés, l'exposant serait donc de $0 - \text{decalage}$ soit $-2^{E-1} + 1$, donc -1023 pour les flottants double précision.
- Mais, par convention, l'exposant pour les nombres dé-normalisés est en fait égal au plus petit exposant de nombre normalisé soit $-2^{E-1} + 1 + 1$. Ce qui change, c'est le bit implicite 0 (dé-normalisé) ou 1 (normalisé) :
  - Plus petit normalisé positif $(1 + m)2^{-2^{E-1}+1+1}$ avec $m = 0$

Nombres dé-normalisés

Pour un format $1, E, M$:

- Mantisse non nulle, champ exposant décalé : E bits à zéro.
- Tous les nombres dé-normalisés ont le même exposant
- Si la règle était la même que pour les nombres normalisés, l'exposant serait donc de $0 - \text{decalage}$ soit $-2^{E-1} + 1$, donc -1023 pour les flottants double précision.
- Mais, par convention, l'exposant pour les nombres dé-normalisés est en fait égal au plus petit exposant de nombre normalisé soit $-2^{E-1} + 1 + 1$. Ce qui change, c'est le bit implicite 0 (dé-normalisé) ou 1 (normalisé) :
  - Plus petit normalisé positif $(1 + m)2^{-2^{E-1}+1+1}$ avec $m = 0$
  - Plus petit dé-normalisé positif $(0 + m)2^{-2^{E-1}+1+1}$ avec m non nul minimum. Ainsi, m s'écrit avec $M - 1$ bits à 0 suivis de 1 donc $m = 2^{-M}$. Conclusion $2^{-M}2^{-2^{E-1}+1+1}$.

Exposant pour un format $(1, E, M)$

Tableau récapitulatif :

Type	Exposant décalé	Mantisse
Zéros : ± 0	0	0
Nombres dénormalisés	0	$\neq 0$, `0.` implicite
Nombres normalisés	1 à $2^E - 2$	quelconque, `1.` implicite
Infinis $\pm\infty$	$2^E - 1$	0
NaN (*Not a Number*)	$2^E - 1$	différente de 0

Exemple (Exposant e d'un nombre normalisé)

Si $E = 8$, alors $e \in [-126; 127]$. L'exposant -127 (qui est décalé vers la valeur 0) est réservé pour zéro et les nombres dénormalisés, tandis que l'exposant 128 (décalé vers 255) est réservé pour coder les infinis et les NaN.

Real 2 float

Calcul du triplet (s, e, m)

Cas des nombres normalisés. On calcule (s, e, m) en base 10

- **Ecriture de 0** : 32 bits à 0 ou 1 suivi de 31 bits nuls.

Real 2 float

Calcul du triplet (s, e, m)

Cas des nombres normalisés. On calcule (s, e, m) en base 10

- **Ecriture de 0** : 32 bits à 0 ou 1 suivi de 31 bits nuls.
- Ecriture sous forme scientifique au standard décimal : Si $X \neq 0$, $X = (-1)^s.2^e.(1 + m)$ avec

Il faudra exprimer ces nombres au format binaire au moyen d'un triplet (s_2, e_2, m_2).

Real 2 float

Calcul du triplet (s, e, m)

Cas des nombres normalisés. On calcule (s, e, m) en base 10

- **Ecriture de 0** : 32 bits à 0 ou 1 suivi de 31 bits nuls.
- Ecriture sous forme scientifique au standard décimal : Si $X \neq 0$, $X = (-1)^s.2^e.(1 + m)$ avec
  - $s \in \{0, 1\}$

Il faudra exprimer ces nombres au format binaire au moyen d'un triplet (s_2, e_2, m_2).

Real 2 float

Calcul du triplet (s, e, m)

Cas des nombres normalisés. On calcule (s, e, m) en base 10

- **Ecriture de 0** : 32 bits à 0 ou 1 suivi de 31 bits nuls.
- Ecriture sous forme scientifique au standard décimal : Si $X \neq 0$, $X = (-1)^s.2^e.(1 + m)$ avec
  - $s \in \{0, 1\}$
  - $e \in \mathbb{Z}$

Il faudra exprimer ces nombres au format binaire au moyen d'un triplet (s_2, e_2, m_2).

Real 2 float

Calcul du triplet (s, e, m)

Cas des nombres normalisés. On calcule (s, e, m) en base 10

- **Ecriture de 0** : 32 bits à 0 ou 1 suivi de 31 bits nuls.
- Ecriture sous forme scientifique au standard décimal : Si $X \neq 0$, $X = (-1)^s.2^e.(1 + m)$ avec
  - $s \in \{0, 1\}$
  - $e \in \mathbb{Z}$
  - $m \in [0, 1[$

  Il faudra exprimer ces nombres au format binaire au moyen d'un triplet (s_2, e_2, m_2).

Real 2 float

Calcul du triplet (s, e, m)

- **Recherche de** s : 0 si X est positif ou nul, 1 sinon.

Real 2 float

Calcul du triplet (s, e, m)

- **Recherche de** s : 0 si X est positif ou nul, 1 sinon.
- **Recherche de** e :

Real 2 float

Calcul du triplet (s, e, m)

- **Recherche de** s : 0 si X est positif ou nul, 1 sinon.
- **Recherche de** e :
  - si $|X| \geq 2$, diviser par 2 la valeur absolue de X autant de fois que nécessaire jusqu'à obtenir un entier de l'intervalle $[1\,;2[$. e est le nombre de divisions effectuées.

Real 2 float

Calcul du triplet (s, e, m)

- **Recherche de** s : 0 si X est positif ou nul, 1 sinon.
- **Recherche de** e :
  - si $|X| \geq 2$, diviser par 2 la valeur absolue de X autant de fois que nécessaire jusqu'à obtenir un entier de l'intervalle $[1\,;2[$. e est le nombre de divisions effectuées.
  - si $|X| < 1$, multiplier par 2 la valeur absolue de X par 2 autant de fois que nécessaire jusqu'à obtenir un entier de l'intervalle $[1\,;2[$. e est l'opposé du nombre de multiplications.

Real 2 float

Calcul du triplet (s, e, m)

- **Recherche de** s : 0 si X est positif ou nul, 1 sinon.
- **Recherche de** e :
  - si $|X| \geq 2$, diviser par 2 la valeur absolue de X autant de fois que nécessaire jusqu'à obtenir un entier de l'intervalle $[1\,;2[$. e est le nombre de divisions effectuées.
  - si $|X| < 1$, multiplier par 2 la valeur absolue de X par 2 autant de fois que nécessaire jusqu'à obtenir un entier de l'intervalle $[1\,;2[$. e est l'opposé du nombre de multiplications.
  - si $2 > |X| \geq 1$ alors $e = 0$.

Real 2 float

Calcul du triplet (s, e, m)

- **Recherche de** s : 0 si X est positif ou nul, 1 sinon.
- **Recherche de** e :
  - si $|X| \geq 2$, diviser par 2 la valeur absolue de X autant de fois que nécessaire jusqu'à obtenir un entier de l'intervalle $[1\,;2[$. e est le nombre de divisions effectuées.
  - si $|X| < 1$, multiplier par 2 la valeur absolue de X par 2 autant de fois que nécessaire jusqu'à obtenir un entier de l'intervalle $[1\,;2[$. e est l'opposé du nombre de multiplications.
  - si $2 > |X| \geq 1$ alors $e = 0$.
- **Recherche de** m **:** Si on connait X, s et e alors il est facile de trouver m :

$$m = \frac{(-1)^s}{2^e} X - 1.$$

Real 2 float

Triplet (s_2, e_2, m_2) en base 2

- Exemple : $X = -9.6$; $s_{10} = 1$; $9.6 \times 2^{-3} = 1.2 \in [1; 2[$ donc $e_{10} = 3$; $m_{10} = 1.2 - 1$.

Real 2 float

Triplet (s_2, e_2, m_2) en base 2

- Exemple : $X = -9.6$; $s_{10} = 1$; $9.6 \times 2^{-3} = 1.2 \in [1; 2[$ donc $e_{10} = 3$; $m_{10} = 1.2 - 1$.
- Connaissant s, e, m de la notation scientifique décimale, on veut s_2, e_2, m_2 tels que

Real 2 float

Triplet (s_2, e_2, m_2) en base 2

- Exemple : $X = -9.6$; $s_{10} = 1$; $9.6 \times 2^{-3} = 1.2 \in [1; 2[$ donc $e_{10} = 3$; $m_{10} = 1.2 - 1$.
- Connaissant s, e, m de la notation scientifique décimale, on veut s_2, e_2, m_2 tels que
  - bit de signe $s_2 = s_{10}$,

Real 2 float

Triplet (s_2, e_2, m_2) en base 2

- Exemple : $X = -9.6$; $s_{10} = 1$; $9.6 \times 2^{-3} = 1.2 \in [1; 2[$ donc $e_{10} = 3$; $m_{10} = 1.2 - 1$.
- Connaissant s, e, m de la notation scientifique décimale, on veut s_2, e_2, m_2 tels que
  - bit de signe $s_2 = s_{10}$,
  - bits d'exposant $e_2 = \mathrm{bin}(e + 127)$ (conversion de l'exposant *décalé* en base 2),

Real 2 float

Triplet (s_2, e_2, m_2) en base 2

- Exemple : $X = -9.6$; $s_{10} = 1$; $9.6 \times 2^{-3} = 1.2 \in [1; 2[$ donc $e_{10} = 3$; $m_{10} = 1.2 - 1$.
- Connaissant s, e, m de la notation scientifique décimale, on veut s_2, e_2, m_2 tels que
  - bit de signe $s_2 = s_{10}$,
  - bits d'exposant $e_2 = \text{bin}(e + 127)$ (conversion de l'exposant *décalé* en base 2),
  - Pour la mantisse sur 23 bits

    Recommencer à **I** avec la partie décimale du résultat jusqu'à avoir 23 bits de mantisse (et même un peu plus pour arrondir)

Real 2 float

Triplet (s_2, e_2, m_2) en base 2

- Exemple : $X = -9.6$; $s_{10} = 1$; $9.6 \times 2^{-3} = 1.2 \in [1; 2[$ donc $e_{10} = 3$; $m_{10} = 1.2 - 1$.
- Connaissant s, e, m de la notation scientifique décimale, on veut s_2, e_2, m_2 tels que
  - bit de signe $s_2 = s_{10}$,
  - bits d'exposant $e_2 = \text{bin}(e + 127)$ (conversion de l'exposant *décalé* en base 2),
  - Pour la mantisse sur 23 bits
    1. Multiplier m_{10} par 2,

    Recommencer à 1 avec la partie décimale du résultat jusqu'à avoir 23 bits de mantisse (et même un peu plus pour arrondir)

Real 2 float

Triplet (s_2, e_2, m_2) en base 2

- Exemple : $X = -9.6$; $s_{10} = 1$; $9.6 \times 2^{-3} = 1.2 \in [1; 2[$ donc $e_{10} = 3$; $m_{10} = 1.2 - 1$.
- Connaissant s, e, m de la notation scientifique décimale, on veut s_2, e_2, m_2 tels que
  - bit de signe $s_2 = s_{10}$,
  - bits d'exposant $e_2 = \text{bin}(e + 127)$ (conversion de l'exposant *décalé* en base 2),
  - Pour la mantisse sur 23 bits
    - I Multiplier m_{10} par 2,
    - II Calculer partie entière (0 ou 1) ; partie décimale.

    Recommencer à I avec la partie décimale du résultat jusqu'à avoir 23 bits de mantisse (et même un peu plus pour arrondir)

Real 2 float

Triplet (s_2, e_2, m_2) en base 2

- Exemple : $X = -9.6$; $s_{10} = 1$; $9.6 \times 2^{-3} = 1.2 \in [1; 2[$ donc $e_{10} = 3$; $m_{10} = 1.2 - 1$.
- Connaissant s, e, m de la notation scientifique décimale, on veut s_2, e_2, m_2 tels que
  - bit de signe $s_2 = s_{10}$,
  - bits d'exposant $e_2 = \text{bin}(e + 127)$ (conversion de l'exposant *décalé* en base 2),
  - Pour la mantisse sur 23 bits
    - I Multiplier m_{10} par 2,
    - II Calculer partie entière (0 ou 1) ; partie décimale.
    - III partie entière : un nouveau bit de la représentation.

    Recommencer à I avec la partie décimale du résultat jusqu'à avoir 23 bits de mantisse (et même un peu plus pour arrondir)

Real 2 float

Exemple

- $X = -9.6,\ s_{10} = 1,\ e_{10} = 3,\ m_{10} = 0.2.$

Real 2 float

Exemple

- $X = -9.6$, $s_{10} = 1$, $e_{10} = 3$, $m_{10} = 0.2$.
- $e_{10} + 127_{10} = 130_{10}$, en base 2 : $e_2 = 10000010_2$

Real 2 float

Exemple

- $X = -9.6$, $s_{10} = 1$, $e_{10} = 3$, $m_{10} = 0.2$.
- $e_{10} + 127_{10} = 130_{10}$, en base 2 : $e_2 = 10000010_2$
- Pour la mantisse :

Real 2 float

Exemple

- $X = -9.6$, $s_{10} = 1$, $e_{10} = 3$, $m_{10} = 0.2$.
- $e_{10} + 127_{10} = 130_{10}$, en base 2 : $e_2 = 10000010_2$
- Pour la mantisse :
  1. $0.2 \times 2 = 0.4 = 0 + 0.4$

Real 2 float

Exemple

- $X = -9.6$, $s_{10} = 1$, $e_{10} = 3$, $m_{10} = 0.2$.
- $e_{10} + 127_{10} = 130_{10}$, en base 2 : $e_2 = 10000010_2$
- Pour la mantisse :
  1. $0.2 \times 2 = 0.4 = 0 + 0.4$
  2. $0.4 \times 2 = 0.8 = 0 + 0.8$

Real 2 float

Exemple

- $X = -9.6$, $s_{10} = 1$, $e_{10} = 3$, $m_{10} = 0.2$.
- $e_{10} + 127_{10} = 130_{10}$, en base 2 : $e_2 = 10000010_2$
- Pour la mantisse :
  1. $0.2 \times 2 = 0.4 = 0 + 0.4$
  2. $0.4 \times 2 = 0.8 = 0 + 0.8$
  3. $0.8 \times 2 = 1.6 = 1 + 0.6$

Real 2 float

Exemple

- $X = -9.6$, $s_{10} = 1$, $e_{10} = 3$, $m_{10} = 0.2$.
- $e_{10} + 127_{10} = 130_{10}$, en base 2 : $e_2 = 10000010_2$
- Pour la mantisse :
  1. $0.2 \times 2 = 0.4 = 0 + 0.4$
  2. $0.4 \times 2 = 0.8 = 0 + 0.8$
  3. $0.8 \times 2 = 1.6 = 1 + 0.6$
  4. $0.6 \times 2 = 1.2 = 1 + 0.2$. On est revenu à 0.2 : séquence infinie.
     Mantisse : **0011 0011 0011 0011 0011 0011 0011**....

Real 2 float

Exemple

-

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}
0	0	1	1	0	0	1	1	0	0
2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}	2^{-17}	2^{-18}	2^{-19}	2^{-20}
1	1	0	0	1	1	0	0	1	1
2^{-21}	2^{-22}	2^{-23}	2^{-24}	2^{-25}	2^{-26}	2^{-27}	...		
0	0	1	1	0	0	1	...		

Real 2 float

Exemple

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}
0	0	1	1	0	0	1	1	0	0
2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}	2^{-17}	2^{-18}	2^{-19}	2^{-20}
1	1	0	0	1	1	0	0	1	1
2^{-21}	2^{-22}	2^{-23}	2^{-24}	2^{-25}	2^{-26}	2^{-27}	...		
0	0	1	1	0	0	1	...		

- On a donc la mantisse comme une somme infinie de coefficients $\sum_{i=1}^{+\infty} \frac{a_i}{2^i}$ mais on veut une somme finie.

Real 2 float

Exemple

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}
0	0	1	1	0	0	1	1	0	0
2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}	2^{-17}	2^{-18}	2^{-19}	2^{-20}
1	1	0	0	1	1	0	0	1	1
2^{-21}	2^{-22}	2^{-23}	2^{-24}	2^{-25}	2^{-26}	2^{-27}	...		
0	0	1	1	0	0	1	...		

- On a donc la mantisse comme une somme infinie de coefficients $\sum_{i=1}^{+\infty} \frac{a_i}{2^i}$ mais on veut une somme finie.
- Question : $\frac{1}{2^{24}} + 0 \times \frac{1}{2^{25}} + 0 \times \frac{1}{2^{26}} + 1 \times \frac{1}{2^{27}}$ est-il plus proche de $\frac{1}{2^{23}}$ ou de 0 ? Donc m est-il plus proche de $\sum_{i=1}^{23} \frac{a_i}{2^i}$ ou de $\left(\sum_{i=1}^{23} \frac{a_i}{2^i}\right) + \frac{1}{2^{24}} + \frac{1}{2^{27}}$?

Real 2 float

Exemple

- $\frac{1}{2^{24}} + 0 \cdot \frac{1}{2^{25}} + 0 \cdot \frac{1}{2^{26}} + 1 \cdot \frac{1}{2^{27}}$ est plus proche de $\frac{1}{2^{23}}$: on ajoute donc 1 au bit de poids faible (le 23ème) et on tient compte des retenues.
  Attention : Dans le pire cas (mais pas ici), $m = \sum_{i=1}^{23} \frac{1}{2^i}$ et on ajoute $\frac{1}{2^{23}}$. Alors $m + \frac{1}{2^{23}} = \frac{\frac{1}{2} - \frac{1}{2^{24}}}{1 - 1\frac{1}{2}} + \frac{1}{2^{23}} = 1$ donc $1 + m = 2$ et il faut changer l'exposant ! !

Real 2 float

Exemple

- $\frac{1}{2^{24}} + 0 \cdot \frac{1}{2^{25}} + 0 \cdot \frac{1}{2^{26}} + 1 \cdot \frac{1}{2^{27}}$ est plus proche de $\frac{1}{2^{23}}$: on ajoute donc 1 au bit de poids faible (le 23ème) et on tient compte des retenues.
  Attention : Dans le pire cas (mais pas ici), $m = \sum_{i=1}^{23} \frac{1}{2^i}$ et on ajoute $\frac{1}{2^{23}}$. Alors $m + \frac{1}{2^{23}} = \frac{\frac{1}{2} - \frac{1}{2^{24}}}{1 - 1\frac{1}{2}} + \frac{1}{2^{23}} = 1$ donc $1 + m = 2$ et il faut changer l'exposant ! !
- Arrondi au plus proche sur 23 bits :

$$M = 0011\,0011\,0011\,0011\,0011\,010$$

Real 2 float

Exemple

- $\frac{1}{2^{24}} + 0 \cdot \frac{1}{2^{25}} + 0 \cdot \frac{1}{2^{26}} + 1 \cdot \frac{1}{2^{27}}$ est plus proche de $\frac{1}{2^{23}}$: on ajoute donc 1 au bit de poids faible (le 23ème) et on tient compte des retenues.
  Attention : Dans le pire cas (mais pas ici), $m = \sum_{i=1}^{23} \frac{1}{2^i}$ et on ajoute $\frac{1}{2^{23}}$. Alors $m + \frac{1}{2^{23}} = \frac{\frac{1}{2} - \frac{1}{2^{24}}}{1 - 1\frac{1}{2}} + \frac{1}{2^{23}} = 1$ donc $1 + m = 2$ et il faut changer l'exposant ! !
- Arrondi au plus proche sur 23 bits :

$$M = 0011\,0011\,0011\,0011\,0011\,010$$

- Finalement : -9.6_{10} est représenté par

$$1_2\ 10000010_2\ 0011\,0011\,0011\,0011\,0011\,010_2$$

Real 2 float : arrondi si $M = 23$

- Dans l'exemple étudié $X = -9.6$, on pousse le calcul des décimales jusqu'au 27ème bit et on peut déterminer sans erreur quel est l'arrondi au plus proche de la matrice.

Real 2 float : arrondi si $M = 23$

- Dans l'exemple étudié $X = -9.6$, on pousse le calcul des décimales jusqu'au 27ème bit et on peut déterminer sans erreur quel est l'arrondi au plus proche de la matrice.
- Les bits 24,25,26 s'écrivent en effet **100** et leur connaissance seule ne permet pas de trancher la question : faut-il arrondir par défaut ou par excès (exactement comme arrondir en base 10 le nombre **9.5** à l'unité au plus proche a 2 réponses).

Real 2 float : arrondi si $M = 23$

- Dans l'exemple étudié $X = -9.6$, on pousse le calcul des décimales jusqu'au 27ème bit et on peut déterminer sans erreur quel est l'arrondi au plus proche de la matrice.
- Les bits 24,25,26 s'écrivent en effet **100** et leur connaissance seule ne permet pas de trancher la question : faut-il arrondir par défaut ou par excès (exactement comme arrondir en base 10 le nombre **9.5** à l'unité au plus proche a 2 réponses).
- Dans nos exemples, on ne pousse pas les calculs trop loin après le dernier bit maintenu (le 23ème ici) et on préfère calculer seulement 3 bits supplémentaires. Le cas où ces 3 bits s'écrivent **100** est géré par la règle dite de l'*arrondi au plus proche pair* (cf. plus loin).

Real 2 float : arrondi si $M = 23$

- Dans l'exemple étudié $X = -9.6$, on pousse le calcul des décimales jusqu'au 27ème bit et on peut déterminer sans erreur quel est l'arrondi au plus proche de la matrice.
- Les bits 24,25,26 s'écrivent en effet **100** et leur connaissance seule ne permet pas de trancher la question : faut-il arrondir par défaut ou par excès (exactement comme arrondir en base 10 le nombre **9.5** à l'unité au plus proche a 2 réponses).
- Dans nos exemples, on ne pousse pas les calculs trop loin après le dernier bit maintenu (le 23ème ici) et on préfère calculer seulement 3 bits supplémentaires. Le cas où ces 3 bits s'écrivent **100** est géré par la règle dite de l'*arrondi au plus proche pair* (cf. plus loin).
- Cas dégénéré : On peut aussi arrondir sans utiliser les bits au delà du 23ème : (cf. arrondi au plus proche pair).

Float 2 real

Soit le flottant 0100 0000 1011 1000 0000 0000 0000 0000. À quel réel correspond-il ?

Signe	Exposant décalé	Bit caché + mantisse
0	1000 0001	(1) 011 1000 0000 0000 0000 0000

Float 2 real

Soit le flottant 0100 0000 1011 1000 0000 0000 0000 0000. À quel réel correspond-il ?

Signe	Exposant décalé	Bit caché + mantisse
0	1000 0001	(1) 011 1000 0000 0000 0000 0000
- Le signe est 0, le nombre est donc positif. Le champ exposant décalé est $e_2 = 10000001_2$, autrement dit $e_{10} = 129$. La valeur réelle de l'exposant est donc $e_{10} - d = 129 - 127 = 2$. Le significande (donc avec le bit implicite) est 1.01110000000000000000_2.

Float 2 real

Soit le flottant 0100 0000 1011 1000 0000 0000 0000 0000. À quel réel correspond-il ?

Signe	Exposant décalé	Bit caché + mantisse
0	1000 0001	(1) 011 1000 0000 0000 0000 0000
- Le signe est 0, le nombre est donc positif. Le champ exposant décalé est $e_2 = 10000001_2$, autrement dit $e_{10} = 129$. La valeur réelle de l'exposant est donc $e_{10} - d = 129 - 127 = 2$. Le significande (donc avec le bit implicite) est 1.01110000000000000000_2.
- Conversion :

$$(-1)^0 \times 2^2 \times (\underbrace{\mathbf{1}}_{\text{implicite}} \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}) = 5.75$$

float 2 real

Plus généralement, partant d'un flottant simple précision normalisé :

- L'écrire en binaire et retrouver chaque champs.

float 2 real

Plus généralement, partant d'un flottant simple précision normalisé :

- L'écrire en binaire et retrouver chaque champs.
- Signe s : bit de poids fort.

float 2 real

Plus généralement, partant d'un flottant simple précision normalisé :

- L'écrire en binaire et retrouver chaque champs.
- Signe s : bit de poids fort.
- Convertir le binaire du champs exposant en un entier e, lui retrancher le décalage $d = 127$ $(2^{8-1} - 1)$.

float 2 real

Plus généralement, partant d'un flottant simple précision normalisé :

- L'écrire en binaire et retrouver chaque champs.
- Signe s : bit de poids fort.
- Convertir le binaire du champs exposant en un entier e, lui retrancher le décalage $d = 127$ $(2^{8-1} - 1)$.
- Partie décimale m (indiquée par la mantisse $b_1 b_2 \dots b_{22} b_{23}$).

$$m = \sum_{i=0}^{22} b_i 2^{-i} = b_1 2^{-1} + b_2 2^{-2} + \dots b_{22} 2^{-22} + b_{23} 2^{-23}.$$

float 2 real

Plus généralement, partant d'un flottant simple précision normalisé :

- L'écrire en binaire et retrouver chaque champs.
- Signe s : bit de poids fort.
- Convertir le binaire du champs exposant en un entier e, lui retrancher le décalage $d = 127$ $(2^{8-1} - 1)$.
- Partie décimale m (indiquée par la mantisse $b_1 b_2 \dots b_{22} b_{23}$).

$$m = \sum_{i=0}^{22} b_i 2^{-i} = b_1 2^{-1} + b_2 2^{-2} + \dots b_{22} 2^{-22} + b_{23} 2^{-23}.$$

- Le nombre réel correspondant est $(-1)^s(\underbrace{\mathbf{1}}_{\text{bit implicite}} + m)2^{e-127}$.

Portée

FIGURE – Quelques nombres positifs (Wikipedia)

Type	Exposant	Mantisse	Valeur approchée	Écart / préc
Zéro	0000 0000	000 0000 0000 0000 0000 0000	0,0	
Plus petit **nombre dénormalisé**	0000 0000	000 0000 0000 0000 0000 0001	$1{,}4\times10^{-45}$	$1{,}4\times10^{-45}$
Nombre dénormalisé suivant	0000 0000	000 0000 0000 0000 0000 0010	$2{,}8\times10^{-45}$	$1{,}4\times10^{-45}$
Nombre dénormalisé suivant	0000 0000	000 0000 0000 0000 0000 0011	$4{,}2\times10^{-45}$	$1{,}4\times10^{-45}$
Autre nombre dénormalisé	0000 0000	100 0000 0000 0000 0000 0000	$5{,}9\times10^{-39}$	
Plus grand nombre dénormalisé	0000 0000	111 1111 1111 1111 1111 1111	1.17549421×10^{-38}	
Plus petit **nombre normalisé**	0000 0001	000 0000 0000 0000 0000 0000	1.17549435×10^{-38}	$1{,}4\times10^{-45}$
Nombre normalisé suivant	0000 0001	000 0000 0000 0000 0000 0001	1.17549449×10^{-38}	$1{,}4\times10^{-45}$
Presque le double	0000 0001	111 1111 1111 1111 1111 1111	$2{,}35098856\times10^{-38}$	$1{,}4\times10^{-45}$
Nombre normalisé suivant	0000 0010	000 0000 0000 0000 0000 0000	$2{,}35098870\times10^{-38}$	$1{,}4\times10^{-45}$
Nombre normalisé suivant	0000 0010	000 0000 0000 0000 0000 0001	2.35098898×10^{-38}	$2{,}8\times10^{-45}$
Presque 1	0111 1110	111 1111 1111 1111 1111 1111	0,99999994	$0{,}6\times10^{-7}$
1	0111 1111	000 0000 0000 0000 0000 0000	1,00000000	
Nombre suivant 1	0111 1111	000 0000 0000 0000 0000 0001	1,00000012	$1{,}2\times10^{-7}$
Presque le plus grand nombre	1111 1110	111 1111 1111 1111 1111 1110	$3{,}40282326\times10^{38}$	
Plus grand nombre normalisé	1111 1110	111 1111 1111 1111 1111 1111	$3{,}40282346\times10^{38}$	2×10^{31}

Un exemple d'arrondi au plus proche

- Soit un nombre de mantisse $1101100000000\ldots$.

Un exemple d'arrondi au plus proche

- Soit un nombre de mantisse 1101100000000 . . . .
- Le significande complet avec bit caché est 1.1101100 . . . .

Un exemple d'arrondi au plus proche

- Soit un nombre de mantisse $1101100000000\ldots$.
- Le significande complet avec bit caché est $1.1101100\ldots$.
- On veut l'arrondir à 3 chiffres après la virgule. On a le choix entre 1.110 ou $1.110 + 0.001 = 1.111$.

Un exemple d'arrondi au plus proche

- Soit un nombre de mantisse $1101100000000\ldots$.

Un exemple d'arrondi au plus proche

- Soit un nombre de mantisse $1101100000000\ldots$.
-

$$2^0 + \frac{1}{2} + \frac{1}{2^2} + 0 + \underbrace{\frac{1}{2^4} + \frac{1}{2^5}}_{\text{partie à arrondir}} \sim 111011$$

$$\text{Arrondi par défaut : } 2^0 + \frac{1}{2^1} + \frac{1}{2^2} + 0 \sim 1.110$$

$$\text{Arrondi par excès : } 2^0 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} \sim 1.111.$$

Un exemple d'arrondi au plus proche

- Soit un nombre de mantisse $1101100000000\ldots$.
- $$2^0 + \frac{1}{2} + \frac{1}{2^2} + 0 + \underbrace{\frac{1}{2^4} + \frac{1}{2^5}}_{\text{partie à arrondir}} \sim 111011$$
  $$\text{Arrondi par défaut : } 2^0 + \frac{1}{2^1} + \frac{1}{2^2} + 0 \sim 1.110$$
  $$\text{Arrondi par excès : } 2^0 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} \sim 1.111.$$
- $r = \frac{1}{2^4} + \frac{1}{2^5}$ est-il plus proche de 0 ou de $\frac{1}{2^3}$?

Un exemple d'arrondi au plus proche

- Soit un nombre de mantisse $1101100000000\ldots$.
-

$$2^0 + \frac{1}{2} + \frac{1}{2^2} + 0 + \underbrace{\frac{1}{2^4} + \frac{1}{2^5}}_{\text{partie à arrondir}} \sim 111011$$

$$\text{Arrondi par défaut : } 2^0 + \frac{1}{2^1} + \frac{1}{2^2} + 0 \sim 1.110$$

$$\text{Arrondi par excès : } 2^0 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} \sim 1.111.$$

- $r = \frac{1}{2^4} + \frac{1}{2^5}$ est-il plus proche de 0 ou de $\frac{1}{2^3}$?
- Plus proche de $\frac{1}{2^3}$.

Un exemple d'arrondi au plus proche

- Soit un nombre de mantisse $1101100000000\ldots$.
-

$$2^0 + \frac{1}{2} + \frac{1}{2^2} + 0 + \underbrace{\frac{1}{2^4} + \frac{1}{2^5}}_{\text{partie à arrondir}} \sim 111011$$

$$\text{Arrondi par défaut : } 2^0 + \frac{1}{2^1} + \frac{1}{2^2} + 0 \sim 1.110$$

$$\text{Arrondi par excès : } 2^0 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} \sim 1.111.$$

- $r = \frac{1}{2^4} + \frac{1}{2^5}$ est-il plus proche de 0 ou de $\frac{1}{2^3}$?
- Plus proche de $\frac{1}{2^3}$.
- Réponse : 1.111.

Choix du nombre le plus proche

Cas de l'examen de 3 bits après le dernier maintenu

- L'exemple précédent était facile car il y avait une seule réponse possible.

Choix du nombre le plus proche

Cas de l'examen de 3 bits après le dernier maintenu

- L'exemple précédent était facile car il y avait une seule réponse possible.
- Mais que se passe-t-il quand on a le choix ? Par exemple, en base 10, comment arrondir à l'unité 1.500 qui est aussi proche de 1 que de 2 ? Arrondir au plus proche pair signifie choisir 2 plutôt que 1.

Choix du nombre le plus proche

Cas de l'examen de 3 bits après le dernier maintenu

- L'exemple précédent était facile car il y avait une seule réponse possible.
- Mais que se passe-t-il quand on a le choix ? Par exemple, en base 10, comment arrondir à l'unité 1.500 qui est aussi proche de 1 que de 2 ? Arrondir au plus proche pair signifie choisir 2 plutôt que 1.
- En base deux, le problème se pose lorsque le nombre après le dernier bit maintenu est 100 (si 3 bits au delà du dernier maintenu).

Choix du nombre le plus proche

Cas de l'examen de 3 bits après le dernier maintenu

- L'exemple précédent était facile car il y avait une seule réponse possible.
- Mais que se passe-t-il quand on a le choix ? Par exemple, en base 10, comment arrondir à l'unité 1.500 qui est aussi proche de 1 que de 2 ? Arrondir au plus proche pair signifie choisir 2 plutôt que 1.
- En base deux, le problème se pose lorsque le nombre après le dernier bit maintenu est 100 (si 3 bits au delà du dernier maintenu).
- En base deux, on prend en général *l'arrondi au plus proche pair*. Il faut que le dernier chiffre de l'écriture binaire soit pair.

Choix du nombre le plus proche

Cas de l'examen de 3 bits après le dernier maintenu

- L'exemple précédent était facile car il y avait une seule réponse possible.
- Mais que se passe-t-il quand on a le choix ? Par exemple, en base 10, comment arrondir à l'unité 1.500 qui est aussi proche de 1 que de 2 ? Arrondir au plus proche pair signifie choisir 2 plutôt que 1.
- En base deux, le problème se pose lorsque le nombre après le dernier bit maintenu est 100 (si 3 bits au delà du dernier maintenu).
- En base deux, on prend en général *l'arrondi au plus proche pair*. Il faut que le dernier chiffre de l'écriture binaire soit pair.
- Arrondir au plus proche pair revient, lorsqu'on a le choix, à privilégier les écritures qui se terminent par 0.

Exemples d'arrondis au plus proche pair

- Arrondir 1.100100 à 3 chiffres après la virgule : plus proche pair 1.100 (0 est pair).

Exemples d'arrondis au plus proche pair

- Arrondir 1.100100 à 3 chiffres après la virgule : plus proche pair 1.100 (0 est pair).
- Arrondir 1.101100 à 3 chiffres après la virgule. 1 est impair. Plus proche pair : $1.101 + 0.001 = 1.110$.

Exemples d'arrondis au plus proche pair

- Arrondir 1.100100 à 3 chiffres après la virgule : plus proche pair 1.100 (0 est pair).
- Arrondir 1.101100 à 3 chiffres après la virgule. 1 est impair. Plus proche pair : $1.101 + 0.001 = 1.110$.
- Arrondir 1.111100 à 3 chiffres après la virgule. 1 est impair. Plus proche pair : $1.111 + 0.001 = 10.000$. Il faut changer l'exposant (ajouter 1 à l'exposant) ! !

Exemples d'arrondis au plus proche pair

- Arrondir 1.100100 à 3 chiffres après la virgule : plus proche pair 1.100 (0 est pair).
- Arrondir 1.101100 à 3 chiffres après la virgule. 1 est impair. Plus proche pair : $1.101 + 0.001 = 1.110$.
- Arrondir 1.111100 à 3 chiffres après la virgule. 1 est impair. Plus proche pair : $1.111 + 0.001 = 10.000$. Il faut changer l'exposant (ajouter 1 à l'exposant) ! !
- Lorsqu'on est dans le cas de figure où il faut changer l'exposant, et que l'exposant est lui même maximum (254=127+127 pour les nombres sur 32 bits), on se retrouve avec un nombre considéré comme infini !

Règle d'arrondi au plus proche pair

- Exemple d'arrondi au troisième chiffre après la virgule :

$$1.01110011 = \underbrace{1.011}_{\text{bits maintenus}} \overbrace{10011}^{\text{bits tronqués}} .$$

Règle d'arrondi au plus proche pair

- Exemple d'arrondi au troisième chiffre après la virgule :

$$1.01110011 = \underbrace{1.011}_{\text{bits maintenus}} \overbrace{10011}^{\text{bits tronqués}} .$$

- On considère les trois chiffres après le dernier bit maintenu.

Règle d'arrondi au plus proche pair

- Exemple d'arrondi au troisième chiffre après la virgule :

$$1.01110011 = \underbrace{1.011}_{\text{bits maintenus}} \overbrace{10011}^{\text{bits tronqués}} .$$

- On considère les trois chiffres après le dernier bit maintenu.
  1. **0xy** : juste tronquer l'expression (**x,y** sont quelconques).

Règle d'arrondi au plus proche pair

- Exemple d'arrondi au troisième chiffre après la virgule :

$$1.01110011 = \underbrace{1.011}_{\text{bits maintenus}} \overbrace{10011}^{\text{bits tronqués}} .$$

- On considère les trois chiffres après le dernier bit maintenu.
  1. **0xy** : juste tronquer l'expression (**x,y** sont quelconques).
  2. **100**. Arrondir au plus proche pair :

Règle d'arrondi au plus proche pair

- Exemple d'arrondi au troisième chiffre après la virgule :

$$1.01110011 = \underbrace{1.011}_{\text{bits maintenus}} \overbrace{10011}^{\text{bits tronqués}} .$$

- On considère les trois chiffres après le dernier bit maintenu.
  1. **0xy** : juste tronquer l'expression (**x,y** sont quelconques).
  2. **100**. Arrondir au plus proche pair :
     - Si le dernier bit mantenu vaut 0 : ne rien faire.

Règle d'arrondi au plus proche pair

- Exemple d'arrondi au troisième chiffre après la virgule :

$$1.01110011 = \underbrace{1.011}_{\text{bits maintenus}} \overbrace{10011}^{\text{bits tronqués}} .$$

- On considère les trois chiffres après le dernier bit maintenu.
  1. **0xy** : juste tronquer l'expression (**x,y** sont quelconques).
  2. **100**. Arrondir au plus proche pair :
     - Si le dernier bit mantenu vaut 0 : ne rien faire.
     - Sinon, ajouter 1 au dernier bit maintenu en tenant compte des retenus.

Règle d'arrondi au plus proche pair

- Exemple d'arrondi au troisième chiffre après la virgule :

$$1.01110011 = \underbrace{1.011}_{\text{bits maintenus}} \overbrace{10011}^{\text{bits tronqués}} .$$

- On considère les trois chiffres après le dernier bit maintenu.
  1. **0xy** : juste tronquer l'expression (**x,y** sont quelconques).
  2. **100**. Arrondir au plus proche pair :
     - Si le dernier bit mantenu vaut 0 : ne rien faire.
     - Sinon, ajouter 1 au dernier bit maintenu en tenant compte des retenus.
  3. **1xy**, avec $x + y > 0$: ajouter 1 au dernier bit maintenu.

Règle d'arrondi au plus proche pair

- Exemple d'arrondi au troisième chiffre après la virgule :

$$1.01110011 = \underbrace{1.011}_{\text{bits maintenus}} \overbrace{10011}^{\text{bits tronqués}} .$$

- On considère les trois chiffres après le dernier bit maintenu.
  1. **0xy** : juste tronquer l'expression (**x,y** sont quelconques).
  2. **100**. Arrondir au plus proche pair :
     - Si le dernier bit mantenu vaut 0 : ne rien faire.
     - Sinon, ajouter 1 au dernier bit maintenu en tenant compte des retenus.
  3. **1xy**, avec $x + y > 0$: ajouter 1 au dernier bit maintenu.
- Dans l'exemple, les trois chiffres après le dernier bit maintenu forment 100 (équidistance). L'arrondi au plus proche pair est $1.011 + 0.001 = 1.100$.

Expressions infinies

- Les nombres flottants représentent des rationnels ayant une *expression finie*. Quid des expressions illimitées ?

Expressions infinies

- Les nombres flottants représentent des rationnels ayant une *expression finie*. Quid des expressions illimitées ?
- $\frac{1}{5} = 0.2$ admet une *expression finie en base 10* (EF10), mais pas $\frac{1}{3}$.

Expressions infinies

- Les nombres flottants représentent des rationnels ayant une *expression finie*. Quid des expressions illimitées ?
- $\frac{1}{5} = 0.2$ admet une *expression finie en base 10* (EF10), mais pas $\frac{1}{3}$.
- En base 2, $\frac{1}{a}$ admet une EF2 si et seulement si $a = 2^n$ $(n > 0)$.

Expressions infinies

- Les nombres flottants représentent des rationnels ayant une *expression finie*. Quid des expressions illimitées ?
- $\frac{1}{5} = 0.2$ admet une *expression finie en base 10* (EF10), mais pas $\frac{1}{3}$.
- En base 2, $\frac{1}{a}$ admet une EF2 si et seulement si $a = 2^n$ $(n > 0)$.
- $\frac{1}{10} = 0.1$ en base 10. Donc $\frac{1}{10}$ a une EF10 mais pas une EF2 car $10 = 2 \times 5$ et que 5 est premier avec 2.

Expressions infinies

- Les nombres flottants représentent des rationnels ayant une *expression finie*. Quid des expressions illimitées ?
- $\frac{1}{5} = 0.2$ admet une *expression finie en base 10* (EF10), mais pas $\frac{1}{3}$.
- En base 2, $\frac{1}{a}$ admet une EF2 si et seulement si $a = 2^n$ $(n > 0)$.
- $\frac{1}{10} = 0.1$ en base 10. Donc $\frac{1}{10}$ a une EF10 mais pas une EF2 car $10 = 2 \times 5$ et que 5 est premier avec 2.
- Il faut donc arrondir. Le standard IEEE-754 prévoit 5 méthodes.

Expressions infinies

- Les nombres flottants représentent des rationnels ayant une *expression finie*. Quid des expressions illimitées ?
- $\frac{1}{5} = 0.2$ admet une *expression finie en base 10* (EF10), mais pas $\frac{1}{3}$.
- En base 2, $\frac{1}{a}$ admet une EF2 si et seulement si $a = 2^n$ $(n > 0)$.
- $\frac{1}{10} = 0.1$ en base 10. Donc $\frac{1}{10}$ a une EF10 mais pas une EF2 car $10 = 2 \times 5$ et que 5 est premier avec 2.
- Il faut donc arrondir. Le standard IEEE-754 prévoit 5 méthodes.
- Règle de l'*Arrondi correct* : Une fois un mode d'arrondi choisi, le résultat d'une opération est déterministe : un seul résultat possible.

Un exemple d'arrondi

Un dixième

- 0.1 en base 10, correspond à la séquence suivante : $s = 0 = S$, $e = -4 + 127$ donc $E = 01111011$, M est une séquence infinie $1.1001\,1001\,1001\,1001\,1001\,100\,110\,0\ldots$ Alors $1 + m = 1.10011001100110011001101_2$.

Un exemple d'arrondi

Un dixième

- 0.1 en base 10, correspond à la séquence suivante : $s = 0 = S$, $e = -4 + 127$ donc $E = 01111011$, M est une séquence infinie $1.1001\,1001\,1001\,1001\,1001\,100\,110\,0\ldots$ Alors $1 + m = 1.10011001100110011001101_2$.
- Du fait des arrondis, le nombre qui représente 0.1 sur un ordinateur avec flottants 32 bits est en fait le nombre $\frac{13421773}{134217728}$ soit 0.100000001490116.

Règle de l'arrondi correct

La norme IEEE 754 impose l'arrondi correct pour les 5 opérations de base et la racine carrée : Un programme les utilisant donne le même résultat sur toute configuration (machine, système, processeur). Sous réserve :

1. qu'il 'y ait pas de précision intermédiaire étendue (ou alors désactivée). Ca veut dire que les résultats intermédiaires du calcul d'une expression ne doivent pas être calculés avec une précision plus grande que celle attendue pour le résultat.

Règle de l'arrondi correct

La norme IEEE 754 impose l'arrondi correct pour les 5 opérations de base et la racine carrée : Un programme les utilisant donne le même résultat sur toute configuration (machine, système, processeur). Sous réserve :

1. qu'il 'y ait pas de précision intermédiaire étendue (ou alors désactivée). Ca veut dire que les résultats intermédiaires du calcul d'une expression ne doivent pas être calculés avec une précision plus grande que celle attendue pour le résultat.
2. le compilateur ne doit pas changer l'ordre des opérations si cela peut conduire à un résultat différent.

Problème d'arrondi célèbre

Patriot

- En 1991, un anti-missile Patriot rate l'interception d'un missile irakien Skud, lequel blesse 100 personnes, en tue 28.

Problème d'arrondi célèbre

Patriot

- En 1991, un anti-missile Patriot rate l'interception d'un missile irakien Skud, lequel blesse 100 personnes, en tue 28.
- Un micro-processeur interne calcule l'heure en multiples de dixièmes de secondes.

Problème d'arrondi célèbre

Patriot

- En 1991, un anti-missile Patriot rate l'interception d'un missile irakien Skud, lequel blesse 100 personnes, en tue 28.
- 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.

Problème d'arrondi célèbre

Patriot

- En 1991, un anti-missile Patriot rate l'interception d'un missile irakien Skud, lequel blesse 100 personnes, en tue 28.
- 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 : $0.1 \simeq 209715 \cdot 2^{-21} = 0.0999999046325683 6$, erreur d'environ 10^{-7}.

Problème d'arrondi célèbre

Patriot

- En 1991, un anti-missile Patriot rate l'interception d'un missile irakien Skud, lequel blesse 100 personnes, en tue 28.
- 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 : $0.1 \simeq 209715 \cdot 2^{-21} = 0.0999999046325683 6$, erreur d'environ 10^{-7}.
- Processeur démarré 100h auparavant. Erreur de 0.34 secondes.

Problème d'arrondi célèbre

Patriot

- En 1991, un anti-missile Patriot rate l'interception d'un missile irakien Skud, lequel blesse 100 personnes, en tue 28.
- 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 : $0.1 \simeq 209715 \cdot 2^{-21} = 0.0999999046325683 6$, erreur d'environ 10^{-7}.
- Processeur démarré 100h auparavant. Erreur de 0.34 secondes.
- **Le système Patriot croit être à un instant t, mais il est en fait à $t - 0.34s$.**. Il utilise une position du Scud vieille de 0.34s. Pendant ce temps, le Skud parcourt 500 m.

Problème d'arrondi célèbre

Patriot

- En 1991, un anti-missile Patriot rate l'interception d'un missile irakien Skud, lequel blesse 100 personnes, en tue 28.
- 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 : $0.1 \simeq 209715 \cdot 2^{-21} = 0.0999999046325683 6$, erreur d'environ 10^{-7}.
- Processeur démarré 100h auparavant. Erreur de 0.34 secondes.
- **Le système Patriot croit être à un instant t, mais il est en fait à $t - 0.34s$.**. Il utilise une position du Scud vieille de 0.34s. Pendant ce temps, le Skud parcourt 500 m.
- Le Patriot rate sa cible, pas le Skud.

Double arrondi

Cas d'examen de 3 bits après le dernier maintenu

- Soit x réel, y l'arrondi en précision p de x.

Double arrondi

Cas d'examen de 3 bits après le dernier maintenu

- Soit x réel, y l'arrondi en précision p de x.
- Soit x' arrondi en précision $q < p$ de y.

Double arrondi

Cas d'examen de 3 bits après le dernier maintenu

- Soit x réel, y l'arrondi en précision p de x.
- Soit x' arrondi en précision $q < p$ de y.
- x' n'est pas toujours l'arrondi en précision q de x !

Double arrondi

Cas d'examen de 3 bits après le dernier maintenu

- Soit x réel, y l'arrondi en précision p de x.
- Soit x' arrondi en précision $q < p$ de y.
- x' n'est pas toujours l'arrondi en précision q de x !
  1. $x = 1.0110100\mathbf{0}101$. Arrondir à 7 chiffres après la virgule.

Double arrondi

Cas d'examen de 3 bits après le dernier maintenu

- Soit x réel, y l'arrondi en précision p de x.
- Soit x' arrondi en précision $q < p$ de y.
- x' n'est pas toujours l'arrondi en précision q de x !
  1. $x = 1.0110100101$. Arrondir à 7 chiffres après la virgule.
  2. Après la décimale 7 de x on a : 101. L'arrondi y à 7 chiffres après la virgule de x est : 1.0110101

Double arrondi

Cas d'examen de 3 bits après le dernier maintenu

- Soit x réel, y l'arrondi en précision p de x.
- Soit x' arrondi en précision $q < p$ de y.
- x' n'est pas toujours l'arrondi en précision q de x !
  1. $x = 1.0110100101$. Arrondir à 7 chiffres après la virgule.
  2. Après la décimale 7 de x on a : 101. L'arrondi y à 7 chiffres après la virgule de x est : 1.0110101
  3. Après la décimale 4 de y : 101. L'arrondi x' à 4 décimales de y est : 1.0111

Double arrondi

Cas d'examen de 3 bits après le dernier maintenu

- Soit x réel, y l'arrondi en précision p de x.
- Soit x' arrondi en précision $q < p$ de y.
- x' n'est pas toujours l'arrondi en précision q de x !
  1. $x = 1.0110100101$. Arrondir à 7 chiffres après la virgule.
  2. Après la décimale 7 de x on a : 101. L'arrondi y à 7 chiffres après la virgule de x est : 1.0110101
  3. Après la décimale 4 de y : 101. L'arrondi x' à 4 décimales de y est : 1.0111
  4. MAIS, après la décimale 4 de x on a 100. suivant la règle de l'arrondi pair, L'arrondi z à 4 décimales de x est 1.0110.

Double arrondi

Cas d'examen de 3 bits après le dernier maintenu

- Soit x réel, y l'arrondi en précision p de x.
- Soit x' arrondi en précision $q < p$ de y.
- x' n'est pas toujours l'arrondi en précision q de x !
  1. $x = 1.0110100101$. Arrondir à 7 chiffres après la virgule.
  2. Après la décimale 7 de x on a : 101. L'arrondi y à 7 chiffres après la virgule de x est : 1.0110101
  3. Après la décimale 4 de y : 101. L'arrondi x' à 4 décimales de y est : 1.0111
  4. MAIS, après la décimale 4 de x on a 100. suivant la règle de l'arrondi pair, L'arrondi z à 4 décimales de x est 1.0110.
  5. Et on a $z \neq x'$!

Double arrondi

Cas d'examen de 3 bits après le dernier maintenu

- Soit x réel, y l'arrondi en précision p de x.
- Soit x' arrondi en précision $q < p$ de y.
- x' n'est pas toujours l'arrondi en précision q de x !
  1. $x = 1.0110100101$. Arrondir à 7 chiffres après la virgule.
  2. Après la décimale 7 de x on a : 101. L'arrondi y à 7 chiffres après la virgule de x est : 1.0110101
  3. Après la décimale 4 de y : 101. L'arrondi x' à 4 décimales de y est : 1.0111
  4. MAIS, après la décimale 4 de x on a 100. suivant la règle de l'arrondi pair, L'arrondi z à 4 décimales de x est 1.0110.
  5. Et on a $z \neq x'$!
- On peut montrer que le problème du double arrondi n'intervient que si on choisit *l'arrondi au plus proche pair*. Pas de chance : c'est le mode d'arrondi le plus répandu !

Pourquoi privilégier l'arrondi au plus proche pair ?

- La méthode de « l'arrondi bancaire » (autre nom pour l'arrondi au plus proche pair) est employée pour éliminer le biais qui surviendrait en arrondissant à chaque fois par excès les nombres dont les trois derniers chiffres seraient 100.

Pourquoi privilégier l'arrondi au plus proche pair ?

- La méthode de « l'arrondi bancaire » (autre nom pour l'arrondi au plus proche pair) est employée pour éliminer le biais qui surviendrait en arrondissant à chaque fois par excès les nombres dont les trois derniers chiffres seraient 100.
- Transposons en base 10 : Imaginons une multinationale qui reçoit un milliards de virements exprimés en centimes d'euros (sur une certaine période) arrondis au dixième de centime sur un de ses comptes en banque.

Pourquoi privilégier l'arrondi au plus proche pair ?

- Supposons que pour un millième de ces virements, la partie fractionnaire soit de la forme $.x500$ où $x \in [\![0, 9]\!]$.

Pourquoi privilégier l'arrondi au plus proche pair ?

- Supposons que pour un millième de ces virements, la partie fractionnaire soit de la forme $.x500$ où $x \in [\![0, 9]\!]$.
- Si la banque arrondi le montant de ces virements au dixième de centime supérieur ($.x + 0.1$, puis répercussion de la retenue), la multinationale gagne 0.05 centime de plus par virement que ce qu'elle aurait dû toucher.
  Au total cela fait $10^6 \times 5 \times 10^{-2} = 5 \times 10^4 = 50\,000$ euros que la multinationale a gagnés au détriment de la banque ! Au centime inférieur, ce serait la banque qui gagnerait de l'argent.

Pourquoi privilégier l'arrondi au plus proche pair ?

- Supposons que pour un millième de ces virements, la partie fractionnaire soit de la forme $.x500$ où $x \in [\![0, 9]\!]$.
- Si la banque arrondi le montant de ces virements au dixième de centime supérieur ($.x + 0.1$, puis répercussion de la retenue), la multinationale gagne 0.05 centime de plus par virement que ce qu'elle aurait dû toucher.
  Au total cela fait $10^6 \times 5 \times 10^{-2} = 5 \times 10^4 = 50\,000$ euros que la multinationale a gagnés au détriment de la banque ! Au centime inférieur, ce serait la banque qui gagnerait de l'argent.
- D'où la nécessité d'arrondir certains montants au centime supérieur et d'autres au centime inférieur pour équilibrer, comme avec l'arrondi au plus proche pair.

Cas des exceptions

En cas de problème, la norme impose de signaler des *Exceptions*

- Diviser un nombre différent de 0 par 0 donne $\pm\infty$

Cas des exceptions

En cas de problème, la norme impose de signaler des *Exceptions*

- Diviser un nombre différent de 0 par 0 donne $\pm\infty$
- Diviser zéro par zéro, ou calculer le logarithme d'un nombre négatif conduisent à générer des **NaN** qu'on peut décider de considérer comme des exceptions.

Cas des exceptions

En cas de problème, la norme impose de signaler des *Exceptions*

- Diviser un nombre différent de 0 par 0 donne $\pm\infty$
- Diviser zéro par zéro, ou calculer le logarithme d'un nombre négatif conduisent à générer des **NaN** qu'on peut décider de considérer comme des exceptions.
- Nombre entier positif plus grand que le plus grand entier représentable (*overflow*). Ou plus petit que le plus petit entier représentable (*underflow*).

Python

- Entrons 0.1 dans l'interpréteur **Python** :

```
>>> 0.1
0.1
```

La valeur dans la machine n'est pas exactement 0.1. **Python** a simplement arrondi l'affichage du résultat.

- Autre illusion :

```
>>> 0.2
0.2
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1+0.2==0.3
False
```

Python

```
>>> round(2.675, 2)
2.67
```

On devrait avoir 2.68 (plus proche pair). Mais le flottant correspondant à 2.675 est (53 chiffres en **Python**) :

$$2.67499999999999982236431605997495353221893310546875$$

Donc arrondi à 2.67.

Python

- Pour savoir où on en est dans les arrondis, importer le module **Décimal** :

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.674999999999999822364316059974953
          5322189331054687 5')
```

- La somme de dix fois le représentant de 0.1, ne vaut pas tout à fait 1 :

```
>>> somme = 0.0
>>> for i in range(0,10):
...         somme += 0.1
...
>>> somme
0.9999999999999999
```

L'addition des flottants n'est pas compatible avec la relation d'ordre...

Arithmétique flottante En arithmétique flottante :

- L'égalité n'est pas réflexive : NaN $\neq$ NaN (hors-programme).
- L'addition et la multiplication sont commutatives :

$$a \diamond b = b \diamond a \text{ pour } \diamond \in \{+, \times\}.$$

- L'addition et la multiplication ne sont pas associatives :

```
>>> 0.1 + (0.2 - 0.3)
2.7755575615628914e-17
>>> (0.1 + 0.2) - 0.3
5.551115123125783e-17
>>> ((0.1 + 0.2) - 0.3) -(0.1+(0.2-0.3))
2.7755575615628914e-17
```

- La multiplication n'est pas distributive par rapport à l'addition :

```
>>> 0.1*(0.2+1) - (0.1*0.2 + 0.1*1)
-1.3877787807814457e-17
```

Epsilon Machine

- Définition : La plus petite valeur qui ajoutée à 1 donne un résultat différent de 1.

Epsilon Machine

- Définition : La plus petite valeur qui ajoutée à 1 donne un résultat différent de 1.
- La plus petite distance entre 1 et le suivant par la fonction `spacing` :

```
>>> import numpy as np
>>> np.finfo(np.float64).eps
2.220446049250313e-16
>>> np.spacing(1)# les deux fonctions sont equivalentes
2.220446049250313e-16
>>> np.spacing(0.2)#plus le nb est petit,
#plus le prochain nb est proche
2.7755575615628914e-17
>>> np.spacing(0)
4.940656458412465e-324
```

Epsilon Machine

- Définition : La plus petite valeur qui ajoutée à 1 donne un résultat différent de 1.
- La plus petite distance entre 1 et le suivant par la fonction `spacing` :
- Plus le nombre est petit (ie plus son exposant dans la représentation IEEE754 est petit), plus le plus proche nombre représentable est proche.

Epsilon Machine

- Définition : La plus petite valeur qui ajoutée à 1 donne un résultat différent de 1.
- La plus petite distance entre 1 et le suivant par la fonction `spacing` :
- Plus le nombre est petit (ie plus son exposant dans la représentation IEEE754 est petit), plus le plus proche nombre représentable est proche.
- On a des informations sur le plus petit nombre normalisé et le plus petit dénormalisé. cf `epsilon_machine.ipynb` .