Représentation des nombres réels

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Représentation du continu en informatique ?

Informatique <3 discret

- « Toute information est un nombre entier naturel »
- Adapté pour représenter des données discrètes
  - Concrètement finies, éventuellement conceptuellement infinies *dénombrables*/non bornées
  - Par ex. nombres de $\mathbb{Z}$, $\mathbb{Q}$,…

Alternative : modèles de calcul continus/analogiques 🙈

Représenter les éléments de $\mathbb{R}$?

- Problème 1 : $\mathbb{R}$ est « trop grand »
  - Tout réel *n'est pas* (associé à) un entier naturel
  - On ne pourra représenter qu'un « petit » sous-ensemble (dénombrable) de $\mathbb{R}$
- Problème 2 : comment représenter des réels « très proches » ?
  - Comment facilement comparer deux représentations ?

Représentation approchée de $\mathbb{R}$: approximation rationnelle

$\mathbb{Q}$ est dense dans $\mathbb{R}$

Idée : pour représenter $r \in \mathbb{R}$ *de façon approchée*

- On trouve un rationnel $q \in \mathbb{Q}$ « proche »
  - Caché sous le tapis : faisable seulement si r est *calculable*
  - Trouver une bonne valeur : problème d'*approximation rationnelle* (ou *diophantienne*) (domaine du calcul formel)
- On représente r par q…

Mesures d'approximation

Mesurer la qualité d'une approximation $\hat{x}$ de x :

- Erreur absolue : $|x - \hat{x}|$
- Erreur relative : $|x - \hat{x}|/x$
  - Souvent le plus pertinent, notamment pour des grandeurs physiques (pensez aux chiffres significatifs…)

Commentaires, exemples

- Arithmétique, représentation : « comme » avec $\mathbb{Z}$
- Comment comparer deux nombres ?
  - Pas directement avec les chiffres ; demande un calcul

Exemples

- $r = 1/3$
  - $q = 1/3$
  - de façon générale, toujours possible de représenter exactement un rationnel par un rationnel…
- $r = \sqrt{2}$
  - $q = 141/100$ (erreur absolue < 0.005)
  - $q = 99/70$ (— < 0.000073)
- $r = \pi$
  - $q = 314/100$ (— < 0.002)
  - $q = 355/113$ (— < 0.0000003)

Représentation approchée de $\mathbb{R}$: approximation décimale

$\mathbb{D}$ est dense dans $\mathbb{R}$

- Même chose qu'avec $\mathbb{Q}$, mais $d \in \mathbb{D}$ de la forme $n/10^h$

Différences avec l'approche $\mathbb{Q}$

- Plus possible de représenter *exactement* tous les rationnels
  - Exemple canonique : $1/3$
- Mais facile de comparer deux décimaux en « lisant leurs chiffres »

Avec un ordinateur

- Même chose, mais plutôt en base 2
- Même avantages/inconvénients qu'en base 10

Représentation approchée de $\mathbb{R}$: nombres à virgule fixe (en base 2)

Format

Représentations de la forme (x^e, x^f) :

- x^e : représentation en base 2 d'une partie entière sur h_e bits (**fixe**)
  - Représente le nombre $\Sigma_{i=0}^{h_e-1} x_i^e 2^i$
- x^f : représentation en base 2 d'une partie fractionnaire sur h_f bits (**fixe**)
  - Représente le nombre $\Sigma_{i=1}^{h_f} x_i^f 2^{-i}$
- (Signe à éventuellement gérer en plus)

Propriétés

- Conceptuellement simple
- Pas très flexible/efficace
  - Quels valeurs pour h_e, h_f ?
  - Compromis nécessaire (à nombre de bits fixé) entre représenter des nombres très grands/très petits (en valeur absolue)

Globalement pas utilisé

Représentation approchée de $\mathbb{R}$: nombres à virgule *flottante* (en base 2)

Format

On stocke :

- Un bit de signe
- Un *exposant* $e \in [\![-a, b]\!]$
- Une *mantisse* $m \in [\![0, s]\!]$

Pour représenter les (nombres) *flottants* de la forme $\pm 2^e \times m$

Propriétés

Utilisation d'un nombre (*a priori*) **fixe** de bits (mais on peut définir des flottants « multi-précision » )

- Un pour le signe
- Par ex. 11 pour l'exposant $e \in [\![-1024, 1023]\!]$ (si en complément à 2)
- Par ex. 52 pour la mantisse $m \in [\![0, 2^{52} - 1]\!]$

Permet de représenter des nombres de valeur absolue entre $\approx 2^{-1024}$ et $\approx 2^{1076}$

- Mais pas *tous* les nombres (réels) de cet intervalle, avec possiblement de grands « trous »

Nombres flottants : format (suite)

Unicité de représentation 🙈

- On peut *a priori* représenter un même nombre de plusieurs façons
  - Par ex. $4 = 2^2 \times 1 = 2^1 \times 2 = 2^{-48} \times 2^{50}$
- Un choix unique possible : utiliser *le* représentant possible avec l'exposant le plus *petit*

Zéros signés

Représentation de zéro :

- `0.0` : bit de signe, exposant, mantisse tout à zéro
- `-0.0` : bit de signe à 1; exposant, mantisse tout à zéro

En général, les flottants utilisés par les langages de programmation sont t.q. `0.0` et `-0.0` sont considérés égaux, mais il y a bien *deux* représentations *physiques* différentes

Nombres flottants : **représentation approchée** de nombres « réels »

Représentation approchée

- Les flottants servent (avant tout) à représenter des nombres de façon approchée (même si on *peut* aussi être exacte (ce qui peut être pratique pour calculer plus vite qu'avec des entiers 🙈))
- Situation « moins claire » qu'avec des entiers (où le résultat d'un calcul est **exact** ou « complètement faux » (voire indéfini))

Exemples

```
utop # 2.0**52.0 +. 1.0 = 2.0**52.0;;
- : bool = false
utop # 2.0**53.0 +. 1.0 = 2.0**53.0;;
- : bool = true
utop # 0.1 +. 0.2 = 0.3;;
- : bool = false
```

Nombres flottants : suite des problèmes

Le successeur d'un flottant ne lui est pas forcément différent

Non-associativité

Les opérations sur les nombres flottants sont (parfois) non-associatives :

```
utop # (2.**53. +. (1. -. 2.)) = ((2.**53. +. 1.) -. 2.);;
- : bool = false
```

Non-distributivité

Lors d'un calcul avec des nombres flottants, la multiplication ne distribue pas (toujours) sur l'addition :

```
utop # 3. *. (0.333333333333333 +. 0.00000000000000031);;
- : float = 1.
utop # 3. *. 0.333333333333333 +. 3. *. 0.00000000000000031;;
- : float = 0.999999999999999778
```

Nombres flottants : encore des problèmes

La *comparaison* de flottants est… délicate

- Distinguer au-delà de la précision de représentation n'a jamais de sens
  - Deux représentations physiques identiques se comparent identiquement
  - On ne peut pas distinguer deux nombres « trop proches » (relativement)
  - *Cf.* `2.**53.` et `2.**53. +. 1.`
- Les approximations de calcul ~~doivent~~ devraient être prises en compte
  - *Cf.* :

```
utop # 0.1 +. 0.2;;
- : float = 0.30000000000000044
utop # ((0.1 +. 0.2) -. 0.3) < 0.0000000000000001;; (* « égaux » *)
- : bool = true
```

Nombre flottants : « comparaison »

Une approche générale : comparer (absolument) à une précision donnée

```
let compare_approx eps x y = abs_float (x -. y) < eps
```

- Deux nombres « suffisamment proches » sont considérés égaux
  - Même si les représentations physiques *sont* différentes
- Mais que veut dire « proche » (quel `eps`) ?
  - Prend en compte l'erreur inhérente à la représentation
  - Si pertinent, adapté à la précision avec laquelle les données sont connues (prise en compte des chiffres significatifs…)
  - *Devrait* nécessiter de borner l'impact des erreurs d'approximation lors de l'enchaînement des calculs (pas pour nous…)
- Un compromis : `eps` donné par la moitié des bits de la mantisse (par ex. $\approx 10^{-8}$ pour les flottants usuels), mais :

  « Il n'y a aucun choix qui soit réellement simple et défendable »

Nombres flottants : précautions

Calcul approché $\neq$ calcul exact

- Par défaut : ne pas s'attendre à un calcul exact
- Ne pas utiliser des flottants pour calculer « dans $\mathbb{R}$ » sans réfléchir
  - Risque de donner un mauvais résultat (tout simplement), sans qu'il soit toujours facile de s'en rendre compte
  - Cf. un prochain TP (avec des lapins)

Exemple : critère d'arrêt dans un schéma d'approximation numérique

- Absurde de faire un nombre d'itérations donnant une précision théorique supérieure à celle des flottants
- Deux méthodes différentes peuvent converger vers des flottants *différents* **tous deux corrects**
  - « La comparaison de flottants est… délicate »

Divergence catastrophique possible

- Quand des erreurs d'approximation s'amplifient hors de tout contrôle !

Divergence catastrophique : exemple

$u_0 = e$; $u_i = (u_{i-1} - 1) \times i$

- Dans $\mathbb{R}$, $u_i \to_\infty 1$
- Avec des flottants…

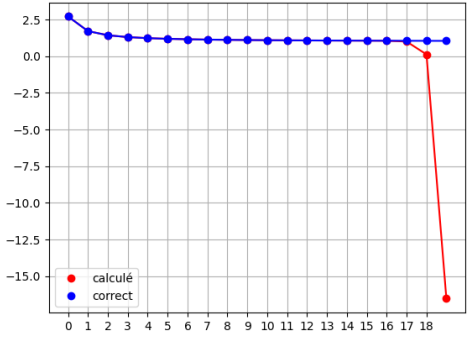
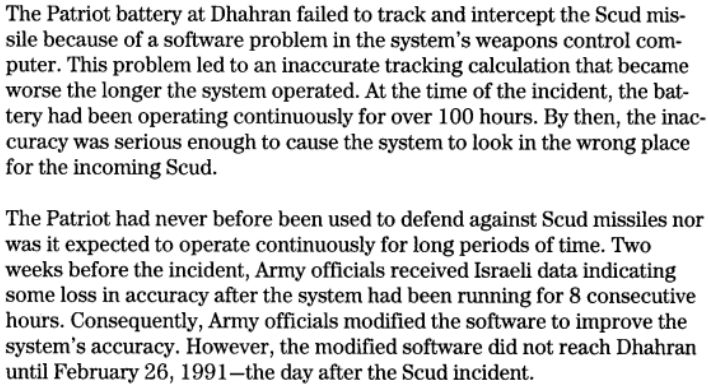


Figure 1: Emprunt à Guillaume Dewaele

Divergence (catastrophique) : exemple de conséquences :(



The Patriot battery at Dhahran failed to track and intercept the Scud missile because of a software problem in the system's weapons control computer. This problem led to an inaccurate tracking calculation that became worse the longer the system operated. At the time of the incident, the battery had been operating continuously for over 100 hours. By then, the inaccuracy was serious enough to cause the system to look in the wrong place for the incoming Scud.

The Patriot had never before been used to defend against Scud missiles nor was it expected to operate continuously for long periods of time. Two weeks before the incident, Army officials received Israeli data indicating some loss in accuracy after the system had been running for 8 consecutive hours. Consequently, Army officials modified the software to improve the system's accuracy. However, the modified software did not reach Dhahran until February 26, 1991—the day after the Scud incident.

Figure 2: Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia

Table des matières

Nombres flottants IEEE754 : format

Format double précision IEEE754

Le plus courant pour des applications « générales »

- 1 bit de signe
- 11 bits d'exposant
  - avec deux exposants spéciaux (tout à zéro, tout à un)
- 52 bits de mantisse
  - représente (usuellement) un nombre entre 1 et 2 comme $2^{-52} \times (2^{52} + m)$, avec m l'entier $\in [\![0, 2^{52} - 1]\!]$ représenté par la mantisse en base deux
  - un bit « gratuit » (grâce à la normalisation : le fait que 2^{52} est toujours inclus ici ; permis par le traitement spécial d'un exposant tout à zéro)
- Permet de représenter des valeurs $\pm\infty$ (pour les dépassements de capacité et certaines erreurs) et des valeurs non numériques « NaN » (un peu comme un `None`) (pour certaines erreurs)

Un peu plus de détails : *cf.* TD

Nombres flottants IEEE754 : ordres de grandeur

Ordres de grandeur

- Plus grand nombre (fini) représentable : $\approx 2^{1023} \approx 2^{1000} \approx 10^{300}$
- Plus petit nombre (fini) représentable (en valeur absolue) : $\approx 2^{-1074} \approx 2^{-1000} \approx 10^{-300}$
  - Différence liée aux exposants spéciaux
- Erreur *relative* de la *meilleure* approximation d'un nombre **représentable** hors cas particuliers très petits $\lessapprox 2^{-53} \approx 10^{-16}$

Valeurs spéciales : exemples

Infinis

- `2. *. (2.**1023.) (* infinity *)`
- `-2. *. (2.**1023.) (* neg_infinity *)`
- `(1. /. (2.**1023.)) *. (1. /. 2.**100.) (* 0. *)`
- `1. /. 0. (* infinity *)`
- `infinity +. infinity (* infinity *)`
- `infinity *. infinity (* infinity *)`
- `infinity -. 1. (* infinity *)`

NaNs

- `0. /. 0. (* nan *)`
- `infinity -. infinity (* nan *)`
- `infinity /. infinity (* nan *)`
- `nan +. nan (* nan *)`
- `nan <> nan (* true *)`

Table des matières

Nombres flottants en C

Types standard usuels

- `float` : format « simple » précision, hors programme
  - typiquement 1 bit de signe, 8 bits d'exposant, 23 bits de mantisse
- `double` : format « double » précision, seul au programme
  - *de facto* implémenté par le format double précision IEEE754

Écriture des littéraux

Le plus courant : notation « pointée » en base dix :

```
double m_pi = 3.14159265358979323846;
double m_pi_2 = 1.57079632679489661923;
double a = 0.;
```

Aussi possible : notation « scientifique » en base dix :

```
double b = .12E-10;
```

Affichage

Le plus courant

Spécifieur de conversion `%f` pour affichage « pointé » (et éventuellement `%e` ou `%E` pour affichage « scientifique »)

- Se fait avec une précision limitée : deux nombres physiquement différents peuvent être affichés identiquement !

Précision supplémentaire

On peut ajouter une précision d'**affichage** par un nombre de chiffres (décimaux) minimum à afficher, comme par ex. : `%.10f`

Démo.

Calcul avec des flottants

Par expressions flottantes

Tous les opérateurs arithmétiques ou de comparaison usuels, **sauf** `%`

Fonctions de bibliothèque disponibles 🙈

Le langage fournit une bibliothèque mathématique standard (hors programme)

- Nécessite l'inclusion du fichier d'entête `math.h` et la compilation avec l'option `-lm`
- Définit des (approximations) de nombres notables (comme $\pi/2$)
- Et des fonctions usuelles (trigo., division, arrondi, log, exponentiation, racines…)

Conversions types entiers ↔ flottants

Règles de conversion

- Conversions types entiers ↔ flottants possibles implicitement ou explicitement
- Conversions implicites d'expressions mixtes entier / flottant : les flottants ont la précédence
  - Par ex. : `1 + 1.0` est de type `double`
- Conversion (nécessairement explicite) flottant → type entier : la partie fractionnaire est ignorée (conversion « vers zéro »)

Exemples

```
1.7 == 1; // false (équivalent à 1.7 == 1.0)
(int)1.7 == 1; // true
(int)-1.7 == -1; // true
```

Conversions : erreurs possibles

Perte de précision

Typiquement pour les conversions type entier $\rightarrow$ type flottant

- Problème déjà évoqué des représentations approchées

Représentation impossible

Typiquement pour les conversions flottant $\rightarrow$ type entier

- Par ex. impossible de représenter 10^{300} avec un type entier standard
- *Undefined behaviour* si essayé…

Démo.

Table des matières

Nombres flottants en OCaml

Type standard

- `float` : format « double » précision IEEE754

Écriture des littéraux

Usuel : comme en C, mais doit commencer par un chiffre

```
utop # 3.14159265358979323846;;
- : float = 3.14159265358979312
utop # 1.57079632679489661923;;
- : float = 1.57079632679489656
utop # 0.;;
- : float = 0.
utop # 0.12E-10;;
- : float = 1.2e-11
utop # .12E-10;;
Error: Syntax error
```

Calcul avec des flottants

Fonctions arithmétiques usuelles

Le typage fort OCaml (et la « nécessité » d'inférence de type, et l'absence de polymorphisme non paramétrique) implique l'utilisation de fonctions distinctes de celles pour les `int`

- Addition : `(+.)`
- Soustraction : `(-.)`
- Multiplication : `( *.)`
- Division : `(/.)`

Et aussi :

- Exponentiation : `( **)`

Fonctions de comparaison

Les mêmes que pour les `int`, les `'a list`… grâce au polymorphisme

Conversions `int` ↔ `float`

Aucune conversion implicite

```
utop # 1 + 2.;;
Error: The constant 2. has type float but an expression was expected of
type int
utop # 1 +. 2;;
Error: The constant 1 has type int but an expression was expected of
type float
```

Conversions explicites

Via les fonctions :

- `int_of_float : float -> int`
- `float_of_int : int -> float`

Mêmes précautions à prendre qu'en C (les problèmes ne viennent pas du langage…)

Doc. de `int_of_float` :

> *Truncate the given floating-point number to an integer. The result is unspecified if the argument is `nan` or falls outside the range of representable integers.*