Arithmetique_sur_les_posits

May 21, 2019

1 Table of Contents

2 Texte d'oral de modélisation - Agrégation Option Informatique

2.1 Préparation à l'agrégation - ENS de Rennes, 2018-19

- *Date* : 15 juin 2018
- *Auteur* : Lilian Besson
- *Texte*: Texte rédigé par Lilian Besson, "Arithmétique avec les posits".

2.2 À propos de ce document

- Ceci est une *proposition* de correction, partielle et probablement non-optimale, pour la partie implémentation d'un *faux* texte d'annale de l'agrégation de mathématiques, option informatique.
- Ce document est un notebook Jupyter, et est open-source sous Licence MIT sur GitHub, comme les autres solutions de textes de modélisation que j'ai écrite cette année.
- L'implémentation sera faite en OCaml, version 4+ :

```
In [2]: Sys.command "ocaml -version";;
        print_endline Sys.ocaml_version;;
```

```
The OCaml toplevel, version 4.04.2
```

```
Out[2]: - : int = 0
```

```
Out[2]: - : unit = ()
```

```
In [3]: let print f =
            ignore (Printf.printf f);
            flush_all();
        ;;
```

```
4.04.2
```

```
Out[3]: val print : ('a, out_channel, unit) format -> unit = <fun>
```

2.3 Question de programmation

La question de programmation pour ce texte était donnée en page XXX :
FIXME

2.3.1 Modélisation

On est libre de choisir l'approche.

2.4 Texte

TODO: convertir en LaTeX et utiliser un modèle ressemblant au maximum à celui des "vrais textes".

Ce texte explique le fonctionnement des *posits*, en détaillant d'abord la signification de leurs bits, puis en discutant leur précision et amplitude adaptative.

Les *posits* sont un nouveau format (2015) pour représenter des nombres réels sur un ordinateur, proposés comme une une alternative aux nombres flottants standards (IEEE). Une présentation plus complète est donnée sur le site `posithub.org`.

L'avantage des *posits* est leur capacité à donner une plus grande précision ou une amplitude adaptative, en utilisant le même nombre de bits qu'un flottant standard. Par exemple, si un logiciel peut passer des flottants IEEE 64-bits à des *posits* 32-bits, elle pourra stocker deux fois plus de nombres en mémoire en même temps ! Cela peut sembler inutile, mais ce genre d'amélioration peut profiter à des calculs embarqués sur des micro-ordinateurs (*e.g.*, capteurs sans fils, robots) la mémoire du système est souvent très limitée, mais aussi à un ordinateur classique qui manipule un grand nombre de données (*e.g.*, apprentissage machine sur des grands corpus de données).

2.4.1 Rappels sur les flottants IEEE 754

On rappelera que les flottants standards sont souvent appelés `float` en OCaml, en Python, en C ainsi en Java. Selon les architectures, les `float` sont des flottants sur 32 ou 64 bits, et généralement les `double` sont des flottants sur 64 ou 128 bits, qui suivent tous les deux la norme IEEE 754. En Python, les `float` utilisent en fait des `double` en C.

```
In [4]: 0.0 +. 2018.0;;
```

```
Out[4]: - : float = 2018.
```

On rappelera aussi que les flottants standars contiennent trois valeurs spéciales, NaN (*"not a number"*, pas un nombre), `infinity` ou `inf` qui représente $+\infty$ et `neg_infinity` ou `-inf` qui représente $-\infty$.

```
In [5]: nan;;
        infinity;;
        -. infinity;;
        neg_infinity;;
```

```
Out[5]: - : float = nan


Out[5]: - : float = infinity


Out[5]: - : float = neg_infinity


Out[5]: - : float = neg_infinity
```

Arithmétique sur les flottants IEEE Les règles de calcul sur les valeurs spéciales et les calculs générant des valeurs spéciales sont illustrées par les exemples suivants :

```
In [6]: (* Apparition de valeurs spéciales *)
        1.0  /. 0.0;;
        -1.0 /. 0.0;;
        infinity /. infinity;;

        (* Calcul étendu aux valeurs spéciales *)
        1.0 +. infinity;; (* absorbant ! *)
        1.0 +. nan;;      (* absorbant ! *)
        (* Pareil pour -, *, / *)

        (* Changements de signe *)
        neg_infinity /. 3.0;;
        neg_infinity /. (-3.0);;

Out[6]: - : float = infinity


Out[6]: - : float = neg_infinity


Out[6]: - : float = nan


Out[6]: - : float = infinity


Out[6]: - : float = nan


Out[6]: - : float = neg_infinity


Out[6]: - : float = infinity
```

On notera aussi qu'avec les flottants IEEE, il y a deux zéros, un négatif et un positif :

```
In [7]: (-. 0.0) == 0.0;;

Out[7]: - : bool = false
```

Ce n'est pas le cas avec les entiers, par exemple :

```
In [8]: 0 == -0;;

Out[8]: - : bool = true
```

Enfin, on rappelle que l'arithmétique sur les nombres réels n'est pas exacte, par exemple :

```
In [9]: (0.2 +. 0.4, 0.6);;

Out[9]: - : float * float = (0.600000000000000089, 0.6)


In [10]: 0.2 +. 0.4 == 0.6;;

Out[10]: - : bool = false
```

Représentation par tableaux de bits Si la valeur NaN et les infinis sont uniques, ils ont en fait de très nombreuses représentations possibles, ce qui est souvent critiqué comme une perte. Par exemple, des flottants IEEE sur 32 bits a environ **16 millions** de façon de représenter NaN ! Voir cette page là (plus complète en anglais). Cette page montre des exemples d'encodage redondant pour les flottants IEEE à 64 bits, et celle ci pour les flottants à 32 bits.

On peut utiliser le module `Int32` pour obtenir un flottant à partir d'une suite de bits. J'utilise `"_"` pour séparer le bit de signe, les bits de l'exposant et la mantisse (ou significant).

```
In [11]: Int32.float_of_bits( Int32.of_string "0b0_01111111_00000000000000000000000" );;

Out[11]: - : float = 1.


In [12]: Int32.float_of_bits( Int32.of_string "0b1_10000000_00000000000000000000000" );;

Out[12]: - : float = -2.
```

Max et min. On ne peut pas utiliser `"0b0_11111111_11111111111111111111111"` pour le max, parce qu'il représente un nan, et on ne peut pas utiliser `"0b0_00000000_00000000000000000000000"` pour le min, parce qu'il représente un zéro.

```
In [13]: Int32.float_of_bits( Int32.of_string "0b0_11111110_11111111111111111111111" );;
```

```
Out[13]: - : float = 3.4028234663852886e+38

In [14]: Int32.float_of_bits( Int32.of_string "0b0_00000001_00000000000000000000000" );;
Out[14]: - : float = 1.17549435082228751e-38
```

Les zéro ont un exposant et une mantise remplie de zéro :

```
In [15]: Int32.float_of_bits( Int32.of_string "0b0_00000000_00000000000000000000000" );;
Out[15]: - : float = 0.

In [16]: (* normalement, donne -0. mais échoue ici *)
         Int32.float_of_bits( Int32.of_string "0b1_00000000_00000000000000000000000" );;

        Exception: Failure "Int32.of_string".
    Raised by primitive operation at file "[16]", line 2, characters 19-78
    Called from file "toplevel/toploop.ml", line 180, characters 17-56
```

Les infinis ont un exposant rempli de 1 et une mantisse remplie de 0.

```
In [17]: Int32.float_of_bits( Int32.of_string "0b0_11111111_00000000000000000000000" );;
Out[17]: - : float = infinity

In [18]: Int32.float_of_bits( Int32.of_string "0b1_11111111_00000000000000000000000" );;
Out[18]: - : float = neg_infinity
```

Codage redondant ! Plusieurs exemples de Not a Number :

```
In [19]: Int32.float_of_bits( Int32.of_string "0b1_11111111_10000000000000000000001" );;
Out[19]: - : float = nan

In [20]: Int32.float_of_bits( Int32.of_string "0b1_11111111_00000000000000000000001" );;
Out[20]: - : float = nan

In [21]: Int32.float_of_bits( Int32.of_string "0b0_11111111_00000100000000000000001" );;
Out[21]: - : float = nan
```

Il y en a *des millions* comme ça, puisqu'un exposant rempli de 1 est toujours un nan ! C'est une perte d'espace énorme ! Exactement, il y a en $2^{24} = 16777216$ (16 millions), puisqu'on peut mettre n'importe quelle valeur sur les 23 bits de la mantisse et le bit de signe !

2.4.2 Nombres universels et *posits*

Les "nombres universels" (unum) ont été proposés par John Gustafson comme une autre manière pour la representation d'un nombre réel en utilisant un nombre fini de bits, afin d'être une alternative aux nombres flottants IEEE. Cf. son livre intitulé *The End of Error. Posits are a hardware-friendly version of unums.*, J. Gustafson, 2015 (voir aussi ce document de référence et pour une introduction plus courte, voir cette page et ce poster).

Un nombre flottant (IEEE 754) a un bit de signe, un ensemble de bits représentant l'exponent et un ensemble de bits appelés le significant (aussi désigné avec le nom "mantisse"). Pour une taille donnée, la longueur des différentes parties est fixée et standardisée Un nombre IEEE 752 stocké sur 64-bits au ainsi, 1 bit de signe, 11 bits d'exponent bits, et 52 bits pour le significant.

Un *posit* reprend cette idée, mais ajoute une quatrième catégorie de bits, appelée *régime*. Un *posit* est un ensemble de bits (ordonnés), qui sont les suivantes (dans cet ordre) :

- bit de signe,
- regime,
- exposant,
- fraction.

Le significant d'un nombre IEEE 754 correspond à la "partie fractionnelle", *.i.e.*, `fraction`. Contrairement aux nombres flottants classiques, les deux parties de l'exposant et la fraction d'un *posit* n'ont pas de longueur fixée. Les bits du signe et du régime ont la priorité, puis les bits restants, s'il y en a, vont dans l'exposant. Enfin, s'il reste des bits après l'exposant, ils vont dans la fraction.

2.4.3 Morceaux d'un *posit*

Pour comprendre les *posits* davantage, et prouver qu'ils ont des avantages en comparaison des nombres flottants classiques, nous allons détailler leur représentation bit-à-bit. L'organisation bit-à-bit d'un nombre *posit* est spécifié par deux nombres, le nombre total de bits, noté n, et le nombre maximum de bits dédiés à l'exponent, noté es. On dit qu'un tel nombre *posit* est un nombre `posit<n, es>`.

Bit de signe Comme pour les flottants IEEE, le premier bit est le bit de signe. S'il est 1, le nombre est négatif, et on prend le complément à 2 (*i.e.*, $0 \mapsto 1, 1 \mapsto 0$ sur les bits) de tous les bits suivants avant d'extraire les informations du reste des bits.

Bits de régime Après le bit de signe suit les bits de régime, dont le nombre est variable. Il peut y avoir de 1 à $n-1$ bits de régime. Comment savoir quand se terminent les bits de régime ? Quand une suite de bits identiques se terminent, soit après $n-1$ bits, soit parce qu'on lit un bit opposé.

Si le premier bit après le bit de signe est un 0, alors les bits de régime continuent jusqu'à trouver un bit à 1 ou lire $n-1$ bits. Et inversement si le premier bit est un 1, les bits de régime continuent jusqu'à trouver un bit à 0.

Le bit qui indique la fin d'une séquence n'est pas inclut dans les bits de régime : le régime n'est qu'une séquence constituée uniquement de 0 ou de 1.

Bits de l'exposant Les bits du signe et du régime sont au début du nombre. S'il reste des bits à lire, les bits de l'exposant sont les suivants. Il peut ne pas y avoir de bits d'exposant. Le nombre maximum de bits d'exposant est spécifié par ce nombre `es`. S'il y a au moins `es` bits après le bit de signe, les bits du régime, et le bit terminant le régime, alors les `es` prochains sont pour l'exposant. Mais s'il y a moins de `es` bits, tous les bits restants sont dans l'exposant.

Bits de la fraction S'il reste des bits après le bit de signe, les bits du régime, le bit terminant le régime, et les bits de l'exposant, ils constituent les bits de la fraction.

2.4.4 Interprétation des morceaux d'un *posit*

Expliquons maintenant comment utiliser ces quatres composants pour représenter un nombre réel (ou plutôt, une approximation décimale d'un nombre réel).

Si b est le signe de bit, alors le signe s du nombre représenté est $(-1)^b$.

Soit m le nombre de bits du régime. Alors posons $k = m$ si le régime consiste en une séquence de 0, et $k = m - 1$ sinon.

$$k = \begin{cases} -m & \text{si le régime est constitué de } m \text{ 0's} \\ m-1 & \text{si le régime est constitué de } m \text{ 1's} \end{cases}$$

La "graîne universelle" (*useed*) u du *posit* est déterminée par `es`, la taille maximum de l'exposant :

$$u = 2^{2^{es}}.$$

L'exposant e est ensuite simplement un entier non signé obtenu en interprétant les bits de l'exposant, avec le bit de poids faible *en dernier* (comme un nombre binaire classique) ce qu'on désigne souvent par "little endian", la petite fin.

La fraction f est $1+$ les bits de fraction interprétés comme s'ils suivent une virgule en binaire. Par exemple, si les bits de fraction sont 10011, alors $f = 1.10011$ en binaire. Sur cet exemple, la partie après la virgule est interprétée comme suit : $0.10011_2 = 1 * 2^{-1} + 0 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4} + 1 * 2^{-5} = 0.59375_{10} = 10011_2/2^6 = 19/32$ (référence).

Pour résumer, la valeur du nombre *posit* est le produit des contributions du bit de signe, des bits de régume, des bits de l'exposant (s'il y en a), et des bits de la fraction (s'il y en a) :

$$x = s \times u^k \times 2^e \times f = (-1)^b f 2^{e+k2^{es}}.$$

2.4.5 Valeurs spéciales avec les *posits*

Il y a deux *posits* spéciaux, avec $n-1$ bits à 0 après le bit de signe. Une séquence de n bits à 0 représente le nombre zéro, et un bit à 1 suivi de $n-1$ bits à 0 représente $\pm\infty$.

Il n'y a qu'un seul zéro pour les nombres posits, alors que les flottants IEEE contiennent deux zéro, un positif et un négatif.

Il n'y a aussi qu'un nombre infini. Pour cette raison, on peut dire que les posits représentent les nombres réels projectifs plutôt que les nombres réels étendus.

Les flottants IEEE ont deux types d'infinis, positif et negatif, ainsi que des "non-nombres" (*Not a Number*). Les posits n'ont qu'un seul élément qui ne correspondent pas à un nombre réel fini, et il s'agit de $\pm\infty$.

- Question : quelle utilité peut avoir un zéro négatif ?

2.4.6 Amplitude dynamique et précision

L'amplitude dynamique et la précision d'un *posit* dépendent de la valeur de `es`. Pour une plus grande valeur de `es`, la contribution du régime et de l'exposant augmentent, et donc des nombres ayant une grande valeur peuvent être représentés. Augmenter `es` augmente l'amplitude des *posits* `posit<n,es>`.

L'amplitude dynamique, mesurée en décades, est calculée par le logarithme en base 10 du rapport entre la plus grande et la plus petite valeur représentable par les *posits* `posit<n,es>`.

Mais augmenter la `es` implique aussi de diminuer le nombre de bits disponibles pour la fraction, ce qui diminue la précision. L'un des avantages des *posits* est cette possibilité de choisir `es` pour ajuster le compromis entre amplitude dynamique et précision pour satisfaire au mieux les besoins d'une application.

- Question : cela n'est pas le cas pour les flottants IEEE ?

Pour des *posits* `posit<n,es>`, la plus grande valeur finie représentable est notée maxpos. Cette valeur est obtenue quand k est aussi grand que possible as, *i.e.*, quand tous les bits après le bit de signe sont des 1. Dans ce cas, $k = n - 2$, donc maxpos vaut

$$u^{n-2} = \left(2^{2^{es}}\right)^{n-2}.$$

Symétriquement, la plus petite valeur positive représentable, notée minpos, est obtenue lors que k est négatif et le plus petit possible, *i.e.*, quand le plus grand nombre possible de bits après le bit de signe sont à 0. Il ne peut pas n'y avoir que des 0 sinon on obtient la représentaion de 0, donc il faut un unique bit à 1 à la fin. Dans ce cas, on obtient $m = n - 2$ et $k = 2 - n$.

$$\text{minpos} = u^{2-n} = \left(2^{2^{es}}\right)^{2-n} = 1/\text{maxpos}$$

L'amplitude dynamique est

$$\log_{10}\left(2^{2^{e}}\right)^{2n-4} = (2n-4) \times 2^{es} \times \log_{10} 2.$$

Par exemple, des *posits* sur 16 bits avec `es`$= 1$ ont une amplitude dynamique de 17 décades, alors que des flottants IEEE sur 16 bits ont une amplitude de 12 décades.

Les *posits*, dans cet exemple, ont une fraction de 12 bits pour les nombres proche de 1 et les flottants ont un significant (ou mantisse) de 10 bits. Donc un nombre `posit<16,1>` a à la fois une plus grand amplitude dynamique et une meilleure précision (près de 1) que les flottants sur 16 bits.

[FIXME use next post.]

Notez que la précision d'un *posit* dépend de sa valeur. On dit qu'ils ont une précision *adaptative*. Les nombres proches de 1 ont une plus grande précision, alors que les nombres très grands et très petits ont un plus faible précision. Et c'est précisément ce que l'on cherche en pratique. Dans la plupart des calculs, les valeurs des nombres sont souvent de l'ordre de 1 (et pas 10^{-15} par exemple), et pour les très grandes et très petites valeurs, on veut surtout qu'il n'y a pas d'erreur de dépassement supérieur (une très grande valeur devient négative), ou inférieur.

2.4.7 Théorie à rajouter

FIXME

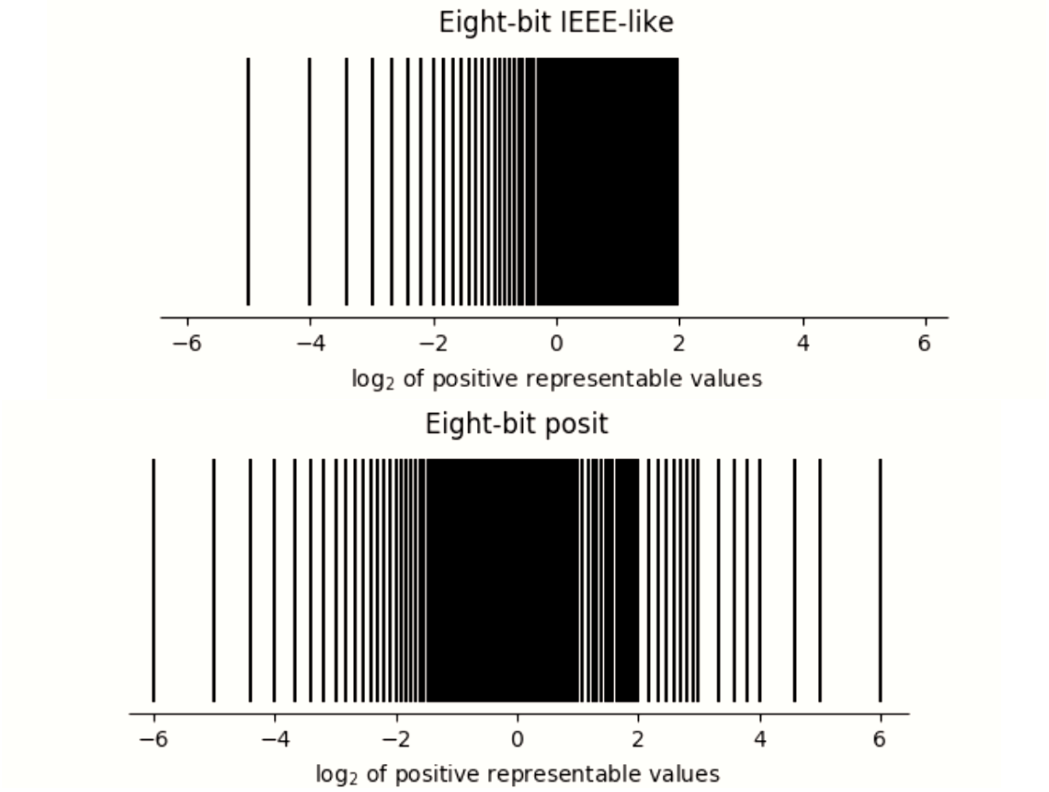
2.4.8 Amplitude adaptative, flottants IEEE *vs* posits

Il n'y pas grand chose à dire, les posits gagnent carrément !
Il faut que j'ajoute de la théorie !

Inspiré par ce post de blog

2.4.9 Exemple sur les nombres stockés sur 8 bits

Inspiré par ce post de blog

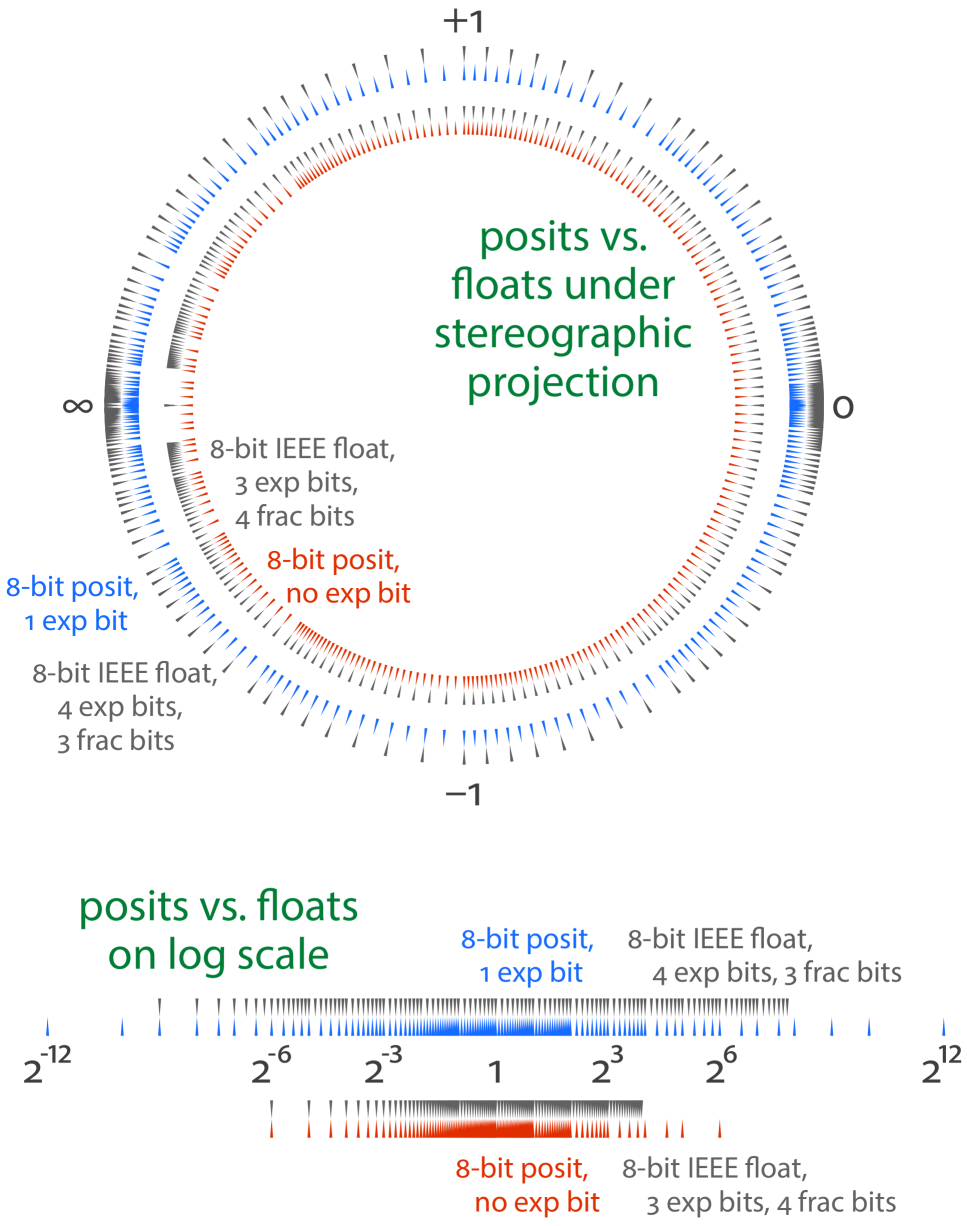


https://www.johndcook.com/eight_bit_posit2.png
FIXME

2.4.10 Arithmétique sur les *posits*

TODO à rédiger. Ce code peut aider : https://github.com/mightymercado/PySigmoid/blob/master/PySigmoi
Dans cette partie, nous considérons les calculs directement implémentés sur les *posits*.
Une première approche consiste à transformer les *posits* en nombres flottants, et utiliser les opérations classiques sur les *posits*, puis à convertir le résultat flottant en un *posit*.

- Question : à `n` et `es` fixés, peut-on convertir un flottant en un *posit* ? De façon unique ? Décrivez l'algorithme.
- Question : cela peut-il fonctionner ? Peut-on espérer garder les avantages décrits plus hauts si on se limite à cette approche naïve ?



comparaison_floats_vs_posits_under_stereographic_projection.png 50%

Proposer des algorithmes qui permettent de calculer les opérations suivantes, pour des *posits* x et y qui partagent le même format `posit<n,es>` :

- $x \mapsto -x$
- $x, y \mapsto x + y$
- $x, y \mapsto x - y$
- $x, y \mapsto x * y$
- $x \mapsto 1/x$
- $x, y \mapsto x/y$

2.5 Solution

On va essayer d'être rapide et de faire simple.

> Si vous êtes curieux ou que vous cherchez à utiliser les *posits* pour votre propre application, cette page liste des implémentationes des *posits* et de leur arithmétique en différents langages (Javascript, Julia, Rust, C++, Python etc).

2.5.1 Types et représentations

On va représenter un *posit* comme un tableau de `bool` en OCaml, pour se rapprocher le plus possible de l'implémentation bas-niveau qui utilisera une séquence de bits.

Notez qu'on a **besoin** d'aussi stocker `es` et `n`, si on veut que notre implémentation soit générique et facile à écrire. Sinon on pourrait écrire chaque fonction suivante de telle sorte qu'elle accepte `es` et `n` en arguments.

```
In [22]: type bits = bool array;;
```

```
Out[22]: type bits = bool array
```

```
In [23]: type posit = {
             es : int; (* prend de la place en plus ! *)
             n  : int; (* mais simplifie l'écriture des fonctions *)
             b  : bits;
         };;
```

```
Out[23]: type posit = { es : int; n : int; b : bits; }
```

> Est-ce que ça coûte très cher de stocker aussi `n` et `es` ? A priori non, parce que pour un *posit* de 32 bits, `b` occupe 32 bits et `n` et `es` sont majorés par $\log_2(32) = 5$ donc occupe $2 \times 5 = 10$ bits au maximum à eux deux. C'est un sur-coût raisonnable pour notre approche.
>
> En pratique, on l'a dit, on choisirait `n` et `es` *une bonne fois pour toute* pour une certaine application.

2.5.2 Fonction demandée

On vous demandait de convertir un *posit* en un nombre flottant. On va appliquer à la lettre la spécification donnée dans le texte.

On découpe tout le traitement en sous fonctions, autant que possible.

Signe d'un posit

```
In [24]: let signe_of_posit (p : posit) : int =
             if p.b.(0) then -1 else 1
         ;;

Out[24]: val signe_of_posit : posit -> int = <fun>
```

```
In [25]: let echange_bits (b : bits) (debut : int) : bits =
             let b2 = Array.copy b in
             for i = debut to (Array.length b) - 1 do
                 b2.(i) <- not b.(i)
             done;
             b2
         ;;

Out[25]: val echange_bits : bits -> int -> bits = <fun>
```

```
In [26]: let echange_bits_apres_le_premier (p : posit) : posit =
             if p.b.(0) then
                 { p with b = echange_bits p.b 1 }
             else
                 { p with b = Array.copy p.b }
         ;;

Out[26]: val echange_bits_apres_le_premier : posit -> posit = <fun>
```

Régime d'un posit

```
In [27]: let regime_of_posit (p : posit) : int =
             let premier_bit_regime = p.b.(1) in
             let encore_les_memes_bits = ref true in
             let m = ref 1 in
             while !encore_les_memes_bits && !m <= p.n - 1
             do
                 if p.b.(!m + 1) <> premier_bit_regime then
                     encore_les_memes_bits := false
                 else
                     incr m
             done;
```

```
            if premier_bit_regime then
                (!m - 1)  (* m bits à 1 *)
            else
                (- !m)    (* m bits à 0 *)
        ;;
```

```
Out[27]: val regime_of_posit : posit -> int = <fun>
```

Binaire vers entier On a d'abord besoin de savoir interpréter une séquence binaire comme un nombre entier non signé. On utilise la méthode de Hörner qui est optimale (en terme de nombre d'opérations arithmétiques) pour ce problème :

```
In [28]: let uint_of_bits (b : bits) : int =
            let taille = Array.length b in
            let i = ref 0 in
            for k = 0 to taille-1 do
                if b.(k) then
                    i := !i * 2 + 1
                else
                    i := !i * 2
            done;
            !i
        ;;
```

```
Out[28]: val uint_of_bits : bits -> int = <fun>
```

> On peut aussi écrire ça avec un simple `Array.fold_left`, mais cette fonction n'est pas disponible dans les versions plus anciennes de OCaml.

On a besoin de quelques tests :

```
In [29]: let t = true and f = false;;
```

```
Out[29]: val t : bool = true
         val f : bool = false
```

```
In [30]: uint_of_bits [|f; f; f|];;
```

```
Out[30]: - : int = 0
```

Le bit de poids faible est bien à la fin :

```
In [31]: uint_of_bits [|t; f; f|];;
```

```
Out[31]: - : int = 4
```

```
In [32]: uint_of_bits [|t; t; f|];;
Out[32]: - : int = 6

In [33]: uint_of_bits [|f; f; t|];;
Out[33]: - : int = 1

In [34]: uint_of_bits [|t; t; t|];;
Out[34]: - : int = 7
```

Exposant d'un posit Pour éviter de la recalculer, on renvoit aussi la taille de l'exposant, pour faciliter le calcul des bits de fraction.

```
In [35]: let exposant_of_posit (p : posit) (m : int) : (int * int) =
             let taille = Array.length p.b in
             let taille_restante = taille - 1 - m - 1 in
             let taille_exposant = min taille_restante p.es in
             (* on enlève le bit de signe, le régime, et le bit de fin du régime *)
             let exposant = uint_of_bits (Array.sub p.b (2 + m) taille_exposant) in
             exposant, taille_exposant
         ;;
Out[35]: val exposant_of_posit : posit -> int -> int * int = <fun>
```

Binaire décimal vers flottants C'est assez rapide, on va écrire une fonction qui prend le tableau des bits après la virgule décimale (*e.g.*, 10011) et donne le nombre flottant (décimal) correspondant à $0.f$ (*e.g.*, $0.10011_2 = 0.59375_{10}$). Il suffira d'ajouter 1_{10} pour obtenir la fraction.

```
In [36]: let float_of_bits_apres_virgule (b : bits) : float =
             let taille = float_of_int (Array.length b) in
             let nominateur = float_of_int (uint_of_bits b) in
             let denominateur = 2.0 ** taille in
             nominateur /. denominateur
         ;;
Out[36]: val float_of_bits_apres_virgule : bits -> float = <fun>
```

Par exemple :

```
In [37]: float_of_bits_apres_virgule [|t;f;f;t;t|];;
Out[37]: - : float = 0.59375

In [38]: float_of_bits_apres_virgule [||];;
Out[38]: - : float = 0.
```

Fraction d'un posit On a juste à extraire les bits de la fraction, s'il en reste.

```
In [39]: let fraction_of_posit (p : posit) (m : int) (taille_exposant : int) : float =
             let debut_fraction = 1 + m + 1 + taille_exposant in
             (* Pas sur de ce calcul *)
             let taille_fraction = (Array.length p.b) - debut_fraction in
             if taille_fraction > 0 then begin
                 let bits_apres_virgule = Array.sub p.b debut_fraction taille_fraction in
                 1.0 +. float_of_bits_apres_virgule (bits_apres_virgule)
             end else (* notez que ce cas n'est pas nécessaire *)
                 1.0
         ;;

Out[39]: val fraction_of_posit : posit -> int -> int -> float = <fun>
```

Puissance entière OCaml ne propose pas d'opérateur d'exponentiation entière, ** est pour les flottants. C'est une bonne occasion de montrer qu'on sait écrire rapidement l'algorithme d'exponentiation rapide :

```
In [40]: let carre (x : int) : int = x * x;;

Out[40]: val carre : int -> int = <fun>
```

```
In [41]: (** Complexité en O(log_2 n) nombre d'appels (donc mémoire),
             et opérations arithmétiques. *)
         let rec expo_int (x : int) (n : int) : int =
             match n with
             | n when n < 0 -> failwith "expo_int avec n négatif n'est pas possible";
             | 0 -> 1
             | 1 -> x
             (* il ne faut PAS écrire
             (expo_int x (n/2)) * (expo_int x (n/2))
             sinon on ne gagne rien : la même fonction est appelée deux fosi *)
             | n when n mod 2 = 0 -> carre (expo_int x (n/2))
             | n when n mod 2 = 1 -> x * (carre (expo_int x (n/2)))
             | _ -> failwith "Valeur incompatible pour expo_int"
         ;;

Out[41]: val expo_int : int -> int -> int = <fun>
```

```
In [42]: expo_int 2 1;;
         expo_int 2 2;;
         expo_int 2 3;;
         expo_int 2 4;;

Out[42]: - : int = 2
```

```
Out[42]: - : int = 4


Out[42]: - : int = 8


Out[42]: - : int = 16
```

Notez qu'on pourrait utiliser l'opérateur de décallage de bits vers la gauche :

```
In [43]: ( lsl );;

Out[43]: - : int -> int -> int = <fun>


In [44]: 2 lsl 0;;
         2 lsl 1;;
         2 lsl 2;;
         2 lsl 3;;

Out[44]: - : int = 2


Out[44]: - : int = 4


Out[44]: - : int = 8


Out[44]: - : int = 16
```

Est-ce qu'un posit est infini ou nul ? C'est très rapide à vérifier, avec cette fonction :

```
In [45]: Array.for_all

Out[45]: - : ('a -> bool) -> 'a array -> bool = <fun>


In [46]: let est_nul (p : posit) : bool =
             Array.for_all (not) p.b
         ;;

Out[46]: val est_nul : posit -> bool = <fun>


In [47]: let est_infini (p : posit) : bool =
             p.b.(0) && (Array.for_all (not) (Array.sub p.b 1 ((Array.length p.b) - 1)))
         ;;

Out[47]: val est_infini : posit -> bool = <fun>
```

Mettre tout ça ensemble On va commencer par écrire une fonction qui affiche un `posit<n,es>`, pour aider au débogage de la suite.

```
In [48]: let print_posit (p : posit) : unit =
             Format.printf "\nposit<%i,%i> " p.n p.es;
             Array.iter (fun a -> Format.printf (if a then "1" else "0")) p.b;
             flush_all();
         ;;

Out[48]: val print_posit : posit -> unit = <fun>
```

Et enfin on assemble le tout :

```
In [49]: (** Complexité linéaire dans la taille des bits de p *)
         let posit_to_float ?(debogue=false) (p : posit) : float =
             if debogue then print_posit p;
             (* Bit de signe *)
             let signe = signe_of_posit p in
             if debogue then Format.printf "\nSigne %i" signe; flush_all();
             (* On doit inverser les bits suivant si le nombre est négatif *)
             let p2 = echange_bits_apres_le_premier p in
             if debogue then print_posit p2;
             (* Bits de régime *)
             let k = regime_of_posit p2 in
             if debogue then Format.printf "\nRegime %i" k; flush_all();
             (* Taille du régime *)
             let m = if k < 0 then -k else k + 1 in
             if debogue then Format.printf "\nTaille du regime %i" m; flush_all();
             (* Bits de l'exposant *)
             let exposant, taille_exposant = exposant_of_posit p2 m in
             if debogue then Format.printf "\nExposant %i" exposant; flush_all();
             if debogue then Format.printf "\nTaille de l'exposant %i" taille_exposant; flush_a
             (* Bits de la fraction *)
             let fraction = fraction_of_posit p2 m taille_exposant in
             if debogue then Format.printf "\nFraction %g" fraction; flush_all();
             (* Gestion de l'infini *)
             if est_infini p then infinity
             else begin
             if est_nul p then 0.0
             else
                 (float_of_int signe) *.
                 (* on fait le plus possible de calcul sur les entiers *)
                 2.0 ** (float_of_int (exposant + k * (expo_int 2 p.es)))
                 *. fraction
             end
         ;;

Out[49]: val posit_to_float : ?debogue:bool -> posit -> float = <fun>
```

Exemples On doit faire des essais, sur des *posits* sur 16 bits, avec par exemple `n=10` et `es=1` comme l'exemple dans le texte. :

```
In [50]: let n = 10 and es = 1;;

Out[50]: val n : int = 10
         val es : int = 1
```

Les valeurs spéciales sont évidemment bien correctes :

```
In [51]: let p1 = { n = n; es = es; b = [|f;f;f;f;f;f;f;f;f;f;f;f;f;f;f;f|] };;
         let _ = posit_to_float ~debogue:true p1;;

Out[51]: val p1 : posit =
           {es = 1; n = 10;
            b =
             [|false; false; false; false; false; false; false; false; false; false;
               false; false; false; false; false; false|]}



posit<10,1> 0000000000000000
Signe 1
posit<10,1> 0000000000000000
Regime -10
Taille du regime 10
Exposant 0
Taille de l'exposant 1


Out[51]: - : float = 0.


In [52]: let p2 = { n = n; es = es; b = [|t;f;f;f;f;f;f;f;f;f;f;f;f;f;f;f|] };;
         let _ = posit_to_float ~debogue:true p2;;

Out[52]: val p2 : posit =
           {es = 1; n = 10;
            b =
             [|true; false; false; false; false; false; false; false; false; false;
               false; false; false; false; false; false|]}


Fraction 1
posit<10,1> 1000000000000000
Signe -1
posit<10,1> 1111111111111111
Regime 9
```

```
Taille du regime 10
Exposant 1
Taille de l'exposant 1


Out[52]: - : float = infinity
```

Essayons d'autres valeurs :

- Positif, avec un régime de 1 et de longueur 4 (donc $m = k - 1 = 3$), tous les autres bits à 0, on obtient le nombre $(-1)^0 * 1.0 * 2^{0+3*2^1} = 64$:

```
In [53]: let p3 = { n = n; es = es;
             b = [|f; t;t;t;t; f;   f;   f;f;f;f;f;f;f;f;f|]
             (* signe regime4t sepa expo fraction *)
         };;
         let _ = posit_to_float ~debogue:true p3;;

Out[53]: val p3 : posit =
           {es = 1; n = 10;
            b =
             [|false; true; true; true; true; false; false; false; false; false;
               false; false; false; false; false; false|]}


Fraction 1.875
posit<10,1> 0111100000000000
Signe 1
posit<10,1> 0111100000000000
Regime 3
Taille du regime 4
Exposant 0
Taille de l'exposant 1


Out[53]: - : float = 64.
```

- Positif, avec un régime de 1 et de longueur 4 (donc $m = k - 1 = 3$), un exposant à 1 et tous les autres bits à 0, on obtient le nombre $(-1)^0 * 1.0 * 2^{1+3*2^1} = 128$:

```
In [54]: let p4 = { n = n; es = es;
             b = [|f; t;t;t;t; f;   t;   f;f;f;f;f;f;f;f;f|]
             (* signe regime4t sepa expo fraction *)
         };;
         let _ = posit_to_float ~debogue:true p4;;
```

```
Out[54]: val p4 : posit =
           {es = 1; n = 10;
            b =
             [|false; true; true; true; true; false; true; false; false; false; false;
               false; false; false; false; false|]}


Fraction 1
posit<10,1> 0111101000000000
Signe 1
posit<10,1> 0111101000000000
Regime 3
Taille du regime 4
Exposant 1
Taille de l'exposant 1


Out[54]: - : float = 128.
```

- Si on change es la taille maximale de l'exposant, on voit qu'on peut atteindre des nombres plus grands. Positif, avec un régime de 1 et de longueur 4 (donc $m = k - 1 = 3$), un exposant à $11_2 = 3$ (lu en binaire !) et tous les autres bits à 0, on obtient le nombre $(-1)^0 * 1.0 * 2^{3+3*2^2} = 32768$:

```
In [55]: let p5 = { n = n; es = 2;
             b = [|f; t;t;t;t; f;   t;t;   f;f;f;f;f;f;f;f|]
             (* signe regime4t sepa expo fraction *)
         };;
         let _ = posit_to_float ~debogue:true p5;;

Out[55]: val p5 : posit =
           {es = 2; n = 10;
            b =
             [|false; true; true; true; true; false; true; true; false; false; false;
               false; false; false; false; false|]}


Fraction 1
posit<10,2> 0111101100000000
Signe 1
posit<10,2> 0111101100000000
Regime 3
Taille du regime 4
Exposant 3
Taille de l'exposant 2


Out[55]: - : float = 32768.
```

Avec des bits de fraction non nuls Les *posits* sont très simples à interpréter tant que leurs bits de fraction sont non nuls, mais ils ne sont pas tellement plus compliqués avec une fraction. Voici un exemple :

```
In [56]: let p6 = { n = 16; es = 3;
             b = [|f; f;f;f;   t;   t;f;t;   t;t;f;t;t;t;f;t|]
             (* signe regime3t sepa expo     fraction *)
         };;
         let _ = posit_to_float ~debogue:true p6;;

Out[56]: val p6 : posit =
           {es = 3; n = 16;
            b =
             [|false; false; false; false; true; true; false; true; true; true; false;
               true; true; true; false; true|]}


Fraction 1
posit<16,3> 0000110111011101
Signe 1
posit<16,3> 0000110111011101
Regime -3
Taille du regime 3
Exposant 5
Taille de l'exposant 3


Out[56]: - : float = 3.55392694473266602e-06
```

2.6 Complexités

2.6.1 En espace

Ces calculs coûtent un espace linéaire en mémoire, parce que pour éviter de modifier le posit d'entrée on créée des copies de son tableau de bits.

2.6.2 En temps

Ces calculs coûtent un temps linéaire en n.

2.7 Conclusion

Voilà pour la question obligatoire de programmation.

2.7.1 Qualités

- On a fait des exemples et *on les garde* dans ce qu'on présente au jury,
- On a testé la fonction exigée sur de petits exemples.

2.7.2 Défauts

- On a implémenté aucun autre développement. A suivre, j'en ajouterai quand je peux.

  Bien-sûr, ce petit notebook ne se prétend pas être une solution optimale, ni exhaustive.

  Vous auriez pu choisir de modéliser le problème avec une autre approche, mais je ne vois pas ce qu'on aurait pu faire différement.

  C'est tout pour aujourd'hui les amis, allez voir ici pour d'autres corrections, et que la force soit avec vous !