Le problème de la transposition informatique pour les algorithmes numériques

Christophe Jermann, Frédéric Goualard

Le problème de la transposition informatique pour les algorithmes numériques

Christophe Jermann et Frédéric Goualard

Nantes Université, École Centrale Nantes,
CNRS, LS2N, UMR 6004, F-44000 Nantes, France

2 juin 2023

Résumé

La transposition informatique d'un algorithme mathématique en un programme informatique suppose la prise en compte des *contraintes de la modélisation computable* induites par l'utilisation d'ensembles numériques finis aux propriétés réduites (entiers, nombres à virgule flottante). Nous rappelons l'importance d'une transposition informatique correcte à l'aide d'exemples d'algorithmes numériques sur les entiers et les réels.

1 Supposons une vache sphérique…

Le ministère de l'Agriculture de Freedonie demanda un jour au Professeur *X.*, célèbre physicien, de trouver le moyen de doubler la production des laiteries du pays. Une quinzaine de jours plus tard, celui-ci convoqua les représentants du ministère et de la presse pour leur exposer un plan très précis permettant d'atteindre et même de dépasser cet objectif. Le Professeur *X.* s'approcha cérémonieusement du tableau noir et, lorsque le brouhaha de la salle retomba, lança d'une voix assurée tout en traçant de savantes équations :

— *Supposons une vache sphérique de densité uniforme…*

Cette fable, censée illustrer avec humour les dangers de la sur-simplification des modèles appliqués à la vie réelle, circule sous des formes diverses depuis de nombreuses années — le magazine *Science* évoquait déjà, par exemple, en 1970, la question des poulets sphériques [18].

Comment calculer la somme S de n valeurs réelles x_i sur une machine ? La *transposition informatique* [3] telle que présentée par Briant [5] et Briant & Bronner [6], par exemple, est un processus en deux étapes où l'on va d'abord passer d'un problème exprimé dans le langage mathématique :

$$S = \sum_{i=1}^{n} x_i$$

à un algorithme en pseudo-code *sur les variables mathématiques* mettant ici en œuvre une *somme récursive* (programme 1).

Programme 1 – Somme récursive de n nombre réels. Algorithme en pseudo-code sur les variables mathématiques.

$S \leftarrow 0$
pour chaque $x_i \in (x_1, \ldots, x_n)$ **faire**
    $S \leftarrow S + x_i$
finpour
afficher S

La deuxième étape consiste ensuite à traduire le pseudo-code dans un langage de programmation, par exemple Python [1], *sur des variables informatiques* (programme 2).

Programme 2 – Somme récursive de n nombre réels. Programme Python issu de la transposition informatique directe de l'algorithme du programme 1.

```
S = 0
for xi in x:
    S += xi
print(S)
```

L'exécution du programme 2 sur la série :

$$x = (9\,007\,199\,254\,740\,992\,;\ 0{,}25\,;\ 1{,}0\,;\ 0{,}75\,;0{,}1\,;\ 0{,}9\,;\ -9\,007\,199\,254\,740\,992) \quad (1)$$

devrait surprendre l'utilisateur lorsqu'il constatera que la somme calculée vaut 0 au lieu de 3. Le processus de transposition classique n'explicite malheureusement pas suffisamment les *contraintes de modélisation computable* [3] : l'algorithme 1 est parfaitement valide car il s'applique sur des variables mathématiques à valeur dans $\mathbb{R}$. Le programme 2 a vocation à être exécuté sur une machine où les ensembles de nombres manipulés sont des sous-ensembles finis de ces ensembles, représentés en binaire et possédant des propriétés réduites. L'arithmétique des nombres à virgule flottante remplaçant les nombres réels n'est pas associative pour l'addition, par exemple. Le code obtenu est donc généralement sous-optimal, voire carrément erroné.

Comme le remarquent Haspekian et Nijimbéré [9], les programmes officiels du lycée sont particulièrement ambigus sur la notion d'algorithme, ce qui entretient le flou entre *algorithme mathématique*, appliqué à des variables à valeurs dans des ensembles infinis, et *algorithme informatique*, exécutable sur une machine finie.

Cependant, il semble clair qu'un programme écrit dans un langage informatique se doit de mettre en œuvre un algorithme qui tient compte des possibilités des ensembles de nombres effectivement manipulés. Le processus de transposition informatique de Briant doit donc être amendé pour que l'algorithme en pseudo-code obtenu après la première étape s'applique sur les ensembles qui seront réellement utilisés sur une machine. Pour le cas de la somme de n réels, par exemple, de nombreux travaux [12, 15, 14, 4] présentent de tels algorithmes sur les nombres à virgule flottante, indépendamment de leur mise en œuvre dans un langage de programmation.

1. Dans la suite, tous les exemples utilisent CPython 3.

Ces algorithmes informatiques ne peuvent être le résultat d'une transcription quasi-automatique d'un algorithme mathématique sur les réels, domaine pour lequel ils ne présentent pas d'intérêt par rapport à une simple somme récursive.

Ne sommes-nous pas aussi lunaires que le Professeur *X.* lorsque nous choisissons d'ignorer le fait qu'un ordinateur ne peut manipuler ni l'ensemble des nombres entiers, ni les nombres réels et que nous produisons un code informatique utilisant un algorithme pensé pour des variables mathématiques ?

Pour montrer l'importance d'une transposition informatique correcte, nous présentons dans la section 3 deux exemples de transposition : un algorithme pour calculer la factorielle d'un entier (section 3.1) et un algorithme pour calculer la moyenne d'un ensemble de nombres réels (section 3.2). La section 2 fixe les notations et rappelle les informations pertinentes sur la représentation en machine des nombres entiers (section 2.1) et des nombres à virgule flottante suivant le standard IEEE 754 [1] (section 2.2).

2 L'arithmétique sur ordinateur

Nous ne rappelons ici que les éléments relatifs à l'arithmétique sur les nombres entiers et à virgule flottante suivant le format IEEE 754 [1] qui sont pertinents pour la suite de cet article. On pourra trouver un exposé plus complet avec des justifications historiques dans notre fascicule « *Le calcul sur ordinateur* » [8] accompagnant l'exposé de Jermann lors de la *Journée Académique 2023* de l'IREM des Pays de la Loire.

2.1 L'arithmétique sur les entiers

Les entiers sont représentés en machine par des chaînes binaires, généralement de tailles fixes. Pour des raisons techniques, on utilise usuellement des chaînes de 32 ou 64 bits. On distingue deux interprétations des chaînes binaires :

— L'interprétation *non signée*, où chaque chaîne ne peut représenter qu'un entier positif ou nul ; le décodage de la chaîne se fait en la considérant comme la représentation en base 2 du nombre ;

— L'interprétation *signée*, où la moitié des chaînes sert à coder des entiers négatifs et l'autre moitié des entiers positifs ou nuls.

Certains langages de programmation offrent des possibilités de manipulation des entiers avec une précision « *arbitraire* », où le nombre de bits utilisé varie en fonction des besoins. La contrepartie est un coût d'utilisation plus important car toutes les opérations sont alors effectuées par du code logiciel au lieu d'être traitées directement par le processeur de la machine. Contrairement à de nombreux langages comme C, C++ et Java, Python 3 offre en standard un seul type d'entiers, avec une précision variable, mais bornée. On trouvera dans l'annexe A de notre fascicule sur le calcul sur ordinateurs [8] une description détaillée de la mise en œuvre dans Python de ce type.

Si l'on souhaite manipuler des types entiers de taille fixe dans Python, on peut utiliser la bibliothèque Numpy, qui offre de nombreux types entiers : `uint8` pour des entiers non signés sur 8 bits et `int8` pour des entiers signés sur 8 bits, par exemple. On a aussi des entiers sur 32 bits (`uint32`, `int32`) et sur 64 bits (`uint64`, `int64`).

En fonction de la taille des chaînes de bits, on peut ainsi représenter les entiers

dans les domaines suivants, par exemple :

	Domaine	
Type	**(non-signé)**	**(signé)**
Entier sur 32 bits	$[0\,;\,2^{32}-1]$	$[-2^{31}\,;\,2^{31}-1]$
Entier sur 64 bits	$[0\,;\,2^{64}-1]$	$[-2^{63}\,;\,2^{63}-1]$
V_{32}	NA	$[-2^{30\times 2^{31}}+1\,;\,2^{30\times 2^{31}}-1]$
V_{64}	NA	$[-2^{30\times 2^{63}}+1\,;\,2^{30\times 2^{63}}-1]$

où V_{32} et V_{64} sont les deux types d'entiers en précision arbitraire que l'on trouve dans Python, en fonction de la mise en œuvre de l'interpréteur du langage. Pour ces types, il n'existe pas de déclinaison non signée.

Pour les types entiers de taille fixe t, tous les calculs sont faits modulo 2^t.

2.2 L'arithmétique à virgule flottante

Le standard IEEE 754 [1] définit un nombre à virgule flottante x par un triplet de trois valeurs :

- Le signe $s \in \{0, 1\}$;
- L'exposant $E \in [E_{\min}\,;\,E_{\max}]$;
- Le signifiant [2] $\sigma \in [0\,;\,2-2^{-p}]$; c'est un nombre fractionnaire exprimé en binaire, avec un bit de partie entière et p bits de partie fractionnaire f :

$$\sigma = b_0, \underbrace{b_{-1}b_{-2}\cdots b_{-p}}_{f}; \quad b_i \in \{0,1\}$$

On a alors :

$$x = (-1)^s \times \sigma \times 2^E$$

Afin d'assurer la continuité des calculs, le standard définit la notion d'infini, positif ou négatif : si un calcul crée un nombre d'une magnitude trop grande pour être représenté, il est remplacé par un infini (on a un *overflow*). On obtient ainsi une *droite réelle achevée*. Lorsqu'un calcul n'a pas de sens sur les réels (exemple : $\sqrt{-1}$), son résultat vaut *NaN* (*Not a Number*).

La figure 1 présente la répartition des nombres flottants sur la ligne réelle pour un format où l'exposant est codé sur 2 bits et le signifiant sur 4 bits.

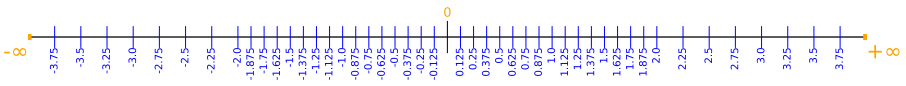


FIGURE 1 – Représentation d'un ensemble de nombres flottants (2 bits d'exposant et 4 bits de signifiant) sur la ligne réelle.

On voit que les nombres en virgule flottante (ou, simplement, *flottants*) sont répartis de manière non uniforme. La distance entre un nombre flottant de la forme $\sigma \times 2^E$

2. Parfois improprement appelé « *mantisse* ».

et le flottant suivant — à l'exclusion des infinis — est $2^{-p} \times 2^E$. On en déduit que si E est strictement supérieur à p, la distance entre un flottant et le suivant devient supérieure à 1. Donc, un type flottant ne peut représenter tous les entiers que dans le domaine $[-2^{p+1}\,;\,2^{p+1}]$. À l'extérieur de cet intervalle, certains entiers ne sont pas représentables.

L'ensemble des nombres flottants n'est pas fermé pour les opérations arithmétiques (la somme de deux flottants peut ne pas être un flottant, par exemple). Il est donc nécessaire d'*arrondir* chaque résultat non représentable. On note $\mathsf{fl}(v)$ le nombre flottant correspondant à la représentation en machine du nombre réel v. Le standard IEEE 754 garantit que le résultat arrondi est toujours le flottant le plus près de la valeur réelle.

L'accumulation des arrondis lors d'un calcul peut engendrer de grandes différences au final entre un résultat réel et la valeur calculée en machine. En particulier, on verra dans la suite l'importance de deux phénomènes :

- L'*absorption*. Lorsque l'on fait la somme ou la différence de deux flottants, on commence par modifier la valeur d'exposant le plus petit pour qu'elle ait le même exposant que le plus grand. On fait ensuite la somme des signifiants. Lorsque les deux nombres ont des magnitudes très différentes, on peut perdre une grande partie des bits significatifs du plus petit nombre. Exemple : soit à faire l'opération avec des signifiants sur 4 bits : $1,010 \times 2^{10} + 1,101 \times 2^2$:

$$\begin{array}{rlll} & 1,010 & \times 2^{10} & \\ + & 1,101 & \times 2^2 & \end{array} \longrightarrow \begin{array}{rlll} & 1,010 & & \times 2^{10} \\ + & 0,000\mathit{00001101} & & \times 2^{10} \end{array}$$

  Les chiffres en italique gris ne peuvent pas être représentés ; le calcul qui est réellement fait est donc : $1,010 \times 2^{10} + 0,000 \times 2^{10}$. La grande valeur a absorbé la petite ;
- La *cancellation*. À l'inverse, la soustraction de deux valeurs très proches issues d'un calcul ne garde que les bits issus des arrondis successifs. Exemple : soit à faire la soustraction avec des signifiants sur 8 bits $1,0010\mathit{111} \times 2^0 - 1,0010\mathit{010} \times 2^0$, où les bits en italique rouge sont entachés d'erreurs. On a :

$$\begin{array}{rlll} & 1,0010\mathit{111} & \times 2^0 \\ - & 1,0010\mathit{010} & \times 2^0 \\ \hline & 0,0000\mathit{101} & \times 2^0 \end{array}$$

  On obtient donc, après normalisation, la valeur $\mathit{1,01} \times 2^{-5}$, qui n'est composée que de bits incertains. La soustraction de valeurs proches n'ajoute pas d'erreur mais elle magnifie les erreurs précédentes.

Le standard IEEE 754 définit plusieurs tailles de nombres flottants. Nous ne nous intéresserons ici qu'au format dit en *double précision* avec les caractéristiques suivantes :

p	E_{min}	E_{max}	$\mathbf{D}_{\mathbb{Z}}$
52	-1022	$+1023$	$[-2^{53}, 2^{53}]$

où $D_{\mathbb{Z}}$ est le plus grand intervalle d'entiers relatifs qui sont tous représentables dans le format flottant.

Le format flottant double précision est celui utilisé par défaut pour représenter les nombres réels dans la plupart des langages de programmation. C'est notamment le cas pour le langage Python.

3 La transposition informatique

Le processus de passage d'une spécification écrite dans le langage mathématique à un programme dans un langage de programmation exécutable sur une machine est rarement explicité en détails dans la littérature. Un des premiers articles sur le sujet, celui de Balacheff [3], précise bien, cependant, que « *ce que l'on place habituellement sous le terme d'informatisation ne constitue pas une simple translitération [...]* ». Il insiste aussi [2] sur le fait qu'« *en tant que dispositif matériel, l'ordinateur impose un ensemble de contraintes qui, elles-mêmes, vont exiger une transformation appropriée pour permettre la mise en œuvre de la représentation adoptée. [...] Les exigences propres à une modélisation computable tiennent à sa vocation à permettre la mise en œuvre* ***autonome***[3] *d'un modèle symbolique par un dispositif informatique [...]* » On a là tous les éléments importants d'une transposition informatique correcte :

- Prise en compte des limitations de l'*univers interne* de l'ordinateur ;
- Autonomisation du processus de calcul : un programme informatique doit pouvoir s'exécuter sur les données d'entrée sans intervention d'un tiers régulateur.

Cependant, Balacheff choisit sciemment (« *nous n'entrerons pas, ici, dans l'analyse des contraintes profondes de représentation et de mise en œuvre des langages de programmation utilisés* » [3]) de ne pas détailler la prise en compte de ces éléments, ce qui semble l'induire plus tard en erreur en ce qui concerne leur impact sur la modélisation (« [...] *j'assimilerai les contraintes de cet univers* [l'univers interne de la machine] [...] *aux contraintes d'expression d'un modèle dans un langage de programmation* »).

On retrouve cette erreur dans les écrits postérieurs sur le processus de transposition informatique : pour Briant et Bronner [6], la pensée informatique s'intègre à la pensée mathématique initiale en la complétant, pas en s'y substituant. De même, Modeste [13] définit le pseudo-code utilisé dans la première étape de transposition comme un « *un langage intermédiaire, inspiré des instructions des langages informatiques mais libéré de certaines contraintes et manipulant directement les objets mathématiques.* »

Comme on va le voir dans les exemples ci-dessous, la prise en compte de l'univers interne de la machine a un impact majeur sur le pseudo-code à écrire, qui ne peut s'expliquer ni se justifier si l'on travaille sur les variables mathématiques.

3.1 Exemple 1 : calcul de la factorielle de n

On souhaite écrire un programme calculant la factorielle $n!$ d'un entier n, définie récursivement par :

$$\begin{cases} 0! & = 1 \\ 1! & = 1 \\ n! & = n \times (n-1)!, \quad n > 1 \end{cases} \tag{2}$$

Le pseudo-code s'en déduit simplement (programme 3). De même, un code Python s'obtient facilement, une fois pris en compte les contraintes de syntaxe du langage (programme 4).

Mais ce code est-il correct ? Lors de l'écriture du pseudo-code, nous n'avons jamais explicité le fait que n devait être un entier naturel car cela allait de soi. Dans un programme informatique, *rien ne va de soi* et cette contrainte devrait apparaître dans notre code Python. Il y a plusieurs manières de faire cela : on peut tester à l'entrée dans la fonction que la valeur `n` est un entier (du type Python `int`) positif ou nul, ou

3. C'est nous qui soulignons.

Programme 3 – Pseudo-code pour le calcul de la fonction factorielle.

fonction factorielle(n)
début
  si $n \leqslant 1$ **alors**
    retourner 1
  sinon
    retourner $n \times$ factorielle$(n-1)$
  finsi
fin

Programme 4 – Transposition informatique directe du pseudo-code pour calculer la fonction factorielle en Python.

```
def factorielle(n):
    if n <= 1:
        return 1
    else:
        return n*factorielle(n-1)
```

l'on peut mettre un commentaire dans la documentation de la fonction précisant le type du paramètre attendu. Le programme 5 montre une solution combinant les deux, en évitant les tests de type à chaque appel récursif grâce à une fonction auxiliaire `factorielle_aux()`.

3.1.1 Mise en œuvre avec des entiers de taille fixe

Pour obtenir de meilleurs performances en temps ou en occupation mémoire, on peut être tenté d'utiliser un type entier de taille fixe tel que ceux offerts par la bibliothèque Numpy, comme ce serait le cas dans la plupart des langages de programmation, tels que C, C++, ou Java. Le programme 6 présente la fonction factorielle calculée avec des entiers non signés (c'est à dire, uniquement positifs ou nuls) sur 64 bits. On vérifie que le paramètre n est d'un type compatible avec un entier (`int`, `numpy.int32`, ...) avec la fonction `isinstance()`, et que sa valeur est codable dans un entier non signé sur 64 bits ($[0\,;\,2^{64}-1]$). On a été obligé de choisir une convention pour la valeur à retourner lorsque la fonction est appelée avec une entrée trop grande ; on a choisi de retourner la valeur 0, suffisamment pathologique dans le cadre d'un calcul de factorielle pour avertir l'utilisateur qu'il y a eu un problème. On a maintenant un algorithme différent du pseudo-code initial du programme 3. Ce nouvel algorithme n'a, par ailleurs, aucun sens dans un pseudo-code qui considère des variables mathématiques à valeur dans $\mathbb{N}$.

Pour s'assurer de la correction des calculs, on a écrit un petit programme vérifiant l'invariant :

$$\frac{n!}{(n-1)!} = n$$

pour n de 1 à 30. La figure 2 compare les valeurs attendues avec les résultats obtenus par la fonction `factorielle()` du programme 6. Pour $n > 20$, la fonction re-

Programme 5 – Calcul de la factorielle de n en Python. Test sur le type de l'entrée.

```python
def factorielle(n):
    """
    Calcul de la factorielle de `n`.

    Le paramètre `n` doit être un entier naturel.
    On retourne `0` si ce n'est pas le cas.
    """
    def factorielle_aux(n):
        """
        Fonction auxiliaire: `n` est assuré d'être
        un entier naturel.
        """
        if n <= 1:
            return 1
        else:
            return n*factorielle_aux(n-1)

    # On vérifie que `n` est un entier naturel
    if isinstance(n,int) and n >= 0:
        return factorielle_aux(n)
    else:
        return 0
```

Programme 6 – Calcul de la factorielle de n avec des entiers non signés sur 64 bits.

```python
import numbers
from numpy import uint64

def factorielle(n):
    """
    Calcul de la factorielle de `n`.

    Le paramètre `n` doit être un entier représentable sur 64 bits.
    On retourne `0` si ce n'est pas le cas.
    """
    if isinstance(n,numbers.Integral) and 0 <= n <= 2**64-1:
        n = uint64(n)
        if n <= 1:
            return uint64(1)
        else:
            return n*factorielle(n-uint64(1))
    else:
        return uint64(0)
```

tourne des valeurs parfois plausibles (exemple : `factorielle(21)` retourne la valeur **1419745402429033768**, qui est plus grande que 20!) qui ne vérifient pas l'invariant : avec des entiers non signés sur 64 bits, tous les calculs sont faits modulo 2^{64} ; or, 21! est supérieure à cette valeur.

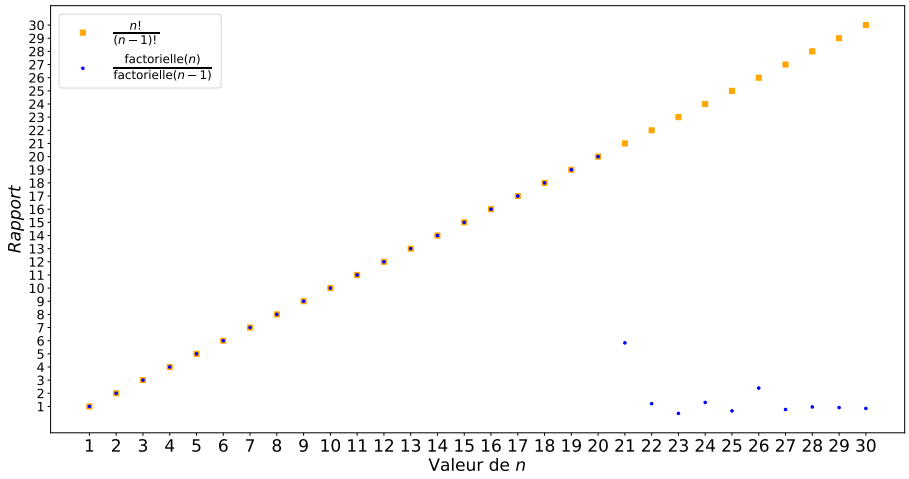


FIGURE 2 – Vérification des valeurs calculées par la fonction Python `factorielle()` du programme 6.

Ainsi, malgré ce que dit le commentaire en tête de la fonction `factorielle()` dans le programme 6, on constate que tous les entiers de l'intervalle $[0\,;2^{64}-1]$ ne sont pas admissibles en entrée, car la valeur de sortie doit aussi être représentable sur 64 bits. On a un problème dès que le nombre de bits nécessaire pour représenter une factorielle est supérieur à celui alloué pour représenter les entiers. La formule de KAMENETSKY [17], basée sur la formule de STIRLING pour calculer une approximation de la fonction factorielle, nous permet de connaître le nombre de bits nécessaires pour représenter $n!$ sans avoir à calculer la factorielle elle-même :

$$\text{Kamenetsky}(n) = \left\lceil \frac{\log_2(2n\pi)}{2} + n\log_2\frac{n}{\mathrm{e}} \right\rceil$$

La figure 3 montre le nombre de bits nécessaire pour représenter $n!$ pour n dans le domaine $[3\,;50]$. On a identifié sur la figure les plus petites valeurs de n pour lesquelles il faut plus de 31, 32, 53, 63 et 64 bits, ce qui correspond à des formats usuels de représentation des entiers en machine. On voit ainsi que si l'on utilise le format `int32` (entier signé sur 32 bits, n'offrant donc que 31 bits pour le plus grand entier positif), habituel dans des langages informatiques comme C, C++ et Java, ou le format `uint32` (entier non signé sur 32 bits), on ne peut pas représenter la factorielle d'un entier supérieur à 12. Avec des entiers sur 63 ou 64 bits, on ne peut pas représenter la factorielle d'un entier supérieur à 20.

Un programme correct doit donc intégrer dans son algorithme les tests permettant de rejeter les appels qui ne pourraient être traités correctement, sous peine de retourner des valeurs fausses, parfois difficilement distinguables des valeurs correctes.

Le programme 7 modifie le test de domaine de la fonction factorielle du programme 6 pour garantir, désormais, que la valeur de sortie se trouve bien dans le domaine de validité des calculs sur des entiers de 64 bits.

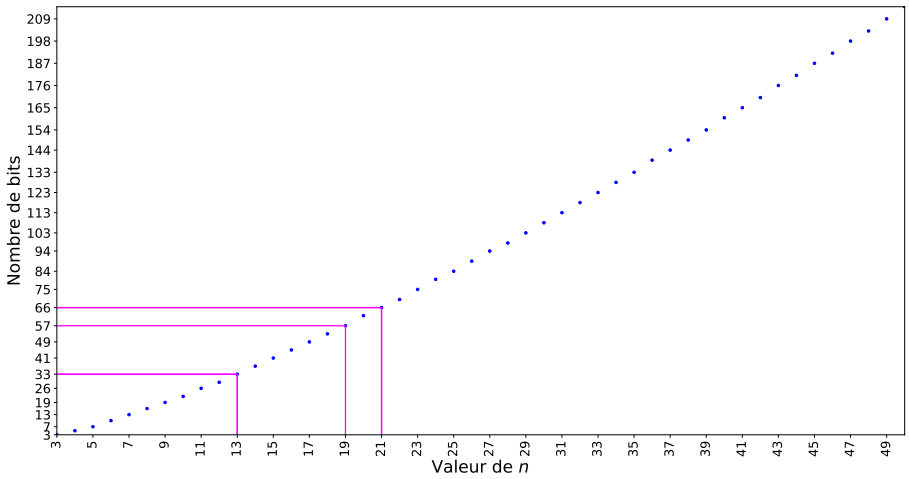


FIGURE 3 – Nombre de bits nécessaires pour représenter $n!$

Programme 7 – Calcul de la factorielle de n avec des entiers non signés sur 64 bits intégrant un test sur le domaine de validité.

```
import numbers
from numpy import uint64

def factorielle(n):
    if isinstance(n,numbers.Integral) and 0 <= n <= 20:
        n = uint64(n)
        if n <= 1:
            return uint64(1)
        else:
            return n*factorielle(n-uint64(1))
    else:
        return uint64(0)
```

3.1.2 Mise en œuvre avec des entiers de taille variable

On peut cependant objecter qu'il n'est nullement nécessaire de se limiter à des entiers sur 64 bits, en particulier avec Python, qui, contrairement à la plupart des langages de programmation, utilise par défaut une représentation des entiers en précision arbitraire [8, p. 53–54]. Bien évidemment, la représentation n'est pas infinie mais, dans le moins favorable des cas, elle permet, en théorie, de manipuler tous les entiers positifs dans le domaine $[0, 2^{30\times 2^{31}} - 1]$, ce qui est énorme. Essayons de tirer partie des capacités de calcul étendu de Python pour calculer quelques grandes valeurs de la fonction factorielle présentée dans le programme 4 :

```
>>> factorielle(500)
1220136825991110068701238785423046926253574342803192842192413588385845373156
↪  3881997605496447502203281863013616477148203584163378722078177200480785200
↪  0515932928547790757193933060377296085908627042917454788242491272634430500
↪  6701732707694610628023104526442188787894657547771498634943677810376442700
↪  4033827365397471386477878495438489595537537990423241061271326984327745700
↪  1554630997720278101456108118837370953101635632443298702956389662891165800
↪  9747695720879269288712817800702651745077684107196243903943225364226052300
↪  4945850129918571501248706961568141625359056693423813008856249246891564100
↪  2677565448188650659384795177536089400574523894033579847636394490531306200
↪  3237490664450488246650759467358620746379251842004593696929810222639719500
↪  2597190945217823331756934581508552332820762820023402626907898342451712000
↪  0620771464097945611612762914595123722991334016955236385094288559201872700
↪  4337951730145863575708283557801587354327688886801203998823847021514676000
↪  5445407663535984174430480128938313896881639487469658817504506926365338100
↪  7505547812864000000000000000000000000000000000000000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000000
>>> factorielle(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
[...]
RecursionError: maximum recursion depth exceeded in comparison
```

Tout se passe effectivement bien lorsque l'on calcule $500!$, un nombre dont la représentation requiert plus de 3700 bits. Par contre, on a une erreur lors du calcul de $1000!$, alors que cette factorielle ne requiert que 8530 bits, ce qui est bien en deçà des possibilités de Python.

On a, en fait, atteint ici une limite du langage Python, qui n'autorise pas plus de 1000 appels de fonction imbriqués (pour des raisons techniques, la borne précise est légèrement plus basse dans notre cas). Malgré l'usage d'une représentation des entiers pratiquement non bornée, on se retrouve donc limité [4] à des valeurs de n inférieures à 1000.

Alors, faut-il reprendre la fonction `factorielle()` du programme 4 et lui ajouter un test sur la valeur maximale autorisée de n, comme on l'a fait dans le programme 7 ? Non, car nous pouvons nous passer de la récursion problématique en réécrivant notre fonction sous une forme itérative (programme 8).

Notre seule limite, maintenant, semble être celle de la représentation des entiers en précision arbitraire de Python. On a vu qu'elle est suffisamment élevée pour répondre à la plupart des demandes. Il est cependant difficile de déterminer une borne précise pour la valeur maximale du paramètre de la fonction `factorielle()` car elle dépend de facteurs extérieurs au programme (paramétrage du système d'exploitation de la machine, taille de la mémoire disponible, …). La fonction `factorielle()` du programme 8 accepte un paramètre de n'importe quel type entier en entrée. Par contre, le type de sortie est toujours « `int`. » C'est pourquoi on effectue une *promotion* de `n` dans le type « `int` » avant la structure répétitive.

4. Il reste néanmoins possible de changer la borne du nombre d'appels avec la fonction `sys.setrecursionlimit()`.

Programme 8 – Calcul itératif de la factorielle de n.

```python
import number

def factorielle(n):
    """
    Calcul de la factorielle de `n`.
    Le paramètre d'entrée peut être de n'importe quel type entier.
    La sortie est toujours de type `int`.
    Retourne 0 si l'entrée n'est pas un entier.
    """
    if isinstance(n,number.Integral):
        n = int(n)
        res = 1
        while n > 1:
            res = res * n
            n = n - 1
            return res
    else:
        return 0
```

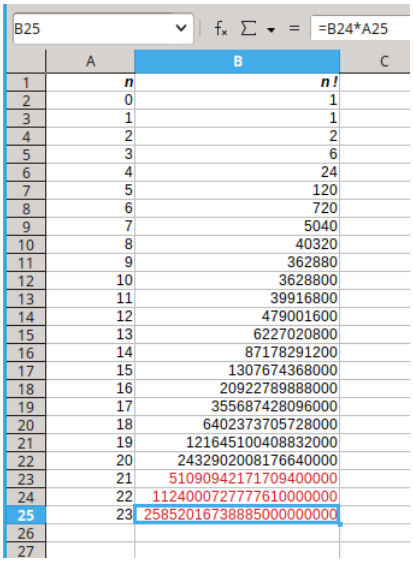
Le programme 8 est complètement différent de l'algorithme en pseudo-code du programme 3. Si l'on ne considère que les variables mathématiques, il n'y a aucune raison d'écrire un pseudo-code itératif plutôt que récursif, bien plus proche de la spécification mathématique de l'équation (2). Seules des considérations de mise en œuvre de l'algorithme nous ont conduit à adopter un algorithme itératif. On voit donc ici un premier échec de l'approche de Briant [5] en deux étapes pour la transposition informatique.

3.1.3 Mise en œuvre avec d'autres langages et applications

La majorité, sinon la totalité, des tableurs ne manipulent que des valeurs de type flottant double précision, même lorsque l'on souhaite utiliser des entiers. C'est par exemple le cas de LibreOffice. La figure 4 montre le calcul de la factorielle des entiers de 0 à 23 avec LibreOffice 7.5.1.2, où les valeurs en rouge sont erronées, comme on peut s'y attendre dès lors que l'on utilise un type flottant double précision à la place d'un type entier : dans ce format, seul l'ensemble des entiers dans l'intervalle $D_{\mathbb{Z}} = [-2^{53}, 2^{53}]$ peut être entièrement représenté. De 2^{53} à 2^{54} seuls les entiers pairs sont représentables ; de 2^{54} à 2^{55}, seuls les multiples de 4 le sont, … L'intervalle de calcul sur les entiers que l'on peut garantir correct est donc plus petit que si l'on utilise un format entier sur 64 bits.

Un des atouts mis en avant pour l'utilisation du langage de programmation AlgoBox dans l'enseignement de l'algorithmique est sa proximité syntaxique avec du pseudo-code. Pour s'en convaincre, il suffit de comparer la définition de la fonction factorielle dans le programme 9 avec le pseudo-code du programme 3.

Contrairement à Python et à la plupart des autres langages de programmation classiques, AlgoBox ne reconnait qu'un seul type numérique : NOMBRE. La documentation ajoute à la possibilité de confusion du langage avec du pseudo-code en ne fournissant aucune indication sur la représentation en machine du type. S'agit-il d'un type magique capable de représenter toutes les valeurs de $\mathbb{Z}$ et $\mathbb{R}$? Seul un petit paragraphe intitulé « *Calculs numériques et arrondis* » peut nous mettre la puce à l'oreille lorsqu'il évoque la possibilité d'erreurs d'arrondi liées à la représentation interne des nombres.



	A	B	C
1	*n*	*n !*	
2	0	1	
3	1	1	
4	2	2	
5	3	6	
6	4	24	
7	5	120	
8	6	720	
9	7	5040	
10	8	40320	
11	9	362880	
12	10	3628800	
13	11	39916800	
14	12	479001600	
15	13	6227020800	
16	14	87178291200	
17	15	1307674368000	
18	16	20922789888000	
19	17	355687428096000	
20	18	6402373705728000	
21	19	121645100408832000	
22	20	2432902008176640000	
23	21	5109094217170940000 0	
24	22	112400727777610000000	
25	23	2585201673888500000000 0	
26			
27			

FIGURE 4 – Calcul de la fonction factorielle avec LibreOffice 7.5.1.2. Les valeurs en rouge sont erronées.

Programme 9 – Fonction AlgoBox pour calculer la fonction factorielle.

```
FONCTION factorielle(n)
VARIABLES_FONCTION
DEBUT_FONCTION
    SI (n < 2) ALORS
        DEBUT_SI
            RENVOYER 1
        FIN_SI
    SINON
        DEBUT_SINON
            RENVOYER n*factorielle(n-1)
        FIN_SINON
FIN_FONCTION
```

Mais, insiste le même paragraphe, ces arrondis n'affectent pas les entiers, donc nous n'avons pas de soucis à nous faire en ce qui concerne notre fonction factorielle().

Afin de nous assurer de la correction des calculs, nous avons refait avec un programme AlgoBox le test illustré par la figure 2 et nous avons calculé la factorielle des entiers de 1 à 69, en divisant à chaque fois $n!$ par $(n-1)!$ pour vérifier que nous obtenions bien n. Le programme était écrit de façon à afficher les valeurs de n pour lesquelles le test échouait. De nombreuses valeurs échouaient au test.

Après consultation du code source d'AlgoBox, il apparait que le type NOMBRE n'est qu'un alias pour le format IEEE 754 en virgule flottante double précision. Comme pour LibreOffice, en dehors de l'intervalle $D_{\mathbb{Z}}$, certains entiers doivent être arrondis. Contrairement à la promesse de la documentation d'AlgoBox, les entiers sont donc bien concernés par les problèmes d'arrondi, eux aussi.

AlgoBox propose par ailleurs une fonction **ALGOBOX_FACTORIELLE(n)** pour calculer la factorielle de n. La documentation précise que n doit être strictement inférieur à 70. C'est une borne bien optimiste si l'on considère qu'à partir de 19 (figure 3), le type NOMBRE ne possède plus assez de précision pour garantir la représentation des résultats. Incidemment, la formule de KAMENETSKY indique qu'il faut 327 bits pour représenter $69!$; c'est bien au-delà de ce que peut manipuler un ordinateur avec les types numériques usuels. La confusion peut venir du fait que le format flottant double précision peut représenter de très grands nombres, jusqu'à environ 10^{308} (ce qui correspondrait à un type entier non signé de $\lfloor 308 \log_2 10 \rfloor = 1023$ bits), mais sa précision ne peut garantir la représentation de tous les entiers jusqu'à cette borne.

3.2 Exemple 2 : calcul de la moyenne de n réels

On souhaite maintenant écrire une fonction pour calculer la moyenne d'une série de valeurs réelles :

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{3}$$

Le pseudo-code s'obtient simplement avec une boucle **pour** (programme 10) et ressemble — la division finale exclue — au pseudo-code du programme 1. Notons que notre pseudo-code reçoit un tuple en entrée, mais ce pourrait être n'importe quel ensemble non ordonné, comme l'addition est commutative et associative sur $\mathbb{R}$.

Programme 10 – Pseudo-code pour le calcul de la moyenne d'une série de n réels.

fonction moyenne$\big((x_1, \ldots, x_n)\big)$
début
  $S \leftarrow 0$
  pour chaque $x_i \in (x_1, \ldots, x_n)$ **faire**
    $S \leftarrow S + x_i$
  finpour
  retourner $\frac{S}{n}$
fin

On trouve le code Python correspondant dans de nombreuses références, par exemple le fascicule de ressources pour le lycée, série Mathématiques, « *Algorithmique et programmation* »[20, p. 7] sur Éduscol (programme 11).

Programme 11 – Transposition directe en Python du pseudo-code de calcul d'une moyenne.

```
def moyenne(serie):
    n = len(serie)
    S = 0
    for xi in serie:
        S = S + xi
    return S/n
```

3.2.1 Transposition informatique

Comme on l'a évoqué au début de la section 3, un code informatique diffère d'un pseudo-code en ce qu'il doit être une mise en œuvre *autonome*, capable de gérer toutes les entrées possibles. La fonction `moyenne()` est susceptible d'être appelée par d'autres fonctions sur lesquelles on n'a aucune prise. Que se passe t-il si la série donnée en entrée contient des valeurs infinies ? Des *Not a Number* ? En Python, on pourrait même imaginer que des valeurs non numériques polluent l'entrée (chaînes de caractère, …), voire que cette entrée ne soit pas *itérable*—c'est à dire, pas un ensemble de valeurs que l'on peut énumérer mais, par exemple, un scalaire. Un code correct se doit d'en tenir compte, soit en prévenant l'utilisateur d'une erreur, soit en éliminant silencieusement ces valeurs de la série avant de calculer la moyenne. Dans tous les cas, le choix fait doit apparaître clairement dans la documentation de la fonction. Dans la suite, par simplicité, on supposera que le paramètre en entrée est une liste ou un tuple dont les valeurs sont au moins numériques.

Par ailleurs, si le code du programme 11 est exécuté dans un interpréteur Python version 2.x et que la série ne contient que des valeurs entières, la division finale sera aussi une division entière. Le remplacement de la ligne « `S = 0` » par « `S = 0.0` » garantirait au moins que les calculs qui suivent sont faits sur les nombres flottants.

De même, le programme 11 génèrera une erreur s'il est appelé avec une série vide. À titre d'exemple, le programme 12 montre une mise en œuvre possible de la fonction `moyenne()` qui teste ses entrées : comme on prévoit de faire tous les calculs en double précision, on s'assure que la série ne contient que des valeurs flottantes d'une précision égale ou inférieure (demie précision ou simple précision). De même, on interdit la présence d'entiers dans la série car ils pourraient être plus grands que ce que l'on peut représenter avec des flottants double précision [5]. Un entier doit simplement être écrit avec une partie fractionnaire nulle pour être accepté (« `1.0` » au lieu de « `1` », par exemple). Si toutes les conditions ne sont pas réunies pour calculer une moyenne, on retourne un NaN, qui signale une erreur. On choisit aussi d'éliminer silencieusement les valeurs indésirables (NaNs et infinis).

3.2.2 Mise en œuvre tenant compte des propriétés des flottants

Les programmes 11 et 12 accumulent les valeurs de la série dans l'ordre dans lequel elles y apparaissent. Cela rend le processus de somme sensible aux erreurs d'absorption et de cancellation. On en a vu un exemple dans le programme 2 pour la série de l'équation 1 en page 2, où la moyenne calculée serait 0 au lieu de $\frac{3}{7}$. Le problème du

5. En toute rigueur, on pourrait au moins tester que les entiers sont dans le domaine $\mathbf{D}_{\mathbb{Z}}$ avant de rejeter la série, ou bien même, qu'ils sont représentables dans notre format flottant, ce que l'on ne fait pas ici par soucis de simplicité.

Programme 12 – Calcul de la moyenne avec filtrage des entrées.

```
import numbers
import numpy
import math

def moyenne(serie):
    """
    Calcul de la moyenne de la liste ou du tuple `serie`. On suppose
    que `serie` est un itérable qui ne contient que des valeurs numériques.
    """
    # Recherche de valeurs qui ne sont pas des flottants
    # en demie, simple, ou double précision.
    badValues = list(filter(lambda x:not isinstance(x,(numpy.half,
                                                       numpy.single,
                                                       numpy.double,float)),
                            serie))
    if len(badValues) != 0:
        return math.nan
    # Suppression silencieuse des valeurs math.nan et math.inf
    serie = list(filter(lambda x:math.isfinite(x),serie))
    n = len(serie)
    if n == 0:
        return 0.0
    else:
        s = 0.0
        for xi in serie:
            s = s + float(xi)
        return s/n
```

calcul correct d'une moyenne est réductible à celui du calcul de la somme de la série car la division finale ne peut introduire qu'une erreur minime du fait des propriétés d'arrondi des opérations arithmétiques garanties par le standard IEEE 754. Dans la suite, on va donc se concentrer sur celui-ci.

De nombreux algorithmes informatiques plus ou moins sophistiqués ont été et sont encore développés aujourd'hui pour sommer une série de valeurs flottantes le plus précisément possible [12, 15, 14, 4]. Kahan [11], par exemple, a proposé dès 1965 d'exploiter le fait que l'erreur commise lors de la somme de deux nombres flottants a et b était toujours un nombre flottant représentable, que l'on pouvait réinjecter dans le calcul. On obtient ainsi l'algorithme de *somme compensée* du programme 13.

Avec le programme 13, on obtient une moyenne de $\mathsf{fl}\left(\frac{4}{7}\right)$ pour la série de l'équation (1), plus satisfaisante que la valeur 0 obtenue avec la somme récursive. Mais, on peut faire encore mieux avec l'algorithme de Shewchuk [16], mis en œuvre dans le programme 14 : on utilise un tableau intermédiaire dans lequel on insère les sommes partielles et les erreurs rencontrées lors de l'ajout de chaque nouvelle valeur de la série. Ce deuxième tableau ne croît que s'il y a une erreur d'arrondi lors d'une itération. On finit par une somme récursive de toutes les sommes partielles et leurs erreurs. Cet algorithme nous permet d'obtenir la bonne moyenne pour notre série, arrondie au flottant représentable le plus proche : $\mathsf{fl}\left(\frac{3}{7}\right)$.

Il existe des algorithmes encore plus précis que celui du programme 14, qui peuvent garantir l'évaluation au plus près de la moyenne d'une série. On peut cependant avoir des surprises même dans ce cas, lorsque l'on appelle la fonction `moyenne()` en lui passant une série de valeurs qui doivent être arrondies pour être représentées dans le format flottant double précision. On peut ainsi découvrir que la moyenne d'une série se trouve en dehors de la plage définie par ses valeurs :

```
>>> moyenne([2.0**53+1, 2.0**53+1]) == 2.0**53
```

Programme 13 – Calcul de la moyenne avec réinjection des erreurs d'arrondi.

```
import numbers
import numpy
import math

def moyenne(serie):
    # Recherche de valeurs qui ne sont pas des flottants
    # en demie, simple, ou double précision.
    badValues = len(filter(lambda x:not isinstance(x,(numpy.half,
                                                      numpy.single,
                                                      numpy.double,float)),
                           serie))
    if len(badValues) != 0:
        return math.nan
    # Suppression silencieuse des valeurs math.nan et math.inf
    serie = list(filter(lambda x:math.isfinite(x),serie))
    n = len(serie)
    if n == 0:
        return 0.0
    else:
        (s, e) = (0.0, 0.0)
        for xi in serie:
            y = float(xi) - e
            t = s + y
            e = (t - s) - y # Calcul de l'erreur courante
            s = t
        return s/n
```

```
True
```

La valeur $2^{53}+1$ est en dehors de l'ensemble $\mathbf{D}_{\mathbb{Z}}$ pour les flottants double précision et est arrondie à la valeur 2^{53} *avant* l'appel de la fonction `moyenne()`.

Pour chaque algorithme de somme, on est capable de déterminer mathématiquement une borne sur l'erreur commise quelle que soit la série à sommer, ce qui nous permet de choisir l'algorithme le plus adéquat en fonction de la précision attendue et de la dégradation des performances que l'on peut admettre.

Chacun des algorithmes de somme plus précis sur les nombres flottants que l'algorithme direct de somme récursive n'a aucun sens si l'on manipule des variables réelles. C'est pourquoi, ils ne peuvent être obtenus par le processus usuel de transposition informatique. Pourtant, il est crucial de les utiliser à la place de l'algorithme du programme 11 issu du pseudo-code sur les variables mathématiques (programme 10) si l'on veut avoir des garanties sur les résultats calculés en machine.

4 Conclusion

Que de chemin parcouru entre les programmes Python 2 et 11 d'une part, travestis de pseudo-code mathématique, et les programmes Python 8 et 14 d'autre part, spécifications exécutables capables de gérer toutes les entrées possibles en fournissant des résultats avec précision ! Pour obtenir ces derniers programmes, il nous faut connaitre, entre autres, les détails de l'arithmétique entière et de l'arithmétique à virgule flottante telle que spécifiée par le standard IEEE 754, ainsi que leurs spécificités dans le langage de programmation utilisé. Peut-on réellement exiger un tel niveau de connaissance des élèves, voire même des enseignants en collège et lycée ? Évidemment, non ! L'important est que les élèves intègrent la nécessité d'expliciter tous les

Programme 14 – Calcul de la moyenne avec accumulation des sommes compensées.

```
import numbers
import numpy
import math

def moyenne(serie):
    """
    Calcul de la moyenne des valeurs de `series`.
    Le paramètre d'entrée ne doit contenir que des valeurs
    flottantes en demie, simple, ou double précision.
    Tous les calculs sont faits en double précision.
    Retourne `math.nan` si la série d'entrée contenait des
    valeurs invalides.
    """
    # Recherche de valeurs qui ne sont pas des flottants
    # en demie, simple, ou double précision.
    badValues = list(filter(lambda x:not isinstance(x,(numpy.half,
                                                       numpy.single,
                                                       numpy.double,float)),
                            serie))
    if len(badValues) != 0:
        return math.nan
    # Suppression silencieuse des valeurs math.nan et math.inf
    serie = list(filter(lambda x:math.isfinite(x),serie))
    n = len(serie)
    if n == 0:
        return 0.0
    else:
        partielles = []
        for xi in serie:
            xi = float(xi)
            i = 0
            for v in partielles:
                if abs(xi) < abs(v):
                    (xi,v) = (v,xi)
                s = xi+v
                e = v - (s - xi)
                if e != 0.0:
                    partielles[i] = e
                    i += 1
                xi = s
            partielles[i:] = [xi]
        S = 0.0
        for xi in partielles:
            S = S + xi
        return S/n
```

traitements dès lors que le code est censé être exécuté par une machine et non plus lu par un opérateur intelligent, et qu'ils soient bien conscients des limites d'un code qui reprendrait un algorithme mathématique. La transposition informatique exige un véritable travail d'adaptation au-delà des aspects syntaxiques liés à un langage de programmation particulier, qui demande une bonne connaissance des propriétés de la machine et de son arithmétique. Les élèves doivent comprendre et se souvenir en permanence qu'un programme est exécuté sur des ensembles de nombres qui n'ont pas les mêmes propriétés que les ensembles mathématiques. Il n'est donc pas possible de réutiliser un algorithme mathématique tel quel et espérer des résultats corrects.

À cet égard, l'utilisation d'un langage de programmation comme AlgoBox pour l'initiation à l'algorithmique s'avère problématique car il entretient la confusion déjà présente dans les programmes officiels [9] entre l'algorithmique mathématique et l'algorithmique informatique en utilisant une syntaxe très proche du pseudo-code employé généralement pour décrire un algorithme mathématique et en définissant un type flou « NOMBRE » sans propriétés ni bornes bien définies pour les variables numériques, alors qu'il est mis en œuvre par un simple type flottant en double précision avec tous ses manques et ses faiblesses.

La réécriture d'un algorithme mathématique pour s'adapter aux possibilités de la machine possède une histoire ancienne et, à la fin des années 1940, Alan TURING lui-même était réputé pour son ingéniosité dans l'écriture de programmes où l'ordre des instructions ne suivait pas la logique de la procédure mathématique mais s'adaptait aux possibilités des mémoires à mercure à lecture/écriture séquentielle de l'EDSAC de l'université de Cambridge [19].

Le code du programme 14 n'est pas nouveau et, avant SHEWCHUK, KAHAN proposait déjà dans les années 1960 un code informatique pour calculer la moyenne d'une série plus précis que la simple somme récursive issue du pseudo-code mathématique. Pourtant, de nombreux outils continuent de proposer un code ressemblant à celui du programme 11 (exemple : la fonction `sum()` de Python[6]). L'immense majorité des ressources pédagogiques consultées pour la rédaction de cet article fait référence aux problèmes de représentation des réels sur une machine, introduit superficiellement la représentation des nombres flottants au format IEEE 754, puis traduit comme si de rien n'était des algorithmes mathématiques en programmes informatiques sans tenir compte du changement d'ensemble utilisé.

En 1967, FORSYTHE [7] présentait les difficultés rencontrées dans l'implémentation d'une « simple » procédure de résolution d'équation quadratique et se désolait que, malgré l'existence d'un code correct proposé quelques années auparavant par William KAHAN, il n'existait probablement pas plus de cinq implémentations correctes dans le monde. Si le nombre a probablement augmenté aujourd'hui, l'utilisation d'un code correct est encore loin d'être universelle. Il est plus que temps que les élèves intègrent massivement tant les difficultés et les pièges de la transposition informatique que l'absolue possibilité d'obtenir un code robuste et correct par la connaissance des propriétés et des limites de la machine, afin qu'ils sachent l'exiger de la part de tous les implémenteurs de bibliothèques numériques.

6. Python propose, cependant, une fonction spécialisée, `math.fsum()`, qui utilise l'algorithme de SHEWCHUK [16, 10], mais qui la connait et l'utilise au quotidien ?

Remerciements

Notre intérêt pour la notion de transposition informatique prend sa source dans une invitation par Magali Hersant, directrice de l'Institut de Recherche en Enseignement des Mathématiques (IREM) des Pays de la Loire et son co-directeur Emmanuel Desmontils, à venir présenter l'arithmétique des ordinateurs lors de la journée académique de l'IREM 2023 à Nantes.

Références

[1] IEEE Standard for Floating-Point Arithmetic. Rapport technique IEEE Std 754-2019 (Révision de IEEE 754-2008), juillet 2019.

[2] Nicolas Balacheff : Didactique et intelligence artificielle. *Recherches en Didactique des Mathematiques*, 14:9, 1994.

[3] Nicolas Balacheff : La transposition informatique, un nouveau problème pour la didactique. *In Colloque « Vingt ans de didactique des mathématiques en France », 15-17 juin 1993*, pages 364–370. La Pensée Sauvage, 1994.

[4] Pierre Blanchard, Nicholas J. Higham et Theo Mary : A Class of Fast and Accurate Summation Algorithms. *SIAM Journal on Scientific Computing*, 42(3): A1541–A1557, janvier 2020.

[5] Nathalie Briant : *Étude didactique de la reprise de l'algèbre par l'introduction de l'algorithmique au niveau de la classe de seconde du lycée français.* Thèse de doctorat, Université Montpellier 2, décembre 2013.

[6] Nathalie Briant et Alain Bronner : Étude d'une transposition didactique de l'algorithmique au lycée : une pensée algorithmique comme un versant de la pensée mathématique. *In Actes de «Espace Mathématique Francophone»*, octobre 2015.

[7] George E. Forsythe : What is a satisfactory quadratic equation solver ? Technical Report CS74, Computer Science Department, Stanford University, août 1967.

[8] Frédéric Goualard et Christophe Jermann : Le calcul sur ordinateur. Fascicule pour la journée académique de l'IREM, IREM des Pays de la Loire, Nantes, avril 2023.

[9] Mariam Haspekian et Claver Nijimbéré : Favoriser l'enseignement de l'algorithmique en mathématiques : une question de distance aux mathématiques ? *Éducation et didactique*, 10(3):121–135, décembre 2016.

[10] Raymond Hettinger : Binary floating point summation accurate to full precision, mars 2005. https://code.activestate.com/recipes/393090-binary-floating-point-summation-accurate-to-full-p/.

[11] W. Kahan : Pracniques : further remarks on reducing truncation errors. *Communications of the ACM*, 8(1):40, janvier 1965.

[12] Peter Kornerup, Vincent Lefevre, Nicolas Louvet et Jean-Michel Muller : On the Computation of Correctly Rounded Sums. *IEEE Transactions on Computers*, 61(3):289–298, mars 2012.

[13] Simon Modeste : *Enseigner l'algorithme pour quoi ? Quelles nouvelles questions pour les mathématiques ? Quels apports pour l'apprentissage de la preuve ?* Thèse de doctorat, Université de Grenoble, décembre 2012.

[14] T. Ogita, S.M. Rump et S. Oishi : Accurate sum and dot product with applications. *In 2004 IEEE International Conference on Robotics and Automation (IEEE Cat. No.04CH37508)*, pages 152–155, Taipei, Taiwan, 2004. IEEE.

[15] Siegfried M. Rump : Ultimately Fast Accurate Summation. *SIAM Journal on Scientific Computing*, 31(5):3466–3502, janvier 2009.

[16] Jonathan Richard Shewchuk : Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Research Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, mai 1996.

[17] N.J.A. Sloane : A034886 : number of digits in $n!$, décembre 2010. The on-line encyclopedia of integer sequences. https://oeis.org/A034886.

[18] Steven D. Stellman : A Spherical Chicken. *Science*, 182(4119):1296–1296, décembre 1973.

[19] Maurice V. Wilkes : Computers Then and Now. *Journal of the ACM*, 15(1):1–7, janvier 1968.

[20] Éduscol : Algorithmique et programmation : ressources pour le lycée général et technologique — mathématiques. Rapport technique, Ministère de l'Éducation Nationale, 2017. https://cache.media.eduscol.education.fr/file/Mathematiques/73/3/Algorithmique_et_programmation_787733.pdf.